



Version 8.5.3 — 4 November 2024

Published by Just Great Software Co. Ltd.

Copyright © 1996–2024 Jan Goyvaerts. All rights reserved.

“EditPad” and “Just Great Software” are trademarks of Jan Goyvaerts

Table of Contents

EditPad Lite & Pro Manual.....1

1. File Menu	4
2. Edit Menu.....	20
3. Project Menu.....	32
4. Search Menu.....	45
5. Go Menu.....	70
6. Block Menu.....	76
7. Mark Menu	88
8. Fold Menu.....	91
9. Tools Menu.....	95
10. Macros Menu	107
11. Extra Menu.....	112
12. Convert Menu	127
13. Options Menu.....	149
14. Options Configure File Types.....	166
15. Options Preferences	214
16. View Menu.....	259
17. View Files Panel.....	271
18. View Explorer Panel.....	275
19. File Filter.....	278
20. View FTP Panel.....	279
21. View File History.....	287
22. View File Navigator	290
23. Help Menu.....	291
24. Customizing Toolbars and Menus.....	305
25. Command Line Parameters	309

Regular Expression Tutorial..... 313

1. Regular Expressions Tutorial.....	315
2. Literal Characters	316
3. Non-Printable Characters.....	318
4. First Look at How a Regex Engine Works Internally	319
5. Character Classes or Character Sets.....	321
6. Character Class Subtraction	323
7. Character Class Intersection	324
8. Shorthand Character Classes	325
9. The Dot Matches (Almost) Any Character	326
10. Start of String and End of String Anchors.....	328
11. Word Boundaries.....	331
12. Alternation with The Vertical Bar or Pipe Symbol.....	333
13. Optional Items.....	335
14. Repetition with Star and Plus	336
15. Use Parentheses for Grouping and Capturing.....	339

16. Using Backreferences To Match The Same Text Again	340
17. Backreferences to Failed Groups.....	343
18. Named Capturing Groups and Backreferences.....	345
19. Branch Reset Groups.....	347
20. Free-Spacing Regular Expressions.....	349
21. Unicode Regular Expressions.....	351
22. Specifying Modes Inside The Regular Expression.....	358
23. Atomic Grouping.....	359
24. Possessive Quantifiers	361
25. Lookahead and Lookbehind Zero-Length Assertions	363
26. Testing The Same Part of a String for More Than One Requirement	366
27. Keep The Text Matched So Far out of The Overall Regex Match.....	368
28. If-Then-Else Conditionals in Regular Expressions	370
29. Matching Nested Constructs with Balancing Groups	373
30. Regular Expression Recursion	377
31. Regular Expression Subroutines	378
32. Infinite Recursion.....	381
33. Quantifiers On Recursion	382
34. Subroutine Calls May or May Not Capture.....	384
35. Backreferences That Specify a Recursion Level.....	388
36. Recursion and Subroutine Calls May or May Not Be Atomic.....	391
37. POSIX Character Classes.....	394
38. Zero-Length Regex Matches	396
39. Continuing at The End of The Previous Match.....	398
40. Replacement Strings Tutorial	399
41. Special Characters.....	401
42. Non-Printable Characters	402
43. Matched Text	403
44. Numbered and Named Backreferences	404
45. Match Context	406
46. Replacement Text Case Conversion.....	407
47. Replacement String Conditionals.....	408

Regular Expression Examples..... 411

1. Sample Regular Expressions	413
2. Matching Numeric Ranges with a Regular Expression	415
3. Matching Floating Point Numbers with a Regular Expression	417
4. How to Find or Validate an Email Address.....	418
5. How to Find or Validate an IP Address	423
6. Matching a Valid Date	425
7. Replacing Numerical Dates with Textual Dates.....	426
8. Finding or Verifying Credit Card Numbers	428
9. Matching Whole Lines of Text.....	430
10. Deleting Duplicate Lines From a File.....	432
11. Example Regexes to Match Common Programming Language Constructs	433
12. Find Two Words Near Each Other	436
13. Runaway Regular Expressions: Catastrophic Backtracking.....	437
14. Runaway Regular Expressions: Too Many Repetitions	444
15. Preventing Regular Expression Denial of Service (ReDoS).....	447

16. Repeating a Capturing Group vs. Capturing a Repeated Group	451
17. Mixing Unicode and 8-bit Character Codes.....	453

Regular Expression Reference..... 455

1. Special and Non-Printable Characters	457
2. Basic Features.....	459
3. Character Classes	460
4. Shorthand Character Classes	463
5. Anchors	465
6. Word Boundaries.....	466
7. Quantifiers	467
8. Unicode Syntax Reference	470
9. Capturing Groups and Backreferences	473
10. Named Groups and Backreferences.....	475
11. Special Groups.....	477
12. Mode Modifiers	480
13. Balancing Groups, Recursion, and Subroutines.....	482
14. Replacement String Characters	486
15. Matched Text and Backreferences in Replacement Strings.....	488
16. Context and Case Conversion in Replacement Strings.....	490
17. Conditionals in Replacement Strings.....	492

Part 1

EditPad Lite & Pro Manual

EditPad 8 Help

Welcome to the documentation for EditPad Lite 8 and EditPad Pro 8. All features that are available in EditPad Lite are also available in EditPad Pro. Some features are available in EditPad Pro but not in EditPad Lite. Those features are indicated as (available in EditPad Pro only) in this help file.

If you are upgrading from EditPad 6 or 7 to EditPad 8, check out what's new in EditPad 8 and learn how to migrate to EditPad 8.

EditPad is very configurable. Almost every aspect can be adjusted to your own tastes and habits. Many settings can be made separately for each file type. To do so, select Options|Configure File Types in the menu. To edit general preferences, use Options|Preferences.

You can access all of EditPad's functionality through the main menu. Once you get used to working with EditPad, you'll mostly rely on the fully configurable toolbar and keyboard shortcuts.

EditPad's menus are:

- File
- Edit
- Project
- Search
- Go
- Block
- Mark
- Fold
- Tools
- Macros
- Extra
- Convert
- Options
- View
- Help

Some functionality can be accessed through panels that dock to the sides of EditPad's window:

- Search and Replace
- Spell Check
- Clip Collection
- Character Map
- Byte Value Editor
- Files Panel
- Explorer Panel
- FTP Panel
- File History
- File Navigator

1. File Menu

File | New

Creates a new tab with a blank, untitled file. If you click the File|New item directly, then the file type for the new file is the one for which you selected “default file type for new files” in the file type configuration. If you select a file type from the File|New submenu, then the new file uses the settings for the file type that you selected.

If you use File|Save As and give the file a name with an extension that belongs to a file type other than the one used by File|New, then the file’s file type is changed to the file type associated with that extension. The file will then also use the settings defined for that file type.

File | Open

The File|Open command shows an open file common dialog. It allows you to open one or more files from the folder of your choice. To select more than one file, hold down the Shift or Control key on the keyboard while you click with the mouse. You can configure the initial folder of the open dialog in the Open Files Preferences.

You can tick the read-only checkbox to force EditPad to open the file in read-only mode, regardless of whether the file is writable or not. This can be useful to make sure you don’t accidentally overwrite the file. It also tells EditPad not to try to open the file for writing.

If the active file is untitled and empty, it is replaced by the file you open. This ensures EditPad does not get cluttered with empty tabs.

If you open a single file that is already open, EditPad simply switches to the copy that is already open, as if you had clicked on its tab rather than attempting to open it again. If that file is open in another project, EditPad switches to that project and to the file. If you open multiple files that are all open in the same project, EditPad switches to that project and to one of the files you wanted to open.

If you open multiple files and some, but not all, of them are open in projects other than the active one, EditPad moves all the files into the active project. If some of the files were open in unmanaged projects, those files are removed from those projects. If some of the files were open in managed projects, those files are closed in those projects, but remain part of the other projects.

If the active project is a managed, the File|Open command does not add the files to the project. The files show up under the project’s tab as outside files. To make the files part of a managed project, either use Project|Add to Project instead of File|Open, or use Project|Add Outside Files after using File|Open.

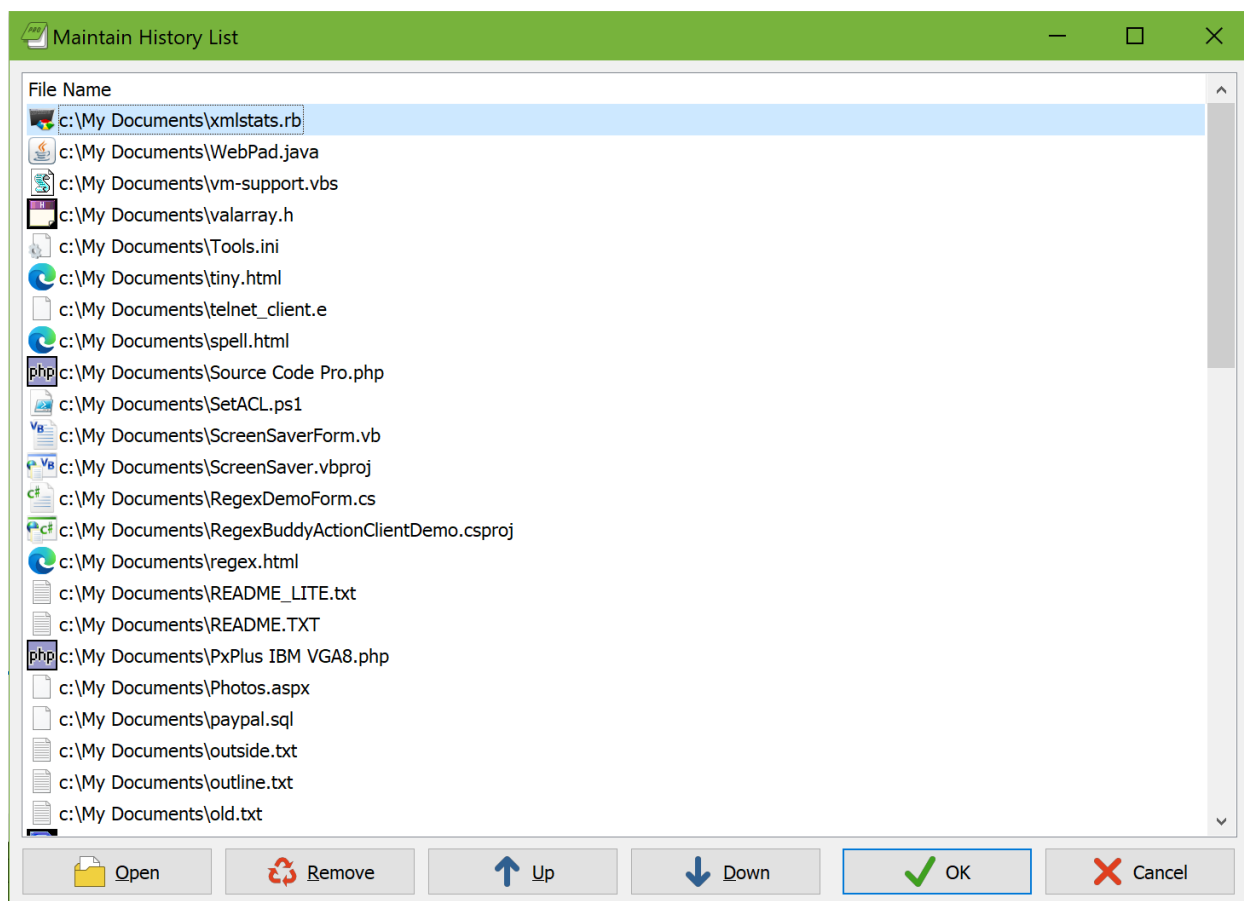
Recently Closed Files

The File|Open command has a submenu that lists recently closed files. Opening a file removes it from the File|Open submenu. Closing a file adds it to the top of the File|Open submenu. Selecting the file in that submenu opens it again.

Only files that were opened individually are added to the menu when they are closed. That includes files opened by double-clicking on them in Windows Explorer or by dragging and dropping them onto EditPad. Files that are opened in bulk such as by opening a project or opening a folder are not added to the File|Open menu when you close them. The Project|Open Project and Project|Open Folder commands have their own submenus for reopening projects and folders. Whether you close files one by one or all at the same time does not matter. How the file was opened determines whether it is added to the File|Open submenu.

At the bottom of the submenu, you will see the “Remove Obsolete Files” and “Remove All Files” items. The former removes all files that no longer exist from the list of recently closed files. The latter clears the list of recently closed files entirely.

Though the menu can display only 16 files, EditPad Pro actually remembers the last 100 files. To access the complete list, select the Maintain List item at the bottom. This item is only available in EditPad Pro.



When working with projects in EditPad Pro, the Project|Add to Project has its own submenu with recently closed files. It works just like the File|Open submenu, except that it remembers the last 100 files that were

closed in that project only. If you haven't worked with a project for a while, the Project|Add to Project submenu may still list closed files that have already dropped off the File|Open submenu.

Maintain List

Though the File|Open submenu can display only 16 files, EditPad Pro actually remembers the last 100 files you closed. To access the complete list, select the Maintain List item at the bottom. This item is only available in EditPad Pro.

To open many files again at once, select all of them and click the Open button. Clicking the Open button does not close the Maintain List dialog. You can select files and click the Open button repeatedly to open multiple sets of files.

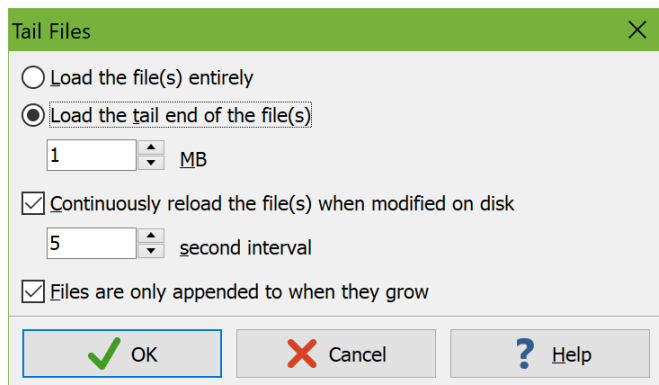
To remove files from the history, select them and click the Remove button. This will only remove files from the list. It will not delete any files. If you want to remove all files that no longer exist, you can do so by selecting "Remove Obsolete Files" in the File|Open submenu itself, rather than selecting "Maintain List" to open the above dialog.

Using the Up and Down buttons, you can change the order of the files in the history list. Press the Control key on the keyboard while clicking the Up or Down button to move a file all the way to the top or the bottom. If you want an item to be visible directly in the File|Open submenu, move it up until it is among the first 16 files in the list.

File|Tail

The File|Tail command is very convenient for dealing with files that may be very large, that may be updated continuously, or that have the information you want at the end of the file. Log files are a prime example. Opening these files with File|Tail instead of File|Open can give you better performance by loading only the end of the file and can keep you up-to-date by continuously reloading that tail end.

File|Tail first shows a file selection dialog just like the File|Open command. After you've selected the file(s) you want to open, it shows a second dialog box pictured below. The options you choose in that dialog are applied to all the files that you selected.



You can choose whether you want EditPad Pro to load the entire file or only a certain number of megabytes at the end of the file. If the file is smaller than the size you specify then the file is loaded entirely. If it is larger then EditPad loads the number of megabytes you specified. It trims the loaded block at the first line break so you don't end up with a partial first line.

If you tick “continuously reload the file(s) when modified on disk” then EditPad Pro checks whether the file has been modified on disk while you are viewing it at the interval that you specify. If it is then the file is reloaded immediately. This allows you to monitor live changes to the file. EditPad stops reloading the file if you switch to another file. It starts reloading the file again when you switch back to it. If the cursor is at the end of the file when it is reloaded, EditPad Pro keeps the cursor at the end of the file. This way you can watch text that is appended to the file scroll by.

If you don't have EditPad Pro continuously reload the file then it will still reload the file when you switch to it or when you switch to EditPad from another application if you ticked the option to automatically reload files in the Open Files Preferences.

Regardless of whether you opt for automatic reload, you can always select File|Reload from Disk in the menu to reload the file at any time.

If you know for certain that other applications will only append to the file while you have it open in EditPad Pro, tick that option. Then when EditPad Pro reloads the file it first checks if the file is larger than last time it was loaded. If it is, only the part of the file between the previous size and the new size is loaded. For large files this will be significantly faster than reloading the entire file. It can enable you to set a shorter reload interval. The part of the file that was previously loaded is kept in memory even if you selected to load only a certain number of megabytes at the end of the file. EditPad Pro does not trim the part that it has loaded to stay under that limit.

If you don't tick the option that files are only appended to when they grow, or if a file turns out to be smaller than last time it was loaded, then EditPad Pro discards what it has loaded before. If you selected to load the entire file then the entire file is reloaded. If you selected to load only the tail end, then EditPad Pro loads the same number of megabytes from the new tail end of the file. If you want to continuously reload at a short interval it's best to have EditPad Pro load a small number of megabytes.

If you only load part of the file, have it continuously reloaded, or tell EditPad Pro to assume the file is only appended to then EditPad Pro gives the file's tab a special “Tail” status. This basically makes the file read-only. If you want to edit the file, either close it and use File|Open to open the whole file normally. Or you can use File|Save As to save the file under a new name. If you loaded only the tail end of the file, then only that tail end is saved into the new file.

Recently Closed Files

The File|Tail command has its own submenu with recently closed files. Closing a file that you opened with File|Tail adds it to the top of the File|Tail submenu. Selecting a file from the submenu tails the file with the same settings that you used for it last time.

Opening a file with File|Open removes it from the File|Tail submenu. Opening a file with File|Tail removes it from the File|Open submenu. This way you don't accidentally use the wrong command to reopen a particular file. By not having duplicates between the menus they can together remember more files.

Read-Only Files

If a file is in use by an application, another application cannot modify it. Files stored on read-only media such as CD-ROMs cannot be modified by applications either. Files that have the read-only attribute set should not be modified by applications, but can be if you have the necessary security privileges to remove the read-only attribute.

If you use EditPad to open a file that is already in use by another application, stored on read-only media, or has the read-only attribute set, EditPad will open that file in “read only” mode. This is indicated on the status bar. Where you normally see whether the current file has been modified or not (the indicator shows “Modified” or shows nothing), you will then see “Read Only”. A file that is in read only mode cannot be edited in EditPad.

You can turn off “read only” mode by clicking on the “Read Only” indicator on the status bar. If the file is in use by another application or stored on a read-only device, however, you still won’t be able to save it even if you can edit it in EditPad. To save your changes, either close the other application that is using the file and then use File|Save, or save your changes to a new file with File|Save As. If the file was opened as read-only because the read-only attribute was set, EditPad will attempt to remove the read-only attribute if you try to save the file.

The File|Open screen has a “read only” checkbox at the bottom. If you tick that, EditPad opens the file as read only, even if the file is perfectly writable. In that case, clicking the “Read Only” indicator on the status bar makes the file editable, as if you had opened it without using the “read only” checkbox.

The Project|Open Folder and Search|Find on Disk dialog boxes also have a “read only” checkbox. If you tick that then all files opened via those dialogs are opened as read-only.

If you have a file open that is not read only and doesn’t have any unsaved changes, you can make it read-only by clicking the blank space in the status bar reserved for the “Modified” or “Read Only” indicator. This only prevents accidental changes to the file in EditPad. EditPad does not set the file’s read-only attribute, nor does it prevent other applications from modifying the file.

Files opened with File|Tail are always read-only. The status bar indicates them with “Tail”. You can’t toggle this indicator. If you want to edit the file, you’ll need to close it and then open it again with File|Open.

File|Favorites

In the Favorites submenu of the File menu, you can keep lists of files that you often work with. This way you can quickly open them to continue working on them.

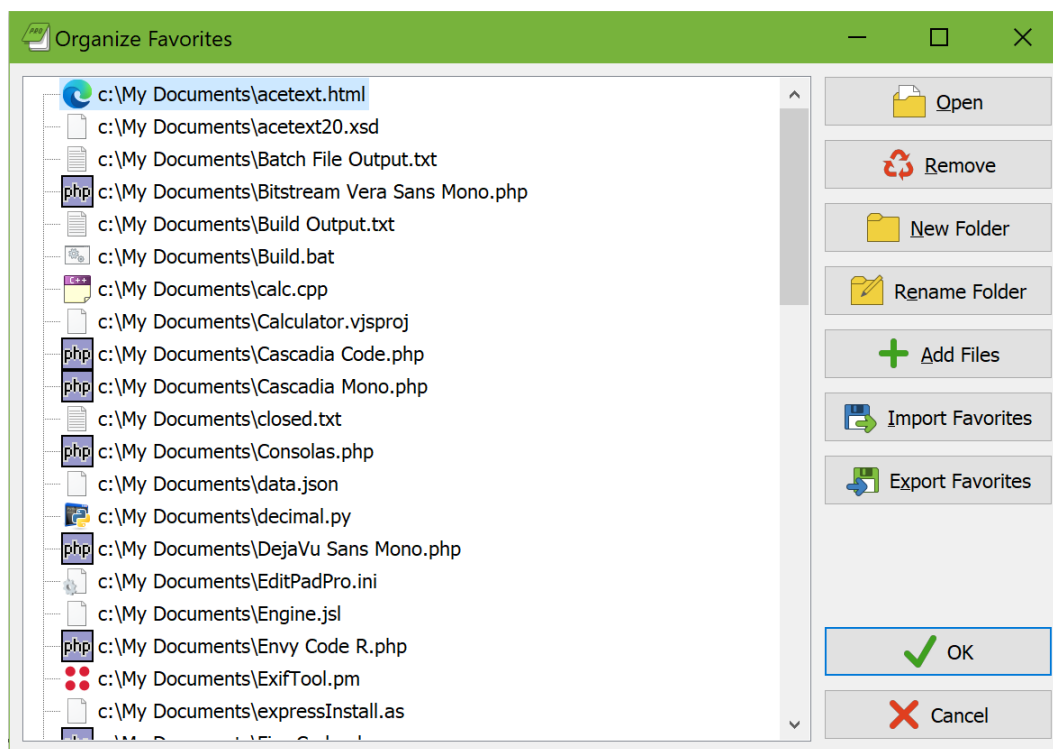
To add a file to your favorites list, first open the file in EditPad. Then select File|Favorites|Add Current File from the menu. The next time you open the File|Favorites menu, you will see the file listed there. The files are sorted alphabetically by their complete path names. If you have added folders via the Organize Favorites screen (see below), you can add a file to one of the folders by selecting the “Add Current File” item from the folder’s menu.

To purge files that no longer exist from the list of favorites, select “Remove Obsolete Files” from the Favorites menu. If you have created folders, non-existent files will be purged from those folders as well. To remove only certain files, whether they still exist or not, use the Organize Favorites screen.

To open one of your favorite files, simply choose it from the Favorites menu. To open many of your favorite files, use Organize Favorites.

Organize Favorites

Click the File|Favorites item directly or select Organize Favorites from the File|Favorites submenu to organize the list of your favorite files, or to open many of those files quickly.



Creating folders is useful if you have more than twenty favorite files. If you add more than 20 files directly to the favorites menu, or to one of the folders, then the menu listing your favorites will become very long and difficult to use. Note that creating a folder here to hold your favorites does not actually create a folder on your hard disk. It simply adds a hierarchical structure to the list of favorites stored in your EditPad Pro preferences. Use the New Folder button to create a new folder to which you can add favorites. You can change the caption of a folder by clicking on the folder and then clicking on the Rename Folder button.

When adding files to the favorites list, they are added to the main list if either nothing is selected or if a file in the main list is selected. To add files to a folder, select either the folder or a file in that folder. Click the Add Files button to find and add files.

You can move files between folders by dragging and dropping them with the mouse. EditPad Pro automatically sorts the list of favorites alphabetically. You cannot rearrange files inside a folder. If you drop a file onto another file then it becomes a sibling of that file. If you drop a file onto a folder then it is placed

inside that folder. If you drop a file on the folder that it is already inside of then it becomes a sibling of that folder.

To open one or more files, select them and click on the Open button. Note that when selecting more than one file, all the files must be in the same folder. You cannot select multiple files across folders. If you want to open files from several folders, open them folder by folder. You can click the Open button as many times as you want to open as many files as you want.

To remove one or more files from the favorites, select them and click on the Remove button. This will not delete the files, but only remove them from the list of favorites. If you want to remove all files that no longer exist, select Remove Obsolete Files, instead of Organize Favorites, from the File|Favorites menu.

File|Templates

The Templates submenu of the File menu works just like the Favorites submenu, with one small but important difference. When you select a file from the Favorites menu, EditPad simply opens that file. If you select a file from the Templates menu, however, EditPad will create a new, untitled file with the content of the file you selected.

If you often edit a particular file, saving the changes back into the same file, you should add it to the Favorites. If you often create new files based on the contents of a particular file, you should add that file to the Templates menu. Using templates, you won't accidentally save the changes over the original file.

The Favorites and Templates menus are completely independent. You can add the same file to both.

File|Save

Saves the active file to disk, overwriting the original, if any, without warning. File|Save is disabled if the active file does not have any unsaved changes.

In Save Files Preferences you can specify if and how backup copies should be created.

If the active file is untitled, File|Save invokes File|Save As.

File|Save As

Saves the active file under a new name. If the active file has previously been saved, the original copy will remain available. If you later use File|Save after using File|Save As, it will save under the new name as well. This is indicated by the file name on the tab which changes after you use File|Save As.

When you select File|Save As, a save file common dialog box appears. You can select the folder in which you want to save and type in a file name. You can configure the initial folder of the save dialog in the Open Files Preferences.

If you do not specify an extension for the file name, EditPad adds one depending on the file filter you selected in the drop-down list at the bottom of the Save As dialog box. It has all files types for which you

ticked “show in file type selection lists” on the Definition page in the file type configuration. There you can also specify the file masks for each file type. If the first file mask is in the form of *.ext then .ext is the default extension for that file type. If the selected filter does not have a default extension then EditPad does not add an extension and saves the file without an extension. To avoid unintentionally saving a file without or with a changed extension, EditPad automatically preselects the file filter of the file type that was detected when opening the file or that was selected by File menu or Options|File Type.

If you do specify an extension, EditPad saves the file with that extension. If the extension is part of another file type then all the file type dependent settings are changed to match the settings for that new file type as made in the file type configuration.

If you want to save a file with a dot in its name, such as “Document 1.0.txt”, then you have to specify the extension. If you omit the extension then EditPad sees the dot that you intended to be part of the file name as the start of the extension.

File|Save All

Rapidly performs a File|Save for all open files in all open projects.

For untitled files, File|Save As is invoked. If you have multiple untitled files then you will get a Save As dialog for each of them. If you cancel the Save As dialog then you cancel saving that file and any other files for which you were not prompted yet. This does not undo saving files for which you already clicked Save on the Save As dialog.

If you have a mixture of untitled files and previously saved files with unsaved changes then EditPad Pro first prompts to save each untitled file. When all untitled files are saved it automatically saves all previously saved files. So if you click Cancel on the Save As dialog for the first untitled file then no files are saved at all.

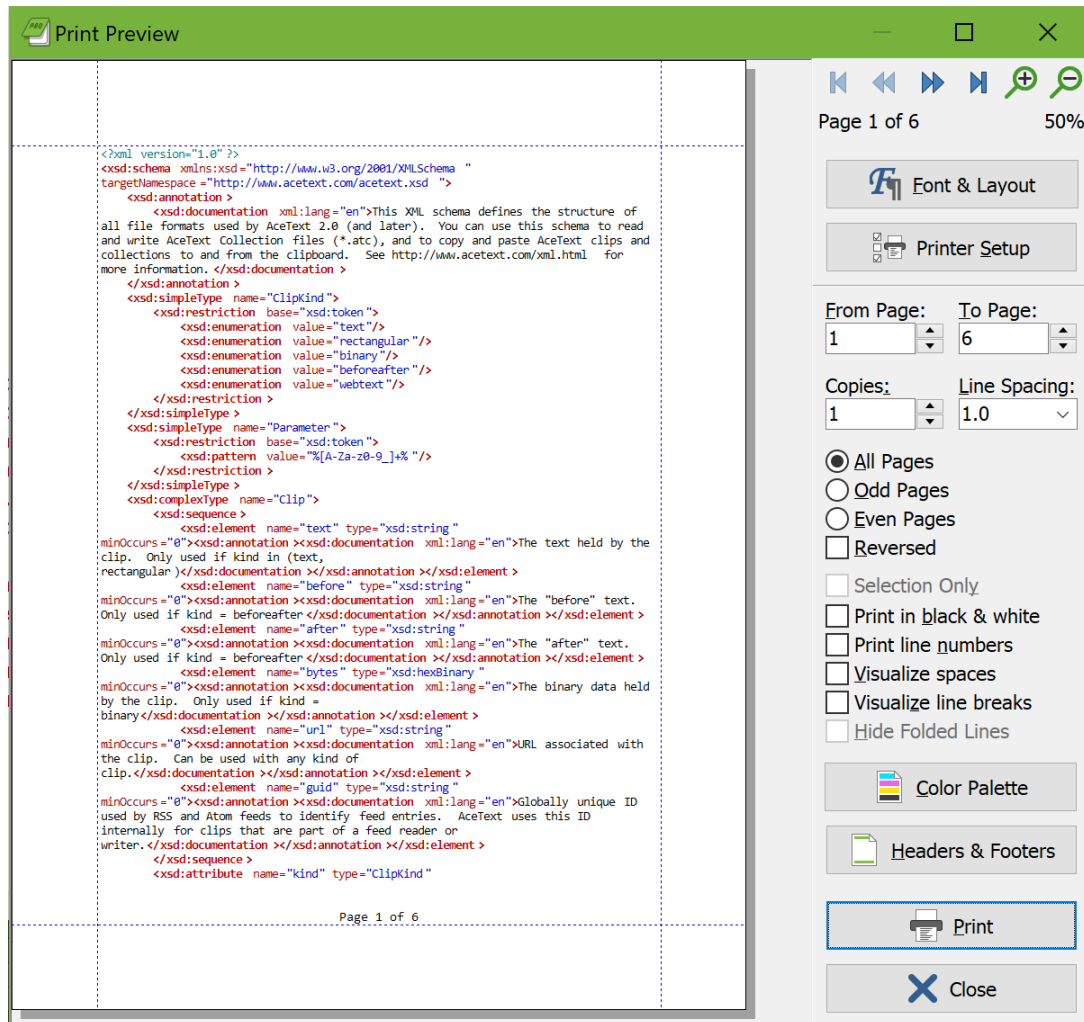
File|Print

Select File|Print in the menu to print the active file. A preview of the printout is shown first.

The margins can be changed by moving the mouse over one of the dotted blue margin lines. The mouse pointer will change to indicate that you can move the margin. A hint box will indicate the size of the margin in centimeters and inches. Click on the line, hold the mouse button down and drag the line until it indicates the desired margin position.

Long lines are wrapped at the right-hand margin if while editing the file you have word wrap turned off or set to wrap at the window border. If you have word wrap set to limit lines to a certain number of characters then the printout also wraps lines at that number of characters. If the space between the margins is insufficient for lines with that number of characters then lines that are too long are clipped at the right-hand margin. You can change word wrapping using the Options|Word Wrap menu item before selecting File|Print.

With the media player buttons at the top, you can browse through the pages that will be printed. You can also press the Page Up and Page Down keys on the keyboard.



The magnifying glass buttons zoom the preview in and out. You can also press the plus and minus keys on the numeric keypad on your keyboard. Zooming does not affect the printout. It only affects the preview. At low magnification, the preview may not be very accurate. Colored backgrounds may have small gaps and edges of characters may be clipped. That is because the printout is based on the printer resolution. Printer resolution is generally many times higher than the screen resolution. If you zoom out the screen doesn't have enough resolution to accurately show the preview. The actual printout will be pixel perfect regardless.

If you want to print with a different font or different text spacing, click the Font & Layout button. This button shows the same text layout configuration screen as the Options|Text Layout menu item. The difference is that it shows the text layouts you have configured for printing rather than the ones you've configured for editing. Another difference is that if you turn on the "allow bitmapped fonts" checkbox, the font list will include printer fonts rather than screen fonts in addition to the TrueType and OpenType fonts that work everywhere. A "printer font" is a font built into your printer's hardware. If you select a printer font then you should set "text layout and direction" to "left to right only" for best results.

Because EditPad keeps separate text layout configurations for printing and editing, any changes you make to text layouts via the print preview only affect the printout. EditPad remembers the layout you used for printing in combination with the layout you used for editing. If you edit a file with a text layout you've called "edit monospaced" and you print it with a text layout you've called "print monospaced" then the next time

you print a file you're editing with the "edit monospaced" layout, the print preview will automatically select the "print monospaced" layout.

Press the Setup button to access your printer's setup screen or to select a different printer than the default printer.

"From page" and "to page" determine the range of pages that will be printed. The value of "from page" must be smaller than or equal to that of "to page", even if you want to print in reversed order (see below). "Copies" is the number of copies that should be printed of each page.

Activating "All pages" will print all pages in the range specified by "from page" and "to page", while "odd pages" will print only the odd-numbered (1, 3, 5, ...) pages in the range, and "even pages" only the even-numbered (2, 4, 6, ...) ones. If the page range is only one page ("from page" equals "to page") and that page is odd-numbered while "even pages" is activated, or the other way around, nothing will happen when you click the Print button. You will not get an error message.

If you tick "Reversed" then the pages will be printed from last to first instead of from first to last. This can be useful if you have an inkjet printer that puts pages in reversed order in the out tray.

If you opened the print preview with File|Print then you can tick the "selection only" box to print only those lines that are selected in EditPad. The "selection only" checkbox works at the level of a single line. If a line is partially selected, it will be printed. If you only want to print exactly what you selected, select Block|Print in the menu instead.

If you have a monochrome printer or if you do not want to waste expensive color ink cartridges, tick "print in black and white" to print everything as black text without any background colors. Bold, italic, underline, and strikeout styles selected in the color palette will remain. Underline and strikeout colors will be blank.

When printing only the selection, syntax coloring may be applied differently when you use Block|Print to print the exact selection rather than using File|Print and turning on "selection only" to print the selected lines. With Block|Print, the syntax coloring will color the block as if it were the whole file, while File|Print applies syntax coloring to the whole file, even with "selection only" turned on.

Turn "print line numbers" on or off to print the file with or without line numbers, regardless of whether you've used Options|Line Numbers to edit the file with line numbers. The print preview remembers this checkbox separately for each file type.

In EditPad Pro, you can use the "visualize spaces" and "visualize line breaks" checkboxes to make spaces and line breaks visible in the printout or not, overriding the Options|Visualize Spaces and Options|Visualize Line Breaks menu items. These checkboxes are also remembered separately for each file type.

If you used the Fold|Fold command to hide certain lines then you can turn on the "hide folded lines" option to exclude those lines from the printout. The printout will not indicate folding ranges with "plus" or "minus" buttons.

You can click the Color Palette button to select a different set of colors. You should select a palette that uses a white background. If you print on colored paper then you should still select a white background in EditPad. EditPad cannot determine the color of your paper. White is assumed. If the print preview shows a colored or even a black background then the actual printout will also have the same colored or black background. This

can waste large amounts of ink. If you click the Print button and the background isn't white then EditPad will warn you about this (if you didn't previously tell EditPad not to warn about ink usage again).

The Color Palette button shows the same color selection dialog as the corresponding button in the file type configuration. If you edit any palettes, those changes also affect the palettes used for editing. If you want to use different palettes for editing and for printing, create new palettes for printing instead of editing existing ones. EditPad remembers the palette used for printing in combination with the palette used for editing. E.g. if you edit a file with the "Borland classic" palette (which uses a blue background) and then select the "Embarcadero" palette for printing it without the blue background then the print preview automatically defaults to the Embarcadero palette next time you print a file that you're editing with the Borland palette.

If EditPad has not remembered a printing palette for the palette you are presently using for editing (because you've never printed since editing with this palette) then the print preview defaults to the companion printing palette of the palette you're editing with. All predefined color palettes have companion printing palettes that result in a similar style of syntax coloring but with a white background and black as the plain text color. If you create custom palettes that you want to share with others then you should also create companion printing palettes.

Click the "Print" button to start printing. If you've printed all pages then this also closes the print preview.

Click the "Close" button or press Escape on the keyboard to close the print preview. If you close without printing then EditPad does not remember your text layout or color palette choices.

Print Headers

Click the "Headers & Footers" button in the Print Preview window to change the headers and footers to be printed, if any.

The headers and footers are printed inside the margins specified on the print preview. If no headers or footers are specified, their space will be claimed by the text body.

A few special placeholders are available in the headers and footers:

%P	Page number of the current page.
%N	Total number of pages.
%D	Current date, printed using the short date format specified in the regional settings in Windows.
%T	Current time, printed using the short time format specified in the regional settings in Windows.
%FD	Date the file was last modified, again printed in short date format.
%FT	Time of the day the file was last modified, again printed in short time format.
%FP	Full path to the file being printed: folder + file name.
%FN	File name (without folder name) of the file being printed.

File | Mail

Select File | Mail from the menu to send the current file to somebody via email. The email composition panel is shown.

Type in your full name followed by your own email address in the field labeled “From”. If you have previously used the email function, you can quickly select your email address from the drop-down list.

Then type in the recipient’s email address in the “To” field. You can either enter the recipient’s full name followed by his or her email address, or you can enter only the email address.

If you want to send your message to more than one recipient, click the Add button next to the “To” field to add the first email address to the list. Then type in the second email address into the “To” field, and click the Add button again. Do so for all recipients. If you make a mistake, you can remove a recipient by selecting his or her email address from the list and clicking the Remove button.

In the “Subject” field, type in the subject of the email message.

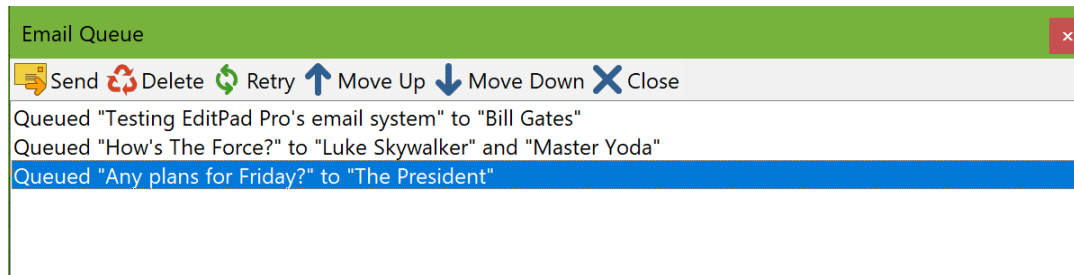
If you want to attach files to the email message, click the Attach button. Select the file you want to send. You can attach more than one file if you want to. If you change your mind about attaching a file, click on it in the list and click the Remove button below the list of attachments.

The Send or Enqueue button does not become enabled until you type in your email address, the recipient’s email address, and the message’s subject. When you click the Send button, the email is placed in the mail queue and sent out immediately. When you click the Enqueue button, the email is added to the mail queue. But it won’t be sent until you click the Send button in the mail queue. Which of the two buttons appears depends on the option to send out email immediately in the Email Preferences.

Before you can actually send mail, you need to specify the outgoing mail server that EditPad Pro should use in the Email Preferences. If you haven’t done this yet, clicking the Send button brings up the Preferences screen.

Mail Queue

The mail queue panel appears after you click the Enqueue or Send button in the mail composition panel. Each time you click the Enqueue or Send button, another email message is added to the bottom of the mail queue. The messages are marked as “queued”.



Click the Send button in the mail queue to start sending the messages in the queue. EditPad Pro begins with the first message in the queue marked as “queued”. EditPad Pro always starts sending the first message, no matter which message you selected in the queue before clicking the Send button. When the first message has been sent, EditPad Pro continues with the next, until all queued messages have been sent. The message that is being sent is marked as “sending”. After it has been sent, it will either be marked as “sent” in case of success, or “failed” in case the message could not be sent. While email is being sent, EditPad Pro tells you what is happening in the caption bar above the queue. The caption indicates “Email Queue” if you have not sent anything yet.

To remove a message from the queue, select it and click the Delete button. You can delete any message, even while EditPad Pro is sending email. Only the message that is currently being sent (if any) cannot be deleted from the queue. If you want to delete all successfully sent messages from the queue, click the Close button.

If EditPad Pro could not send a particular message, it will be marked as “failed”. If you want to try the message again without making any changes to it, select it and click the Retry button. This marks the message as “queued”. You can do this whether EditPad Pro is sending messages or not. Alternatively, you can cancel sending the message by deleting it from the queue. EditPad Pro does not automatically retry failed messages. You have to mark them again as “queued”.

To move a message up or down in the queue, select it and click the Move Up or Move Down button. You can move any message, regardless of its status. However, the order of the messages only matters for those marked as “queued”. The first one in the list is the one that will be sent next, regardless of which message is currently being sent (if any).

Click the Close button to close the mail queue panel. This also deletes all successfully sent messages from the panel. The only way to make the mail queue appear again after closing it, is to add another message to the queue through the mail composition panel. Therefore, you cannot close the mail queue while email is being sent. If the panel takes up too much space, click on the minimize button on the caption bar below the queue. This hides the list of messages and the buttons, but not the caption bar. This allows you to continue monitoring the progress in the caption bar, and to see the queue again by clicking the minimize button turned restore button again.

Note that the email queue sends email in the background. This means that you can continue using EditPad Pro as usual, including composing new emails, while email is being sent. This is particularly useful when sending large attachments over a slow connection. The only function you cannot use is File | Exit.

File | Mail as Attachment

Select “Mail as Attachment” from the File menu to start with a blank email message and attach the current file to the email. The mail composition panel will appear to allow you to further prepare the email message.

File | Reload from Disk

Select File | Reload from Disk from the menu to revert the active file to the status it had when it was last saved. This can be useful if you are viewing a file that is being written to by another program.

If you switch tabs within EditPad or if you switch to another application and then back to EditPad, then EditPad automatically checks whether the file on disk has been changed. If so, and you have not modified the file in EditPad, it is automatically reloaded. If you did modify the file in EditPad, you are asked if you want to keep the changes made in EditPad or reload from disk. In the Open Files Preferences, you can disable the automatic reload, or make EditPad always prompt before reloading.

If you click Keep Changes, EditPad does not reload the file. It will not overwrite the file on disk unless you use File | Save. If you want to keep both the changes you've made in EditPad and the modified copy of the file on disk, click the Keep Changes button, and then use File | Save As to save the changes in EditPad into a new file.

EditPad Pro gives you an additional option to see the difference between the file in EditPad and the file on disk. If you click the See Difference button, EditPad Pro invokes Extra | Compare with File on Disk. EditPad Pro does *not* reload the file. It does not ask you again to reload the file unless and until it is changed on disk again. If you want to reload the file from disk after looking at the differences, use the File | Reload from Disk menu item.

If you used File | Tail to continuously reload a file at a certain interval, you can still use File | Reload from Disk to reload the file before the interval has elapsed. If you used File | Tail to load only the tail end of the file, then File | Reload from Disk appends or reloads the tail end of the file in the same way that continuous reloading does.

File | Save Copy As

The Save Copy As command in the File menu works just like File | Save As. The only difference is that after File | Save Copy As, the File | Save command continues to save the file using the original file name. The file name indicated by the tab does not change.

The Save Copy As command adds an extra “open the saved file” checkbox at the bottom of the file selection screen. If you turn on this option, EditPad opens the copy of the file that you just saved. You'll end up with two tabs. One for the original file and one for the copy. You can edit them independently.

See File | Save As for more information on how EditPad saves files. See View | File History for a quicker way to save milestone copies of your files.

File | Rename / Move

Use this function to move the active file into a different folder, and/or to change its file name.

This feature is similar to File | Save As, except that the original file is deleted after the file has been saved under the new name. This effectively renames or moves the file.

File | Export to HTML or RTF

The Export to HTML or RTF command in the File menu saves a copy of the file just like File | Save Copy As. But the file filter on the export dialog only lets you choose between HTML and RTF. It converts the file from plain text to HTML or RTF. It preserves the appearance of the file in EditPad as much as possible. You will see the text with EditPad's syntax coloring when you open the exported file in a web browser or a word processor.

This can be very useful for adding code examples to presentations or documents. If you only need to show part of your file, you can use Block | Export to HTML or RTF, Edit | Copy as HTML, or Edit | Copy as RTF.

File | Delete

After you confirm the warning message, File | Delete closes the active file and deletes it from disk.

If Windows keeps a Recycle Bin for the drive the active file is on then the file is deleted by moving it into the Recycle Bin.

File | Close

Closes the active file. If the file has unsaved modifications then by default EditPad asks you whether you want to save or discard the changes. On the Open Files page in the Preferences you can choose if EditPad should automatically save or discard unsaved changes without prompting.

EditPad automatically starts with a new blank file when you close the last file in the project.

You can also close the active file by clicking with the mouse wheel on the file's tab. If you have enabled the X button to be shown on tabs or the tab control in the Tabs Preferences then you can use the X button to close the file. Closing a file by closing its tab is exactly the same as using the File | Close menu item.

If the file is part of a managed project then closing open files does not remove the file from the project. Files that are part of managed projects remain part of the project as closed files. The Files Panel and Search Panel can show and search through files that are closed but are still part of managed projects. Closing outside files removes them from the project, because they were never an actual part of the project.

If you want to remove an open file from a managed project, use Project | Remove From Project to close and remove the file. If you want to remove files that you have already closed from a managed project, use Project | Remove Closed Files.

To close EditPad itself, click on the X button in the upper right of the Window, or select File | Exit in the menu.

File | Close All

Select Close All in the File menu to close all open project and all open files. If a file has unsaved modifications then by default EditPad asks you whether you want to save or discard the changes. If multiple files have unsaved changes then you get one prompt for each file. The prompt then has buttons to save all files or to discard changes to all files. If you click Cancel Close at any point then no files are closed. Choosing to save a file saves it immediately. On the Open Files page in the Preferences you can choose if EditPad should automatically save or discard unsaved changes without prompting.

After closing all files, EditPad starts with a new untitled project with one blank untitled file.

To close all files in the current project only, use Project|Close All Files. Or close the project itself with Project|Close Project. To close EditPad itself, click on the X button in the upper right of the Window or select File|Exit in the menu.

File | Close All but Current

Closes all files in the current project except the one you're currently viewing. If a file has unsaved modifications then by default EditPad asks you whether you want to save or discard the changes. If multiple files have unsaved changes then you get one prompt for each file. The prompt then has buttons to save all files or to discard changes to all files. If you click Cancel Close at any point then no files are closed. Choosing to save a file saves it immediately. On the Open Files page in the Preferences you can choose if EditPad should automatically save or discard unsaved changes without prompting.

If you want to close all files in all projects, except the current file, first use Project|Close All but Current to close all other projects. Then use File|Close All but Current to close all other files in the current project.

File | Exit

Closes all files and terminates EditPad completely.

If you have enabled EditPad's icon next to the system clock in the System Preferences then File|Exit is not the same as clicking the X button in the upper right corner. Clicking the X button keeps EditPad in memory and keeps the icon visible, enabling EditPad to pop up instantly next time you want to edit a file. File|Exit removes the icon and shuts down EditPad completely.

If the icon next to the system clock is disabled, clicking the X button is the same as using File|Exit.

2. Edit Menu

Edit | Undo

Use Edit | Undo to undo the last editing action. Repeat to undo more actions.

You can undo the undo with Edit | Redo if you do so right away. If you take any other action that is remembered by the undo function then the redo list is cleared. Only trivial actions such as moving the cursor do not clear the redo list.

You can undo several actions in one go via the submenu of the Undo command. When you select an action, that action and all actions listed above it in the submenu are undone at once. They are all added to the redo list.

If you use the menu item or toolbar button to invoke this command then it is invoked on EditPad's main editor, where you edit files. If you press the shortcut key on the keyboard, it is invoked on whichever editor is showing the text cursor (vertical blinking bar), whether that's the main editor, the search box, or the replace box.

The undo feature remembers all changes you made to any file since you opened it in EditPad. Saving a file does not clear the undo list. Switching between files switches between the undo lists of those files. EditPad remembers the changes you made to each file even while you work with other files, until you close each file.

Undoing and redoing actions updates the file's "modified" status as indicated by the status bar and tab color. When you undo all changes you made since last saving the file, or redo all changes you undid since last saving the file, the file is indicated as being unmodified.

In the Editor Preferences you can enable indicators for lines that were edited since you last opened or saved the file. If an edit adds an indicator that a line was edited then undoing the edit removes the indicator. If you save the file then the indicators change. Undoing an edit that you made before you saved the file then indicates the lines affected by the undo as edited since the last save.

EditPad may use quite a lot of memory to remember all the changes you made to all the files you have open. Some commands, such as a search-and-replace across all files, may result in very large numbers of changes. EditPad's undo history includes a safeguard to make sure EditPad does not run out of memory. If the undo history grows too large, EditPad automatically discards the oldest changes. In extreme cases, like a search-and-replace that makes millions of replacements, the undo history may be cleared entirely.

In EditPad Lite, the memory limits are set automatically to make sure EditPad Lite doesn't use up all of your computer's memory, while still being able to keep a complete undo history in most situations. Unless you're making millions of changes in a search-and-replace, EditPad Lite's undo history can easily keep track of a full day's worth of text editing.

In EditPad Pro, you can configure the limits in the System Preferences. The default limits use the same balance as EditPad Lite. The actual numbers depend on the amount of RAM your PC has.

In practical terms, you needn't worry about EditPad's undo history. It'll remember virtually everything, without causing your computer to run out of memory and crash. If you do need to undo very long editing sessions, EditPad's backup options and File History may be more practical anyway.

Edit | Redo

If you change your mind after undoing an action with Edit | Undo, use Edit | Redo to redo it.

The Redo function works just like the Undo function. The only difference is that while the Undo function tries to remember all edits you made since you opened the file, the Redo function only remembers the edits you undid until you make another edit that is added to the Undo menu.

In contrast with many other text editors, however, EditPad Lite and Pro do not clear the Redo list when you take a trivial action such as moving the text cursor or making a selection. This gives you more opportunity to change your mind and redo undone actions. The Redo command then undoes the trivial action before redoing what you wanted to redo.

If you use the menu item or toolbar button to invoke this command then it is invoked on EditPad's main editor, where you edit files. If you press the shortcut key on the keyboard, it is invoked on whichever editor is showing the text cursor (vertical blinking bar), whether that's the main editor, the search box, or the replace box.

Edit | Cut

Deletes the current selection and places it on the Windows clipboard. Use Edit | Copy to put the selection on the clipboard without deleting it. If there is no current selection, and the option to copy the active line is turned on in the Editor Preferences, then the entire current paragraph is deleted and placed onto the clipboard.

This command is always invoked on whichever editor is showing the text cursor (vertical blinking bar), regardless of whether you use the main menu, a toolbar button, or a keyboard shortcut.

Any data previously held by the clipboard is removed from it. Use Edit | Cut Append instead if you do not want this.

If you select a block that includes one or more folded sections and cut the block to the clipboard then all selected lines, including lines hidden by folding, are cut to the clipboard. When pasting back into EditPad Pro, the folded sections remain folded.

Edit | Cut uses the same Windows clipboard formats as Edit | Copy. These are described in the Edit | Copy topic.

Edit | Cut Append

Deletes the current selection and places it on the Windows clipboard. Use Edit | Copy Append to do this without deleting the selection from EditPad. If there is no current selection, and the option to copy the active

line is turned on in the Editor Preferences, then the entire current paragraph is deleted and placed onto the clipboard.

If the clipboard already holds plain text data then the selection is appended to the text already on the clipboard, even if the text was placed on the clipboard by another application. Edit|Paste then pastes both the original text plus the selection appended to it.

This command is always invoked on whichever editor is showing the text cursor (vertical blinking bar), regardless of whether you use the main menu, a toolbar button, or a keyboard shortcut.

If the file uses syntax coloring and the clipboard already holds HTML or RTF copied by the same EditPad Pro instance then that HTML or RTF is appended to as well. If the clipboard holds HTML or RTF copied by another application then Edit|Cut Append removes that from the clipboard.

Edit|Copy

Places the current selection on the Windows clipboard. If there is no current selection, and the option to copy the active line is turned on in the Editor Preferences, then the entire current paragraph is placed onto the clipboard.

This command is always invoked on whichever editor is showing the text cursor (vertical blinking bar), regardless of whether you use the main menu, a toolbar button, or a keyboard shortcut.

Any data previously held by the clipboard is removed from it. Use Edit|Copy Append instead if you do not want this.

If you select a block that includes one or more folded sections and copy the block to the clipboard, all selected lines, including lines hidden by folding, will be copied to the clipboard. Use Fold|Copy Visible Lines if you don't want to copy the hidden lines. When pasting back into EditPad Pro, the folded sections remain folded.

Clipboard Formats

EditPad makes the copied text available on the clipboard in two formats. The first is unformatted Unicode text. This is the most basic Windows clipboard format that is understood by all Windows software. If you copy multiple lines of text to the clipboard then EditPad copies the text with Windows line breaks. This ensures maximum compatibility with other Windows applications. Many Windows applications do not understand any other line break styles.

The other format is EditPad's own data format. This format supports all the same encodings, line break styles, and even binary data that EditPad supports. This data is used when you paste your text back into EditPad or another application from Just Great Software. If you paste into an editor using the same encoding then all bytes are preserved exactly.

When you copy and paste within a single EditPad instance, the data is not actually transferred to the clipboard. EditPad handles that internally. This allows you to copy and paste blocks of any size. Other applications may not be able to handle huge amounts of clipboard data. Their developers may not have taken

precautions to make their applications refuse to paste more text than they can handle with good performance. So to prevent choking other applications, EditPad will not put more than 50 million characters of unformatted Unicode text on the clipboard. It will also not put more than 2,047 MB (just under 2 GB) of any data format on the clipboard.

The Windows clipboard does not have a standard format for binary data. If you're editing a file in hexadecimal mode then EditPad Pro needs to convert the bytes you're copying into text. The position of the cursor determines how EditPad Pro does this. If you copy from the hexadecimal section then EditPad Pro places the hexadecimal representation on the clipboard, such as "74657374". If you copy from the ASCII section, then EditPad Pro copies the textual representation, such as "test". If you want to paste into another hex editor, you should copy the hexadecimal representation. Hex editors know how to interpret this. If you copy from the ASCII section you may get different bytes in the other hex editor.

If you copy and paste between files in hexadecimal mode in EditPad Pro then it doesn't matter where you copy from. EditPad Pro can use its own data format to transfer binary data. If you copy from a file in hexadecimal mode and paste into a file in text mode, then EditPad Pro pastes the hexadecimal representation if you copy from the hex section and the textual representation if you copy from the ASCII section.

EditPad Pro supports two additional clipboard formats: HTML and RTF. Edit|Copy makes either or both of these formats available if you have turned on Edit|Copy as HTML or Edit|Copy as RTF and the file you're copying from uses syntax coloring.

Edit|Copy Append

Places the current selection on the Windows clipboard. If there is no current selection, and the option to copy the active line is turned on in the Editor Preferences, then the entire current paragraph is placed onto the clipboard.

If the clipboard holds text data then the selection is appended to the text already on the clipboard. Edit|Paste then pastes both the original text plus the selection appended to it.

This command is always invoked on whichever editor is showing the text cursor (vertical blinking bar), regardless of whether you use the main menu, a toolbar button, or a keyboard shortcut.

If the file uses syntax coloring and the clipboard already holds HTML or RTF copied by the same EditPad Pro instance then that HTML or RTF is appended to as well. If the clipboard holds HTML or RTF copied by another application then Edit|Copy Append removes that from the clipboard.

Edit|Copy as HTML

The Edit|Copy as HTML menu item is a toggle switch. Selecting the menu item toggles the switch and also immediately invokes Edit|Copy so that the contents of the clipboard immediately reflect the state of the switch. Turning on Edit|Copy as HTML does not turn off Edit|Copy as RTF. Both can be active at the same time.

When Edit|Copy as HTML is checked, Edit|Cut and Edit|Copy make the cut or copied text available in HTML format on the clipboard in addition to EditPad Pro's usual clipboard formats. Edit|Cut Append and Edit|Copy Append can append to the HTML already on the clipboard if it was put there by EditPad Pro.

Considerations When Copying Formatted Text

The HTML or RTF that EditPad Pro places on the clipboard is the same text that you selected in EditPad Pro with added HTML or RTF tags that apply the same font and colors to the text as you see in EditPad Pro. When you paste into a web editor or a word processor that understands the HTML or RTF clipboard format, the pasted text will have (almost) the same appearance in your web editor or word processor as it had in EditPad Pro. That includes any highlighting such as search matches or misspelled words. You may want to turn off Extra|Live Spelling before pasting into an important document. Double-click actions like opening URLs are not added to the HTML or RTF.

The copied text uses the same colors as you're using in EditPad, including any background colors. If you want to paste into a document that will be printed on white paper, you may want to switch EditPad to one of its color palettes designed for printing. If you want to paste into a website that uses a dark theme, you may want to switch EditPad to a dark palette first.

EditPad always makes the text available on the clipboard as unformatted text. Applications such as Notepad (and even EditPad itself) that can only paste unformatted text will paste that unformatted text even if EditPad also made HTML or RTF available on the clipboard. You can use Block|Export to HTML or RTF and then open the exported file if you want to use EditPad to look at the HTML or RTF that EditPad generates.

HTML and/or RTF are offered as an additional format. The HTML or RTF is only generated when another application requests it. When copying and pasting within EditPad or between EditPad and another plain text editor, no time or memory is wasted to generate the HTML or RTF. If you like syntax coloring to be preserved when pasting into word processors then you can leave this option turned on permanently. You only need to turn it off if you want to paste unformatted text into an application that would prefer to paste the HTML or RTF.

Some applications can paste only HTML or RTF. Microsoft Excel, for example, can paste HTML but not RTF. So you may need to tick both Edit|Copy as HTML and Edit|Copy as RTF to be able to paste formatted text in all your applications.

Some applications can paste both HTML and RTF, but support one format better than the other. In that case you may want to tick only one of the two formats to see which gives better results when pasting. Microsoft Word, for example, supports both formats. But it interprets some background colors differently depending on the format.

Edit|Copy as RTF

The Edit|Copy as RTF menu item is a toggle switch. Selecting the menu item toggles the switch and also immediately invokes Edit|Copy so that the contents of the clipboard immediately reflect the state of the switch. Turning on Edit|Copy as RTF does not turn off Edit|Copy as HTML. Both can be active at the same time.

When Edit|Copy as RTF is checked, Edit|Cut and Edit|Copy make the cut or copied text available in rich text format on the clipboard in addition to EditPad Pro's usual clipboard formats. Edit|Cut Append and Edit|Copy Append can append to the rich text already on the clipboard if it was put there by EditPad Pro.

The considerations when copying formatted text explained for Edit|Copy as HTML also apply to Edit|Copy as RTF.

Edit|Paste

If you select Edit|Paste when the Windows clipboard holds textual data then that text is inserted into the active file at the current position of the text cursor. If EditPad is in overwrite mode, and you're not pasting whole lines (see below), the pasted text overwrites the text after the cursor, as if you had typed in the text. Pressing the Insert key on the keyboard or clicking the Insert/Overwrite indicator on the status bar toggles between insert and overwrite modes.

This command is always invoked on whichever editor is showing the text cursor (vertical blinking bar), regardless of whether you use the main menu, a toolbar button, or a keyboard shortcut.

If the text was copied by EditPad or another Just Great Software application then it will be on the clipboard in EditPad's own data format. This format supports all the same encodings, line break styles, and even binary data that EditPad supports. If the file you are pasting into uses the same encoding as the file you copied from then all characters and even any invalid bytes the text may contain are pasted unchanged. If the file you are pasting into uses a different encoding, then EditPad converts the pasted text to the file's encoding.

If the text was copied by another application then EditPad pastes unformatted Unicode text. This is the most basic Windows clipboard format that all applications are supposed to support. If the file you are pasting into is not a Unicode file, then EditPad converts the pasted text to the file's encoding.

If you paste in characters that are not supported by the file's encoding, then EditPad has no way to convert those characters into bytes to store them into the file. Such characters will be lost. They are permanently changed into **question marks** to indicate the actual characters you tried to paste could not be represented. In order to paste the actual characters, first use Edit|Undo to remove the question marks. Then use Convert|Text Encoding with the "encode original data with another character set" option. Select a Unicode transformation or any encoding that supports the characters you want to paste, as well as those already present in the file. EditPad then changes the bytes in the file to represent the same characters in the new encoding. Now you can paste your text again and get the actual characters. Note that you have to undo pasting the question marks. Changing the encoding does not magically restore the characters. Since EditPad Pro uses the file's actual encoding for in-memory storage rather than Unicode, newly entered or pasted characters that cannot be represented cannot be stored.

EditPad's own clipboard data format can store line breaks of any style. Unformatted Unicode text is supposed to use only Windows-style line breaks. Regardless of which is being pasted, EditPad makes sure that you don't accidentally mix up the line break style of your file. Any line breaks in the pasted text that use a line break style that is not used at all by the file you're editing are converted into the file's dominant line break style. So if you copy from a Windows text file and paste into a UNIX text file, for example, the Windows-style line breaks that you copied are pasted as UNIX-style line breaks.

If you want to paste a block of text from another application as a rectangular selection, first make a rectangular selection in EditPad, and then paste. EditPad will then interpret the text on the clipboard as a

rectangular block and replace the selection with it. Text copied from EditPad Pro is always pasted in the way (linear or rectangular) you had it selected when you cut or copied it.

In EditPad Pro, the option “paste whole lines when lines are copied as a whole” affects how text is pasted if you copied complete lines to the clipboard. Copying a complete line means to copy everything from the start of the line to the end of the line, including the line break at the end of the line. Copying multiple lines completely means copying everything from the start of the first line in the block until the end of the last line in the block, including the line break at the end of the last line. When the option “paste whole lines when lines are copied as a whole” is on, lines that were copied as a whole are always pasted as if the cursor were at the start of the line when you’re pasting. Thus, lines copied as a whole are always pasted as a whole before the line that the cursor is on when pasting. This makes it easy to move blocks of lines around without worrying about the horizontal position of the text cursor. If this option is off, text is always pasted at the exact spot the text cursor is at, even when whole lines were copied. Only whole lines copied in EditPad can be pasted as whole lines. EditPad cannot determine whether text copied from other applications is a whole line or not. EditPad Lite does not have this option and always pastes at the exact spot the text cursor is at.

In hexadecimal mode, if the clipboard contains a hexadecimal representation of characters, the effect of the Paste command depends on whether the text cursor is in the hexadecimal section at the left, or the ASCII section at the right. If you paste into the hexadecimal section, EditPad pastes the bytes represented by the text on the clipboard. If you paste into the ASCII section, EditPad will paste the text on the clipboard “as is”. If the clipboard holds “74657374”, for example, then pasting into the hex section inserts “test”, while pasting into the ASCII section inserts “74657374” into the file.

If the text on the clipboard is not text with hexadecimal values, EditPad pastes the text regardless of whether the cursor is in the text or hexadecimal section. If the clipboard holds something you copied in hexadecimal mode in EditPad, then EditPad always pastes the bytes that you copied, even if you switched the cursor between the ASCII and hex sections. If you copied text from a file in EditPad that you’re editing in text mode, then the hex interpretation described in the previous paragraph does take place.

If you copy and paste between EditPad and another hex editing application, you may need to switch EditPad to hexadecimal mode and place the cursor in the hexadecimal section of the editor to get the results you expect. The Windows clipboard does not have a standard format for binary data. So most hex editors copy the hexadecimal representation of the selected bytes.

Edit | Swap with Clipboard

If you select Edit | Swap with Clipboard when the Windows clipboard holds textual data, then it is swapped with the current selection in the active file. That is, the selection is put on the clipboard as happens when you use Edit | Cut. The text previously held by the clipboard is then pasted into the text like Edit | Paste does.

This command is always invoked on whichever editor is showing the text cursor (vertical blinking bar), regardless of whether you use the main menu, a toolbar button, or a keyboard shortcut.

Edit | Select All

Selects all the text in the active editor. This command is always invoked on whichever editor is showing the text cursor (vertical blinking bar), regardless of whether you use the main menu, a toolbar button, or a keyboard shortcut.

Edit|Insert Matching Bracket

The Insert Matching Bracket command is only available when bracket matching is enabled on the Brackets page in the file type configuration for the active file's file type. The syntax coloring scheme determines exactly which brackets are matched. See that section in this help file to learn how bracket matching works in EditPad Pro.

Just like the syntax coloring scheme determines which brackets are matched, it also determines what is inserted by the Insert Matching Bracket command. When an unclosed opening bracket is highlighted, the corresponding closing bracket is inserted. When an unmatched closing bracket is highlighted, the corresponding opening bracket is inserted. The HTML and XML schemes insert matching opening and closing tags.

The Insert Matching Bracket command is intended to be used while typing with minimal cursor movement. If you wanted to enter "this bold word" into an HTML file, you would first enter "this bold", then use Edit|Insert Matching Bracket, and then enter " word". When you use Edit|Insert Matching Bracket, the cursor is automatically placed after the inserted tag.

Do not use Insert Matching Bracket immediately after typing "this ". Though it may seem neat to close the bold tag immediately, before typing its contents, this requires unnecessary cursor movement. You'd end up with the cursor after "this " requiring you to move the cursor back over the "" tag, then type "bold", then move the cursor forward over "" and then type " word". Since EditPad Pro automatically highlights the "" tag as long as it is unclosed, there is no need to close it immediately. Simply leave it unclosed until you have typed in everything you want between the tag, and then use Edit|Insert Matching Bracket.

If you use Insert Matching Bracket when an unmatched closing bracket is highlighted, it inserts the matching opening bracket. The cursor is then placed to the left of the bracket that was inserted. This way you can easily use Edit|Insert Matching Bracket repeatedly to insert multiple opening brackets. E.g. if you type "unclosed)]]" into a C++ file, then put the cursor before "unclosed", and then use Edit|Insert Matching Bracket three times, you'll get "[{(unclosed)]]".

The Insert Matching Bracket command can match up an unmatched bracket even when matching brackets are highlighted. In the C statement `if (a == b) && (c == d)` the last `)` is unmatched. If you put the cursor before the first `(`, EditPad Pro will highlight it as being matched with the `)` after `b`. This clearly shows that the `if` statement isn't properly enclosed in parentheses. If you use Edit|Insert Matching Bracket with the cursor at this position, EditPad Pro looks for the nearest unmatched bracket. In this case that is the final `)` in the statement. EditPad Pro inserts `(`, correctly completing `if ((a == b) && (c == d))`.

If you have turned on Options|Auto Break then automatic breaking and indentation is immediately applied to the inserted bracket. In this case, if you have an unclosed `{` in your C code, for example, place the cursor at the end of the last statement in the block. Then use Edit|Insert Matching Bracket to insert a line break, outdent the line, and insert the closing brace all at once.

If you do want to insert matching brackets immediately after entering an opening bracket, you can turn on Edit|Auto Match Brackets. Then when you type "this " EditPad immediately inserts "" after the cursor. Unlike Insert Matching Bracket, Auto Match Brackets does not move the cursor. So you can continue typing "bold" to get "this bold". But to complete the phrase you will need to move the cursor over the "" before typing " word".

Edit | Auto Match Brackets

If you have a habit of immediately entering a closing bracket or having Edit | Insert Matching Bracket do so after entering an opening bracket, you can turn on Edit | Auto Match Brackets to have EditPad automatically enter the closing bracket for you. If you also have Options | Auto Break turned on then breaking and indentation is immediately applied to both the opening bracket that you entered and the closing bracket that was automatically entered. The cursor stays before the closing bracket and any breaking or indentation applied to it. So you can immediately enter the contents of the bracket pair.

Edit | Auto Match Brackets is only available when bracket matching is enabled on the Brackets page in the file type configuration for the active file's file type. The syntax coloring scheme determines exactly which brackets are matched. See that section in this help file to learn how bracket matching works in EditPad Pro.

The advantage of automatically inserting matching brackets is that you are less likely to end up with unpaired brackets. The disadvantage is that you have to move the cursor beyond the closing bracket yourself. If you want to avoid that, turn off Auto Match Brackets and use Edit | Insert Matching Bracket instead.

Note that “auto match brackets” is a file type specific setting in EditPad Pro. If you change the setting through the Options menu or the toolbar, it is applied to the active file only. If you create a new file or open an existing one then EditPad uses the setting you made on the Brackets page for the type of file you are creating or opening. Also, when you use File | Save As to save the file as a different file type, the setting for the current file changes to what you specified for the new file type.

Edit | Insert Date & Time

Inserts the current date and time at the position of the text cursor. If you click this item directly or use its keyboard shortcut, the date and/or time are inserted using the format you last used. If you've never used this command before, it defaults to the short date and time formats set in Windows.

To use a different date and time format, or to insert the date only or the time only, select Other Date & Time format in the submenu of the Insert Date & Time menu item. A screen will pop up with a list of date and time specifiers. You can type in the date and/or time format that you want in the edit box at the top. You can double-click a specifier in the list to add it to the format. The “actual date and time” box shows a preview of the text that is inserted when you click OK.

The Insert Date & Time submenu keeps a history of the last 16 formats that you used. Click on one to insert the current date and/or time using that format.

Date and Time Format Specifiers

These are the date and time placeholders that you can use. Format specifiers may be written in uppercase as well as in lowercase letters. Both produce the same result.

Specifier	Description
c	Displays the date in short format followed by the time in long format as specified in the regional settings in the Windows Control Panel.

d	Displays the day as a number without a leading zero (1-31).
dd	Displays the day as a number with a leading zero (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat) in the language of the current Windows locale.
dddd	Displays the day as a full name (Sunday-Saturday) in the language of the current Windows locale.
dddddd	Displays the date using the short date format specified in the regional settings in the Windows Control Panel.
ddddddd	Displays the date using the long date format specified in the regional settings in the Windows Control Panel.
m	Displays the month as a number without a leading zero (1-12). If the m specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mm	Displays the month as a number with a leading zero (01-12). If the mm specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mmm	Displays the month as an abbreviation (Jan-Dec) in the language of the current Windows locale.
mmmm	Displays the month as a full name (January-December) in the language of the current Windows locale.
yy	Displays the year as a two-digit number (00-99).
yyyy	Displays the year as a four-digit number (0000-9999).
h	Displays the hour without a leading zero (0-23).
hh	Displays the hour with a leading zero (00-23).
n	Displays the minute without a leading zero (0-59).
nn	Displays the minute with a leading zero (00-59).
s	Displays the second without a leading zero (0-59).
ss	Displays the second with a leading zero (00-59).
z	Displays the millisecond without a leading zero (0-999).
zzz	Displays the millisecond with a leading zero (000-999).
t	Displays the time using the short time format specified in the regional settings in the Windows Control Panel.
tt	Displays the time using the long time format specified in the regional settings in the Windows Control Panel.
am/pm	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'am' for any hour before noon, and 'pm' for any hour after noon. The am/pm specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
a/p	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'a' for any hour before noon, and 'p' for any hour after noon. The a/p specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
/	Displays the date separator character specified in the regional settings in the Windows Control Panel.

:	Displays the time separator character specified in the regional settings in the Windows Control Panel.
&apos ; xx&apos ; / "xx"	Characters enclosed in single or double quotes are displayed as-is, and do not affect formatting.

Edit | Insert Page Break

Select Insert Page Break from the Edit menu to insert a page break into the current file at the current position of the text cursor.

A page break shows up as a horizontal line while you edit the file in EditPad Pro. When you print the file, a new page is started at each page break. The first paragraph below the horizontal line indicating the page break is the first paragraph on the new page.

By default, pressing Ctrl+Enter on the keyboard also inserts a page break. But you can change Ctrl+Enter to insert a different kind of line break. You can do this for the active file with the Convert | Line Break Style menu item. You can change the default for each file type on the Encoding page in the file type configuration.

Changing Ctrl+Enter to insert something other than a page break does not affect the Edit | Insert Page Break menu item. The menu item always inserts a page break.

Edit | Duplicate Line

Duplicates the line the text cursor is on.

If you use the menu item or toolbar button to invoke this command then it is invoked on EditPad's main editor, where you edit files. If you press the shortcut key on the keyboard, it is invoked on whichever editor is showing the text cursor (vertical blinking bar), whether that's the main editor, the search box, or the replace box.

Edit | Delete

Deletes the selected part of the current file. If nothing is selected then the character to the right of the cursor is deleted.

This command is always invoked on whichever editor is showing the text cursor (vertical blinking bar), regardless of whether you use the main menu, a toolbar button, or a keyboard shortcut.

The Edit | Delete menu item is not affected by Block | Persistent Selections. It always deletes the selection if there is one. To delete the character to the right of the cursor when there is a persistent selection, press the Delete key on the keyboard.

Edit | Delete Line

Deletes the line the text cursor is currently on. If the cursor is on the first line of a folded block of lines then all the lines folded underneath the current line are also deleted.

If you use the menu item or toolbar button to invoke this command then it is invoked on EditPad's main editor, where you edit files. If you press the shortcut key on the keyboard, it is invoked on whichever editor is showing the text cursor (vertical blinking bar), whether that's the main editor, the search box, or the replace box.

3. Project Menu

About EditPad Pro Projects

An EditPad Pro Project is a series of files that you can open all at once through Project | Open Project. If you regularly work with a set of related files, it is convenient to create a project file for them. You could have a project for your web site, for the source code of an application you are working on, the chapters of your next book, etc.

When you start EditPad Pro, it opens with an untitled project containing one untitled blank file. To create a project, simply open all the files that should be in the project. If you already have some files open that you don't want to be in the new project, select Project | New Project in the menu before opening the files. After opening the files, save the project with Project | Save Project As. From that point on, EditPad Pro automatically maintains the project. Whenever you open new files into the project, or close files, the project's file list is updated.

If you prefer to actively manage the list of files in the project, turn on the Managed Project option in the Project menu. Then you can use the commands below that item to add files to the project or remove them. You can still open and close files as usual using the File menu, but opened files won't be added to the project, and closed files won't be removed.

Project files also keep track of the last editing position, bookmarks and editing options for each file in the project. When you reopen a project the next time you start EditPad Pro, it will be in exactly the same state as it was in when you last closed the project. EditPad Pro will not bother you with prompts to save the project.

You can have as many projects open at the same time as you want. Creating a new project or opening an existing one does not close the files or projects that you already have open. By default, once you have two or more projects open, a second row of tabs will appear above the row of tabs for files. The new row of tabs allows you to switch between open projects. The tabs for files will list only those files that are part of the active project. Tabs can be configured in the Tabs Preferences.

Project | New Project

Select New Project in the Project menu to start with an untitled project containing a blank, untitled file.

An untitled project is simply a holding space for files. When working with lots of files at the same time, it's often handier to create a bunch of untitled, temporary projects to hold different sets of files. Instead of having one long row of tabs for all the files, you'll have one row of tabs for the projects, and one row of tabs showing only the files in the active project. It's often faster to switch between files by clicking on one project tab and on one file tab, than to scroll through a long list of file tabs.

If you regularly work with the same set of files, save the project with Project | Save Project As. From that point on, EditPad Pro automatically maintains the project. Whenever you open new files into the project, or close files, the project's file list is updated.

If you prefer to actively manage the list of files in the project, turn on the Managed Project option in the Project menu. Then you can use the commands below that item to add files to the project or remove them.

You can still open and close files as usual using the File menu, but opened files won't be added to the project, and closed files won't be removed.

Project | Open Project

Opens an EditPad Pro Project saved with Project | Save Project. Click the Project | Open Project item directly to show a file selection window from which to select the .epp file. Or use the submenu of the Project | Open Project item to reopen a recently closed project. Opening a project removes it from the Project | Open submenu. Closing a project adds it to the top of the Project | Open submenu. You can manage the Project | Open Project submenu just like the File | Open submenu.

You can have as many projects open at the same time as you want. Opening a project does not close any projects that are already open. If you opened individual files before opening the project, those files are in an untitled project that will remain open. If the active project is untitled and holds only a blank untitled file, then the blank project is closed when you open a project.

Each file can be open only once in a single EditPad instance. If the project you're opening holds a file that is already open in another project that is open in EditPad, the file is closed in the other project. If that project is unmanaged, that causes the file to be removed from the other project. If the other project is managed, the file is still closed in that project, but it remains part of it. The file's options, cursor position, and any unsaved changes are all preserved. The file is closed in the other project and moved "as is" to the newly opened project.

Project | Favorite Projects

The Project | Favorite Projects menu works just like the File | Favorites menu, except that it lists EditPad Pro project files rather than text files. You have to save a project before you can add it to the favorites.

Project | Save Project As

Select Save Project As from the Project menu to save the current project under a different name. From that point on, EditPad Pro will automatically maintain the project, updating the file you saved it into. Whenever you open new files into the project, or close files, the project is updated. If the project was previously saved under another name, the old project file will not be updated unless you reopen the old project.

You only need to use Save Project As when you want to save an untitled project, or you want to create a new copy of a previously saved project. There is no need to repeatedly save projects under the same name to prevent losing changes like you would do with files. EditPad Pro always saves project files automatically when the state of the project changes as soon as you've saved the project once to give it a file name.

Project | Save All Files in Project

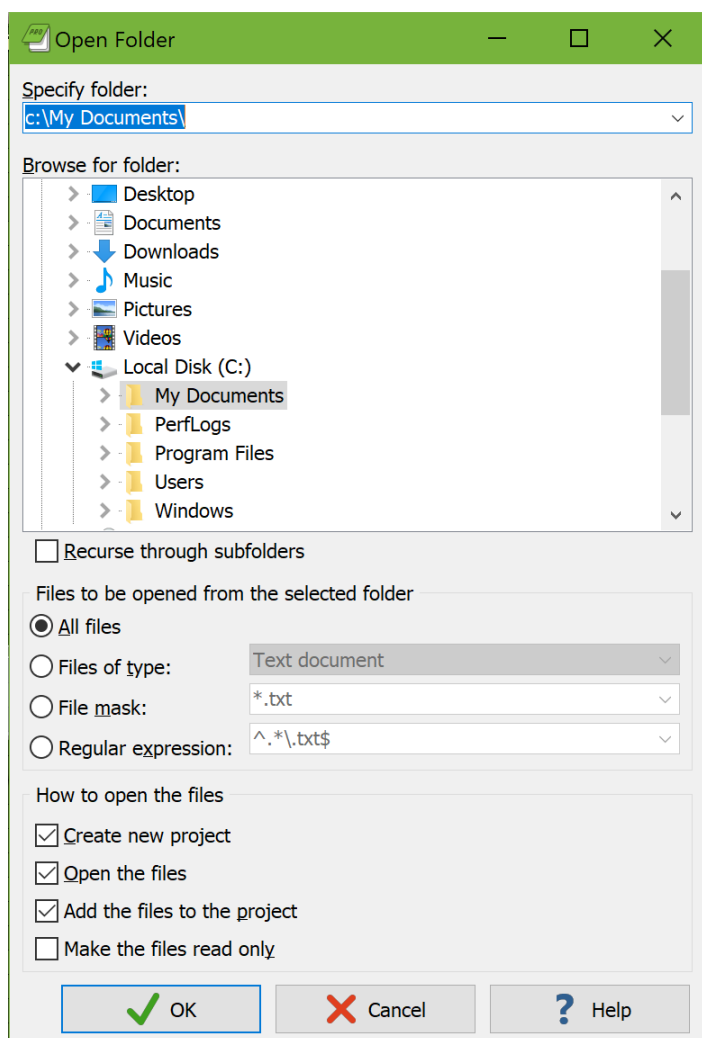
Rapidly performs a File | Save for all open files in the active project.

For untitled files, File | Save As is invoked. If the project has multiple untitled files then you will get a Save As dialog for each of them. If you cancel the Save As dialog then you cancel saving that file and any other files for which you were not prompted yet. This does not undo saving files for which you already clicked Save on the Save As dialog.

If the project has a mixture of untitled files and previously saved files with unsaved changes then EditPad Pro first prompts to save each untitled file. When all untitled files are saved it automatically saves all previously saved files. So if you click Cancel on the Save As dialog for the first untitled file then no files are saved at all.

Project | Open Folder

Select Open Folder from the Project menu to open all files or files of a certain type from a particular folder.



You can type in the path to the folder containing the files you want to open into the drop-down list at the top. If you have recently opened files from this folder, you can select it from the drop-down list. EditPad Pro remembers the last 16 folders. Another way to select the folder is to browse for it using the folder tree. The drop-down list and folder tree are automatically kept in sync.

Tick the option “recurse through subfolders” if you also want to open files contained in subfolders of the folder you selected.

If you want to open all files in the folder (and optionally its subfolders), select the “all files” option. If you want to open only files from a certain type, you can select “files of type” and pick one of the file types from the drop-down list. The list will contain all the file types you defined in Configure File Types. This will open all files that match the file type’s file mask.

To open files matching another list of extensions, select “file mask” and type in the list of extensions or file masks, separated by semicolons, into the “file mask” drop-down list. You can also select a recently used file mask from the drop-down list.

If you choose the “regular expression” option, then the regular expression must find a (partial) match in the name of a file for it to be included in the search. This regex is used to filter files by their names only. If you want to search through the contents of the files, use the Find on Disk command in the Search menu instead of the Open Folder command.

Tick “create new project” to open the files into a new, untitled project. This is the same as using Project|New Project before using Project|Open Folder.

Tick “open the files” to make EditPad Pro actually open all of the files. If you turn this off, then “add files to the project” is automatically turned on, and EditPad Pro adds the files to the project without opening them. The files will show up in the Files Panel as closed files. Since only managed projects can contain closed files, turning off “open the files” automatically changes the current or the new project that the files are added into a managed project.

Tick “add the files to the project” to add the files to the project. If you turn this off, then “open the files” is automatically turned on, and the files will be opened as outside files, without adding them to the project. Since only managed projects can have files open that aren’t part of the project, turning off “add the files to the project” automatically changes the current or the new project that the files are opened into a managed project.

Tick “make the files read only” to open the files in read-only mode.

Recently Opened Folders

The Open Folder item has a submenu that lists recently opened folders. Each time you use Project|Open Folder, the opened folder is added to the top of this submenu. Though the menu shows only the folder’s path, all the settings from the Open Folder window are remembered by the menu. Select Maintain List in the submenu to see the settings used for each folder. When you select one of the remembered folders, EditPad Pro opens the files contained by that folder, using the same file mask and other options as you last used for that folder. If you have opened files from this folder more than once, the folder will appear only once in the Reopen Folder menu, with the most recently used settings.

“Remove Obsolete Folders” removes all folders that no longer exist from the Open Folders submenu. EditPad Pro does not check whether the folders still contain any files matching the file mask. EditPad Pro only checks whether the folders themselves still exist. “Remove All Folders” clears the Open Folders submenu.

Project | Managed Project

New projects created with Project | New Project are always unmanaged projects. The blank project that EditPad Pro starts up with is also unmanaged. If you used EditPad Pro 6 in the past, projects in EditPad Pro 6 were always unmanaged.

In EditPad Pro, an **unmanaged project** means that you won't be explicitly managing which files are part of the project and which aren't. The project simply consists of the files that you have open in the project. Opening a file automatically adds it to the project, and closing a file automatically removes it from the project.

Unmanaged projects are a great way to keep groups of files that you are working with, without requiring any extra effort from you. Simply save the project once to give it a file name. From then on, EditPad Pro automatically updates the project. The project will always open the set of files you had open in it last time.

A **managed project** means that EditPad Pro expects you to manage which files are part of the project, and which aren't. A managed project can contain three kinds of files: **open files** are part of the project and are in EditPad; **closed files** are part of the project, but are not open in EditPad; **outside files** are not part of the project, but are open under the project's tab in EditPad. If you close a managed project and then open it again, the "open files" are opened again, the "closed files" are not opened but remain part of the project, and the outside files are gone (they may still exist as separate files, but the project won't remember them). Open files and outside files have their own tabs under the project's tab. Closed files do not have tabs until you open them again.

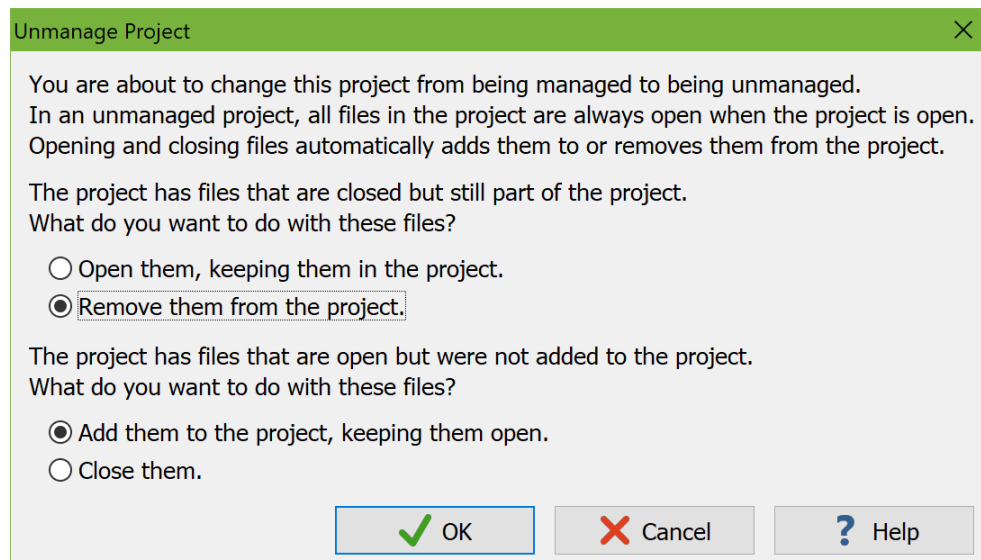
In the Files Panel, all files in unmanaged projects, and open files in managed projects are listed using a plain font. Outside files in managed projects are listed in italics. Closed files in managed projects are listed with a dimmed font. Closed files may not be listed at all if you turned off the option to show closed files in the Files Panel. Double-clicking a closed file opens it.

Managed projects take a bit more effort. It's worth it for projects that consists of large sets of files because a managed project doesn't force you to open all the files that it contains. You can quickly access closed files from the Files Panel and you can even search through closed files without opening them (as long as the project is open). Managed projects are also great for preventing accidental changes. The list of files in the project remains the same, unless you explicitly add files to the project or remove files from the project.

When you use File | Open with a managed project or drag-and-drop a file onto a managed project, the file is opened in the project as an outside file. They disappear from the project when you close them. Use Project | Add to Project instead to open a file and make it part of a managed project. Or if the file is already open as an outside file, use Project | Add Active File or Project | Add Outside Files to make it part of the project.

To turn an unmanaged project into a managed one, select the Managed Project item in the Project menu or click the corresponding toolbar button. The icon on the project's tab changes from the red unmanaged project icon to the blue managed project icon. All files that were open in the unmanaged project become part of the managed project as open files.

You can turn a managed project back into an unmanaged one by selecting the Managed Project item in the Project menu again. If the managed project consisted of open files only, all that happens is that the icon on the project's tab changes from blue to red to indicate the project is unmanaged. If the managed project contained closed files and/or outside files, EditPad Pro will ask what you want to do with those files.



Unmanaged projects cannot contain closed files. If you try to turn a managed project that contains closed files into an unmanaged project, you need to choose what to do with the closed files. If you select to open them, the closed files will be opened and remain part of the unmanaged project as open files. If you select to remove them, the closed files are removed from the project. They will remain as individual files on your hard disk.

Unmanaged projects cannot have outside files. If you try to turn a managed project that has outside files into an unmanaged project, you need to choose what to do with the outside files. If you select to keep them open, the outside files remain open and will be added to the unmanaged project as open files. If you select to close them, the outside files are closed and removed from the project. They will remain as individual files on your hard disk.

The project menu offers several commands that you can use to handle open files, closed files, and outside files in managed projects:

- Add to Project: Select files from disk to be opened and added to the project as open files.
- Add to Project Unopened: Select files from disk to be added to the project as closed files.
- Add Active File: If the active file is an outside file, add it to the project as an open file.
- Add Outside Files: If the project has one or more outside files, add all of them to the project as open files.
- Open Closed Files: If the project contains one or more closed files, open them all.
- Remove From Project: Close the active file and remove it from the project.
- Remove Closed Files: If the project contains one or more closed files, remove them all from the project.
- Close Outside Files: If the project has one or more outside files, close them all and remove them from the project.
- Close All Files: Close all files in the project. If the project has outside files, those are removed from the project. Open files remain part of the project as closed files.

Project | Add to Project

The Add to Project command in the Project menu works like the File | Open command with two small but important differences.

If the active project is a managed project, files opened with Project|Add to Project are opened and added to the project as open files. Files opened with File|Open are opened as outside files that are not part of the project. If you close the managed project and then reopen it, files opened with Project|Add to Project are opened again along with the project, while files opened with File|Open are not remembered by the project.

If you have already opened files with File|Open and they now appear as outside files in your managed project, you can use Project|Add Outside Files to make them part of the project.

If the active project is an unmanaged project, then there is no difference between opening files using File|Open or Project|Add to Project. Both commands add the files and open files to unmanaged projects.

The other difference is the submenu with recently closed files. This applies to both managed and unmanaged projects. The File|Open submenu lists files that were opened and then closed in any project, including untitled projects that were never saved. This list is saved by EditPad along with its general settings and history information. The Project|Add to Project submenu lists files that were opened and then closed in the current project only. This list is saved in the .epp project file.

If you haven't worked with a particular project for a while, any files that you had open in that project and then closed will have long dropped off the 100 files that the File|Open submenu remembers. But when you open that project again, the Project|Add to Project submenu will still remember the last 100 files you closed while working with that project.

The File|Open and Project|Add to Project submenus remember all files that were closed even if you didn't use that menu item to open them. For example, if you double-click on a file in Windows Explorer and then close the file, it will be listed at the top in both File|Open and Project|Add to Project. If you then switch to another project, the file will still be at the top in File|Open but won't be in the Project|Add to Project submenu for the project you switched to. Files that are opened in bulk such as by using Project|Open Folder are not remembered by either menu.

Project|Add to Project Unopened

The Add to Project Unopened command in the Project menu shows a file selection window just like File|Open and Project|Add to Project. Unlike these two commands, Add to Project Unopened does not open any files. The selected files are added to the project as closed files.

Only managed projects can contain closed files. If the active project is unmanaged, using the Add to Project Unopened command automatically turns the project into a managed one, as if you had selected Project|Managed Project in the menu.

Project|Add Active File

The Add Active File command in the Project menu is only available if the active project is managed and the active file is an outside file. An "outside file" is a file that is open under the project's tab, but is not actually part of the project. When a project is closed and reopened, outside files are not remembered. Use the Add Active File command to make the file a part of the project, so it will be remembered by the project.

If the active project has many outside files, you can use Project|Add Outside Files instead if you want to make them all part of the project.

Project|Add Outside Files

The Add Outside Files command in the Project menu is only available if the active project is managed and it has one or more outside files. An “outside file” is a file that is open under the project’s tab, but is not actually part of the project. When a project is closed and reopened, outside files are not remembered. Use the Add Outside Files command to make all such files part of the project, so they will be remembered by the project.

If you want to add some of the outside files but not all of them, use the Project|Add Active File to add them one by one.

Project|Open Closed Files

The Open Closed Files command in the Project menu is only available if the active project is managed and it has one or more closed files. A “closed file” in a managed project is a file that was closed but is still part of the project. The Open Closed Files command opens all such files in the active project.

If you want to open some, but not all, closed files in a managed project, use the Files Panel. If you want to search through all files in a managed project, there is no need to open closed files. Simply turn on the Closed Files search option to search through all files that are part of the project, whether they are open or closed.

The Open Closed Files command is not available for unmanaged projects, because closing a file in an unmanaged project automatically removes it from the project. For managed projects, the Open Closed Files command does not reopen files that were removed from managed projects with Project|Remove From Project and does not reopen outside files that were not part of the project when they were closed. Such files may still be listed in the Project|Add to Project submenu. You can use that submenu to reopen them.

Project|Remove from Project

The Remove from Project command in the Project menu closes the active file and removes it from the project. For unmanaged projects, there is no difference between Project|Remove from Project and File|Close. For managed projects, files closed with Project|Remove from Project are completely removed from the project, while files closed with File|Close remain part of the project if they were added to the project.

If you want to remove a file that is already closed from a managed project, use the Files Panel. If you want to remove all closed files from a managed project, use Project|Remove Closed Files. If you want to remove all outside files from a project, use Project|Close Outside Files.

Files that are removed from a project are added to the File|Open and Project|Add to Project submenus that list recently closed files, unless the files were opened in bulk with Project|Open Folder or a similar command.

Project | Remove Closed Files

The Open Closed Files command in the Project menu is only available if the active project is managed and it has one or more closed files. A “closed file” in a managed project is a file that was closed but is still part of the project. The Remove Closed Files command removes all such files from the active project.

If you want to remove only closed files that no longer exist on disk, use Project | Remove Obsolete Files. If you want to remove only certain closed files from the project, use the Files Panel.

Files that are removed from a project are added to the File | Open and Project | Add to Project submenus that list recently closed files, unless the files were opened in bulk with Project | Open Folder or a similar command.

Project | Remove Obsolete Files

The Remove Obsolete Files command in the Project menu removes all files that no longer serve any purpose from the project. A file does not serve any purpose when two conditions are true. First it needs to be a closed file or its tab needs to be empty. This means EditPad has no contents that it could save to the file. Second, the file must either be untitled or have been loaded from a file on disk that no longer exists. This means EditPad cannot reload the file from disk. Files that were opened from FTP are never closed by this command. A file with no contents in EditPad and no contents on disk is obsolete.

Obsolete files are not added to the File | Open and Project | Add to Project submenus that list recently closed files.

Project | Close Outside Files

The Close Outside Files command in the Project menu is only available if the active project is managed and it has one or more outside files. An “outside file” is a file that is open under the project’s tab, but is not actually part of the project. When a project is closed and reopened, outside files are not remembered. Use the Close Outside Files to remove all such files from the project immediately.

Files that are removed from a project are added to the File | Open and Project | Add to Project submenus that list recently closed files, unless the files were opened in bulk with Project | Open Folder or a similar command.

Project | Close All Files

Use the Close All command in the Project menu to close all files in the active project. The result is the same as if you had used File | Close for each file in the project. The project itself remains open.

If the project is unmanaged, all files are removed from the project.

If the project is managed, files that are part of managed projects remain part of the project as closed files. The Files Panel and Search Panel can show and search through files that are closed but are still part of managed projects. Any outside files that were open under the project’s tab are removed from the project, because they were never an actual part of the project.

Project | Reload Files from Disk

Select Reload Files from Disk from the Project menu to revert all open files in the active project to the status they had when they were last saved. This can be useful if you are viewing files that are being written to by other programs.

If one or more of the files in the project have unsaved changes in EditPad Pro, or if you turned on the option to always prompt in the Open Files Preferences, then EditPad Pro asks you to confirm which files you want to reload. Files that have unsaved changes in EditPad Pro are marked as “Modified” in the list. Those changes are lost if you reload those files. Tick the checkbox next to each file that you want to reload.

Project | Save Copy of Project As

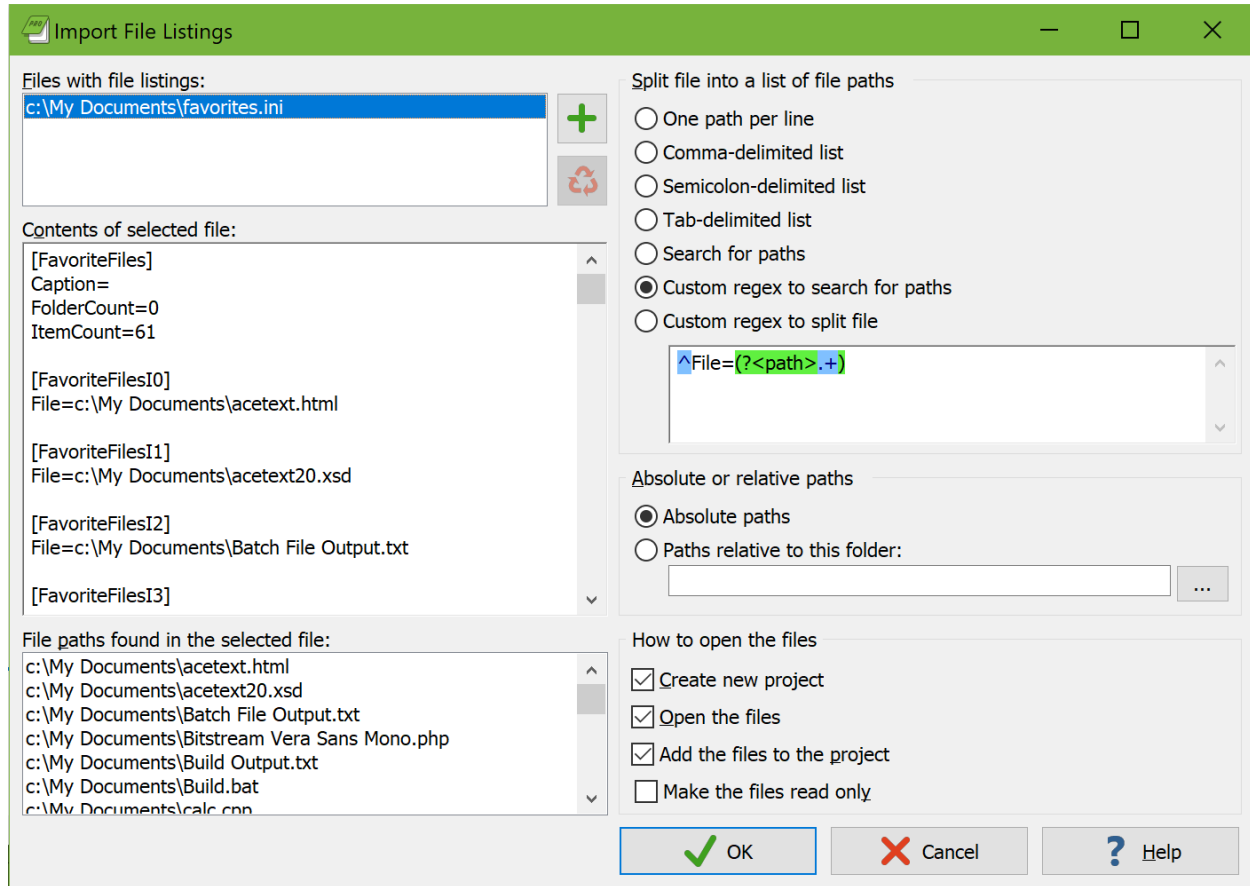
Select Save Copy of Project As to save the project under a new name, but continue updating the project under its original name. Use this command if you want to save a milestone copy of the project that you can go back to later. EditPad Pro will not keep the copy of the project updated along with the original. The new copy will stay as it is, until you open it or overwrite it when saving another project. By opening the copy of the project, you can go back to the project in the state it was in when you saved the copy.

Project | Rename / Move Project

Select Rename/Move Project in the Project menu to save the project under a new name and delete the original project. This effectively renames the project or moves it to a new location on disk.

Project | Import File Listing

Select Import File Listing in the Project menu to read a list of file paths from one or more text files, and then open the files indicated by the paths in that list.



In the window that appears, click the button with the green plus symbol next to the “files with file listings” list to select one or more text files that list the files or folders you want to search through. You can use the green plus button repeatedly to add files from different folders to the list. The red X button deletes the selected file from the list.

Click on one of the files that you added to the “files with file listings” list. EditPad Pro then shows the raw contents of that file in the “contents of the selected file” box. EditPad Pro also shows the file and folder paths that it has detected in that file in the “file paths found in the selected file” list. The settings on the right-hand side of the Import File Listings screen determine how EditPad Pro detects those paths. EditPad Pro automatically filters out anything that does not look like a valid path.

Choose one of the options in the “split file into a list of file paths” box to tell EditPad Pro how your list of paths is delimited. If your list doesn’t use a consistent delimiter, select “search for paths” to tell EditPad Pro to extract all absolute paths from the file, regardless of any other text that may occur in the file. If you want EditPad Pro to extract only certain paths from the file, select one of the two “custom regex” options and type in a regular expression in the box below them. The “custom regex to search for paths” option needs a regular expression that matches the paths you want to mark in the file selection. EditPad Pro uses the whole regex match as the path unless it contains a named capturing group called “path”. E.g. `^File=(?'path'.*)` extracts the paths from “File” values in an .ini file. The “custom regex to split file” option needs a regular expression that matches the delimiters between those paths. E.g. `[\r\n;]+` allows line breaks and semicolons as delimiters.

If you select “absolute paths”, EditPad Pro only uses fully qualified paths such as `c:\folder\file.txt` and `\\server\share\folder\file.txt`. Any relative paths in the file listing you’re importing are ignored. If you want EditPad Pro to process relative paths as well then you need to select the “paths relative to this folder option”. Type in the base folder below that option, or click the (...) button to select it from a folder tree. Note that if your file contains text in addition to paths, you need to use one of the “custom regex” options to tell EditPad Pro how to find only the actual paths. Otherwise, EditPad Pro will treat each word in the text as a file name. You cannot use the “search for paths” option because that option finds absolute paths only, regardless of the “absolute or relative paths” setting.

Once you’ve set the options that make EditPad Pro find the paths that you want to import, you need to tell EditPad Pro what you want to do with the files those paths point to. Tick “create new project” to open the files into a new, untitled project. This is the same as using **Project | New Project** before using **Project | Import File Listings**.

Tick “open the files” to make EditPad Pro actually open all of the files. If you turn this off, then “add files to the project” is automatically turned on, and EditPad Pro adds the files to the project without opening them. The files will show up in the Files Panel as closed files. Since only managed projects can contain closed files, turning off “open the files” automatically changes the current or the new project that the files are added into a managed project.

Tick “add the files to the project” to add the files to the project. If you turn this off, then “open the files” is automatically turned on, and the files will be opened as outside files, without adding them to the project. Since only managed projects can have files open that aren’t part of the project, turning off “add the files to the project” automatically changes the current or the new project that the files are opened into a managed project.

Tick “make the files read only” to open the files in read-only mode.

Recently Imported File Listings

The Import File Listings item has a submenu that lists recently imported file listings. Each time you use **Project | Imported File Listing**, the files you imported are added to the top of this submenu. If you imported a single file, the submenu lists the full path to the file. If you imported multiple files at once, the submenu has one item with a comma-delimited list listing all the file names that fit within the maximum length for a menu item. The submenu item remembers all the settings you made in the Import File Listings window. When you select one of the remembered file listings from the submenu, it is imported again using the exact same settings.

“Remove Obsolete” removes all files that no longer exist from the Import File Listings submenu. This removes all file listings that try to import from a file that no longer exists. If a file listing imports from files that still exist, but no paths can be found in those files, then the file listing is removed as well.

Project | Export File Listing

Select **Export File Listing** to save a text file that lists the full paths to all the files in the active project. The paths are delimited by line breaks. If the project is managed then open files, closed files, and outside files are all added to the exported file listing.

Project | Delete Project

Select Delete Project in the Project menu to close the active project and delete the .epp file that you saved with Project | Save Project As. Only the .epp project file is deleted. The files that were open in the project are not deleted.

Project | Close Project

Closes the current project and all the files in the project. If any files in the project have unsaved modifications, depending on the choice you made in Open Files Preferences, EditPad will ask if you want to save or not, automatically save, or automatically discard the changes. When you close the last project, EditPad automatically starts with a new untitled project, with one blank untitled file.

If you previously saved the project with Project | Save Project As, then the closed project is added to the top of the Project | Open Project submenu. You can use that submenu to quickly reopen the project. When you open the project again, it will show up with the same set of open files as you had open when you closed the project. The only exception are outside files in managed projects. Since outside files aren't part of the project, they aren't remembered by it and aren't reopened when you reopen the project.

If you did not previously save the project, it will be closed without asking you whether you want to save it. If you want to work with the same set of files again, you'll have to reopen the files separately.

Project | Close All but Current

Closes all projects and all the files in them, except for the current project, as if you used Project | Close Project on each of those projects.

4. Search Menu

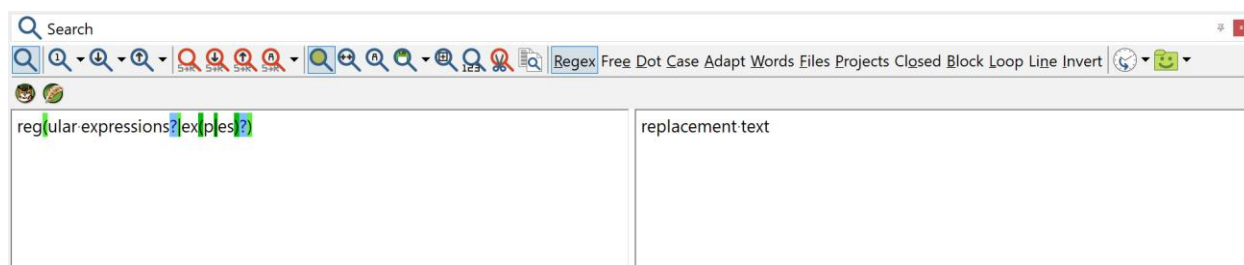
Search | Prepare to Search

Use the Prepare to Search command in the Search menu to enter a search term that you can then use for searching or for searching and replacing. Its default keyboard shortcut is Ctrl+F. Exactly what the Prepare to Search command does depends on the state EditPad is in and on your preferences.

EditPad provides two search interfaces. When you start EditPad the first time, the **search toolbar** is docked to the bottom of EditPad's window. The search toolbar provides all of EditPad's search and replace commands. It has two small edit boxes for entering a short single-line search term and replacement text. You can show or hide the search toolbar by right-clicking on any toolbar or the main menu and selecting Search.



If you use the Search | Multi-Line Search Panel command, EditPad switches to the full **search panel**. This panel has two large multi-line edit controls for entering the search term and the replacement text. It also incorporates the search toolbar with all the search and replace commands, but without the two small edit boxes.



If the full search panel is visible, the Prepare to Search item puts keyboard focus in the edit box for the search term. If the search panel is not visible but the search toolbar is visible, Prepare to Search puts keyboard focus in the edit box for the search term. If the search panel and search toolbar are invisible, Prepare to Search opens the full search panel and puts keyboard focus in the edit box for the search term.

If the “use the current word or the selected text as the default search text” preference is turned on, then the Prepare to Search command uses the selected text as the default search term, as long as the selection does not span multiple lines. If there is no selection, then the word under the cursor is used as the default search term. If there is no selection and no word under the cursor, then the last used search term stays in place. If the preference is turned off, the last used search term always stays in place. The Prepare to Search command always selects the search term. You can replace the search term directly by entering a new one.

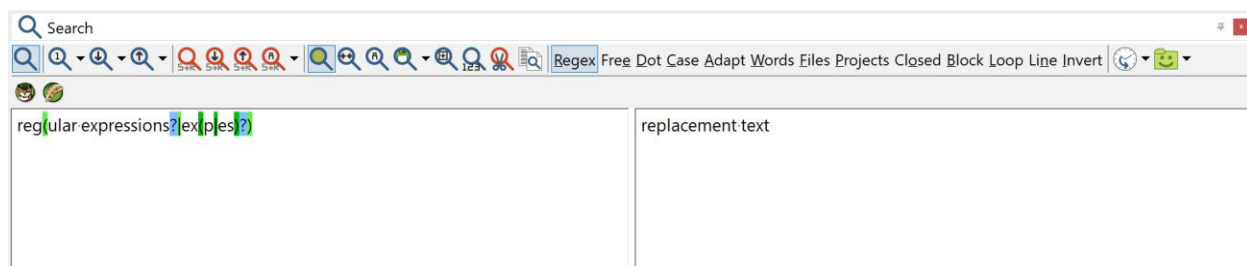
If the “automatically turn on “selection only” when multiple lines of text are selected” preference is turned on, then the Prepare to Search command automatically turns on the Selection Only or Block search option when there is a selection that spans multiple lines. When the preference is off, the state of this option is unchanged.

Search | Multi-Line Search Panel

EditPad provides two search interfaces. When used by itself, the **search toolbar** is docked to the bottom of EditPad's window. The search toolbar provides all of EditPad's search and replace commands. It has two small edit boxes for entering a short single-line search term and replacement text.



The full **search panel** has two large multi-line edit controls for entering the search term and the replacement text. It also incorporates the search toolbar with all the search and replace commands, but without the two small edit boxes.



When you start EditPad for the first time, only the search toolbar is visible. You can switch to using the full search panel by selecting Multi-Line Search Panel in the Search menu. You can switch back to using the search toolbar only by using Multi-Line Search Panel again.

When the search toolbar is visible but the full search panel is not, you can hide the search toolbar by right-clicking on any toolbar or the main menu and selecting Search. When both the search toolbar and search panel are invisible, the Multi-Line Search Panel makes both visible as usual, but using Multi-Line Search Panel again will hide both the search toolbar and search panel again. If you want to bring back the search toolbar without the full search panel, right-click again on any toolbar or the main menu and select Search again.

To summarize: EditPad has 3 modes for its search interface: toolbar and panel hidden, toolbar visible alone, panel with toolbar visible. The Multi-Line Search Panel command switches from one of the first 2 modes to the “panel with toolbar visible” mode, and back to whichever of the first 2 modes you were using. Right-clicking on a toolbar and selecting Search switches between the first two modes.

Both the search and replace edit boxes on the search panel are full-blown multi-line edit controls, just like EditPad's main editor. They provide syntax coloring for regular expressions and match placeholders. Press Shift+Enter on the keyboard to type in and search for or replace with a newline. To type in a tab, press Shift+Tab. To switch from search to replace and back, press Tab.

If you right-click the search or replace edit box, you will see a list of the last 16 search or replace texts you used. Click on one to use it again.

Search | Instant Highlight

Select Instant Highlight in the Search menu or press its keyboard shortcut to highlight all occurrences of the selected text or the word under the cursor, without using the search toolbar or panel. Highlighting persists

when you switch between files. Use Instant Highlight with the same selection or on the same word to turn it off. If you turn on the “double-clicking a word instantly highlights all occurrences of that word” search preference, then double-clicking a word also turns Instant Highlight on and off.

Only one search term can be highlighted at any time. If you use both Highlight All and Instant Highlight then using either command turns off the other. All other settings on the search toolbar are ignored. The instant search commands have their own Instant Search Options.

To change the color or underlining style used for highlighting search matches, customize the color palette and change the “Editor: Highlighted search match” color.

Search | Instant Incremental Search

Select the Instant Incremental Search item in the Search menu or press its keyboard shortcut to enter a search term without using the Search panel. Turning on this command has no immediate effect. When you enter text into the file, the text is not inserted into the file. Instead it is searched for. Words or single-line phrases that you paste are also searched for.

While you enter the search term in instant incremental search mode, EditPad Pro remembers previous search matches as you type. If you type “Test”, EditPad Pro searches for “T” starting from the position of the text cursor. As you type, it searches for “Te” beginning at the start of the match for “T”, then for “Tes” at the start of the match for “Te”, and finally for “Test”. If there is no search match, nothing happens.

When you press Backspace, EditPad Pro goes back to the previously highlighted match for “Tes”. If you continue pressing Backspace, it goes back to the match for “Te”, “T”, and finally goes back to where the search started. So while entering the search term, you can easily change your mind about the search term you’re looking for by backing up with the Backspace key, and typing in the new search term. Backing up does not cause EditPad Pro to forget match positions. If you type in “Joe” after erasing “Test”, EditPad Pro searches for “J” until “Joe” beginning from the original starting position. If you then back up again and retype “Test”, the match previously found for “Test” is highlighted.

You can find other occurrences of the most recently found search term by pressing the arrow keys on the keyboard. The Up and Left keys find the previous match. The Down and Right keys find the next match. While an instant incremental search is in progress, the Instant Find Next and Instant Find Previous commands also use the incremental search term.

If you take any other action in the file besides scrolling then the instant incremental search mode is automatically disabled. You can also select the menu item or press the keyboard shortcut or toolbar button again to turn off instant incremental search.

This command is intended to be used without the Search toolbar. By default, it has a button on the main toolbar. This button has a drop-down menu with instant search options that affect instant searches. This command ignores all settings on the search toolbar.

Use Search | Incremental Search instead if you want to perform an incremental search by entering your search term into the Search box.

Search | Instant Find Next

Search | Instant Find Next looks for the next occurrence in the current file of the word under the text cursor. When the word is found, Instant Find Next selects it. If there is no word under the text cursor, or it doesn't occur again in the file, nothing happens at all. If some text is selected when you invoke Instant Find Next, and the selection does not span more than one line, Instant Find Next will search for the selected text rather than the word under the cursor.

The purpose of Instant Find Next and Search | Instant Find Previous is to quickly “walk” through the file, stopping at each spot where the word occurs. Since a successful match selects the word that was searched for, you can instantly search for further matches.

If Instant Incremental Search is active, then Instant Find Next finds the next occurrence of the incremental search term.

This command is intended to be used without the Search toolbar. By default, it has a button on the main toolbar. To the left of this button is the Instant Incremental Search button. It has a drop-down menu with instant search options that affect Instant Find Next.

Search | Instant Find Previous

Instant Find Previous works just like Instant Find Next, except that it searches backwards through the current file, starting at the current position of the text cursor, or the start of the selection (if any). If you press Instant Find Previous as many times as you pressed Instant Find Next, or vice versa, you'll be back where you started.

Search | Instant Replace

The Instant Replace command in the Search menu allows you to instantly replace all occurrences of a word or of part of a line simply by editing one of those occurrences. The Instant Replace command ignores all settings on the search toolbar. The instant search commands have their own Instant Search Options.

Instant Replace can work in two ways. If there is no selection, or if the selection does not span multiple lines, then Instant Replace highlights all occurrences of the selected text or of the word under the cursor throughout the file. If you turn on the “alt+double-clicking a word instantly replaces all occurrences of that word” search preference, then double-clicking a word activates Instant Replace for that word throughout the file. Nothing happens if the word you try to activate Instant Replace on occurs only once in the file. While Instant Replace is active, editing the occurrence under the cursor edits all other occurrences in the file. If selections are not persistent, moving the cursor away from the occurrence under the cursor immediately ends Instant Replace. If selections are persistent, instant replace ends when you make a new selection or you make an edit outside any highlighted occurrences.

The other way to use Instant Replace is to make a multi-line selection before invoking Instant Replace. Then Instant Replace highlights all occurrences of the word under the cursor within the selection. The selection is turned into the search range. Editing the word under the cursor edits all occurrences of that word within the search range. You can move the cursor to another word within the search to edit all occurrences of that word. You can make a selection within the search range to edit all occurrences of that selection. If selections are not

persistent, moving the cursor outside of the search range immediately ends Instant Replace. If selections are persistent, making an edit outside any highlighted occurrences ends the Instant Replace. Making a selection outside the search range does not end Instant Replace when selections are persistent.

Only one search term can be highlighted at any time. Instant Replace turns off both Highlight All and Instant Highlight.

To change the color or underlining style used for highlighting search matches, customize the color palette and change the “Editor: Highlighted replacement” color. Change the “Editor: Search range” color to change the search range highlight.

Instant Search Options

Instant searches do not use the search options that you can set on the Search toolbar. Instant searches have a separate set of reduced search options that only apply to instant searches.

You can set the options for instant searches via the Search|Instant Search Options menu item. Directly clicking this menu item shows a popup dialog with a checkbox for each option that applies to instant searches. If you only need to toggle one search option, it will be quicker to use the submenu of Search|Instant Search Options to toggle that option. By default, the Instant Incremental Search button on the main toolbar has a drop-down menu with the same instant search options.

These are the three options that apply to instant searches:

- Case Sensitive
- Whole Words Only
- Loop Automatically

Instant Search Option: Case Sensitive

If you turn on the “case sensitive” instant search option then instant search commands take the difference between lowercase and uppercase letters into account. E.g. “Dog” will be considered to be different from “dog”. Otherwise, “dog”, “Dog”, “DOG”, and even “doG” are all the same.

Instant Search Option: Whole Words Only

If you turn on the “whole words only” instant search option then the instant search commands only match whole words. If this option is on, when looking for “cat”, EditPad does not consider the first three letters of “category” a valid search match. Otherwise, it does.

Instant Search Option: Loop Automatically

Turn on the “loop automatically” instant search option if you want Search|Instant Find Next and Search|Instant Find Previous to automatically restart searching from the start or end of the file when no

further search matches can be found. When EditPad finds a match after having looped (i.e. restarted from the start or end), then the glyph on instant find button that you clicked will flash briefly, showing a “looped” glyph.

Search | Find First

After using Prepare to Search or pressing Ctrl+F and entering a search term, click the Find First button on the Search toolbar or select the Find First item in the Search menu to find the first search match. If the “selection only” option is on, Find First starts searching from the start of the search range. If “all files” is on, Find First starts searching from the start of the first file in the current project. If “all projects” is on, Find First starts searching from the start of the first file in the first project. If none of these three options are on, Find First starts searching from the start of the current file.

If the search term can be found then EditPad switches to the file it was found in and selects the matched text. If the search term cannot be found then nothing happens. Only the Find First button will briefly flash its icon to indicate it failed.

In EditPad Pro, the Find First button on the search toolbar has a drop-down menu. This drop-down menu has three items that you can use to find a specific search match, counting search matches from the start. The Find 10th item finds the 10th search match. It starts searching from the beginning to find the first search match. It then immediately continues to find the second, third etc. until it arrives at the 10th match. The 10th match is then selected. If there are less than 10 search matches, nothing happens. Only the Find First button will briefly flash its icon to indicate the search failed. Similarly, the Find 100th item skips the first 99 search matches and selects match #100.

The Find Nth item asks you for the number of the match you want to find. If you enter 42, for example, EditPad skips the first 41 search matches and highlights the 42nd.

Search | Find Next

After using Prepare to Search or pressing Ctrl+F and entering a search term, click the Next button on the Search toolbar or select the Find Next item in the Search menu to find the next search match. If the edit box for entering the search term still has keyboard focus, you can also find the next search match by pressing Ctrl+F again or by pressing Enter.

Find Next starts searching from the current position of the text cursor in the current file. If the “selection only” option is on, Find Next searches until the end of the search range. If “all files” is on, Find Next searches until the end of the file and then continues at the start of the next file until a search match is found. If “all projects” is on, and Find Next reaches the last file in the current project, Find Next will continue with the next project.

If the search term can be found, EditPad switches to the file it was found in and selects the matched text. If the search term cannot be found, nothing happens. Only the Find Next button will briefly flash its icon to indicate it failed.

In EditPad Pro, the Find Next button on the search toolbar has a drop-down menu. The first item in the drop-down menu is Find in Next File. When you select this item, EditPad starts searching from the start of

the next file. You can use Find in Next File even when the “all files” option is off. If the search term cannot be found in the next file, and one or more of the “all files”, “all projects” and “loop automatically” options are on, Find in Next File continues the search just like the Find Next command itself.

The drop-down menu also has three items that you can use to find a specific search match, counting search matches from the start. The Find 10th Next item skips over the next 9 search matches and highlights the 10th. The result is the same as if you had clicked the Find Next button 10 times. If there are no further 10 search matches, the result is slightly different. Find Next 10th then does not do anything other than flashing the Find Next button to indicate failure. Similarly, the Find 100th Next item skips over the next 99 search matches and selects match #100.

The Find Nth Next item asks you for the number of the match you want to find. If you enter 42, for example, EditPad skips the next 41 search matches and highlights the 42nd.

Search | Find Previous

After using Prepare to Search or pressing Ctrl+F and entering a search term, click the Previous button on the Search toolbar or select the Find Previous item in the Search menu to find the previous search match. Find Previous starts searching from the current position of the text cursor in the current file. If the “selection only” option is on, Find Previous searches until the start of the search range. If “all files” is on, Find Previous searches until the start of the file and then continues at the end of the previous file until a search match is found. If “all projects” is on, and Find Previous reaches the first file in the current project, Find Previous will continue with the previous project.

If the search term can be found, EditPad switches to the file it was found in, and selects the matched text. If the search term cannot be found, nothing happens. Only the Find Previous button will briefly flash its icon to indicate it failed.

The Find Previous command is not available in regular expression mode unless you also turn on the line by line mode. Regular expressions cannot search backwards. But EditPad can iterate over the lines in your file from bottom to top and apply the regex from left to right on each line to find the previous line in which the regex can find a match.

In EditPad Pro, the Find Previous button on the search toolbar has a drop-down menu. The first item in the drop-down menu is Find in Previous File. When you select this item, EditPad starts searching from the end of the previous file. You can use Find in Previous File even when the “all files” option is off. If the search term cannot be found in the previous file, and one or more of the “all files”, “all projects” and “loop automatically” options are on, Find in Previous File continues the search just like the Find Previous command itself.

The drop-down menu also has three items that you can use to find a specific search match, counting search matches from the start. The Find 10th Previous item skips over the previous 9 search matches and highlights the 10th. The result is the same as if you had clicked the Find Previous button 10 times. If there are no 10 preceding search matches, the result is slightly different. Find 10th Previous then does not do anything other than flashing the Previous button to indicate failure. Similarly, the Find 100th Previous item skips the previous 99 search matches and selects match #100.

The Find Previous Nth item asks you for the number of the match you want to find. If you enter 42, for example, EditPad skips the 41 preceding search matches and highlights the 42nd.

Search | Find Last

Select the Find Last item in the Search menu to find the last search match. If the “selection only” option is on, Find Last starts searching from the end of the search range. If “all files” is on, Find Last starts searching from the end of the last file in the current project. If “all projects” is on, Find Last starts searching from the end of the last file in the first project. If none of these three options are on, Find Last starts searching from the end of the current file. Find Last searches backwards until a match is found.

If the search term can be found, EditPad switches to the file it was found in, and selects the matched text. If the search term cannot be found, nothing happens. Only the Find Last button on the search panel will briefly flash its icon to indicate it failed. Since there’s no Last button, the First button flashes instead.

In EditPad Pro, the Find Last command is also available through the drop-down menu of the First button on the search panel’s toolbar.

The Find Last command is not available in regular expression mode. Regular expressions cannot search backwards.

Search | Replace Current

After finding a search match with Find First, Find Next, Find Previous, or any of the other search commands, click the Replace button on the Search toolbar or select the Replace Current item in the Search menu to replace the selected search match with the replacement text.

Search | Replace and Find Next

After finding a search match, click the Next button to the right of the Replace button on the Search toolbar or select the Replace and Find Next item in the Search menu to replace the selected search match with the replacement text, and to immediately continue searching for the next search match. This button is a shortcut to clicking the Replace button followed by clicking the Find Next button.

You can selectively replace matches by starting with the Find First button to find the first match. Then click Find Next if you don’t want to replace the match. Or click Replace and Find Next if you do want to replace the match. Either way the next match is selected. Repeat this until no further matches can be found. You can make any other edits while doing this. You can even change the search term or replacement. Simply click Find Next or Find Previous to go to the next or previous match that you haven’t replaced yet.

Search | Replace and Find Previous

After finding a search match, click the Previous button to the right of the Replace button on the Search toolbar or select the Replace and Find Previous item in the Search menu to replace the selected search match with the replacement text, and to immediately continue searching for the previous search match. This button is a shortcut to clicking the Replace button followed by clicking the Find Previous button.

The Replace and Find Previous command is not available in regular expression mode. Regular expressions cannot search backwards.

Search | Replace All

After using Prepare to Search or pressing Ctrl+F and entering a search term and replacement text, click the Replace All button on the Search toolbar or select the Replace All item in the Search menu to find the first search match. Replace All starts searching from the beginning just like Find First does. If it finds a match, it replaces it like Replace Current would, and then continues searching like Find Next would, replacing all further search matches.

The only difference between clicking Replace All and using all the individual action buttons is that Replace All is silent. It simply replaces all matches without moving the selection or the text cursor. Even when searching through all projects or all files, the current file remains active throughout the operation.

If no search matches can be found, EditPad briefly flashes the Replace All button's icon to indicate failure. If you used the keyboard shortcut to do the Replace All with the search panel hidden, check the status bar for the result.

If there are a lot of search matches, replacing them all might take a while. A progress meter will pop up to indicate EditPad's progress after a second or two.

In EditPad Pro, the Replace All button on the search toolbar has a drop-down menu with two items: Replace All Next and Replace All Previous. Replace All Next performs the search-and-replace on the part of the file after the position of the text cursor. If a search match is already highlighted, that match is replaced too. Effectively, Replace All Next does in one click what you can do by clicking Replace and Find Next until no further matches can be found.

The Replace All Previous button does the same, except that it searches backwards, performing the search-and-replace on the part of the file before the position of the text cursor. If a search match is already highlighted, that match is replaced too. Replace All Previous is not available in regular expression mode. Regular expressions cannot search backwards.

EditPad does not have a “prompt on replace” option. If you only want to replace certain matches, use the technique described in the topic for Search | Find Next. The benefit of EditPad's way of working is that you can make other edits and even change the search term or replacement text in between making replacements. You can even go backwards with Search | Replace and Find Previous.

Search | Highlight All

After using Prepare to Search or pressing Ctrl+F and entering a search term, click the Highlight button on the Search toolbar or select the Highlight All item in the Search menu to turn on the search match highlighting mode. In this mode, all search matches will be highlighted. The Selection Only option is ignored.

When you switch between files, search matches in the newly activated line are highlighted. The “all files” setting does not affect match highlighting. To stop highlighting search matches, simply click the Highlight button again.

To change the color or underlining style used for highlighting search matches, customize the color palette and change the “Editor: Highlighted search match” color.

Search | Incremental Search

Click the Incremental button on the Search toolbar or select the Incremental item in the Search menu to toggle incremental search mode on or off. When off, entering a search term has no immediate effect. When on, editing the search term makes EditPad Pro look for the next search match immediately, starting from the current position of the text cursor.

While you enter the search term in instant incremental search mode, EditPad Pro remembers previous search matches as you type. If you type “Test”, EditPad Pro searches for “T” starting from the position of the text cursor. As you type, it searches for “Te” beginning at the start of the match for “T”, then for “Tes” at the start of the match for “Te”, and finally for “Test”. If there is no search match, nothing happens.

When you press Backspace, EditPad Pro goes back to the previously highlighted match for “Tes”. If you continue pressing Backspace, it goes back to the match for “Te”, “T”, and finally goes back to where the search started. So while entering the search term, you can easily change your mind about the search term you’re looking for by backing up with the Backspace key, and typing in the new search term. Backing up does not cause EditPad Pro to forget match positions. If you type in “Joe” after erasing “Test”, EditPad Pro searches for “J” until “Joe” beginning from the original starting position. If you then back up again and retype “Test”, the match previously found for “Test” is highlighted.

What does cause EditPad Pro to forget previous incremental search matches is taking any action in the main editor, including something as trivial as moving the text cursor or switching to another file. EditPad Pro then moves the starting position for the next incremental search to the new position of the text cursor.

The incremental search mode respects all search options, including “all files“, “selection only“, “loop automatically“, etc.

Search | List All Matches

After using Prepare to Search or pressing Ctrl+F and entering a search term, use the List All Matches command in the Search menu to display a list of all search matches with one line of context in a side panel. All matches in the current file, current project, or all files will be listed depending on the All Files, All Projects, and Closed Files search options.

In the side panel, you can double-click a highlighted search match to activate the file it was found in and select the match. If you edit files after listing all matches, the stored match positions are adjusted so that double-clicking a search match highlights it, even if its position in the file has shifted. The actual search matches and their context are not updated when you edit files. Use the List All Matches command again to repeat the search.

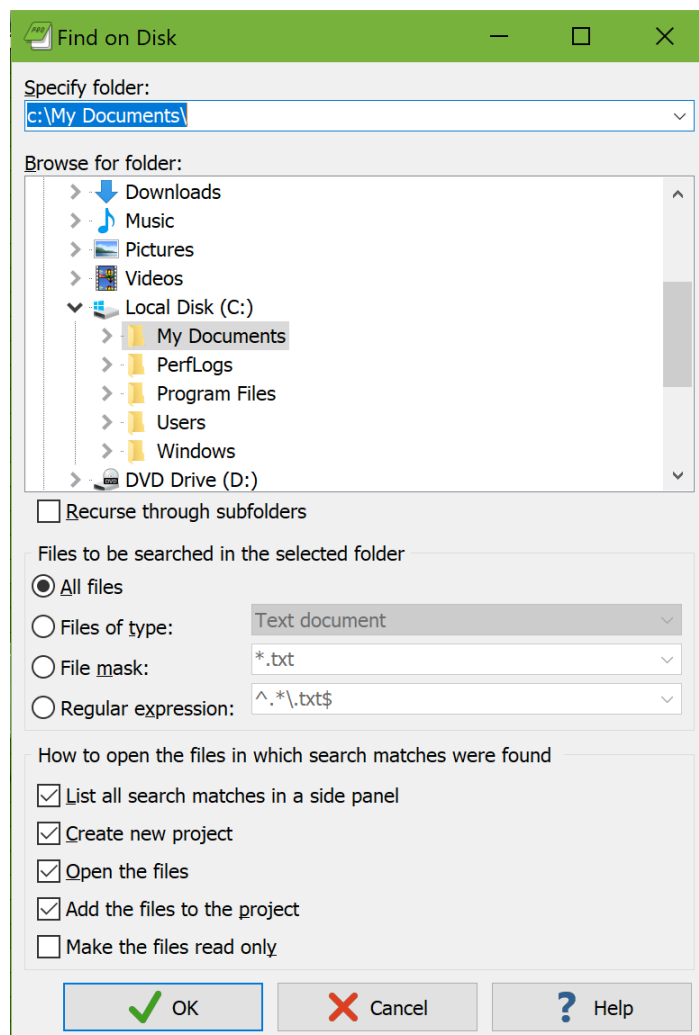
If you use a regular expression that may find zero-length matches then the List All Matches command does not add those zero-length matches to the Search Matches panel. If the regex finds only zero-length matches then the List All Matches panel remains empty. If some matches are zero-length and some are not then the non-zero-length matches are added to the Search Matches panel.

Search | Find on Disk

After using Prepare to Search or pressing Ctrl+F and entering a search term, use the Find on Disk command in the Search menu to search through the files in a particular folder, even if you don't have those files open in EditPad Pro. The disk-based search uses the search term and search options from the search toolbar or panel. Only the All Files, All Projects, and Closed Files search options are ignored.

If the disk-based search includes files that you have open in EditPad Pro, then the Find on Disk command searches through the files open in EditPad Pro rather than through the files on disk. This means that if the files have unsaved changes in EditPad Pro, the unsaved changes are searched through rather than the original file on disk.

The Find on Disk command shows a folder selection screen that is very similar to the one shown by Project | Open Folder.



You can enter the path to the folder containing the files you want to search through into the drop-down list at the top. If you have recently searched through files from this folder, you can select it from the drop-down

list. EditPad Pro remembers the last 16 folders. Another way to select the folder is to browse for it using the folder tree. The drop-down list and folder tree are automatically kept in sync.

Tick the option “recurse through subfolders” if you also want to search through files contained in subfolders of the folder you selected.

If you want to search through all files in the folder (and optionally its subfolders), select the “all files” option. If you want to search through only files of a certain type, you can select “files of type” and pick one of the file types from the drop-down list. The list contains all the file types you defined in Configure File Types and for which you ticked “show in file type selection lists”. This makes Find on Disk search through all files that match the file type’s file mask.

To search through files matching another list of extensions, select “file mask” and enter the list of extensions or file masks, separated by semicolons, into the “file mask” drop-down list. You can also select a recently used file mask from the drop-down list.

If you choose the “regular expression” option, then the regular expression must find a (partial) match in the name of a file for it to be included in the search. This regex is used to filter files by their names only. The text or regular expression that you want to use to search through the contents of the files needs to be specified on the search panel or in the search toolbar before you use the Find on Disk command.

Tick “list all matches in a side panel” to list all the search matches in all the files with one line of context in a side panel. This option makes Find on Disk work like Search|List All Matches, except that it searches through files on disk rather than files open in EditPad. If you use a regular expression that may find zero-length matches then the Find on Disk command does not add those zero-length matches to the Search Matches panel. If only zero-length matches are found in a certain file then that file will not be opened or added to the project either. If you want Find on Disk to find files with a regex that finds zero-length matches then you need to untick “list all matches in a side panel”.

Tick “create new project” to open files with search matches into a new, untitled project. This is the same as using Project|New Project before using Search|Find on Disk. This option has no effect if “open the files” and “add the files to the project” are both turned off.

Tick “open the files” to make EditPad Pro actually open the files in which search matches are found. If you turn this off, but turn on “add the files to the project”, then the files show up in the Files Panel as closed files. Since only managed projects can contain closed files, turning off “open the files” automatically changes the current or the new project that the files are added to into a managed project.

Tick “add the files to the project” to add the files in which search matches are found to the project. If you don’t work with projects or if you work with unmanaged projects then you should tick both “open the files” and “add the files to the project” or untick both these options depending on whether you want to open the files. If you turn on “open the files” and turn off “add the files to the project” then the files are opened as outside files, without adding them to the project. If you turn off “open the files” and turn on “add the files to the project” then the files are added to the project as closed files. Either way, turning one of these options on and turning the other option off forces the new project (if you ticked “create new project”) or the active project to become a managed project. Only managed projects can have files open that aren’t part of the project or have closed files be part of the project.

Tick “make the files read only” to open the files in read-only mode.

Recently Searched Through Folders

The Find on Disk item has a submenu that lists recently searched through folders. Each time you use Search|Find on Disk, the folder you search through is added to the top of this submenu. Though the menu shows only the folder's path, all the settings from the Find on Disk window are remembered by the menu. Select Maintain List in the submenu to see the settings used for each folder. When you select one of the remembered folders, EditPad Pro immediately searches through that folder, using the same settings for displaying search matches and opening files. If you have searched through the same folder more than once then the folder appears only once in the Reopen Folder menu with the most recently used settings.

The Find on Disk submenu does not remember the search term and other options set on the search toolbar or search panel. When you select a folder from the Find on Disk submenu, the present search term and search options are used. This way you can easily run different searches through the same folder.

“Remove Obsolete Folders” removes all folders that no longer exist from the Open Folders submenu. “Remove All Folders” clears the Find on Disk submenu.

Search|Show List of All Matches

When you use Search|List All Matches and when you use Search|Find on Disk with the option “list all matches in a side panel”, then EditPad Pro opens a side panel to which it adds all the search matches found by these commands. You can close this side panel by clicking the X on the panel's caption bar. If you've done this and you want to see the most recent list of search matches again without having to repeat the search, select Show List of All Matches in the Search menu. If you haven't run a search that fills the Search Matches panel then Show List of All Search Matches simply reveals an empty list of matches.

Search|Fold Lines

After using Prepare to Search or pressing Ctrl+F and entering a search term, click the Fold button on the Search toolbar or select the Fold Lines item in the Search menu to fold all lines in which the search term cannot be found underneath lines in which it can be found. The result is that only lines in which the search term can be found remain visible. The only exception is that the very first line in the file remains visible even if it doesn't have any search matches. All lines until the first line with a search match will be folded underneath the very first line in the file.

If there are no search matches at all then EditPad briefly flashes the Fold Lines button's icon to indicate failure. If there are a lot of search matches, folding them all might take a while. A progress meter will pop up to indicate EditPad's progress after a second or two.

Using Fold Lines together with Highlight All is a very quick way to get a condensed view of all the matches in the file. Unlike the match highlighting mode, Fold Lines is automatically disabled when you switch between files, or edit the search term. Click the Fold button again to refold the current file. Folding a file based on search matches deletes all automatic folding points, and all folding points that you created. When you disable the search match folding, new automatic folding points will appear. But folding points that you added with Fold|Fold are not restored. Also unlike match highlighting, which instantly updates itself, the folding is not updated as you insert or delete the search term while you edit the file.

After folding the lines with search matches, you can use **Fold|Copy Visible Lines** to copy the lines with search matches to the clipboard, and **Fold|Delete Folded Lines** to delete the lines without search matches. Since folding keeps the first line visible, however, the very first line in the file will be copied or remain behind, even if it doesn't have any search matches.

Search | Count Matches

After using **Prepare to Search** or pressing **Ctrl+F** and entering a search term, click the **Count** button on the Search toolbar or select the **Count Matches** item in the Search menu to count all search matches. **Count Matches** starts searching from the beginning just like **Find First** does. Then it continues searching like **Find Next** would, until the search fails. None of the search matches are selected. After the search, **Count Matches** pops up a message box to tell you how many matches it found.

If there are no search matches at all, EditPad briefly flashes the **Count** button's icon to indicate failure. If there are a lot of search matches, counting them all might take a while. A progress meter will pop up to indicate EditPad's progress after a second or two.

Search | Cut Matches

After using **Prepare to Search** or pressing **Ctrl+F** and entering a search term, click the **Cut** button on the Search toolbar or select the **Cut Matches** item in the Search menu to cut all search matches to the clipboard. All search matches are placed onto the clipboard delimited by line breaks. The search matches are then removed from the files in EditPad.

If there are no search matches at all, EditPad briefly flashes the **Cut** button's icon to indicate failure. If there are a lot of search matches, cutting them all might take a while. A progress meter will pop up to indicate EditPad's progress after a second or two.

Search | Copy Matches

After using **Prepare to Search** or pressing **Ctrl+F** and entering a search term, click the **Copy** button on the Search toolbar or select the **Copy Matches** item in the Search menu to copy all search matches to the clipboard. All search matches are placed onto the clipboard delimited by line breaks.

If there are no search matches at all, EditPad briefly flashes the **Copy** button's icon to indicate failure. If there are a lot of search matches, copying them all might take a while. A progress meter will pop up to indicate EditPad's progress after a second or two.

Search Options

EditPad offers 13 search options. By default all these have abbreviated buttons on the Search toolbar. You can toggle them with **Alt+letter** key combinations when the search toolbar or search panel has keyboard focus. You can also toggle them via the **Search|Search Options** menu item. Directly clicking this menu item shows a popup dialog with a checkbox for each option. If you only need to toggle one search option, it is quicker to use the submenu of **Search|Search Options** to toggle that option.

These options only affect searches done via the search toolbar or search panel. For instant searches, there is a separate reduced set of Instant Search Options.

Regular Expression (Regex)
 Free-Spacing (Free)
 Dot Matches Line Breaks (Dot)
 Case Sensitive (Case)
 Adapt Case (Adapt)
 Whole Words Only (Words)
 All Files (Files)
 All Projects (Projects)
 Closed Files (Closed)
 Selection Only (Block)
 Loop Automatically (Loop)
 Line by Line (Line)
 Inverted Line by Line (Invert)

Search Option: Regular Expression (Regex)

EditPad's search-and-replace module includes a full-featured regular expression engine. Its regular expression flavor is compatible with popular regular expression flavors such as those used by Perl, Java, and .NET. In the regular expression flavor comparison, EditPad's flavor is indicated as "JGsoft V2". To search using a regular expression, simply turn on the regular expression search option by clicking the Regex button on the Search toolbar.

When using capturing groups in a regular expression, you can insert the groups' contents in the replacement text using backreferences `\0`, `\1`, `\2`, etc. `\0` inserts the whole regex match, while `\1` inserts the text matched by the first capturing group, `\2` the second group, etc. EditPad Pro supports up to 99 backreferences. If you want to insert a backreference followed by a literal number into the replacement text, use two digits for the backreference. The replacement text `\15` will replace the regex match with the contents of the first backreference followed by the digit 5. You can also use dollar signs instead of backslashes. There's no difference between `\1` and `$1` in the replacement text in EditPad.

If you want to pad or control the case of backreferences, use the `%GROUP1%` syntax for match placeholders.

If you use `\r` and `\n` in your regex, remember that Windows text files use `\r\n` to terminate lines. EditPad's regular expression engine matches `\r` and `\n` to a single CR and a single LF. If you want differences in line breaks to be handled automatically, use the full search panel and press Shift+Enter to type in an actual line break into your regular expression. EditPad's regular expression engine will match each literal line break in your regular expression to either a CRLF pair, or a single LF, or a single CR.

The caret and dollar always match at the start and the end of a line, in addition to the start and the end of a file. EditPad Pro does not have an option to change this. To match the start or the end of the file only, use `\A` or `\z` instead. You can make the regex ignore whitespace with the "free-spacing" option. The behavior of the dot can be changed with the "dot matches line breaks" option.

Search Option: Free-Spacing (Free)

When searching for a regular expression, you can click the Free button on the Search toolbar to turn on the “free-spacing” search option. A free-spacing regular expression ignores whitespace and line breaks. It treats `#` as starting a comment.

Turning on this option is the same as putting the mode modifier `(?x)` at the start of your regular expression. If you do use the mode modifier `(?x)` or `(?-x)` then the mode modifier overrides EditPad’s search option.

Search Option: Dot Matches Line Breaks (Dot)

When searching for a regular expression, you can click the Dot button on the Search toolbar to turn on the “dot matches line breaks” search option. This option makes the dot match any character, including line break characters.

Turning on this option is the same as putting the mode modifier `(?s)` at the start of your regular expression. If you do use the mode modifier `(?s)` or `(?-s)` then the mode modifier overrides EditPad’s search option.

Search Option: Case Sensitive (Case)

If you click the Case button on the Search toolbar to turn on the “case sensitive” search option, then the difference between lowercase and uppercase letters is taken into account. E.g. “Dog” is considered to be different from “dog”. Otherwise, “dog”, “Dog”, “DOG”, and even “doG” are all the same.

Search Option: Adapt Case (Adapt)

When the “case sensitive” option is off, you can click the Adapt button on the Search toolbar to turn on the “adapt case” search option. When you do so, EditPad Pro adapts the casing of the replacement text to the casing of the search match to be replaced. EditPad Pro recognizes four casings: all uppercase, all lowercase, first letter capitalized, and first letter of each word capitalized. If EditPad Pro doesn’t recognize the casing, the replacement text is used exactly as you entered it.

E.g. when searching for “My DoG” and replacing with “My CaT”, then “my dog” is replaced with “my cat” (all lowercase), “MY DOG” with “MY CAT” (all uppercase), “My dog” with “My cat” (first letter capitalized) and “My Dog” with “My Cat” (first letter of each word capitalized). If EditPad Pro doesn’t recognize the casing, e.g. “my DOG”, then “My CaT” is substituted the way you entered it.

Search Option: Whole Words Only (Words)

Click the Words button on the Search toolbar to turn on the “whole words only” search option if only entire words should be valid search results. If this option is on, when looking for “cat”, EditPad does not consider the first three letters of “category” a valid search match. Otherwise, it does.

Search Option: All Files (Files)

Click the Dot button on the Search toolbar to turn on the “all files” search option to search through all open files in the current project. If a search command cannot find a match in the current file then it automatically continues with the next file until either a match is found, or all files have been searched through.

The Fold Lines search command always works on the current file only, regardless of the “all files” option. The Highlight All option is a global toggle that always highlights the current file, even when you switch files, until you turn it off, regardless of the “all files” option. All the other search commands respect the “all files” option as explained above.

Search Option: All Projects (Projects)

To search through all open files in all open projects, click the Projects button on the Search toolbar to turn on the “all projects” search option. If a search command cannot find a match in the current file then it automatically continues with the next file. When all files in the current project have been searched through, EditPad Pro continues with the first file in the next project. EditPad Pro continues searching until either a match is found, or all files in all projects have been searched through.

The Fold Lines search command always works on the current file only, regardless of the “all projects” option. The Highlight All option is a global toggle that always highlights the current file, even when you switch files, until you turn it off, regardless of the “all projects” option. All the other search commands respect the “all projects” option as explained above.

Search Option: Selection Only (Block)

Click the Block button on the Search toolbar to turn on the “selection only” search option to search only through the selected part of the current file.

If you turn on this option and invoke a search command such as Find First that processes only a single search match then the selection is turned into the search range. The search range highlight color is different from the selection highlight color. You can change these colors by customizing the color palette. “Editor: Selected text” is the second item in the list of individual colors. “Editor: Search range” is further down the list.

If a search match is found then that search match is selected. You can then edit the search match or any part of the file inside or outside the search range. As long as you don’t turn off the “selection only” search option, the search range remains highlighted.

If no search match is found then the text that was selected still becomes the search range. But it also remains selected. So you won’t see that it is the search range unless you change or clear the selection. The selection highlight goes on top of the search range highlight.

When you invoke the next search command, EditPad uses the existing search range if there is no selection or if the selection falls entirely inside the search range. If the selection includes text outside the search range, then EditPad uses the selection as the new search range.

In practice, this mechanism is very intuitive. When you search, EditPad selects the search match, which always falls inside the search range. So you can click Search|Find Next and Search|Find Previous without paying much attention to the search range. It will stay put. When you want to search through a different part of the file, simply select it, make sure “selection only” is on, and search. You don’t need to check whether there was a search range already. EditPad will search through the block of text that you selected.

Commands such as replace all that processes all search matches use the existing search range when there is no selection or the selection falls entirely within the existing search range. If the selection falls outside the search range, the search range is removed. If the search range is removed or there was no search range to begin with, then these commands perform their action (such as replacing all matches) within the selected text only, without turning that text into a search range. The text remains selected, regardless of whether any matches were found.

In the Search Preferences you can choose whether EditPad automatically turns the “selection only” on or off when you prepare to search. When this preference is on, preparing to search when there is a multi-line selection turns on “selection only”. Preparing to search when there is no selection or a selection within a single line turns off “selection only”. Preparing to search does not remove an existing search range. So you can turn the “selection only” option back on to keep using the existing search range in case that option was turned off when preparing your next search.

If you manually turn off “selection only” then any existing search range is removed immediately.

Search Option: Loop Automatically (Loop)

Click the Loop button on the Search toolbar to turn on the “loop automatically” search option if you want Find Next, Find Previous, Replace and Find Next and Replace and Find Previous to automatically restart searching from the start or end of the file when no further search matches can be found. If the “all files” option is on then EditPad restarts from the first file (find next) or the last file (find previous) in the project. If “all projects” is also on then EditPad restarts from the first file in the first project (find next) or the last file in the last project (find previous).

When EditPad finds a match after having looped (i.e. restarted from the start or end), then the glyph on the search command button that you clicked flashes briefly, showing a “looped” glyph.

Search Option: Closed Files (Closed)

When the option to search through all files or all projects is turned on, and one of the project’s you’re searching through is a managed project that contains closed files, you can click the Closed button on the Search toolbar to turn on the “closed files” option to search through the closed files in addition to searching through the open files.

Commands that select found matches such as Find Next and commands that modify files such as Replace All automatically open closed files in which matches are found so that the found match can be selected or the file can be modified. Closed files in which no search matches are found are not opened.

Search Option: Line by Line (Line)

Click the Line button on the Search toolbar to turn on the “line by line” search option to search through each line separately, and to treat the entire line as the search match if part of it matches. This means that when this option is on, a search match can never span across multiple lines. Commands like Replace All and Copy Matches will replace or copy entire lines, even when the search term matches only part of the line.

Search Option: Inverted Line by Line (Invert)

When using the Line by Line search option, you can turn on Inverted Line by Line to match all lines in which the search term cannot be found. Click the Invert button on the Search toolbar to toggle this option.

Search | History

Click the History button on the Search panel to switch between the currently displayed search term and search options, and the ones you used before. This way you can easily work with two search actions. If you need to do a search-and-replace in several blocks in a file, for example, and you want to find the block by searching for another word, first search for the word and then do the search-and-replace in the first block. Then click the History button to instantly get back the search term. Find the second block, and click the History button again to get the search-and-replace search term and options.

EditPad’s search function remembers up to 16 search settings. You can access them via the submenu of the History button. Pick a search term from the menu to use it again. All the Search Options are also set the way they were when you last used the search term you just picked from the history.

If you only want to recall the search text or the replacement text, without changing any of the search options, right-click on the search box or replace box. The right-click menu lists the last 16 search texts or replacement texts.

Search | Favorites

If you regularly use a particular search term, you should add it to your search favorites. To do so, select Search|Favorites|Add Search Term in the menu. Or, click on the downward pointing arrow next to the Favorites button on the search panel, and select Add Search Term in the drop-down menu. The search text, replacement text, and all search options are stored in the favorites. To make it easy to differentiate between search terms, you’ll be asked for a label when adding a favorite search term.

If you click directly on the Favorites button rather than the arrow next to it, the Organize Favorites screen appears. Select a search term and click on the Use button to set the search text, replacement text and search options to those stored in the favorites item. Click the Remove button to delete a favorite.

With the New Folder and Rename Folder, you can organize your favorite search terms into folders. Organizing favorites into folders makes it easier to retrieve the search term you want. To move favorites into a folder, simply drag-and-drop them with the mouse.

Search | RegexBuddy

Use regular expressions with EditPad's search and replace to automate many editing task that would be tedious to do manually. The regex syntax is quite terse, making complex regular expressions sometimes difficult to interpret.

To alleviate this complexity, we have developed a product called RegexBuddy. RegexBuddy makes it very easy to write and edit regular expressions. Use RegexBuddy's detailed and descriptive regex tree, and the easy-to-grasp regex building blocks instead of or in combination with the regex syntax to define your search patterns.

Click the RegexBuddy button on the Search panel or select the RegexBuddy item in the Search menu to start RegexBuddy. The search and replace texts you entered in EditPad are automatically sent to RegexBuddy for editing. When you have prepared the new regular expression and replacement text, click the Send button in RegexBuddy. RegexBuddy then closes and sends the new regex and replacement to EditPad. If you change your mind, close RegexBuddy without clicking the Send button.

You can also use RegexBuddy to test your search patterns in a safe and intuitive sandbox. Very handy is the ability to collect your own libraries of regular expressions. Or use RegexBuddy's standard library of regular expressions for common search patterns.

You can use RegexBuddy in combination with EditPad, as well as with any other application you use regular expressions with. If you are a programmer, RegexBuddy has full support for a variety of programming languages and regex libraries.

See <https://www.regexbuddy.com> for more information about RegexBuddy.

Please note that RegexBuddy is a separate product. If you want to use RegexBuddy with EditPad, you will need to purchase both products.

Search | RegexMagic

Use regular expressions with EditPad's search and replace to automate many editing task that would be tedious to do manually. The regex syntax is quite terse, making complex regular expressions sometimes difficult to interpret.

To make it easy to create regular expressions, we have developed RegexMagic. Instead of dealing with the cryptic regular expression syntax, use RegexMagic's powerful patterns for matching characters, text, numbers, dates, times, email addresses, URLs, card numbers, etc. RegexMagic can detect these patterns when you mark your sample text. You can combine multiple patterns to match exactly what you want.

Click the RegexMagic button on the Search panel or select the RegexMagic item in the Search menu to start RegexMagic. When you have generated the new regular expression and replacement text, click the Send button in RegexMagic. RegexMagic will then close and send the new regular expression to EditPad. If you change your mind, close RegexMagic without clicking the Send button.

Use RegexMagic in combination with EditPad, as well as any other application you use regular expressions with. RegexMagic supports all popular regular expression flavors. If you are a programmer, RegexMagic has full support for a variety of programming languages and regex libraries.

See <https://www.regexmagic.com> for more information about RegexMagic.

Please note that RegexMagic is a separate product. If you want to use RegexMagic with EditPad, you will need to purchase both products.

Match Placeholders

Match placeholders can be used to insert search matches or match counts in the search text or replacement text.

Placeholder	Meaning and Examples	Availability
%LINE%	The line the search match was found on.	Replacement text.
%LINEN%	The number of the line the match was found on.	Replacement text.
%MATCH%	The search match.	Replacement text.
%MATCHN%	The number of the search match. Counts the number of search matches already found. %MATCHN% will be one for the first replacement.	Replacement text.
%GROUP1%, %GROUP2%, etc.	%GROUP1% is a backreference to a capturing group, equivalent to <code>\1</code> in a regular expression or <code>\1</code> or <code>\$1</code> in the replacement text. The advantage of %GROUP1% is that you can use the padding and case conversion specifiers listed below.	Search term and replacement text, but only when using regular expressions with numbered capturing groups.
%GROUPNAME%, %GROUP anothername%, etc.	%GROUPNAME% is a backreference to a named capturing group, equivalent to <code>\k'NAME'</code> or <code>(?P=NAME)</code> in a regular expression and to <code>\${NAME}</code> or <code>\g<NAME></code> in the replacement text. The NAME part of the placeholder is case sensitive. The advantage of %GROUPNAME% is that you can use the padding and case conversion specifiers listed below.	Search term and replacement text, but only when using regular expressions with named capturing groups.

Padding

You can add additional specifiers to all of the above placeholders. You can pad the placeholder's value to a certain length, and control the casing of any letters in its value. The specifiers must appear before the second % sign in the placeholder, separated from the placeholder's name with a colon. E.g. %MATCH:6L% inserts the match padded with spaces at the left to a length of 6 characters. You can add both padding and case placeholders. %MATCH:U:6L% inserts the padded match converted to uppercase.

Padding specifiers start with a number indicating the length, followed by a letter indicating the padding style. The length is the number of characters the placeholder should insert into the regular expression or replacement text. If the length of the placeholder's value exceeds the requested length, it will be inserted unchanged. It won't be truncated to fit the length. If the value is shorter, it will be padded according to the padding style you specified.

The L or “left” padding style puts spaces before the placeholder’s value. This style is useful for padding numbers or currency values to line them up in columns. The R or “right” padding style puts spaces after the placeholder’s value. This style is useful for padding words or text to line them up in columns. The C or “center” padding style puts the same number of spaces before and after the placeholder’s value. If an odd number of spaces is needed for padding, one more space will be placed before the value than after it.

The Z or “zero” padding style puts zeros before the placeholder’s value. This style is useful for padding sequence numbers.

Case Conversion

Case conversion specifiers consist of a letter only. The specifier letters are case insensitive. Both “U” and “u” convert the placeholder’s value to uppercase. L converts it to lowercase, I to initial caps (first letter in each word capitalized) and F to first cap only (first character in the value capitalized). E.g. %MATCH:I% inserts the match formatted as a title.

Arithmetic

Arithmetic specifiers perform basic arithmetic on the placeholder’s value. This works with any placeholder that, at least prior to padding, represents an integer number. If the placeholder does not represent a number, the arithmetic specifiers are ignored. For placeholders like %MATCH% that can sometimes be numeric and other times not, the arithmetic specifiers are used whenever the placeholder happens to represent an integer.

An arithmetic specifier consists of one or more operator and integer pairs. The operator can be +, -, *, or / to signify addition, subtraction, multiplication, or integer division. It must be followed by a positive integer. Arithmetic specifiers are evaluated strictly from left to right. When %MATCHN% evaluates to 2, %MATCHN:+1*2% evaluates to 6 because $2+1=3$ and $3*2=6$. Multiplication and division do not take precedence over addition and subtraction. Integer division drops the fractional part of the division’s result, so %MATCHN:/3% evaluates to 0 for the first two matches.

Path Placeholders in Search Terms

You can use path placeholders in the search term and the replacement string to search for or replace with part or all of the path and the file name of the file that is being searched through. When searching or replacing through all open files, the values for these placeholders automatically change as the search progresses to the next file. In the examples below, the file being edited is C:\data\files\web\log\foo.bar.txt. If the current file is untitled, all the placeholders below will be replaced with nothing.

If the file was opened via FTP, use %FTPFILE% to specify the full path to the file on the FTP server. %FTPSEVER% is the domain name of the server, and %FTPURL%: the full ftp:// URL to the file. You can prefix the placeholders listed below with FTP to get various parts of the path to the file on the server (excluding the domain name). %FTPDRIVE%, though not an error, will always be blank. %FTPPATH% has a leading forward slash while %FTPFOLDER% does not.

If the file is part of a project, you can use %PROJECTFILE% to reference the full path to the .epp file that the project was saved into. You can prefix the placeholders listed below with PROJECT to get the various parts of the path to the .epp file.

Placeholder	Meaning	Example
%FILE%	The entire path plus filename to the file	C:\data\files\web\log\foo.bar.txt
%FILENAME%	The file name without path	foo.bar.txt
%FILENAMENOEXT%	The file name without the extension	foo.bar
%FILENAMENODOT%	The file name cut off at the first dot	foo
%FILEEXT%	The extension of the file name without the dot	txt
%FILELONGEXT%	Everything in the file name after the first dot	bar.txt
%PATH%	The full path without trailing delimiter to the file	C:\data\files\web\log
%DRIVE%	The drive the file is on, without trailing delimiter	C: for drive letter paths; \\server for UNC paths
%FOLDER%	The full path without the drive and without leading or trailing delimiters	data\files\web\log
%FOLDER1%	First folder in the path	data
%FOLDER2%	Second folder in the path	files
(...etc...)		
%FOLDER99%	99th folder in the path.	<i>(nothing)</i>
%FOLDER<1%	Last folder in the path	log
%FOLDER<2%	Second folder from the end in the path	web
(...etc...)		
%FOLDER<99%	99th folder from the end in the path.	<i>(nothing)</i>
%PATH1%	First folder in the path, without delimiters	data
%PATH2%	First two folders in the path, without leading or trailing delimiters	data\files
(...etc...)		
%PATH99%	First 99 folders in the path, without leading or trailing delimiters	data\files\web\log
%PATH<1%	Last folder in the path, without delimiters	log
%PATH<2%	Last two folders in the path, without leading or trailing delimiters	web\log
(...etc...)		
%PATH<99%	Last 99 folders in the path, without leading or trailing delimiters	data\files\web\log
%PATH-1%	Path without the drive or the first	files\web\log

	folder	
%PATH-2%	Path without the drive or the first two folders	web\log
(...etc...)		
%PATH-99%	Path without the drive or the first 99 folders.	(<i>nothing</i>)
%PATH<-1%	Path without the drive or the last folder	data\files\web
%PATH<-2%	Path without the drive or the last two folders	data\files
(...etc...)		
%PATH<-99%	Path without the drive or the last 99 folders.	(<i>nothing</i>)

Combining Path Placeholders

You can string several path placeholders together to form a complete path. If you have a file `c:\data\test\file.txt` then `d:\%FOLDER2%\%FILENAME%` will be substituted with `d:\test\file.txt`. However, if the original file is `c:\more\file.txt` then the same path will be replaced with `d:\file.txt` because `%FOLDER2%` is empty. The result is an invalid path.

The solution is to use combined path placeholders, like this: `d:\%FOLDER2\FILENAME%`. The first example will be substituted with `d:\test\file.txt` just the same, and the second will be substituted with `d:\file.txt`, a valid path. You can combine any number of path placeholders into a single path placeholder, separating them either with backslashes (`\`) or forward slashes (`/`). Place the entire combined placeholder between two percentage signs.

A slash between two placeholders inside the combined placeholder is only added if there is actually something to separate inside the placeholder. Slashes between two placeholders will never cause a slash to be put at the start or the end of the entire resulting path. In the above example, the backslash inside the placeholder is only included in the final path if `%FOLDER2%` is not empty.

A slash right after the first percentage sign makes sure that the resulting path starts with a slash. If the entire resulting path is empty, or if it already starts with a slash, then the slash is not added.

A slash right before the final percentage sign makes sure that the resulting path ends with a slash. If the entire resulting path is empty, or if it already ends with a slash, then the slash is not added.

Mixing backslashes and forward slashes is not permitted. Using a forward slash inside a combined placeholder, will convert all backslashes in the resulting path to forward slashes. This is useful when creating URLs based on file names, as URLs use forward slashes, but Windows file names use backslashes.

Example: If the original path is `c:\data\files\web\log\foo.bar.txt`

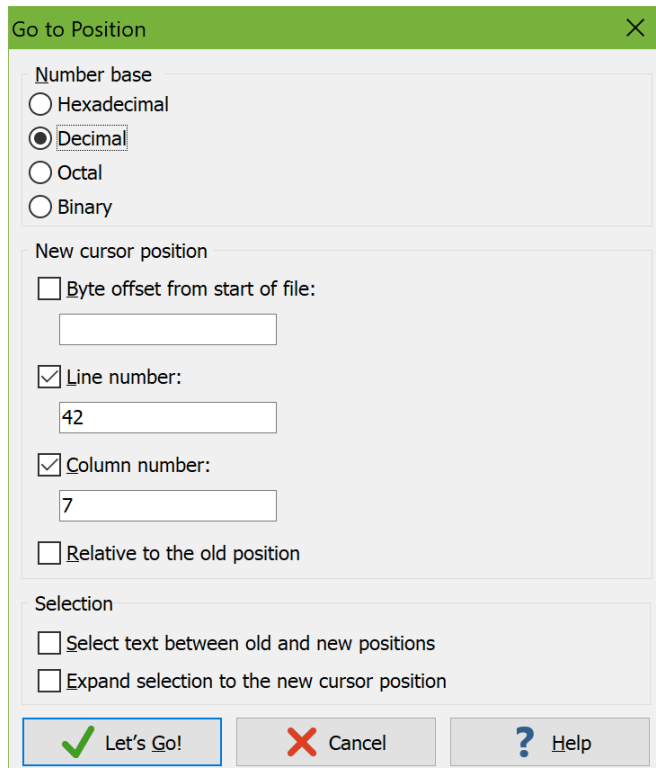
Placeholders	Resulting path
%\FOLDER1\%	\data\

%\FOLDER5\%	(nothing)
%PATH-2\FILENAME%	web\log\foo.bar.txt
%PATH-2/FILENAME%	web/log/foo.bar.txt
%PATH-4\FILENAME%	foo.bar.txt
%DRIVE\PATH-2\FILENAME%	c:\web\log\foo.bar.txt
%DRIVE\PATH-4\FILENAME%	c:\foo.bar.txt
%\FOLDER1\FOLDER4\%	\data\log\
%\FOLDER1\FOLDER5\%	\data\

5. Go Menu

Go | Go to Position

The Go to Position in the Go menu lets you move the text cursor to a specific position in the file.



Go to Position [X]

Number base

☐ Hexadecimal

☒ **Decimal**

☐ Octal

☐ Binary

New cursor position

☐ Byte offset from start of file:

☒ **Line number:**

☒ **Column number:**

☐ Relative to the old position

Selection

☐ Select text between old and new positions

☐ Expand selection to the new cursor position

The “byte offset” option counts bytes from the start of the file like the address column does in hexadecimal mode. A negative byte offset counts bytes from the end of the file. You can also go to a byte offset in text mode. In text mode, if the file has a byte order marker, then the byte offset excludes the byte order marker. So byte offset 0 is before the first character in the file.

In text mode, you can move the cursor to a specific line number, column number, or both. You can enter a negative line number to move to a line relative to the end of the file. -1 is the last line, -2 the penultimate line, and so on. You can enter a negative column number to move to a column relative to the end of the line.

If you tick “relative to the old position”, then a positive byte offset, line number, or column number moves the text cursor that many bytes, lines, or columns downward in the file. Negative numbers move the cursor upward.

Tick “select text between old and new positions” to select the text between the position of the text cursor before you used the Go to Position command and the position you had that command move the cursor to. Any other text that was selected before is unselected.

The "expand selection to the new cursor position" option is only enabled if there was a selection before you used the Go to Position command. If you tick it, Go to Position selects the text between the old selection and the new position of the cursor. The text that was previously selected remains selected.

Go | Go to Matching Bracket

On the Brackets page in the file type configuration, you can make EditPad Pro highlight matching brackets for certain file types. When the text cursor is next to a character that is treated as a bracket by the syntax coloring scheme, then both that bracket and its matching counterpart are highlighted. In that situation, you can select Go to Matching Bracket from the Go menu to move the text cursor to the matching bracket. Most schemes match up characters like parentheses and braces as brackets. Some can also match up “begin” and “end” keywords or even markup tags.

Go | Go to Unmatched Bracket

On the Brackets page in the file type configuration, you can make EditPad Pro highlight matching brackets for certain file types. Most schemes match up characters like parentheses and braces as brackets. Some can also match up “begin” and “end” keywords or even markup tags.

This can help you to make sure all brackets are correctly paired and nested. Select Go to Unmatched Bracket in the Go menu to move the text cursor to the nearest bracket that does not have a matching bracket. This bracket is then highlighted as an unmatched bracket.

If there are no unmatched brackets in the file, the cursor is not moved. If the cursor was next to or between a pair of highlighted brackets, that highlighting is removed until you move the cursor again.

Go | Back in Editing Positions

Select Back in Editing Positions in the Go menu to move the text cursor to the position in the current file where you last inserted or deleted some text. This makes it very comfortable to scroll through a file to look something up while editing. You can always jump back to the editing position where you left off to continue editing.

If you use the Back in Editing Position command a second time right after the first time, the cursor is moved to the editing position before the previous one. If you use it a third time, the cursor is moved to the editing position before that. You can use the command up to 16 times to revisit the last 16 spots in the file where you made changes.

If you go back too far, use the Go | Forward in Editing Positions command to move forward through the most recent editing positions.

EditPad remembers the last 16 editing positions for all files that you have open. The Back in Editing Positions command never switches between files. You can use Go | Back in Edited Files and Go | Forward in Edited Files to switch between recently edited files.

Go|Forward in Editing Position

If you have used Go|Back in Editing Positions to go back to a position where you previously made a change in the active file, then you can use the Forward in Editing Positions command in the Go menu to move forward in the list of recent editing positions.

Go|Next Edit

On the Editor page in the Preferences you can turn on an option to “indicate lines that were edited since the file was last opened or saved with highlighting in the left margin”. When this is on, you can use the Go|Next Edit menu item to move the text cursor to the line further down in the file that is indicated as having been edited since the file was last opened or saved. If there is no such line, the cursor is moved to the last line. If the cursor is already on such a line, then Go|Next Edit skips over all such adjacent lines, moving the cursor to the next edited line after the next unedited line. Basically, it jumps from one block of edited lines to the first line in the next block of edited lines.

Go|Previous Edit

On the Editor page in the Preferences you can turn on an option to “indicate lines that were edited since the file was last opened or saved with highlighting in the left margin”. When this is on, you can use the Go|Previous Edit menu item to move the text cursor to the line further up in the file that is indicated as having been edited since the file was last opened or saved. If there is no such line, the cursor is moved to the first line. If the cursor is already on such a line, then Go|Previous Edit skips over all such adjacent lines, moving the cursor to the previous edited line after the previous unedited line. Basically, it jumps from one block of edited lines to the last line in the previous block of edited lines.

Go|Next Unsaved Edit

On the Editor page in the Preferences you can turn on an option to “indicate lines that were edited since the file was last opened or saved with highlighting in the left margin”. If you open a file and start editing it, edited lines are indicated as edited since last saving the file (as those lines have unsaved changes). If you then save the file, all edited lines have their indicators changed to edited since last opening the file (as those lines no longer have unsaved changes). Lines you edit after saving are then again indicated as edited since last saving.

The Go|Next Unsaved Edit command only cares about lines indicated as edited since last saving the file. It moves the cursor to the next line indicated as such. If there is no such line, the cursor is moved to the last line. If the cursor is already on such a line, then Go|Next Unsaved Edit skips over all such adjacent lines, jumping from one block of lines edited since saving to the first line in the next block of lines edited since saving.

Go|Previous Unsaved Edit

On the Editor page in the Preferences you can turn on an option to “indicate lines that were edited since the file was last opened or saved with highlighting in the left margin”. If you open a file and start editing it, edited

lines are indicated as edited since last saving the file (as those lines have unsaved changes). If you then save the file, all edited lines have their indicators changed to edited since last opening the file (as those lines no longer have unsaved changes). Lines you edit after saving are then again indicated as edited since last saving.

The Go|Previous Unsaved Edit command only cares about lines indicated as edited since last saving the file. It moves the cursor to the previous line indicated as such. If there is no such line, the cursor is moved to the first line. If the cursor is already on such a line, then Go|Previous Unsaved Edit skips over all such adjacent lines, jumping from one block of lines edited since saving to the first line in the previous block of lines edited since saving.

Go|Back in Edited Files

Select Back in Edited Files in the Go menu to activate the last file that you made a change to. If that file is already active, the file you last made a change to other than the active file is activated. When you have many files open, you can use this command to quickly go back to the file you were editing after you've looked at other files. E.g. if you want to copy and paste something from another file into the file you're editing, switch to the other file, copy the text you want to paste, and then use Go|Back in Edited Files to instantly go back to the spot where you want to paste.

If you use the Previously Edited File command a second time, EditPad activates the second most recently edited file. EditPad remembers up to 16 files.

You can use the submenu of the Back in Edited Files command to directly select the file you want to go back to. If you have already gone back to a previously edited file then this file has a check mark in the submenu. Then you can use the same submenu to go forward in edited files too.

Essentially, the Back in Edited Files and Forward in Edited Files commands work like the Back and Forward buttons in a web browser. But they only remember files that you've actually edited, not every file you've viewed. If you want to switch between recently viewed files, regardless of whether you've edited them, use the Ctrl+Tab key combination with the "use most recent order when switching tabs with Ctrl+Tab" preference turned on.

Go|Forward in Edited Files

If you have used Go|Back in Edited Files to go back to a previously edited file, then you can use the Forward in Edited Files command in the Go menu to move forward in the list of recently edited files.

Go|Forward in Edited Files does not have a submenu. The Go|Back in Edited Files submenu lists all recently edited files. If you have already gone back to a previously edited file then this file has a check mark in the submenu. Selecting a file above the one with the check mark moves you forward.

Go|Next File

The Go|Next File menu item activates the next file tab in the current project. If the last file in the project is active, then the first file is activated.

If you record this command as part of a macro and the macro plays back the Go|Next File command when the last file is already active, then macro playback is stopped. The first file is not activated. This makes it possible to record a macro that operates on all files by recording Go|Next File as the last command in the macro. Then you can play back the macro and repeat it any number of times, knowing that it will stop at the last file.

The Next File and Previous File commands always switch between files in the order their tabs have. If you want to switch between recently edited files, use the Go|Back in Edited Files and Go|Forward in Edited Files commands. If you want to switch between recently viewed files, regardless of whether you've edited them, use the Ctrl+Tab key combination with the “use most recent order when switching tabs with Ctrl+Tab” preference turned on. If you quickly want to go to a specific file, use the Files Panel.

Go|Previous File

The Go|Previous File menu item activates the previous file tab in the current project. If the first file in the project is active, then the last file is activated. During macro playback, if the first file is active, macro playback is stopped without switching between files. See Go|Next File for details.

Go|Sort File Tabs Alphanumerically

Pick Sort File Tabs Alphanumerically from the Go menu to sort the tabs of all files in the current project alphanumerically by their file names. Alphanumeric order is like alphabetic order. But if your file names contain numbers, those numbers are interpreted as a whole. So Chapter2.txt comes before Chapter10.txt because 2 comes before 10. This sort order is also known as “natural sort”. Windows Explorer uses the same method for sorting files and folders by their names.

Note that the file tabs are sorted only at the moment you click on Sort File Tabs Alphanumerically. If you open more files, the alphanumerical order is not maintained. Tabs for newly opened files are either placed to the immediate right-hand side of the tab that was active when you opened the additional files, or after the last file tab. You can configure this in the Tabs Preferences.

You can also rearrange the tabs in any order you want by dragging them with the mouse. To drag a tab, click on it, hold the mouse button down, and move the mouse to the left or right. The tab will follow the mouse pointer until you release the mouse button. If you drag the tab to the left edge of the leftmost tab, and there are more tabs, presently invisible, to the left of that tab, then all the tabs will scroll to the right, allowing you to move the tab you are dragging further to the left. Similarly, you can drag the tab to the right edge of the rightmost tab to make the tabs scroll to the left.

You can make Windows Explorer sort files alphabetically rather than alphanumerically by editing the registry. First select HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER depending on whether you want to change this for all accounts on your computer or only your own account. Then navigate to the key SOFTWARE\Microsoft\Windows\Currentversion\Policies\Explorer. Add the entry NoStrCmpLogical as a DWORD value set to 1. You may need to restart your computer for this to take effect. The Go|Sort File Tabs Alphanumerically command in EditPad respects this setting. It will sort files alphabetically rather than alphanumerically when Windows Explorer does so. The label on the menu item does not change to indicate this.

Go|Next Project

The Go|Next Project item activates the next project tab. If the last project is active, then the first project will be activated. During macro playback, if the last project is active, macro playback is stopped without switching between projects.

If you like to use the mouse, you can quickly switch between projects by clicking on their tabs, or by double-clicking on the project's node in the Files Panel.

Go|Previous Project

The Go|Previous Project menu item activates the previous project tab. If the first project is active, then the last project is activated. During macro playback, if the first project is active, macro playback is stopped without switching between projects.

If you like to use the mouse, you can quickly switch between projects by clicking on their tabs, or by double-clicking on the project's node in the Files Panel.

Go|Sort Project Tabs Alphanumerically

Pick Sort Project Tabs Alphanumerically from the Go menu to sort the tabs of all projects alphanumerically by their names. Alphanumeric order is like alphabetic order. But if your file names contain numbers, those numbers are interpreted as a whole. So Project2.epp comes before Project10.epp because 2 comes before 10. This sort order is also known as “natural sort”. Windows Explorer uses the same method for sorting files and folders by their names.

Note that the project tabs are sorted only at the moment you click on Sort Project Tabs Alphanumerically. If you open more projects, the alphanumerical order is not maintained. Tabs for newly opened projects are always placed to the immediate right-hand side of the tab that was active when you opened the additional files.

You can also rearrange the tabs in any order you want by dragging them with the mouse. To drag a tab, click on it, hold the mouse button down, and move the mouse to the left or right. The tab will follow the mouse pointer until you release the mouse button. If you drag the tab to the left edge of the leftmost tab, and there are more tabs, presently invisible, to the left of that tab, then all the tabs will scroll to the right, allowing you to move the tab you are dragging further to the left. Similarly, you can drag the tab to the right edge of the rightmost tab to make the tabs scroll to the left.

As explained for Go|Sort File Tabs Alphanumerically, Windows Explorer can be set to sort alphabetically rather than alphanumerically. If you did this then Go|Sort Project Tabs Alphanumerically also uses alphabetic sort. The menu item's label does not change to reflect this.

6. Block Menu

Block|Duplicate

Select Duplicate from the Block menu to insert a copy of the selected text at the current position of the text cursor. The newly inserted copy will become the selected text.

This command is intended to be used with persistent selections. You can move the text cursor without clearing the selection only when selections are persistent.

Selecting text, moving the text cursor, and invoking Block|Duplicate has the same effect as selecting text, invoking Edit|Copy, moving the text cursor, and invoking Edit|Paste. The only difference is that Block|Duplicate does not use the clipboard, so the contents of the clipboard are preserved.

The option “paste whole lines when lines are copied as a whole” affects how text is duplicated when you have selected complete lines. Selecting a complete line means to select everything from the start of the line to the end of the line, including the line break at the end of the line. Selecting multiple lines completely means selecting everything from the start of the first line in the block until the end of the last line in the block, including the line break at the end of the last line. When the option “paste whole lines when lines are copied as a whole” is on, whole lines are always duplicated as if the cursor were at the start of the line when you invoke Block|Duplicate. Thus, whole lines are always duplicated as a whole before the line that the cursor is on when duplicating. This makes it easy to duplicate blocks of lines without worrying about the horizontal position of the text cursor. If this option is off, the selected lines are duplicated at the exact spot the text cursor is at, even when whole lines are selected.

If you use the menu item or toolbar button to invoke this command then it is invoked on EditPad’s main editor, where you edit files. If you press the shortcut key on the keyboard, it is invoked on whichever editor is showing the text cursor (vertical blinking bar), whether that’s the main editor, the search box, or the replace box.

Block|Move

Select Move from the Block menu to delete the selected text and insert it again at the current position of the text cursor. Block|Move only becomes enabled when you have turned on Block|Persistent Selections. You can move the text cursor without clearing the selection only when selections are persistent.

Selecting text, moving the text cursor, and invoking Block|Move has the same effect as selecting text, invoking Edit|Cut, moving the text cursor, and invoking Edit|Paste. The only difference is that Block|Move does not use the clipboard, so the contents of the clipboard are preserved.

The option “paste whole lines when lines are copied as a whole” affects how text is moved when you have selected complete lines. Selecting a complete line means to select everything from the start of the line to the end of the line, including the line break at the end of the line. Selecting multiple lines completely means selecting everything from the start of the first line in the block until the end of the last line in the block, including the line break at the end of the last line. When the option “paste whole lines when lines are copied as a whole” is on, whole lines are always moved as if the cursor were at the start of the line when you invoke Block|Move. Thus, whole lines are always moved as a whole before the line that the cursor is on when

moving. This makes it easy to move blocks of lines without worrying about the horizontal position of the text cursor. If this option is off, the selected lines are moved to the exact spot the text cursor is at, even when whole lines are selected.

If you use the menu item or toolbar button to invoke this command then it is invoked on EditPad's main editor, where you edit files. If you press the shortcut key on the keyboard, it is invoked on whichever editor is showing the text cursor (vertical blinking bar), whether that's the main editor, the search box, or the replace box.

Block | Swap Selections

If you have used View | Split Editor to split EditPad Pro's editor in two, you can select a different part of the active file in each of the split editors. You can then use the Swap Selections item in the Block menu to swap the two selected blocks. If your file contains the sentence "Harry met Sally" and you select "Harry" in one view and "Sally" in the other view, then Block | Swap Selections changes the sentence into "Sally met Harry".

Block | Indent

Select Indent from the Block menu to insert a certain number of spaces at the start of each line in the selected portion of the current file. How many spaces are inserted depends on the "block indent" setting for the type of file you are editing. You can change this setting on the Editor page in the file type configuration. If the "block indent" setting is an integer multiple of the "tab size" setting, and the option "tab inserts spaces" is off, Block | Indent uses tab characters instead of spaces to indent the selection.

If the selection spans more than one line, you can also indent it by pressing Tab on the keyboard. The difference between using Block | Indent and the Tab key is that Block | Indent uses the "block indent" setting while Tab uses the "tab size" setting to determine the amount of indent. If "tab inserts spaces" is off, pressing tab will indent the selection by inserting exactly one tab character at the beginning of each paragraph.

Note that if word wrap is on, only the first line of each paragraph is indented. Wrapped lines are only indented along with the first line if you've turned on Options | Indent Wrapped Lines.

If a rectangular block is selected, Block | Indent works differently. Instead of indenting the selected lines by inserting spaces or tabs at the start of the line, it inserts the spaces immediately before the first column in the selection, effectively shifting the block a number of spaces to the right.

Block | Outdent

Select Outdent from the Block menu to outdent or unindent each line in the selected portion of the current file. EditPad does this by deleting a certain number of spaces from the start of each line. How many spaces are deleted depends on the "block indent" setting for the type of file you are editing. You can change this setting on the Editor page in the file type configuration. If there are not that many spaces at the beginning of a certain line, all the spaces at the beginning of that line are deleted. But the other lines will still be outdented by the specified amount.

If the lines you are outdenting are indented with tabs and the “block indent” setting is smaller than the tab size, then those tabs are replaced with as many spaces as the tab size minus the indentation size so that those lines are outdented by the same number of columns as lines indented by spaces.

If the selection spans more than one line, you can also outdent it by pressing Shift+Tab on the keyboard. The difference between using Block|Outdent and the Tab key is that Block|Outdent uses the “block indent” setting while Tab uses the “tab size” setting to determine the amount.

If a rectangular block is selected, Block|Outdent works differently. Instead of deleting spaces or tabs at the start of the line, it will delete spaces and tabs immediately to the left of the first column in the selection, effectively shifting the block a number of spaces to the left. Spaces inside the selected block are not deleted.

Block|Move Lines & Columns

The Lines and Columns submenu of the Block menu lists various commands for shifting blocks of text around. It also has Prefix and Suffix commands for inserting something at the start or end of each selected line.

When you’ve made a linear selection (a selection that follows the flow of the text), you can only move lines. Lines are moved entirely. If the first and/or last line in the selection are only partially selected, the selection first expands itself to include those lines entirely. When moving the block one line up, the block and the line above it swap places. After the action, the unselected line that was above the block will be below it. Moving the block one line down does the opposite.

When you’ve made a rectangular selection, only the selected columns are moved. When moving the rectangular block one line up, the columns spanning the width of the selection in the line above the selection are removed. Then the selected block is moved up one line, filling up the space created by removing part of the line that was above the selection. The removed characters are then reinserted in the empty space created on the line that used to be the last line in the selection. If the line above the block did not have any characters to remove, spaces are inserted into the last line instead. The original block remains selected, now positioned one line higher in the file. Moving a rectangular block one line down does the opposite.

Rectangular selections can also be shifted to the left or the right. When shifting a block to the left, the character immediately to the left of the block on each line partially selected by the block is removed. This character is then reinserted on the same line immediately to the right of the block. The original block remains selected, now positioned one column closer to the left. When shifting a block to the right, the character immediately to the right of the block is removed, and reinserted to the left of the block. If there is no character to the right of the block on a particular line, a space is inserted to the left of the block instead. You can shift a block as far out to the right as you want.

If no text is selected at all, you can still use the commands to move lines up or down. Then, the line containing the text cursor is moved.

Block|Persistent Selections

By default, selections in EditPad Pro behave like in most other Windows applications. When you move the text cursor, by pressing a key on the keyboard or clicking somewhere with the mouse, the selection

disappears. When you enter some text or paste from the clipboard, the selected text is replaced by the text you entered or pasted. Pressing the Backspace or Delete key deletes the selection.

If you turn on Persistent Selections in the Block menu or via the corresponding toolbar button then moving the text cursor does not cause the selection to disappear. Also, entering or pasting text does not delete the selected text. The typed or pasted text is inserted at the position of the text cursor, whether that is inside or outside of the selection. If it is inside the selection, the selection is expanded to include both the text originally selected and the newly entered text.

Pressing the Backspace or Delete key does not delete a persistent selection. It only deletes one character to the left or right of the cursor. Press Ctrl+Delete to delete a persistent selection.

A key benefit of persistent selections is that you can copy and move text within a file using Block|Duplicate and Block|Move. Persistent selections are also easily expanded with Block|Expand Selection.

Block|Rectangular Selections

There are several ways to select text in EditPad using the keyboard or the mouse. Whichever way you choose, it involves marking a starting and an ending position for the selection. All the text between the starting and the ending position becomes selected, following the flow of the text like you would follow it when reading it out loud. This is how almost all Windows programs select text, and is appropriate in most situations.

But in some situations, such as when editing text files with information organized in tables, it is more useful to make a rectangular selection, also called a column selection. A rectangular selection is much like a selection of cells in a spreadsheet application, except that in EditPad you select characters instead of cells.

Before you can make rectangular selections, a few conditions must be met. First, you need to use a monospaced (fixed width) font such as Consolas. You can change the font via Options|Font. You can make the font monospaced with the monospaced left-to-right text layout. Characters must be properly aligned into columns to be able to make rectangular selections. That only happens when the font is monospaced.

Second, word wrap must be off. You can turn it off using Options|Word Wrap. When long lines are wrapped, editing a long line causes the text to be rewrapped. This is not a problem when the selection flows along with the text. But when using rectangular selections, this rewrapping would cause the selection to change.

Many text editors do not support rectangular selections. Those that do, are usually IDEs or editors from the DOS world that do not support word wrapping or proportionally spaced (variable width) fonts. EditPad is a flexible editor. It supports both rectangular selections as well as modern conveniences like word wrap and proportionally spaced fonts. But it can't support all this at the same time for the reasons explained above.

There are two ways to make a rectangular selection. First, you can make it the usual way using the keyboard or the mouse, while holding down the Alt button on the keyboard. Second, you can pick Rectangular Selections from the Block menu. This inverts the meaning of the Alt key while making a selection. After choosing Rectangular Selections in the Block menu, a selection made the normal way is rectangular, and a selection made while pressing Alt flows along with the text.

If you did not choose a monospaced font and turn off word wrap, then the selection always flows along with the text. If you pick Block|Rectangular Selections from the menu, EditPad Pro warns you that rectangular

selections are presently not possible. It will offer to adjust the settings for you to make rectangular selection possible.

If you turn on Block|Rectangular Selections then it remains active until you turn it off. If you change the file's settings or if you switch to another file that does not allow rectangular selections then Block|Rectangular Selections has no effect on that file. It will appear inactive even while EditPad Pro remembers that it was active. If you click it then it will offer to adjust the settings. But even if you don't click it, Block|Rectangular Selections will reactivate if you change settings again or switch to a file that allows rectangular selections. To turn it off permanently, click Block|Rectangular Selections while it shows that it is active.

EditPad Lite does not have the Block|Rectangular Selections menu item. But it does allow you to make rectangular selections using the Alt key. You just need to make sure word wrap is off and the font is monospaced.

Column Editing

Column editing is supported by both EditPad Lite and EditPad Pro. Column editing is automatically active if the cursor is at the left- or right-hand edge of the selection. In EditPad Pro, column editing works differently when selections are persistent.

Column editing often results in a new rectangular selection that is zero columns wide. Such a zero-column selection appears as a vertical line between two columns. You can think of it as a multi-line text cursor. You can make such a selection simply by creating a rectangular selection that starts and ends on the same column on different lines.

Column Editing with Non-Persistent Selections

When selections are not persistent, the cursor is always at an edge of the selection. Moving the cursor away from the selection removes the selection. So column editing is always active if there is a non-persistent rectangular selection.

Entering a character on the keyboard replaces the selected columns with that character on each line. You get a new zero-column selection after the entered character on each line. Entering a character on a zero-column selection inserts the character at the selected column on each line and shifts the zero-column selection one column to the right. Combined, entering multiple characters replaces the selected columns with those characters on each line.

When one or more columns are selected, pressing Backspace or Delete on the keyboard deletes the selected columns. It doesn't matter whether the cursor is at the left- or right-hand edge. The selection itself remains where it is but becomes zero columns wide.

Pressing Backspace on a zero-column selection deletes the character to the left of the selection on each selected line. The selection moves one column to the left along with the cursor. Pressing Delete on a zero-column selection deletes the character to the right of the selection on each selected line. The selection stays where it is.

Pasting text that does not contain any line breaks replaces the selected columns with the pasted text on each line. You get a new zero-column selection after the pasted text on each line. Pasting text without line breaks on a zero-column selection inserts the pasted text on each line and shifts the zero-column selection as many columns to the right as were pasted.

Copying a rectangular selection and pasting it over another rectangular selection deletes all the selected columns. The pasted rectangle is then inserted at what was the left-hand edge of the deleted selection. The first line on the clipboard is inserted into the first selected line, the second line on the clipboard into the second selected line, and so on. Nothing is selected after pasting. If there are fewer lines on the clipboard than were selected then the whole selection is still deleted. Nothing is inserted on the extra lines that were selected. If there are more lines on the clipboard than were selected, those lines are inserted at the same column into the lines below the selection. If there are not enough lines in the file below the selection then as many lines are added as needed. They are padded with spaces until the column that you are pasting at.

Copying a linear selection and pasting it over a rectangular selection deletes all the selected columns. All of the pasted text, including its line breaks, is then inserted at what was the upper left corner of the deleted selection. Nothing is inserted on the second and following lines that columns were deleted from.

So, by turning off persistent selections, you can easily replace or fill a rectangular selection with new text.

Column Editing with Persistent Selections

When selections are persistent, moving the cursor away from the edge of the selection cancels column editing without removing the selection. Column editing becomes active again as soon as you move the cursor back to the edge of the selection. Any edits you make while the cursor is not at the selection's edge are made only at the position of the cursor, whether that is inside or outside the selection.

Entering a character on the keyboard inserts that character on each selected line at the column that the text cursor is on. You get a new zero-column selection after the entered character on each line. Nothing is deleted.

Pressing Backspace deletes the character to the left of the cursor's column on each selected line. Pressing Delete deletes the character to the right of the cursor's column on each selected line. If the deleted column was selected, the selection shrinks by one column. If the deleted column was to the left of the selection, the selection shifts one column to the left. Regardless, the cursor stays at the same edge of the selection.

Pasting text that does not contain any line breaks inserts the pasted text at the cursor's column on each selected line. You get a new rectangular selection of the pasted text on each line it was pasted into.

Pasting text that does contain line breaks clears and ignores any persistent selection and is thus unaffected by column editing. Pasting a linear block simply inserts all its text at the cursor's position.

Pasting a rectangular block inserts the first pasted line at the cursor, the second line at the cursor's column on the line below the cursor, and so on. If the cursor was at the bottom of a rectangular selection then the second and following lines are pasted below the selection. If there are not enough lines in the file below the cursor to paste all the lines on the clipboard then additional lines are added to the file. They are padded with spaces until the cursor's column.

So, by turning on persistent selections, you can easily prefix or suffix a rectangular selection with new text.

Block|Begin Selection

Select Begin Selection from the Block menu to tell EditPad Pro to remember the current position of the text cursor. Doing this does not have any visible effect. After using Block|Begin Selection, you can use Block|End Selection to select text starting at the position remembered by Block|Begin Selection. If you have more than one file open in EditPad Pro, Begin Selection remembers its position separately for each file. Selecting Begin Selection when working with one file does not wipe out the position remembered for any other file.

These commands are very useful when you want to make a very large selection, or if you want to perform a search to find the start and the end of the selection. You can do whatever you want between invoking Begin Selection and invoking End Selection, including selecting part of the text in any way. The position stored by Begin Selection is remembered until you use Begin Selection again.

The Begin Selection command works slightly differently when selections are persistent. Then it is possible that part of the file is already selected, without the text cursor being positioned at the end of the selection. In that case, Begin Selection moves the starting position of the selection to the current position of the text cursor, and nothing is remembered. If the new beginning is inside the old selection then the selection shrinks. If the new beginning is further away from the ending than the old beginning then the selection expands. If the new beginning is below the ending position when the old beginning was above the ending, or vice versa, then the selection pivots around the ending position. If you want to expand rather than pivot the selection, use Block|Expand Selection instead.

If there is no selection, Begin Selection remembers the position for use with End Selection, whether selections are persistent or not.

If you use the menu item or toolbar button to invoke this command then it is invoked on EditPad's main editor, where you edit files. If you press the shortcut key on the keyboard, it is invoked on whichever editor is showing the text cursor (vertical blinking bar), whether that's the main editor, the search box, or the replace box.

Block|End Selection

After using Block|Begin Selection, you can select End Selection from the Block menu to select the text between the current position of the text cursor and the position stored by Begin Selection. You can use End Selection as many times as you want after using Begin Selection. The selection made by End Selection always starts at the same position, until you use Begin Selection again. If you want to make a rectangular selection, select Block|Rectangular Selections from the menu before selecting End Selection.

The End Selection command works slightly differently when selections are persistent. Then it is possible that part of the file is already selected, without the text cursor being positioned at the end of the selection. In that case, End Selection selects the text between the starting position of the current selection and the position of the text cursor. If the new ending is inside the old selection then the selection shrinks. If the new ending is further away from the beginning than the old ending then the selection expands. If the new ending is above the beginning position when the old ending was below the beginning, or vice versa, then the selection pivots around the beginning position. If you want to expand rather than pivot the selection, use Block|Expand Selection instead.

Using End Selection when there is a persistent selection does not affect the position remembered by Begin Selection. That position is still used when there is no selection.

If you use the menu item or toolbar button to invoke this command then it is invoked on EditPad's main editor, where you edit files. If you press the shortcut key on the keyboard, it is invoked on whichever editor is showing the text cursor (vertical blinking bar), whether that's the main editor, the search box, or the replace box.

Block| Expand Selection

The Expand Selection command in the Block menu is only useful when selections are persistent. Persistent selections allow you to move the text cursor away from the end of the selection. The Expand Selection command creates a new selection that includes the old selection, and all text between the old selection and the position of the text cursor. The position of the text cursor becomes the new ending point of the selection.

The most significant difference between Block|End Selection and Block|Expand Selection is that End Selection pivots the selection if you've moved the text cursor above the spot where you started making the selection while you ended the selection below the starting position, or vice versa. This causes the old selection to be completely deselected, as the starting position stays put. Only the text between the old starting position and the new ending position is selected.

Expand Selection, on the other hand, always expands the selection. If the new ending is above the old beginning while the old ending was below it, or vice versa, Expand Selection simply makes the old ending position the new beginning position.

Whether selections are persistent or not, you can achieve the same effect by pressing Shift on the keyboard while clicking with the mouse at the position where the new selection should end. The advantage of Block|Expand Selection is that you can assign a keyboard shortcut to it and expand the selection with the keyboard.

Block| Between Matching Brackets

The Between Matching Brackets command is only available if you activated bracket matching on the Brackets page of the file type configuration for the active file's file type. The syntax coloring scheme determines exactly which brackets are matched. See that section in this help file to learn how bracket matching works in EditPad Pro.

When you invoke the Between Matching Brackets command in the Block menu, EditPad Pro selects the text between the nearest pair of matching brackets that surround the text cursor. If some text was already selected, Between Matching Brackets expands the selection to touch the nearest pair of matching brackets surrounding the selection. The brackets are not included in the selection. If the selection already touches the nearest pair of matching brackets, then Between Matching Brackets expands the selection to include the brackets. Repeating the command expands it to touch the next nearest pair of matching brackets. You can repeat the command as long as there is a pair of brackets surrounding the selection.

Block| Unselect

Use the Unselect command in the Block menu to remove the selection without moving the text cursor. Only the selection is removed. The text that was selected remains.

Block| Go to Beginning

Select Go to Beginning in the Block menu or press the corresponding keyboard shortcut to quickly move the text cursor to the beginning of the selection. This does not change the selection or where it begins or ends.

This command is particularly useful when selections are persistent. It allows you to jump back to the selection after you've moved away from it.

Block| Go to End

Select Go to End in the Block menu or press the corresponding keyboard shortcut to quickly move the text cursor to the end of the selection. This does not change the selection or where it begins or ends.

This command is particularly useful when selections are persistent. It allows you to jump back to the selection after you've moved away from it.

Block| Comment

Select Comment from the Block menu to comment out a piece of code in a source code file.

If the current file is being syntax colored by a syntax coloring scheme that contains a character string for single-line comments, Block| Comment automatically uses those characters. If there is no selection then the comment characters are inserted at the first non-whitespace character on the line that the cursor is on. If you've made a selection that flows along with the text then the comment characters are inserted at the first non-whitespace character on each the line. Essentially, indentation is preserved after lines are commented out. Blank lines are not commented out.

If you've made a rectangular selection then the comment characters are inserted on each line covered by the selection at the leftmost column in the selection, even if the line is blank. You can make a rectangular selection at the start of the lines you want to comment out of you prefer all the comment characters to be at the start of the line. If rectangular selections are not available, you can make a normal selection and use Block| Prefix instead to insert the comment characters.

If the syntax coloring scheme does not define characters for single-line comments, but does define characters for multi-line comments, then Block| Comment inserts the characters that start a multi-line comment in the same way that it inserts characters that start a single-line comment. It then additionally inserts the characters that end a multi-line comment at the end of each line or at the right-hand edge of the selection if it is rectangular.

If you want to change the comment characters defined by the syntax coloring scheme then you need to edit the scheme using the syntax coloring scheme editor. You can find the Block|Comment setting in the topmost node in the scheme editor that is automatically selected after you open a scheme file in the syntax coloring scheme editor.

If the current file type does not use a syntax coloring scheme, or the scheme does not define any characters to comment out something, then Block|Comment asks you for the characters used to start a single-line comment. Those are then used in the same way as single-line comment characters from a single coloring scheme. EditPad remembers the comment characters you provided separately for each file while you have the file open.

If only one line was selected or if there was no selection then Block|Comment moves the text cursor to the next line. This allows you to quickly comment out multiple lines without having to select them if you assign a keyboard shortcut to Block|Comment.

To remove the comment characters inserted by Block|Comment, use Block|Uncomment or Block|Toggle Comment.

You can use Block|Comment on lines that are already commented out. Those are then doubly commented out. This allows you to comment out a large block in which some lines are commented out and then uncomment the whole block with Block|Uncomment. The lines that were originally commented out will still be commented out.

Block|Uncomment

Select Uncomment from the Block menu to remove the comment characters added by Block|Comment. When you've made a selection that flows with the text, Block|Uncomment skips over any whitespace at the start of each selected line to look for comment characters to remove. Comment characters after the first non-whitespace character on a line are left in place. When you've made a rectangular selection, Block|Uncomment only removes the comment characters if they start at the left-hand edge of the selection and are completely covered by the selection. Comment characters that start at another column or that are not or only partially selected are left in place.

If the syntax coloring scheme defines characters that start and end multi-line comments then Block|Uncomment removes the characters that start a multi-line comment in the same way that it removes characters that start a single-line comment. On each line where it managed to remove the start of a comment it then also tries to remove the characters that end the comment. For a linear selection, Block|Uncomment skips over any whitespace at the end of the line to look for comment characters to remove. For a rectangular selection, Block|Uncomment only removes the characters that end comments if those characters end at the right-hand edge of the selection and are completely covered by the selection.

Though Block|Uncomment requires you to be a bit precise when selecting the block to be uncommented, this makes it possible to comment out blocks that already have comments in them, and then uncomment those blocks while leaving the original comments.

If only one line was selected or if there was no selection then Block|Uncomment moves the text cursor to the next line. It does this even if the current line had no comment characters to remove. This allows you to quickly uncomment multiple lines without having to select them if you assign a keyboard shortcut to Block|Uncomment.

If you want to change the comment characters defined by the syntax coloring scheme then you need to edit the scheme using the syntax coloring scheme editor. You can find the Block|Comment setting in the topmost node in the scheme editor that is automatically selected after you open a scheme file in the syntax coloring scheme editor.

Block|Toggle Comment

The Toggle Comment command in the Block menu combines the functionality of Block|Comment and Block|Uncomment. On each line in the selection, it attempts to remove the comment characters exactly like Block|Uncomment does. If it does not remove any comment characters from a particular line then it adds comment characters to that line just like Block|Comment does.

If only one line was selected or if there was no selection then Block|Toggle Comment moves the text cursor to the next line. This allows you to quickly comment and uncomment multiple lines without having to select them if you assign a keyboard shortcut to Block|Toggle Comment.

The advantage of Block|Toggle Comment is that it allows you to use the same keyboard shortcut for commenting and uncommenting lines. But if you need to comment out a larger block in which some lines are already commented out that need to stay commented out, then you need to use Block|Comment.

Block|Prefix

Select Prefix in the Block menu to prefix the selected block with one or more characters. If you've made a selection that flows along with the text, the characters are inserted at the very start of each line in the selection. If you've made a rectangular selection, the characters are inserted on each line in the selection, at the leftmost column in the selection. The inserted characters become part of the selection.

Block|Suffix

Select Suffix in the Block menu to suffix the selected block with one or more characters. If you've made a selection that flows along with the text, the characters are inserted at the end of each line in the selection. If you've made a rectangular selection, the characters are inserted on each line in the selection, after the rightmost column in the selection. The inserted characters become part of the selection.

Block|Insert File

Select Insert File from the Block menu to insert the contents of another file into the file you are currently editing. You will be asked to select the file. The text is inserted at the current position of the text cursor. If you select more than one file at the same time, their contents are inserted in alphabetical order of the names of the files

Block| Write

Select Write from the Block menu to save the selected portion of the file you are editing into another file. If the other file already exists, EditPad shows a warning and if you confirm, the other file is overwritten with the selected portion of the current file. If you want to add the selection to the file rather than overwrite it, use Block| Append instead.

If you do not specify an extension for the filename then Block| Write adds an extension depending on the file type you selected in the dialog box. This follows the same rules as File| Save As.

Block| Append

Select Append from the Block menu to save the selected portion of the file you are editing into another file. If the other file already exists, the selection is added to the end of that file. If the file does not yet exist, it is created. Use Block| Write if you want to overwrite the file if it exists.

If you do not specify an extension for the filename then Block| Append adds an extension depending on the file type you selected in the dialog box. This follows the same rules as File| Save As.

Block| Export to HTML or RTF

Select Export to HTML or RTF command in the Block menu to export the selected portion of the file you are editing into another file just like Block| Write does. But the file filter on the export dialog only lets you choose between HTML and RTF. It converts the selection from plain text to HTML or RTF. It preserves the appearance of the selected text in EditPad as much as possible. You will see the text with EditPad's syntax coloring when you open the exported file in a web browser or a word processor.

Block| Print

Select Print from the Block menu to print the selected portion of the current file, rather than the entire file. A preview of the printout is shown first. It allows you to make some changes like which font to print with, which pages should be printed. You can also access the printer setup through the preview window. See the topic on File| Print for a full explanation of the print preview.

There are two key differences between selecting Block| Print in the menu compared with selecting File| Print and turning on the "selection only" in the print preview. The checkbox works at the level of a single line. If a line is partially selected, it is printed entirely. Block| Print, however, sends the exact selection to the print preview, and ultimately the printer. If you want to print a rectangular block then Block| Print is definitely more useful.

Syntax coloring may also be printed differently. Since File| Print sends the whole file to the print preview, the printed lines are colored the same whether you're only printing the selected lines, or you're printing the whole file. But Block| Print only sends the selection to the print preview. Syntax coloring will treat the selection as if it is a complete file because the rest of the file isn't available to the printout. If your selection cuts through syntactic elements, they won't be colored properly in the printout when using Block| Print.

7. Mark Menu

Mark|Go to Next Bookmark

Select Go to Next Bookmark in the Mark menu to move the text cursor to the first bookmark after the cursor in the active file. Repeat this command to move through the bookmarks in the order they are in the file, regardless of the numbers on the bookmarks.

Mark|Go to Previous Bookmark

Select Go to Previous Bookmark in the Mark menu to move the text cursor to the nearest bookmark before the cursor in the active file. Repeat this command to move through the bookmarks in the order they are in the file, regardless of the numbers on the bookmarks.

Mark|Set Any Bookmark

Use the Set Any Bookmark command in the Mark menu to set a bookmark at the current position of the cursor. The number on the bookmark will be the lowest-numbered bookmark that isn't used yet in the active file. When Mark|Project-Wide Bookmarks is active, the number will be the lowest-numbered bookmark that isn't used yet in the active project.

If all 10 numbered bookmarks have been set, the Set Any Bookmark command sets a numberless bookmark. You can place as many numberless bookmarks as you like. Numberless bookmarks cannot be jumped to directly with the Go to Bookmark X commands. You can jump to them with the Go to Next Bookmark and Go to Previous Bookmark commands.

Unlike the Set Bookmark X commands, the Set Any Bookmark command never removes a bookmark from another location. It always sets a new bookmark if the line the cursor is on doesn't have a bookmark yet. If the line the cursor is on does have a bookmark, that bookmark is removed and no new bookmark is set. Using Set Any Bookmark again on a line that already has a numberless bookmark is the only way to remove individual numberless bookmarks.

Though bookmarks are shown in the left margin, they are placed at character positions. Bookmarks stay with the character they were placed at when you edit the file. If you delete the character, the bookmark shifts to the nearest remaining character.

Bookmarks are not saved into a file when you use File|Save. If you tick "preserve cursor position and folding" on the Save Files page in the Preferences then EditPad Pro automatically remembers the bookmarks you set for all the files listed in the File|Open and File|Favorites submenus. EditPad Pro then also saves bookmarks positions in Projects. When opening files by opening a project, the files' bookmarks are restored from the project file. When opening a file without opening a project, its bookmarks are retrieved from the File|Open or File|Favorites submenu if the file is listed there, even if you didn't use the reopen menu or the favorites menu to open the file.

Mark | Project-Wide Bookmarks

By default, bookmarks are separate for each file. You can use Mark|Set Bookmark X to set the same bookmark in each file. Mark|Go to Bookmark X will only jump to bookmarks in the current file. The Mark menu will list the bookmarks for the current file.

If you select the Project-Wide Bookmarks item in the Mark menu to make bookmarks project-wide, you will only have one set of 10 numbered bookmarks for all files in the current project. The Mark menu will show the bookmarks for the current project. If you set a bookmark, that bookmark will be set at the position of the text cursor in the current file, and removed from all other files in the project. When jumping to a numbered bookmark, all the files in the current project will be searched through to find the bookmark. The file it was found in will be activated.

If you set the same bookmark in multiple files in the project with project-wide bookmarks turned off, and then turn on project-wide bookmarks, nothing will happen initially. The bookmark will remain set in all the files it is set in. If you try to jump to the bookmark, and it exists in the current file, EditPad Pro will jump to the bookmark in the current file. If not, it will search through the files in the project and jump to the first file it finds. Since this makes it hard to predict where you'll end up if you jump to the bookmark, you should set it again after turning on project-wide bookmarks. Setting the bookmark again will then remove it from all the other files in the project, giving it one place to jump to.

Whether to use separate bookmarks for each file or project-wide bookmarks depends on your personal editing style, and the task at hand. When working with a handful of very large files, using separate bookmarks gives you 10 numbered bookmarks for each file. You can use the tabs to switch between files. When you're working with a very large number of files and you repeatedly need to switch between them, project-wide bookmarks allow you to instantly switch to a particular spot in a particular file.

Regardless of the state of the Project-Wide Bookmarks option, you can use the Mark|Set Any Bookmark command to set as many numberless bookmarks in as many files as you like. The Project-Wide Bookmarks option only affects numbered bookmarks, of which you can have no more than 10.

Mark | Go to Bookmark 1, 2, 3...

You can quickly jump to any bookmark by pressing Ctrl+1, Ctrl+2, Ctrl+3, etc... on the keyboard. Jumping to a bookmark positions the text cursor on the line on which you set the bookmark with the given number through Mark|Set Bookmark. If the requested bookmark does not exist, nothing happens.

If you remember you set a bookmark but don't remember which number, open the Mark menu. The 10 Go to Bookmark X items in the menu will display the first few words on the line on which each of the bookmarks is set.

Bookmarks can be either separate for each file, or project-wide. When bookmarks are project-wide, the Go to Bookmark command searches for the bookmark in all files in the active project. If the bookmark is found in another file, it will activate that file. When bookmarks are separate for each file, the Go to Bookmark command will never switch to another file.

Mark | Set Bookmark 1, 2, 3...

You can place up to ten numbered bookmarks in every file open in EditPad Pro using the ten Set Bookmark X commands in the Mark menu. You can quickly do so by pressing Shift+Ctrl+1, Shift+Ctrl+2, etc. on the keyboard. When you place a bookmark, you will see a rectangle with the bookmark's number in the left margin, indicating the bookmark's position. You can change the rectangle's and the number's color by customizing the color palette and changing the "Editor: Bookmark icons" color.

You can remove a bookmark by setting the same bookmark again on the same line. You can move a bookmark to another line simply by setting it on another line. If you set a bookmark on a line that already has another bookmark, the other bookmark is removed and the new bookmark is set.

Though bookmarks are shown in the left margin, they are placed at character positions. Bookmarks stay with the character they were placed at when you edit the file. If you delete the character, the bookmark shifts to the nearest remaining character. If you enter a line break in the middle of the line, the bookmark ends up on the line before or after the line break depending on whether the bookmark was placed with a character before or after the line break.

Bookmarks are not saved into a file when you use File|Save. EditPad Pro does automatically remember the bookmarks you set for all files that are listed in the File|Open submenu, and all files you added to your favorites. Projects also store the bookmarks for all the files in the project. When opening files by opening a project, the files' bookmarks are restored from the project file. When opening a file without opening a project, its bookmarks are retrieved from the File|Open submenu or the favorites if the file is listed there, even if you didn't use the File|Open submenu or the Favorites menu to open the file.

Mark | Remove All Bookmarks

The Remove All Bookmarks command in the Mark menu removes all the bookmarks from the active file. If Mark|Project-Wide Bookmarks is enabled, it also removes all the bookmarks from all the other files in the active project.

If you want to remove a single bookmark, place the cursor on the line that the bookmark is on and use the Mark|Set Any Bookmark command.

8. Fold Menu

Fold | Fold

Select Fold in the Fold menu to fold the selected lines. The first line in the selection remains visible, while the others are hidden. They are “folded” underneath the first line. A folding icon appears to the left of the folded line to indicate it is folded. If you click it, the hidden lines become visible again. The folding icon changes to indicate the line is unfolded. Click the folding icon again to refold the range. You can change the appearance of the folding icons, folded lines, and unfolded ranges on the Editor page in the Preferences.

If you did not select part of the text, and the text cursor is inside a foldable range that is unfolded, then the Fold command folds that range.

While folded lines are invisible, they are still fully part of the file, and still take part in all editing actions. If you select a block that includes one or more folded sections and copy the block to the clipboard with Edit | Copy, for example, then all selected lines, including lines hidden by folding, are copied to the clipboard. Folding only affects the display. The only editing commands that take the folding into account are Edit | Delete Line, Fold | Copy Visible Lines and Fold | Delete Folded Lines.

On the Navigation page in the file type configuration, you can choose if and how EditPad Pro should add automatic folding points. Automatic folding points appear as an unfolded range that you can fold with the mouse or the Fold command.

Fold | Unfold

Select Unfold in the Fold menu to unfold a section previously folded with Fold | Fold. The Unfold command only becomes enabled when you’ve placed the text cursor on the first (still visible) line of a folded section. The unfolded section remains marked as a folding point, so you can easily re-fold it.

If you’ve made a selection, then all folding points inside the selection are unfolded.

Use Fold | Remove Fold to unfold the section and remove the folding point at the same time.

If the current file type uses automatic folding points, then unfolded ranges automatically disappear when you edit the file. They are replaced by new automatic folding points that may be the same or different, depending on how your edit affects the file. Folded ranges remain folded until you unfold them.

Fold | Toggle Fold

If the text cursor is inside a foldable range (as indicated in the left margin), the Toggle Fold command in the Fold menu unfolds the foldable range if it was folded, and folds it if it wasn’t folded. This is the same as clicking the folding icon in the left margin.

If the text cursor is not inside a foldable range, and some text is selected, the Toggle Fold command folds the selected text by creating a new folding range, just like the Fold command in the Fold menu does.

Fold | Select Fold

If the text cursor is on a folded line then the Select Fold command in the Fold menu selects that line and all the lines that are folded under it, without unfolding the line. If the cursor is inside a foldable range that is unfolded then that foldable range is selected. If the folded line or foldable range is inside another foldable range, only the folded line or the innermost foldable range is selected. If you repeat the command then the next larger foldable range containing the already selected range is selected.

Fold | Fold Unselected

The Fold Unselected command in the Fold menu folds all existing folding ranges that do not span any selected text. If you did not select any text, the Fold Unselected command folds all folding ranges that do not contain the text cursor.

The Fold Unselected command does nothing with the folding ranges that are (partially) selected. Use Fold|Fold or Fold|Unfold to fold or unfold those. Using Fold|Fold Unselected followed by Fold|Unfold makes it easy to see the structure of the whole file while focusing on the part of the file that you're editing.

Fold | Remove Fold

Select Remove Unfold in the Fold menu to unfold a section previously folded with Fold|Fold, and delete the folding point in the process. The Remove Fold command is enabled only when you've placed the text cursor on the first (still visible) line of a folded section.

If you've made a selection, then all folding points inside the selection are removed.

Use Fold|Unfold to unfold a section without deleting the folding point.

If the current file type uses automatic folding points, then automatically added folding points that you removed may automatically reappear when you edit the file.

Fold | Go to Next Fold

Select Go to Next Fold in the Fold menu to move the text cursor to the first line of the nearest folding range after the cursor. This folding range remains folded or unfolded, as it was.

Fold | Go to Previous Fold

Select Go to Previous Fold in the Fold menu to move the text cursor to the first line of the nearest folding range before the cursor. This folding range remains folded or unfolded, as it was.

Fold | Fold All

The Fold All command in the Fold menu folds all automatic folding points, and any unfolded section still marked with a folding point.

Fold | Unfold All

The Unfold All command in the Fold menu unfolds all sections that you have folded with the Fold command.

Fold | Toggle All Folds

If some or all of the foldable ranges in the file are folded, then the Toggle All Folds command in the Fold menu unfolds all those ranges, just like Fold | Unfold All does. Ranges that were unfolded remain unfolded.

If all of the foldable ranges in the file are already unfolded, then the Toggle All Folds command folds them all, just like Fold | Fold All does.

Fold | Remove All

The Unfold All command in the Fold menu removes all folding points that you added with the Fold command.

If the current file type uses automatic folding points, then automatically added folding points that you removed may automatically reappear when you edit the file.

Fold | Copy Visible Lines

The Copy Visible Lines item in the Fold menu copies the selected text to the clipboard. There are two key differences between the Copy Visible Lines command and the regular Copy command in the Edit menu.

If the selection is not rectangular, and certain lines are only partially selected, then Copy Visible Lines command copies those lines entirely. The Copy command copies partially selected lines partially.

If certain lines are folded, the Copy Visible Lines command does not copy the lines that are made invisible by the folding. The regular Copy command does copy them.

Text copied by the regular Copy command does preserve folding ranges when pasted back into EditPad Pro. All the text is pasted, and remains folded under the pasted folding range. If you paste into another application, then all the text appears without the folding range.

Text copied by the Copy Visible Lines command is always pasted without any folding ranges.

Fold | Delete Folded Lines

Select the Delete Folded Lines item in the Fold menu to delete all lines in the current selection that were made invisible by folding them.

9. Tools Menu

The Tools menu lists all the tools that are defined for the file type of the active file. Select one from the menu to run it.

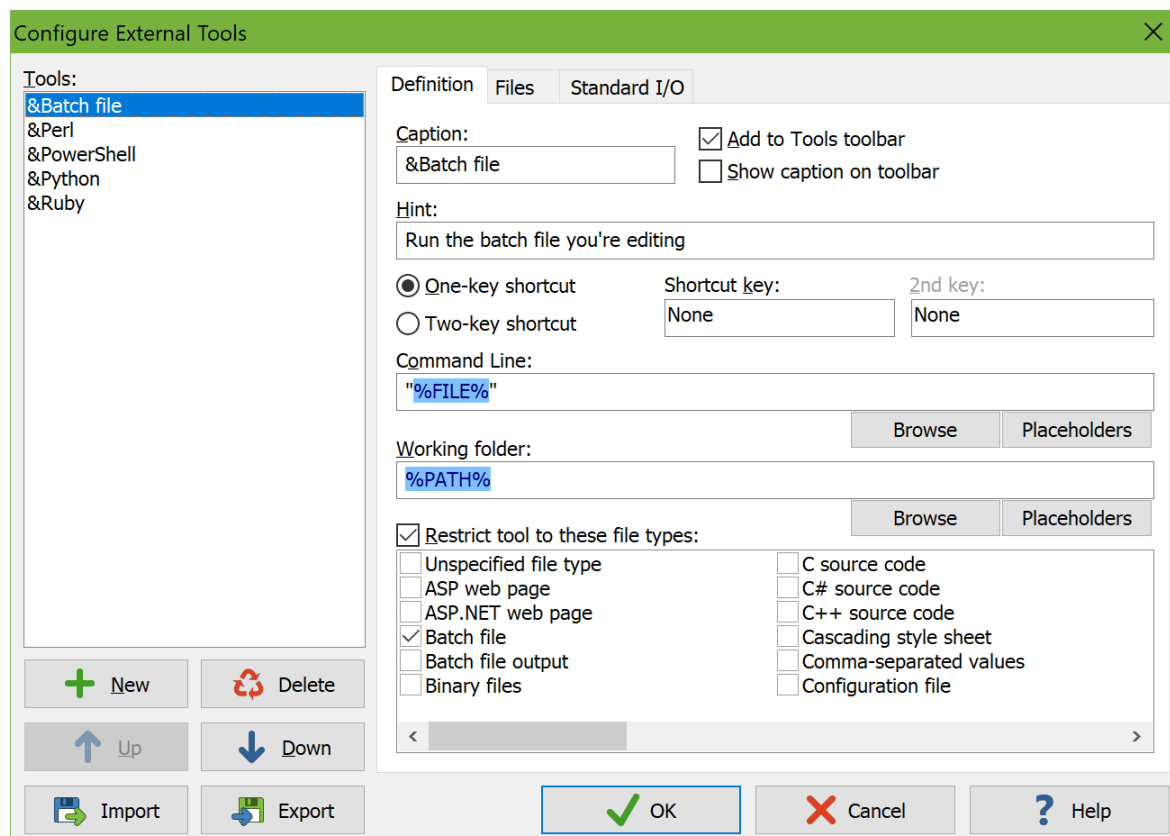
Use the Configure Tools item to configure the tools to be added to the Tools menu. If you have not yet defined any tools then this is the only choice you can make in the Tools menu.

Tools can be configured to display their output in a message panel inside EditPad Pro. After running a tool you can use the Message Panel item in the Tools menu to show or hide this panel.

Configure Tools

EditPad Pro has the capability to run any external command or tool. A tool can be any executable program or application, a script run by an interpreter, or even a URL.

To configure tools, select Configure Tools in the Tools menu. The tool configuration dialog appears. The screen is divided into two parts. The left-hand side has a list with all the currently defined tools. If you haven't defined any tools yet then this list is empty. At the right-hand are three tabs that hold the settings for the tool that's currently selected in the list at the left. The tabs don't appear until you've added at least one tool.



To change a tool's settings, simply click on it in the list and make the changes you want. To create a new tool, click the New button below the list and set the options as you want them. To clone or duplicate a tool, hold down the Control key on the keyboard while clicking the New button. To delete a tool, click on it and click the Delete button. The order of the tools in the list is the order in which they will be shown in the Tools menu. Select a tool and click the Up or Down button to move it.

You can select multiple tools by holding down the Shift or Control key on the keyboard while clicking in the list of tools. The controls under the tabs will indicate the settings for all selected tools. If an edit box or drop-down list shows a value, that means all the selected tools use the same value. Otherwise, the edit box will be blank. If a checkbox is checked or cleared, that means it is checked or cleared for all the selected tools. Otherwise, the checkbox will be filled with a square. If you make a change to any setting, that change is applied to all the selected tools.

You can save tool configurations into a file that you can share with other people. To save one or more tools, select them in the list and click the Export button below the list. You'll be asked for a name of the .ini file into which all the tools will be saved. To load a tool configuration file you've received from somebody, click the Import button. The loaded tools will be added to the list.

For each tool, you can specify three sets of options:

- Definition: Basic settings you need to make for every tool, so EditPad Pro knows which tool to run.
- Files: Options for opening and saving files to be passed on the tool's command line.
- Standard I/O: Options for transferring text to and from console applications. Console applications are textual applications that run in a Command Prompt window instead of using a graphical interface like EditPad Pro.

Tool Definition

When adding a tool to EditPad Pro's Tools menu, there are three pages of settings that you can make for each tool. On the "Definition" page you can set the basic options needed to add any application to EditPad Pro's Tools menu.

"Caption" is the caption that this tool's menu item in the Tools menu will have. If you want to make a certain character an access key shortcut in the menu, precede it with an ampersand (&). The character that follows the ampersand will be shown underlined in the menu (unless you disabled the underlining of access key shortcuts in Windows).

Check "add to Tools toolbar" if you want a button for the tool to appear on the Tools toolbar. This toolbar is hidden by default. You can make it visible by right-clicking on the main menu or any toolbar and selecting Tools.

Check "show caption tool toolbar" if you want the toolbar button for the tool to show the caption that you specified. If not, only the tool's icon is shown. The icon is automatically loaded from the file or application you specify on the command line. Showing the caption can be useful if you have multiple tool configurations for the same application. It does make the tool take up much more space on the toolbar. Items in the Tools menu always show the caption.

"Hint" is the text that will be shown in the status bar when the mouse is pointing at this tool's menu item in the Tools menu. You may leave this blank if the caption is descriptive enough.

If you will be using the tool often, you can assign a keyboard shortcut key combination to it. Click on the “shortcut key” field, and then press the shortcut key combination you want on the keyboard. E.g. if you press F2 on the keyboard, the “shortcut key” field will display F2. The menu item for the tool in the Tools menu will also indicate F2. When you press F2 on the keyboard while working with EditPad Pro, the tool will be run. To remove the shortcut from the tool, click in the “shortcut key” field and press the backspace key on the keyboard.

For the “Command Line”, you need to type in what you would type in if you were to launch the tool from the Run item in the Windows Start menu. It is best to include the full path to the executable and any file parameters. Note that you must *not* use the <, > or | characters for file redirection. If you need file redirection, use the Standard I/O page. If the path to the tool’s executable has spaces in it, you must enclose it in double quotes. Click the Browse button to browse for the executable file.

If you specify a document file instead of an executable file as the command line, then EditPad Pro will launch the document’s associated application, and pass the document file as a parameter. Windows Explorer does the same when you double-click on a document.

If you specify a URL as the command line, then EditPad Pro opens the URL in the web browser configured in the System Preferences.

“Working Folder” is the folder that will appear to be the current one to the tool when it is run by EditPad Pro.

You can use special path placeholders to use the full path to the file you are currently editing, or parts of the file name or path in the command line or in the working folder. If you have marked any of the four temporary file options (see below), you can also use placeholders for the path and file name of the temporary file. There are also path placeholders that allow you to select one of the files open in EditPad Pro or a file on disk to be passed on the command line. Note that many applications cannot handle file names with spaces in them on the command line, unless they are enclosed by double quotes. So it is best to always put double quotes around paths you compose using path placeholders, like I did in the above screen shot.

Click the Placeholders button to get a screen listing all path placeholders, making it easy to use them on the command line. The screen will use the file and project you currently have open in EditPad as example paths. If the active file or project are untitled, the example will be a dummy path. The temporary file placeholders will only be available if you’ve turned on at least one of the options to open or save a temporary file.

When running EditPad Pro from a removable drive, the drive might get a different drive letter if it is inserted into another computer. In that situation, you should reference other applications stored on the same device using the %EPPDRIVE% and %EPPPATH% path placeholders. Then your tools will always reference the files on your drive, regardless of the drive letter it gets.

Since most tools will only make sense when used on a file of a certain type, you can select the file types this tool applies to in the list at the right. Do this by ticking the appropriate checkboxes. Not ticking any checkbox has the same effect as ticking all of them. The Tools menu will show all the tools that apply to the type of file you are currently editing.

Command Line Placeholders for Tools

EditPad Pro supports a range of placeholders that you can use on a tool's command line to pass information about the file or parts of the file you're editing in EditPad Pro.

Placeholder	Meaning in tool command line
%POS%	Number of bytes before the position of the text cursor. Depending on the file's encoding, the number of bytes may not equal the number of characters.
%LINE%	Number of the line the text cursor is on.
%COL%	Number of the column the text cursor is on.
%LINETEXT%	Text of the line the text cursor is on.
%CHAR%	The character immediately to the right of the text cursor.
%WORD%	The word under the text cursor.
%SELSTART%	Number of bytes before the start of the selection.
%SELSTOP%	Number of bytes before the end of the selection. Subtracting %SELSTART% from %SELSTOP% gives the length of the selection in bytes.
%WORD%	The selected text if the selection does not span multiple lines. The selected part of the first line in the selection if the selection spans multiple lines. The word under the cursor if there is no selection.
%ENCODING%	Encoding used by the active file, such as <code>utf-8</code> , <code>windows-1252</code> , <code>iso-8859-1</code> , etc.

Path Placeholders

You can use path placeholders to use part or all of the path and the file name of the file you are currently editing. You can use them in the command line and working folder for tools. This way, you can have a tool work with the file you are editing in EditPad Pro. In the examples below, the file being edited is `C:\data\files\web\log\foo.bar.txt`. If the current file is untitled, all the placeholders below will be replaced with nothing.

If you want to use the path or name of the temporary file that EditPad Pro can open or save when running a tool, simply prepend `TEMP` to the name of the placeholder. E.g. `%TEMPFILE%` is the full path and filename to the temporary file. `%TEMPPATH%` is the folder the temporary file is in.

If the file was opened via FTP, use `%FTPFILE%` to specify the full path to the file on the FTP server. `%FTPSEVER%` is the domain name of the server, and `%FTPURL%`: the full `ftp://` URL to the file.

If you want to pass more than one file on the tool's command line, you can use `%PICK1FILE%` through `%PICK9FILE%` to specify (parts of) the paths of up to 9 files that you have already open in EditPad Pro. If you use these placeholders, each time you run the tool a window will pop up for you to select the file(s) to be passed on the command line from the files you have open in EditPad Pro. The files you select will be saved automatically so the tool can get their current contents. If you don't want the files to be saved, use `%TEMP1FILE%` through `%TEMP9FILE%` instead. These placeholders show the same file picker for files open in EditPad Pro. They don't save the files but save temporary copies instead. The paths to the temporary copies are passed to the tool.

If you want to pass any file, regardless of whether it is open in EditPad Pro or not, use %DISK1FILE% through %DISK9FILE%. These placeholders show a regular open file dialog so you can select a file from disk. If you select a file that is open in EditPad Pro, any unsaved changes are not saved.

All the %FILE% placeholders described in this section represent a whole series of placeholders that hold various parts of the file's path.

Placeholder	Meaning	Example
%FILE%	The entire path plus filename to the file	C:\data\files\web\log\foo.bar.txt
%FILENAME%	The file name without path	foo.bar.txt
%FILENAMENOEXT%	The file name without the extension	foo.bar
%FILENAMENODOT%	The file name cut off at the first dot	foo
%FILEEXT%	The extension of the file name without the dot	txt
%FILELONGEXT%	Everything in the file name after the first dot	bar.txt
%PATH%	The full path without trailing delimiter to the file	C:\data\files\web\log
%DRIVE%	The drive the file is on, without trailing delimiter	C: for drive letter paths; \\server for UNC paths
%FOLDER%	The full path without the drive and without leading or trailing delimiters	data\files\web\log
%FOLDER1%	First folder in the path	data
%FOLDER2%	Second folder in the path	files
(...etc...)		
%FOLDER99%	99th folder in the path.	<i>(nothing)</i>
%FOLDER<1%	Last folder in the path	log
%FOLDER<2%	Second folder from the end in the path	web
(...etc...)		
%FOLDER<99%	99th folder from the end in the path.	<i>(nothing)</i>
%PATH1%	First folder in the path, without delimiters	data
%PATH2%	First two folders in the path, without leading or trailing delimiters	data\files
(...etc...)		
%PATH99%	First 99 folders in the path, without leading or trailing delimiters	data\files\web\log
%PATH<1%	Last folder in the path, without delimiters	log
%PATH<2%	Last two folders in the path, without leading or trailing delimiters	web\log
(...etc...)		

%PATH<99%	Last 99 folders in the path, without leading or trailing delimiters	data\files\web\log
%PATH-1%	Path without the drive or the first folder	files\web\log
%PATH-2%	Path without the drive or the first two folders	web\log
(...etc...)		
%PATH-99%	Path without the drive or the first 99 folders.	<i>(nothing)</i>
%PATH<-1%	Path without the drive or the last folder	data\files\web
%PATH<-2%	Path without the drive or the last two folders	data\files
(...etc...)		
%PATH<-99%	Path without the drive or the last 99 folders.	<i>(nothing)</i>

Combining Path Placeholders

You can string several path placeholders together to form a complete path. If you have a file `c:\data\test\file.txt` then `d:\%FOLDER2%\%FILENAME%` will be substituted with `d:\test\file.txt`. However, if the original file is `c:\more\file.txt` then the same path will be replaced with `d:\file.txt` because `%FOLDER2%` is empty. The result is an invalid path.

The solution is to use combined path placeholders, like this: `d:\%FOLDER2%FILENAME%`. The first example will be substituted with `d:\test\file.txt` just the same, and the second will be substituted with `d:\file.txt`, a valid path. You can combine any number of path placeholders into a single path placeholder, separating them either with backslashes (`\`) or forward slashes (`/`). Place the entire combined placeholder between two percentage signs.

A slash between two placeholders inside the combined placeholder is only added if there is actually something to separate inside the placeholder. Slashes between two placeholders will never cause a slash to be put at the start or the end of the entire resulting path. In the above example, the backslash inside the placeholder is only included in the final path if `%FOLDER2%` is not empty.

A slash right after the first percentage sign makes sure that the resulting path starts with a slash. If the entire resulting path is empty, or if it already starts with a slash, then the slash is not added.

A slash right before the final percentage sign makes sure that the resulting path ends with a slash. If the entire resulting path is empty, or if it already ends with a slash, then the slash is not added.

Mixing backslashes and forward slashes is not permitted. Using a forward slash inside a combined placeholder, will convert all backslashes in the resulting path to forward slashes. This is useful when creating URLs based on file names, as URLs use forward slashes, but Windows file names use backslashes.

Example: If the original path is `c:\data\files\web\log\foo.bar.txt`

Placeholders	Resulting path
%\FOLDER1\%	\data\
%\FOLDER5\%	(nothing)
%PATH-2\FILENAME%	web\log\foo.bar.txt
%PATH-2/FILENAME%	web/log/foo.bar.txt
%PATH-4\FILENAME%	foo.bar.txt
%DRIVE\PATH-2\FILENAME%	c:\web\log\foo.bar.txt
%DRIVE\PATH-4\FILENAME%	c:\foo.bar.txt
%\FOLDER1\FOLDER4\%	\data\log\
%\FOLDER1\FOLDER5\%	\data\

Placeholders That Prompt

If the parameters you want to pass to the tool aren't always the same, you can add a placeholder that EditPad prompts for when you run the tool. Anything in the form of %PLACEHOLDER% that EditPad doesn't recognize is treated as a custom placeholder. You will be prompted to enter the text that this placeholder should be substituted with.

If you want to pass the paths to one or more files you have open in EditPad, use a path placeholder prefixed with PICK and a number from 1 to 9. E.g. %PICK1FILE% lets you pick file #1 and adds its whole path to the command line. If you use multiple placeholders with the same number you will be prompted for only one file. You can use different numbers to be prompted for up to 9 files.

If you want to pass the path to a temporary copy of a file you have open in EditPad, prefix TEMP with a number from 1 to 9 instead. E.g. %TEMP2FILE% lets you pick file #2 and passes the full path to a temporary copy of the file. That copy is created by EditPad before it runs the tool.

If you want to pass the path to any other file, use a path placeholder prefixed with DISK and a number from 1 to 9. EditPad will show a file selection dialog that lets you pick a file on disk when you run the tool.

Tool Files

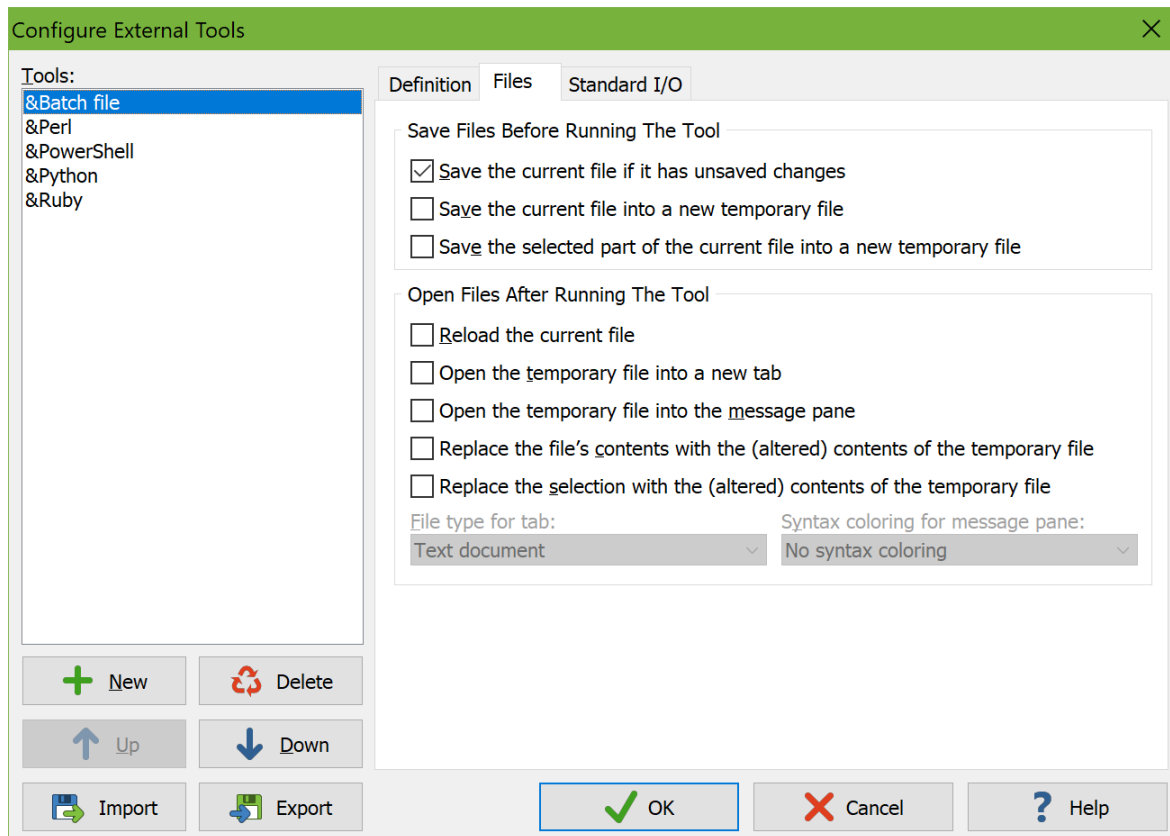
When adding a tool to EditPad Pro's Tools menu, there are three pages of settings that you can make for each tool. On the "Files" page you can choose if EditPad Pro should save the current file or selection, or open files modified by the tool.

Tick "save the current file if it has unsaved changes" if EditPad Pro should automatically do a File|Save before the tool is run. If the file is untitled, you will be prompted for a file name. You can use the %FILE% path placeholder to pass the file on the tool's command line.

Tick "save the current file into a new temporary file" if EditPad Pro should automatically do a File|Save Copy As and save the file under a new, temporary name before the tool is run. This is useful if the tool will modify the file's contents but you do not want to lose the original, or if you want to test changes you made to

a file with a tool without having to save those changes first. The temporary file will be deleted after the tool has finished running.

Tick “save the selected part of current file into a new temporary file” if EditPad Pro should automatically do a Block|Write before the tool is run. You can use this to allow the tool to read or modify the selection. You cannot use this option in combination with saving the entire file into a temporary file, but you can use it in combination with both “open temporary file” (see above) and “replace selection” (see below). The temporary file will be deleted after the tool has finished running.



Turn on “reload the current file” to force EditPad Pro to reload the file that was active when you invoked the tool as soon as the tool has finished running. You should do so if the tool will modify the current file, even if you’ve turned on the option to reload modified files in the Open Files Preferences. Otherwise, EditPad Pro may not notice the file has changed until you switch between tabs in EditPad or between EditPad and another application.

Tick “open the temporary file into a new tab” if the tool will create or modify the temporary file and you want to see the changes in EditPad. If this option is checked, EditPad will open the file into a new tab and then delete the temporary file from disk. If you want to save the tool’s result, use File|Save As. You can select the file type that EditPad Pro should use for the file. Select “same as current file” if the tool doesn’t output a specific file type.

Tick “open the temporary file into the message panel” if the tool will create or modify the temporary file and you want to see the changes in a side panel in EditPad. If this option is checked, EditPad will open the file into the message panel and then delete the temporary file from disk. You can select the syntax coloring scheme that the message panel should use. This scheme can be a scheme that is not used by any file type.

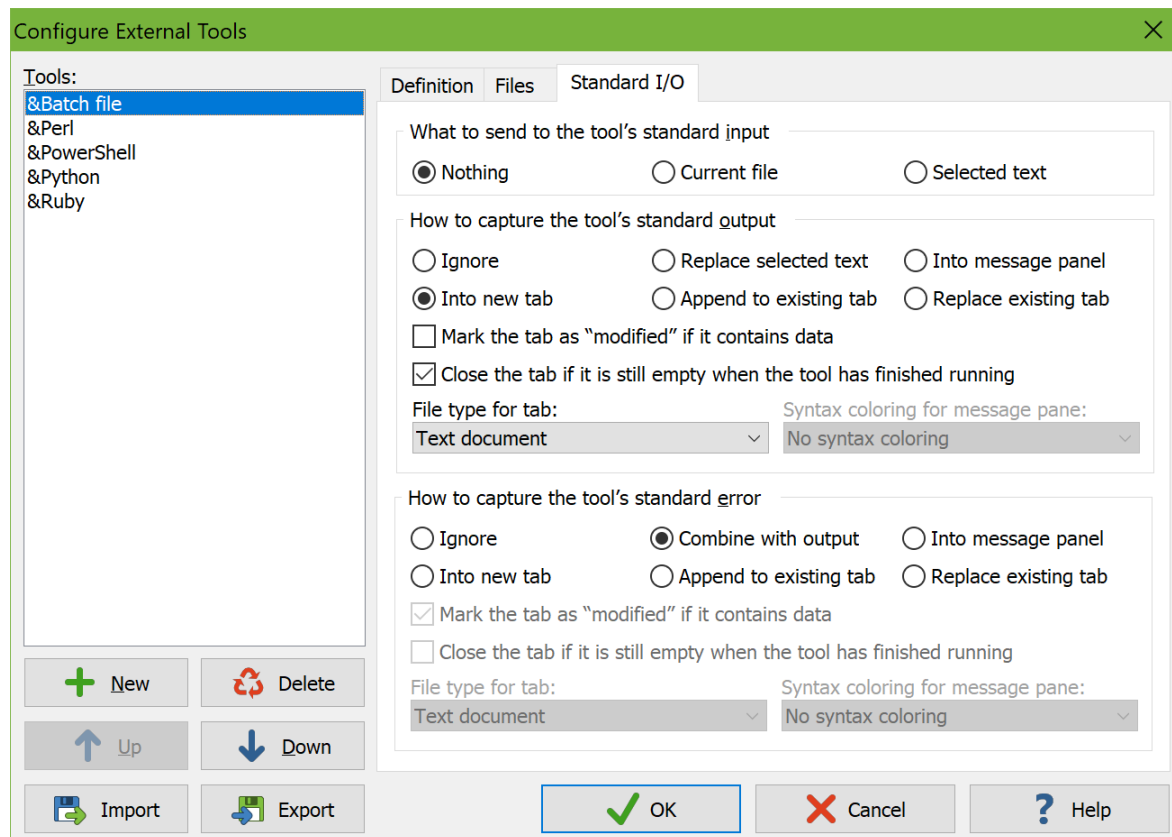
Tick “replace the file’s contents with the (altered) contents of the temporary file” to make EditPad Pro delete all text in the active file and then do a Block|Insert File using the temporary file, after the tool has finished running.

Tick “replace the selection with the (altered) contents of the temporary file” if EditPad Pro should automatically delete the current selection and do a Block|Insert File using the temporary file, after the tool has finished running. Together with the “save the selected part of current file into a new temporary file” option, you can add a tool to EditPad Pro’s menu that modifies the current selection. You could write a script to add special functionality that EditPad Pro does not provide.

If you’ve turned on at least one of the options to either open or save a temporary file, you can use the %TEMPFILE% path placeholder to pass the temporary file’s name on the tool’s command line. All of the options will work on a single temporary file for each tool.

Tool Standard I/O

When adding a tool to EditPad Pro’s Tools menu, there are three pages of settings that you can make for each tool. If you are adding a console or text mode application to EditPad Pro’s Tools menu, then you can instruct EditPad Pro to communicate with the application via the options on the “Standard I/O” page.



By default, console applications read input from the keyboard and write output to the screen. On the DOS and Windows command line, you can use the less than and greater than symbols to redirect standard input

and output. The command line “sometool <input.txt >output.txt” will cause sometool to read from input.txt and write to output.txt.

You cannot use these symbols on the command line when configuring EditPad Pro to run a console application. However, you do have a variety of options for redirecting standard I/O on the “Standard I/O” page.

What to Send to Standard Input

If a tool expects data via standard input (i.e. it expects a DOS/Windows command line like “sometool <input.txt”), EditPad Pro is capable of sending either the current file entirely, or only the selected part of the current file. The current file is whichever file you’re editing at the moment you run the tool by selecting it in the Tools menu.

If the tool is a text-based console application that accepts input from the keyboard, anything EditPad sends to the tool’s standard input will be processed by the tool as if you typed it on the keyboard. This does not work with graphical applications, since those do not use standard input to retrieve keyboard input.

Before running the tool, EditPad Pro makes an internal copy of the text it needs to send to standard input. If the tool takes a long time to run, editing or closing the file EditPad is sending to the tool will not affect the tool’s execution.

How to Capture Standard Output

If a text-based console application writes its output to the screen, or if an application outputs data to a file with a command line like “sometool >output.txt”, EditPad Pro can capture that output in various ways:

- **Ignore:** The tool’s standard output will be lost. Select this option if the tool doesn’t output anything useful to the screen, or if it’s a graphical application that doesn’t support standard output.
- **Replace selected text:** EditPad will replace the selected part of the file that was active when you invoked the tool with the tool’s output. If there is no selection, the tool’s output will be inserted at the text cursor position. This option is particularly useful in combination with the option to send the selected text to standard input. That way, an external tool can provide text processing functions not provided by EditPad Pro itself.
- **Into message panel:** EditPad will capture the tool’s output into a separate panel. Since EditPad Pro has only one message panel, the output of any previous tools that was captured into the message panel will be lost. Choose this option when a tool outputs status or error messages that you want to inspect but not keep. The message panel supports syntax coloring. EditPad Pro ships with a few “Tool Errors” and “Tool Output” syntax coloring schemes that make the output of some popular tools a bit more readable. You can use the Syntax Coloring Scheme Editor to create your own schemes, or you can download schemes in the Colors and Syntax tab in the File Type Configuration.
- **Into new tab:** EditPad will create a new file tab to capture the tool’s output. If you run the tool more than once, a new tab will be created each time. Select this option if the tool outputs a lot of data or a complete file. You can edit, save and close the new tab like any other file you’d edit in EditPad.
- **Append to existing tab:** EditPad will create a new tab to capture the tool’s output, and continue using that tab each time you run that tool, appending additional data each time. Choose this option if you want to save the results of multiple invocations of a tool into a single file.

- **Replace existing tab:** EditPad will create a new tab to capture the tool's output, and replace that tab's contents whenever you run the tool again. Make this choice if the tool outputs too much information to neatly fit into the message panel, and you only want to keep the results of the last invocation of the tool.

If you choose one of the three options to capture the output into a tab, you can make three additional settings for that tab:

- **Mark the tab as “modified” if it contains data:** Turn on this option if you usually want to save the captured output into a file. If you close the file without saving, EditPad Pro will prompt just like when you close an unsaved file. Turn off this option if you usually want to discard the output. Note that if you edit the output, the tab will still become marked as modified.
- **Close the tab if it remains empty:** This option is particularly handy when you've chosen the “into new tab” capturing option. It automatically closes the tab if nothing was captured, so your workspace doesn't get cluttered with empty tabs.
- **File type for tab:** Since standard output doesn't have a file name, EditPad Pro cannot use the file masks from the file type configuration to determine the output's file type. Therefore, you can select one of the file types from the list. Choose the “same as current file” option for text processing tools that return a modified version of the file you're editing.

If you close the tab EditPad Pro is using to capture standard output while the tool is still running, the tool's output will be lost. However, the tool will keep running until normal completion. EditPad Pro will keep clearing the tool's standard output without saving it anywhere.

How to Capture Standard Error

Standard error works just like standard output. While standard output is intended for console applications to produce their output, standard error is intended for their error messages. Not all console applications respect this distinction. Some may send error messages to standard output. When the application is run from a command prompt with output appearing on the screen, standard output and error are automatically combined onto the single screen. This makes some programmers forget to make their console applications distinguish between standard output and error.

Therefore, EditPad Pro offers the “combine with output” option. EditPad Pro will then mingle standard error with standard output, just like a console screen would. The combined output will end up in whichever location (tab or message panel) you selected for standard output.

All the other choices for standard error are identical to those for standard output. If you select one of the three tab options for both standard output and standard error, you will get two new tabs: one for standard output, and one for standard error.

Since EditPad Pro has only one message panel, the only way to capture both standard output and standard error into the message panel is by selecting the “message panel” option for standard output, and the “combine with output” option for standard error.

Tools | Message Panel

On the Files page in the tool configuration you can choose to load a temporary file created or modified by a tool into the Message panel. On the I/O page you can choose to have the tool's standard output and/or error redirected to the Message panel. The Message panel is a side panel in EditPad Pro. It has just one read-only edit box for displaying tool output. You can right-click the edit box to select and copy the text, and to toggle word wrap.

If you've closed the Message panel after running a tool, you can use the Message Panel item in the Tools menu to make it visible again. This item only appears in the Tools menu after you've run a tool that uses the Message Panel. You cannot make the Message Panel visible if it hasn't been used by a tool yet.



The screenshot shows a window titled "Build.bat output + errors". The content is as follows:

```
1 dir(s) moved.

S:\JGsoft\EditPad7\Windows\Pro>MOVE S:\Delphi\dxVCL\Library\Delphi15saved S:\Delphi\dxVCL\Library\Delphi15
1 dir(s) moved.

S:\JGsoft\EditPad7\Windows\Pro>CD ..

S:\JGsoft\EditPad7\Windows>CD ..

S:\JGsoft\EditPad7>DEL S:\JGsoft\EditPad7\Bin\Pro\*.map
Could Not Find S:\JGsoft\EditPad7\Bin\Pro\*.map
```

10. Macros Menu

Via the Macros menu you can record and play back keystroke macros. With keystroke macros, you can automate repetitive editing tasks.

Macros that you have already recorded appear at the top of the Macros menu. Selecting a macro in the menu executes it once. Select Organize Macros in the Macros menu if you want to run a macro multiple times.

Macros | Record Macro

To record a macro, select the Record Macro item in the Macros menu. EditPad Pro will ask you to enter a name for the macro. When you confirm the macro's properties, recording begins immediately.

When you're done recording the macro, select the Stop Recording item in the Macros menu. You'll notice it's the only available command during recording. You cannot play back and record macros at the same time.

To play back a macro, simply select it from the Macros menu or press its keyboard shortcut. The macro will be executed once. Select Organize Macros in the Macros menu if you want to run a macro multiple times.

EditPad Pro automatically saves macros. You won't be prompted for a file name. If you want to save a macro into a file so you can share it with other people, use the Export command in the Organize Macros screen.

How EditPad Pro Records Macros

Before you start recording and playing back macros, it's important that you understand exactly what EditPad Pro records. If you understand macros, they're a powerful and quick way to automate otherwise tedious and repetitive editing tasks. If you don't understand macros, they're a quick way to completely mess up your files.

Mouse Clicks Are Not Recorded

Clicks are relative to the position and size of EditPad's windows, to the location where you've scrolled, etc. A click at a certain position on the screen may have a totally different meaning from one situation to the next. Therefore, mouse clicks are not recorded. If you move the mouse over the text editing area, the mouse pointer will indicate you can't click. If you try, the click will have no effect at all.

Menu item commands and button commands are recorded. You can invoke them with the mouse while recording a macro. EditPad Pro will record the command that you clicked on, rather than the click itself.

Only Your Actual Actions Are Recorded

EditPad Pro only records exactly those keystrokes that you type, or exactly those commands that you invoke when running a macro. E.g. if you select the Search|Find Next menu item while recording a macro, the macro only records that you selected Search|Find Next. It does *not* record the search term, the search options, etc. If you play back the macro, it will simply execute Search|Find Next again. If you've changed the

search term or search options since recording the macro, the new term and options will be used for the search.

By recording only your actual actions, you can make generic macros. E.g. if you want a macro that deletes every line with a search match, you can record Search|Find Next and Edit|Delete Line. You can then use that macro to delete the matches of any search term. Simply enter the correct search term before running the macro.

This also applies to any keystrokes you type into the editor, search box or replace box. The macro will record the actual keystrokes. Their effect may be different when you run the macro, depending on the position of the text cursor and if any text was selected. Keep this in mind when recording macros. You may want to press a few extra keys to make sure the macro works well. E.g. instead of requiring a macro to have the cursor at the start of the line, simply press the Home key when you start recording the macro. Even if the cursor was already at the start of the line when you recorded the macro, it will still record that you pressed the Home key.

How to Record Search Terms

If you want to record search terms as part of a macro, you can do so by selecting the search term from the Search|History or Search|Favorites menu. The macro will then record the search text, replace text and all search options. You can temporarily add a search term to the favorites before recording the macro. The macro will store the actual search terms, not the fact that you selected an item from the history or favorites. If you later delete the item from the history or favorites, the macro will still work.

Macros Record Command Options

Some commands in the Extra menu, like Delete Duplicate Lines and Compare Files show a screen with options before doing their work. If you use these commands while recording, the macro will record the state of the options screen when you click the OK button. It will not record whether you toggled any options, but the final result. When you play back the macro, the command will be executed with those options every time, without showing the options screen.

This rule may seem to contradict the “only your actual actions are recorded” rule. However, there’s a key difference between the search options and the alphabetic sort options. You can set the search options in EditPad Pro’s main window. If you toggle a search option, the option’s state is recorded. Therefore, you can “prepare” the search options before playing back a macro that didn’t record them. You could even record a macro that does nothing but set search options.

The Delete Duplicate Lines command’s options can only be set when you’ve already told EditPad Pro to delete duplicate lines. You can’t prepare the options prior to executing the macro.

The state of options that you can prepare are not recorded by macros, since actions that change the state of options that you can prepare are recorded. Commands with options that you cannot prepare record the state of the options as part of the command.

Failed and Inconsequential Commands Are Recorded

Pressing the Home key on the keyboard while the text cursor is already at the start of the line does nothing. However, macros still record that inconsequential key press. When the macro is played back, it will move the cursor to the start of the line if it isn't there already.

Search commands are special in EditPad Pro. They can fail. If you click the Find Next button and it flashes, that means it failed. There was no next search match. While recording a macro, the Find Next command is still recorded.

Failed Commands Halt Macro Playback

Though not exactly a recording issue, it's important to remember that if a search command fails during macro playback, it will stop the macro immediately. This allows you to run macros that run until failure to process all search terms.

Organize Macros

Select Organize Macros in the Macros menu to organize, play back and record macros.

Repeated Playback

Playing back a macro via the Organize Macros allows you to play back the same macro many times. Simply enter the number of times the macro should run in the Repeat box, and click the Play Macro button. The macro may run fewer times than you specified, but never more.

If the macro includes a search command, and the search fails to find a match, macro playback will stop immediately. This enables you to record a macro that does something to a search match, and then run that macro on all search matches. Suppose you record a macro that executes the Search|Find Next command and then the Edit|Delete Line command. If you then execute this macro with Repeat set to 100, the macro will delete the lines containing the next 100 search matches. If there are fewer than 100 search matches, all their lines will be deleted, and the macro will stop. You can use the Search|Count Matches command to determine the number of times the macro should run, or you could simply enter a large enough number.

However, there is a reason why you still need to enter an upper limit for the number of times the macro should run. You can't tell EditPad Pro to run the macro until the search fails. The reason is that it's quite possible for a search command to never fail. E.g. a macro that runs Search|Find First but doesn't delete the search match will always find that search match. A macro using Search|Find Next with the "loop automatically" option turned on will loop forever if it doesn't delete search matches. By specifying a reasonable number of times for the macro to repeat itself, you can prevent it from going on forever.

Recording

Click the Record button to record a new macro. EditPad Pro will ask you to enter a name for the macro. When you confirm the macro's properties, the Organize Macros screen closes and recording begins

immediately. There are a number of important issues to keep in mind when recording macros. Make sure to review them.

When you're done recording the macro, select the Stop Recording item in the Macros menu. You'll notice it's the only available command during recording. You cannot play back and record macros at the same time.

Organizing Macros

Click the Properties button to rename a macro or to change its shortcut key, or to rename a folder. You can add folders with the New Folder button. To put macros into a folder, simply drag and drop them with the mouse. If you have a lot of macros, organizing them in folders makes it easier to find them later.

Import and Export

EditPad Pro automatically saves all your macros internally along with your preferences, history lists, etc. If you want to share a macro with other people, or copy it from one computer to another, use the Export Macro button. The selected macro is then saved into the file that you specified. You can open the file in EditPad Pro to view its contents if you like. However, macro files are not intended to be edited.

If you've received a macro that you'd like to run in a file, first import it with the Import Macro button. The macro is then added to your list of macros. You can play back imported macros just like macros that you recorded yourself. The macro file is no longer needed once you've imported it.

You can export multiple macros into a single file by selecting multiple macros before clicking the Export Macro button. If you include one or more folders in your selection, all the macros in those folders and their subfolders will be exported. If you click the Export Macro button a second time and select an existing file, EditPad Pro will replace that file rather than append to it. When importing a file that holds more than one macro, all of the macros will be imported. If you wanted to import only some, you can delete the extraneous ones after importing the file.

Macros | Record Instant Macro

The Record Instant Macro command in the Macros menu records a macro just like Macros | Record Macro does. The only difference is that EditPad Pro doesn't ask you for a name and keyboard shortcut for the macro. The only way to play back the instant macro is with Macros | Play Instant Macro.

Use instant macros when you want to quickly record and play back a group of keystrokes a few times. Only one instant macro is preserved at a time. Recording a new instant macro deletes the previous instant macro without warning. The instant macro is not preserved when you close and restart EditPad Pro. The only way to preserve an instant macro is to use Keep Instant Macro to add it to the Macros menu.

Macros | Play Instant Macro

Use the Play Instant Macro command in the Macros menu to play back the macro you last recorded with Macros | Record Instant Macro.

Macros | Keep Instant Macro

Use the Keep Instant Macro command in the Macros menu to add the macro that you last recorded with Macros | Record Instant Macro to the Macros menu. This is the only way to preserve an instant macro before recording another one or before closing EditPad Pro.

Macro Properties

The Macro Properties screen appears when you select the Record Macro item in the Macros menu, and when you click the Properties button when organizing macros.

EditPad Pro macros have only two properties: a name, and an optional keyboard shortcut. Each macro must have a unique name. If you start recording a new macro with the same name as an existing macro, the existing macro is automatically and silently replaced.

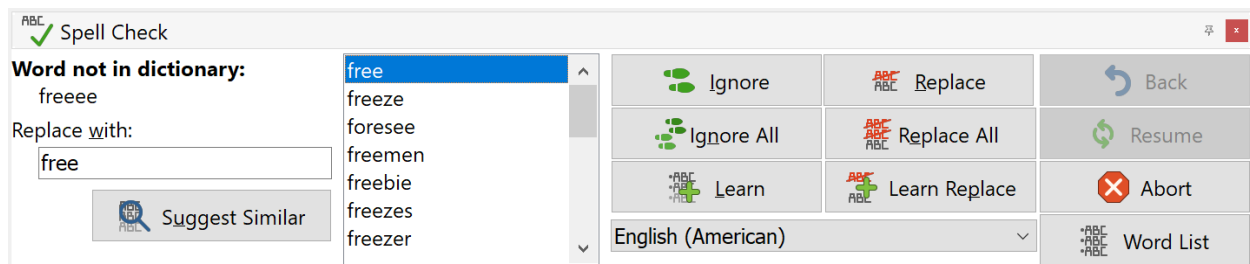
The keyboard shortcut can either be a single key combination, or a double key combination. Assigning a shortcut to a macro works just like assigning shortcuts in the Keyboard preferences. The procedure is explained in detail in the help topic about Keyboard preferences. If you assign a shortcut to a macro that is already assigned to another macro, the new macro will take over the shortcut from the old macro. You cannot assign a shortcut that's already used for a menu item to a macro.

11. Extra Menu

Extra | Spell Check

Select Extra | Spell Check from the Extra menu to check the spelling of the current file. The spell check function always starts from the beginning of the file.

The panel below appears during the spell check process:



When a word is not found in the dictionary, it is selected in the editor and reported in the upper left area of the spell check panel. EditPad Pro starts searching through the dictionary for words similar to the misspelled one, and shows them in the list. As long as “(searching...)” is visible, EditPad Pro is still looking for more possible, correctly spelled substitutions. You do not have to wait until EditPad Pro is finished. You can correct the error right away.

If the word is really misspelled, you can type in the correctly spelled word in the “Replace with” field. You can also click on an item in the list with EditPad Pro’s suggestions. That word is then placed in the “Replace with” field. If you click on “Suggest similar”, EditPad Pro starts looking for words in the dictionary similar to what you typed into the “Replace with” field.

Then you need to click one of the three replace buttons: “Replace” replaces the misspelled word with the replacement just this time. “Replace All” replaces the misspelled word with the replacement every time it is found during the current EditPad Pro session. When you pick File | Exit or switch to a different language, the replacement is forgotten. If you click “Learn Replace” then EditPad Pro adds the replacement to its word list. It then automatically replaces the misspelled word with the replacement every time it is found even after you quit and restart EditPad Pro.

You can also fix the error directly in the editor. After doing so, click the “Resume” button to continue to check the spelling of the rest of the file.

If the word is correctly spelled, you can click “Ignore” to accept the word as correct just this time. “Ignore All” makes EditPad Pro ignore all occurrences of this word until you switch languages or close EditPad Pro. If you click “Learn” then EditPad Pro adds the word to its word list. From then on that word is accepted as correctly spelled.

If you make a mistake, click on “Back” to restore the original word.

Click “Abort” to stop spell checking and close the panel.

The “Word List” button allows you to edit the list of words that you added to the custom dictionary. When you click the button, a new window appears. The left-hand side of the window contains the list of words that you specified as correctly spelled by clicking on the “Learn” button. The right-hand side of the window contains pairs of words separated by an equals sign (=). These are the automatic replacements you added by clicking the “Learn Replace” button. Whenever the spell checker finds the word at the left side of the equals sign, it replaces it with the word at the right of the equals sign.

Extra | Spell Check Selection

Select “Spell Check Selection” from the Extra menu to check the selected portion of the current file for spelling errors. See Extra | Spell Check for a description of the spell check panel.

Extra | Spell Check Project

Select “Spell Check Project” from the Extra menu to check all files in the current project for spelling errors. See Extra | Spell Check for a description of the spell check panel.

Extra | Spell Check All

Extra | Spell Check All activates the first file and starts the spell check process. When the first file has been completed, the second file is activated and the spell checker is invoked on that file. This continues until all files have been checked or you clicked the Abort button on the spell check page.

See Extra | Spell Check for a description of the spell check panel.

Extra | Live Spelling

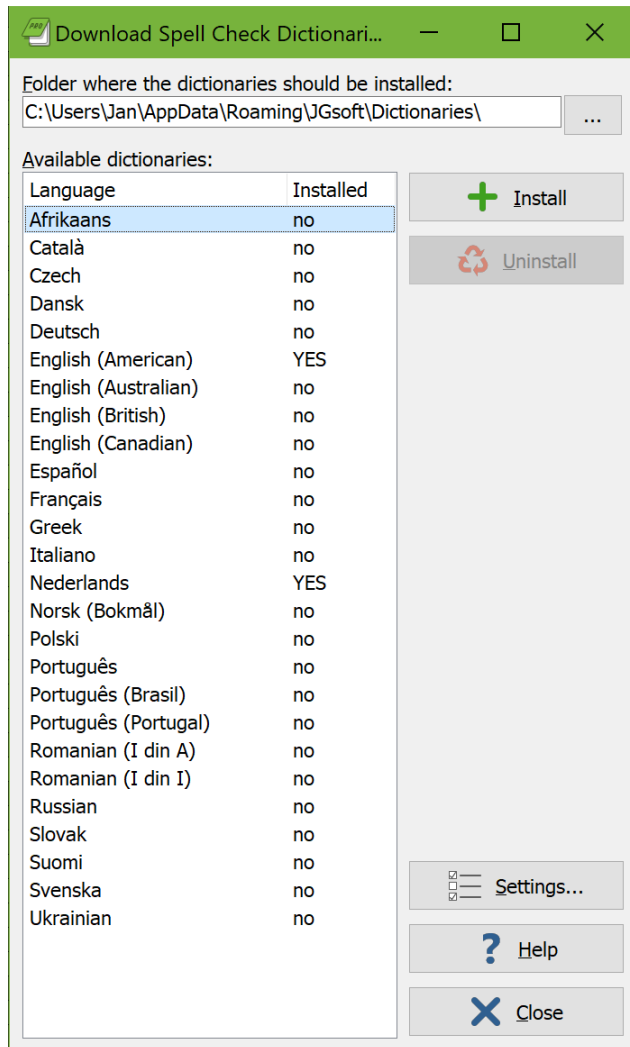
Pick Live Spelling from the Extra menu to turn on or off live spell checking for the current file. Live spell checking marks all words that do not appear in the dictionary. By default they appear with a red wavy underline. You can choose whether live spelling is enabled by default on the Color & Syntax page in the file type configuration. Click the Customize button there and change the “Editor: Misspelled word” color to change the appearance of misspelled words.

Live spelling, as well as the regular spell check in EditPad Pro, works together with EditPad Pro’s syntax coloring. Each syntax coloring scheme can exclude certain parts of the file from the spell check. Schemes for programming languages, such as the C++ and Java schemes, restrict the spell checker to strings and comments. Those are likely to contain natural language, and therefore words that should appear in the dictionary. Other parts of the file are likely to contain specialized programming syntax, consisting of words and character sequences that are not English words.

If a word is marked as misspelled, you can double click on it to open the spell check panel. The panel presents you with a list of suggested replacements for the word. It also allows you to ignore the word or add it to the dictionary. As soon as the word has passed the spell check, the spell check panel closes automatically.

Download Spell Checker Dictionaries

Before you can use Extra|Spell Check or Extra|Live Spelling, you need to download and install spell checker dictionaries for one or more languages. You can easily do so right within EditPad Pro. Pick Options|Configure File Types from the menu, and click on the Colors and Syntax tab. Click the button “Download Spell Checker Dictionaries”. EditPad Pro then connects to the Internet to fetch a list of available languages.



If you want, you can choose the folder into which the dictionaries should be installed. If you specify a folder that does not exist then EditPad Pro creates it. All dictionaries must be installed into the same folder. If you have already installed some dictionaries, and then install another dictionary into a different folder, EditPad Pro moves the previously installed dictionaries to the new folder.

To install a dictionary, click on the language you want in the list, and click the Install button. EditPad Pro then downloads the dictionary and installs it into the folder you specified. A progress meter appears while downloading.

You can have as many dictionaries installed as you want. There is no need to uninstall dictionaries. However, if you want, you can select a language and click the Uninstall button to delete a spell checker dictionary.

If you are behind a proxy server, and EditPad Pro is unable to detect your proxy settings, you can change them by clicking the Settings button.

Due to firewall and other network settings, it is possible that EditPad Pro cannot connect to the Internet directly. In that case, you can manually download and install the dictionaries from <https://www.editpadpro.com/spell.html>.

Extra | Sort Alphabetically A-Z

If no part of the active file has been selected then Extra | Sort Alphabetically sorts the entire file alphabetically, paragraph by paragraph.

To sort the entire file on a specific column of text, select that column of text before selecting Extra | Sort Alphabetically. Make sure that the selection does not span more than one paragraph. All the paragraphs in the file are sorted, using the selected column as the sort key.

If the selection spans more than one paragraph, then the selected paragraphs are sorted. If you want to sort only a selected number of paragraphs on a certain column then make a rectangular selection first. The paragraphs that are partially covered by the rectangular selection are sorted entirely as if the selection was a normal one. However, the sort order is determined only by the text covered by the selection.

If you are working with a text file that contains information arranged in columns, padded with spaces, you could make a rectangular selection of the third column and then use Extra | Sort Alphabetically. The data is then sorted on the third column.

Alphabetic sort works on a character by character basis, even for numbers. “A19” sorts before “A4” because 1 sorts before 4. Use Extra | Sort Alphanumerically A-Z if you want numbers to sort logically.

Extra | Sort Alphabetically Z-A

Extra | Sort Alphabetically Z-A works just like Extra | Sort Alphabetically A-Z, except that the file or the selected text is sorted in reverse alphabetical order.

Alphabetic sort works on a character by character basis, even for numbers. “A4” reverse sorts before “A19” because 4 reverse sorts before 1. Use Extra | Sort Alphanumerically Z-A, 9-0 if you want numbers to sort logically.

Extra | Sort Alphanumerically 0-9, A-Z

Extra | Sort Alphanumerically 0-9, A-Z works just like Extra | Sort Alphabetically A-Z, except that it sorts numbers logically. When the text contains numbers that consist of multiple digits, the numbers are compared as a whole. So “A4” sorts before “A10” because 4 is less than 10. This sort order is also known as “natural sort”.

Extra | Sort Alphanumerically Z-A, 9-0

Extra | Sort Alphanumerically Z-A, 9-0 works just like Extra | Sort Alphanumerically 0-9, A-Z, except that the file or the selected text is sorted in reverse alphanumeric order.

When the text contains numbers that consist of multiple digits, the numbers are compared as a whole. So “A10” reverse sorts before “A4” because 10 is greater than 4.

Extra | Delete Duplicate Lines

Select the Delete Duplicate Lines item in the Extra menu to delete lines with (nearly) identical text on them. The status bar indicates how many lines were deleted. You can make a number of choices as to what EditPad Pro considers to be a duplicate line.

Scope

If you’ve selected part of the file before using the Delete Duplicate Lines command, you can limit the command to delete only lines that are selected. If the first and/or last line in the selection are only partially selected, the selection is expanded to include them entirely. If the selection is rectangular then lines covered by the selection are deleted entirely.

Proximity of Duplicate Lines

Select “anywhere in the scope” to delete all lines that are duplicated anywhere. The first copy of the line remains. All the others are deleted. If you’ve set the scope to “selected lines” then the lines must be duplicated inside the selection. Lines that are not duplicated inside the selection are not deleted, even if they have duplicate lines outside the selection.

Select “adjacent lines only” if you only want to delete a line’s duplicates if they’re immediately below the line they duplicate, without any other lines between them. If the file’s lines are sorted alphabetically, then the end result of “anywhere in the scope” and “adjacent lines only” is the same. In a sorted file, all duplicates are sorted together. Selecting “adjacent lines only”, however, deletes the duplicate lines significantly faster, certainly when the number of lines in the file is large. If you select “anywhere in the scope”, EditPad Pro has to compare each line with every other line in the file.

Comparison Options

By turning on one or more comparison options, you can tell EditPad Pro to consider lines as duplicates even when they aren’t identical.

The option “compare selected columns only” is only available when you’ve made a selection that does not span more than one line, or when you’ve made a rectangular selection. With this option, EditPad Pro only compares the selected columns. If the selection spans from column 10 to column 18, for example, EditPad Pro compares columns 10 through 18 of each line. If those 9 characters are the same the lines are considered to be duplicates. If a line has less than 10 characters then it is considered to be blank. This has important consequences (see next section).

“Ignore differences in leading spaces and tabs” treats lines that only differ in the number of spaces and tabs at the start of the line as duplicates. Similarly, “ignore trailing spaces and tabs” ignores differences in spaces and tabs at the end of each line. “Ignore all differences in spaces and tabs” is more than a combination of the two previous options. EditPad Pro then completely ignores all spaces and tabs, including spaces and tabs in the middle of lines.

“Ignore difference in case” compares lines without regard to the difference between uppercase and lowercase letters.

Lines to Delete

You have to select one or two choices in the “lines to delete” section. Every line in the file belongs to one of the 3 categories. Selecting none of the options would have no effect. Selecting all of them would delete all the lines in the file.

Turn on “2nd and following occurrences of duplicate lines” and turn off the other two options to delete all duplicate lines, leaving only unique lines in the file, regardless of whether they were previously unique. Use this to delete unnecessary duplicates from a file.

Turn on both “2nd...” and “1st occurrence of duplicate lines” to delete all duplicate lines, leaving only lines that were previously unique.

Turn on both “2nd...” and “non-duplicated lines” to leave only one copy of all lines that had duplicates. If you paste the contents of two lists that consist of unique lines (when viewed separately) into a file in EditPad, then you can use this combination to get the lines that occurred in both files, but not the lines that occurred in only one of the files.

If you want to keep only lines that occur a certain number of times, use the Delete Duplicate Lines several times. If you only want lines that occur 3 times or more, for example, use it twice with the “1st occurrence...” and “non-duplicated...” options turned on. Then use it again with the “2nd occurrence...” and “non-duplicated...” options. The first time you delete the lines that occur only once. The second time you delete lines that occurred only twice. The third time you delete the duplicates of lines that occur four times or more.

Blank Lines

Since blank lines are technically all duplicates of each other, EditPad Pro offers you an extra choice for blank lines. You can choose to either delete all blank lines, not to delete any blank lines, or to only delete duplicate blank lines. The “duplicate blank lines” option takes into account the “proximity” setting, deleting either all but the first blank lines (“anywhere in the scope”), or only replacing subsequent blank lines with a single blank line (“adjacent lines only”).

If you’ve turned on the “compare selected columns only” option, a line may be considered blank even when it isn’t. If a line is shorter than the leftmost column in the selection, it is considered to be blank, even if it does have text on it.

Lines with only spaces and tabs on them are only considered to be blank if you’ve turned on one of the options to ignore differences in spaces and tabs. On a line with only spaces and tabs, all spaces and tabs are considered to be both leading and trailing at the same time.

Extra | Delete Blank Lines

Select Delete Blank Lines in the Extra menu to delete all totally blank lines from the file. Only lines that have no characters at all are deleted. Lines that consist solely of whitespace are not deleted.

If you want to delete lines that have whitespace too, first use Extra | Trim Trailing Whitespace to remove the whitespace, and then use Delete Blank Lines.

Extra | Consolidate Blank Lines

Select Consolidate Blank Lines in the Extra menu to replace all blocks of consecutive totally blank lines with a single blank line. Only lines that have no characters at all are consolidated. Lines that consist solely of whitespace are ignored.

If you want to consolidate lines that have whitespace too, first use Extra | Trim Trailing Whitespace to remove the whitespace, and then use Consolidate Blank Lines.

Extra | Trim Whitespace

Select Trim Whitespace in the Extra menu to remove whitespace from the start and the end of each line. If you selected part of the file, only the selected lines are trimmed. If not, all lines in the file are trimmed.

If you've made a rectangular selection, EditPad Pro only deletes whitespace inside the rectangular block. Whitespace in the leftmost selected column and adjacent columns to the right is deleted. Whitespace in the rightmost selected column and adjacent columns to the left is also deleted. If a line is short and doesn't extend to the rightmost selected column, any whitespace at the end of the line covered by the selection is deleted.

Extra | Trim Leading Whitespace

Select Trim Leading Whitespace in the Extra menu to remove whitespace from the start of each line. If you selected part of the file, only the selected lines are trimmed. If not, all lines in the file are trimmed.

If you've made a rectangular selection, EditPad Pro only deletes whitespace inside the rectangular block. Whitespace in the leftmost selected column and adjacent columns to the right is deleted. This whitespace needn't be at the start of the line

Extra | Trim Trailing Whitespace

Select Trim Trailing Whitespace in the Extra menu to remove whitespace from the end of each line. If you selected part of the file, only the selected lines are trimmed. If not, all lines in the file are trimmed.

If you've made a rectangular selection, EditPad Pro only deletes whitespace inside the rectangular block. Whitespace in the rightmost selected column and adjacent columns to the left is deleted. This whitespace

needn't be at the end of the line. If a line is short and doesn't extend to the rightmost selected column, any whitespace at the end of the line covered by the selection is deleted.

In EditPad Pro, the Editor page in the file type configuration has an option to “trim all trailing whitespace upon save”. If you turn this on then Extra|Trim Trailing Whitespace is automatically invoked just before the file is saved. If any spaces are actually trimmed then this appears as a separate item in the undo history.

There is also a “trim trailing whitespace” option in EditPad Pro's file type configuration. That option sets the default state of Extra|Auto Trim Trailing Whitespace.

Extra|Auto Trim Trailing Whitespace

If you turn on Auto Trim Trailing Whitespace in the Extra menu then EditPad Pro automatically trims trailing whitespace from lines that you have edited. This happens when you move the cursor away from the line.

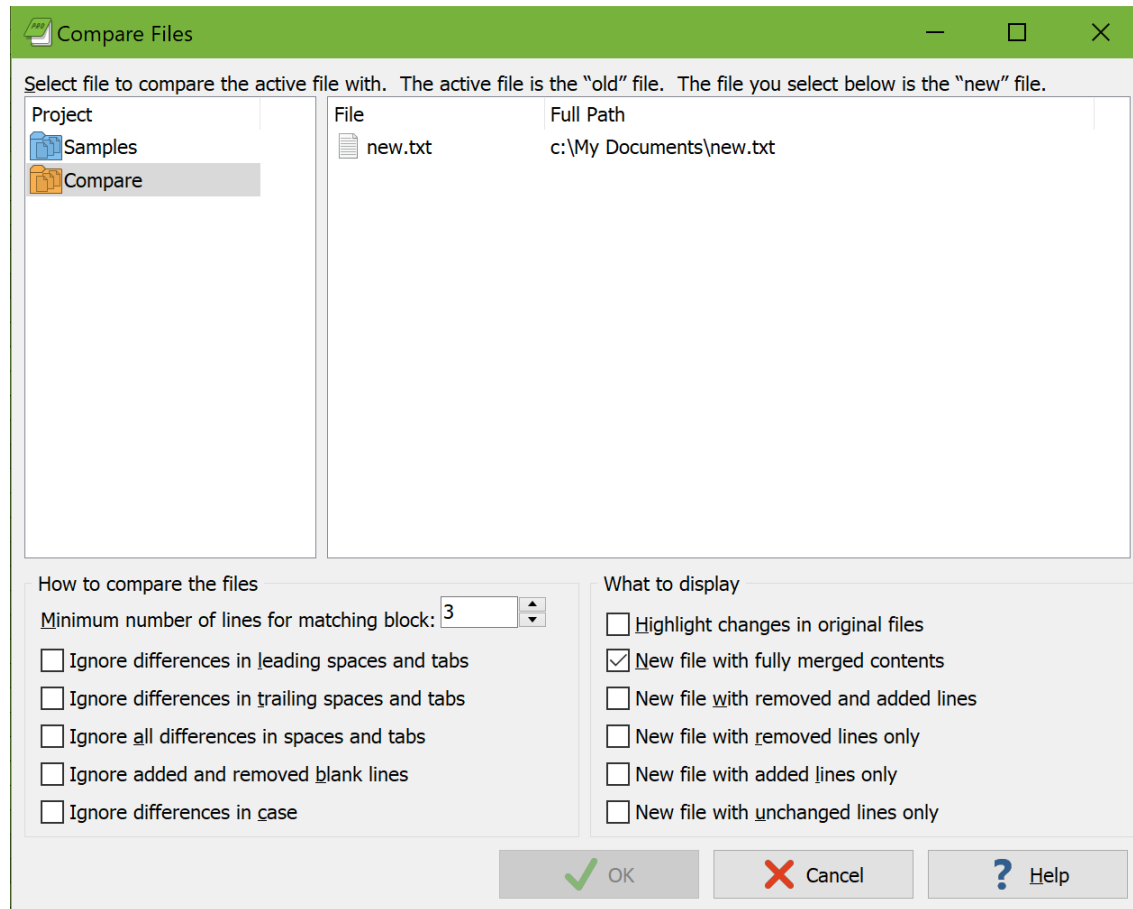
Turning on this option does not trim whitespace from lines that you don't edit. Any trailing whitespace that was already in the file when you opened it or that you added before turning on Auto Trim Trailing Whitespace remains in the file until you edit those lines. If you want to remove all trailing whitespace immediately, use Extra|Trim Trailing Whitespace. Then you can turn on Auto Trim Trailing Whitespace to keep the file free of trailing whitespace.

You can turn on Auto Trim Trailing Whitespace by default by ticking “trim trailing whitespace” on the Editor page in the file type configuration.

Extra|Compare Files

If you have two versions of the same text file, you can use Extra|Compare Files to visualize the differences between those files. After that, you can edit the generated difference file to merge both versions into a single, new file.

First you need to open the two files you want to compare. The Extra|Compare Files menu item is grayed out until you have two or more files open.



Selecting Files

To compare the files, first activate the file that contains the original or older version of the document by clicking on its tab. Then select **Extra|Compare Files** from the menu. In the window that appears, click on the file that contains the newer version of the document, in the list of files that are currently open. The active file is omitted from that list, since you cannot compare a file with itself.

The file that was active at the time you selected **Extra|Compare Files** in the menu is considered to be the “old” file. The file that you select in the file comparison options screen is considered to be the “new” file.

How to Compare The Files

You can specify several options when comparing files. The setting for the minimum match size may have a significant impact on the comparison. It is explained in detail in its own topic. 3 is a good default value.

You can choose to ignore differences in leading spaces and tabs. That is, when comparing two lines, EditPad Pro ignores any space and tab characters at the start of the lines. Similarly, you can choose to ignore trailing spaces and tabs. That is, spaces and tabs at the end of a line. You can mark both options to ignore both leading and trailing spaces and tabs. If you mark “ignore all differences in spaces and tabs”, then EditPad Pro ignores all spaces and tabs throughout both files when comparing their lines. These options are useful when comparing files where differences in whitespace have little or no meaning.

If you turn on “ignore added and removed blank lines”, EditPad Pro does not add blank lines that occur only in one of the two compared files to any new file created by the file comparison. Those lines are also not highlighted in the original files. Lines with only spaces and tabs are considered to be blank only if you’ve turned on one of the options to ignore differences in spaces and tabs.

Finally, “ignore differences in case” tells EditPad Pro to ignore whether a character is uppercase or lowercase, making “A” identical to “a”.

What to Display

If you select to highlight changes in the original files, EditPad Pro marks lines present in the “old” or “original” file in red when they are not present in the “new” or “edited” file. It also marks lines in the “new” file in green when they are not present in the “old” file. You can change these colors by customizing the color palette and editing the “Editor: Compare files: deleted line” and “Editor: Compare files: added line” colors.

If you turn on “new file with fully merged contents” then EditPad Pro creates a new tab with the two files merged, based on their differences. The tab is labeled “OldFile compared with NewFile”. Lines that are the same in both files are added once and displayed on a normal background. Lines only present in the “old” file are added on a red background. Lines only present in the “new” file are added on a green background. If the text on a line was changed between the “old” and “new” versions of the file then EditPad Pro first adds the old version of the line on a red background. After that it adds the new version of the line on a green background. If consecutive lines were changed then EditPad Pro first adds a block with the old versions of all the consecutive lines. After that it adds a block with the new versions of the consecutive lines.

“New file with removed and added lines” is the same as “new file with fully merged contents”, except that lines present in both the “old” and “new” files are not added. Thus all lines in this file are highlighted red and green. Consecutive changed lines are grouped in the same way. The tab is labeled “Differences between OldFile and NewFile”.

“New file with removed lines only” contains a tab with lines that are present in the “old” file, but not in the “new” file. Since all lines are old lines, none of them are highlighted. The tab’s label is “Lines from OldFile removed in NewFile”.

Similarly, “new files with added lines only” creates a tab labeled “Lines not in OldFile added to NewFile” with lines present in the “new” file, but not in the “old” file.

Finally, “new file with unchanged lines only” creates a tab labeled “Lines unchanged between OldFile and NewFile”. This tab shows all the lines present in both files.

All new tabs with difference output behave like regular files in EditPad Pro. You can use any editing command on them and save them for later use. The History panel can remind you of the two files that were compared. It shows you which was the “old” or “original” file and which was the “new” or “edited” file along with their color indicators.

Minimum number of lines for matching block

When you pick Extra|Compare Files from the menu, EditPad Pro asks for a file to compare the active one with. Near the bottom of the selection window, you will see the spinner box labeled “minimum number of lines for a matching block”.

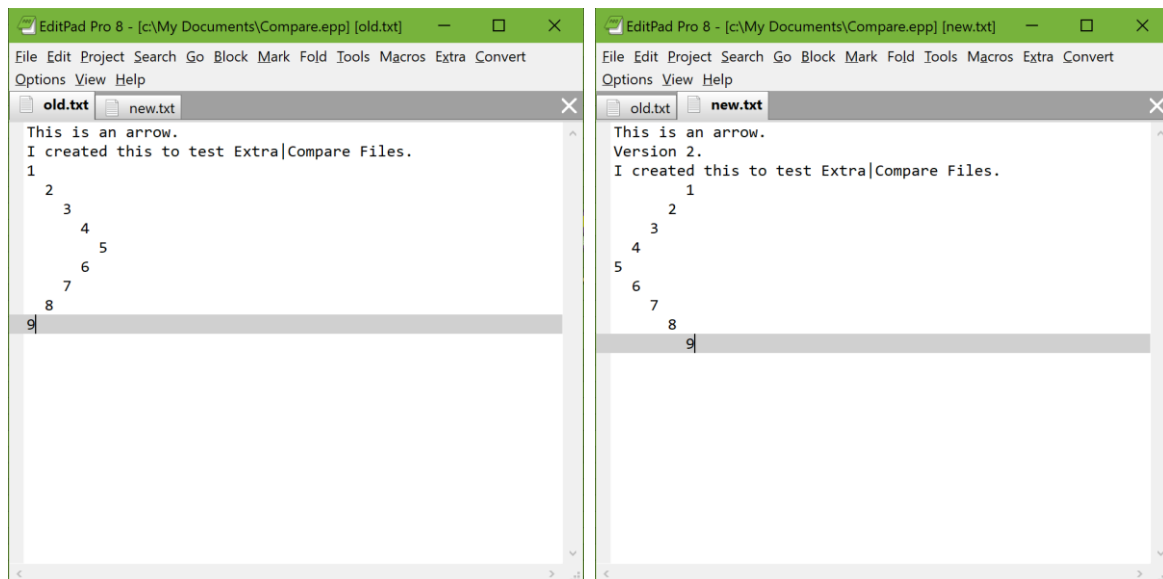
The default setting is 3. You can set it to any number from 1 up to and including 9.

This value is used by EditPad Pro in the following situation. While comparing the files, it encounters a section of one or more changed lines. In the difference output, these lines are grouped into a single red block showing the original lines, followed by a single green block with the new lines. This is done because all those lines are most likely connected somehow: a chapter in a book that was heavily edited, a source code routine that has been modified, etc. Keeping those lines together maintains their logical connection so you can easily see what happened when inspecting the output of Extra|Compare Files.

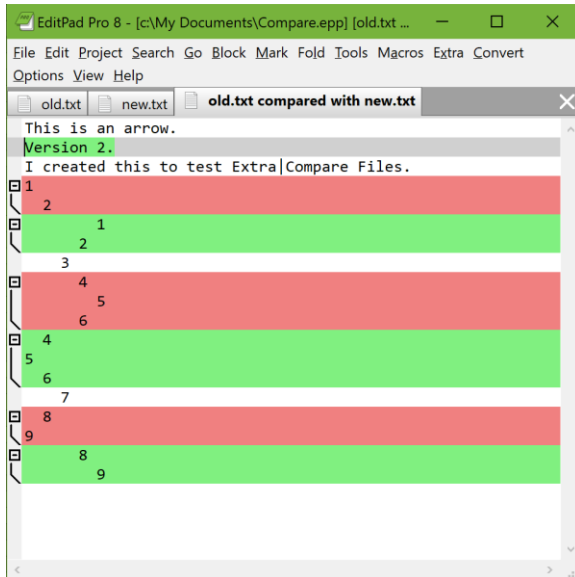
To even better maintain the logical structure of the document, you can have EditPad Pro also include a few lines in the block that are identical in both versions of the document. This happens when you set “minimum number of lines for a matching block” to a number greater than one (1).

The setting indicates the minimum number of consecutive lines that are identical in both files that are required for EditPad Pro to stop grouping the modified lines and add the identical lines as an unchanged block. The graphical illustrations below make this clear.

First you see the old and new version of our test document:



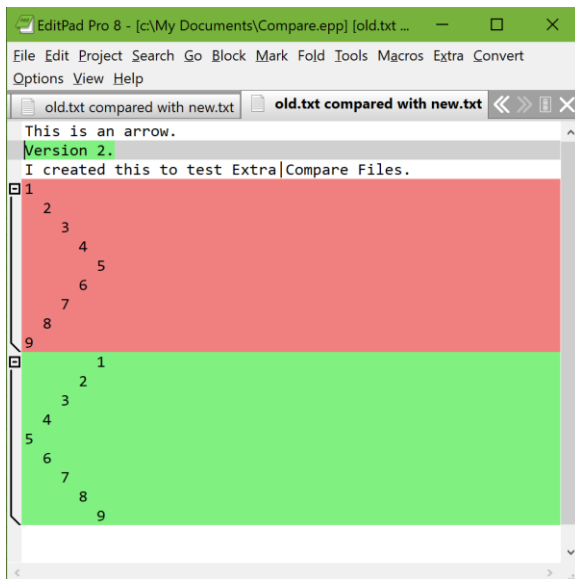
Then we use Options|Compare Files, and set “minimum number of lines for a matching block” to one, effectively turning off the feature. This is the result:



You see that EditPad Pro detects the line “Version 2” as being inserted.

It also detects that the lines numbered 3 and 7 are present in both files, and that the blocks 1-2, 4-6 and 8-9 have changed. This is technically exact. It is the optimum solution. But logically it makes no sense. The arrow is broken into pieces.

If we try again and set “minimum number of lines for a matching block” to anything greater than 1, we get the following result:



Much better! The “I created...” line is still included on its own since it does not follow a block of changed lines, but an inserted one.

However, the lines containing the numbers 3 and 7, even though they are present in both files, are displayed in the middle of the changed block as if they had been changed too. This way, our logical structure, the arrow,

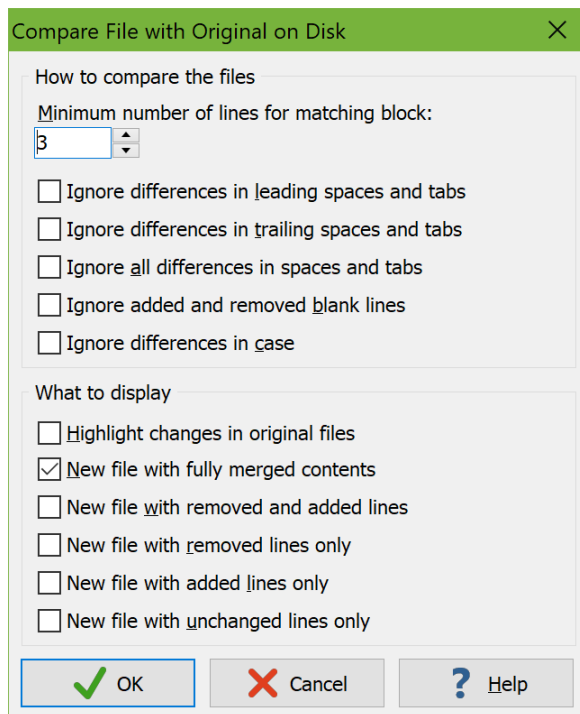
is kept together nicely and we can instantly see what happened: the arrow's direction was changed. This was not at all so clear in our first comparison attempt.

Since you will not be comparing arrows but real documents such as source code files, the “minimum number of lines for a matching block” can be configured for the task at hand. Many source code files contain lines with nothing but a curly brace or a “begin” or “end”. These lines add structure to the source, and not content. When comparing two versions of a piece of source, these lines are likely to be matched all over the place, breaking up the logical structure of the source code in the difference output.

If things don't look right, experiment with the “minimum number of lines for a matching block” setting.

Extra | Compare with File on Disk

The Compare with File on Disk item in the Extra menu works just like the Compare Files item, with one key difference. Whereas Compare Files asks you which file you want to compare the active file with, Compare with File on Disk compares the active file with the same file on disk. If you have made changes to a file in EditPad Pro and you haven't saved those changes yet, you can use Compare with File on Disk to see the changes that you made.



Extra | Next Comparison Mark

If you have used Extra | Compare Files or Extra | Compare with File on Disk, you can use Extra | Next Comparison Mark to find the block of differing paragraphs after the text cursor's position.

Extra | Previous Comparison Mark

If you have used Extra | Compare Files or Extra | Compare with File on Disk, you can use Extra | Previous Comparison Mark to find the block of differing paragraphs before the text cursor's position.

Extra | Clear Comparison Marks

After you've used Extra | Compare Files or Extra | Compare with File on Disk, the files you compared may have their differences highlighted. If you're done comparing the files but want to continue working with them in EditPad Pro, select Clear Comparison Marks in the Extra menu to remove the red and green highlights.

If you don't want to continue working with the files, simply close them. You don't need to remove the comparison marks first.

Extra | Statistics

Choose Statistics from the Extra menu to have EditPad Pro compute a bunch of statistics about the file you are currently editing. EditPad Pro creates a new tab to hold the statistics. An example:

```

Filename:
S:\JGsoft\EditPad8\Help\source\extrastatistics.txt
Last saved: 27-Sep-2019 15:03
File size: 3,268 bytes
Number of paragraphs: 38
Number of words: 493
Number of letters and digits: 2,325
Number of printable characters: 2,580
Total number of characters: 3,240
Selection size: 650 bytes
Selected words: 71
Selected letters and digits: 345
Selected printable characters: 377
Total number of selected characters: 650

```

The first two lines indicate under which name the file was saved, and when it was last saved. The third line indicates the size of the file in bytes. The total file size is the total number of characters in the file, including hidden characters such as line breaks.

If part of the file was selected before you picked Extra | Statistics from the menu, then the size in bytes of the selection is indicated as well.

“Number of paragraphs” indicates the number of paragraphs in the file. A paragraph is a sequence of characters terminated by pressing Enter on the keyboard, which inserts a hard return into the file. If you turn off word wrapping, then each paragraph is displayed as one line. The number of lines is not indicated in the statistics, because it varies with word wrapping.

“Number of words” indicates the number of sequences of “letters and digits”, separated by sequences of “non-letters”. A sequence consisting of a single letter or digit is also considered a word. A “letter or digit” is any character that is matched by the character class `\w` in regular expressions. This includes letters, digits, and

underscores in all scripts. You can test what EditPad Pro considers a letter by picking Search|Prepare to Search from the menu, typing `\w` into the search box, turning on the option “regular expressions“, and clicking the Search button. To search for words, use `\w+` (slash lowercase w plus) instead.

“Number of characters” indicates the number of printable characters. This includes all characters except spaces, tabs, line breaks and other control characters. “Total number of characters” counts all characters, including whitespace, line breaks, and control characters.

Extra | Project Statistics

Like Extra|Statistics, Project Statistics creates a new tab with statistics. It computes the same statistics, but for all files in the current project rather than the current file only. The order of the files in the statistics is the same order in which they are arranged in EditPad Pro’s tabs. If you want the files to be sorted alphabetically, use Go|Sort File Tabs Alphanumerically prior to using Extra|Statistics for All Files.

Extra | Statistics for All Files

Like Extra|Statistics, Statistics for All Files creates a new tab with statistics. It computes the same statistics, but for all files in all projects rather than the current file only. The order of the files in the statistics is the same order in which they are arranged in EditPad Pro’s tabs. If you want the files to be sorted alphabetically, use Go|Sort File Tabs Alphanumerically prior to using Extra|Statistics for All Files.

12. Convert Menu

Convert | Text Encoding

Computers deal with numbers, not with characters. When you save a text file, each character is mapped to a number, and the numbers are stored on disk. When you open a text file, the numbers are read and mapped back to characters. When saving a file in one application, and opening that file in another application, both applications need to use the same character mappings.

Traditional character mappings or code pages use only 8 bits per character. This means that only 256 distinct characters can be represented in any text file. As a result, different character mappings are used for different languages and scripts. Since different computer manufacturers had different ideas about how to create character mappings, there's a wide variety of legacy character mappings. EditPad supports a wide range of these.

In addition to conversion problems, the main problem with using traditional character mappings is that it is impossible to create text files written in multiple languages using multiple scripts. You can't mix Chinese, Russian and French in a text file, unless you use Unicode. Unicode is a standard that aims to encompass all traditional character mappings, and all scripts used by current and historical human languages.

How to Make a File Readable in EditPad

If you've received a text file from another person, or opened a file created on another computer, it may not immediately be readable in EditPad. Two things need to be set right. You need to use a font that can display the characters in your file. If you see hollow rectangles instead of characters or if characters are missing entirely then you are not using the correct font. You also need to use the correct encoding for the file so that EditPad knows which characters are represented by the bytes in the file. If you see incorrect characters (Chinese gibberish instead of English, for example) then you need to change the encoding.

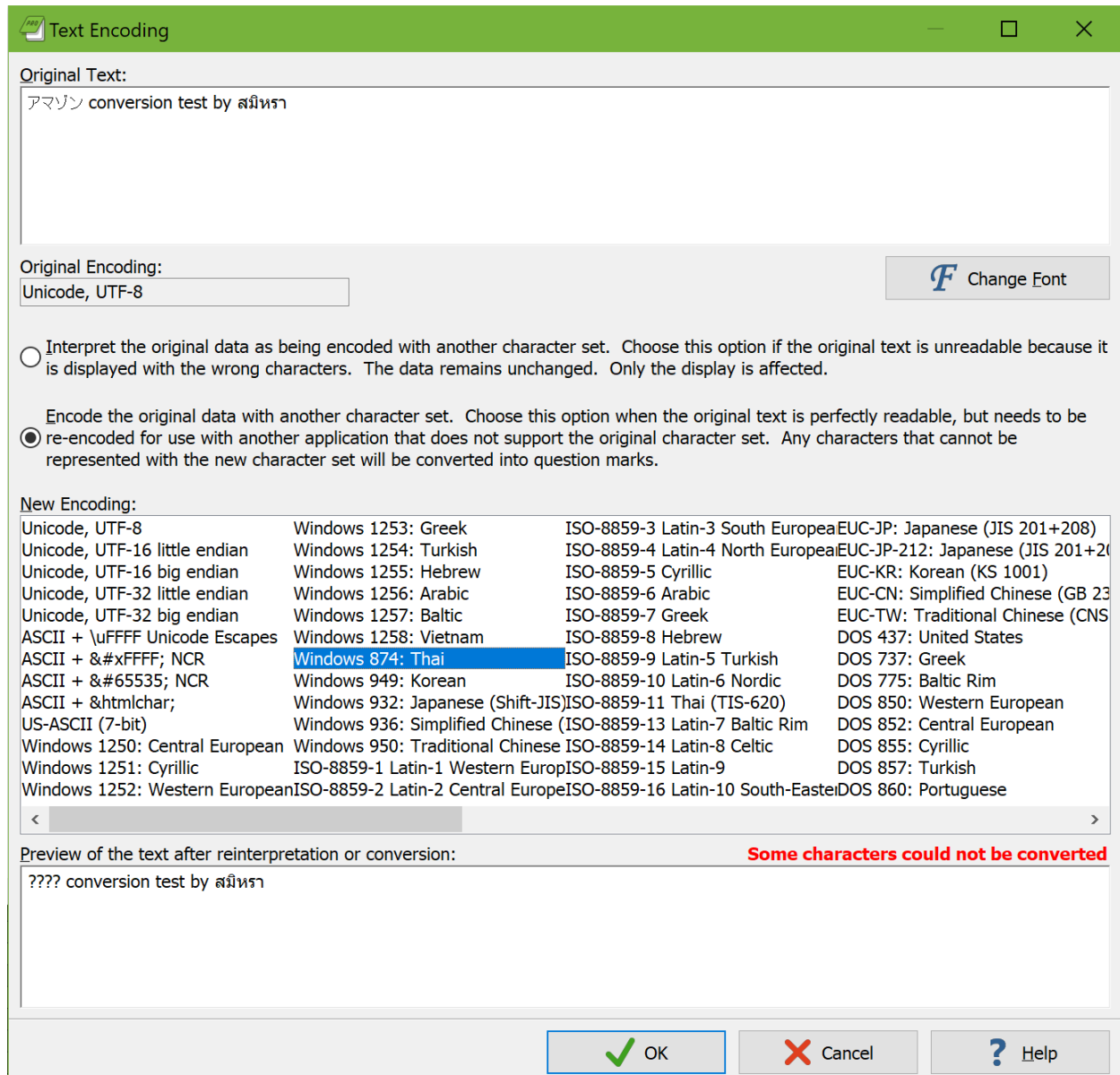
Select the Correct Font

While all fonts contain English characters, far fewer fonts contain Chinese, Thai, or Arabic characters. In EditPad, select Options | Font in the menu to select a font that supports the language your file is written in. Windows includes many different fonts tailored to specific languages or scripts.

If you use multiple scripts in a single file, then you probably won't have a single font that can (nicely) display all of those scripts. Fortunately, EditPad can use any number of fonts at the same time. It can automatically use each font only for those scripts that each font supports. Select Options | Text Layout in the menu. Set "text layout and direction" to "complex script". Select a main font and any number of fallback fonts. If the main font doesn't support a certain script, EditPad displays that script using the first fallback font that does support that script.

Select the Correct Encoding

If you see incorrect characters, select Text Encoding in the Convert menu to change the encoding EditPad uses for that file.



At the top of the screen, you will see part of the file as EditPad interprets it now, along with the encoding used. Make sure “**interpret the original data as being encoded with another character set**” is marked. Then try selecting a new encoding. The result appears immediately in the preview at the bottom of the screen. Keep trying different encodings until you find one that produces a readable file.

If the file was created on a computer running Windows, try the Windows encodings first. All Windows computers use one of the Windows encodings as the default. UTF-8 and UTF-16 little endian are also likely, as Unicode is becoming popular among modern Windows applications.

If the file was created on a computer running a UNIX variant such as Linux, try the ISO-8859 and EUC encodings. UTF-8 and UTF-16 little endian are also likely.

If the file was created on a modern Mac running OS X, it probably uses UTF-8. If it was created on an older Mac, try the Mac encodings.

If the file was created by an old DOS application, try one of the DOS character sets. The DOS character sets were used by Microsoft's MS-DOS and DOS versions from other companies. If you know that the file is supposed to contain "line drawing symbols", the DOS character sets are also very likely. The DOS character sets are the only ones that contain line drawing symbols. DOS applications used characters that look like lines, bevels and corners to draw pseudo-graphical interfaces on character-based screens. DOS predates Unicode, so the Unicode formats are unlikely, even if they contain line drawing symbols.

If the file is written in Russian or Ukrainian, the KOI8-R and KOI8-U encodings are very likely candidates, even for files created on Windows or UNIX systems. Particularly ISO-8859-5 has never reached the popularity of KOI8.

If the file was created on an old mainframe or an IBM AS/400 (renamed iSeries) system, try the EBCDIC encodings. EBCDIC was the de facto standard in the days computers used punch cards. If EBCDIC doesn't produce a readable file, try the DOS character sets, which were used by IBM's PC-DOS.

Non-Representable Characters Replaced with Question Marks

There are two ways in which a text editor can keep files in memory while editing them. Some editors use Unicode internally. On Windows that is typically UTF-16 LE. When you open a file that uses any other encoding, it is converted into UTF-16 LE in memory. When you save the file, it is converted back into the other encoding. The benefit is that the developers of such editors have only one encoding to deal with for all editing functions. The downside is that the conversion takes extra time and extra memory. Such editors usually don't perform well with very large files. If a file is loaded with the wrong encoding, it has to be reloaded with the correct one. If you don't notice the wrong encoding is being used, or if it contains bytes that are invalid for the encoding that the file actually uses, data loss may occur. There's no way to preserve invalid byte sequences when converting to UTF-16.

When EditPad loads a file, it loads its actual bytes into memory. In EditPad Pro you can even see those bytes by switching to hexadecimal mode. That is what the term "original data" in the choices in the Convert|Text Encoding window refers to. Because EditPad keeps the file's original bytes in memory, it can instantly change how it interprets those bytes. Simply use the "interpret" option in the Text Encoding window and select another encoding. This option does not change the contents of the file at all, nor is there any need to reload the file. It only changes how EditPad translates the bytes into characters.

While you edit a file, EditPad converts those bytes into characters on-the-fly for display. When you type in or paste in new text, EditPad immediately converts the characters you enter into the appropriate bytes for the file's encoding. That means that if your file doesn't use Unicode, you can only enter or paste characters that are supported by the file's encoding. EditPad's Character Map can show you all those characters.

If you paste characters that are not supported by the file's encoding, EditPad has no way to convert those characters into bytes to store them into the file. Such characters are lost. They are permanently changed into **question marks** to indicate the actual characters you tried to paste could not be represented. In order to paste the actual characters, first use Edit|Undo to remove the question marks. Then use Convert|Text

Encoding with the “encode original data with another character set” option. Select a Unicode transformation or any encoding that supports the characters you want to paste, as well as those already present in the file. EditPad then changes the bytes in the file to represent the same characters in the new encoding. Now you can paste your text again and get the actual characters. Note that you have to undo pasting the question marks. Changing the encoding does not magically restore the characters. Since EditPad Pro uses the file’s actual encoding for in-memory storage rather than Unicode, newly entered or pasted characters that cannot be represented cannot be stored.

How to Make an EditPad File Readable by Others

To make sure other people can read files you’ve written in EditPad, simply save it in an encoding that the other person can read. If he or she also uses Windows, you don’t need to do anything. EditPad’s default text encoding settings save the file in your computer’s default Windows code page.

If the file is written in English, you also have little to worry about. English text is encoded the same way in UTF-8, all Windows code pages, all ISO-8859 code pages, all DOS code pages, all Mac code pages, and also KOI8.

If your document uses non-English characters, and you’re not sure which encodings the other person can read, the UTF-8 encoding is a safe bet. UTF-8 files usually start with a byte order marker to identify them. EditPad and many other applications running on Windows, Linux and Mac OS X detect the byte order marker, and automatically interpret the file as UTF-8. Since UTF-8 is a Unicode transformation, it supports all modern human languages. All characters present in any of the non-Unicode code pages supported by EditPad are also present in the Unicode mapping.

To change a file’s encoding, select Text Encoding in the Convert menu. Mark the “**encode original data with another character set**” option and select the encoding you want to convert the file into. If you get a bold red warning that some characters could not be converted, this means that the encoding you are trying to convert the file into cannot represent some characters you’ve used in the file. Those characters will be replaced by question marks if you proceed with the conversion. The Unicode encodings are the only ones that can represent all characters from all human languages. The others are typically limited to English plus one language (e.g. Chinese) or one group of languages (e.g. Western European languages).

Convert | Line Break Style

At the end of each line there is a character or a pair of characters that terminates the line. This character or pair of characters is known as a line break. Mainly due to historic reasons, there are eight different types of line breaks that you may encounter in text files from different sources. EditPad supports all eight equally. So in EditPad the line break style doesn’t really matter. But it may matter if you need to use your files with other applications or in other environments. Many applications only support the one line break style that is native to their platform.

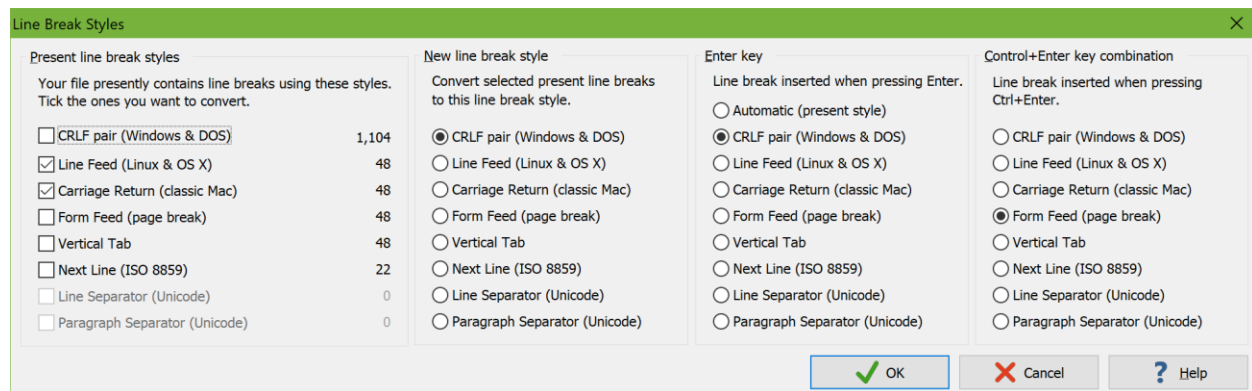
These are the eight line break styles:

- Carriage return and line feed pair: This is the only line break style that consists of two characters. It is the native line break style of DOS and Windows. It is represented by `\r\n` in regular expressions.

- Line feed: A line feed character on its own is the native line break style on UNIX and derivatives such as Linux and OS X. It is often referred to as a “newline” character on these platforms. It is represented by `\n` in regular expressions.
- Carriage return: A carriage return on its own was the native line break style on classic Mac systems (prior to OS X). It is represented by `\r` in regular expressions.
- Form feed: The form feed control character tells printers to advance to the next page. EditPad treats it as a page break. It appears as a horizontal line while editing. It ends a page when printing. It is represented by `\f` in regular expressions.
- Vertical tab: The vertical tab control character was historically used to add vertical space between printed lines. Some modern applications use the vertical tab as an alternative line break to allow line breaks within line-based data or to distinguish between a line break that ends a paragraph and one that does not. In EditPad a vertical tab is just another line break. You can use `\x0B` to represent it in EditPad’s regex flavor. It is represented by `\v` in C-style languages and some regex flavors.
- Next line: Control character from the ISO-8859 encodings. You can use `\u0085` to represent it in a regular expression.
- Line separator: Unicode character to indicate the end of a line. It is represented by `\p{Zl}` in regular expressions.
- Paragraph separator: Unicode character to indicate the end of a paragraph. It appears as any other line break in EditPad. It is represented by `\p{Zp}` in regular expressions.

To actually see the line breaks in your file, turn on Options | Visualize Line Breaks. On the Editor page in the Preferences you can choose how line breaks are visualized. On the Status Bar page you can turn on a status bar indicator for the line break style.

Select Line Break Style in the Convert menu if you want to convert existing line breaks to a different line break style or if you want to determine the style of new line breaks. Clicking the line break style status bar indicator also invokes this command.



The leftmost column in the Line Break Style dialog box shows you a complete count of all the line break styles that your file presently contains. If you want to convert certain line breaks into a different style, tick the line breaks you want to convert in the leftmost column. When you have ticked one or more line break styles to be converted, you can select the line break style you want to convert them to in the second column. If the line break style that you want is grayed out in the second column, untick it in the first column. The “next line” and Unicode line break styles may be permanently grayed out if they are not supported by your file’s encoding.

Regardless of whether you have ticked any line break styles in the left most column, you can use the third and fourth columns to select the kind of line break that is inserted when you press Enter or Ctrl+Enter on the

keyboard. Line break styles not supported by your file's encoding are grayed out. Changing the line break style for Enter or Ctrl+Enter only affects future line breaks inserted by the Enter key.

Quick Conversion to Windows or UNIX

The Line Break Style item in the Convert menu has a submenu with items that allow you to quickly convert a file to Windows or UNIX line breaks without having to go through the dialog box. The Windows option converts LF-only and CR-only line breaks to CRLF and make the Enter key insert CRLF pairs. You could achieve the same via the dialog box with the settings as shown in the screen shot above. The UNIX option converts CRLF pairs and CR-only line breaks to LF-only line breaks. If you plan to upload a file via EditPad Pro's built-in FTP to a Linux server, you should convert it to the UNIX line break style first.

Convert | Case | Adjust Case

On the Colors and Syntax page in the file type configuration you can select a syntax coloring scheme for each file type. A syntax coloring scheme can apply case conversion rules to specific parts of the text. The HTML scheme included with EditPad, for example, specifies that HTML tags should be lowercase.

The Adjust Case item in the Case submenu of the Convert menu is only enabled when the syntax coloring scheme used for the active file specifies such case conversion rules. Click it to convert the selected text according to the scheme's case conversion rules. If there is no selection, the command converts the case of the line that the cursor is on.

If you'd like this conversion to happen automatically as you enter text, turn on Convert | Case | Auto Adjust Case.

Convert | Case | Auto Adjust Case

On the Colors and Syntax page in the file type configuration you can select a syntax coloring scheme for each file type. A syntax coloring scheme can apply case conversion rules to specific parts of the text. The HTML scheme included with EditPad, for example, specifies that HTML tags should be lowercase.

The Auto Adjust Case item in the Case submenu of the Convert menu is only enabled when the syntax coloring scheme used for the active file specifies such case conversion rules. This item is a toggle that you can turn on and off. When on, case conversion rules are applied automatically to any text you enter when you move the cursor away from the line you entered the text on. This delay ensures that Auto Adjust Case does not incorrectly adjust partially entered words. The Delphi scheme, for example, converts keywords such as "begin" to lowercase. But if you enter "Begin" as part of a longer identifier such as "BeginOperation()" then no conversion should take place. By holding off the conversion until you move the cursor away from the line, EditPad ensures you do not end up with "beginOperation()".

On the Editor page in the file type configuration you can toggle the "adjust case" checkbox to turn Auto Adjust Case on or off by default.

If you want to apply this conversion to text that is already present in the file or to text that you have pasted, use the **Convert | Case | Adjust Case** command. You can use this command regardless of whether **Auto Adjust Case** is on or off.

Convert | Case | Uppercase

Select **Uppercase** in the **Case** submenu of the **Convert** menu to convert all lowercase letters in the selected text to uppercase letters. If there is no selection, the line the cursor is on is converted to uppercase.

Convert | Case | Lowercase

Select **Lowercase** in the **Case** submenu of the **Convert** menu to convert all uppercase letters in the selected text to lowercase letters. If there is no selection, the line the cursor is on is converted to lowercase.

Convert | Case | Initial Caps

Select **Initial Caps** in the **Case** submenu of the **Convert** menu to convert the first letter of each word in the selected text to uppercase and all other letters to lowercase. If there is no selection, the line the cursor is on is converted to initial caps.

Convert | Case | Invert Case

Turns all uppercase letters in the selection into lowercase letters, and all lowercase letters into uppercase. If there is no selection, the line the cursor is on is inverted.

If you have been typing with **Caps Lock** on, you do not have to start over. Let this function come to the rescue.

Convert | Quotes | Adjust Quotes

On the **Colors and Syntax** page in the file type configuration you can select a syntax coloring scheme for each file type. A syntax coloring scheme can specify that certain straight quotes should be converted into smart quotes or that certain smart quotes should be converted into straight quotes.

The **Adjust Quotes** item in the **Quotes** submenu of the **Convert** menu is only enabled when the syntax coloring scheme used for the active file specifies such quote conversion rules. Click it to convert the selected text according to the scheme's quote conversion rules. If there is no selection, the command converts the quotes on the line that the cursor is on.

The syntax coloring scheme specifies which straight quotes can be converted to smart quotes and which straight quotes that smart quotes can be converted into. This allows the syntax coloring scheme to ensure that characters that are part of the file's syntax are not converted. The **HTML** scheme, for example, specifies that straight single and double quotes within the body text should be converted to smart quotes. It does not allow

any quote conversion in HTML tags because those use quotes to delimit attribute values. Thus when you use Adjust Quotes on an HTML file, EditPad converts the quotes that the reader will see into smart quotes, while leaving the quotes that are part of the HTML syntax alone.

EditPad Pro can convert single quotes, double quotes, backticks, and less-than and greater-than signs into smart quotes. It can convert smart quotes back to those characters. But the syntax coloring scheme can restrict the quote conversion to only some of these characters. The Delphi language, for example, uses single straight quotes to delimit strings. The Delphi syntax coloring scheme allows double straight quotes to be converted to smart quotes in strings. It does not allow single straight quotes to be converted because that would break the string. If you do want to have smart single quotes inside a Delphi string, you can select the single straight quotes you want to convert and use Convert|Quotes|Straight ⇒ Smart Quotes to force the conversion.

The syntax coloring scheme does not specify the style of smart quotes that should be used. You can choose that via Convert|Quotes|Quotation Style. It uses the settings in the same way as Convert|Quotes|Straight ⇒ Smart Quotes and Convert|Quotes|Smart ⇒ Straight Quotes. The settings are described in detail in those topics. If the syntax coloring scheme does not allow certain characters to be converted, then any options using them are implicitly turned off.

In the Quotation Style dialog you can also tick “replace smart quotes before adjusting quotes”. If you do then Adjust Quotes does what Convert|Quotes|Replace Smart Quotes does before converting between straight and smart quotes according to the syntax coloring scheme. This can be useful if you’re pasting text that may contain straight quotes or smart quotes in a different style that you want to convert to your chosen smart quotation style. Then you need only use Adjust Quotes to do both conversions.

If you’d like the quote conversion based on the syntax coloring scheme to happen automatically as you enter text, turn on Convert|Quotes|Auto Adjust Quotes.

Convert|Quotes|Auto Adjust Quotes

The Auto Adjust Quotes item in the Quotes submenu of the Convert menu is a toggle that you can turn on and off. You can toggle it on or off by default using the “adjust quotes” checkbox in the on the Editor page in the file type configuration.

When on EditPad automatically adjusts straight and smart quotes in any text you enter when you move the cursor away from the line you entered the text on. This delay ensures that EditPad Pro can correctly distinguish between single quotes and apostrophes. It also ensures that the syntax coloring scheme can match up opening and closing quotes when the line is complete before any conversion is done as that may affect whether those quotes should be converted or not.

If your file uses a syntax coloring scheme that has rules for adjusting quotes then the conversion done by Auto Adjust Quotes is exactly the same as that done by Convert|Quotes|Adjust Quotes. It is explained in detail in that topic. The toolbar icon for Auto Adjust Quotes reflects this by showing a golden straight and a golden smart quote with a bidirectional arrow in between like the Adjust Quotes icon, but with an added pencil to indicate automatic editing.

If your file does not use a syntax coloring scheme or it does not have any rules for adjusting quotes then Auto Adjust Quotes converts straight quotes that you enter into smart quotes in the same way as Convert|Quotes|Straight ⇒ Smart Quotes. The toolbar icon for Auto Adjust Quotes reflects this by

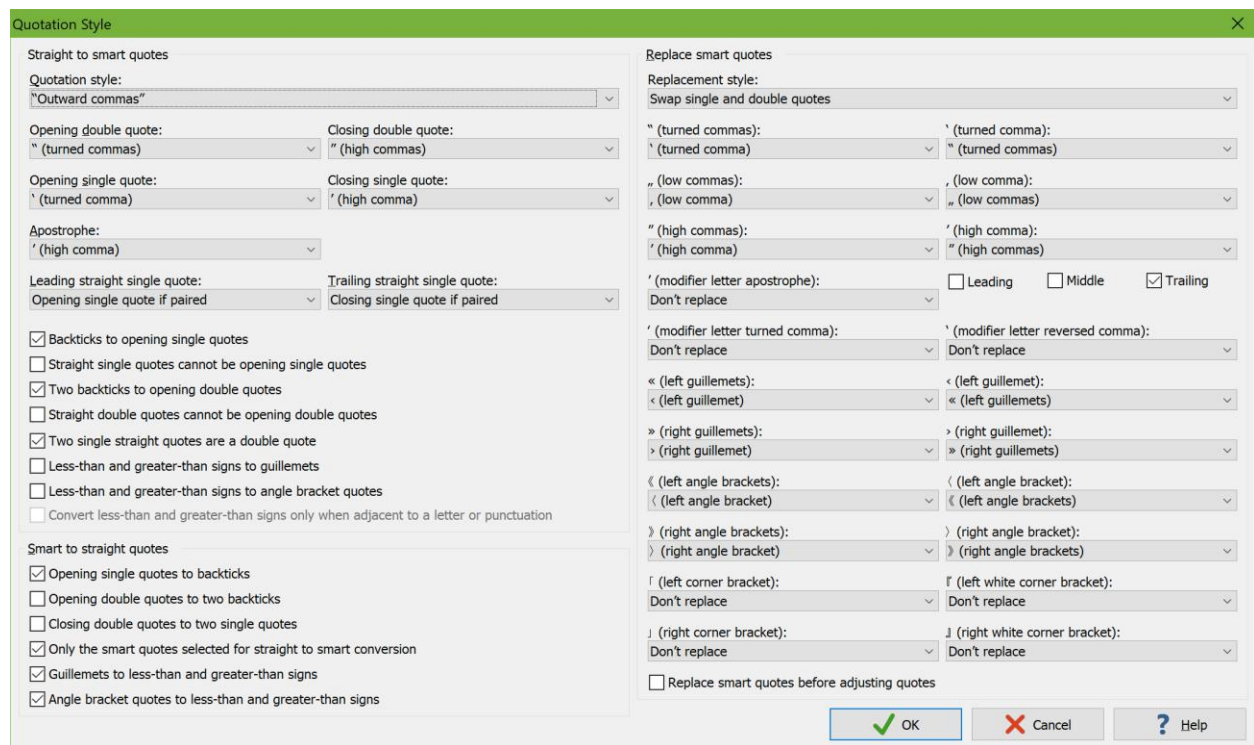
showing a red straight and a green smart quote with a unidirectional arrow in between like the Straight ⇒ Smart Quotes icon, but with an added pencil to indicate automatic editing.

You can use the Convert|Quotes|Adjust Quotes or Convert|Quotes|Straight ⇒ Smart Quotes command to apply this conversion to text that is already present in the file or to text that you have pasted. These two commands are available regardless of whether Auto Adjust Quotes is on or off.

Convert|Quotes|Straight ⇒ Smart Quotes

Select Straight ⇒ Smart Quotes in the Quotes submenu of the Convert menu to convert straight quotes in the selected text into smart quotes. If there is no selection, the quotes on the line that the cursor is on are converted.

Exactly which straight quotes are converted into exactly which smart quotes depends on the choices you made in Convert|Quotes|Quotation Style. The Straight ⇒ Smart Quotes command uses the settings in the “straight to smart quotes” group in the upper left corner of that dialog. It does not use any of the other settings.



The “**quotation style**” drop-down list is a speed setting. Selecting a style changes the 5 different quotation characters that you can choose. Pretty much all quotation styles used around the world are available in EditPad Pro. You can make a custom style by selecting quotation characters from the drop-down lists for double quotes, single quotes, and apostrophes.

EditPad is smart enough to distinguish between apostrophes and single quotes. The sentence “In the ’90s we gave ’em hell.” has two apostrophes and zero single quotes. An apostrophe is not a quote, even if it looks like

one, but a character that indicates something was omitted. In American typography, the apostrophe should look like a high comma or a tiny 9 (which is also used for closing single quotes). The apostrophe should not look like a turned comma or a tiny 6 (which is only used for opening single quotes). This is true despite the fact that word processors from major American software companies get this wrong.

If you enter the example sentence “In the ‘90s we gave ‘em hell.” with two straight single quotes then EditPad Pro is smart enough to realize that they are not a pair of single quotes but rather two apostrophes. EditPad Pro only converts a single straight quote into an opening quote if there is a matching closing single quote. This is also the reason why Convert|Quotes|Auto Adjust Quotes converts quotes only after you’ve moved away from the line. EditPad Pro waits to see if any straight quote you enter will end up being paired or alone before deciding how to convert it.

To enable EditPad Pro’s automatic differentiation between single quotes and apostrophes, set “**leading straight single quote**” to “opening single quote if paired” and “**trailing straight single quote**” to “closing single quote if paired”. Then unpaired single quotes are converted into apostrophes. If your language has different rules for apostrophes or no apostrophes at all, you can change these to always be an apostrophe or always be an opening or closing single quote.

Exactly what your smart quotes look like also depends on the font you are using to display the file, of course. Some fonts render the turned comma (which looks like a tiny number 6) as a reversed comma (which looks like a mirror image of a tiny 9). But this does not affect the discussion of apostrophes versus opening single quotes.

In English text, the “high comma” character is normally used for both closing single quotes and for apostrophes. The Unicode standard recommends this. It is the default in EditPad. One issue with this is that the high comma is a punctuation character. By default, EditPad does not consider it to be part of a word. Double-clicking “isn’t” selects “isn” or “t” but not “isn’t”. You could change this by turning on the “other punctuation” option in the text layout configuration. But then any punctuation around a word is selected when you double-click it. The regex `\w` for matching a word character never matches the high comma.

If you prefer your apostrophes to be word characters, you can select “modifier letter apostrophe” as your smart quote. Most fonts display this character identically to the high comma. But it is a different character that is in the “modifier letter” Unicode category. This category contains various symbols that may not be letters but should act as letters. EditPad always considers them to be part of a word. You will need to save your files as Unicode since the legacy Windows code pages do not support the modifier letter apostrophe.

Some languages don’t use apostrophes but other characters with a similar function. You can select these in EditPad too. The “modifier letter turned comma” marks the phonemic glottal stop in many Polynesian languages, such as the ‘okina in Hawai‘i. The “modifier letter reversed comma” is used in languages such as Armenian.

In some circles it is customary to use backticks (`) as opening single quotes. On a US keyboard, the backtick can be found to the left of the 1 key in the upper left corner of the keyboard. If you use this style, check “backticks to opening single quotes” to have EditPad Pro interpret backticks as opening quotes. Backticks are never interpreted as apostrophes. If you exclusively use backticks to enter opening quotes then you should check “**straight single quotes cannot be opening single quotes**”. Then EditPad Pro knows that a straight single quote is either an apostrophe or a closing quote.

There is a similar custom of using two backticks (`) as opening double quotes. Check **“two backticks to opening double quotes”** if you use this style. Also check **“straight double quotes cannot be opening double quotes”** if you use the style with double backticks exclusively.

People who use two backticks as opening double quotes sometimes use two single straight quotes (‘'') rather than a double straight quote (") as closing double quotes. If you do this, check **“two single straight quotes are a double quote”**. If you don’t, EditPad Pro interprets two single straight quotes as two single quotes.

You have two ways of adding «guillemets» or «angle brackets quotes» to your text files in EditPad. One way is to set “quotation style” to one of the guillemet or angle bracket options. Then any straight quotes that you enter are converted into guillemets or angle bracket quotes. This can be useful when working with file formats like HTML where less-than and greater-than signs are part of the syntax and shouldn’t be converted. The other way is to check **“less-than and greater-than signs to guillemets”** or **“less-than and greater-than signs to angle bracket quotes”**. Then you can enter < and > on the keyboard and have them converted into ‹ and › or ‹ and ›. If your text may also include actual less-than and greater-than signs then check **“convert less-than and greater-than signs only when adjacent to a letter or punctuation”**. Then < and > are not converted when surrounded by spaces as they would typically be in a mathematical formula.

Do not confuse a guillemet with a guillemot. A guillemet is a quotation character. A guillemot is a species of bird.

Convert | Quotes | Smart ⇒ Straight Quotes

Select Smart ⇒ Straight Quotes in the Quotes submenu of the Convert menu to convert smart quotes in the selected text into straight quotes. If there is no selection, the quotes on the line that the cursor is on are converted.

Exactly which smart quotes are converted into exactly which straight quotes depends on the choices you made in Convert | Quotes | Quotation Style. The Smart ⇒ Straight Quotes command uses the settings in the “smart to straight quotes” group in the lower left corner of that dialog. Some of these settings refer to the five quotation characters that you can select in the “straight to smart quotes” group in the upper left corner. No settings other than those five quotation characters from the upper left group are used by Smart ⇒ Straight Quotes.

Check **“opening single quotes to backticks”** if you want opening single smart quotes to be converted to backticks (`) instead of single straight quotes. This produces the ASCII quotation style of using a backtick as the opening single quote and a straight quote as the closing single quote. This rule is applied to the character you select in the “opening single quote” drop-down list if you did not select the same character in the “closing single quote” drop-down list. The character you select in the “closing single quote” drop-down list is never converted into a backtick. The high comma (`) is always converted into a single straight quote if you selected it in the “apostrophe” drop-down list.

Other characters may be converted into backticks if you did not select them in either of the “opening single quote” and “closing single quote” drop-down lists. This always applies to turned commas (`) and low commas (,). High commas (`) are converted into backticks if you did not select the high comma in the “apostrophe” drop-down list. Single left guillemets («) and single left angle brackets (<) are converted into backticks if you did not turn on the option to convert them into less-than signs.

Check **“opening double quotes to two backticks”** if you want opening double smart quotes to be converted to two backticks (``) instead of one double straight quote. This rule is applied to the character you select in the “opening double quote” drop-down list if you did not select the same character in the “closing double quote” drop-down list. The character you select in the “closing double quote” drop-down list is never converted into a backtick.

Other characters may be converted into two backticks if you did not select them in either of the “opening double quote” and “closing double quote” drop-down lists. This always applies to double turned commas (``) and double low commas (,,). Double left guillemets (««) and double left angle brackets (<<) are converted into two backticks if you did not turn on the option to convert them into less-than signs.

Checking **“closing double quotes to two single quotes”** converts closing double smart quotes into two single straight quotes instead of one double straight quote. This option is best used in combination with

“opening double quotes to two backticks”. Together they produce the ASCII quotation style of using two backticks as opening quotes and two single straight quotes as closing quotes. Some people find this style more balanced than using two backticks for the opening quote and one double straight quote for the closing quote. With a monospaced font, two backticks or two single straight quotes are twice as wide as one double straight quote.

The character you select in the “closing double quote” is converted into two single straight quotes if you did not select the same character in the “opening double quote” drop-down list. If you select the same character in both drop-down lists then it is always converted into straight double quotes.

Other characters may be converted into two single straight quotes if you did not select them either of the “opening double quote” and “closing double quote” drop-down lists. This always applies to double high commas (”). Double right guillemets (») and double right angle brackets (») are converted into two single straight quotes if you did not turn on the option to convert them into greater-than signs.

Check **“only the smart quotes selected for straight to smart conversion”** to convert only the characters that you selected in the five drop-down lists in the “straight to smart quotes” group into straight quotes. Otherwise, all of the characters that are available in those drop-down lists are converted into straight quotes. The sole exception are corner brackets. Those are only converted into straight quotes if you selected them in the drop-down lists.

Check **“guillemets to less-than and greater-than signs”** to convert left guillemets into less-than signs and right guillemets into greater-than signs. Double guillemets are converted into two less-than or greater-than signs. If you don’t check this option then guillemets may be converted into straight quotes and/or backticks or be left alone depending on the above options.

Check **“angle brackets to less-than and greater-than signs”** to convert left angle bracket quotes into less-than signs and right angle bracket quotes into greater-than signs. Double angle bracket quotes are converted into two less-than or greater-than signs. If you don’t check this option then angle bracket quotes may be converted into straight quotes and/or backticks or be left alone depending on the above options.

You can convert guillemets and angle bracket quotes into less-than and greater-than signs at the same time.

Convert | Quotes | Replace Smart Quotes

Select Replace Smart Quotes in the Quotes submenu of the Convert menu to convert the selected text from one style of smart quotes into another style of smart quotes. If there is no selection, the quotes on the line that the cursor is on are converted.

Before you can use this command, you need to use Convert | Quotes | Quotation Style to specify which smart quotes should be replaced with which other smart quotes. If you forget, the Replace Smart Quotes command pops up a message box to remind you. Replace Smart Quotes only uses the settings in the right-hand side of the Quotation Style dialog box.

The “replacement style” drop-down list is a speed setting. Selecting something in this drop-down list changes all of the other drop-down lists. The provided replacement styles all convert from one common smart quotation style to another smart quotation style.

You can replace specific quote characters with another quote character by selecting the replacement character from the drop-down list of the character you want to replace. You can select the same character from multiple drop-down lists to replace all those characters with the same character. You can select “don’t replace” to leave certain smart quotes alone.

The high comma (’) drop-down list is accompanied by three check boxes. Check “leading” to convert high commas when they occur before a word, “middle” when they occur within a word, and “trailing” when they occur after a word. This allows you to selectively replace high commas depending on whether they are used as quotation characters or as apostrophes. All other characters are replaced wherever they occur.

The option “replace smart quotes before adjusting quotes” is not used by Convert|Quotes|Replace Smart Quotes. That option only affects Convert|Quotes|Adjust Quotes and Convert|Quotes|Auto Adjust Quotes.

Convert|Quotes|Quotes ⇒ Primes

Select Quotes ⇒ Primes in the Quotes submenu of the Convert menu to convert all quotes in the selected text into primes. If there is no selection, the quotes on the line that the cursor is on are converted.

The prime (′) is the correct symbol to use to indicate units such as feet or arcminutes. The double prime (″) represents inches and arcseconds. Triple primes (″″) represent a ligne (1/12 of an inch) or a third (1/60 of a second). Quadruple primes represent a fourth (1/60 of a third). In mathematics, primes are used to indicate derivatives. In music, primes represent octaves. The middle C is c′, while c″ is one octave above c′ and c″″ is two octaves higher.

5′8″ (using primes) is the correct way to write 5 feet 8 inches. 5'8″ is how you would enter 5 feet 8 inches on a keyboard. You’d have to leave it as that in a plain ASCII text file. 5’8” (using high commas) is nonsense produced by (automatic) straight to smart quote conversion.

Depending on the font you’re using, the difference between a prime (′) and a high comma (’) can be subtle. The prime is straight while the high comma is curved like a little 9. Even if your font doesn’t give high commas much curvature, you should take care to use the correct symbol. If you switch fonts later, the difference can be more glaring. Arial is a font that makes primes and quotes look rather different. If somebody uses Convert|Quotes|Replace Smart Quotes to change the quotation style later, high commas may be replaced but primes will stay as primes.

The Quotes ⇒ Primes command converts straight quotes, high commas (’ and ”), turned commas (‘ and “), and all the characters you selected in the five “straight to smart quotes” drop-down lists in Convert|Quotes|Quotation Style into primes. This ensures that the straight quotes you enter on the keyboard are converted into primes, even if those straight quotes were already converted into smart quotes by Convert|Quotes|Auto Adjust Quotes.

Individual single quotes are converted into single primes. Two consecutive single quotes and double quotes are converted into double primes. Three consecutive single quotes, a single and double quote pair, and a double and single quote pair are converted into triple primes.

Quotes ⇒ Primes does not produce quadruple primes. Most fonts included with Windows do not support them. Two consecutive double quotes become two double primes. Four consecutive single quotes become a triple prime followed by a single prime.

Quotes ⇒ Primes converts backticks (`) into reverse primes. This only works if you did not allow Convert|Quotes|Auto Adjust Quotes to convert backticks into smart quotes first. Since Auto Adjust Quotes waits until you move away from the line you were editing, you can use Quotes ⇒ Primes immediately after entering the backticks to convert them into primes. Many fonts included with Windows do not support reverse primes, however.

Convert|Quotes|Quotation Style

Select Quotation Style in the Quotes submenu of the Convert menu to choose which quotation characters are converted by all the other items in the Convert|Quotes submenu. The choices in the “straight to smart quotes” group are explained in the Convert|Quotes|Straight ⇒ Smart Quotes topic. The options in the “smart to straight quotes” group are explained in the Convert|Quotes|Smart ⇒ Straight Quotes topic. The “replace smart quotes” group is explained in the Convert|Quotes|Replace Smart Quotes topic.

Whenever you change settings in the Quotation Style dialog, those settings are used for all future quote conversions in all files until you change the settings again.

Convert|Lines|Wrapping ⇒ Line Breaks

When Options|Word Wrap is on, EditPad breaks up long lines so they fit within the width of the EditPad window or are no longer than a certain number of characters. This wrapping is dynamic. It is not saved into the file. Therefore, the file will look different if you change the word wrap settings or open it in another application.

Use Convert|Lines|Wrapping ⇒ Line Breaks if you want to solidify a text in its wrapped state. This converts all soft line breaks introduced by the wrapping into hard line breaks as if you had pressed Enter at each position where a long line is wrapped.

You will also be asked if you want to align the text at the left side only, or if you want to align it at both sides. If you choose to align to the left side only, only line breaks are inserted, as explained above. If you choose to align to both sides, EditPad Pro also inserts spaces into each line, so that each line has the same number of characters. The spaces are distributed evenly across each line. They are inserted next to spaces already present on the original line. When the file is viewed with a monospaced font such as Consolas then both the left and the right side of the text will be perfectly aligned.

This function also turns off word wrap.

If you want to limit the maximum length of each line in the file, you can first set the maximum line length by using the drop-down menu of the word wrap button on the toolbar. Then use Convert|Lines|Wrapping ⇒ Line Breaks.

You should use Convert|Lines|Wrapping ⇒ Line Breaks when you have finished editing a file that will be used by an application or system that does not support word wrapping, or does not support lines longer than

a certain number of characters. For example, when preparing a message to be posted to a Usenet newsgroup or a message to be sent via email, you should set word wrap to 72 characters and then use **Convert|Lines|Wrapping ⇒ Line Breaks**. There are still a lot of newsreaders and email clients in use that cannot word wrap messages.

Convert|Lines|Line Breaks ⇒ Wrapping

Convert|Lines|Line Breaks ⇒ Wrapping attempts to do the opposite of **Convert|Lines|Wrapping ⇒ Line Breaks**. Email clients and newsreaders, for example, often limit the maximum line length, inserting additional line breaks into the file that you did not type in by pressing Enter on the keyboard. This is done because older email and newsgroup software cannot handle long lines.

The problem with this is that after the hard line breaks have been inserted, the text becomes difficult to edit. The text will not be nicely and automatically rewrapped when you insert new text into the middle of a paragraph, like EditPad does when word wrap is on. With **Convert|Lines|Line Breaks ⇒ Wrapping** you can tell EditPad Pro to attempt to remove the line breaks that were inserted into the file for the purpose of emulating word wrap. “Attempt” is the operative word because EditPad Pro has no way of detecting whether a line break was automatically inserted to limit the length of the line, or if it was inserted by a person pressing Enter on the keyboard. Technically, there is no difference between the two. But usually, EditPad Pro does a remarkably good job.

If there is no selection when you use this command then it unwraps the entire file. If there is a selection, only the selected lines are unwrapped. The algorithm that decides whether a line break should be removed or not may make different decisions when unwrapping the whole file versus unwrapping only part of the file. When unwrapping part of the file, it assumes that the selection is your entire document. If you have a file with lines that were pasted in from different files that were hard wrapped at different line lengths, then separately unwrapping blocks of lines wrapped at different line lengths may produce better results than unwrapping everything at once.

Convert|Lines|Double ⇒ Single Spacing

A text file is double-spaced if there is a blank line between each line of text. Select **Double ⇒ Single Spacing** in the Lines submenu of the Convert menu to remove those blank lines. If you select multiple lines of text first, only the selected block is converted. If there is no selection or the selection does not span multiple lines, then this conversion affects the entire file.

If you use the double to single spacing conversion on a file that is not double-spaced, EditPad Pro removes as many alternating blank lines as possible. If part of the file is single spaced and part is double spaced, the whole file becomes single spaced. If the entire file is single spaced, but blank lines are used to separate paragraphs, then those blank lines are removed.

Certain files appear double-spaced in EditPad but single-spaced in other applications such as Notepad. This is caused by a different interpretation of line break characters. Usually, those files were incorrectly transferred between a Windows and UNIX system, such as a PC and a web server. This causes incorrect line break characters to be added to the file. You can easily fix such files in EditPad Pro. First, select **Convert|Lines|Double ⇒ Single Spacing** from the menu to remove the unwanted blank lines. Blank lines that appeared in the original file will remain. Then, select **Convert|Line Break Style|Windows** to make all the

line breaks consistent with the Windows format. If **Convert|Line Break Style|Windows** is grayed out (i.e. unavailable), then the file already uses the correct line breaks.

Convert|Lines|Single ⇒ Double Spacing

A text file is double-spaced if there is a blank line between each line of text. Select **Single ⇒ Double Spacing** in the **Lines** submenu of the **Convert** menu to insert a blank line between each line of text, making the current file double spaced. If you select multiple lines of text first, only the selected block is converted. If there is no selection or the selection does not span multiple lines, then this conversion affects the entire file.

Note that EditPad Pro does not force you to maintain the double spaced format of the file. Pressing **Enter** will insert a single line break as usual. Press **Enter** twice to insert a double line break. You can type text into the blank lines added by the conversion if you want. This could be useful to annotate a manuscript, for example.

Convert|Tabs|Tabs ⇒ Spaces

Select **Tabs ⇒ Spaces** in the **Lines** submenu of the **Convert** menu to convert all tab characters in the active file into spaces. This command only has its desired effect when using a monospaced font or text layout. If you select some text first then only the selected lines are converted. If there is no selection then this conversion affects the entire file.

Each tab in the file is replaced with the same number of spaces as the number of columns between the tab's position and the next tab stop. If the text is monospaced and thus the width of a space is the same as the width of any other character, then the result is that after the conversion, the layout of the text remains exactly the same. You would only see a difference when visualizing spaces.

To get a proper conversion, you need to make sure EditPad is showing your file with the correct tab size before converting tabs to spaces. You can configure the tab size on the **Tabbing** page in the file type configuration. The conversion correctly handles tab stops at specific columns and elastic tab stops. It also converts tab-separated values to space-separated values if the tab separator is a tab.

Only actual tab characters are converted. If you have “value separator” set to something other than “Tab” in **File Types|Tabbing** then those value separators are not replaced or padded with spaces. Actual tabs within the values are still converted.

If you send a file to somebody else, then converting tabs to spaces ensures that columns and indentation in your file will appear exactly the same for them as they do for you, even if their text editor uses a different tab size.

Convert|Tabs|Spaces ⇒ Tabs

Select **Spaces ⇒ Tabs** in the **Lines** submenu of the **Convert** menu to convert as many spaces into tab characters as possible, without changing the layout of the file. This command only has its desired effect when

using a monospaced font or text layout. If you select some text first then only the selected lines are converted. If there is no selection then this conversion affects the entire file.

The conversion uses the tab size that you specified on the Tabbing page in the file type configuration. It does not take the settings for tab stops at specific columns, elastic tabbing, or tab-separated values into account. For best results, those should all be unused or turned off.

Convert | Tabs | Indentation Spaces ⇒ Tabs

Select Indentation Spaces ⇒ Tabs in the Lines submenu of the Convert menu to convert as many spaces at the start of each line into tab characters as possible, without changing the layout of the file. Spaces that occur after a non-whitespace character on a line are not converted. This command only has its desired effect when using a monospaced font or text layout. If you select some text first then only the selected lines are converted. If there is no selection then this conversion affects the entire file.

The conversion uses the tab size that you specified on the Tabbing page in the file type configuration. It does not take tab stops at specific columns into account. The settings for elastic tab stops and tab-separated values don't matter because they don't affect tabs used for indentation.

Convert | Characters | Characters ⇒ \uFFFF

Select the Characters ⇒ \uFFFF command in the Characters submenu of the Convert menu to replace all non-ASCII characters in the file with an escape sequence in the form of \uFFFF. If you select some text before using this command, only non-ASCII characters in the selection are replaced. Otherwise, all non-ASCII characters in the file are replaced.

Using this command does not change the encoding EditPad Pro uses for this file. Saving the file after replacing all non-ASCII characters this way only results in an ASCII file if the encoding you're using for the file is compatible with ASCII. For example, the Windows code pages will result in an ASCII file, but UTF-16 will not.

In Convert | Text Encoding you can select "ASCII + \uFFFF Unicode Escapes" as the file's encoding. When using this encoding, EditPad Pro displays the actual characters for all \uFFFF escapes in the file. When you type non-ASCII characters into the file, EditPad Pro will display the character you typed, but store its \uFFFF escape in the file. If you type "après-ski" using this encoding, EditPad Pro will display "après-ski", but store "apr\u00E8s-ski" in the file. If you open that file in Notepad (which cannot interpret Unicode escapes), you'll see "apr\u00E8s-ski".

Use Convert | Characters ⇒ \uFFFF along with an encoding such as UTF-8 or Windows 1252 when you want to use Unicode escapes in only part of the file. EditPad will display the Unicode escapes while you edit the file. You can see where Unicode escapes are used and where actual characters are used.

Use the "ASCII + \uFFFF Unicode Escapes" encoding when working with files where Unicode escapes are the only way of representing non-ASCII characters. EditPad will display the actual characters while you edit the file. You won't see the Unicode escapes. EditPad makes sure all newly typed or pasted characters are stored as Unicode escapes in the file.

If you have a file that contains non-ASCII characters that are not written as Unicode escapes then you can convert that file into pure ASCII + \uFFFF Unicode Escapes in three steps. First, open the file in EditPad and make sure all characters are displayed correctly. If not, use Convert|Text Encoding to select the proper encoding with the “interpret” option. Second, use Convert|Characters ⇒ \uFFFF to convert all non-ASCII characters to Unicode escapes. The actual escapes will appear. Finally, use Convert|Text Encoding and select to convert to ASCII + \uFFFF Unicode Escapes. The Unicode escapes now again appear as actual characters. If you save the file then it will be saved as pure ASCII with Unicode escapes.

In the Encoding section in the file type configuration, there is an option to make EditPad Pro detect whether a file consists of pure ASCII with Unicode Escapes. If that option is on, EditPad Pro automatically selects the “ASCII + \uFFFF Unicode Escapes” encoding for such files and displays the actual characters rather than the escapes. By default, this option is off for all file types except the Java file type. In Java source code files, Unicode escapes are 100% equivalent to actual characters, even for ASCII characters.

Convert|Characters| \uFFFF ⇒ Characters

Select the \uFFFF ⇒ Characters command in the Characters submenu of the Convert menu to replace all escape sequences in the form of \uFFFF into their actual characters. If you select some text before using this command, only escape sequences in the selection are replaced. Otherwise, all escape sequences in the file are replaced.

The conversion will only be successful if all the characters can be represented in the encoding used by the file. If not, use Convert|Text Encoding to change the encoding before converting the escapes. Escapes that represent characters that are not supported by the file’s encoding will not be converted. EditPad will pop up a message to alert you when this happens.

Use the \uFFFF ⇒ Characters command when you actually want to remove the escapes from your file. Do not use this command if you have a file that is pure ASCII with \uFFFF escapes and you want to be able to see the actual characters while keeping Unicode escapes in the file. To do that, use Convert|Text Encoding and select the “ASCII + \uFFFF Unicode Escapes” encoding. This encoding makes EditPad Pro display the actual characters, but keeps the escapes in the file. If you type or paste non-ASCII characters while using the “ASCII + \uFFFF Unicode Escapes” encoding, they will be automatically stored as Unicode escapes in the file.

**Convert|Characters| Characters ⇒ **

Select the Characters ⇒ command in the Characters submenu of the Convert menu to replace all non-ASCII characters in the file with a decimal character reference in the form of . If you select some text before using this command, only non-ASCII characters in the selection are replaced. Otherwise, all non-ASCII characters in the file are replaced.

Using this command does not change the encoding EditPad Pro uses for this file. Saving the file after replacing all non-ASCII characters this way only results in an ASCII file if the encoding you’re using for the file is compatible with ASCII. For example, the Windows code pages will result in an ASCII file, but UTF-16 will not.

In Convert|Text Encoding you can select “ASCII + NCR” as the file’s encoding. When using this encoding, EditPad Pro displays the actual characters for all character references in the file. When you type non-ASCII characters into the file, EditPad Pro will display the character you typed, but store its NCR in the file. If you type “après-ski” using this encoding, EditPad Pro will display “après-ski”, but store “après-ski” in the file. If you open that file in Notepad (which cannot interpret numeric character references), you’ll see “après-ski”.

Use the Convert|Characters ⇒ along with an encoding such as UTF-8 or Windows 1252 when you want to use numeric character references in only part of the file. EditPad will display the numeric character references while you edit the file. You can see where numeric character references are used and where actual characters are used.

Use the “ASCII + NCR” encoding when working with files where numeric character references are the only way of representing non-ASCII characters. EditPad will display the actual characters while you edit the file. You won’t see the numeric character references. EditPad makes sure all newly typed or pasted characters are stored as numeric character references in the file.

If you have a file that contains non-ASCII characters that are not written as numeric character references then you can convert that file to being pure ASCII + NCR in three steps. First, open the file in EditPad and make sure all characters are displayed correctly. If not, use Convert|Text Encoding to select the proper encoding with the “interpret” option. Second, use Convert|Characters ⇒ to convert all non-ASCII characters to numeric character references. The actual escapes will appear. Finally, use Convert|Text Encoding and select to convert to ASCII + NCR. The numeric character references now again appear as actual characters. If you save the file then it will be saved as pure ASCII with numeric character references.

In the Encoding section in the file type configuration, there is an option to make EditPad Pro detect whether a file consists of pure ASCII with numeric character references. If that option is on, EditPad Pro automatically selects the “ASCII + NCR” encoding for such files and displays the actual characters rather than the numeric character references.

Convert|Characters|Characters ⇒ ÿ FF;

This command does the same as Convert|Characters|Characters ⇒ except that it replaces characters with hexadecimal references instead of decimal references. So “après-ski” becomes “après-ski”.

Convert|Characters| and ÿ FF; ⇒ Characters

Select the and ÿ FF; ⇒ Characters command in the Characters submenu of the Convert menu to replace all decimal and hexadecimal character references in the form of and ÿ FF; into their actual characters. If you select some text before using this command, only numeric character references in the selection are replaced. Otherwise, all numeric character references in the file are replaced.

This command does not convert numeric character references that represent the characters <, >, ';, ", and & that have special meanings in XML. To convert these, use Convert|Characters|XML Entities ⇒ Reserved Characters.

The conversion will only be successful if all the characters can be represented in the encoding used by the file. If not, use **Convert|Text Encoding** to change the encoding before converting the numeric character references. numeric character references that represent characters that are not supported by the file's encoding will not be converted. EditPad will pop up a message to alert you when this happens.

Use the **** and **** \Rightarrow **Characters** command when you actually want to remove the numeric character references from your file. Do not use this command if you have a file that is pure ASCII with numeric character references and you want to be able to see the actual characters while keeping numeric character references in the file. To do that, use **Convert|Text Encoding** and select the “ASCII + NCR” encoding. This encoding makes EditPad Pro display the actual characters, but keeps the numeric character references in the file. If you type or paste non-ASCII characters while using the “ASCII + NCR” encoding, they will be automatically stored as numeric character references in the file.

Convert|Characters|Characters \Rightarrow &htmlchar;

Select the **Characters \Rightarrow &htmlchar;** command in the **Characters** submenu of the **Convert** menu to replace all characters that have HTML entities with those HTML entities. It replaces “après-ski” with “après-ski”. If you select some text before using this command, only characters in the selection are replaced. Otherwise, all characters in the file that have HTML entities are replaced.

Characters that do not have HTML entities are not changed. If you want to convert your file into pure ASCII using HTML entities and numeric character references for non-ASCII characters, first use the **Characters \Rightarrow &htmlchar;** command to replace those characters that have named HTML entities. Then use the **Characters \Rightarrow ** or **Characters \Rightarrow ** command to replace the remaining non-ASCII characters that do not have named HTML entities.

This command does not replace reserved characters such as `<`. Thus you can safely use this command on an entire HTML file without destroying its HTML tags. If you do want to replace reserved characters such as `<` with entities such as `<`, use the **Convert|Characters|Reserved Characters \Rightarrow XML Entities** menu item.

Convert|Characters|&htmlchar; \Rightarrow Characters

Select the **&htmlchar; \Rightarrow Characters** command in the **Characters** submenu of the **Convert** menu to replace all named HTML entities into their actual characters. If you select some text before using this command, only HTML entities in the selection are replaced. Otherwise, all HTML entities in the file are replaced.

The conversion will only be successful if all the characters can be represented in the encoding used by the file. If not, use **Convert|Text Encoding** to change the encoding before converting the HTML entities. HTML entities that represent characters that are not supported by the file's encoding will not be converted. EditPad will pop up a message to alert you when this happens.

This command does not replace numeric character references. Only named HTML entities are replaced. To replace both named HTML entities and numeric character references, use both the **&htmlchar; \Rightarrow Characters** command and the **Convert|Characters| and \Rightarrow Characters** command.

This command also does not replace HTML entities that represent reserved characters such as `<`. To replace those, use **Convert|Characters|XML Entities \Rightarrow Reserved Characters**.

Use the `&htmlchar; ⇒ Characters` command when you actually want to remove the HTML entities from your file. Do not use this command if you have a file that is pure ASCII with HTML entities and you want to be able to see the actual characters while keeping HTML entities in the file. To do that, use `Convert|Text Encoding` and select the “ASCII + `&htmlchar;`” encoding. This encoding makes EditPad Pro display the actual characters, but keeps the HTML entities in the file. If you type or paste non-ASCII characters while using the “ASCII + NCR” encoding, they will be automatically stored as HTML entities in the file.

Convert | Characters | Reserved Characters ⇒ XML Entities

Select the `Reserved Characters ⇒ XML Entities` command in the `Characters` submenu of the `Convert` menu to replace the characters `<`, `>`, `'`, `"`, and `&` that have special meanings in XML with their respective XML entities `<`, `>`, `'`, `"`, and `&`. If you select some text before using this command, only characters in the selection are replaced. Otherwise, all reserved characters in the file are replaced.

Convert | Characters | XML Entities ⇒ Reserved Characters

Select the `XML Entities ⇒ Reserved Characters` command in the `Characters` submenu of the `Convert` menu to replace XML entities `<`, `>`, `'`, `"`, and `&` with the characters `<`, `>`, `'`, `"`, and `&` that they represent. Numeric character references that represent these 5 characters are also converted. If you select some text before using this command, only characters in the selection are replaced. Otherwise, all reserved characters in the file are replaced.

Convert | ROT-13

Select `ROT-13` in the `Convert` menu to apply ROT-13 character rotation to the current selection. This makes the text unreadable. Applying ROT-13 again restores the original text.

Before the Internet went commercial, people had the good habit to use ROT-13 to conceal possible offensive postings in newsgroups and then add a warning message in plain text. This made sure that nobody would read the text by accident and be offended. If one was offended after applying ROT-13, he or she would only have himself or herself to blame for ignoring the warning.

13. Options Menu

Options | File Type

When you open a file, EditPad uses the settings you made in the file type configuration to pick a file type for the file. The settings of that file type are then applied to the file. You can pick a different file type for the active file via the File Type submenu in the Options menu.

Changing the file type makes EditPad apply all the settings of that file type to the active file. It does not change the file's name or extension to match the new file type's file mask. Use File|Rename / Move to change the file's name and extension.

If you change a file's file type via Options|File Type then EditPad Pro remembers that when you close and reopen the file if you tick "preserve file settings" on the Save Files page in the Preferences. EditPad Lite does not have this option.

Options | Word Wrap

Select Word Wrap in the Options menu to toggle word wrapping on or off. By default, word wrap takes place at the right-hand edge of EditPad's window. If you want the text to wrap at a certain number of characters, use the drop-down menu of the word wrap button on the toolbar. You can select one of a few commonly used line lengths, or define your own.

Note that "word wrap" is a file type specific setting in EditPad. If you change the setting through the Options menu or the toolbar, it is applied to the active file only. If you create a new file or open an existing one then EditPad uses the setting you made on the Editor page for the type of file you are creating or opening. Also, when you use File|Save As to save the file as a different file type, the setting for the current file changes to what you specified for the new file type.

If you change a file's word wrap setting, EditPad Pro remembers that when you close and reopen the file if you tick "preserve file settings" on the Save Files page in the Preferences. EditPad Lite does not have this option.

Word wrapping is only used in text mode. In hexadecimal mode, use Options|Record Size.

Options | Indent Wrapped Lines

When word wrap is on, you can select Indent Wrapped Lines in the Options menu to toggle wrapping line indentation on or off. When off, wrapped lines start at the first column regardless of the indentation of the original line. When on, wrapped lines are indented a certain amount. By default wrapped lines are indented by the same amount as the line they were wrapped from. You can specify a different indentation amount for wrapped lines on the Editor page in the file type configuration. Wrapped lines can be indented a specific number of columns or they can be indented a certain number of columns more or less than the line they were wrapped from. This is known as "hanging indent". Whether they are indented by default is also set in the file type configuration.

This option never affects the first line in each paragraph (block of lines wrapped from the same line in the file). The first line in the paragraph is only indented if the line starts with spaces or tabs.

If you change a file's wrapped line indentation setting, EditPad Pro remembers that when you close and reopen the file if you tick "preserve file settings" on the Save Files page in the Preferences.

Options | Keep Indent

Turn on Options | Keep Indent if you want the next line to automatically start at the same column position as the previous line whenever you press Enter on the keyboard while in the editor. EditPad accomplishes this by counting the number of spaces and tabs at the beginning of the previous line and inserting them at the beginning of the new line you created by pressing Enter.

When Keep Indent is on, using the Backspace key when there is only whitespace to the left of the cursor either deletes a single space, or deletes multiple spaces to go back to the previous level of indentation. The "backspace unindents" option in the Cursor Preferences determines this.

If you want EditPad Pro to automatically increase or decrease indentation based on the file's syntax when you press Enter, then turn on Options | Auto Indent. Turning off Keep Indent automatically turns off Auto Indent.

Note that "keep indent" is a file type specific setting in EditPad. If you change the setting through the Options menu or the toolbar, it is applied to the active file only. If you create a new file or open an existing one then EditPad uses the setting you made on the Tabbing page for the type of file you are creating or opening. Also, when you use File | Save As to save the file as a different file type, the setting for the current file changes to what you specified for the new file type.

If you change a file's "keep indent" setting, EditPad Pro remembers that when you close and reopen the file if you tick "preserve file settings" on the Save Files page in the Preferences. EditPad Lite does not have this option.

Options | Auto Indent

Turn on Options | Auto Indent if you want the indentation of the next line to be increased or decreased automatically relative to the previous line whenever you press Enter on the keyboard while in the editor. Auto Indent is only available if the syntax coloring scheme you selected for the file's file type specifies indentation rules. Those rules may use the indentation style you select on the Brackets page in the file type configuration.

Turning on Auto Indent automatically turns on Options | Keep Indent. If the syntax coloring scheme does not apply any indentation rules to the text on the current line then the new line gets the same amount of indentation when you press Enter.

Auto Indent may change indentation after you enter text on the new line after pressing Enter. The file types for C-style languages included with EditPad Pro, for example, automatically outdent a line with a closing brace by one level. But as you enter your code, that closing brace may not be there yet. So if you have these three lines:

```
void routine() {
    if (condition) {
        statement;
    }
}
```

If you press Enter with the cursor at the third line then the new fourth line is automatically indented by four spaces. There is nothing on the third line that affects indentation. If you enter a } on the fourth line and press Enter or move the cursor away from the line then Auto Indent kicks in and outdents the fourth line by one step, producing the desired result:

```
void routine() {
    if (condition) {
        statement;
    }
}
```

EditPad Pro's Auto Indent is just a typing aid. It allows you to enter the above example without having to press the Space bar or Backspace key to add or remove any indentation at all. EditPad Pro's Auto Indent does not analyze your whole file to automatically fix any indentation issues. Auto Indent does not fix the indentation of lines that you paste. When you press Enter, it looks at the contents of the line the cursor is on to determine the indentation level of the next line. When you enter text at the start of a line, it adjusts the indentation of that line when you move the cursor away from the line. It then only looks at the text at the start of that line.

Auto Indent does not interfere if you manually adjust the indentation of a line. In the above example, if you use the Space bar or Backspace key between pressing Enter and typing the } then the } is inserted at the position of the cursor. EditPad assumes that you know where you want the } to be when you manually edit the indentation.

Automatic indentation is always applied to line breaks that are inserted automatically when Options|Auto Break is on. Options|Auto Indent only affects line breaks that you enter manually.

Note that “auto indent” is a file type specific setting in EditPad Pro. If you change the setting through the Options menu or the toolbar, it is applied to the active file only. If you create a new file or open an existing one then EditPad uses the setting you made on the Tabbing page for the type of file you are creating or opening. Also, when you use File|Save As to save the file as a different file type, the setting for the current file changes to what you specified for the new file type.

If you change a file's “auto indent” setting, EditPad Pro remembers that when you close and reopen the file if you tick “preserve file settings” on the Save Files page in the Preferences.

Options|Auto Break

Options|Auto Break is only available if the syntax coloring scheme you selected for the file's file type specifies automatic breaking rules. Those rules may use the breaking and indentation style you select on the Brackets page in the file type configuration. Most of the syntax coloring schemes for programming languages use that breaking and indentation style for curly braces or keywords that surround blocks of code. Some schemes like those for Visual Basic provide their own rules. The HTML and XML schemes use the rules for markup tags.

Automatic breaking can insert line breaks before and after the text that you enter via the keyboard. The syntax coloring scheme can specify this to be done immediately or when you move the cursor to another line.

The schemes provided with EditPad Pro apply automatic breaking immediately unless they are breaking around a keyword. The Delphi scheme, for example, breaks around the “begin” and “end” keywords. It delays automatic breaking so that you can still enter something like “BeginOperation” without getting a line break in the middle of your identifier. Automatic breaking is smart enough not to insert a duplicate line break if you press the Enter key immediately after typing “begin”. Automatic breaking never removes any line breaks that you enter.

If the syntax coloring scheme specifies automatic indentation rules then those rules are always applied to any line breaks that are inserted automatically. It doesn’t matter whether Options|Keep Indent and Options|Auto Indent are on or off.

Options|Auto Break is particularly powerful when used in combination with Edit|Auto Match Brackets. When both are on, if you enter an opening bracket, or a keyword or tag that the syntax coloring scheme matches as an opening bracket, then Auto Break first breaks and indents the opening bracket. Then Auto Match Brackets inserts the closing bracket. Then Auto Break kicks in again and breaks and indents the closing bracket. These three steps happen as one operation. So when editing a file using one of EditPad Pro’s syntax coloring schemes for C-style languages, pressing { on the keyboard gives you a new { } block with the cursor on a blank line between them with extra indentation.

Note that “auto break” is a file type specific setting in EditPad Pro. If you change the setting through the Options menu or the toolbar, it is applied to the active file only. If you create a new file or open an existing one then EditPad uses the setting you made on the Brackets page for the type of file you are creating or opening. Also, when you use File|Save As to save the file as a different file type, the setting for the current file changes to what you specified for the new file type.

Options|Visualize Indentation

Turn on Visualize Indentation in the Options menu if you want EditPad Pro to indicate indentation levels with vertical lines or different background colors for the indentation.

On the Navigation page in the file type configuration you can specify whether indentation indicators should be on or off by default for each file type. There you can also specify which lines should have indentation indicators and whether indentation indicators may go through insufficiently indented lines.

To change the color and visual style of the indentation indicators, customize the color palette and change the “Editor: Indentation level 1” through “Editor: Indentation level 4” colors. You can have 4 different colors for the indentation indicators to make it easier to follow deeply nested indentation. The 4 colors are cycled every 4 indentation levels. EditPad Pro can show indentation indicators up to 63 levels deep. You can make colors 1 and 3 as well as 2 and 4 the same if you want to have two alternating colors for indentation indicators. Or you can make all 4 the same if you want the indicators to be more subtle.

Selecting a vertical style for the indentation levels in the color palette makes indentation indicators appear as vertical lines. Selecting a background color applies that background color to the spaces and tabs used for indentation, whether they are visualized or not. If you allow indentation indicators to go through insufficiently indented lines then the text color of the indentation levels is applied to the insufficiently indented text.

If you change a file’s “visualize indentation” setting, EditPad Pro remembers that when you close and reopen the file if you tick “preserve file settings” on the Save Files page in the Preferences.

Options | Visualize Spaces

Turn on Visualize Spaces in the Options menu if you want spaces and tabs to be visualized. Spaces are indicated by a vertically centered dot. Tabs are indicated by a » character. This option is useful when working with files where extraneous whitespace can lead to problems, or where the difference between tabs and spaces matters.

To change the color of spaces and tabs, customize the color palette and change the “Editor: Whitespace” color. Its text color is only applied to visualized spaces and tabs. But its background color is applied to spaces and tabs whether they are visualized or not. So if you set a background color other than “default” for the “Editor: Whitespace” color then spaces and tabs appear as colored rectangles, with or without their symbol.

Note that “visualize spaces” is a file type specific setting in EditPad Pro. If you change the setting through the Options menu or the toolbar, it is applied to the active file only. If you create a new file or open an existing one then EditPad uses the setting you made on the Editor page for the type of file you are creating or opening. Also, when you use File|Save As to save the file as a different file type, the setting for the current file changes to what you specified for the new file type.

If you change a file’s “visualize spaces” setting, EditPad Pro remembers that when you close and reopen the file if you tick “preserve file settings” on the Save Files page in the Preferences.

Options | Visualize Line Breaks

Turn on Visualize Line Breaks in the Options menu if you always want to see a ¶ character (Pilcrow symbol) at the end of each paragraph, indicating the invisible line break character(s) at the end of each line. To change the color of the ¶ character, customize the color palette and change the “Editor: Line breaks” color. Instead of showing the generic ¶ character for all line breaks, EditPad Pro can show specific symbols that indicate the line break style. You can configure this on the Editor page in the Preferences.

If you turn off Options|Visualize Line Breaks then line break symbols are only displayed when line break characters are selected. They are then shown with colors for selected text in reverse. If you never want to see any line break symbols, turn off “always visualize line breaks in selected text” in the Editor Preferences.

Note that “visualize line breaks” is a file type specific setting in EditPad Pro. If you change the setting through the Options menu or the toolbar, it is applied to the active file only. If you create a new file or open an existing one then EditPad uses the setting you made on the Editor page for the type of file you are creating or opening. Also, when you use File|Save As to save the file as a different file type, the setting for the current file changes to what you specified for the new file type.

Options | Record Size

What Options|Word Wrap is to text editing, Options|Record Size is to hexadecimal editing. Use this menu item to change the number of bytes that EditPad Pro displays on each row in hexadecimal mode. If you enter zero, you get EditPad Pro’s default behavior of showing the smallest multiple of 8 bytes that fits within the width of EditPad’s window. If you enter a positive number, that’s the number of bytes EditPad Pro displays on a row. You can enter any number. It doesn’t have to be divisible by 8 or by 2.

The Record Size command is particularly useful for editing binary files in which each record takes up a specific number of bytes. Set the record size to that number of bytes to have one record on each line.

Note that the record size is a file type specific setting in EditPad Pro. If you change the setting through the Options menu or the toolbar, it is applied to the active file only. If you create a new file or open an existing one then EditPad uses the setting you made on the Encoding page for the type of file you are creating or opening. Also, when you use File|Save As to save the file as a different file type, the setting for the current file changes to what you specified for the new file type.

Options | Line Numbers

Turn on Options|Line Numbers to have EditPad display a number before each line in the left margin.

You can set the default state on the Editor page in the file type configuration. There you can also choose whether word wrap should affect line numbers or not. Tick “count physical lines only” if you do not want word wrap to affect line numbering. Then only the first line of each paragraph gets a line number. Otherwise all lines are numbered.

You may temporarily see negative line numbers if you open a very large file and immediately jump to the end of the file. EditPad scans the file for line breaks from top to bottom and from bottom to top at the same time using two background threads. This allows you to immediately start working with the lines at the top and at the bottom of the file, no matter how large your file is. But EditPad won’t know the correct line numbers for the lines scanned from bottom to top until all line breaks have been counted. So while line break scanning continues in the background, EditPad shows negative line numbers for the bottom half of the file. The last line in the file gets number -1, the penultimate line -2, and so on. When line break scanning is completed the line numbers are updated immediately to positive numbers counting from the start of the file.

Options | Column Numbers

Turn on Options|Column Numbers to have EditPad display a horizontal ruler above the file indicating column numbers. Column numbers are only displayed when using a fixed-width font or a monospaced text layout.

Note that “column numbers” is a file type specific setting in EditPad Pro. If you change the setting through the Options menu or the toolbar, it is applied to the active file only. If you create a new file or open an existing one then EditPad uses the setting you made on the Editor page for the type of file you are creating or opening. Also, when you use File|Save As to save the file as a different file type, the setting for the current file changes to what you specified for the new file type.

Options | Font

The standard Windows font selection dialog appears when you select Font in the Options menu. Select the font you want to use and click OK. You can also select a recently used font from the Options|Font submenu.

The font setting applies to the entire file. It only changes the way it is displayed. EditPad is a plain text editor. It does not support any text formatting.

Changing the font via Options | Font only affects the active file. In EditPad, the choice of font is part of the text layout configuration. That is the combination of settings that determines how EditPad displays text. The default text layout configuration can be set for each file type on the Editor page in the file type configuration.

Options | Font only overrides the main font of the text layout configuration. Complex script text layouts may use fallback fonts to use different fonts for different scripts when you don't have a font that supports all the languages or scripts in your file all by itself. To change the fallback fonts, you need to edit the text layout configuration. Overriding the main font with Options | Font does not disable the fallback fonts.

Fonts with Chinese, Japanese and Korean characters are available in two variants: the regular variant (e.g. MS Mincho), and a rotated variant for vertical printing (e.g. @MS Mincho). The rotated variant always has the same name as the regular variant, but with an @ symbol in front of it. In Options | Font, you should select the regular variant, so characters will appear upright on the screen. In the print preview, you can select the rotated variant if you want. If you print with the rotated variant, the text will appear printed vertically if you rotate the printed sheet of paper 90 degrees clockwise. If you print with the regular variant, the printed text will flow from left to right, like you edit it in EditPad.

Options | Text Layout

Text Layout Configuration

Select the text layout configuration that you want to use

- Left-to-right
- Proportionally spaced left-to-right
- Monospaced left-to-right
- Monospaced ideographic width
- Complex script left-to-right
- Complex script right-to-left
- Monospaced complex left-to-right
- Monospaced complex right-to-left

+ New

Delete

Up

Down

Text layout and direction

☐ Complex script, predominantly left-to-right
☐ Complex script, predominantly right-to-left
☒ Left-to-right only
☐ Monospaced left-to-right only
☐ ASCII characters with full ideographic width

Characters to treat as word characters (in addition to letters and digits)

☒ Underscores
☐ Hyphens
☐ Other punctuation
☐ Math symbols
☒ Currency symbols
☐ Other symbols
☐ Words determined by complex script analysis (bidirectional cursor only)

Main font

☐ Allow bitmapped fonts
 Consolas
☐ Bold ☐ Italic Size: 10
 Visualize control characters:
 IBM PC glyphs (depends on font)

Line and character spacing

Increase (or decrease) the line height by 0 pixels

Add 0 pixels of extra space between lines

Increase (or decrease) the character width by 0 pixels

Selected text layout configuration

Name of this text layout configuration:
Left-to-right

Example:
Latin العربية

Text cursor movement

☒ Monodirectional (left is always left and right is always right)
☐ Bidirectional (left and right reverse when the text direction reverses)

Selection of words

☒ Select only the word
☐ Select the word plus everything up to the next word

Text cursor appearance

Insert mode text cursor:
Insertion cursor

Overwrite mode text cursor:
Overwrite cursor

Fallback fonts (complex script only)

+ Add

Delete

Up

Down

OK Cancel Help

In EditPad, a “text layout” is a combination of settings that control how text is displayed and how the text cursor navigates through that text. The settings include the font, text direction, text cursor behavior, which characters are word characters, and how the text should be spaced.

The default text layout is configured for each file type in File Types | Editor. Via the Text Layout submenu in the Options menu you can select a different text layout for the active file. You can use the Configure Text Layout item in the Text Layout submenu to configure a new text layout or to edit an existing one. If you edit an existing text layout, the changes will be applied to any file and to any file type that uses it.

Select The Text Layout Configuration That You Want to Use

The Text Layout Configuration screen shows the details of the text layout configuration that you select in the list in the top left corner. Any changes you make on the screen are automatically applied to the selected layout and persist as you choose different layouts in the list. The changes become permanent when you click OK. The layout that is selected in the list when you click OK becomes the new default layout.

Click the New and Delete buttons to add or remove layouts. You must have at least one text layout configuration. If you have more than one, you can use the Up and Down buttons to change their order. The order does not affect anything other than the order in which the text layouts configurations appear in selection lists.

EditPad comes with a number of preconfigured text layouts. If you find the options on this screen bewildering, simply choose the preconfigured layout that matches your needs, and ignore all the other settings. You can fully edit and delete all the preconfigured text layouts if you don't like them.

- Left-to-right: Normal settings with best performance for editing text in European languages and ideographic languages (Chinese, Japanese, Korean). The default font is monospaced. The layout does respect individual character width if the font is not purely monospaced or if you select another font.
- Proportionally spaced left-to-right: Like left-to-right, but the default font is proportionally spaced.
- Monospaced left-to-right: Like left-to-right, but the text is forced to be monospaced. Columns are guaranteed to line up perfectly even if the font is not purely monospaced. This is the best choice for working with source code and text files with tabular data.
- Monospaced ideographic width: Like monospaced left-to-right, but ASCII characters are given the same width as ideographs. This is the best choice if you want columns of mixed ASCII and ideographic text to line up perfectly.
- Complex script left-to-right: Supports text in any language, including complex scripts (e.g. Indic scripts) and right-to-left scripts (Hebrew, Arabic). Choose this for editing text that is written from left to right, perhaps mixed with an occasional word or phrase written from right to left.
- Complex script right-to-left: For writing text in scripts such as Hebrew or Arabic that are written from right to left, perhaps mixed with an occasional word or phrase written from left to right.
- Monospaced complex left-to-right: Like “complex script left-to-right”, but using monospaced fonts for as many scripts as possible. Text is not forced to be monospaced, so columns may not line up perfectly.
- Monospaced complex right-to-left: Like “complex script right-to-left”, but using monospaced fonts for as many scripts as possible. Text is not forced to be monospaced, so columns may not line up perfectly.

Selected Text Layout Configuration

The section in the upper right corner provides a box to type in the name of the text layout configuration. This name is only used to help you identify it in selection lists when you have prepared more than one text layout configuration.

In the Example box you can type in some text to see how the selected text layout configuration causes the editor to behave.

Text Layout and Direction

- Complex script, predominantly left-to-right: Text is written from left to right and can be mixed with text written from right to left. Choose this for complex scripts such as the Indic scripts, or for text in any language that mixes in the occasional word or phrase in a right-to-left or complex script.
- Complex script, predominantly right-to-left: Text is written from right to left and can be mixed with text written from left to right. Choose this for writing text in scripts written from right to left such as Hebrew or Arabic.
- Left-to-right only: Text is always written from left to right. Complex scripts and right-to-left scripts are not supported. Choose this for best performance for editing text in European languages and ideographic languages (Chinese, Japanese, Korean) that is written from left to right without exception.
- Monospaced left-to-right only: Text is always written from left to right and is forced to be monospaced. Complex scripts and right-to-left scripts are not supported. Each character is given the same horizontal width even if the font specifies different widths for different characters. This guarantees columns to be lined up perfectly. To keep the text readable, you should choose a monospaced font.
- ASCII characters with full ideographic width: You can choose this option in combination with any of the four preceding options. In most fonts, ASCII characters (English letters, digits, and punctuation) are about half the width of ideographs. This option substitutes full-width characters for the ASCII characters so they are the same width as ideographs. If you turn this on in combination with “monospaced left-to-right only” then columns that mix English letters and digits with ideographs will line up perfectly.

Text Cursor Movement

- Monodirectional: The left arrow key on the keyboard always moves the cursor to the left on the screen and the right arrow key always moves the cursor to the right on the screen, regardless of the predominant or actual text direction.
- Bidirectional: This option is only available if you have chosen one of the complex script options in the “text layout and direction” list. The direction that the left and right arrow keys move the cursor into depends on the predominant text direction selected in the “text layout and direction” list and on the actual text direction of the word that the cursor is pointing to when you press the left or right arrow key on the keyboard.
 - Predominantly left-to-right: The left key moves to the preceding character in logical order, and the right key moves to the following character in logical order.
 - Actual left-to-right: The left key moves left, and the right key moves right.
 - Actual right-to-left: The actual direction is reversed from the predominant direction. The left key moves right, and the right key moves left.
 - Predominantly right-to-left: The left key moves to the following character in logical order, and the right key moves to the preceding character in logical order.
 - Actual left-to-right: The actual direction is reversed from the predominant direction. The left key moves right, and the right key moves left.
 - Actual right-to-left: The left key moves left, and the right key moves right.

Selection of Words

- Select only the word: Pressing Ctrl+Shift+Right moves the cursor to the end of the word that the cursor is on. The selection stops at the end of the word. This is the default behavior for all Just Great Software applications. It makes it easy to select a specific word or string of words without any extraneous spaces or characters. To include the space after the last word, press Ctrl+Shift+Right once more, and then Ctrl+Shift+Left.
- Select the word plus everything to the next word: Pressing Ctrl+Shift+Right moves the cursor to the start of the word after the one that the cursor is on. The selection includes the word that the cursor was on and the non-word characters between that word and the next word that the cursor is moved to. This is how text editors usually behave on the Windows platform.

Character Sequences to Treat as words

Characters that are considered to be letters or digits by the Unicode standard are selected when you double-click them. Ctrl+Left and Ctrl+Right move the cursor to the start of the preceding or following sequence of letters and digits. Ideographs are considered to be letters.

You can use the six checkboxes to add underscores, hyphens, punctuation other than underscores or hyphens, math symbols, currency symbols, or symbols other than math or currency symbols to the characters that are selected when you double-click them. If you tick all of them then double-clicking selects any run of non-whitespace characters.

If you selected the “bidirectional” text cursor movement option then you can turn on “words determined by complex script analysis” to allow Ctrl+Left and Ctrl+Right to place the cursor between two letters for languages such as Thai that don’t write spaces between words.

Text Cursor Appearance

Select a predefined cursor or click the Configure button to show the text cursor configuration screen. There you can configure the looks of the blinking text cursor (and even make it stop blinking).

A text layout uses two cursors. One cursor is used for insert mode, where typing in text pushes ahead the text after the cursor. The other cursor is used for overwrite mode, where typing in text replaces the characters after the cursor. Pressing the Insert key on the keyboard toggles between insert and overwrite mode.

Main Font

Select the font that you want to use from the drop-down list. Turn on “allow bitmapped fonts” to include bitmapped fonts in the list. Otherwise, only TrueType and OpenType fonts are included. Using a TrueType or OpenType font is recommended. Bitmapped fonts may not be displayed perfectly (e.g. italics may be clipped) and only support a few specific sizes.

If you access the text layout configuration screen from the print preview, then turning on “allow bitmapped fonts” will include printer fonts rather than screen fonts in the list, in addition to the TrueType and

OpenType fonts that work everywhere. A “printer font” is a font built into your printer’s hardware. If you select a printer font, set “text layout and direction” to “left-to-right only” for best results.

Normally, plain text files should not contain control characters other than tabs and line breaks. EditPad has separate settings for visualizing tabs and line breaks. Most fonts are unable to display control characters. Therefore EditPad provides several options for visualizing control characters. The first 3 are independent of the font. The last 3 require support from your main font.

- Two stair-stepped letters: Similar to how EditPad visualizes specific line break styles, such as ^B for backspace.
- Control+letter: Indicates ASCII control character with the Ctrl+letter combination that they are traditionally entered with on character terminals, such as [^]H for backspace. Indicates ISO-8859 control characters as Esc+letter.
- Stair-stepped hexadecimal numbers: Hexadecimal index of the control character in the ASCII or ISO-8859 character tables, such as ⁰8 for backspace.
- IBM PC glyphs: The original IBM PC displayed various graphical symbols when an application wrote an ASCII control character to the screen. If your font supports the Unicode characters that represent these symbols then EditPad can display these symbols. The backspace control character then appears as an inverse bullet ■, for example. ISO-8859 control characters are visualized with two stair-stepped letters. The IBM PC predates ISO-8859.
- Unicode control pictures: The Unicode standard has code points specific to visualizing the ASCII control characters. Few fonts support these. Those that do typically render them as 2 or 3 stair-stepped letters, such as ⁵ for backspace. As you can see, these letters can be tiny with some fonts. Unicode does not have such code points for the ISO-8859 control characters. So EditPad uses its own stair-stepped letter visualization for those.
- Actual control characters: This option tells EditPad to blindly rely on the font to be able to display control characters. With most fonts this makes the control characters completely invisible. This can cause text spacing issues in EditPad. So only use this option if you know your font can render all ASCII and ISO-8859 control characters.

Fallback Fonts

Not all fonts are capable of displaying text in all scripts or languages. If you have selected one of the complex script options in the “text layout and direction” list, you can specify one or more “fallback” fonts. If the main font does not support a particular script, EditPad will try to use one of the fallback fonts. It starts with the topmost font at the list and continues to attempt fonts lower in the list until it finds a font that supports the script you are typing with. If none of the fonts supports the script, then the text will appear as squares.

To figure out which scripts a particular font supports, first type or paste some text using those scripts into the Example box. Make sure one of the complex script options is selected. Then remove all fallback fonts. Now you can change the main font and see which characters that font can display. When you’ve come up with a list of fonts that, if used together, can display all of your characters, select your preferred font as the main font. Then add all the others as fallback fonts.

Line and Character Spacing

By default all the spacing options are set to zero. This tells EditPad to use the default spacing for the font you have selected.

If you find that lines are spaced apart too widely, specify a negative value for “increase (or decrease) the line height“. Set to “add 0 pixels of extra space between lines”.

If you find that lines are spaced too closely together, specify a positive value for “increase (or decrease) the line height” and/or “add ... pixels of extra space between lines”. The difference between the two is that when you select a line of text, increasing the line height increases the height of the selection highlighting, while adding extra space between lines does not. If you select multiple lines of text, extra space between lines shows up as gaps between the selected lines. Adding extra space between lines may make it easier to distinguish between lines.

The “increase (or decrease) the character width by ... pixels” setting is only used when you select “monospaced left-to-right” only in the “text layout and direction” list. You can specify a positive value to increase the character or column width, or a negative value to decrease it. This can be useful if your chosen font is not perfectly monospaced and because of that characters appear spaced too widely or too closely.

Text Cursor Configuration

You can access the text cursor configuration screen from the text layout configuration screen by clicking one of the Configure buttons in the “text cursor appearance” section.

Existing Text Cursor Configurations

The Text Cursor Configuration screen shows the details of the text cursor configuration that you select in the list at the top. Any changes you make on the screen are automatically applied to the selected cursor and persist as you choose different cursors in the list. The changes become permanent when you click OK. The cursor that is selected in the list when you click OK becomes the new default cursor.

Click the New and Delete buttons to add or remove cursors. You must have at least one text cursor configuration. If you have more than one, you can use the Up and Down buttons to change their order. The order does not affect anything other than the order in which the text cursor configurations appear in selection lists.

EditPad comes with a number of preconfigured text cursors. You can fully edit or delete all the preconfigured text cursors if you don’t like them.

- Insertion cursor: Blinking vertical bar similar to the standard Windows cursor, except that it is thicker and fully black, even on a gray background.
- Bidirectional insertion cursor: Like the insertion cursor, but with a little flag that indicates whether the keyboard layout is left-to-right (e.g. you’re typing in English) or right-to-left (e.g. you’re typing in Hebrew). The flag is larger than what you get with the standard Windows cursor and is shown even if you don’t have any right-to-left layouts installed.

- Underbar cursor: Blinking horizontal bar that lies under the character. This mimics the text cursor that was common in DOS applications.
- Overwrite cursor: Blinking rectangle that covers the bottom half of the character. In EditPad this is the default cursor for overwrite mode. In this mode, which is toggled with the Insert key on the keyboard, typing text overwrites the following characters instead of pushing them ahead.
- Standard Windows cursor: The standard Windows cursor is a very thin blinking vertical bar that is XOR-ed on the screen, making it very hard to see on anything except a pure black or pure white background. If you have a right-to-left keyboard layout installed, the cursor gets a tiny flag indicating keyboard direction. You should only use this cursor if you rely on accessibility software such as a screen reader or magnification tool that fails to track any of EditPad's other cursor shapes.

Text Cursor Configuration [X]

Existing text cursor configurations

- Insertion cursor
- Bidirectional insertion cursor
- Underbar cursor
- Overwrite cursor
- Standard Windows cursor

[+ New] [Delete] [Up] [Down]

Selected text cursor configuration

Name of this text cursor configuration: Example:

Shape

☐ Standard Windows text cursor
☒ Vertical bar in front of the character
☐ Vertical bar with a flag indicating keyboard direction
☐ Vertical bar with a flag indicating text direction
☐ Horizontal bar under the character
☐ Half rectangle covering half the character
☐ Rectangle covering the whole character

Blinking style

☒ Do not blink
☐ Blink on and off
☐ Alternate between two colors

Sizes

Width: Height: Flag:

Colors

☒ Regular
☐ Alternate
☒ Dragging

[OK] [Cancel] [Help]

Selected Text Cursor Configuration

Type in the name of the text cursor configuration. This name is only used to help you identify it in selection lists when you have prepared more than one text cursor configuration.

In the Example box you can type in some text to see what the cursor looks like. The box has a word in Latin and Arabic so you can see the difference in cursor appearance, if any, based on the text direction of the word that the cursor is on.

Shape

- **Standard Windows Text cursor:** The standard Windows cursor is a very thin blinking vertical bar that is XOR-ed on the screen, making it very hard to see on anything except a pure black or pure white background. If you have a right-to-left keyboard layout installed, the cursor gets a tiny flag indicating keyboard direction. You should only use this cursor if you rely on accessibility software such as a screen reader or magnification tool that fails to track any of EditPad's other cursor shapes. The standard Windows cursor provides no configuration options.
- **Vertical bar in front of the character:** On the Windows platform, the normal cursor shape is a vertical bar that is positioned in front of the character that it points to. That is to the left of the character for left-to-right text, and to the right of the character for right-to-left text.
- **Vertical bar with a flag indicating keyboard direction:** A vertical bar positioned in front of the character that it points to, with a little flag (triangle) at the top that indicates the direction of the active keyboard layout. When the cursor points to a character in left-to-right text, it is placed to the left of that character. When the cursor point to a character in right-to-left text, it is placed to the right of that character. The direction of the cursor's flag is independent of the text under the cursor. The cursor's flag points to the right when the active keyboard layout is for a left-to-right language. The cursor's flag points to the left when the active keyboard layout is for a right-to-left language.
- **Vertical bar with a flag indicating text direction:** A vertical bar positioned in front of the character that it points to, with a little flag (triangle) at the top that points to that character. When the cursor points to a character in left-to-right text, it is placed to the left of that character with its flag pointing to the right towards that character. When the cursor point to a character in right-to-left text, it is placed to the right of that character with its flag pointing to the left towards that character.
- **Horizontal bar under the character:** In DOS applications, the cursor was a horizontal line under the character that the cursor points to.
- **Half rectangle covering half the character:** The cursor covers the bottom half of the character that it points to. This is a traditional cursor shape to indicate typing will overwrite the character rather than push it ahead.
- **Rectangle covering the whole character:** The cursor makes the character invisible. This can also be used to indicate overwrite mode.

Blinking Style

- **Do not blink:** The cursor is permanently visible in a single color. Choose this option if the blinking distracts you or if it confuses accessibility software such as screen readers or magnification tools.
- **Blink on and off:** The usual blinking style for text cursors on the Windows platform. The cursor is permanently visible while you type (quickly). When you stop typing for about half a second, the

cursor blinks by becoming temporarily invisible. Blinking makes it easier to locate the cursor with your eyes in a large block of text.

- **Alternate between two colors:** Makes the cursor blink when you stop typing like “on and off”. But instead of making the cursor invisible, it is displayed with an alternate color. This option gives the cursor maximum visibility: the blinking animation attracts the eye while keeping the cursor permanently visible.

Sizes

- **Width:** Width in pixels for the vertical bar shape.
- **Height:** Height in pixels for the horizontal bar shape.
- **Flag:** Length in pixels of the edges of the flag that indicates text direction.

Colors

- **Regular:** Used for all shapes and blinking styles except the standard Windows cursor.
- **Alternate:** Alternate color used by the “alternate between two colors” blinking style.
- **Dragging:** Color of a second “ghost” cursor that appears while dragging and dropping text with the mouse. It indicates the position the text is moved or copied to when you release the mouse button.

Options | Right-to-Left

Select Right-to-Left in the Options menu to quickly toggle the active file between left-to-right and right-to-left editing. What this command really does is switch between the “default text layout” and the “layout for opposite text direction” that you selected for the active file’s file type on the Editor page in the file type configuration.

If you have at least one right-to-left keyboard layout installed in Windows, then you can also do this with the Ctrl and Shift keys on the keyboard. To switch to left-to-right, hold down either Ctrl key while pressing and releasing the left-hand Shift key. To switch to right-to-left, hold down either Ctrl key while pressing the right-hand Shift key. These key combinations too switch between the “default text layout” and the “layout for opposite text direction”.

Options | Stay On Top

If you turn on Options | Stay On Top, EditPad’s window remains visible even when you switch to another application.

Options | Export Preferences

Use Options | Export Preferences to save your preferences into an .ini file. This allows you to transport them to another computer. The exported settings include those you make in Options | Preferences and Options | Configure File Types. In EditPad Pro, they also include your tools and your macros.

The exported settings do not include transient information such as history lists. It also does not include the size and layout of EditPad's window as that is dependent on monitor size and resolution.

The Export Preferences and Import Preferences commands are intended for sharing settings between team members and for migrating settings from an older version of EditPad to a newer version of EditPad. They are not intended to backup up and restore EditPad's exact state. If you want to do that, back up and restore all the files in the folder %APPDATA%\JGsoft\EditPad Pro 8 under your Windows user profile.

Options | Import Preferences

Imports the preferences and file type configuration from an .ini file exported with Options|Export Preferences. In EditPad Pro, this also imports tools and macros.

EditPad 8 can import preferences exported by EditPad 5, 6, 7, or 8. Importing preferences from EditPad 5, 6, or 7 into EditPad 8 may not result in the exact same settings as you had in the previous versions. Some settings may work differently in EditPad 8. Settings that are new in EditPad 8 (and thus can't be imported from older preferences) will either be reset to their defaults or be turned off, depending on whether the related functionality existed in the previous EditPad version at all.

14. Options | Configure File Types

One of the most powerful aspects about the way EditPad Lite and Pro work with preferences is that many settings can be made for each specific file type that you regularly edit or view with EditPad. For example: you may want to use word wrapping at the window border for plain text files, but no word wrapping for source code. Once you make the correct settings for each file type, you won't have to change them each time you open a file.

When you open or save a file, EditPad compares its name against the file mask for each known file type and applies the settings made for that file type. If the file name doesn't match any file type, the settings for "Unspecified file type" are used.

EditPad also uses the list of file types to build the "Files of type" drop-down list at the bottom of the file selection windows for opening and saving files. The filters in the Explorer Panel, the FTP Panel, the Open Folder command, and the Find on Disk command also allow you to select files based on their file types.

To configure file types, select Configure File Types in the Options menu. The file type configuration dialog appears. The dialog is divided into two parts. At the left-hand, you'll see a list of currently defined file types. At the right-hand are four tabs that hold the settings for the file type that's currently selected in the list at the left.

To change a file type's settings, simply click on it in the list and make the changes you want. Hold down the Shift or Ctrl key to select multiple file types. Any changes you make are applied to all selected file types. To create a new file type, click the New button below the list and set the options as you want them. To clone or duplicate a file type, hold down the Control key on the keyboard while clicking the New button.

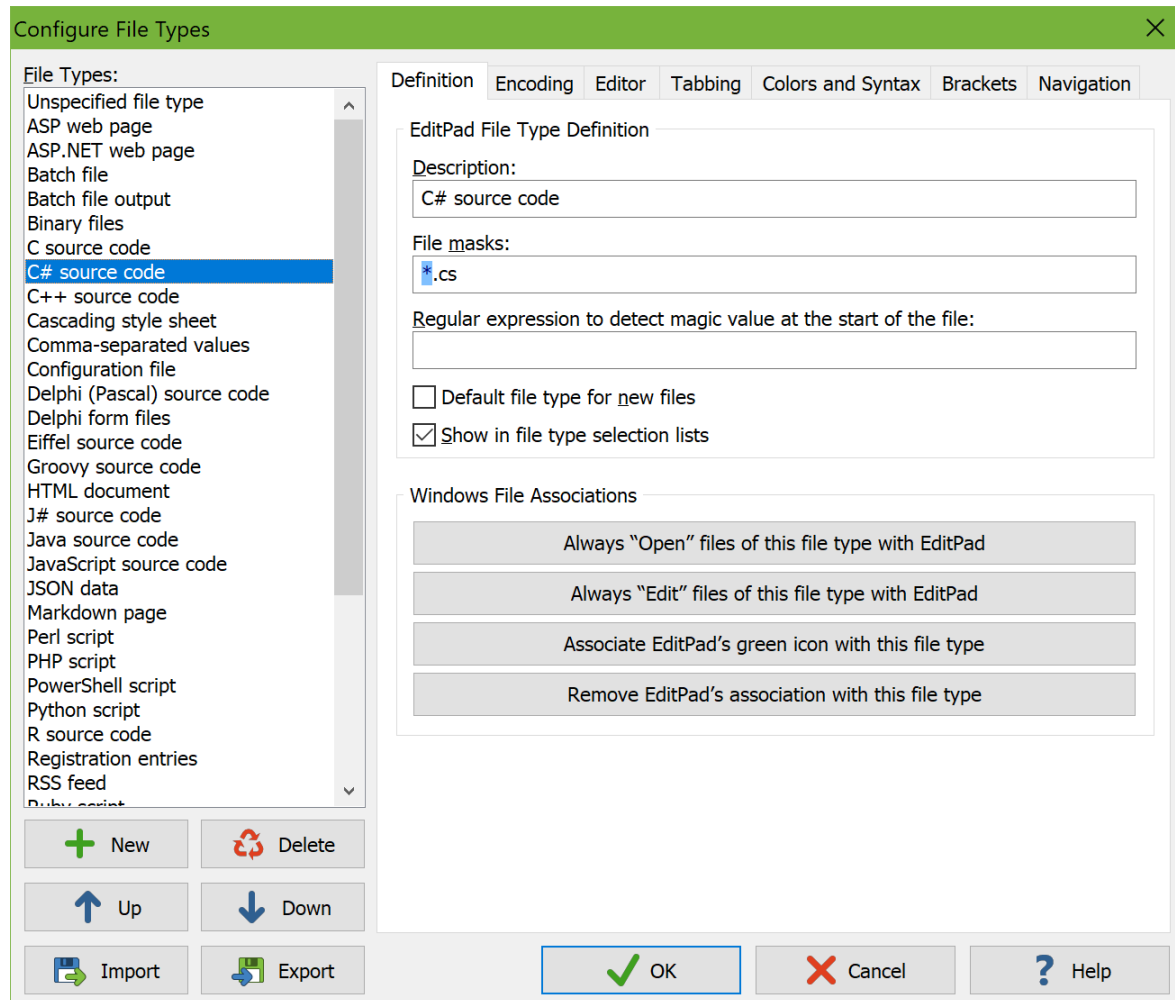
To delete a file type, click on it and click the Delete button. Before deleting a file type you don't use, consider if you should simply untick "show in file type selection lists" to hide the file type without disabling it.

The order of the file types matters. The order you give them in the configuration screen is the order they'll have in any drop-down list showing the available file types. If a file matches the file mask of more than one file type, then EditPad uses the bottommost file type in the list with a matching mask for that file. Place file types with more specific file masks below those with more general file masks. The "unspecified file type" file type is the most general one, and always sits at the top of the list. The other file types defined in EditPad's default setting have no overlap in their file masks, so their order doesn't matter. They're ordered alphabetically by default. If you edit the file masks of the predefined file types or add your own file types, you may have to reorder them if the file masks of two file types can match a single file.

You can save individual file type configurations into a file that you can share with other people. To save a file type, click on it in the list and click the Export button below the list. If you select multiple file types, all of them are exported. To load a file type configuration file you've received from somebody, click the Import button. The loaded file type will be added to the list.

File Type Definition

On the Definition tab in the file type configuration, you can indicate how the file type should be identified and which files it applies to.



The **description** is used whenever you need to make a choice between file types, such as the “files of type” list in open and save windows or the Options | File Type command. You cannot change the description of the first few file types. These file types have a special meaning in EditPad. You cannot delete these file types or change their position in the list, but you can adjust all the settings to your own tastes and habits as you can do for all the other file types.

The **file masks** you assign to a file type are particularly important. When you open a file, EditPad compares the file’s name against the file masks for each file type. If it finds a file type with a mask that matches the file name, that file type’s settings will be used for the file being opened. If more than one file type’s mask matches, the *bottommost* matching file type is used. Therefore, file types with more specific masks should be placed below file types with more general masks. The file type named “unspecified file type” has a mask of “*.*” that matches all file names. So “unspecified file type” needs to be at the top of the list so that it will be the last file type that EditPad checks when looking for a matching file type.

In a file mask, the asterisk (*) represents any number (including none) of any character. The question mark (?) represents a single character. On the Windows platform, the type of a file is usually determined by its name's extension. The extension consists of a dot followed by (usually) three letters. E.g. text files have a .txt extension. The file mask *.txt will match all text files with a .txt extension. You can delimit multiple file masks with semicolons. The file mask *.c;*.h matches all C source and header files (with a .c and .h extension respectively). If you specify a semicolon-delimited list of file masks, a file's name needs to match only one of them for the file type to be applied to that file.

File masks also support a simple character class notation, which matches one character from a list or a range of characters. E.g. a file mask such as www.200509[0123][0-9].log or www.200509??.log could be used to match all web logs from September 2005.

Sometimes, a file's type cannot be easily derived from its name. While file name extensions are common on the Windows platform, they're not on other platforms like UNIX. For such file types, you can specify a **magic value** regular expression. A magic value is simply some text or data at the start of a file that reveals the file's type. A regular expression is a pattern for matching text.

When EditPad has finished comparing the file's name against the file masks of all file types, and the only matching file type is "unspecified file type", EditPad will try to match the magic value regular expression of each file type at the start of the file. The regex is only attempted at the very start of the file, as if the regular expression started with the anchor `\A`. Again, should more than one file type have a matching regular expression, the bottommost file type will be used.

Out of the box, EditPad Pro ships with several file types with magic value regular expressions. The "HTML" file type uses `(?i)\s*<[!DOCTYPE\s+]?HTML` to match a `<!DOCTYPE HTML` or `<HTML` tag at the start of the file, case insensitively. The "Perl script" file type has `#![-.\s/a-zA-Z0-9]*(?:/env)?perl` which matches the "shebang" at the start of a Perl script. On the UNIX platform, Perl scripts usually don't have an extension, but do have the shebang. On the Windows platform, the shebang is typically missing, but Perl scripts are given a .pl extension. EditPad will recognize the file either way, first trying the file masks to look for the .pl extension, and then trying the regular expression to match the shebang. The "XML" file type uses `<[?xml | \s*<[a-zA-Z0-9_]+ \s+ +[^\<>]*?xmlns\s*=` to match the `<?xml` declaration or a root tag with the `xmlns` attribute. XML files often have an extension that indicates the application that saved the file, rather than the fact that it's in XML format. E.g. PowerGREP uses the .pgf, .pga, .pgr and .pgl extensions for PowerGREP file selections, actions, results and libraries, rather than the generic .xml extension.

Check "**default for new files**" to use the file type for new files created by clicking the File menu directly or by pressing its keyboard shortcut Ctrl+N.

Check "**show in file type selection lists**" if you want the file type to appear in lists where you can select a file type. That includes the file type drop-down lists in all file dialogs such as those used by File|Open and File|Save As. It also includes the file type submenus of File|New and Options|File Type. You should check this option for the file types that you usually work with, but not for others. This way the file type selection lists remain uncluttered. The file types that you hide remain fully functional. Their file masks and magic regexes are still used when detecting the file types of files you open. You can still save files of this type by manually entering their extension in the Save As dialog.

File Type Associations

Using the three buttons on the Definition tab in the file type configuration, you can quickly create or remove file associations to the current file type. The associations will be made with or removed from each of the extensions listed in the file type's file mask.

Windows itself determines file types by file name extension only. Therefore, associations will only be made with file masks in the style of *.ext. Other file masks are ignored when making Windows file associations.

If you click the button **“Always ‘Open’ this file type with EditPad”**, EditPad will associate itself with the “open” action. This means that whenever you double-click on a file of this type in Windows Explorer, it will be opened in EditPad.

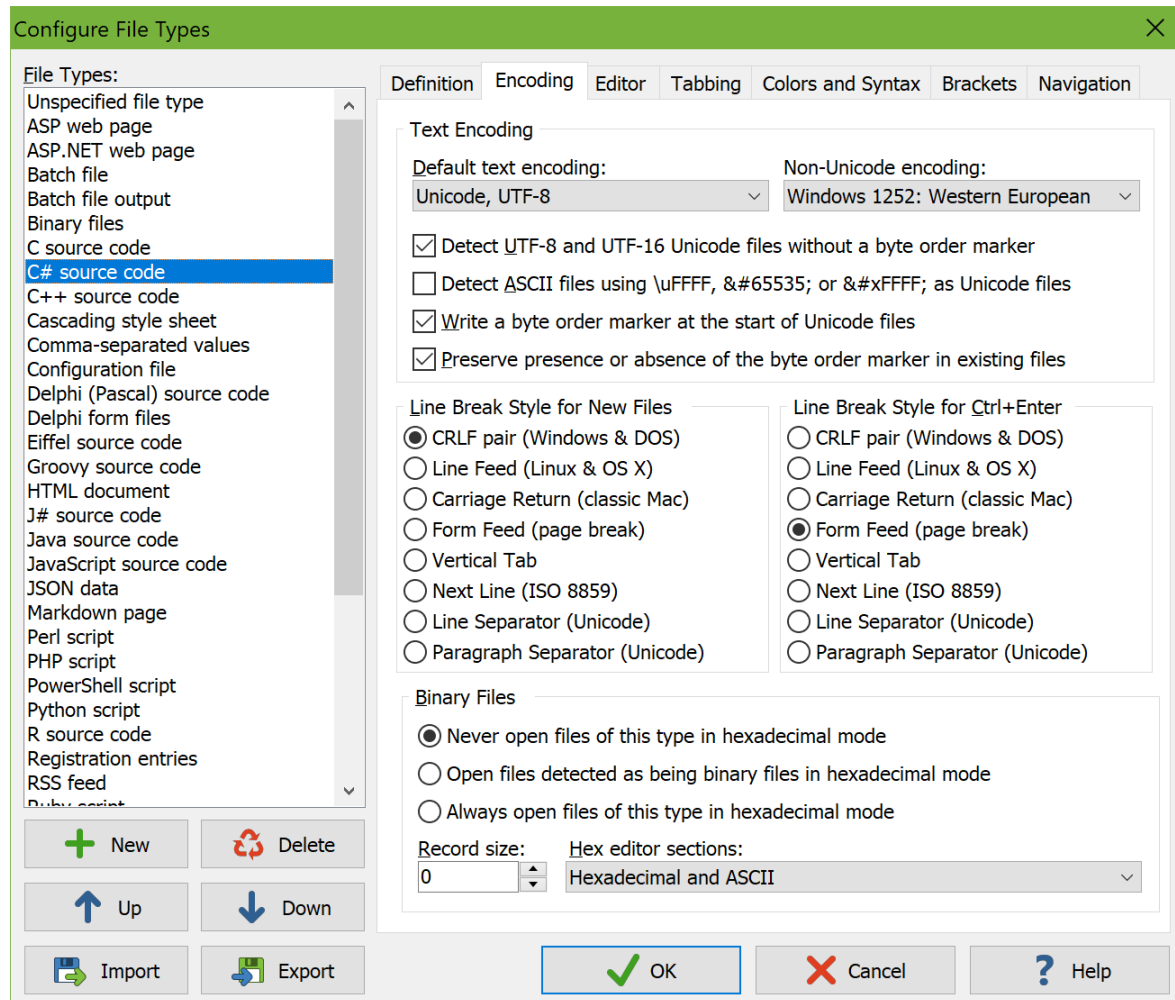
If you click the button **“Always ‘Edit’ this file type with EditPad”**, EditPad will associate itself with the “edit” action. This means that you can then right-click on files of this type in Windows Explorer and select “Edit” from the context menu to open the file in EditPad. If you do this for HTML files, you will be able to view the source for any web page by clicking on the Edit button in the toolbar of Microsoft Internet Explorer.

If you click **“Associate EditPad’s green icon with this file type”**, then Windows Explorer will indicate files of this type with the green notebook icon that represents EditPad itself. You can use this button to make files that you edit exclusively with EditPad stand out in Windows Explorer or other file management applications. EditPad itself also uses the icon associated with a file type to indicate file types on tabs and file management side panels.

Click the button **“Remove EditPad’s association with this file type”** to remove the associations made by clicking on one or more of the three previous buttons. Associations made in another way will not be removed.

File Type Encoding

On the Encoding tab in the file type configuration, you can indicate how EditPad should encode and decode files of a particular type.



Computers deal with numbers, not with characters. When you save a text file, each character is mapped to a number, and the numbers are stored on disk. When you open a text file, the numbers are read and mapped back to characters. When saving a file in one application, and opening that file in another application, both applications need to use the same character mappings.

Traditional character mappings or code pages use only 8 bits per character. This means that only 256 distinct characters can be represented in any text file. As a result, different character mappings are used for different languages and scripts. Since different computer manufacturers had different ideas about how to create character mappings, there's a wide variety of legacy character mappings. EditPad supports a wide range of these.

In addition to conversion problems, the main problem with using traditional character mappings is that it is impossible to create text files written in multiple languages using multiple scripts. You can't mix Chinese,

Russian, and French in a text file, unless you use Unicode. Unicode is a standard that aims to encompass all traditional character mappings, and all scripts used by current and historical human languages.

If you only edit files created on your own Windows computer, or on other Windows computers using the same regional settings, there's not much to configure. Simply leave the “**default text encoding**” set to the default Windows code page, e.g. Windows 1252 for English and other Western European languages. If you edit files created on Windows computers with different regional settings, you may need to select a different Windows code page. You can either change the default for the file type, or use Convert|Text Encoding for a one-time change.

Unicode

On the Windows platform, Unicode files should start with a byte order marker. It is often abbreviated as BOM. It is also known as a Unicode signature. The byte order marker is a special code that indicates the Unicode encoding (UTF-8, UTF-16 or UTF-32) used by the file. EditPad always detects the byte order marker and always treats the file with the corresponding Unicode encoding.

Unfortunately, some applications cannot deal with files starting with a byte order marker. XML parsers are a notorious example. If an application that claims to support Unicode fails to read Unicode files saved by EditPad, try turning on the option not to write the byte order marker.

If you turn on the option “**preserve presence or absence of the byte order marker in existing files**”, then EditPad will keep the BOM in files that already have it, and never add it to Unicode files previously saved without a BOM. When preserving the BOM or its absence, the option “write a byte order marker at the start of Unicode files” is only used for files that you newly create with EditPad and for files that you convert from a non-Unicode encoding to Unicode. Non-Unicode files cannot have a BOM. So there is no presence or absence of the BOM to maintain if the file was not previously Unicode.

If you want EditPad to open Unicode files saved without a byte order marker, you'll either need to set the default encoding for the file type to the proper Unicode encoding, or turn on the option to auto-detect UTF-8 and UTF-16 files without a byte order marker.

To auto-detect UTF-8 files, EditPad Pro checks if the file contains any bytes with the high order bit set (values 0x80 through 0xFF). If it does, and all the values define valid UTF-8 sequences, the file is treated as a UTF-8 file. The chances of a normal text document written in a Windows code page being incorrectly detected as UTF-8 are practically zero. Note that files containing only English text are indistinguishable from UTF-8 when encoded in any Windows, DOS or ISO-8859 code page. This is one of the design goals of UTF-8. These files contain no bytes with the high order bit set. EditPad Pro will use the file type's default code page for such files. This makes no difference as long as you don't add text using letters outside the English alphabet.

Reading a UTF-16 file as if it was encoded with a Windows code page will cause every other character in the file to appear as a NULL character. These will show up as squares or spaces in EditPad. EditPad can detect this situation in many cases and read the file as UTF-16. However, for files containing genuine NULL characters, you may need to turn off the option to detect UTF-8 and UTF-16 files without the byte order marker.

Some file formats consist of pure ASCII with non-ASCII characters represented by Unicode escapes in the form of \uFFFF or by numeric character references in the form of or . EditPad Pro has ASCII + \uFFFF and ASCII + NCR text encodings that you can use to edit such files showing the actual

Unicode characters in EditPad, but saving the Unicode escapes or numeric character references in the file. Turn on “Detect ASCII files using \uFFFF, or as Unicode files” to automatically use these encodings for files that consist of pure ASCII and that contain at least one of these Unicode escapes or numeric character references. By default, this option is only on for Java source code, because in Java there is no difference between a Unicode escape and the actual Unicode character. You can also turn it on for HTML or XML if you like to write your HTML and XML files in pure ASCII with character references.

If you set the default encoding to Unicode and you open a file without a BOM then EditPad uses the same auto-detection methods to verify that the default encoding is correct for the file. It verifies the default encoding even if you disabled the auto-detection methods. If the bytes in the file are not valid for the default encoding then EditPad uses the detection methods that you did enable to try the other Unicode encodings. If those fail too, then EditPad falls back on the “**non-Unicode encoding**” that you selected for the file type. This way selecting UTF-16 as your default encoding doesn’t cause files using a Windows code page to appear as Chinese gibberish. You can select the Windows code page as the non-Unicode encoding to fall back on.

If the presence or absence of a BOM is important to you then you should turn on the Unicode signature status bar indicator. It indicates whether the file has a BOM or not. You can click the indicator to add or remove the BOM if you selected to preserve its presence or absence for the active file’s file type.

Line Break Style

If the differences in character mappings weren’t enough, different operating systems also use different characters to end lines. EditPad automatically and transparently handles all eight line break styles that Unicode text files may use. If you open a file, EditPad maintains that file’s line break style when you edit it. EditPad only changes the line break style if you use Convert | Line Break Style.

Unfortunately, many other applications are not as versatile as EditPad. Most applications expect a file to use the line break style of the host operating system. Many Windows applications display all text on one long line if a file uses UNIX line breaks. The Linux shell won’t properly recognize the shebang of Perl scripts saved in Windows format (causing CGI scripts to break “mysteriously”).

In such situations, you should set a default line break style for the affected file types. When you create a new file by selecting a file type from the drop-down menu of the new file button on the toolbar, EditPad gives that file the default line break style of the chosen file type.

Note that EditPad never silently converts existing line breaks to a different style when you open a file. If you set the default line break style for Perl scripts to UNIX, and then open a Perl script using Windows line breaks, then EditPad saves that script with Windows line breaks unless you use Convert | Line Break Style | To UNIX.

If you need to deal with different line break styles, you should turn on the line break style status bar indicator. It indicates the line break style being used by the current file and whether it uses that style exclusively or mixed with other styles.

Binary Files

EditPad Pro can edit binary files in hexadecimal mode. If you know that files of a certain type don’t contain (much) human-readable content, select “always open files of this type in hexadecimal mode”. If you know

files of a certain type to be text files, select “never open files of this type in hexadecimal mode” to prevent stray NULL characters from making EditPad think the file is binary.

If you’re not sure, select “open files detected as being binary in hexadecimal mode”. Then EditPad Pro will check if the file contains any NULL characters. Text files should not contain NULL characters, though improperly created text files might. Binary files frequently contain NULL characters.

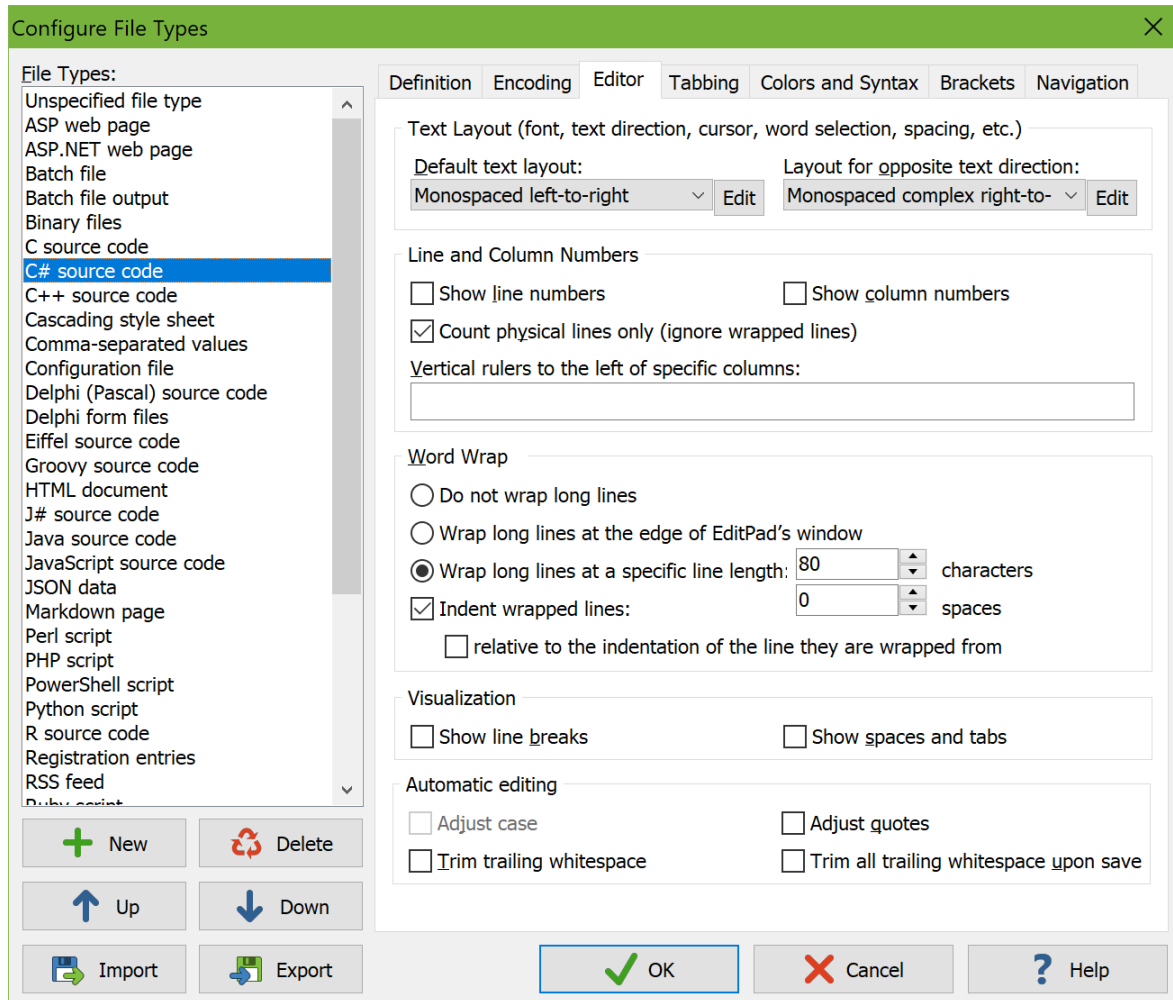
Making the wrong choice here causes no harm. You can instantly switch between text and hexadecimal mode by picking View|Hexadecimal in the menu. Unlike many other editors, EditPad Pro will preserve the exact contents of binary files even when you view them in text mode. Even files with NULL characters will be properly displayed in text mode. (Many applications truncate text files at the first NULL, since that character is often used as an end-of-data signal.)

The “**record size**” is the number of bytes that EditPad Pro displays on each line in hexadecimal mode. If you enter zero, you get EditPad Pro’s default behavior of showing the smallest multiple of 8 bytes that fits within the width of EditPad’s window. If you enter a positive number, that’s the number of bytes EditPad Pro displays on a line. You can enter any number. It doesn’t have to be divisible by 8 or by 2.

Set “**hex editor sections**” to “hexadecimal and ASCII” to get the typical hex editor view with the hexadecimal representation of the bytes in the center of the editor, and the ASCII representation of the bytes in the right-hand column. Choose “hexadecimal only” or “ASCII only” to see only either representation. Select “split hexadecimal and ASCII” if you want one view to display the hexadecimal representation and the other view the ASCII representation after using View|Split Editor. If the editor is not split, there is no difference between the “split hexadecimal and ASCII” and “hexadecimal and ASCII” choices.

File Type Editor Options

On the Editor tab in the file type configuration, you indicate the default editor options for each file type. Many of these options can be changed for individual files via the Options menu. The Options menu only affects a single file. The file type configuration sets the default.



Text Layout

In EditPad, a “text layout” is a combination of settings that control how text is displayed and how the text cursor navigates through that text. The settings include the font, text direction, text cursor behavior, which characters are word characters, and how the text should be spaced.

You can select two text layouts for each file type. The “default text layout” is used when you first open a file of this file type. It can be a left-to-right or right-to-left text layout. The “layout for opposite text direction” is used when you turn on Options | Right-to-Left. This layout must be right-to-left if the default is left-to-right, and left-to-right if the default is right-to-left.

You can select previously configured text layouts from the drop-down lists. Click the Edit button next to the list to edit the text layout configurations. The list of text layouts is shared by all file types. If you edit a text layout configuration that is used by multiple file types, the changes apply to all those file types.

See the section describing the Options|Text Layout menu item for a full explanation of the text layout configuration screen.

Line and Column Numbers

Turn on “show line numbers” to have EditPad display a number before each line in the left margin. In that case, you need to indicate if you want visible lines or physical lines to be numbered. This only makes a difference when word wrapping is on. When numbering visible lines, each line is counted, including lines created by the word wrapping. When numbering physical lines, the lines are numbered as if word wrapping would be off. You can toggle line numbering for a single file with Options|Line Numbers.

Turn on “show column numbers” if EditPad should display a horizontal ruler above the file indicating column numbers. You can toggle this for a single file with Options|Column Numbers. Column numbers are only displayed when using a fixed-width font.

In the box labeled “vertical rulers to the left of specific columns” you can enter a comma-delimited list of column numbers. EditPad will draw a vertical line to the left of the columns you enter. This makes it easier to work with files that use a columnar layout. You can put a + before a number to make that column relative to the previous column. E.g.: 10,+5,+5,37,+7,+12 would be equivalent to 10,15,20,37,44,56.

Word Wrap

The word wrap setting determines what happens with lines that are too long to fit inside EditPad’s window. If you don’t want long lines to wrap, they will extend beyond EditPad’s window and be partially invisible. A horizontal scroll bar will appear.

You can make long lines wrap either at the edge of EditPad’s window, or at a specific line length. When wrapping at the window edge, lines will be automatically re-wrapped when you resize EditPad. When wrapping at a specific line length, horizontal scrolling may still be needed if the specified length doesn’t fit inside EditPad’s width.

It is usually convenient to wrap at the window edge, as it eliminates the need for horizontal scrolling. You’ll probably want to turn it off for files that use a line-based structure such as software source code, so the word wrapping doesn’t confuse your view of the lines. Note that EditPad’s word wrapping is fully dynamic. Line breaks introduced by the word wrapping are not saved into the file at all, unless you use Convert|Wrapping ⇒ Line Breaks. So there’s no risk at data loss when turning on word wrap for line-based files.

If you turn on “wrapped lines should maintain indentation”, lines created by the word wrapping are indented a certain number of spaces. Enter a positive number of spaces. If you also tick “relative to the indentation of the line they are wrapped from” then they are by the same amount as the line they were broken off from, plus or minus the number of spaces you enter (which can be zero). If you turn it off, wrapped lines will start at the first column regardless of the indentation of the original line. Turning on this option is useful when turning on word wrap for files using nested block structures, such as XML files and various programming languages.

Word wrap can be toggled for individual files with Options|Word Wrap and Options|Indent Wrapped Lines.

Visualization

Turn on “show line breaks” if you always want to see a ¶ character at the end of each paragraph, indicating the invisible line break character(s). Note that if line break characters are selected, the paragraph symbol will always appear to show you that they have indeed been selected. You can quickly change this option by picking Options|Paragraph Symbol from the menu.

Select “show spaces and tabs” if you want spaces and tabs to be visualized. Spaces will be indicated by a vertically centered dot. Tabs are indicated by a » character. This option is useful when working with files where extraneous whitespace can lead to problems, or where the difference between tabs and spaces matters. You can quickly toggle it by picking Options|Visualize Spaces from the menu.

Automatic Editing

Turn on “adjust case” if you want the case conversion rules specified by the syntax coloring scheme to be applied automatically to text that you enter. This option is disabled if the scheme you selected on the Colors and Syntax page does not specify any case conversion rules. This option determines the default state of the Convert|Case|Auto Adjust Case menu item. See the topic on that menu item for more details on automatic case conversion.

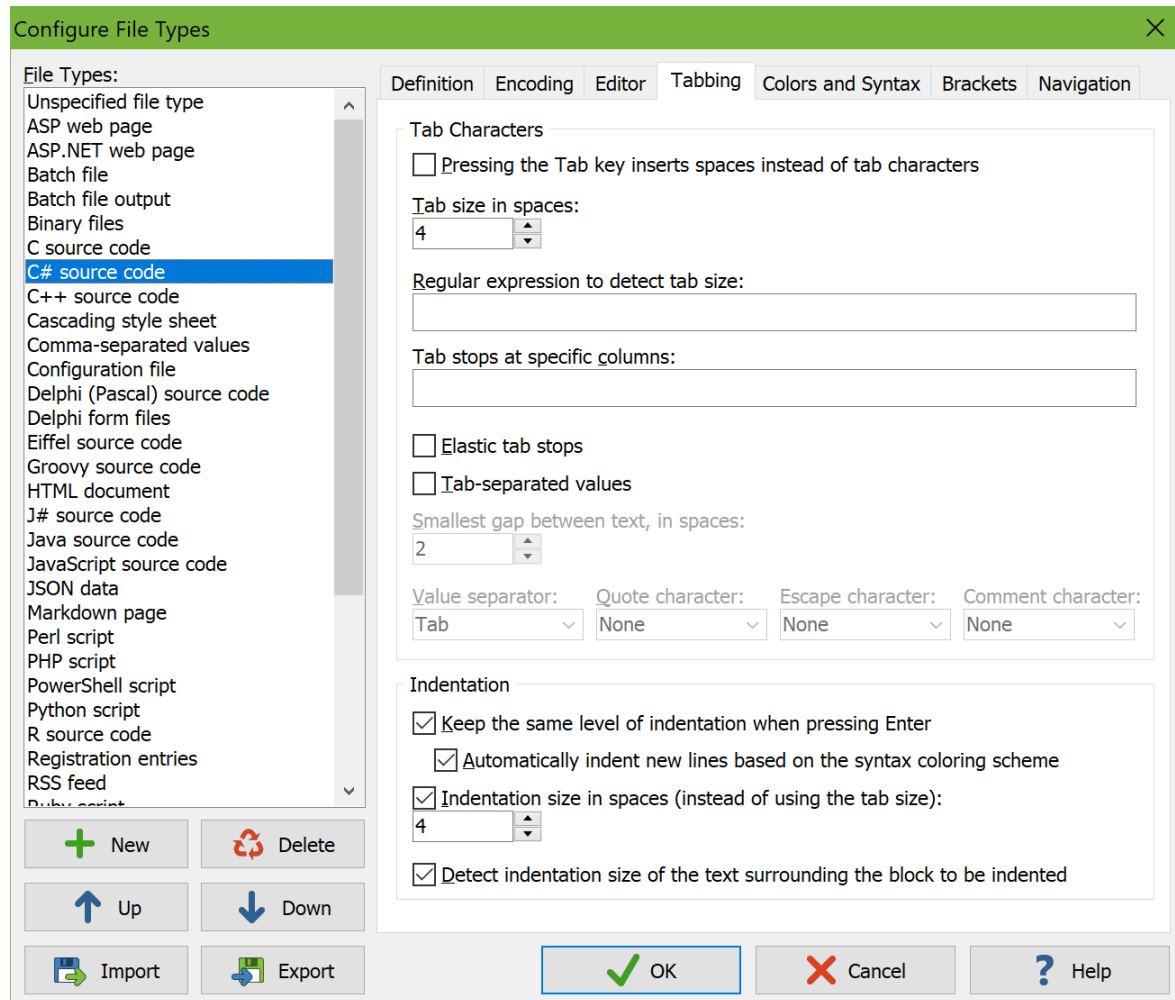
Turn on “adjust quotes” if you want the quote conversion rules specified by the syntax coloring scheme to be applied automatically to text that you enter. This option is disabled if the scheme you selected on the Colors and Syntax page does not specify any quote conversion rules. This option determines the default state of the Convert|Quotes|Auto Adjust Quotes menu item. See the topic on that menu item for more details on automatic quote conversion.

Turn on “trim trailing whitespace” to automatically trim whitespace from the end of any line that you edit when you move the cursor away from that line. This option does not trim trailing whitespace that was already present in the file from lines that you don’t edit. You can toggle this for the active file with Extra|Auto Trim Trailing Whitespace.

Turn on “trim all trailing whitespace upon save” to automatically invoke Extra|Trim Trailing Whitespace when you save a file. This trims all trailing whitespace from all lines.

File Type Tabbing

On the Tabbing page in the file type configuration you can specify tab stops and indentation sizes for files of the selected file type.



Tab Characters

Normally, pressing the Tab key on the keyboard inserts a special tab character that has a variable width depending on the column where you insert it. The tab's width can change if its column changes when you edit the line. Its width can also be different when others view your file with different tab stops.

If you turn on “pressing Tab inserts spaces instead of tab characters” then EditPad Pro inserts the number of spaces until the next tab stop when you press the Tab key.

“Tab size in spaces” is the maximum width of a tab character expressed in an equivalent number of spaces. A typical value is 8 spaces. This puts a tab stop at every 8th column. A tab is then between 1 and 7 spaces wide so that the character after the tab is at the next tab stop. This option also determines the number of spaces that is inserted if you turn on “pressing Tab inserts spaces”.

When you open a file EditPad looks for the first match of the “regular expression to detect tab size” in the first 4 KB of the file. If your files contain some kind of comment to indicate the tab size then you can provide a regex to match this comment and automatically adjust the tab size to that comment. The regex should have a capturing group named “tab”. If this group matches a number between 2 and 32 then that is used as the tab size in spaces. If the group does not match a number but it matches between 2 and 8 characters then the number of characters matched by the group is used as the tab size in spaces. It doesn’t matter what those characters are as long as the group didn’t match an integer number. If the regex also contains a group named “tabspace” and that group matched some text, then that text overrides the option to make the Tab key insert spaces. If the group “tabspace” matched “0”, “false”, “tab”, or “no” then the option is turned off. Otherwise it is turned on.

You can also place tab stops at specific columns by entering a comma-delimited list of column numbers. Pressing Tab then makes the cursor jump to the next tab stop column, by inserting either a single tab character or the number of spaces needed to reach the next tab stop (depending on the “pressing Tab inserts spaces” option). After the last specific column tab stops are spaced according to the “tab size in spaces” setting. You can put a + before a number to make that tab stop relative to the previous tab stop. E.g.: 10,+5,+5,37,+7,+12 would be equivalent to 10,15,20,37,44,56.

Elastic Tab Stops

Traditional fixed tab stops, like you can set in EditPad with the above options, make each tab a certain number of spaces wide. This works well when using tabs for indentation. It even works well if others view your file with a different tab size if your file only uses tabs for indentation.

But if you want to use tabs to line up columns, then your columns aren't likely to still line up nicely when somebody else views your file with a different tab size. You could work around this by using spaces. But then you'll have to use a monospaced font to make sure your spaces and your characters line up correctly.

Elastic tab stops are the solution to neatly lining up columns irrespective of tab size and font spacing. The screen shot shows a source code file using a proportionally spaced font. The file uses tabs for indentation, to line up function parameters (int start and int length) and to line up the /* and */ that delimit comments. Spaces and tabs are visualized. You can clearly see that all the columns are lined up with only a single tab.

The way this works is that when adjacent lines use the same tab stop, that tab stop is moved to the same position on all those lines. That position is the furthest position to the right that is needed to give each tab that uses that tab stop its minimum width on all lines in that block of adjacent lines. For tabs used for indentation, the minimum width is determined by the “tab size in spaces” and the “tab stops at specific columns” settings above. For tabs used in the middle of a line, the minimum width is the “smallest gap between text, in spaces” that you specify.

You can see this in the screen shot. The tabs on lines 1 and 18 use the “smallest gap” size because they are rightmost tabs on lines 1–2 and 18–19. Though the tabs on lines 2 and 19 are indentation tabs, they still use the elastic tab stop that was pushed out further to the right by lines 1 and 18. You can also see that the tab stops on lines 1–2 and on lines 18–19 are not at the same position as the tab stops on any other lines. Because line 3 has no tab stops, the tab stop on lines 1–2 is independent of any other tab stops. The tab stop on lines 18–19 is also independent because lines 17 and 20 have no tabs.

```

1  int someDemoCode(» int start,
2      »              int length)
3  {
4      »      x()»              /*.try.making» */
5      »      print("hello!")» /*.this.comment» */
6      »      »              »
7      »      doSomethingComplicated()» /*.a.bit.longer» */
8      »      for(i in range(start,length))
9      »      {
10     »          »      if(isValid(i))
11     »          »      {
12     »              »      count++
13     »          »      }
14     »      }
15     »      return count
16 }
17
18 void editedDemoCode(» int start,
19     »               int length)
20 {
21     »      x()»              /*.we.have.made» */
22     »      print("hello!")» /*.this.comment» */
23     »      »              »
24     »      doSomethingSimple()» /*.a.little.bit.longer» */
25     »      for(i in range(start,length))
26     »      {
27     »          »      if(isValid(i))
28     »          »      {
29     »              »      count++
30     »          »      }
31     »      }
32     »      return count
33 }

```

The first tab stop on lines 4 to 14 uses the “tab size in spaces” because that tab stop is only used for indentation on those lines. The second tab stop on lines 4 to 7 is determined by the tab on line 7 as it is the furthest to the right. The second tab stop on lines 10 to 13 is independent of the second tab stop on lines 4 to 7 because lines 8 and 9 have only one tab.

Elastic tab stops are moved immediately as you edit your file. Making some edits is the best way to get the hang of how elastic tab stops work. If you change "someDemoCode" into "editedDemoCode" then you shift the tab on line 1 to the right. This shifts the tab stop to the right. The spacing of the tab on line 2 automatically and immediately adjusts to the new position of the tab stop. If you shorten “doSomethingComplicated” into “doSomethingSimple” then the comments on lines 4 to 7 immediately shift to the left. If you insert “little” into “a bit longer” then you end up shifting the tab on line 7 beyond the tab

on line 5. This immediately moves the 3rd tab stop on lines 4 to 6 to line up with the new 3rd tab stop position on line 7. Throughout the edit the 3 */ delimiters continue to be neatly lined up by their tabs.

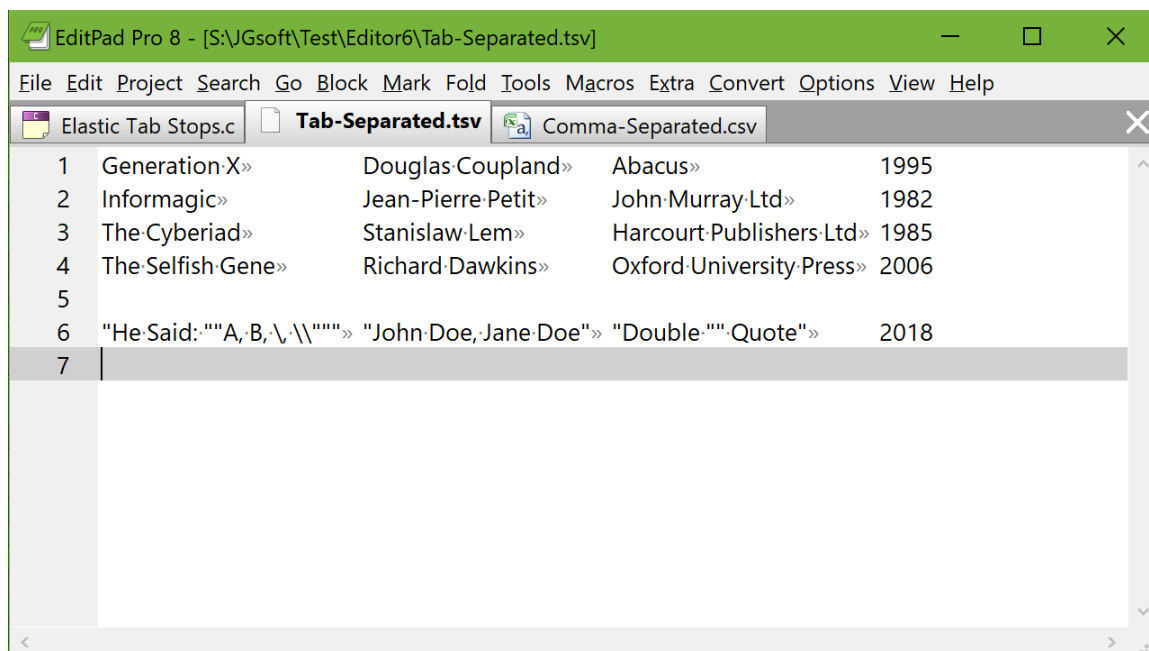
While elastic tab stops are very flexible, they do have a few limitations. You only get elastic tab stops if you actually use tabs. To be able to line up “int length” with “int start” we need to have a tab before “int start”. If you remove the tabs on lines 1 and 18, then the tabs on lines 2 and 19 no longer have tab stops on adjacent lines. They will revert to the indentation tab size.

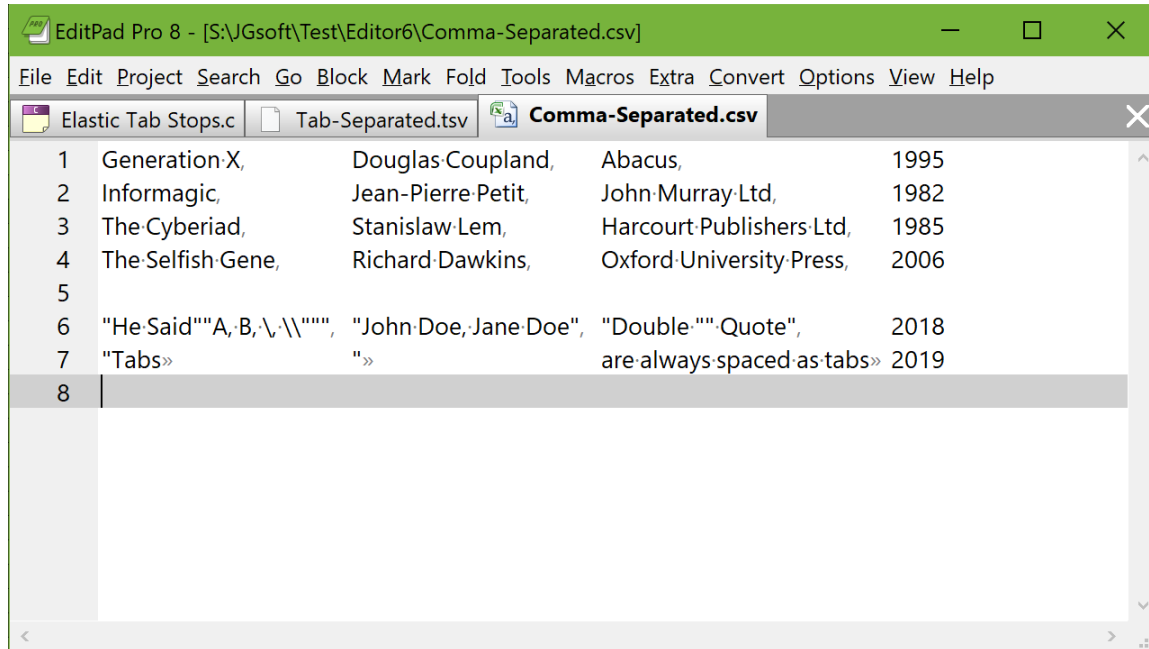
Blank lines form a barrier between blocks of lines that use the same number of tab stops. If you need a blank line between two lines of which you want the tab stops to line up then you need to put the same number of tabs on that blank line. The tabs on lines 6 and 23 in the screen shot ensure that the comments on the preceding and following line are aligned. To preserve tabs on otherwise blank lines, you need to turn off the two automatic whitespace trimming options on the Editor page.

You need to use exactly one tab to line up each column. Without elastic tab stops, you would need multiple tabs after “x()” to line up its comment with the comments on the next few lines, as “x()” is several tab widths shorter than “doSomethingComplicated()”. If you enable elastic tab stops when those multiple tabs are present, the comments won’t line up as they won’t use the same tab stops on adjacent lines. If you want to switch from fixed to elastic tab stops and your file may be using tabs in the middle of lines then it’s probably best to use Convert|Tabs|Tabs -> Spaces first to fix the existing layout of the file. As you edit the file you can then use tabs in the middle of lines to line up columns.

Tab-Separated Values

The tab-separated values option also makes tab stops elastic. The key difference is that it lines up all the tab stops on all the lines throughout the file. Blank lines or lines with fewer tabs do not break the alignment. The result is that EditPad lines up your tab-separated values in perfectly aligned columns, no matter how large the file is.





You can make this option line up columns using a different delimiter too. Set “value separator” to the character that should be spaced as if it were a tab. Only a single character can be used. The Comma-Separated.csv file in the screen shot uses a comma as the value separator. The value separator is shown with the same color as visualized tabs and spaces. It is always visible, regardless of whether tabs and spaces are visualized.

If you need to be able to use the value separator within values too, you can specify a quote character and an escape character. For CSV files the quote is usually a straight double quote. The escape is usually a backslash. In the screen shot, the quote is a straight double quote. No escape character is used. As you can see, commas between double quotes are spaced as normal commas. This is how Microsoft Excel saves CSV files.

If you specify a comment character then EditPad ignores the value separator character on lines that have the comment character as their very first character. If such a line doesn’t have any tabs then the length of that line does not affect the width of any of the columns.

Tabs are always spaced as tabs even if you set “value separator” to something else. Tabs cannot be quoted, escaped, or commented out.

To line up the tab stops in all lines of the file, EditPad has to continuously keep track of the width of the text between all the tabs on all the lines in your file. EditPad uses a background thread for this. When scrolling through a very large file with uneven columns widths, you may notice columns shifting while the background thread processes the file and finds wider columns. Once the whole file has been scanned the columns widths will be stable.

For performance reasons, only the first 64 tab stops on each line will be elastic. Additional tab stops will use the default tab size you specify for the file type. You may want to use a wider tab size than the typical 8 spaces to get more columns to line up when you have more than 64 per line. No tab stops will be elastic if your file is larger than the huge file threshold which you can configure in the Save Files Preferences.

Indentation

When you turn on “keep the same level of indentation when pressing Enter”, EditPad automatically puts the same number of spaces and/or tabs at the start of the new line as the previous line starts with. This option can be toggled for individual files with Options|Keep Indent.

The option “automatically indent or outdent new lines based on the syntax coloring scheme” is only enabled if you turn on the preceding option and if the syntax coloring scheme that you select on the Colors & Syntax page provides automatic indentation rules. Check it to use those automatic indentation rules. This option can be toggled for individual files with Options|Auto Indent. The help topic for Options|Auto Indent explains how automatic indentation works in detail.

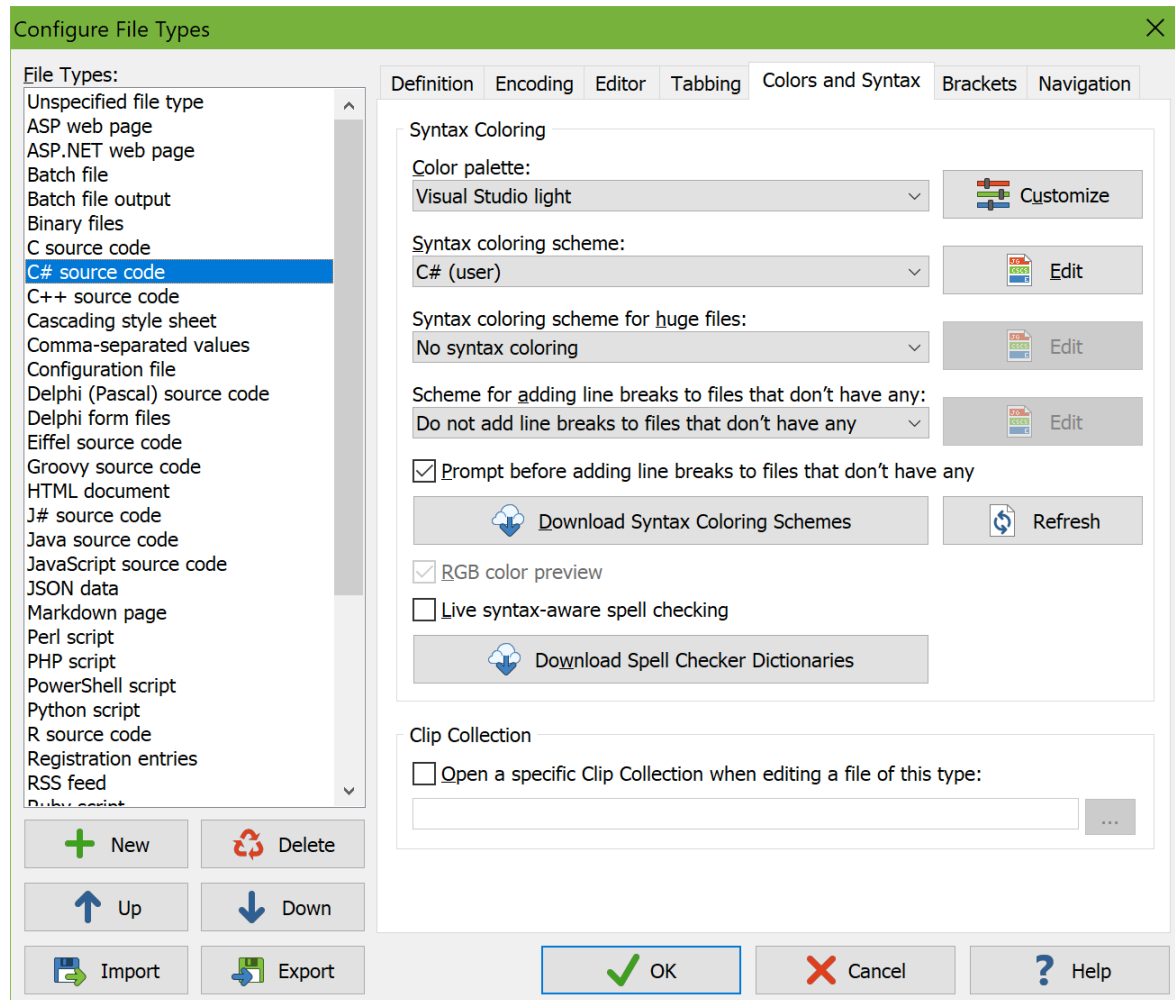
“Indentation size in spaces” is the number of spaces that Block|Indent and Block|Outdent use to indent or outdent the current selection. If the indentation size is the same as the tab size or a multiple of the tab size, and the option to make Tab insert spaces is off, the Indent and Outdent functions use tab characters. The indentation size is also used when automatic indentation needs to increase or decrease the indentation of the new line by one level.

If you edit files from different sources that use different indentation sizes, turn on “detect indentation size of the text surrounding the block to be indented”. If you do so then EditPad counts the amount of indentation of the lines above and below the selection each time you use Block|Indent or Block|Outdent. This also happens when automatic indentation needs to increase or decrease the indentation of the new line by one level.

To detect the indentation size, EditPad counts how many lines are indented by tabs and how many lines are indented by 2 to 8 spaces or a multiple thereof. If more lines are indented by tabs than by any of the multiples of 2 to 8 spaces then the indentation size is a single tab character. Otherwise the indentation size is the most common multiple of spaces that lines are indented with. If your file is consistently uses a tab or between 2 to 8 spaces as the indentation level then EditPad detects this perfectly. If your file uses inconsistent indentation then it uses the most common indentation size. If no lines are indented yet then EditPad uses the indentation size you specified above.

File Type Colors and Syntax

On the Colors and Syntax page in the file type configuration, you can configure whether EditPad applies any syntax coloring to your files.



Syntax coloring highlights different parts of a file in different colors. This makes it easier to edit text files that need to adhere to a certain syntax or formatting, such as programming source code or markup files. The different colors help guide your eyes through the structure of the file.

In EditPad, the actual highlighting is determined by a combination of two settings. The “color palette” associates named colors with actual red-green-blue colors. Select the palette that best matches your color preferences. Click the Customize button next to the drop-down list with color palettes to edit the individual colors. The list of palettes is shared by all file types. If you edit a palette used by multiple file types then the changes apply to all those file types.

The “syntax coloring scheme” uses the named colors to highlight different parts of the file. EditPad Pro ships with many syntax coloring schemes for a variety of file formats and programming languages. Simply select the one you want to use from the “syntax coloring scheme” drop-down list. Select “none” at the top of the list if you want to disable syntax coloring.

If no syntax coloring scheme is available for the file type you are defining, click the download button. EditPad Pro then connects to the Internet and allows you to download many custom syntax color schemes created and shared by other EditPad Pro users. To create your own syntax coloring schemes, use the Custom Syntax Coloring Scheme Editor. After editing a scheme or creating new ones, click the Refresh button to make EditPad Pro read in the new and edited schemes.

Because color palettes are separate from syntax coloring schemes, anyone can easily customize EditPad's colors to their taste by customizing one color palette. They can use the same palette with all their file types and have consistent and pleasing colors for all files. There is no need to edit any syntax coloring schemes, which is much harder than customizing a palette, just to change some colors. If you create a syntax coloring scheme for a file format not supported by EditPad, make sure that your scheme uses the named colors in a logical way so that it looks good with the palettes included with EditPad Pro. Then it will also look good with the custom palettes of anyone who might use your scheme.

Huge Files

Applying syntax coloring to an entire file takes up too much time and memory for very large files. For files larger than the huge files threshold set in the Open Files Preferences, EditPad Pro automatically disables syntax coloring schemes that require the entire file to be processed. If you selected such a scheme in the "syntax coloring scheme" drop-down list then you can select an alternative scheme in the "syntax coloring scheme for huge files" drop-down list. This second list only shows schemes that do not highlight anything that might span more than one line. This means the scheme only needs to process the visible part of the file. It can instantly apply syntax coloring to files of any size.

Some of the syntax coloring schemes supplied with EditPad Pro come in two versions, one of which is marked (fast). The regular scheme supports the full syntax of the programming language or file format it is intended for, but cannot be used with huge files. The (fast) scheme does not highlight things that span multiple lines, and works with files of any size. For example, the "XML" scheme handles the full XML syntax. The "XML (fast)" scheme highlights everything except comments and CDATA sections that span multiple lines. If a scheme supplied with EditPad Pro is available for huge files and is not marked (fast), that means that the scheme handles the full syntax of the programming language or file format that it is intended for. For example, batch files themselves are line-based, so the scheme for batch files never needs to highlight something that spans multiple lines.

Files without Line Breaks

Some files may not have any line breaks at all when they are computer-generated. XML and JSON files are prime examples. Files without line breaks are very cumbersome to work with in a text editor. EditPad Pro can automatically add line breaks and indentation to such files when loading them using a syntax coloring scheme that is designed for that purpose. If you want this, select a scheme from the "syntax coloring scheme for adding line breaks to files that don't have any" drop-down list.

The line breaks and indentation are added to the file while it is loaded. The file on disk is not modified while loading. But if you save a file to which EditPad Pro automatically added line breaks and indentation then those line breaks and indentation are saved into the file.

For some file formats, such as XML, EditPad Pro has an extra scheme marked (breaking). The (breaking) scheme supports the minimal amount of syntax to correctly add line breaks and indentation. This ensures no time is wasted while loading the file. The scheme needs to process the whole file before you can access the

end of the file. Because the (breaking) scheme does not handle the full syntax, you should only use it for adding line breaks. Though EditPad does allow you to select it for syntax coloring, you'll likely find it does not apply enough detail. The XML (breaking) scheme, for example, highlights XML tags including all their attributes in a single color. The other XML schemes apply different colors to attributes and attribute values.

RGB Color Preview

Some syntax coloring schemes can show text with actual colors specified in the file. If the scheme you selected does then you can check "RGB color preview" to enable this. The HTML and CSS schemes included with EditPad, for example, use this to show CSS colors as their actual colors. CSS colors are then highlighted using the actual RGB colors specified by the CSS color with the text color automatically chosen by EditPad to sufficiently contrast with the highlight color to remain readable. If you turn off "RGB color preview" then the HTML and CSS schemes highlight CSS colors with the "markup attribute value" color just like any other attribute.

Live Spell Checking

EditPad Pro can apply live spell checking. The "live syntax-aware spell checking" determines the default state of the Extra | Live Spelling menu item. Read the help topic on that menu item to learn how live spelling works together with syntax coloring in EditPad Pro.

The spell checker only works if you have previously downloaded and installed one or more Just Great Software spell checker dictionaries. Click the Download Spell Checker Dictionaries button to download some or all of the spell checker dictionaries free of charge.

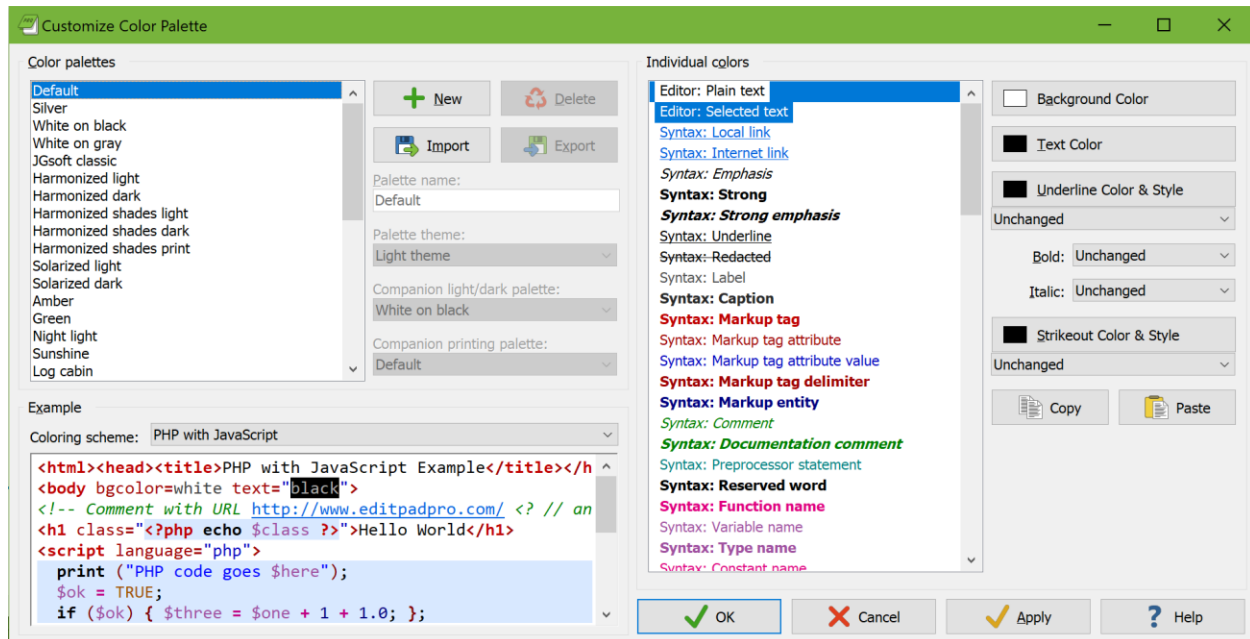
Clip Collection

You can specify a default Clip Collection to be used for each file type. If you activate a file that has a Clip Collection associated with its file type, that collection is automatically opened. By using different collections for each file type, you always have the appropriate text snippets at your fingertips. E.g. you could associate the HTML file type with a collection containing HTML tags, and the Java source code file type with your favorite Java code snippets.

EditPad Pro automatically saves modified collections. If you already have a collection open when you activate a file that has a Clip Collection associated with its file type, the collection that was already open is automatically saved before opening the file type's collection.

Color Palettes

Click the Customize button next to the drop-down list with color palettes on Colors and Syntax page in the file type configuration to choose the colors EditPad's editor control uses.



Color Palettes

EditPad comes with a long list of predefined color palettes.

- **Default:** The default EditPad color palette. Its plain text color uses the default window text and background colors in the active Windows theme. In Windows 7 and prior you could change these colors in the Appearance section in the Windows Control Panel. Usually they're black on pure white. This palette has distinct colors for almost all of the items. Choose this configuration if you like very detailed syntax coloring.
- **Silver:** Same colors as the default palette, but with black text on a silver background. This is easier on the eyes if your monitor has a very high brightness.
- **White on black:** Pure white text on a pure black background, with colorful syntax highlighting. This is a high contrast palette that works well as a dark palette on monitors with washed-out blacks (backlight bleeding on older LCD monitors).
- **White on gray:** White text on a dark gray background, with colorful syntax highlighting. The reduced contrast is easier on the eyes on monitors with very deep blacks.
- **JGsoft classic:** The color scheme used by EditPad Pro 4 and 5. It uses the default window text and background from the Windows Control Panel. Uses a limited set of different colors for syntax coloring.
- **Harmonized light:** Uses shades of dark gray for plain text and shades of light gray for background and highlighting. Uses 12 colors that are equidistant on the color wheel and of equal brightness for syntax coloring. This gives your files a uniform low-contrast look. Bracket and search match highlighting is done with underlining using some of those 12 colors.

- Harmonized dark: Uses the opposite shades of gray as “harmonized light” and exactly the same 12 colors for syntax coloring.
- Harmonized shades light: Same as “harmonized light”, but adds a lighter, less saturated shade of the 12 colors for things like bracket and search match highlighting.
- Harmonized shades dark: Same as “harmonized dark”, but adds a darker, less saturated shade of the 12 colors for things like bracket and search match highlighting.
- Harmonized shades print: Same as “harmonized shades light”, but using pure black on pure white for plain text.
- Solarized light: Using the 16 colors of the Solarized palette created by Ethan Schoonover.
- Solarized dark: Same as “solarized light” but using the opposite content and background tones.
- Amber: Amber text on a dark gray background like old terminal monitors. Uses a mixture of green and other colors for syntax coloring.
- Green: Green text on a dark gray background like old terminal monitors. Uses a mixture of amber and other colors for syntax coloring.
- Night light: Very dark palette with amber text. Uses some yellow and green for syntax coloring. Does not use any blue colors.
- Sunshine: Colorful palette with a pastel yellow background.
- Log cabin: Colorful palette with a dark brown background.
- Blue sky: Colorful palette with a sky blue background.
- Twilight: Colorful palette with a navy blue background.
- Monochrome light: Dark gray text on a light gray background. Uses different shades of gray for highlighting. Uses bold, italic, underlining, and a few different shades of gray for syntax coloring.
- Monochrome amber: Amber text on a dark gray background like old terminal monitors. Uses different shades of gray for highlighting. Uses bold, italic, underlining, and a few different shades of amber for syntax coloring.
- Monochrome green: Green text on a dark gray background like old terminal monitors. Uses different shades of gray for highlighting. Uses bold, italic, underlining, and a few different shades of green for syntax coloring.
- Monochrome dark: As “monochrome light” with the inverse shades of gray.
- Monochrome print: As “monochrome light” but with pure black text on a pure white background.
- Red-green color blind light: Intended for people who cannot perceive the difference between red and green. All colors have equal amounts of red and green. Dark gray text on a light gray background. Uses shades of yellow and blue for highlighting and syntax coloring.
- Red-green color blind dark: As “red-green color blind light” but with light gray text on a dark gray background.
- Red-green color blind print: As “red-green color blind light” but with pure black text on a pure white background.
- Yellow-blue color blind light: Intended for people who cannot perceive the difference between yellow and blue. All colors have equal amounts of green and blue. Dark gray text on a light gray background. Uses shades of red and cyan for highlighting and syntax coloring.
- Yellow-blue color blind dark: As “yellow-blue color blind light” but with light gray text on a dark gray background.
- Yellow-blue color blind print: As “yellow-blue color blind light” but with pure black text on a pure white background.
- Visual Studio light: Emulates the colors used by Microsoft’s Visual Studio light theme.
- Visual Studio dark: Emulates the colors used by Microsoft’s Visual Studio dark theme.
- Embarcadero: Emulates the colors used by recent versions of Delphi and C++Builder.
- Embarcadero dark: Emulates the colors used by the dark theme in recent versions of Delphi and C++Builder.

- Borland classic: Emulates the colors used by old Borland development tools for DOS, with yellow text on a blue background.
- PowerShell ISE light: Emulates the light colors used by the Windows PowerShell ISE.
- PowerShell ISE dark: Emulates the dark colors used by the Windows PowerShell ISE.

The built-in palettes cannot be customized. If you try to edit any of the individual colors, EditPad automatically makes a copy of the palette that you can then edit. You can also duplicate the selected palette by clicking the New button. Click the Delete button to remove a custom palette.

You can share palettes with others by clicking the Export button to save a palette as an .ini file. The Import button allows you to import an .ini file saved by the Export button. It will not import any other .ini files.

Each palette should have a name so you can select it in the list of palettes in the file type configuration.

If you create light and dark versions of the same palette, then you should set the “palette theme” option accordingly. After you have set “palette theme” you can select a custom palette of the opposite theme in the “companion light/dark palette”. If you do this then the View|Dark Theme menu item swaps your file types between these two palettes when you toggle the theme.

If you use File|Print or Block|Print then EditPad prints your file with the colors specified in your palette. If your palette’s background color is not white then EditPad will fill the entire page with ink. In that case you may want to create an extra version of your palette that has the plain text colors set to pure black text on a pure white background. You can then select that palette as the “companion printing palette” to your other palettes. You do not need to designate a palette as a printing palette. The “companion printing palette” drop-down list automatically lists all custom palettes that have the plain text colors set to pure black text on a pure white background. If you edit a file using a color palette for which you selected a companion printing palette then the print preview defaults to that printing palette instead of the palette you are editing with.

Changing Individual Colors

If you customize one of the harmonized, solarized, or monochrome palettes then the color picker restricts you to the specific set of harmonized, solarized, or monochrome colors that are available to the palette you’re customizing. To change a color, simply click the Color button and then click on the color you want.

For all other palettes, the color picker shows a grid of 5 rows of 8 common colors. Any other colors used by your palette are shown below the 40 common colors. The other colors are shown in the order of the first individual color that uses that color. The default palette, for example, uses different shades of red for markup tags and markup attributes. The red of markup tags is listed first because “markup tags” is above “markup attributes” in the list of individual colors. Each color is shown only once. All the color buttons show the same color picker. So the Background Color button also shows colors that are only used for text, for example.

You can pick a more specific color by clicking the More button to show the color hexagon. The colors on the outer edge of the hexagon are always fully saturated. The colors nearer to the center of the hexagon are progressively desaturated towards white, gray, or black. Click on the vertical grayscale slider to change the luminance of the colors in the center of the hexagon. If you want to pick a pure grayscale value, click on the horizontal line of small gray hexagons. If you want pure white or black, click on the larger black or white hexagon to the left or right of the gray hexagons. If you want to use a specific RGB value, enter three numbers between 0 and 255 in the edit boxes that are shaded red, green, and blue.

The color hexagon, along with the vertical grayscale slider, makes it easy to pick related colors. The concepts that follow may be easier to understand if you see the large color hexagon as a circle with discrete steps, and the small hexagons that comprise it as positions at a certain radius and angle. Colors with the same distance to the center (same radius) have the same amount of saturation. The colors at the edge are always fully saturated and the color at the center is always fully desaturated. You can change the hue without changing the saturation by selecting another color at the same distance (same radius, different angle). You can change the saturation without changing the hue by selecting a color closer to or further away from the center (different radius, same angle). For colors that are not fully saturated, the grayscale slider controls the luminance. To make a color lighter or darker, first click on the grayscale slider to change the luminance. Then click on the same spot in the color hexagon to select the same color with the different luminance.

Syntax elements that use colors of different hues but similar saturation and luminance allow those elements to be distinguished without drawing attention to some over the others. Syntax elements for important structural parts of the file can be given colors with more luminance while syntax elements for less important parts or more common parts (large blocks of text) can be given colors with less luminance to make it easier for the eye to gravitate towards the important elements. This can be further enhanced by using bold text for the most important syntax elements. If the base (plain text) color of the scheme is white, gray, or black, then setting the luminance slider at that level of white, gray, or black allows you to use more saturated colors for more important elements and less saturated colors for less important elements, as an alternative technique to using luminance. Color schemes that use very saturated colors appear bolder, while color schemes with (somewhat) desaturated colors (using luminance instead of saturation for highlighting) appear more relaxing.

If you prefer to use the standard Windows color selection dialog box to pick your colors, right-click the Background Color or Text Color button. This dialog allows you to specify hue and saturation using a two-dimensional grid and luminance using a vertical slider. You can also enter HSL or RGB values with the keyboard. If you do this while customizing a harmonized, solarized, or monochrome palette then you can break the restricted nature of the palette.

If you want to use exactly the same color for two items, first select the item that already has the color you want. Click the Color button that has the color you want. One of the color squares in the color picker should have a beveled edge. Note its location in the grid of squares.

After observing the color, click on the item in the list that should have the same color. The color picker automatically closes when you do this. Click the Color button that should have the same color. Click on the same square in the grid that you observed.

You can use the Copy and Paste buttons to make two individual colors identical. Clicking Paste changes all the colors and styles of the selected individual color to the ones that you copied.

Default Colors and Unchanged Styles

Colors can be set to “default” and text styles can be set to “unchanged”. For the “editor: plain text” color the default colors are the window text and background colors configured in the Windows Control Panel (as part of the Windows theme or via the advanced appearance settings). The default plain text style is the one you selected in Options | Text Layout or Options | Font.

For all other individual colors, the default color and style depends on how the colors are layered. Multiple colors can apply to the same text. These colors may be layered onto the same piece of text, in decreasing order of priority:

1. Selection
2. Matching bracket
3. Search match
4. Search range
5. Misspelled word
6. Syntax color
7. Comparison marks
8. Folded line
9. Active line highlight
10. Syntax coloring subscheme highlight

All colors (background, text, underline, strikeout) and all styles (bold, italic, underline, strikeout) are determined separately. EditPad takes each color and each style from the individual color with the highest priority that applies to that text that has that color or style set to something other than “default” or “unchanged”. If all individual colors that apply have the text or background color set to “default” or a style set to “unchanged” then the color or style of the “editor: plain text” color are used for that text. If all individual colors that apply have the underline or strikeout color set to “default” then the text color is used for underline or strikeout.

A few examples make this easier to understand. The “white on black” palette has the “editor: selected text” color set to black text on a purple background. So with this palette, any text you select is black on purple because the selection highlight always has the highest priority. But the “harmonized dark” palette sets the selection background to a shade of gray but the selection foreground to “default”. With this palette, the selection highlight only appears as a different background color. Selected text keeps the colors it gets from syntax coloring. Of course, this only works well if the background color for selected text contrasts well with all the text colors that may be used by syntax coloring.

If the selection contains a highlighted search match then that search match gets a double underline with both these palettes. That underline remains when you select the search match as both palettes have the underline style for “editor: selected text” set to “unchanged”. The underlining remains even though the yellow background that the “white on black” palette specifies for highlighted search matches disappears when the search match is selected.

List of Individual Colors

The colors prefixed with “syntax” are the named colors used by syntax coloring schemes. The descriptions given here are the suggested uses of these colors. The syntax coloring schemes provided with EditPad Pro strictly follow these suggested uses, though most schemes do not use all of the colors. Syntax coloring schemes created by other EditPad Pro users may use these colors to highlight other things.

The colors prefixed with “editor” are used to draw various parts of EditPad’s editor control. These colors are not used by syntax coloring schemes. The editor uses these colors even when syntax coloring is disabled.

The colors prefixed with “regex” are used by the Search Panel to apply syntax coloring to regular expressions.

- Editor, plain text: The default text colors. EditPad’s background is filled with this color, and text that is not syntax colored is drawn in this text color.
- Editor, selected text: Selection highlight.
- Syntax, Local link: Clickable link to a file on the user’s computer or local network. Only use this color for scheme elements with an action to open the file.

- Syntax, Internet link: Clickable link to a web page, file or email address. Only use this color for scheme elements with an action to open a file or URL, or start an email.
- Syntax, Emphasis: Text in a document that will appear in italics when the document is printed or published.
- Syntax, Strong: Text in a document that will appear bold when the document is printed or published.
- Syntax, Strong emphasis: Text in a document that will appear bold and in italics when the document is printed or published.
- Syntax, Underline: Text in a document that will appear underlined with the document is printed or published.
- Syntax, Redacted: Text that will not appear at all when the document is printed or published.
- Syntax, Label: An indicator that does not need emphasis, such as a fixed label before the actual data.
- Syntax, Caption: An indicator that can use some emphasis, such as a subsection header or a title.
- Syntax, Markup tag: Opening or closing tag in markup languages.
- Syntax, Markup tag attribute: Attribute name in markup languages.
- Syntax, Markup tag attribute value: Attribute value in markup languages.
- Syntax, Markup tag delimiter: Punctuation that starts or ends a tag in markup languages.
- Syntax, Markup entity: An entity name or a numeric character reference in markup languages.
- Syntax, Comment: Human-readable text used for information only.
- Syntax, Documentation comment: Human-readable text of particular importance, used for information only.
- Syntax, Preprocessor statement: Any kind of meta-information, such as compiler and preprocessor directives.
- Syntax, Reserved word: Words or character combinations with a specific meaning and specific use, such as keywords in a programming language.
- Syntax, Function name: The name of a (built-in) function call in a programming language, or a reference in a document.
- Syntax, Variable name: The name of a (built-in) variable in a programming language, or a placeholder for a changeable value or macro in a document.
- Syntax, Type name: The name of a (built-in) type or class in a programming language.
- Syntax, Constant name: The name of a (built-in) constant in a programming language, or a fixed placeholder in a document.
- Syntax, Constant value: A literal value that does not fit one of the following literal types.
- Syntax, Character: A single human-readable character (letter). Can also be used for escaped characters within character strings.
- Syntax, Character string: Human-readable text to be processed by software.
- Syntax, Text pattern: A text pattern such as wild cards or a regular expression.
- Syntax, Integer number: A whole number. Could be decimal, hexadecimal, octal, or binary.
- Syntax, Floating point number: A number with a fractional part and/or an exponent.
- Syntax, Date and time: Any date or time.
- Syntax, Address: Text defining the location of a file or server, such as an IP address or URI. Use this for scheme elements that do not have an action to open or navigate to the address.
- Syntax, Operator: Any kind of mathematical operator or other symbol with a specific meaning or effect.
- Syntax, Bracket: Round, square, or curly brackets used in expressions, such as parentheses and square brackets in C-style languages.
- Syntax, Structural brackets: Round, square, or curly brackets used to group statements or items, such as curly braces in C-style languages.
- Syntax, Section header: Heading that starts a new section.
- Syntax, Success message: Message in tool output, a log file, or a report indicating successful completion.

- Syntax, Hint message: Informative hint message in tool output, a log file, or a report.
- Syntax, Warning message: Non-fatal warning message in tool output, a log file, or a report.
- Syntax, Error message: Fatal error message in tool output, a log file, or a report.
- Syntax, Markup highlight: Highlight color for subschemes that highlight markup in files that do not predominantly consist of markup.
- Syntax, Code highlight: Highlight color for subschemes that highlight procedural code in files that do not predominantly consist of procedural code.
- Syntax, Fountain highlight 1 to 10: Can be used in specialized schemes to make up to 10 different kinds of items stand out, such as to highlight different kinds of entries in log files.
- Editor, search range: When you start a search with the selection only option turned on, and a match is found, the match is selected and the selection searched through is highlighted in the search range color.
- Editor, highlight active line: When you turn on the option to highlight the active line, the line containing the text cursor is highlighted in this color.
- Editor, page break: Color of the horizontal line indicating a page break.
- Editor, line breaks: Color used to draw line break symbols when you've turned on the option to visualize line breaks.
- Editor, whitespace: Color used to draw whitespace. The background color is always used if you set it to anything other than "default". The text color is used to draw the space and tab symbols when you turn on the option to visualize spaces.
- Editor, control characters: Color used in text mode to draw control characters other than tabs and line breaks. Such control characters normally should not appear in text files.
- Editor, margin and line numbers: Color used for the left and top margins in which line and column numbers are displayed.
- Editor, extra space between lines: If the text layout adds extra space between lines then this color is used for that extra space. This can simulate the appearance of lined paper.
- Editor, matching brackets: Color used to highlight matching brackets that do not contain any unmatched brackets. This color is also used to highlight the current byte in hexadecimal mode.
- Editor, incorrectly nested brackets: Color used to highlight matching brackets that contain unmatched brackets.
- Editor, unmatched brackets: Color used to highlight brackets that do not have a matching opening or closing bracket.
- Editor, bookmark icons: Color used to draw the square and number of bookmarks in the left margin.
- Editor, folding icons: Color used to draw text folding buttons and lines in the left margin.
- Editor, folded lines: Highlight applied to the first line in a block of folded lines when the block is actually folded.
- Editor, column leader: Color used for vertical rulers marking columns.
- Editor, line edited since file last saved: If you turn on indicators for edited lines then this color is shown in the left margin for lines that you edit. Specify a background color if you want a big rectangle as your indicator. Specify a vertical line if you want a thin line as your indicator.
- Editor, line edited since file last opened: If you turn on indicators for edited lines then this color is shown in the left margin for lines that you have edited since last opening the file but not since last saving the file. Specify a background color if you want a big rectangle as your indicator. Specify a vertical line if you want a thin line as your indicator.
- Editor, indentation level 1 to 4: Used to visualize indentation.
- Editor, misspelled word: When using live spell checking, misspelled colors are indicated using these colors and font style.
- Editor, highlighted search match: Used by Search | Highlight All and Search | Instant Highlight.
- Editor, highlighted replacements: Used by Search | Instant Replace.

- Editor, compare files, added line: When using Extra|Compare Files, lines present in the “new” file but not in the “old” file are highlighted using this color.
- Editor, compare files, deleted line: When using Extra|Compare Files, lines present in the “old” file but not in the “new” file are highlighted using this color.
- Regex, Syntax error: Any invalid combination of characters in regular expressions.
- Regex, regular expression token: Characters with a special meaning in regular expressions not covered by any of the other colors.
- Regex, escaped literal: A special character that was escaped to be treated as a literal character.
- Regex, Comment: A comment in a regular expression.
- Regex, meta token: Characters that control the behavior of the regular expression but that do not match anything themselves.
- Regex, Character class brackets: The brackets around a character class in a regular expression.
- Regex, Character class literal text: Characters that are matched literally inside a character class in a regular expression.
- Regex, Character class token: Character with a special meaning inside a character class in a regular expression.
- Regex, Character class range: Characters that form a range inside a character class in a regular expression.
- Regex, Group (nesting level 1 to 5): Any pair of parentheses in a regular expression. Nested parentheses are highlighted in different colors to make it easier to see which opening parenthesis is paired with which closing parenthesis.

Example

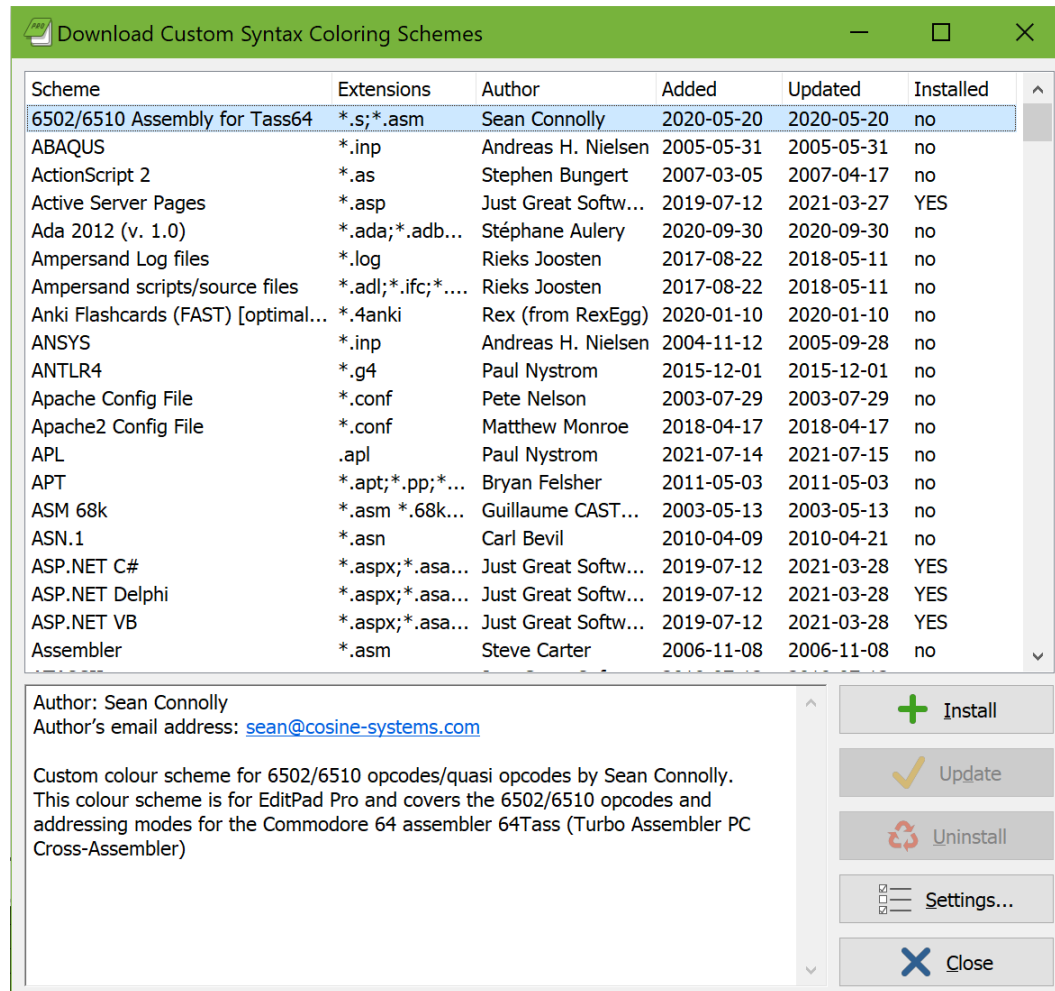
When configuring syntax highlighting colors, you can select one of the available colorings schemes to see an example. Each coloring scheme has its own example text that shows the most important color elements of the scheme. You can double-click any text in the example to select the individual color that was applied by the syntax coloring scheme.

The example does not necessarily show all color elements. You can type in or paste in your own example to test the colors. Your example won’t be saved. If you select a different coloring scheme in the drop-down list then the example is reset to what is stored in the scheme.

Download Custom Syntax Coloring Schemes

When you click on the button to download custom syntax coloring schemes on the Colors and Syntax page in the file type configuration, EditPad Pro connects to the Internet. It downloads a list of custom syntax coloring schemes that have been created and generously shared by other EditPad Pro users. This list is then displayed in the window shown below.

Note: Any schemes you download are *not* automatically put to use. You will need to create or edit a file type to use a scheme that you downloaded.



The list presents you the following information about the available schemes:

- **Scheme:** The name of the scheme. It is possible that more than one scheme with the same name is available. If so, you can install all of them or only one, as you prefer.
- **Extensions:** The extensions typically used for the files that the scheme provides syntax coloring for. They are only shown for your information. You can apply any scheme to any file type, as you see fit.
- **Author:** The author of the scheme. If you have comments or suggestions about a particular coloring scheme, you should contact this person.
- **Added:** The date the scheme was first added to the list of available schemes.
- **Updated:** The date the scheme was last updated.
- **Installed:** Indicates whether this scheme is installed on your computer or not.

If you click the **Install** button, EditPad Pro immediately downloads the scheme and saves it into the %APPDATA%\JGsoft\EditPad Pro 8 folder.

Click the **Update** button to download a scheme that you already installed again.

If you click the **Uninstall** button, EditPad Pro deletes the scheme from your computer when you close the window for downloading custom syntax coloring schemes.

If you are behind a proxy server, and EditPad Pro is unable to detect your proxy settings, you can change them by clicking the **Settings** button.

Using Custom Syntax Coloring Schemes

After downloading a custom syntax coloring scheme, it is *not* automatically put to use.

All the schemes that you download are listed in the “syntax coloring” drop-down list on the Colors and Syntax page in the file type configuration. They appear in alphabetic order among the schemes that ship with EditPad Pro. This drop-down list is shown in the screen shot below.

To use one of the schemes, create or edit a file type, and select the coloring scheme you want from the Syntax Coloring drop-down list.

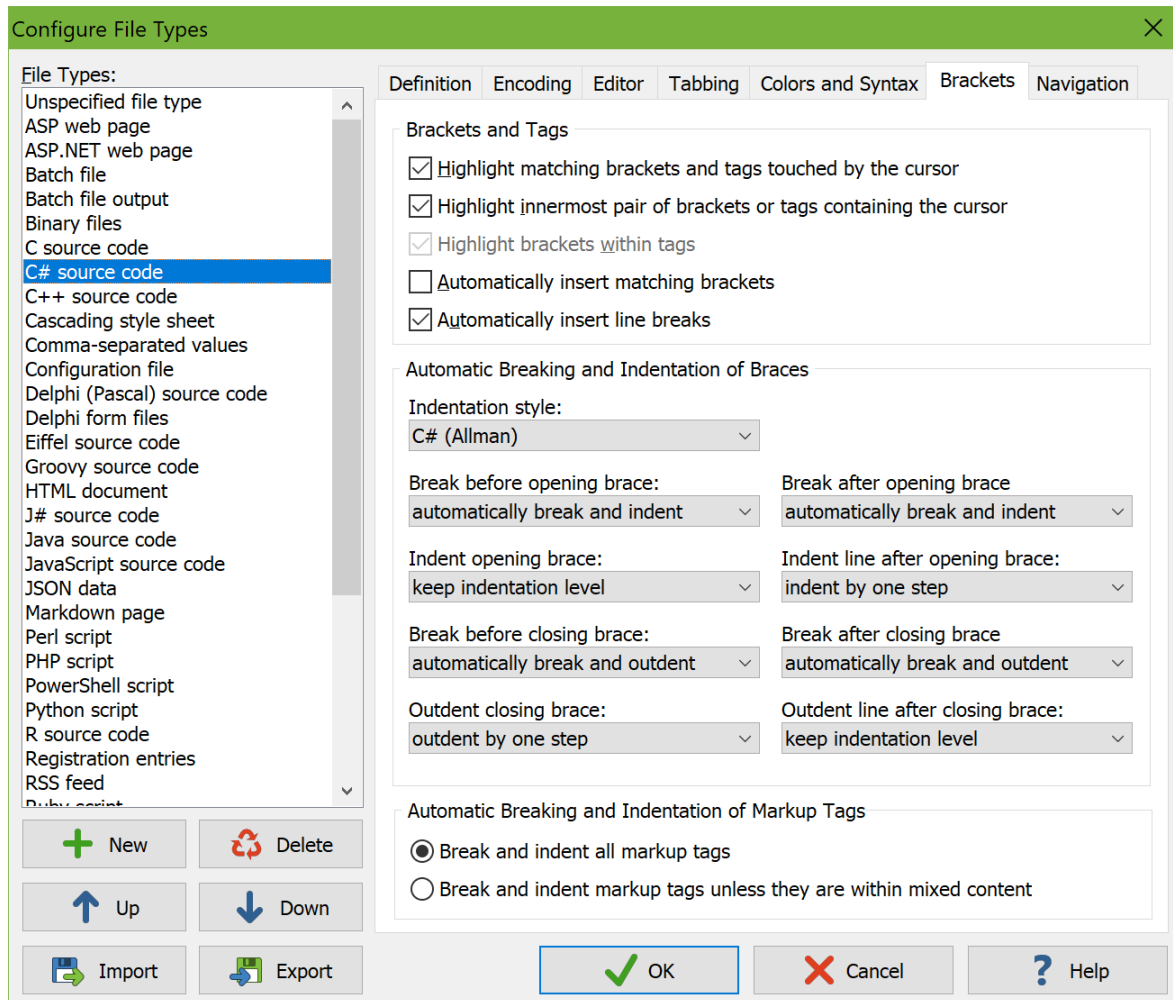
Syntax Coloring Scheme Editor

With the Syntax Coloring Scheme Editor, you can create your own syntax coloring schemes for use with EditPad Pro. You can also edit the schemes included with EditPad Pro, and those you’ve downloaded. The syntax coloring schemes files have a .jgscs extension. You can’t edit them with any program except the Syntax Coloring Scheme Editor. You can download the scheme editor at <https://www.editpadpro.com/cscs.html>. The editor is a free download for all licensed EditPad Pro users. Full documentation is included.

The syntax coloring schemes included with EditPad Pro are stored in the folder where you have EditPad Pro installed, typically C:\Program Files\Just Great Software\EditPad Pro 8. Windows’s default security settings do not allow normal users to modify files under C:\Program Files. Therefore, schemes that you download and schemes that you edit are saved under your Windows user profile, typically C:\Users\yourname\AppData\Roaming\JGsoft\EditPad Pro 8. If a scheme with the same file name exists in both the user profile folder and the program files folder, only the scheme in the user profile folder will be available in EditPad. On the Colors and Syntax page in the file type configuration, schemes loaded from your Windows user profile are marked with (user).

File Type Brackets

On the Brackets page in the file type configuration, you can configure bracket matching and automatic breaking and indentation.



Turn on “highlight matching brackets touched by the cursor” to highlight the bracket touched by the cursor and that bracket’s corresponding opening or closing bracket. The cursor touches a bracket when it is positioned immediately to the left or right of the bracket. If the bracket consists of multiple characters, the cursor also touches it if it is positioned between two characters that are part of the bracket. If the cursor touches two brackets at the same time that belong to different pairs, the bracket to the right of the cursor is highlighted. If the corresponding bracket is missing, the bracket touched by the cursor is highlighted by itself, in a different color.

Turn on “highlight innermost pair of brackets containing the cursor” to highlight the pair of opening and closing brackets nearest to the text cursor that surround the text cursor. Brackets surround the cursor if the cursor is positioned to the right of the last character in the opening bracket and to the left of the last character in the closing bracket. If the innermost pair of matching brackets contains an unmatched opening bracket to the left of the cursor or an unmatched closing bracket to the right of the cursor, then that unpaired bracket is highlighted alone in a different color and the innermost pair is not highlighted.

If you turn on both bracket highlighting options, then touching brackets are highlighted if the cursor touches a bracket. The innermost pair containing the cursor is highlighted if the cursor does not touch any brackets.

Exactly which brackets are highlighted and how they are paired up depends on the syntax coloring scheme you selected on the Colors and Syntax page. If the syntax coloring scheme does not define any bracket pairs, EditPad matches up nested pairs of (), [], and {} throughout the file. The provided syntax coloring schemes for programming languages match up these brackets too, depending on how they are used in the programming language. They also match up quotes that are used to delimit strings and characters that delimit multi-line comments.

The syntax coloring schemes for markup formats such as HTML and XML match up entire HTML and XML tags. They can also match up the < and > that delimit the tags. Turn on “highlight brackets within tags” if you want < and > to be paired as brackets when the cursor is inside an HTML or XML tag. Turn it off if you always want the HTML or XML tag to be highlighted along with its corresponding opening or closing tag.

Syntax coloring schemes can specify which closing bracket should be generated when the file contains an unpaired opening bracket. The option “automatically insert matching brackets” is only enabled when the syntax coloring scheme that you selected does so. The option determines the default state of the Edit|Auto Match Brackets menu item. When on, EditPad automatically generates the closing bracket when you enter an opening bracket. See the Edit|Auto Match Brackets topic for more details.

Syntax coloring schemes can also specify that line breaks should be automatically inserted before or after certain text. If the selected scheme does so, you can enable this by ticking “automatically insert line breaks”. You can toggle it for the active file with Options|Auto Break.

Automatic Breaking And Indentation of Braces

A syntax coloring scheme can designate certain text as opening braces or as closing braces for the purposes of automatic breaking and indentation. The schemes for C-style languages included with EditPad do this for braces that surround code blocks. But schemes for other languages such as Delphi that use keywords to delimit blocks also designate the “begin” and “end” keywords as braces. When using such a scheme, you can specify how EditPad should break and indent before those braces.

Schemes for languages such as Visual Basic that have more complicated sets of structural keywords have hard-coded breaking and indentation rules. When using such a syntax coloring scheme, your choices for breaking and indenting around braces are ignored.

EditPad Pro supports the most common indentation styles. Selecting one from the drop-down list updates the eight individual rules for breaking and indenting before and after opening and closing braces. Changing one of the individual rules automatically updates the style drop-down list as well.

The following examples show the results of each of the indentation styles. Invisible line breaks were automatically inserted. Pilcrow signs indicate a line break added by pressing Enter on the keyboard. No indentation spaces were entered by pressing the space bar on the keyboard. No automatically inserted indentation spaces or line breaks were removed by pressing backspace. It is important to resist the urge to backspace excess indentation before typing a closing brace. When you type the closing brace, EditPad automatically outdents the brace by one step.

Java (K&R)

```
function {
  if {
    true
  } else {
    false
  }
}
```

If you want the original K&R style with the { of a function block on a separate line, you need one extra press of the Enter key. EditPad's rules for breaking around braces treat all braces equally. There's no way to specify to break before certain braces but not before other braces. If that's what you need then you need to tell EditPad Pro not to break before the brace and press Enter manually before typing the brace when it does need to be on a separate line.

```
function{
{
  if {
    true
  } else {
    false
  }
}
```

C++ (Stroustrup)

```
function {
  if {
    true
  }
  else {
    false
  }
}
```

C# (Allman)

```
function
{
  if
  {
    true
  }
  else
  {
    false
  }
}
```

Pascal (Allman)

The Pascal variation of the Allman style does not automatically insert line breaks after the “end” keyword so that you can have the semicolon immediately after the “end”. Though it will automatically break after the “begin” keyword, you will often still need to enter the line break after “begin” yourself. EditPad can’t do it as soon as you type “begin” because you may intend to type “beginSomeOperation()” which is not a keyword.

```
function
begin
  if
    begin
      true
    end
  else
    begin
      false
    end;
end;
```

C (Whitesmiths)

```
function
{
  if
  {
    true
  }
  else
  {
    false
  }
}
```

Lisp

Since EditPad can’t know how many blocks you want to close on any one line, you need to press Enter after typing the last closing brace. EditPad will correctly outdent the next line by as many closing braces you typed before pressing Enter.

```
function
{if
  {true}
else
  {false}}
```

Horstmann

```
function
{ if
  { true
  }
  else
```

```

    { false
  }
}

```

Ratliff

```

function {
  if {
    true
  }
  else {
    false
  }
}

```

GNU

```

function
{
  if
  {
    true
  }
  else
  {
    false
  }
}

```

Automatic Breaking And Indentation of Markup Tags

A syntax coloring scheme can designate certain text as markup tags for the purposes of automatic breaking and indentation. The HTML and XML schemes included with EditPad do this. The ASP.NET and PHP schemes do this as well, on top of designating braces as such.

EditPad Pro supports two indentation styles for markup tags. It can either break around all tags, or break only around tags that aren't part of mixed content. A tag contains mixed content if its content starts with text rather than with another markup tag. The following is what you get when breaking around all tags. All line breaks and indentation were inserted automatically.

```

<html>
  <head>
    <title>
      All Tags
    </title>
  </head>
  <body>
    <p>
      Break around
      <i>
        all
      </i>
    </p>
  </body>
</html>

```



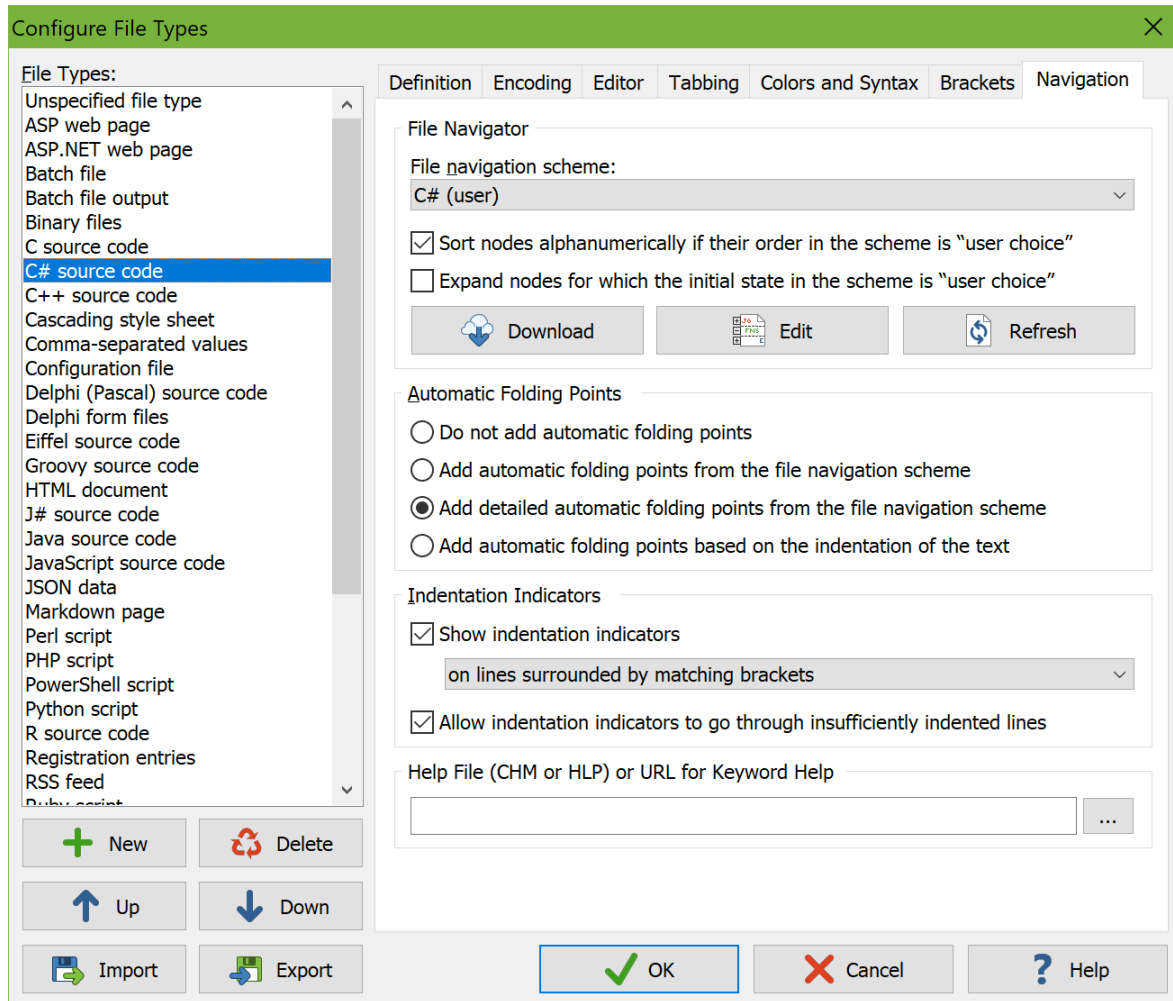
```
        markup tags.  
    </p>  
</body>  
</html>
```

This is what you get when breaking around tags except those inside mixed content. Again, all line breaks and indentation were inserted automatically.

```
<html>  
  <head>  
    <title>Don't break mixed content</title>  
  </head>  
  <body>  
    <p>Break around tags <i>except</i> within mixed content.</p>  
  </body>  
</html>
```

File Type Navigation

On the Navigation page in the file type configuration, you can configure the file navigation and line folding text editing aids.



File Navigator

You can access the File Navigator by selecting View|File Navigator in the menu. The File Navigator displays the structure of the file in a collapsible tree. By clicking on items in the tree, you can quickly navigate to various parts of the file.

The File Navigator requires a file navigation scheme to do its job. You can select a predefined scheme from the file “navigation scheme” drop-down list.

If no file navigation scheme is available for the file type you are defining, click the download button. EditPad Pro then connects to the Internet and allows you to download file navigation schemes created and shared by other EditPad Pro users. To create your own file navigation schemes, use the File Navigation Scheme Editor.

After editing a scheme or creating new ones, click the Refresh button to make EditPad Pro read in the new and edited schemes.

Automatic Folding Points

Automatic folding points appear as small squares in the left margin, with a vertical line extending down from the square to indicate the range. These allow you to quickly fold logical parts of the text to get a better overview of the overall structure. You can change their appearance on the Editor page in the Preferences. When EditPad Pro adds automatic folding points, any unused (i.e. expanded) folding points you created with Fold|Fold and Fold|Unfold are removed.

EditPad Pro can obtain automatic folding points from two sources. Many, but not all, file navigation schemes also define foldable ranges when they associate different parts of the file with various nodes in the file navigation tree. These ranges usually follow the syntax of the file. Some file navigation schemes mark some of their foldable ranges as being “detailed”. For example, schemes for C-style languages add regular folding ranges for classes and functions, and detailed folding ranges for all other pairs of curly braces. If you select to add automatic folding points from the file navigation scheme then only the regular folding ranges appear in EditPad Pro. If you select to add detailed automatic folding points from the file navigation scheme then regular and detailed folding ranges appear in EditPad Pro. Both types of folding ranges behave in exactly the same way. The only difference is that the detailed ranges only show up when you choose.

Alternatively, EditPad Pro can use a file’s indentation as the basis for folding points. Whenever a line is followed by one or more lines that are indented further than itself, that line becomes a folding point. Its range stops before the next line with the same or a smaller indentation than the foldable line. EditPad Pro nests folding points this way up to three levels deep. Further levels down are only added if the foldable range is longer than half the number of lines that EditPad Pro can display at a time.

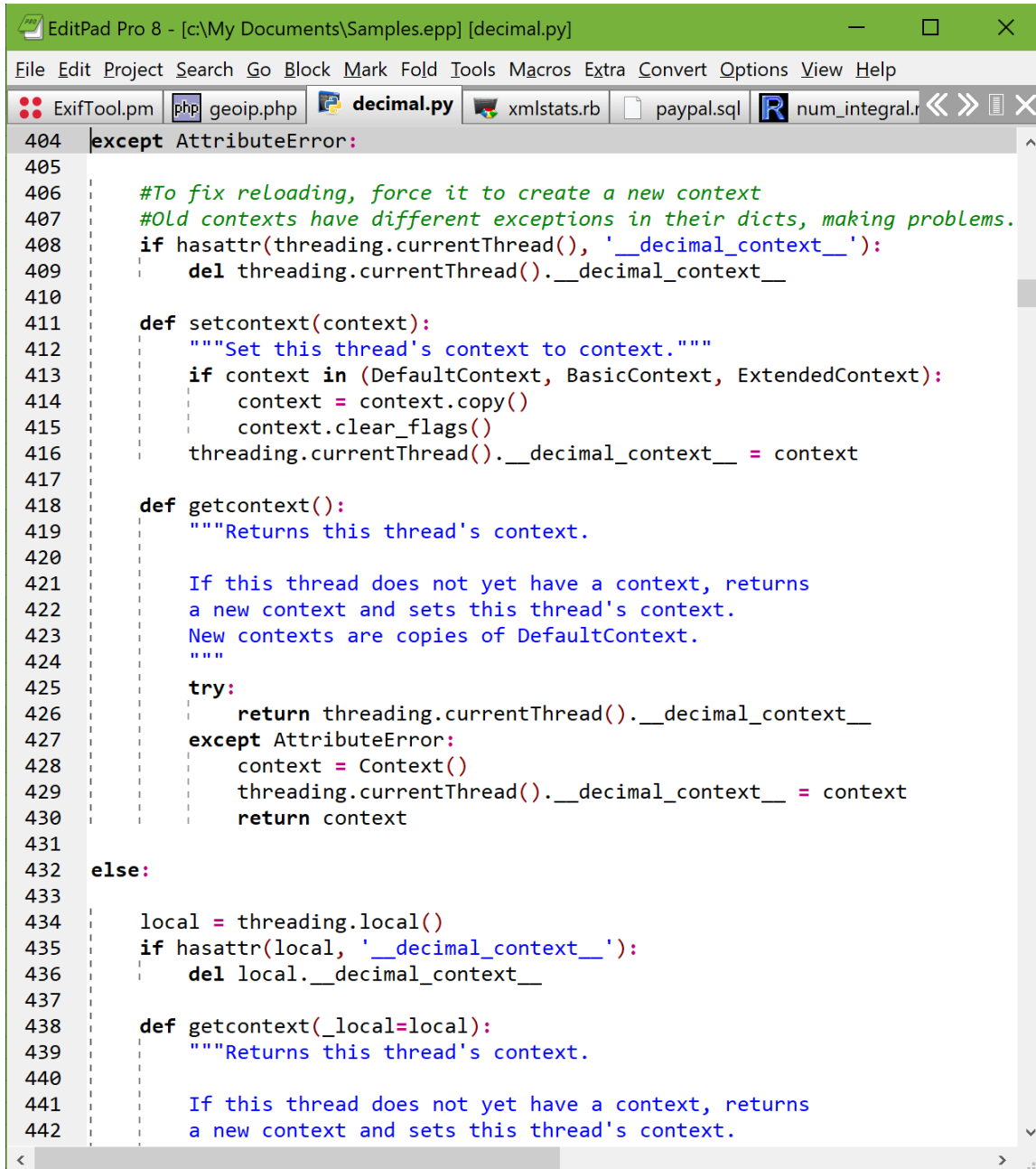
Note that when enabling automatic folding points, you can still use the Fold|Fold to fold different blocks of text. They just won’t persist when you unfold them and edit the file.

Indentation Indicators

The “show indentation indicators” checkbox determines the default state of the Options|Visualize Indentation menu item for each file type. When on, EditPad Pro indicates indentation levels with vertical lines or colored backgrounds. These indicators can help you to see whether the indentation in your file is correct and consistent.

The menu item only lets you toggle indentation indicators on or off. Exactly which lines get indentation indicators is determined by the settings in the Indentation Indicators group in the file type configuration. The drop-down list gives you four options to choose on which lines indentation indicators start. If you pick one of the latter two then you can also choose whether indentation indicators may go through insufficiently indented lines or not.

On All Indented Lines



```

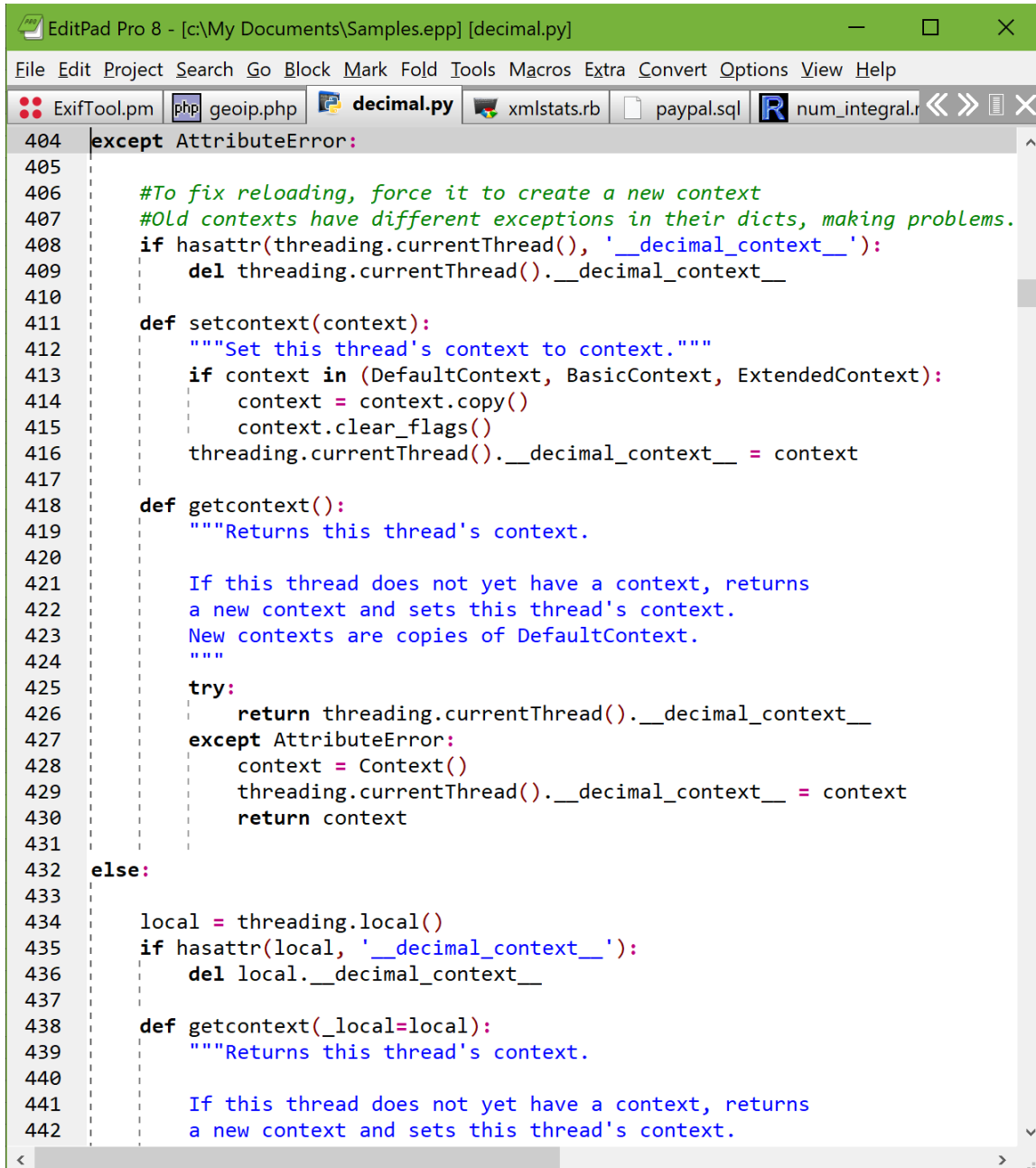
404 except AttributeError:
405
406     #To fix reloading, force it to create a new context
407     #Old contexts have different exceptions in their dicts, making problems.
408     if hasattr(threading.currentThread(), '__decimal_context__'):
409         del threading.currentThread().__decimal_context__
410
411     def setcontext(context):
412         """Set this thread's context to context."""
413         if context in (DefaultContext, BasicContext, ExtendedContext):
414             context = context.copy()
415             context.clear_flags()
416             threading.currentThread().__decimal_context__ = context
417
418     def getcontext():
419         """Returns this thread's context.
420
421         If this thread does not yet have a context, returns
422         a new context and sets this thread's context.
423         New contexts are copies of DefaultContext.
424         """
425         try:
426             return threading.currentThread().__decimal_context__
427         except AttributeError:
428             context = Context()
429             threading.currentThread().__decimal_context__ = context
430             return context
431
432 else:
433
434     local = threading.local()
435     if hasattr(local, '__decimal_context__'):
436         del local.__decimal_context__
437
438     def getcontext(_local=local):
439         """Returns this thread's context.
440
441         If this thread does not yet have a context, returns
442         a new context and sets this thread's context.
  
```

Start an indentation indicator on every line that is indented further than the preceding non-blank line at the indentation level of that preceding line. The indentation indicator stops at the last non-blank line that is indented further than the indicated indentation level.

The difference between this option and the next one is essentially that this option does not add indentation indicators on blank lines before and after an indented block. Indentation indicators do run through blank lines in the middle of the indented block they're indicating. You can see the difference between the screen shots above and below most obviously around the non-indented "else:".

Showing indentation indicators on all lines is the best option for file formats where the indentation itself determines the block structure of the file and there is no line to close the block that is indented at the same level as the line starting the block. EditPad Pro uses it by default for the Python file type.

On All Indented Lines And Enclosed Blank Lines



The screenshot shows the EditPad Pro 8 interface with a Python file named `decimal.py` open. The window title bar indicates the file path is `c:\My Documents\Samples.epp [decimal.py]`. The menu bar includes File, Edit, Project, Search, Go, Block, Mark, Fold, Tools, Macros, Extra, Convert, Options, View, and Help. The toolbar shows icons for various file operations and a tab bar with several open files: `ExifTool.pm`, `geoip.php`, `decimal.py` (active), `xmlstats.rb`, `paypal.sql`, and `num_integral.r`. The main editor area displays Python code with line numbers from 404 to 442. Vertical dashed lines represent indentation indicators, placed at the start of each line of code that is indented relative to the previous line. The code includes an `except` block for `AttributeError`, a `def` for `setcontext`, a `def` for `getcontext`, and an `else:` block with its own `def` for `getcontext`. Comments in the code explain the purpose of the context management, such as fixing reloading issues and ensuring new contexts are copies of the default context.

```

404 except AttributeError:
405
406     #To fix reloading, force it to create a new context
407     #Old contexts have different exceptions in their dicts, making problems.
408     if hasattr(threading.currentThread(), '__decimal_context__'):
409         del threading.currentThread().__decimal_context__
410
411     def setcontext(context):
412         """Set this thread's context to context."""
413         if context in (DefaultContext, BasicContext, ExtendedContext):
414             context = context.copy()
415             context.clear_flags()
416             threading.currentThread().__decimal_context__ = context
417
418     def getcontext():
419         """Returns this thread's context.
420
421         If this thread does not yet have a context, returns
422         a new context and sets this thread's context.
423         New contexts are copies of DefaultContext.
424         """
425         try:
426             return threading.currentThread().__decimal_context__
427         except AttributeError:
428             context = Context()
429             threading.currentThread().__decimal_context__ = context
430             return context
431
432 else:
433
434     local = threading.local()
435     if hasattr(local, '__decimal_context__'):
436         del local.__decimal_context__
437
438     def getcontext(_local=local):
439         """Returns this thread's context.
440
441         If this thread does not yet have a context, returns
442         a new context and sets this thread's context.

```

Start an indentation indicator below every line at that line's indentation level if that line is followed by one or more lines that are indented further than that line. The indicator stops before the first line that is indented by the same amount or less than the line below which the indicator started. Indentation indicators run through all blank lines in between.

This option is useful as a fallback for file formats where you do have braces or keywords around blocks but for which you don't have a syntax coloring scheme to match up those braces or keywords nor a file navigation scheme to turn blocks into foldable ranges.

On Lines Surrounded By Matching Brackets

Start an indentation indicator below every line that has an opening bracket that is paired with a closing bracket on a later line, indicating the indentation level of the line the opening bracket is on even if the bracket is not at the start of its line.

Indentation indicators run until the line that the paired closing bracket is on. If any lines between the line with the opening bracket and the closing bracket are not indented beyond the line with the opening bracket then the option to allow indentation indicators to go through insufficiently indented lines determines whether the indentation indicator stops on the line before the insufficiently indented line or whether it goes through that line to continue until the line with the closing bracket. The above two screen shots show how the indentation indicator either stops before or goes through the insufficiently indented comment.

Showing indentation indicators on lines surrounded by matching brackets makes it very easy to spot whether all your blocks are correctly nested and indented and whether all your braces are correctly paired. All of EditPad Pro's predefined file types that have a syntax coloring scheme that supports bracket matching use this option.

```

EditPad Pro 8 - [*c:\My Documents\Samples.epp] [*WebPad.java]
File Edit Project Search Go Block Mark Fold Tools Macros Extra Convert Options View Help
WebPad.java Engine.jsl Calculator.vjsproj acetext.html RegexBuddyActionClic << >>

271 // Panel which allows for the enabling and disabling of all the actions.
272 private JPanel createPanel() {
273     textArea = new JTextArea();
274     JScrollPane scrollPane = new JScrollPane(textArea);
275     textArea.addMouseListener(new MouseAdapter() {
276         public void mousePressed(MouseEvent e) {
277             if(e.isPopupTrigger()) {
278                 popup.show(textArea, e.getX(), e.getY());
279             }
280         }
281         public void mouseReleased(MouseEvent e) {
282             if(e.isPopupTrigger()) {
283                 popup.show(textArea, e.getX(), e.getY());
284             }
285         }
286     });
287
288     JPanel panel = new JPanel(new BorderLayout());
289     // Insufficiently indented line
290     panel.setPreferredSize(new Dimension(450, 200));
291     panel.add(scrollPane, BorderLayout.CENTER);
292
293     return panel;
294 }
295
296 // Creates the status bar.
297 private JLabel createStatusBar() {
298     status = new JLabel("Ready...");
299     status.setBorder(BorderFactory.createEtchedBorder());
300
301     return status;
302 }
303
304 /*
305  * This method acts as the Action handler delegate for all the actions.
306  * The Cut, Copy and Paste Actions operate on the JTextArea.
307  */
308 public void actionPerformed(ActionEvent evt) {
309     String command = evt.getActionCommand();

```

```

271 // Panel which allows for the enabling and disabling of all the actions.
272 private JPanel createPanel() {
273     textArea = new JTextArea();
274     JScrollPane scrollPane = new JScrollPane(textArea);
275     textArea.addMouseListener(new MouseAdapter() {
276         public void mousePressed(MouseEvent e) {
277             if(e.isPopupTrigger()) {
278                 popup.show(textArea, e.getX(), e.getY());
279             }
280         }
281         public void mouseReleased(MouseEvent e) {
282             if(e.isPopupTrigger()) {
283                 popup.show(textArea, e.getX(), e.getY());
284             }
285         }
286     });
287
288     JPanel panel = new JPanel(new BorderLayout());
289 // Insufficiently indented line
290     panel.setPreferredSize(new Dimension(450, 200));
291     panel.add(scrollPane, BorderLayout.CENTER);
292
293     return panel;
294 }
295
296 // Creates the status bar.
297 private JLabel createStatusBar() {
298     status = new JLabel("Ready...");
299     status.setBorder(BorderFactory.createEtchedBorder());
300
301     return status;
302 }
303
304 /*
305  * This method acts as the Action handler delegate for all the actions.
306  * The Cut, Copy and Paste Actions operate on the JTextArea.
307  */
308 public void actionPerformed(ActionEvent evt) {
309     String command = evt.getActionCommand();

```

On Lines Inside Foldable Ranges

Start an indentation indicator below every line that starts a foldable range at the indentation level of that line. The indentation indicator stops at the last line in the foldable range. This works equally with automatic folding points (as configured above) and blocks that you've manually folded with Fold|Fold.

If any lines in the foldable range are not indented beyond the first line in the range then the option to allow indentation indicators to go through insufficiently indented lines determines whether the indentation indicator stops on the line before the insufficiently indented line or whether it goes through that line to

continue until the last line in the foldable range. The above two screen shots show how the indentation indicator either stops before or goes through the insufficiently indented comment.

As you can see in the screen shots, with EditPad Pro's Java file type, showing indentation indicators based on folding produces very similar results to indentation indicators based on brackets because the Java syntax coloring scheme matches up braces as brackets and the Java file navigation scheme matches up braces as foldable ranges. The screen shots show fewer indentation indicators based on folding because the automatic folding points were set to exclude detailed folding points.

```

271 // Panel which allows for the enabling and disabling of all the actions. ^
272 private JPanel createPanel() {
273     textArea = new JTextArea();
274     JScrollPane scrollPane = new JScrollPane(textArea);
275     textArea.addMouseListener(new MouseAdapter() {
276         public void mousePressed(MouseEvent e) {
277             if(e.isPopupTrigger()) {
278                 popup.show(textArea, e.getX(), e.getY());
279             }
280         }
281         public void mouseReleased(MouseEvent e) {
282             if(e.isPopupTrigger()) {
283                 popup.show(textArea, e.getX(), e.getY());
284             }
285         }
286     });
287
288     JPanel panel = new JPanel(new BorderLayout());
289 // Insufficiently indented Line
290     panel.setPreferredSize(new Dimension(450, 200));
291     panel.add(scrollPane, BorderLayout.CENTER);
292
293     return panel;
294 }
295
296 // Creates the status bar.
297 private JLabel createStatusBar() {
298     status = new JLabel("Ready...");
299     status.setBorder(BorderFactory.createEtchedBorder());
300
301     return status;
302 }
303
304 /*
305  * This method acts as the Action handler delegate for all the actions.
306  * The Cut, Copy and Paste Actions operate on the JTextArea.
307  */
308 public void actionPerformed(ActionEvent evt) {
309     String command = evt.getActionCommand();

```

The screenshot shows the EditPad Pro 8 interface with a Java file named WebPad.java. The code is as follows:

```

271 // Panel which allows for the enabling and disabling of all the actions.
272 private JPanel createPanel() {
273     textArea = new JTextArea();
274     JScrollPane scrollPane = new JScrollPane(textArea);
275     textArea.addMouseListener(new MouseAdapter() {
276         public void mousePressed(MouseEvent e) {
277             if(e.isPopupTrigger()) {
278                 popup.show(textArea, e.getX(), e.getY());
279             }
280         }
281         public void mouseReleased(MouseEvent e) {
282             if(e.isPopupTrigger()) {
283                 popup.show(textArea, e.getX(), e.getY());
284             }
285         }
286     });
287
288     JPanel panel = new JPanel(new BorderLayout());
289     // Insufficiently indented line
290     panel.setPreferredSize(new Dimension(450, 200));
291     panel.add(scrollPane, BorderLayout.CENTER);
292
293     return panel;
294 }
295
296 // Creates the status bar.
297 private JLabel createStatusBar() {
298     status = new JLabel("Ready...");
299     status.setBorder(BorderFactory.createEtchedBorder());
300
301     return status;
302 }
303
304 /*
305  * This method acts as the Action handler delegate for all the actions.
306  * The Cut, Copy and Paste Actions operate on the JTextArea.
307  */
308 public void actionPerformed(ActionEvent evt) {
309     String command = evt.getActionCommand();

```

The code is displayed with indentation indicators on the left side of the editor. The indicators are vertical lines of varying colors (blue, green, yellow, orange, red) that correspond to the indentation levels of the code. The code is also color-coded: comments are green, keywords are black, and identifiers are black. The line 289 is highlighted with a yellow background.

Indentation Indicator Color And Style

To change the color and visual style of the indentation indicators, customize the color palette and change the “Editor: Indentation level 1” through “Editor: Indentation level 4” colors. You can have 4 different colors for the indentation indicators to make it easier to follow deeply nested indentation. The 4 colors are cycled every 4 indentation levels. EditPad Pro can show indentation indicators up to 63 levels deep. You can make colors 1 and 3 as well as 2 and 4 the same if you want to have two alternating colors for indentation indicators. Or you can make all 4 the same if you want the indicators to be more subtle.

Selecting a vertical style for the indentation levels in the color palette makes indentation indicators appear as vertical lines. Selecting a background color applies that background color to the spaces and tabs used for indentation, whether they are visualized or not. If you allow indentation indicators to go through insufficiently indented lines then the text color of the indentation levels is applied to the insufficiently indented text.

Help File or URL for Keyword Help

When you press F1 in EditPad Pro, normally EditPad Pro's help file appears. But if you enable keyword help for a file type, and the main editor has keyboard focus when you press F1, then EditPad Pro can show help for the file type you're working with. If you selected a block of text that does not span more than one line before pressing F1, EditPad Pro shows help for the selected text. Otherwise, it shows help for the word under the cursor.

If you specify the full path to a HLP or CHM file, EditPad Pro automatically looks up the selected text or the word under the cursor in the index of the HLP or CHM file. You can click the (...) button to select a HLP or CHM file on your computer. If you are running Windows Vista or Windows 7 and you specify a HLP file, you may be prompted to download the WinHelp viewer from Microsoft if you haven't done so already.

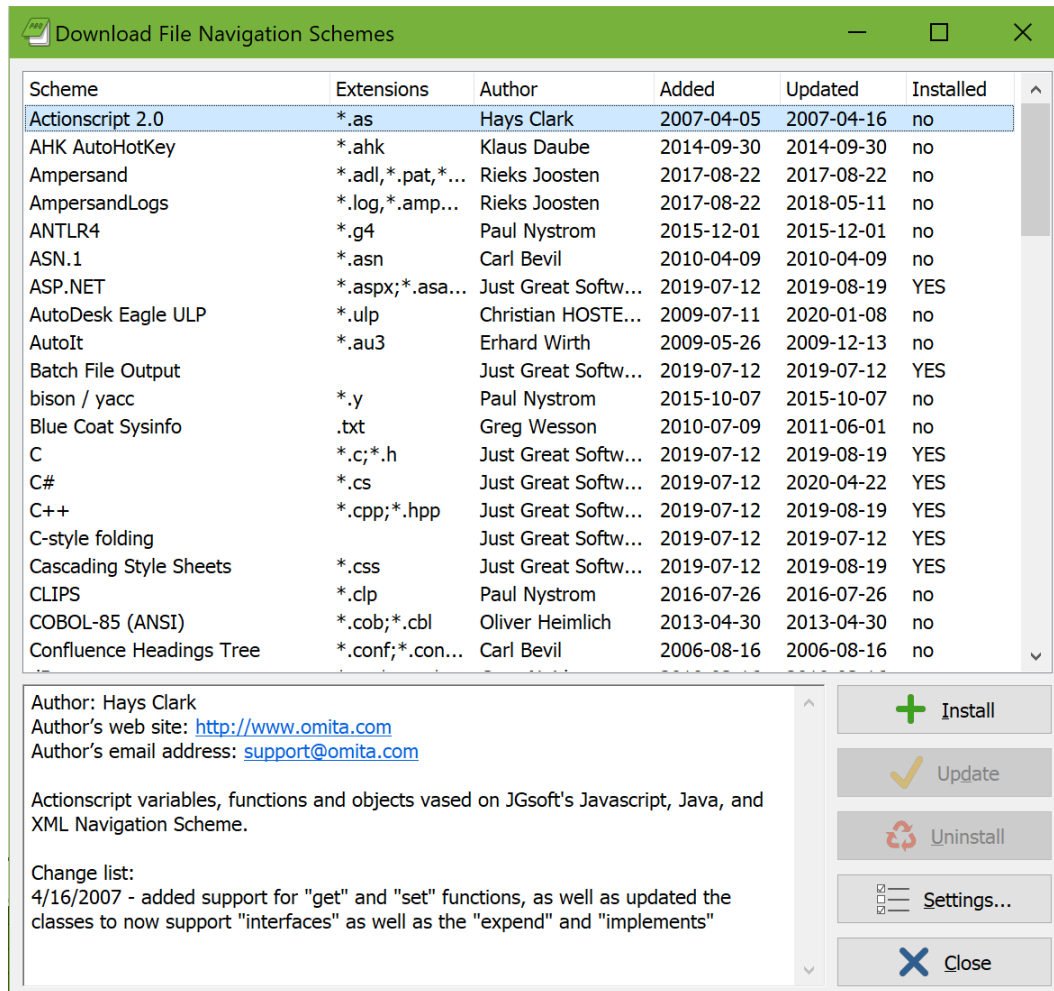
You can also specify the URL to a web page. If you do, use the %KEYWORD% placeholder in the URL. EditPad Pro substitutes this placeholder with the selected text or the word under the cursor. If you want to start a Google search when you press F1, for example, set the URL to `http://www.google.com/search?q=%KEYWORD%`. EditPad Pro automatically URL-encodes any special characters in the selected text.

Finally, you can specify the path to any document or a command line to any application, including command line parameters. You can use the %KEYWORD% placeholder anywhere on the command line. EditPad Pro substitutes the placeholder without giving special treatment to any characters in the selection or the word under the cursor.

Download File Navigation Schemes

When you click on the button to download file navigation schemes in the Navigation file type settings, EditPad Pro connects to the Internet. It downloads a list of custom syntax coloring schemes that have been created and generously shared by other EditPad Pro users. This list is then displayed in the window shown below.

Note: Any schemes you download are *not* automatically put to use. You will need to create or edit a file type to use a scheme that you downloaded.



The list presents you the following information about the available schemes:

- **Scheme:** The name of the scheme. It is possible that more than one scheme with the same name is available. If so, you can install all of them or only one, as you prefer.
- **Extensions:** The extensions typically used for the files that the scheme provides syntax coloring for. They are only shown for your information. You can apply any scheme to any file type, as you see fit.
- **Author:** The author of the scheme. If you have comments or suggestions about a particular file navigation scheme, you should contact this person.
- **Added:** The date the scheme was first added to the list of available schemes.
- **Updated:** The date the scheme was last updated.
- **Installed:** Indicates whether this scheme is installed on your computer or not.

If you click the **Install** button, EditPad Pro immediately downloads the scheme and saves it into the %APPDATA%\JGsoft\EditPad Pro 8 folder.

Click the **Update** button to download a scheme that you already installed again.

If you click the **Uninstall** button, EditPad Pro deletes the scheme from your computer when you close the window for downloading file navigation schemes.

If you are behind a proxy server, and EditPad Pro is unable to detect your proxy settings, you can change them by clicking the **Settings** button.

Using File Navigation Schemes

After downloading a file navigation scheme, it is *not* automatically put to use.

All the schemes that you download are listed in the File Navigation Scheme drop-down list on the Navigation page in the file type configuration. They appear in alphabetic order among the schemes that ship with EditPad Pro. This drop-down list is shown in the screen shot below.

To use one of the schemes, create or edit a file type, and select the file navigation scheme you want from the File Navigation Scheme drop-down list.

File Navigation Scheme Editor

With the File Navigation Scheme Editor, you can create your own file navigation schemes for use with EditPad Pro. You can also edit the schemes included with EditPad Pro, and those you've downloaded. The file navigation schemes files have a .jgfn extension. You can't edit them with any program except the File Navigation Scheme Editor. You can download the scheme editor at <https://www.editpadpro.com/fns.html>. The editor is a free download for all licensed EditPad Pro users. Full documentation is included.

The file navigation schemes included with EditPad Pro are stored in the folder where you have EditPad Pro installed, typically C:\Program Files\Just Great Software\EditPad Pro 8. On Windows Vista and Windows 7, the default security settings do not allow normal users to modify files under C:\Program Files. Therefore, schemes that you download and schemes that you edit are saved under your Windows user profile, typically C:\Users\yourname\AppData\Roaming\JGsoft\EditPad Pro 8. If a scheme with the same file name exists in both the user profile folder and the program files folder, only the scheme in the user profile folder will be available in EditPad. On the Navigation page in the file type configuration, schemes loaded from your Windows user profile are marked with (user).

15. Options | Preferences

When you pick Options|Preferences in the menu, the EditPad Preferences screen appears. Since there are a lot of things you can configure, the available settings are split into several tabbed pages.

EditPad is very configurable, and the number of choices may seem a bit overwhelming. Fortunately, if a certain option seems meaningless to you, you can simply leave it in its default state. The default settings have been carefully chosen so that EditPad is fully functional even if you do not make any changes in the Preferences screen at all. In its default configuration, EditPad will work like most Windows text editors.

All the options you can set in the Preferences screen are global settings. They affect EditPad's overall behavior. Many editing and display settings are available through the file type configuration. Those allow you to use different settings for different kinds of files.

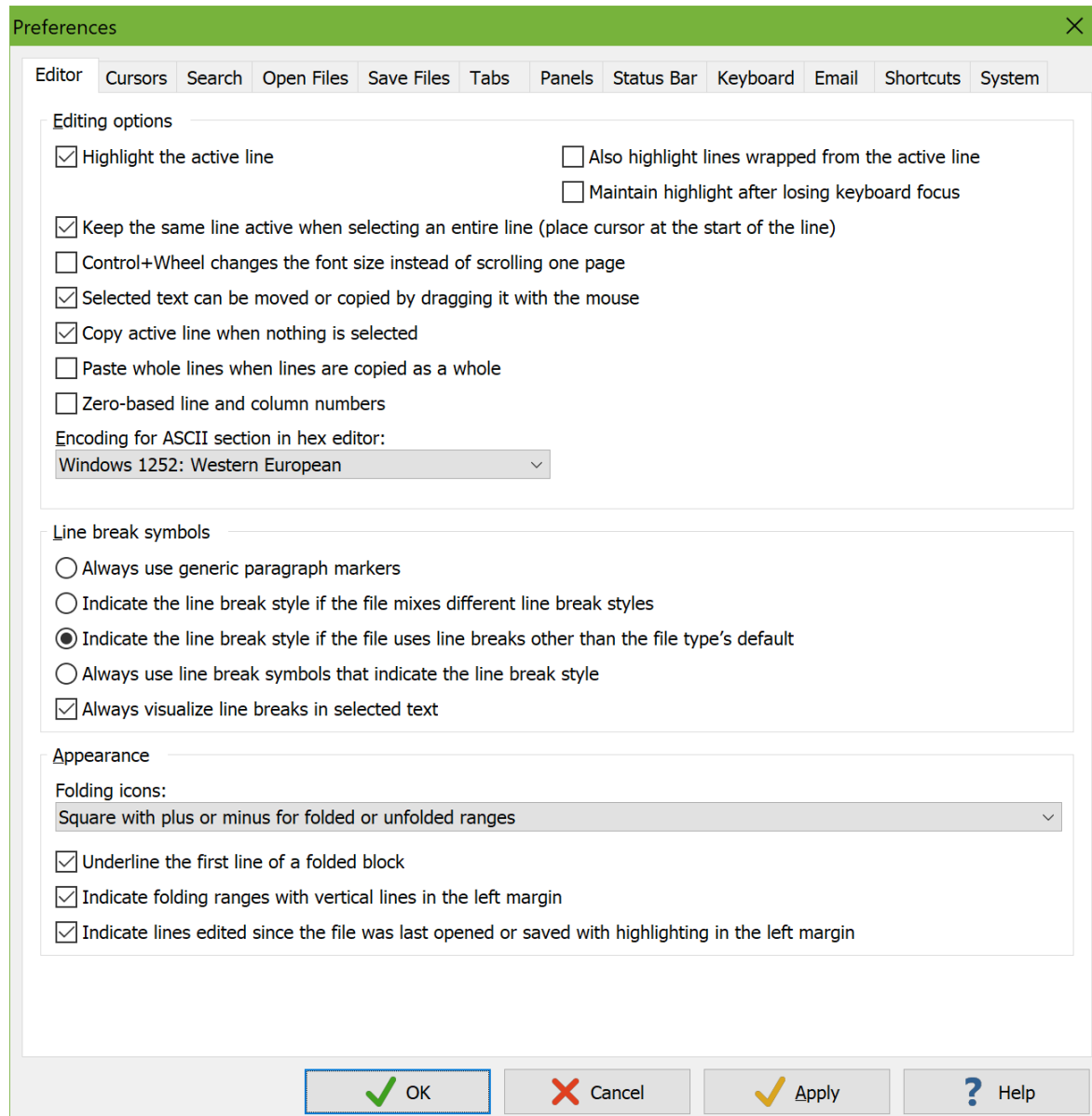
The available pages are:

- Editor
- Cursors
- Search
- Open Files
- Save Files
- Tabs
- Panels
- Status Bar
- Keyboard
- Email
- Shortcuts
- System

The main menu, all toolbars, and various right-click menus can be configured by right-clicking on the main menu or on any toolbar and selecting Customize.

Editor Preferences

On the Editor tab of the Preferences you can set the options that affect basic editing tasks that are not file type specific.



Editing Options

Highlighting the active line makes it easier to keep track of where you are in the file, particularly when switching between EditPad and other applications. The active line is the line the text cursor is on. You can turn on “also highlight lines wrapped from the active line” to highlight the entire paragraph that the active line is part of when word wrap is on. This option has no effect when word wrap is off. Just like the text cursor itself, the active line is only highlighted when the editor has keyboard focus. If you want it to be highlighted permanently, turn on “maintain highlight after losing keyboard focus” too. You can configure the

color of the active line in the color palette for each file type. Click on “editor: highlight active line” in the list. Click the Background Color button to change the highlight color.

You can select an entire line by double-clicking its line number, Ctrl+double clicking the line itself, or triple-clicking the line itself. The selection will include the line break at the end of the line. If you turn on “keep the same line active when selecting an entire line” then EditPad places the cursor at the start of the selected line. This way the selected line is the line that is highlighted as the active line. Shift+Arrow Up will expand the selection while Shift+Arrow Down will clear the selection. If you turn off this option then EditPad places the cursor at the end of the selection, which is at the start of the next line. The line below the selection will be highlighted as the active line. Shift+Arrow Up will clear the selection while Shift+Arrow Down will expand the selection.

In EditPad, rotating the mouse wheel scrolls the active file 3 lines up or down. Holding down the Ctrl key while rotating the mouse wheel scrolls the active file one screen up or down. Essentially, holding down the Ctrl key speeds up scrolling with the mouse wheel. In many other applications, Ctrl+Wheel zooms in or out. As a plain text editor, EditPad Pro does not have the ability to zoom. But EditPad Pro can mimic zooming by increasing or decreasing the font size of the active file. Turn on “Control+Wheel changes the font size instead of scrolling one page” if you prefer to change the font rather than to scroll quickly when using Ctrl+Wheel.

By default, “selected text can be moved or copied by dragging it with the mouse” is turned on. This allows drag-and-drop editing within EditPad. It also allows you to drag text from EditPad into other applications. People with limited dexterity using a mouse may find themselves accidentally moving text when trying to select text. Turning off this option prevents that.

In EditPad, the Edit|Cut and Edit|Copy are always enabled by default. If no text is selected, these commands cut or copy the active line. This allows you to quickly cut and copy whole lines, as it removes the need to select them. You can turn off the option “copy active line when nothing is selected” if you want EditPad to disable the Cut and Copy commands when no text is selected, as most Windows applications do.

Normally, when you paste text, that text is inserted at the position of the text cursor, regardless of where the cursor is placed and which text is being pasted. If you often copy and paste whole lines of text, you may want to turn on “paste whole lines when lines are copied as a whole”. A line is copied as a whole if all the text on that line and the line break at the end of the line are selected when you copy them. If the option to paste whole lines is on, and you copy a whole line in EditPad, and then paste it, the line is pasted before the line that the cursor is on, as if the cursor was positioned at the start of the line. This allows you to quickly copy and paste whole lines without worrying about the horizontal position of the text cursor. This option also affects the Block|Move and Block|Duplicate commands in the same way. If you move or duplicate a whole line, the line is moved or copied as if the cursor was on the start of the line that it is on. This option only affects whole lines copied in EditPad. When pasting text copied from another application, EditPad cannot determine whether it was a complete line or not.

Turn on “zero-based line and column numbers” if you want EditPad Pro to start counting lines and columns from zero instead of one. This option affects the status bar, as well as Options|Line Numbers and Options|Column Numbers.

Normally, the ASCII section in the hex editor uses the file type’s encoding or its non-Unicode encoding to translate bytes into characters for display in the right-hand section of the hex editor and for translating characters that you type into bytes. But the hex editor can only work with 8-bit code pages as it displays one character per byte. If neither the file type’s encoding nor its non-Unicode encoding are 8-bit code pages then

the hex editor falls back to the “encoding for the ASCII section in the hex editor” that you specify here in the Editor Preferences.

Line Break Symbols

You can show or hide line break symbols with the Options | Visualize Line Breaks command. You can set the default in the editor options for each file type. These two determine whether line breaks are visualized or not. The Line Break Symbols section in the Preferences determines how they are visualized.

- Always use paragraph markers: Always visualize line breaks using the ¶ symbol.
- Indicate the line break style if the file mixes different line break styles: Visualize line breaks using the ¶ symbol if all the line breaks use the same style. Visualize all line breaks using stair-stepped letters if the file uses two or more different line break styles. Page breaks are ignored when checking for different line break styles. You can always tell a page break from another kind of line break by the horizontal line that EditPad draws after a page break.
- Indicate the line break style if the file uses line breaks other than the file type’s default: Visualize line breaks using the ¶ symbol if all the line breaks other than page breaks in the file use the default line break style set in the encoding settings for the file type. If the file has at least one line break that is not a page break and that does not use the file type’s default line break style then all line breaks are indicated with stair-stepped letters. This option is the default. It makes it easy to see whether all line breaks are as expected (¶ symbol used for all line breaks) or whether there’s at least one odd line break (line-break specific symbols used for all line breaks).
- Always use line break symbols that indicate the line break style: Always visualize line breaks using stair-stepped letters.

Line breaks are visualized with stair-stepped letters as follows:

- CRLF pair (Windows && DOS)^{C_RL_F}
- Line Feed (Linux && OS X)^{L_F}
- Carriage Return (classic Mac)^{C_R}
- Form Feed (page break)^{F_F}
- Vertical Tab^{V_T}
- Next Line (ISO 8859)^{N_L}
- Line Separator (Unicode)^{L_S}
- Paragraph Separator (Unicode)^{P_S}

If you choose to show the generic ¶ symbol then you’ll need to rely on the status bar to determine the line break style used by the file.

By default, EditPad shows line break symbols for all line breaks that are part of a selected block, even when you’ve turned off the option to visualize line breaks. EditPad Lite, which does not have any options for visualizing line breaks, does this too. This makes it easy to tell the difference between selecting a line of text and selecting a line of text including the line break at the end of the line. In EditPad Pro, you can disable this by turning off “always visualize line breaks in selected text”. If you do this, there is no visual difference between selecting a line with or without the line break that terminates it.

Appearance

When you use the Fold|Fold command to fold a range of lines, EditPad Pro indicate the folding point or foldable range with a folding icon in the left margin. The icon is different depending on whether the folding point is folded or unfolded. The default is a square with a plus for folded ranges and a minus for unfolded ranges. This is similar to how folded nodes in tree views were indicated in older versions of Windows. The second option is a filled square for folded ranges and a hollow square for unfolded ranges. The final option is a rightward pointing chevron for folded ranges and a downward pointing chevron for unfolded ranges. This is similar to how folded nodes in tree views are indicated in recent versions of Windows. To change the color of the folding icons, customize the color palette and change the “Editor: Folding icons” color.

When the range is folded, only its first line remains visible. To make it more obvious that some lines are hidden, turn on “underline the first line of a folded block”. What this option really does is to layer the “Editor: Folded line” color from the color palette onto the first line of folded ranges. You can customize the color palette to apply a background highlight or change the text color instead of or in addition to underlining the line. You can of course change the underline style and color too.

When the range is unfolded, the second and following lines in the range are indicated with a vertical line in the left margin. This vertical line has a kink on the last line in the range to make it more obvious where the range ends. When folding ranges are nested, up to 3 vertical lines can appear side by side in the left margin. When folding ranges are nested 4 or more levels deep, the 4th and deeper levels do not get an extra vertical line. But they still add a little horizontal bump to the vertical line to indicate the last line in the folding range. If you find the vertical lines distracting then you can disable them. Then only the folding icons appear. You may prefer this style if you only work with automatic folding ranges that span logical blocks that are obvious from the file’s content and you don’t need an extra indicator to see where each range ends.

Turn on “indicate lines edited since the file was last opened or saved with highlighting in the left margin” if you want EditPad to keep track of which lines you’ve edited. When this option is off, EditPad does not track edited lines at all. So turning it on only shows indicators for lines you edit after turning on the option. To change the color of the indicators, customize the color palette and change the two “Editor: Line edited since file last opened/saved” colors. Specify a background color if you want to fill in the whole margin area (but not line numbers if they are shown). Specify a vertical line style if you want a thin vertical line immediately to the left of edited lines.

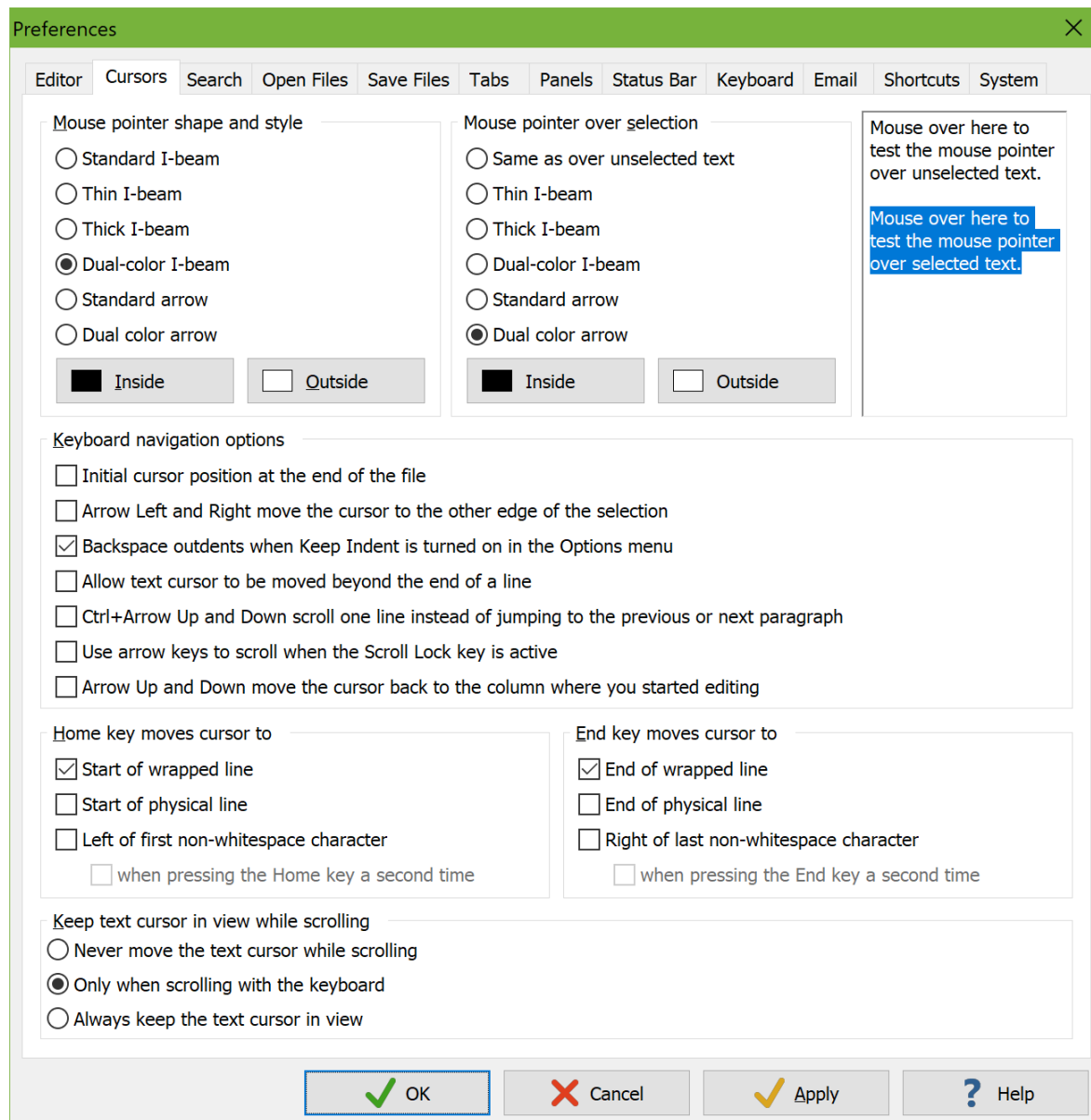
When you edit a line, it is highlighted in the margin with the “line edited since file last saved” color. When you save the file, all lines highlighted that way change their highlight to the “line edited since file last opened” color. If you edit such a line again, its highlight changes to “line edited since file last saved” again.

If you undo an edit that you made after you last saved the file (or if you didn’t save the file since last opening it) then the margin highlighting applied by the edit is undone as well. If you save the file and then undo an edit that you made before you saved the file then lines affected by the undo become highlighted as “line edited since file last saved”.

Cursors Preferences

On the Cursors tab in the Preferences screen, you can configure the appearance of EditPad's mouse pointer. You can also configure how EditPad responds to the navigational keys on the keyboard.

The appearance of the text cursor (blinking vertical bar) cannot be set in the Preferences. That can be set in the text layout configuration because EditPad's text cursor can indicate text direction.



Mouse Pointer Shape and Style

EditPad Pro allows you to select the mouse pointer shape and style. "Standard I-beam" is the standard mouse pointer for text editing controls, typically an I-shaped beam. "Thin I-beam", "thick I-beam" and "dual-color

I-beam” are custom EditPad cursors for which you can pick your own colors. By choosing colors that contrast well with the background color you’ve chosen for the editor, you can make the mouse pointer highly visible. “Standard arrow” is the regular Windows mouse pointer. “Dual color arrow” is another custom EditPad cursor in the shape of an arrow.

EditPad’s custom pointers may not work in all situations. Some remote desktop software, for example, cannot handle them. The mouse pointer may either be incorrectly displayed, or simply be invisible. In that case, simply select one of the standard pointers, which always work.

Keyboard Navigation Options

If you open a file you’ve never opened before, or if you turned off the option to preserve the cursor position on the Save Files page, then EditPad places the cursor at the very start of the file. You can turn on “initial cursor position at the end of the file” if you want the cursor to be at the end of the file instead. This works even with huge files. When you open a large file, EditPad scans the file for line breaks from top to bottom and from bottom to top at the same time. So both the start and end of the file are immediately accessible. Regardless of where you initially have the cursor, you can press Ctrl+Home to jump to the start of the file or Ctrl+End to jump to the end of the file immediately after opening it.

In some editors, like Notepad, pressing Arrow Left or Right moves the cursor one character left or right and clears the selection if there is one. In other editors, like Wordpad, pressing Arrow Left or Right while there is a selection puts the cursor on the left or right edge of the selection and then clears the selection. By default, EditPad behaves like Notepad. If you turn on “arrow Left and Right move the cursor to the other edge of the selection” then EditPad behaves like Wordpad. This option only has an effect when Block|Persistent Selections is off. When selections are persistent, the Left and Right arrow keys always move the cursor one character and never clear the selection. (Notepad and Wordpad do not support persistent selections.)

If you turn on Keep Indent in the Options menu or in the editor options in the file type configuration then pressing Enter to create a new line automatically duplicates all whitespace at the start of the previous line onto the new line. When Keep Indent is on, there are two ways in which the Backspace key can delete this whitespace at the start of a line. If you turn on “Backspace outdents” then pressing Backspace deletes as many whitespace characters as needed to line up the cursor with the previous indentation level. That is the amount of spaces used to indent the first line before the line the cursor is on that has less indentation than the horizontal position of the text cursor when you press the Backspace key. So if you start with a blank file, type 3 spaces, type text, press Enter (which indents the new line by 3 spaces), type 3 more spaces, type text, press Enter (which indents the 3rd line by 6 spaces) and then press Backspace, then 3 spaces are deleted to line up the cursor with the first line’s indentation. Pressing backspace again deletes 3 more spaces. If you turn off “backspace outdents”, then pressing Backspace always deletes one character. This option has no effect when Keep Indent is off. In that case backspace always deletes one character.

If you turn on “allow text cursor to be moved beyond the end of a line”, you can position the text cursor after the last character of the line. If you press the right arrow key when the cursor is at the end of the line, it moves one position to the right. If you click with the mouse beyond the end of the line, the cursor is placed where you clicked. If you start typing when the cursor is beyond the end of the line, EditPad automatically fills up the line with spaces up to the position where you started typing. When you do not allow the cursor to be moved beyond the end of a line, pressing the right arrow key when the cursor is at the end of a line moves it to the start of the next line. When you click the mouse beyond the end of the line, the cursor is placed after the last character on the line.

You can choose what happens when you press Control+Arrow Up or Down on the keyboard. By default, this scrolls the text one line up or down without moving the text cursor, as if you had clicked on the up or down arrow button on the scroll bar, or as if you had rotated the mouse wheel. Ctrl+Arrow Up and Down work this way in most programmer's text editors. If you prefer to use the mouse rather than the keyboard for scrolling, you can configure the Ctrl+Arrow Up and Down keys to make the text cursor jump to the next or previous paragraph. Ctrl+Arrow Up and Down work this way in some word processors such as Microsoft Word.

If you turn on “use arrow keys when the Scroll Lock key is active”, then you can push the Scroll Lock key on the keyboard to scroll with the arrow keys. Pressing any of the arrow keys then scrolls the view instead of moving the text cursor. The Home and End keys scroll to the top and the end of the file. Press Scroll Lock again to restore the normal arrow key behavior. If this option is off then EditPad Pro ignores the state of the Scroll Lock key.

In Windows text editors, pressing the Arrow Up or Down keys on the keyboard keeps the text cursor on the same column. For example, if you put the cursor on line 1, column 1 of an existing text file, type “abc”, and then press Arrow Down, then the cursor will be on line 2, column 4. If you type “abc” on the 2nd line, it will be “stair-stepped” relative to the “abc” on the first line. If you turn on the option “Arrow Up and Down move the cursor back to the column where you started editing”, then in the previous example, pressing Arrow Down moves the cursor to line 2, column 1. The horizontal movement caused by typing “abc” is undone by pressing Arrow Down. If you type “abc” on the second line, it will appear just below the “abc” on the first line. The result is that turning on this option makes it much easier to edit data arranged in columns.

Home And End Key Move Cursor

EditPad lets you configure the behavior of the Home key and End key independently. The same options are available for both keys.

The “wrapped line” and “physical line” options come into play when word wrap is enabled. A “wrapped line” is one line of text as it appears in EditPad. A “physical line” is the text between two line breaks or one paragraph of text. Turning “wrapped line” on and “physical line” off gives you the traditional behavior of the Home and End keys in a word processor. Turning “wrapped line” off and “physical line” on make the Home and End keys oblivious to word wrapping. Turning on both makes the first key press move to the start or end of the line (if not already there) and the second press to the start or end of the paragraph. A third press does not move the cursor back to the line that it was on originally.

Traditionally the Home and End keys move the cursor to the very start or very end of the line. In some modern editors, the Home key moves it to the left of the first non-whitespace character of the line it is on and the End key to the right of the last non-whitespace character. This is sometimes known as a “smart Home key” or “smart End key”. EditPad has these options too. If the cursor is already to the left/right of the first/last non-whitespace character then the Home/End key moves the cursor to the very start/end of the line. If the cursor is already at the very start/end of the line then it is moved to the left/right of the first/last non-whitespace character. Repeatedly pressing Home or End will keep moving the cursor between those two positions. If the cursor is on another position on the line then the option “when pressing the key a second time” determines what happens. If this is off then the first press of the Home/End key moves the cursor next to the first/last non-whitespace character. If it is on then the first key press moves the cursor to the very start or end of the line.

If you turn on the option to move the cursor next to the first/last non-whitespace character and also turn on both “wrapped lines” and “physical lines” then there are potentially four positions that the cursor could be

moved to depending on whether the line that the cursor is on has trailing whitespace and whether the paragraph that the line is part of has leading or trailing whitespace. If all four positions are possible then the first two key presses move the cursor to the two positions on the wrapped line as described in the previous paragraph. The third key press then moves the cursor to the first or last line in the paragraph, either at the very start/end or next to the first/last non-whitespace character depending on the “when pressing the key a second time” option. Further presses then cycle between those two positions on the first or last line in the paragraph.

Keep Text Cursor in View While Scrolling

“Keep text cursor in view while scrolling” determines what happens to the text cursor when you scroll the text. Normally, the text cursor keeps its position relative to the text when you scroll the text. This may cause the text cursor to be scrolled off-screen. Depending on your text editing habits, this may be desirable or undesirable. If you allow the text cursor to be scrolled off-screen, it keeps its position relative to the text. After scrolling, you can immediately continue typing at the same position. When you do so, the text is scrolled automatically to make the cursor visible again. If you choose to keep the text cursor visible, it is moved to the topmost or bottommost visible line when scrolling would otherwise have made it invisible. This way, you can always see exactly where the text cursor is pointing to.

By default, EditPad Pro keeps the text cursor visible when you scroll with the keyboard (by pressing Ctrl+Arrow Up/Down or Ctrl+Page Up/Down), but not when you scroll with the scroll bars or by rotating the mouse wheel. This is how most text editors for programmers work. EditPad Pro gives you the choice. Most general-purpose text editors do not allow keyboard scrolling at all.

Mouse Actions

Below you can find a list of action you can carry out in the editor using the mouse.

Dragging means to move the mouse before releasing the mouse button you pressed. If you move the mouse pointer to the edge of the editor space while dragging, the text will start to scroll automatically.

Modifier keys like shift or control must be pressed before pressing the mouse button and kept depressed until the mouse button is released.

Left click: Moves the text cursor to the spot where you clicked. Any text that was previously selected becomes unselected, unless you enabled persistent selections and clicked outside the selection.

Shift+Left click: Moves the text cursor and expands or shrinks the selection. If there is no selection, the text between the old and new cursor positions becomes selected. If you click outside of the selection, the selection plus the text between the selection and the new cursor position becomes selected. If you click inside the selection, the new selection is the text between the original start of the selection and the new cursor position.

Ctrl+Left click: Moves the text cursor to the spot where you clicked. Selects the line that you clicked on entirely, unless there already is a selection and you enabled persistent selections.

Ctrl+Shift+Left click: Moves the text cursor and expands the selection just like Shift+Left click does. In addition, if the selection starts and/or ends in the middle of a line, the selection will be expanded to include the partially selected lines entirely.

Left click+drag: When clicking outside the selection, a new selection is created from the point where you press the mouse button until the point where you release it. When clicking inside the selection, the selected text deleted and inserted again at the spot (outside the selection) where you release the mouse button.

Shift+Left click+drag: Expands or shrinks the selection like Shift+Left click, but then the text cursor is moved and the selection adjusted until you release the mouse button.

If you press Alt while changing the selection with any of the above left click methods, the selection becomes rectangular. If Block|Rectangular Selections is active, pressing Alt makes the selection linear.

Left double click: Moves the text cursor to the spot where you clicked. If you double-clicked on a word, the word becomes selected. If you double-click whitespace, all adjacent whitespace becomes selected. If you double-click anything else, all characters up to the preceding and following word become selected. All occurrences of the newly selected text will be highlighted if you turned on the option to make double-clicking highlight all occurrences of a word in the Search Preferences.

Shift+Left double click: Moves the text cursor and expands or shrinks the selection just like Shift+Left single click does. In addition, if the selection starts and/or ends in the middle of a word, the selection will be expanded to include the words at the start and/or end entirely.

Left double click+drag: A new selection is created from the point where you double-clicked until the point where you release it. If the selection starts and/or ends in the middle of a word, the selection will be expanded to include the words at the start and/or end entirely.

Shift+Double click+drag Expands or shrinks the selection like Shift+Left double click, but then the text cursor is moved and the selection adjusted until you release the mouse button. Words at the edge of the adjusted selection will be selected entirely.

Left triple click: Moves the text cursor to the spot where you clicked. Selects the line that you clicked on entirely (even when selections are persistent).

Shift+Left triple click: Moves the text cursor and expands or shrinks the selection just like Shift+Left single click does. In addition, if the selection starts and/or ends in the middle of a line, the selection will be expanded to include the partially selected lines entirely.

Left triple click+drag: A new selection is created from the point where you triple-clicked until the point where you release it. If the selection starts and/or ends in the middle of a line, the selection will be expanded to include the partially selected lines entirely.

Shift+Triple click+drag: Expands or shrinks the selection like Shift+Left triple click, but then the text cursor is moved and the selection adjusted until you release the mouse button. Lines part of the adjusted selection will be selected entirely.

Rotate wheel: Scrolls the text a single line up or down.

Shift+Wheel: Moves the text cursor a line up or down, like pressing the up or down arrow keys on the keyboard.

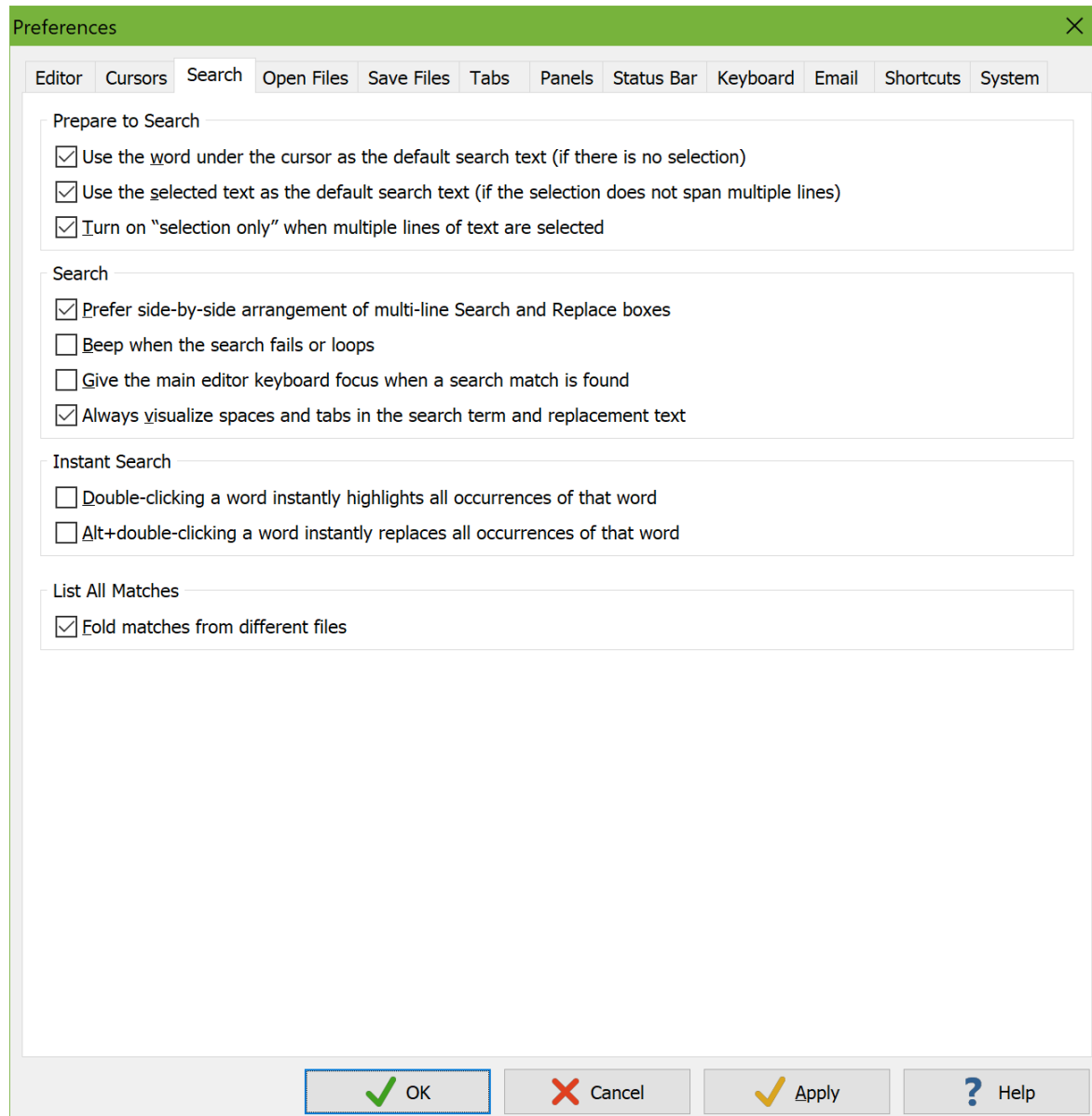
Ctrl+Wheel: Scrolls the text an entire screen up or down.

Shift+Ctrl+Wheel: Moves the text cursor a screen up or down, like pressing page up or down on the keyboard.

Scrolling the text with the wheel, the text cursor keeps its position relative to the text. This may cause the text cursor to be scrolled off screen and become invisible. If you do not like this, select “always keep the text cursor in view” in the Cursor Preferences. Then, scrolling with the wheel will also move the text cursor if needed to keep it visible.

Search Preferences

On the Search tab of the Preferences you can set the options that affect the Search panel and the commands in the Search menu.



Prepare to Search

When you press Ctrl+F or select Search|Prepare to Search in the menu, EditPad moves keyboard focus to the Search panel (and makes it visible if needed). EditPad can also prepare a default search term.

Turn on “use the word under the cursor as the default search text (if there is no selection)” if you want EditPad to use the word under the cursor in the main editor as your default search term when nothing is

selected. Turn on “use the selected text as the default search text (if the selection does not span multiple lines)” if you want the text that you selected before pressing Ctrl+F to be the default search term. You can turn on both options at the same time. If neither option is enabled or applicable then the Search panel opens with the most recently used search text. You can then use the selected text as the search text by right-clicking the Search box and selecting Selection as Search Text.

In EditPad Lite these options provide a quick way to search for a word or a bit of text (that you select) under or near the cursor. Though EditPad Pro supports this too, it provides an even quicker way. Search|Instant Find Next and Search|Instant Find Previous search for the next or previous occurrence of the word under the cursor or the selected text without using the Search panel at all. Any search term you entered on the Search panel remains.

EditPad never uses multiple lines of text as the default search text. If you want to search for multiple lines of text, switch to the multi-line search panel and then right-click the Search box and select Selection as Search Text.

EditPad can automatically turn on the Selection Only search option on if multiple lines of text are selected when you prepare to search. If you enable this, then EditPad also automatically turns off Selection Only when there is not a multi-line selection when you prepare to search.

Search

The multi-line search panel automatically arranges the Search and Replace boxes to optimally use the available space. If you make the panel short and wide then the Search and Replace boxes are placed side-by-side. If you make the panel tall and narrow then the Search and Replace boxes are placed above and below each other. This is not something that you can configure.

If you make the panel tall and wide enough so that there is enough room for either arrangement then EditPad places the Search box above the Replace box. In the default layout of EditPad’s panels, the Search panel has the same width as the main editor. So stacking the Search and Replace boxes vertically allows them to show a full line of text without wrapping. If one box contains more lines of text than the other box then EditPad adjusts their relative heights to give more space to the box that needs to display more lines. This avoids having to scroll one box to see all its lines while the other box is showing a lot of blank space.

If you turn on “prefer side-by-side arrangement of multi-line Search and Replace boxes” then EditPad places the Search box to the left of the Replace box when there is enough space for either arrangement. Both will take up half the width of the Search panel.

When a search action fails, EditPad Pro does nothing except flash the icon of the button you clicked on the Search panel. If this is not obvious enough for you, you can turn on the option “beep when the search fails or loops”. A standard “error” beep then sounds when a search fails. If the “loop automatically” search option is on and EditPad Pro finds a match after restarting from the beginning, then a standard “notification” beep sounds.

Turn on “give the main editor keyboard focus when a search match is found” if you want to be able to immediately edit the search match after doing a Find First or Find Next. You can press F3 on the keyboard to continue to the next search match even when the editor has keyboard focus and even after the search panel has been closed. But if you prefer to be able to continue editing the text you’re searching for rather than the text found, turn this option off.

Unintended spaces and line breaks may cause EditPad to apparently not find search terms that do occur in the text but without those extra spaces or line breaks. To avoid this, you can turn on the option to always visualize spaces and tabs in the search panel. When this option is off, the Search panel only visualizes these characters when you've turned on the option to visualize spaces and line breaks in the file you're editing.

Instant Search

Turn on “double-clicking a word instantly highlights all occurrences of that word” if you want to be able to invoke the Search|Instant Highlight command by double-clicking. Double-clicking still selects the word you've double-clicked, regardless of whether this option is on or off.

Turn on “Alt+double-clicking a word instantly replaces all occurrences of that word” if you want to be able to invoke the Search|Instant Replace command by double-clicking while holding down the Alt key.

A syntax coloring scheme can assign a double-click action to certain text. Many of the schemes included with EditPad display URLs as hyperlinks, for example. Double-clicking the URL opens it in your browser. When you double-click text that has a double-click action, only that action is performed. Nothing is selected and nothing is highlighted or replaced.

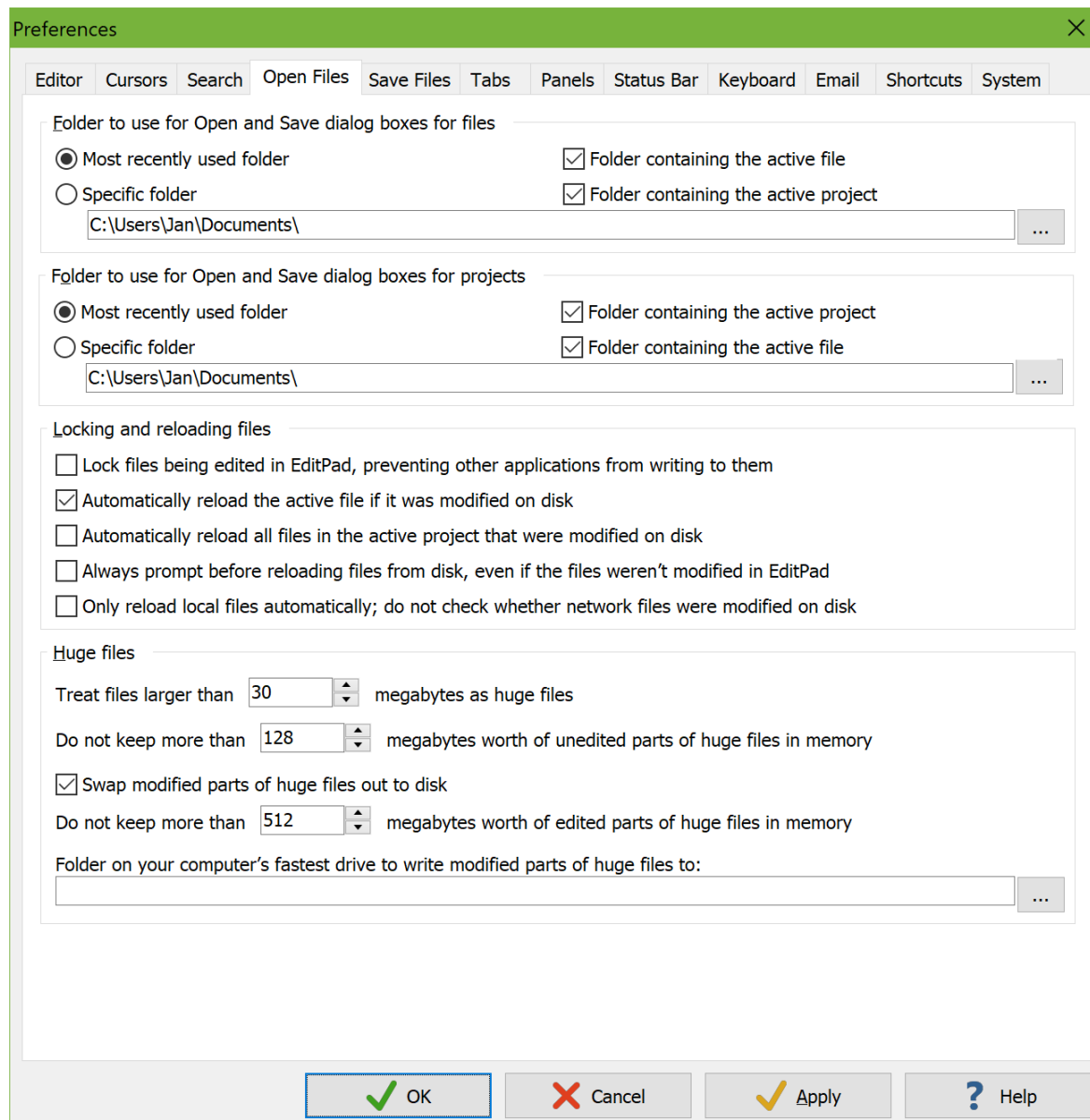
If you double-click text that does not have a double-click action then the word that you double-clicked on is selected. Double-clicking selects the word regardless of whether or not Instant Highlight or Instant Replace is invoked on that word.

List All Matches

Search|List All Matches and Search|Find on Disk can display a list of matches in a side panel. When the Search Matches panel shows matches from multiple files, you can fold each file's search matches away under the file's name. Turn on “fold matches from different files” if you want the matches to be folded this way by default.

Open Files Preferences

On the Open Files tab in the Preferences screen, you can configure how EditPad should handle files when opening them.



Folder to Use for Open and Save Dialog Boxes

You can choose the initial folder of the file selection dialog boxes that appear when you want to open or save a file. First, you can choose which folder is the default when you do not have any file or project open in EditPad, or when the active file or project is untitled. The default folder can either be the last folder you opened a file from or saved a file into, or be a specific folder such as your “My Documents” folder.

In addition to specifying the default folder, you can choose if the folder containing the active project or file should be used instead of the default. If you turn on the active file option, the folder containing the active file will be used for opening or saving a file when you have a file open. If you turn on the active project option, and the active file option is off or the active file is untitled, the folder containing the active project will be used instead. The default folder is only used if you turned off both options, or the active file and project are both untitled.

You can also set a default folder for opening and saving projects in the same way.

Locking and Reloading Files

By default, EditPad does not keep a lock on files. This means that other applications or users can modify files that you have open in EditPad. If you don't want this to happen, turn on "lock files being edited in EditPad, preventing other applications from writing to them". The option whether or not to lock files only applies to files smaller than the huge file threshold (see below). Files larger than the huge file threshold are always locked. EditPad Pro doesn't read those files into memory entirely, so it needs to keep access to the original file on disk.

When not locking files, EditPad can automatically reload files that have been modified on disk by another application or user. If you turn on this option, EditPad will check whether a file's modification date has changed each time you switch between files in EditPad, and each time you switch between EditPad and another application.

By default, when you switch from another application to EditPad, EditPad only checks whether the active file was modified on disk. If you have multiple files open in EditPad that were all modified on disk, you'll only be prompted to reload the active file. When you switch to the other files, you'll be prompted for each of them at the moment you switch to them. If you turn on "automatically reload all files in the active project that were modified on disk", then EditPad Pro checks all files in the active project whenever you switch from another application to EditPad. If multiple files were modified, you'll be prompted for all of them at the same time. You'll be able to choose for each file whether you want to reload it or not. Regardless of whether this option is on or off, when you switch files within EditPad, EditPad checks whether the file that you're switching to needs to be reloaded.

When automatically reloading, EditPad will prompt when you've modified the file in EditPad, but not when you haven't modified the file in EditPad. Turn on "always prompt before reloading files from disk" to make EditPad prompt to reload the file even when you haven't modified it in EditPad.

When editing files over a slow network connection, checking whether a file was modified on disk may cause a brief delay when switching between files in EditPad or when switching from another application to EditPad. If you experience this, turn on "only reload local files automatically; do not check whether network files were modified on disk". Then the options to automatically reload will only apply to files stored on your own computer. Modern storage devices allow EditPad to do this check instantly. EditPad won't check files stored on another PC or server that you're accessing via the Windows network, so a slow network doesn't slow down your work in EditPad.

Huge Files

EditPad Pro is capable of handling files of almost any size, including files larger than 4 gigabytes (a common file size limit). However, loading such large files entirely into memory would quickly exhaust the available memory of most computers, slowing down the system to a crawl or even causing Windows itself to crash.

The total amount of RAM available to EditPad is all your computer's RAM if you are using the 64-bit version of EditPad. It is a maximum of 2 GB (2048 MB), or the actual amount of RAM in your PC if it is less than 2 GB, when using the 32-bit version of EditPad. You are using the 64-bit version of EditPad if Help|About indicates "x64" after the version number. If it indicates "WOW64" then you are using the 32-bit version of EditPad on 64-bit Windows. You should install the 64-bit version so that EditPad can use all of your computer's RAM. If it indicates "x86" then you have a 32-bit version of Windows which cannot run the 64-bit version of EditPad.

To be able to edit files of any size, including files larger than the amount of RAM available to EditPad, you can specify a threshold for "huge files". Files smaller than this threshold are read into memory entirely for maximum performance and flexibility. You can choose any threshold between 10 megabytes and 1/16th of the total amount of RAM available to EditPad. Increasing this limit improves EditPad's performance. But it makes EditPad use more memory, particularly when you open many large files at the same time.

Files larger than the threshold are treated as "huge files". EditPad reads huge files into memory as needed. When EditPad reads a part of a huge file into memory, it may unload another part of the same file or of another file that EditPad isn't working with at the moment. If the unloaded part is needed again later, EditPad reads it back from the original file. Frequent unloading and reloading of file parts slows EditPad down. You will certainly notice this if your files are stored on a mechanical hard drive or on a network drive. For best performance, copy huge files over to a local SSD.

To be able to partially read files as needed, EditPad Pro needs to keep a lock on those files. This means that EditPad Pro will lock files larger than the huge file threshold, even when you've turned off the option to keep a lock on open files.

Unedited parts of huge files can always be read again from the original file. If your PC is starved for RAM, you can tell EditPad Pro to not keep more than 50 megabytes worth of unedited parts of huge files in memory. But if your PC has plenty of RAM available, you can increase this limit up to 1/8th of the amount of RAM available to EditPad to improve performance, particularly if your huge files aren't stored on a fast SSD.

Edited parts of huge files cannot be read from the original file. EditPad Pro keeps all edited parts in memory until it exhausts the maximum number of megabytes worth of edited parts that you allow it to keep in memory. You can set this limit between 50 MB and half of the amount of RAM available to EditPad. When the limit is exceeded, EditPad writes some of the edited parts to a temporary file and unloads them from memory to bring memory usage of edited parts back under the limit. If an edited part that was unloaded needs to be displayed or edited again, it is reloaded from the temporary file. When you save a huge file all its parts that were edited become unedited parts again.

The size of the temporary file is only limited by available disk space. You can specify the folder in which this temporary file should be created. For best performance, that folder should be on a fast SSD. If you do not specify a folder, EditPad uses the folder that Windows uses for temporary files. Each EditPad instance creates only one temporary file to hold the edited parts of all the huge files you edit with it.

You can turn off the option to swap edited parts of huge files to disk. EditPad Pro then keeps all those parts in memory even if their size exceeds the amount of RAM in your PC. EditPad Pro then relies on Windows itself to swap out the memory used by EditPad and other applications to disk using the Windows page file.

To avoid constantly re-reading entire files and to avoid taking too much (CPU) time on very large files, EditPad disables some of its functionality for huge files. This way you don't need to wait on EditPad while processing huge files and you don't have your laptop's battery drained unnecessarily. Syntax coloring is disabled for huge files, unless you've selected a "fast" syntax coloring scheme in the file type configuration. Fast syntax coloring schemes are schemes that don't require the whole file to be processed. File navigation schemes are also disabled as they always require the whole file to be processed. The File Navigator will remain blank and automatic folding points will not appear. EditPad does not preserve the cursor position and folding for huge files. It will instantly open the file with the cursor at the top or at the bottom. A position in the middle of the file would have to wait for line breaks to be scanned, which can take some time for huge files. Switching from hexadecimal mode to text mode also resets the cursor to the start or end of the file.

Save Files Preferences

On the Save Files tab in the Preferences screen, you can configure how EditPad should handle files when saving them.

The screenshot shows the 'Preferences' dialog box with the 'Save Files' tab selected. The dialog is organized into several sections:

- Backup copies:**
 - ☐ Do not create backup copies
 - ☐ Single backup appending .bak extension
 - ☐ Single backup with .~* extension
 - ☐ Single backup appending ~
 - ☐ Multi backup appending .bak, .bak2, ...
 - ☐ Multi backup appending .YYYYMMDDhhmmss
 - ☒ Multi backup prepending "Backup X of ..."
 - ☐ Hidden history folder
 - ☒ Quick backups
 - ☒ Make backups of .epp project files
 - ☒ Reduce backups older than
 - 14 days to one per day
 - 8 weeks to one per week
 - ☒ Limit number of backup copies
 - 99 backup copies per file
 - ☒ Limit backup copies by size
 - 50 MB of backups per file
 - ☐ Save backups in a specific folder or subfolder
- Preserve File Status:**
 - ☒ Preserve file settings
 - ☒ Preserve cursor position and folding
- Closing files with unsaved changes:**
 - ☒ Prompt to save unsaved changes
 - ☐ Save unsaved changes automatically
 - ☐ Discard unsaved changes
- Closing empty files:**
 - ☐ Prompt to save empty files
 - ☐ Save empty files automatically
 - ☒ Keep previously saved file
- Closing untitled files:**
 - ☒ Prompt to save untitled files
 - ☐ Save untitled files automatically
 - ☐ Discard untitled files
- Automatic save:**
 - ☒ Save working copies every
 - 5 minutes
 - ☐ Save actual files every
 - ☐ Save working copies in a specific folder or subfolder
 - ☐ Automatically save and reopen the workspace when shutting down and restarting EditPad Pro
 - C:\Users\Jan\Documents\EditPad Pro 8 Workspa

At the bottom of the dialog are four buttons: OK (with a green checkmark), Cancel (with a red X), Apply (with a yellow checkmark), and Help (with a blue question mark).

Backup Copies

To make sure you never lose any data in case you change your mind after saving a file, you should turn on one of the options to create backup copies. The “single backup” options keep one backup copy of each file in the same folder as the original. The “multi backup” options keep multiple backup copies in the same folder. The “hidden history” option creates a hidden “__history” folder below each folder, and put the backup copies there. The “multi backup” and “hidden history” options work best with EditPad Pro’s File History. The File History lets you easily compare, delete and revert to backup copies.

By default, EditPad uses a quick method to make backups. It moves the original file into its backup location, and then writes the file you're saving as a new file. This works just fine when editing files on your Windows PC or on a Windows server. It may cause issues if you're editing a file on other systems, such as a Linux server that makes itself visible on the Windows network using Samba. It can also cause issues on Windows when editing files with special (security) attributes. When EditPad moves the file into a backup location and writes a new file, the newly saved file may not have the same UNIX file permissions or Windows security attributes of the original file. To prevent this, turn off "quick backups". EditPad will then make backups by copying the original file, and the original file will be modified with the new text you're saving.

When you save a project, EditPad Pro creates an .epp file. EditPad Pro then automatically saves this file at regular intervals. If you turn on the option "make backups of .epp project files" then EditPad Pro creates a backup copy of that .epp file before overwriting it. The backup is named according to the "single backup" or "multi backup" option that you selected. The backup is made only if the project hasn't been backed up yet since you last opened it or if the previous backup was made on a previous day. So if you open and close the same project multiple times per day, you get one backup copy per session. If you keep the same project open for days on end without restarting EditPad Pro or your PC then you get one backup copy per day. The options for reducing and limiting backup copies and for putting them in a specific folder also apply to backup copies of .epp files.

When choosing "multi backup" or "hidden history", you should enable one or more of the three backup limitation options to prevent an ever-increasing number of backup files from taking up too much disk space. The limits are enforced on the backup copies of one particular file when EditPad needs to create another backup copy of that file.

EditPad can selectively reduce backup copies based on their age. Backup copies older than your specified number of weeks will be reduced to one per week, keeping only the first backup made during each week. Backup copies older than your specified number of days will be reduced to one per day, keeping only the first backup made during each day. This reduction may reduce the backup copies below the maximum number or the maximum size that you specify.

The benefit of selectively weeding out backup copies this way is that it allows you to go back further in time with a given number of backup copies. Say you save a particular file 10 times a day, 5 days a week, over the course of a year. If you reduce backup copies to one a day after 7 days and to one a week after 4 weeks then you need to allow a total of 113 backup copies to keep a year's worth of backups. If you just kept the 113 most recent backup copies then you'd only be able to go back two weeks and a day.

If you limit the number of backup copies for each original file then EditPad deletes the oldest backup copies to bring the total number under the limit if reducing them to one a day and one a week didn't already sufficiently reduce the number of backup copies. The backup copy that is about to be created is counted towards the limit. So you'll never have more backups than the number you specified.

If you limit the backup copies by size, then EditPad counts the total size of all the backup copies that remain after reducing them by age and reducing their number. EditPad then deletes the oldest remaining backup copies to bring the total size of the existing backups below the size limit that you specified. This may end up deleting all existing backup copies if the newest backup copy alone is larger than the total size limit. The size of the backup that is about to be created is not taken into account. This ensures that each file always has at least one backup copy, even if that backup copy alone is larger than the backup size limit. So after the limits are enforced and after the new backup copy is created, the total size of the backups of the file that was saved may exceed the size limit by the size of the most recent backup copy.

Normally, EditPad saves backups in the same folder as the original files, or in a hidden “__history” subfolder in that folder. If you’d like the backups to be created in a subfolder of the folder that contains the original file then turn on the “save backups in a specific folder or subfolder” option and type in the name of the subfolder. If you’d like all backups to be saved into a particular folder then turn on “save backups in a specific folder or subfolder” and specify the full path to the folder where you would like to keep your backup copies. To make sure there are no conflicts between backup copies of files with the same name saved in different folders, EditPad will create subfolders under the path you specify to organize the backup copies.

If you save backups on a separate drive and that drive is running low on disk space then you may want to use a file manager to delete all files older than a certain age and/or larger than a certain size. EditPad does no bookkeeping of its backup files. Deleting them using another application causes no issues. EditPad checks the backup folder for existing backup copies each time you save a file. In EditPad Pro you can also use the File History to delete all backup copies of the current file or project.

Closing Unsaved Files

The “closing files with unsaved changes” option gives you three choices as to what EditPad should do when you try to close a file that was previously saved and to which you have made changes that were not yet saved to disk:

- Show a warning message and give you one last opportunity to decide whether to save the file or to discard your changes. This is the default since most other software behaves this way too.
- EditPad can automatically save the file without asking you first. This is the best way to make sure you will not lose any work. For safety, you should also select one of the options to make multiple backup copies so you can revert any accidentally saved changes later.
- The last option tells EditPad to close the file without warning. Any unsaved work will be lost. Use this option with caution.

The option “closing empty files” is a special case for closing previously saved files that have unsaved changes that made the file completely empty. You get the same three choices. You may want to make a different choice. Prompts to save are intended to prevent data loss caused by forgetting to save your work. But if a file was made empty, accidentally or automatically saving the empty file could be a bigger risk as it deletes the data that was previously in the file.

- Show a warning message and give you one last opportunity to decide whether to save the file or to discard your changes. If you choose to save when prompted, EditPad will overwrite the existing file with an empty file.
- EditPad can save empty files automatically. For safety, you should also select one of the options to make multiple backup copies so you can restore the contents of any file that was emptied accidentally later.
- The default option tells EditPad to close empty files without warning, keeping the previously saved file. This option is the least likely to cause data loss.

The “closing untitled files” option gives you three similar choices as to what EditPad should do when you try to close a file that has never been saved and is not empty:

- Show a warning message and give you one last opportunity to decide whether to give the file a name and save it or to discard the file. This is the default since most other software behaves this way too.

- EditPad can automatically save the file without asking you first. This is the best way to make sure you will not lose any work. The file will be saved in the default folder for files you specified in the Open Files Preferences. It's name will continue to be "Untitled". If you open the file again and edit it, EditPad will again use the "closing untitled files" setting for that file. EditPad will not use the "closing files with unsaved changes" setting unless you save the file with a name other than "Untitled".
- The last option tells EditPad to close the file without warning. Any unsaved work will be lost. Use this option with caution.

Windows 8 and later don't really give applications the opportunity to prompt to save unsaved changes when you want to shut down or restart your PC (or Windows decides to restart on its own). If you have working copies enabled (see below), then EditPad does not prompt to save your changes when Windows shuts down or restarts. Instead it ensures that all working copies are up-to-date and then immediately allows Windows to proceed with the shutdown or restart. If you chose to automatically save or discard without prompting, then EditPad will do that instead when Windows shuts down or restarts.

Automatic Save

If you want EditPad Pro to automatically save your work regularly, select to save working copies. EditPad Pro then saves a copy of all modified files every couple of minutes. When working on Document.txt, a file "Working copy of Document.txt" is saved into the same folder as Document.txt. When you save Document.txt or close Document.txt without saving, the working copy is automatically deleted.

If the document is not properly closed, because of a power loss or a software crash, the working copy is not deleted. When you open Document.txt next time, EditPad Pro automatically opens both Document.txt and its working copy. You can then choose which file you want to keep and which you want to delete, possibly after using Extra | Compare Files to assist with your choice.

Enabling working copies is a particularly good idea when working on a laptop running on battery power. EditPad ensures that all working copies are saved when Windows notifies that your PC is going to sleep or into hibernation. This way you don't lose any work if your laptop runs out of battery power while it was asleep.

Note: If you have both "Document.txt" and "Working copy of Document.txt" open in EditPad Pro, and you start editing the former, the latter is automatically overwritten after a few minutes. So if you want to keep the working copy, you should use File | Save As before editing the base document. If you don't want to keep the working copy, use File | Delete to remove it.

If you'd like all working copies to be saved into a particular folder, turn on "save working copies in a specific folder or subfolder" and specify the full path to the folder where you would like to keep your working copies. If you'd like the working copies to be created in a subfolder of the folder that contains the original file and choose the name of that subfolder, turn on the "save working copies in a specific folder or subfolder" option and type in the name of the subfolder.

If you do not specify the full path to a folder for working copies then working copies of unsaved files are saved in the "specific folder" you have specified in the "folder to open for open and save dialog boxes for files" section on the Open Files tab in the Preferences, even if you have "most recently used folder" selected there. If you restart EditPad Pro after a system crash, the working copies of unsaved files are opened automatically when you start EditPad.

Instead of saving working copies every few minutes, you can tell EditPad to save the actual files every few minutes. If you use this option it's probably best to also enable one of the "multi backup" options and not to restrict the number of backup copies by count or by size so you'll have older backups to go back to if your edits don't pan out. When EditPad saves actual files every few minutes it still creates working copies of untitled files at the same interval. If you save an untitled file then EditPad removes the working copy and starts saving the actual file at the specified interval.

If you want to continue working with the same set of files the next time you start EditPad Pro, turn on "automatically save and reopen workspace". You can also choose the file into which EditPad Pro should have its workspace. The workspace contains a list of projects and files that you have open. The next time you start EditPad Pro, those projects and files are automatically reopened. If you restart EditPad Pro after a system crash, it opens the workspace along with all working copies of all files in the workspace.

Preserve File Status

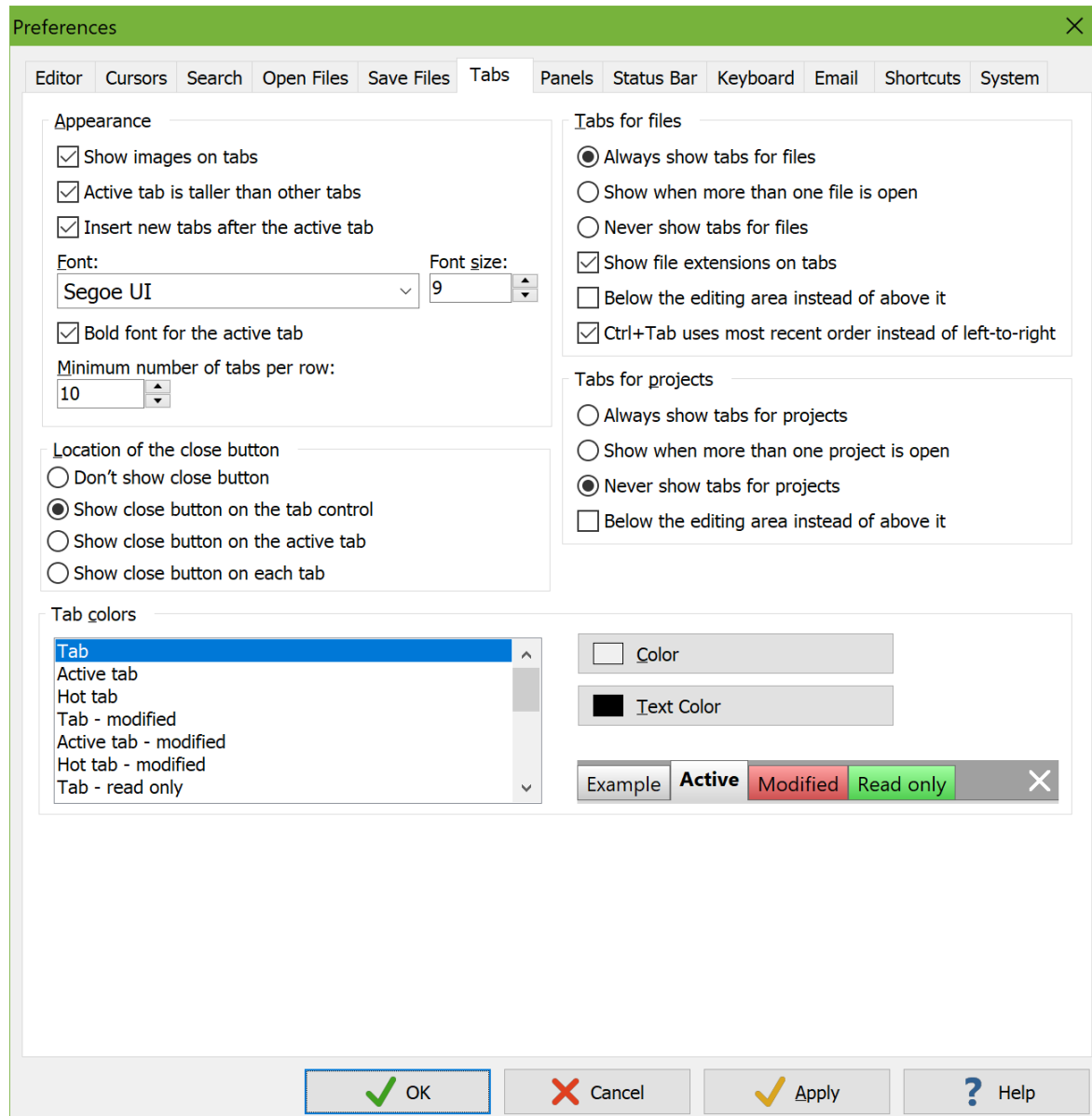
By default, EditPad Pro remembers a whole bunch of status information for the files that you edit. This information is stored for all the files listed in the File|Open and File|Favorites submenus. It is also saved into .epp project files whenever you save a project. This way, each file will appear the next time you open it in EditPad Pro the way it did last time.

With "preserve file settings" turned on, EditPad Pro stores the settings from the Options menu that you've changed from their defaults in the file type configuration. With "preserve cursor position and folding" turned on, EditPad Pro remembers the position of the text cursor in the file, which part of the file was selected, bookmarks, and folding.

Opening large files may take a bit longer when preserving the cursor position, as EditPad Pro then has to scan the file for line breaks up to the cursor position before it can show the file. Because of this, EditPad Pro won't remember the cursor position for files larger than the huge file threshold.

Tabs Preferences

On the Tabs tab in the Preferences screen, you can configure how tab pages work in EditPad Pro. Tabs enable you to quickly switch between files and projects. You can also quickly open and close files via the tabs.



Appearance

By default, EditPad shows images on tabs. File tabs show the icon associated with the file's extension. Project tabs show the Project|New Project icon for unmanaged projects, and the Project|Managed Project icon for managed projects.

The active file tab color is different from the other tabs. If this distinction is not clear enough, you can either change the tab colors as explained below or turn on “active tab is taller than other tabs”. When this option is selected, the active tab will have a larger height and a piece of background will be visible above all the other tabs. Turning off this option gives the tab control a much more compact look.

When you open files or create new files, the option “insert new tabs after the active tab” determines whether the tabs for the newly opened or created files are inserted immediately after the active tab. If you turn off this option then the new tabs are added at the end after the last tab.

You can also change the font and font size used for the captions shown on the tabs. A bigger font is more readable but allows fewer tabs to fit on the screen. You can make the active tab use the bold variant of your chosen font to make it stand out even more.

Normally EditPad makes its tab exactly as wide as needed to show its image (if enabled) and its entire caption. This may not allow many tabs to be visible at the same time if you have files or projects with long names. You can specify a minimum number of tabs to be visible per row. Tabs with long captions are then made narrower if needed to allow the specified minimum number of tabs to fit.

Location of the close button

EditPad can show an X button to make it easy to close tabs. You can have one close button at the right-hand edge of the tab control, after the last tab. Clicking this closes the active tab. Alternatively, you can have a close button directly on the tab. You can have it on the active tab only, or on each tab. A close button on each tab makes it easy to close tabs with a left click, but you have to be careful not to accidentally close tabs when switching between them.

If you choose not to show the close button, you can still close the active tab with the File|Close command, or any tab by clicking on it with the mouse wheel. Not showing the close button allows more tabs to fit on the screen.

Tabs for Files and Projects

EditPad Pro can display up to two rows of tabs. One row for files, and another row for projects. The file tabs will only show the files in the active project. Switching between project tabs also switches between the files active in those projects.

Tabs are very convenient when working with a relatively small set of files. When working with hundreds or even thousands of files, EditPad Pro’s Files Panel is a more appropriate tool. If you prefer to use the Files Panel, you can disable the tabs to save screen space. By default, EditPad always shows the tabs for files, but only shows the tabs for projects when you have more than one project open.

Pressing Ctrl+Tab on the keyboard switches between file tabs. If you turn on “use most recent order when switching tabs with Ctrl+Tab”, then pressing Tab repeatedly while holding down Ctrl moves back through tabs you’ve previously activated, in reverse order. Unlike the Go|Back in Edited Files, Ctrl+Tab remembers all tabs you’ve activated, regardless of whether you edited those files.

When you release the Ctrl key after pressing Tab one or more times, the tab that is activated is moved to the last position in the list of recently activated tabs. So if you press Ctrl+Tab and release both keys and do this

repeatedly, you'll be switching back and forth between two files. Essentially, Ctrl+Tab switches between files in EditPad like Alt+Tab switches between applications in Windows.

If you turn off “use most recent order when switching tabs with Ctrl+Tab”, pressing Ctrl+Tab moves through the file tabs from left to right, and pressing Shift+Ctrl+Tab moves through file tabs from right to left.

Tab Colors

You can configure the colors of the tabs. Select an item in the list and click the Color and Text Color buttons to change the background and text colors of the selected tab style.

- Tab: Regular tab
- Active Tab: The tab of the file you are currently viewing
- Hot Tab: The tab underneath the mouse pointer
- Tab - modified: Tab of a modified file
- Active Tab - modified: The tab of the file you are currently editing
- Hot Tab - modified: The tab of a modified file underneath the mouse pointer
- Tab - read only: Tab of a read only file
- Active Tab - read only: The tab of the read only file you are currently viewing
- Hot Tab - read only: The tab of a read only file underneath the mouse pointer
- Tab button: Color of the X button directly on a tab
- Pressed tab button: Color of the X directly on a tab when you've clicked on it with the mouse
- Hot tab button: Color of the X button directly on a tab underneath the mouse pointer
- Tab control background: The empty space above and to the right of the tabs
- Tab control button: Color of the X and other buttons at the right end of the tab row
- Pressed tab control button: Color of the X or other button when you've clicked on it with the mouse
- Hot tab control button: Color of the X or other button underneath the mouse pointer
- Disabled tab control button: Color of disabled buttons at the right end of the tab row. The scrolling buttons become disabled when you can't scroll further.

Tab Actions

You can do more with the tabs in EditPad, beyond activating a file by clicking on its tab.

To quickly close a tab, move the mouse pointer over it and push down the wheel button on your mouse. If the file didn't have any unsaved changes, EditPad will close it instantly, without even activating it.

You can access other common file-related commands such as File|Save and File|Close All but Current quickly by clicking on one of the tabs with the right-hand button of your mouse. If the tab you right-clicked on isn't the active tab, EditPad will activate the file first. If a command you often use isn't available in the right-click menu, you can add it by customizing the toolbars and menus.

Right-clicking a file tab also shows a few special commands such as the Read Only command for toggling the file's read-only status, and Copy Path to Clipboard for placing the file's full path onto the clipboard.

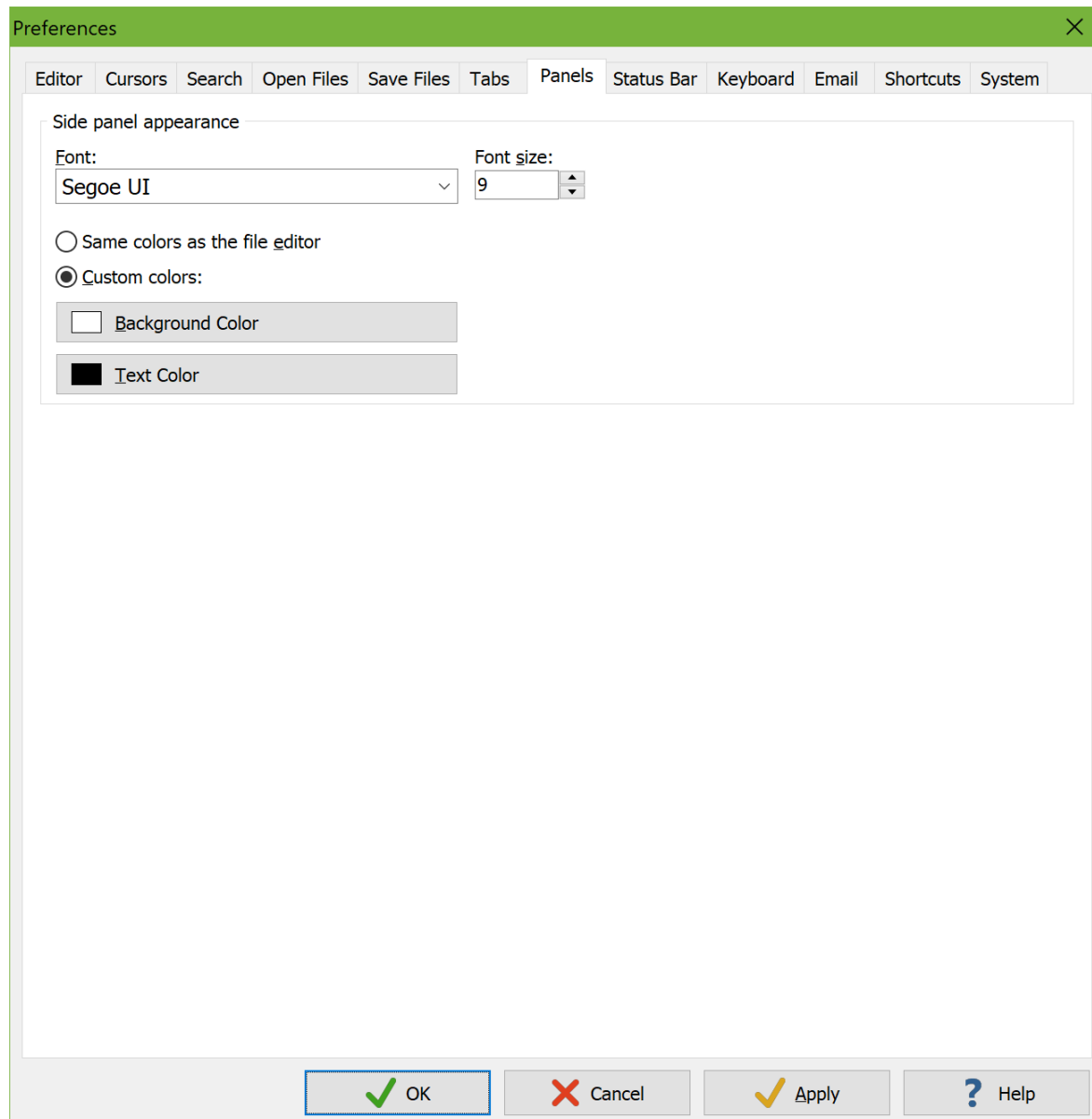
To quickly start with a blank, untitled file, double-click on the empty space after the last tab. Other common commands for opening files and commands affecting all files are available in the context menu that appears when you right-click on the empty space after the last tab. This menu is also configurable.

Right-clicking on a project tab shows a context menu with various commands to manage the project. You can configure this menu.

To quickly start with an empty project, double-click the empty space after the last project tab. This empty space too has a configurable context menu. It shows commands for opening projects and commands affecting all projects.

Side Panel Preferences

On the Panels tab in the Preferences screen, you can change the appearance of EditPad's side panels.



You can select a font face from the drop down list. Set the font size using the spinner box.

If you select "same colors as the file editor" then the side panels use the text and background colors of the plain text color in the color palette of the file type of the active file. If you use different palettes for different file types, then the side panel colors will change when you switch between files that use different palettes. This can cause a very slight delay if you have many side panels open.

If you select "custom colors" then the side panels always use the same colors. Click the Background Color button to change the background of the side panels. Click the Text Color button to change the font's color. If

you select a light background and a dark text color, then turning on View|Dark Theme swaps the colors. If the theme is already dark then selecting a dark background and a light text color makes View|Dark Theme swap the colors when you turn it off.

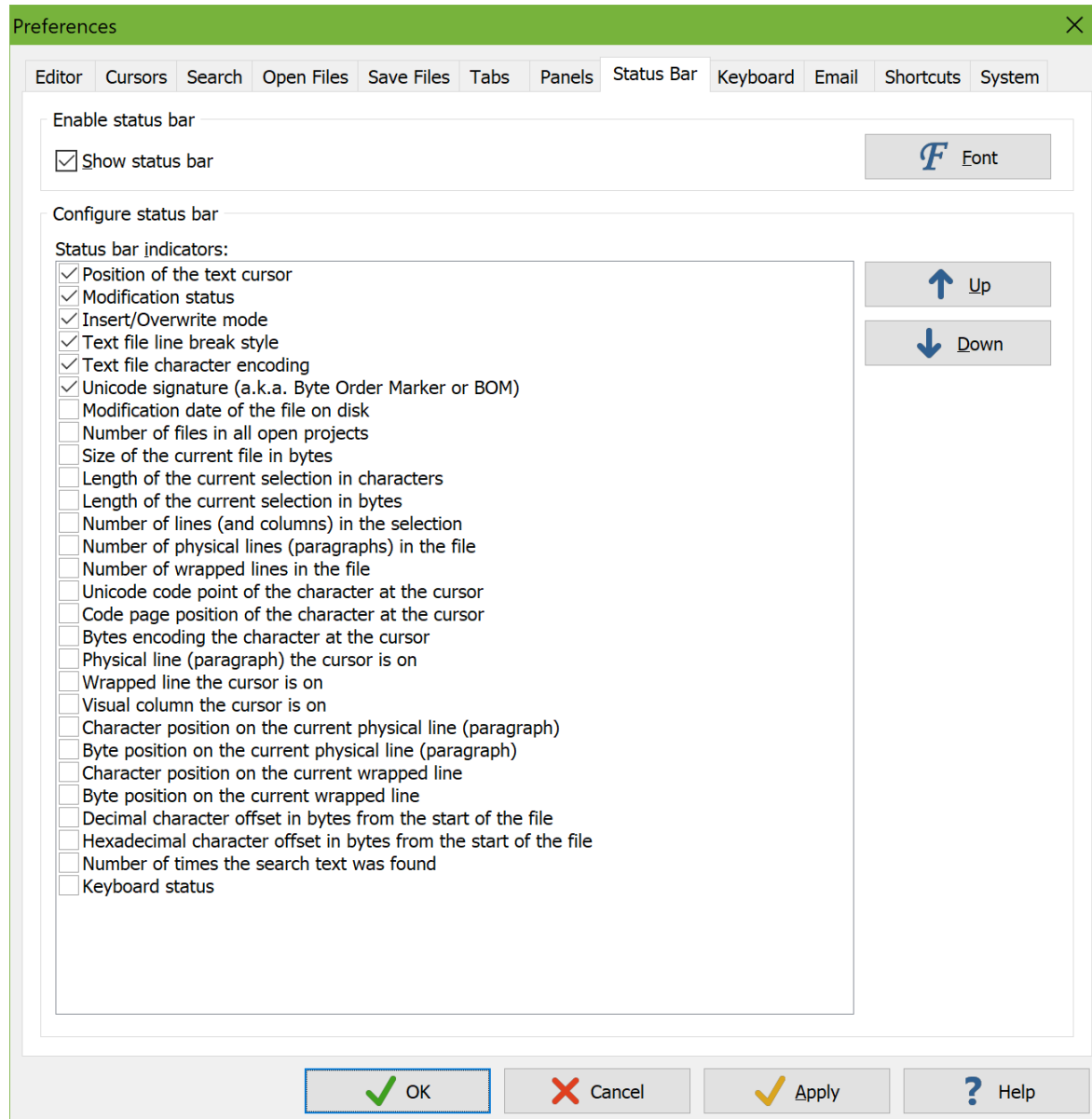
These settings are used by the Clip Collection, Files Panel, Explorer Panel, FTP Panel, File History, and File Navigator panels.

These settings are not used by the Search Panel. The Search and Replace boxes always use the same font and colors as the file you're editing. The font and colors for files can be set via Options|Configure File Types.

The Character Map panel uses the colors you specify for the side panels. But it always uses the same font as the file you're editing.

Status Bar Preferences

While both EditPad Lite and Pro have a status bar, it is only configurable in EditPad Pro. EditPad Pro can show many more indicators on the status bar. You may not want to clutter the screen with all of them. Select Options | Preferences in the menu, and click on the Status Bar tab.



You can show or hide indicators on the status bar by marking or clearing the checkboxes at the left of the items in the list. You can also hide the status bar altogether by turning off “show status bar”. Click the Font button to select a different font face or font size for the status bar. The status bar automatically adjusts its height and the widths of its indicators to the size of your font.

You can change the order in which the items are shown on the status bar by dragging and dropping them in the list. Click on an item, hold the mouse button down and drag it to the spot where you want it and release the mouse button. The topmost item in the list will become the leftmost item on the status bar.

The following status bar indicators are available:

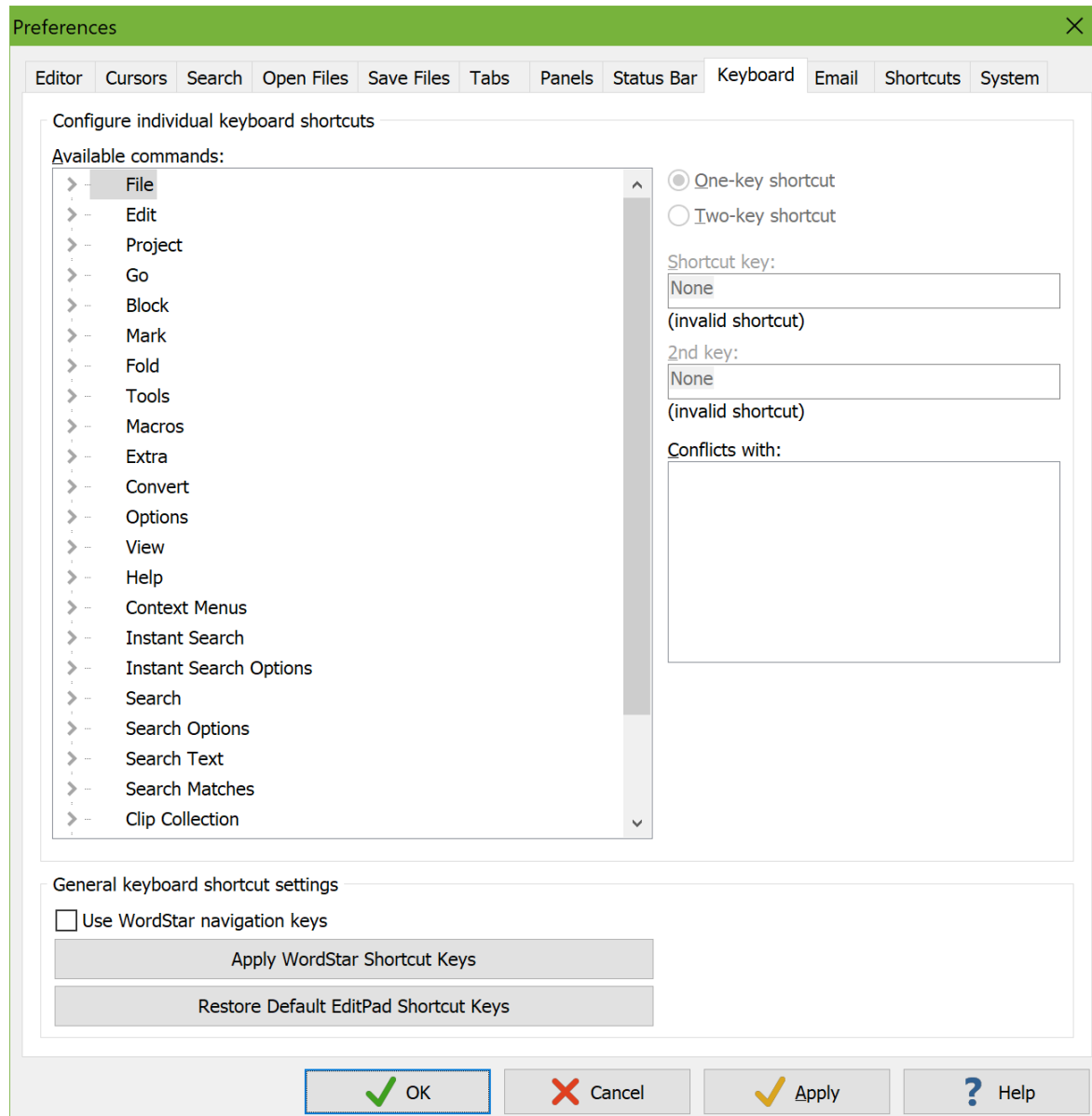
- Position of the text cursor: Number of the line and column the cursor is on. When word wrap is on, the line number depends on the “count physical lines only” setting in File Type|Editor. It is the same number shown in the margin when Options|Line Numbers is on. The column number indicates the visual column, which can differ from the number of characters to the left of the cursor. A single tab character may be multiple columns wide. Combining characters like diacritics may have no width. In hexadecimal mode, Clicking this indicator invokes Go|Go to Position.
- Modification status: Indicates “Modified” if the file has unsaved changes in EditPad. Indicates “Read Only” if the file is read-only. Indicates “Tail” if the file was opened with File|Tail. Otherwise the indicator is blank. Clicking it when blank makes the file read-only. Clicking it when showing “Read Only” makes the file editable.
- Insert/Overwrite mode: Indicates “Insert” when typing into the file pushes forward existing text to the right of the cursor. Indicates “Overwrite” when typing into the file overwrites existing text to the right of the cursor. Clicking this indicator toggles between insert and overwrite mode.
- Text file line break style: This indicator can show a number of things depending on how consistent the file is in its use of line break styles. If the file uses a single line break style, then that style, such as CRLF, is indicated. If a file uses a single line break style plus page breaks, then that style is indicated +FF, such as CRLF+FF. If a file uses one of CRLF, LF, or CR exclusively, but also contains Unicode line breaks, then this is indicated with +Unicode, such as CRLF+Unicode. If a file uses one of CRLF, LF, or CR exclusively, but also contains vertical tabs and/or page breaks (and possibly also Unicode line breaks) then this is indicated with +Other, such as CRLF+Other. If the file uses a mixture of CRLF, LF, and CR then this is indicated as "Mixed" with the dominant line break style between parentheses. If the file uses a mixture of Unicode line breaks then this is indicated as "Unicode" with the dominant style between parentheses. Clicking this indicator invokes Convert|Line Break Style. This command tells you how many line breaks of each style the file contains, with options to convert them. If you use this command to specify a specific line break style to be used by the Enter key and the file contains line breaks of different styles then the status bar indicator changes to “Forced” with the chosen style between parentheses.
- Text file character encoding: Indicates the encoding EditPad uses to interpret the bytes in this file as characters. Clicking the indicator invokes Convert|Text Encoding.
- Unicode signature (a.k.a. Byte Order Marker or BOM): Indicates “BOM” if the file uses a Unicode encoding and has a Unicode signature or Byte Order Marker at the start of the file. Indicates “no BOM” if the file uses a Unicode encoding and does not have a Unicode signature. Indicates “---” if the file does not use a Unicode encoding. You can click the indicator to toggle between “BOM” and “no BOM”. This will add the BOM to the file or remove the BOM from the file when you save the file if “preserve presence or absence of the byte order marker in existing files” is enabled in the file type’s encoding settings.
- Modification date of the file on disk: The date and time that the file was last saved.
- Number of files in all open projects: Total number of files in all the projects that you have open in EditPad Pro. For managed projects, that includes open files, closed files, and outside files.
- Size of the current file in bytes: Total size in bytes the file would have on disk if you were to save it.
- Length of the current selection in bytes: Size of the selection in bytes. Depending on the encoding used by the file, the size in bytes may or may not correspond with the size in characters.
- Length of the current selection in characters: Number of characters in the selection. Tabs are counted as a single character. Combining characters such as diacritics are counted if they are typed separately from their base character.

- Number of lines (and columns) in the selection: Number of lines that are (partially) selected. When word wrap is on, all wrapped lines are counted, regardless of the “count physical lines only” setting in File Type|Editor. If the selection is rectangular, the indicator follows the number of lines with a colon and a second number indicating the number of visual columns in the selection. This is the width of the rectangular selection.
- Number of physical lines (paragraphs) in the file: Number of lines in the file, counted as if word wrapping were off.
- Number of wrapped lines in the file: When word wrap is on, this indicator counts the number of lines in the file, including wrapped lines. When word wrap is off, the number of physical lines is shown.
- Unicode code point of the character at the cursor: Indicates the Unicode code point of the character to the right of the cursor in hexadecimal and in decimal. This indicator works even when the file’s encoding is not Unicode.
- Code page position of the character at the cursor: If the file uses a legacy code page, this indicator shows the index in that code page of the character to the right of the cursor in hexadecimal and decimal notation. If the file uses Unicode, the Unicode code point is indicated instead.
- Bytes encoding the character at the cursor: Indicates the actual bytes in the file that represent the character to the right of the cursor. The bytes are shown in hexadecimal notation like you would see in hexadecimal mode. For encodings that store files as pure ASCII using character escapes for non-ASCII characters, this indicator shows the ASCII text used to escape non-ASCII characters.
- Physical line (paragraph) the cursor is on: Number of the line the cursor is on, counting lines as if word wrap were off.
- Wrapped line the cursor is on: Number of the line the cursor is on. When word wrap is on, wrapped lines are counted too, regardless of the “count physical lines only” setting in File Type|Editor.
- Visual column the cursor is on: The visual column position of the cursor. This can differ from the number of characters to the left of the cursor. A single tab character may be multiple columns wide. Combining characters like diacritics may have no width.
- Character position on the current physical line (paragraph): Number of characters on the line to the left of the cursor. When word wrap is on, all characters before the cursor up until the previous line break in the file are counted, even if those characters are wrapped into multiple lines. Tabs are counted as a single character. Combining characters such as diacritics are counted if they are typed separately from their base character.
- Character position on the current wrapped line: Number of characters on the line to the left of the cursor. When word wrap is on, only the characters before the cursor on the same wrapped line are counted. Tabs are counted as a single character. Combining characters such as diacritics are counted if they are typed separately from their base character.
- Byte position on the current physical line (paragraph): Number of bytes on the line to the left of the cursor. When word wrap is on, all characters before the cursor up until the previous line break in the file are counted, even if those characters are wrapped into multiple lines.
- Byte position on the current wrapped line: Number of bytes on the line to the left of the cursor. When word wrap is on, only the characters before the cursor on the same wrapped line are counted.
- Decimal character offset in bytes from the start of the file: Decimal representation of the number of bytes in the file before the cursor position, counting all bytes from the start of the file.
- Hexadecimal character offset in bytes from the start of the file: Hexadecimal representation of the number of bytes in the file before the cursor position, counting all bytes from the start of the file.
- Number of times the search text was found: Incremented by commands such as Search|Find Next and Search|Find Previous (thus showing the number of matches stopped on rather than the total number of matches in the file). Reset to zero (or one if a match is found) by commands like Search|Find First. Shows the total number of matches found by commands like Search|Replace All and Search|Copy Matches. Invokes Search|Count Matches when clicked.

- Keyboard status: If you have pressed the first key in a two-stage keyboard shortcut then the first key is shown in this status bar indicator. If not, the indicator shows NUM, CAPS, and/or SCROLL if the Num Lock, Caps Lock, and/or Scroll Lock keys on the keyboard are active.

Keyboard Preferences

On the Keyboard tab in the Preferences screen, you can change the shortcut key combination for each item in EditPad's main menu.



Configure Individual Keyboard Shortcuts

To assign a new shortcut key combination to a menu item, select the menu item from the list. Then click on the box labeled “shortcut key”. Press the key combination you want on the keyboard. To remove a key combination from an item, click on the “shortcut key” box and press the delete or backspace key on the keyboard.

Each shortcut key combination can be used by only one menu item. If the shortcut key combination you assigned to the currently selected menu item is already used for one or more other menu items, those other menu items will be listed in the box labeled “conflicts with”.

You can also assign shortcut keys to tools and to macros when configuring tools and recording macros. You cannot assign the same shortcut key to both a menu item and a tool or macro.

If you run out of key combinations, EditPad also supports two-key keyboard shortcuts. When you press the first key combination of a two-key shortcut, EditPad will do nothing except remember that you pressed that key. The next key you press is then taken as the second key. If it forms a valid two-key shortcut, the corresponding action is executed. If not, neither key press will have any effect.

To configure a two-key shortcut, click on the “two-key shortcut” radio button. Then click on the “shortcut key” box and press the first key combination. The first key must be either a Ctrl+Letter combination (possibly including Shift and/or Alt), or a function key (possibly including Ctrl, Shift and/or Alt). Up to 15 different key combinations can be used as the first key in a two-key pair. Then click on the “2nd key” box and press the second key combination in the two-key shortcut. The second key combination can be any key, including a letter without a modifier (Ctrl, Shift or Alt). If you specify a letter without a modifier, both the letter without the modifier and the letter with the same modifiers as the first key will work. E.g. if you assign Ctrl+K, B to the Begin Selection command, you can begin a selection by pressing Ctrl+K followed by B, or Ctrl+K followed by Ctrl+B to begin a selection. Specifying a letter without modifiers as the second key makes it easier to quickly press both keys, since it doesn’t matter if you release the Ctrl key before the second key (the letter B in the example) or after it.

When using two-key keyboard shortcuts, it is a good idea to add the keyboard status indicator to the status bar. When you press the first key in a two-key combination, the keyboard indicator will show that key. That way you know the next key you press will be interpreted as the second key in the combination. After you press the second key, the indicator reverts to showing num lock, caps lock and scroll lock.

General Keyboard Shortcut Settings

EditPad supports the classic Wordstar navigation keys for people who are used to them. WordStar is an old word processor that used Ctrl+Letter key combinations to navigate through the document. WordStar was created at a time when most keyboards did not yet have separate arrow key blocks. The checkbox only enables the navigation keys. It does not change any of the key combinations that you assigned to EditPad’s menu items. Note that many of these navigation keys conflict with standard Windows shortcut keys. E.g. Ctrl+S is the standard Windows shortcut for File|Save. In WordStar, Ctrl+S moves the text cursor one position to the left, just like the left arrow key.

Click the “Apply WordStar Shortcut Keys” button to assign WordStar key combinations to various menu items. E.g. Ctrl+K, S will be assigned to File|Save. Shortcut keys assigned to EditPad menu items that don’t have a WordStar equivalent will not be changed. The assignment is a one-time event. You can freely reconfigure any of the changed keys.

Click the “Restore All Default Shortcut Keys” button to restore the default EditPad shortcuts for all menu items. It will also turn off the option to use WordStar navigation keys, since many of those conflict with EditPad’s default shortcuts.

The keyboard navigation keys cannot be configured. They are always available, even when you’ve enabled WordStar navigation keys.

Wordstar Navigation Keys

EditPad Pro supports the classic Wordstar navigation keys for people who are used to them. WordStar is an old word processor that used Ctrl+Letter key combinations to navigate through the document. WordStar was created at a time when most keyboards did not yet have separate arrow key blocks. Note that many of these navigation keys conflict with standard Windows shortcut keys. E.g. Ctrl+S is the standard Windows shortcut for File|Save. In WordStar, Ctrl+S moves the text cursor one position to the left, just like the left arrow key.

You can enable the WordStar navigation keys in the Keyboard Preferences. When enabled, they are recognized by every full text editor control in EditPad, such as the main editor, the search box, the replace box, the clip editor, etc.

The shortcuts with Ctrl+Q are two-key combinations. First, press Ctrl+Q. Then release the Q key and press the second letter in the key combination. Whether you press Ctrl along with the second key or not makes no difference.

Cursor navigation keys

Ctrl+S moves the text cursor one position to the left.

Ctrl+D moves the text cursor one position to the right.

Ctrl+E moves the text cursor one line upward.

Ctrl+X moves the text cursor one line downward.

Ctrl+A moves the text cursor to the start of the previous word or the end of the previous line, whichever is closer.

Ctrl+F moves the text cursor to the start of the next line or the end of the current line, whichever is closer.

Ctrl+R moves the text cursor up an entire screen.

Ctrl+C moves the text cursor down an entire screen.

Ctrl+W scrolls down one line. Cursor moves up one line unless it is already at the top (configurable).

Ctrl+Z scrolls up one line. Cursor moves down one line unless it is already at the top (configurable).

Ctrl+Q, S moves the text cursor to the beginning of the line (configurable).

Ctrl+Q, D moves the text cursor to the end of the line.

Ctrl+Q, E moves the text cursor to the top of the screen.

Ctrl+Q, X moves the text cursor to the bottom of the screen.

Ctrl+Q, R moves the text cursor to the start of the file.

Ctrl+Q, C moves the text cursor to the end of the file.

Editing commands

Ctrl+I inserts a tab character.

Ctrl+N inserts a line break.

Ctrl+G deletes the character to the right of the text cursor.

Ctrl+H deletes the character to the left of the text cursor.

Ctrl+T deletes the part of the current word to the right of the text cursor. If the cursor is not on a word, all characters to the right of the cursor up to the start of the next word are deleted.

Ctrl+Q, Y deletes all the text on the current line to the right of the text cursor.

Ctrl+Q, T deletes all the text on the current line to the left of the text cursor.

Ctrl+V toggles between insert and overwrite mode.

Email Preferences

Select Options|Preferences in the menu and click on the Email tab to configure EditPad Pro for sending email.

The screenshot shows the 'Preferences' dialog box with the 'Email' tab selected. The dialog has a green title bar and a close button (X) in the top right corner. The tabs at the top are: Editor, Cursors, Search, Open Files, Save Files, Tabs, Panels, Status Bar, Keyboard, Email (selected), Shortcuts, and System.

Double-clicking highlighted email addresses

- ☒ Compose a new email in EditPad Pro
- ☐ Launch default email client
- ☐ Launch a specific email client

Below the radio buttons is a text input field and a button with three dots (...).

Email queue

- ☒ Automatically send email messages placed into the email queue
- ☒ Close email queue automatically after all messages have been sent

Email server settings

Mail server (SMTP): Encryption: Explicit TLS Port: 587

User name: Password:

At the bottom of the dialog are four buttons: OK (with a green checkmark icon), Cancel (with a red X icon), Apply (with a yellow checkmark icon), and Help (with a blue question mark icon).

Double-Clicking Highlighted Email Addresses

When you double-click a highlighted email address in a text file, EditPad Pro can do one of three things:

- Compose a new email in EditPad Pro. This is the same as selecting File|Mail in the menu and entering the email address.

- Launch default email client. EditPad Pro will attempt to launch the application associated with the `mailto:` URL protocol on your computer.
- Launch a specific email client. Click on the (...) button to select the .exe file of the email software you want EditPad Pro to launch. You can also type in the command line directly and specify command line parameters. If you enter `%EMAIL%` on the command line, EditPad Pro will substitute that placeholder with the actual email address. If you don't enter `%EMAIL%` on the command line, EditPad Pro will launch the .exe with the email address as the sole command line parameter.

If you don't want email addresses to be highlighted and clickable at all then you need to select a different syntax coloring scheme for the file type that you're working with. The syntax coloring scheme determines which parts of your file, if any, are highlighted. You can select the scheme on the Colors and Syntax tab in the File Type Configuration.

Email Queue

If you turn on “automatically send email messages placed into the email queue”, EditPad Pro will start sending email right away when you click the Send button in the email composition panel. This way you do not have to click on the Send button in the mail queue. You should turn on this option if you have a permanent Internet connection.

Turn on “close email queue automatically after all messages have been sent” to make the mail queue disappear automatically when all emails have been sent successfully. If there was a problem sending some or all of the messages in the queue, then the queue will remain visible, so you can see which messages were not sent.

Email Server Settings

Before you can use File|Mail, you need to specify the mail server EditPad Pro should use to send email. EditPad Pro only supports the SMTP protocol, which is the de facto standard for sending email on the Internet. If you connect to the internet through a public dial-up or broadband ISP, you can use your ISP's SMTP server. Typical names for the SMTP server are `mail.isp.com`, `relay.isp.com` and `smtp.isp.com`, where `isp.com` is your internet provider's domain name. The standard port for SMTP connections is 25.

Most SMTP servers do not use encryption. Some require SSL or TLS encryption, and some allow the email client to negotiate for TLS encryption. Your email provider should tell you which encryption method you need to select. If they don't, you may be able to guess the encryption method from the port number your server uses. Port 465 is the standard port for SMTP connections encrypted with SSL. Port 587 is the standard port for SMTP connections encrypted with TLS. If you select one of these encryption methods in EditPad, the port number is changed automatically. You can still choose a different port number. If your SMTP server uses encryption on port 25, set the encryption method to “TLS, if available”.

For Gmail, set the server to `smtp.gmail.com` and use TLS encryption on port 587.

If you connect to the Internet through a corporate network, you will have to ask your network administrator which server you can use. MS Exchange and other proprietary protocols are not supported by EditPad Pro.

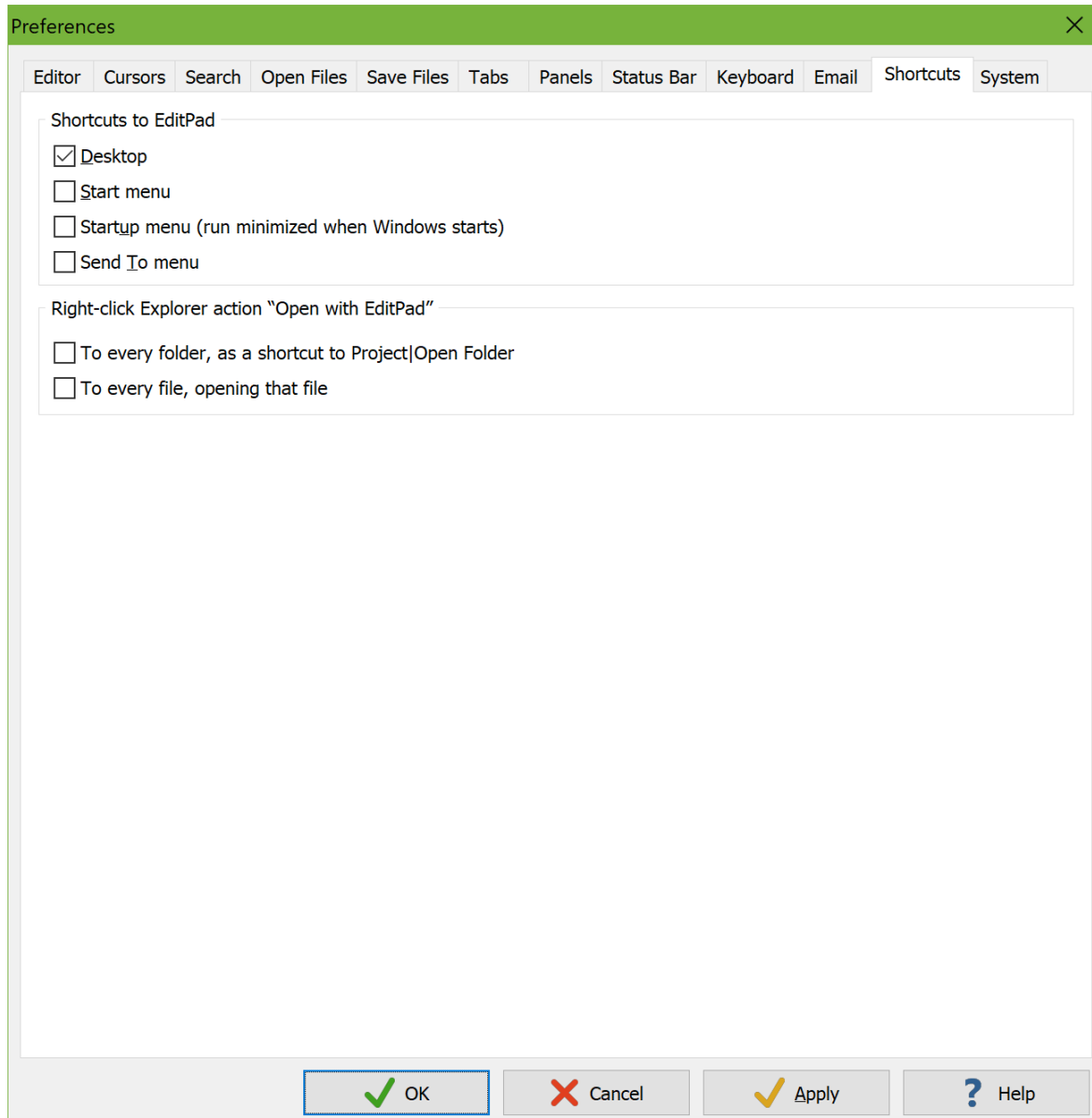
Email Server Authentication

Many SMTP servers only allow you to send email after logging in with your email username and email password. Often, the requirement is that you check your incoming email before you can send email. This is called “POP-before-SMTP authentication”. When you mark this option, you can type in your email username and email password, as well as the server from which you retrieve incoming mail. EditPad Pro supports the POP3 protocol, which is the de facto standard for receiving email on the Internet. Typical server names are mail.isp.com, pop.isp.com and pop3.isp.com. The standard port is 110 for unencrypted POP3 and TLS-encrypted POP3. The standard port is 995 for SSL-encrypted POP3. Note that EditPad Pro will not actually download any of your email. It will only log onto the server, and then disconnect, for the purpose of authenticating you so you can send email using EditPad Pro.

If your ISP requires a username and password to send email, but does not require you to check your incoming email before sending email, the ISP is using “SMTP authentication”. When you mark this option, you can type in your email username and email password that EditPad Pro can use to connect to the SMTP server.

For Gmail, select “SMTP authentication” and enter your full @gmail.com email address and your Gmail password.

Shortcuts Preferences



Create Shortcuts to EditPad

The Shortcuts page in the Preferences window makes it easy for you to create shortcut icons to EditPad. Select the various places where you want a shortcut icon. When you click OK or Apply on the Preferences dialog, all shortcuts that you've checked are created and all shortcuts that you've unchecked are deleted.

If you place a shortcut in the "Startup menu" then EditPad is run minimized when Windows starts. This allows EditPad to pop up instantly when you want to open a text file.

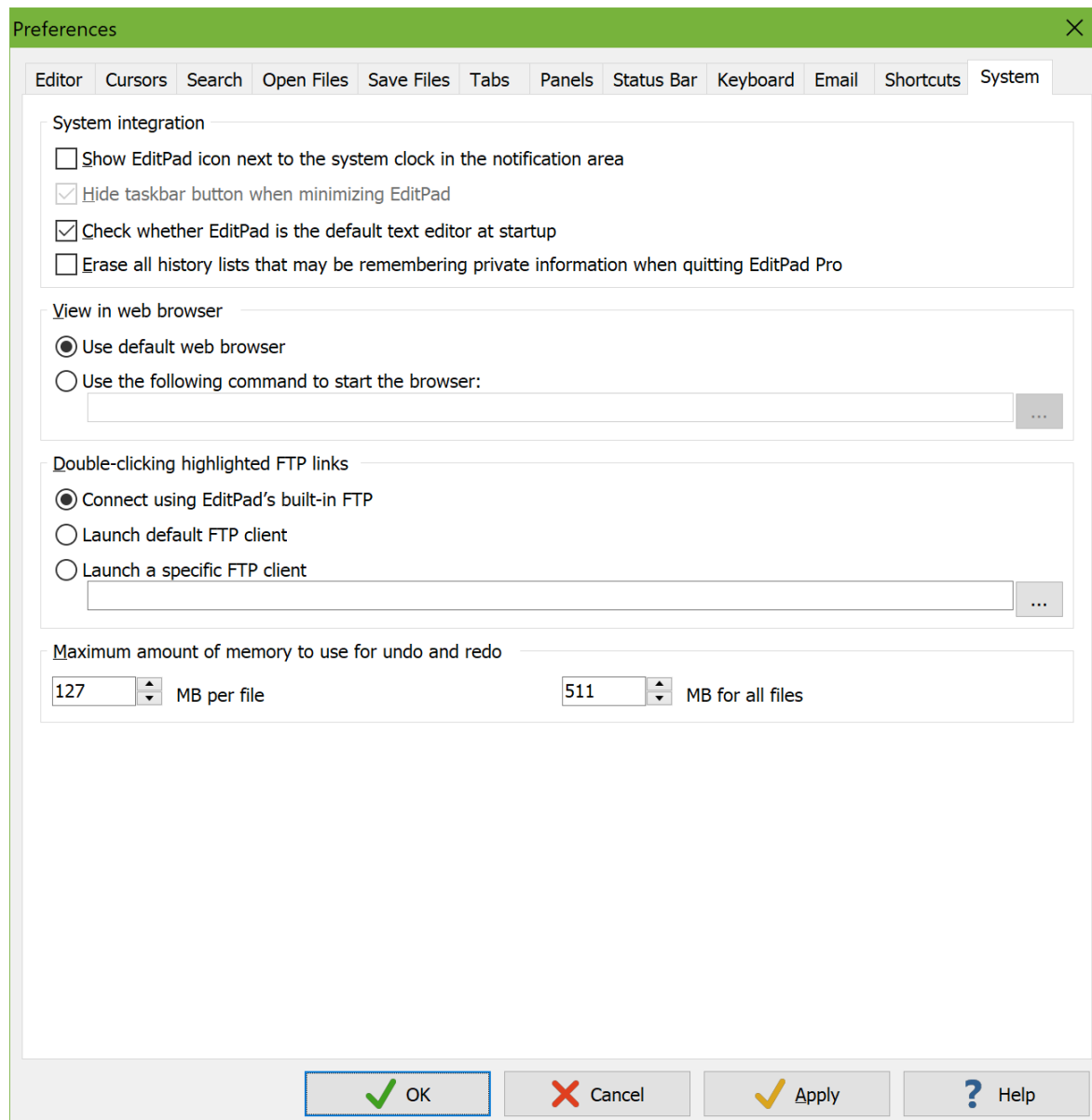
Put a shortcut in the “Send To menu” so you can quickly open any file in EditPad from within Windows Explorer by right-clicking on the file, selecting Send To from the context menu and then selecting EditPad.

Right-Click Explorer Action

EditPad can also add an “Open with EditPad” item to the context menu of every file and/or every folder in Windows Explorer. The context menu is the menu that appears when you click on an item with the right-hand mouse button.

System Preferences

On the System page in Options | Preferences, you can make some choices how EditPad Pro interacts with the rest of your computer and network.



System Integration

If “Show EditPad icon next to the system clock in the notification area” is activated, a green EditPad icon will be visible next to the system clock (this area is called the “notification area”) when EditPad is running. When you close EditPad in this situation, it will not really close but only hide itself so the system tray icon remains visible. You can then make EditPad quickly reappear by clicking on the system tray icon. You can right-click on it to quickly start with a new file or open an existing one. To completely shut down EditPad, select File | Exit in the menu.

Turn off this option to make EditPad behave like a regular Windows program.

When the “Show EditPad icon next to the system clock in the notification area” option is activated, you can choose if you want EditPad’s button on the taskbar to hide or not when you minimize EditPad by marking or clearing “Hide taskbar button when minimizing EditPad”. Note that the taskbar button will always hide when you close EditPad.

Turn on “check if EditPad is the default text editor at startup” if you want EditPad to check whether it is still associated with .txt files each time you start it. If this option is on, and EditPad detects it is no longer associated with .txt files, it will ask you whether you want EditPad to associate itself with .txt files or not.

Turn on “erase all history lists that may be remembering private information when quitting EditPad Pro” to make EditPad Pro forget the lists of recent items kept by these commands:

File|Open File|Mail Project|Open Project Project|Open Folder Project|Import File Listing
Search|History Search|Find on Disk Explorer Panel|Set Home Folder Explorer Panel|Filter FTP
Panel|Connect FTP Panel|Filter

View in Web Browser

If the View|Browser doesn’t properly detect your web browser, you can specify a specific application to be used as the web browser. Click the (...) button to browse for the .exe file of the application you want to use. You can also manually enter a command line. If you include %URL% as part of the command line parameters, that placeholder will be substituted with an http:// or file:// URL to the file to be opened. If you don’t enter %URL% on the command line, EditPad Pro will launch the .exe with the URL as the sole command line parameter.

Double-Clicking Highlighted FTP Links

EditPad Lite will always open FTP links in your computer’s default FTP client. EditPad Pro provides three choices. The default is to use EditPad’s built-in FTP. This is the most comfortable option to edit files stored on FTP servers. The “launch default FTP client” runs whichever application is configured on your computer to handle ftp: URLs.

Finally, you can specify a specific application to be used as the FTP client. Click the (...) button to browse for the .exe file of the application you want to use. You can also manually enter a command line. If you include %URL% as part of the command line parameters, that placeholder will be substituted with an ftp:// URL to the file to be opened. If you don’t enter %URL% on the command line, EditPad Pro will launch the .exe with the URL as the sole command line parameter.

Maximum Amount of Memory to Use for Undo and Redo

As explained in the help topic for the Edit|Undo command, EditPad normally keeps an undo history for all changes you’ve made to all files that you have open in EditPad since you opened them. During very heavy editing sessions, this may cause the undo history to use too much memory. As a safeguard, EditPad will automatically discard the oldest changes from the undo history when it grows too large. In EditPad Lite these

limits are automatically set based on the amount of RAM your PC has. In EditPad Pro, you can configure the limits yourself.

The limit per file limits the undo history for each individual file. If the undo history for a file grows beyond this point, the oldest changes for that file are discarded from its undo history. EditPad does this even if there are older changes in other files, and even when the total undo history size for all files has not yet exceeded the limit for all files.

The limit for all files limits the combined size of the undo history for all open files. If the total undo history size grows beyond this point, the oldest changes are discarded from the undo history. EditPad does this regardless of which file(s) the oldest changes were made in, and regardless of how much memory the undo history for those files uses.

Set the limit for all files to allow enough free memory for the files themselves that you edit in EditPad, as well as for Windows and for all the other applications you're running. Set the limit per file lower than the limit for all files to make sure that making lots of changes to one file does not clear out the undo histories for all the other files you have open in EditPad.

16. View Menu

View | Clip Collection

Select Clip Collection from the View menu to open EditPad Pro's Clip Collection panel. There you can store common blocks of text and templates for quick repeated reuse. When editing complex documents, move chunks of text into the collection for temporary storage. The Clip Collection is also a great place to jot down notes and ideas, and to keep important information at your fingertips.

AceText, a product available separately, is specifically designed to enable you to very flexibly work with large numbers of text snippets. If you have installed AceText, EditPad Pro will use AceText's Clip Collection functionality, and display the collection that is active in AceText. EditPad Pro and AceText will automatically keep the collection synchronized when you edit it in AceText or EditPad Pro.

If you do not have AceText, EditPad Pro will manage the collection of clips by itself. You may want to check out the free evaluation version of AceText. AceText offers a lot of extra functionality. Most importantly, AceText enables you to use your clip collections in combination with applications other than EditPad Pro. AceText cooperates with almost all Windows software.

Adding Text to The Collection

To add text to the collection, simply select it in the editor and click the "New Clip" button in the toolbar above the Clip Collection. You can also drag and drop the selection from the editor into the collection. Drag and drop clips inside the collection to rearrange them.

You can edit a clip's label or view and edit the text it holds by clicking the Edit Clip button. If you use AceText, this will activate the AceText Editor. If not, you can edit the clip directly in EditPad Pro.

If you want to add many clips to a collection, organize them in a folder to make it easier to find the right clip later. Simply click the New Folder button to create a new folder. Add clips to the folder by selecting the folder before clicking the New Clip button. You can also drag and drop text or clips onto the folder.

Using Clips from The Collection

To insert the contents of a clip into the file you are editing, simply double-click it or drag-and-drop it onto the spot where you want to insert it. If the clip holds before and after text, it will be placed around the text you selected in the editor. If you did not select any text, the before and after texts are inserted right next to each other, and the text cursor is placed between them. Before and after text makes it very easy to insert common text blocks that have an opening and closing part, such as HTML or XML tags.

Opening and Saving Collections

If you are not using AceText, EditPad Pro will automatically save Clip Collections. This way, you never have to worry about losing any clips. The Save button allows you to save the collection under a different name. To

open a collection later, click the Open button and select the file, or click the downward pointing arrow next to the Open button to select a collection you recently worked with.

To make Clip Collections even more transparent, you can assign a default collection to each file type in the file type configuration. That collection is then automatically opened whenever you switch to a file of that type. E.g. you can assign a collection with HTML tags to the HTML file type. That way, the clip collection will always be ready for inserting HTML tags whenever you edit an HTML file.

EditPad Pro can also import TextPad Collection Library files (*.tcl files). After clicking the Open button on the Clip Collection panel's toolbar, select the "TextPad Collection Libraries" file type in the drop-down list at the bottom of the file selection screen. EditPad Pro cannot save TextPad Collection Library files.

Sharing Collections

Click the Share Collections button to download Clip Collections shared by others, or to upload your own collections. A new window will pop up. EditPad Pro will connect to the Internet immediately to download the list of available collections. If your computer can only access the internet via a proxy server, click the Settings button in the lower left corner of the screen to enter your proxy server's settings.

The Download tab shows all available collections, including the ones you uploaded. The list shows the collection's title, the file name used by the author, how many clips it contains, who shared it, when it was first shared and when it was last updated. Click on a collection to get a more detailed description, and the author's name and email address (if provided by the author). Click the Download button to download the selected scheme.

The Share tab shows the collections that you previously uploaded. EditPad Pro automatically uses your EditPad Pro license to identify you. You can check your license details in the about box. To share a collection, select its file, and enter a title and your name. You can also enter your email address and web site URL if you want. In the Description box, explain what your collection is about. Click the Upload button to upload it. If you select a collection you previously uploaded, EditPad Pro will fill out all the fields with what you entered when you uploaded that collection. This way you can easily update a previously shared collection, or use the same details for another one. If you upload a collection with the same file name as a collection you already shared, the newly uploaded collection will replace the old one. If you upload a collection with another file name, it will be added as a new collection, even if you give it the same title or description as another one. Click the Delete button to stop sharing a collection.

EditPad Pro downloads and uploads in the background. The Progress tab shows all collections in the queue, and recently completed transfers. If an error occurs, EditPad Pro will not pop up a message box. The Progress tab will indicate the error.

Edit Clip

To edit a clip in the Clip Collection, select the clip, and click the Edit Clip button in the Clip Collection toolbar.

A clip consists of the following parts:

Label: The label that identifies the clip in the collection. If you do not type in a label, the start of the text itself is used as the label.

Kind: EditPad Pro supports different kinds of clips. “Plain text” clips hold ordinary text. A word, a sentence, several paragraphs, or a complete text document of any size. “Rectangular text block” clips hold a rectangular selection. “Binary data” clips hold data in hexadecimal format. If you create a clip while EditPad Pro is in hexadecimal mode, the clip will hold “binary data”.

“Before and after text” are a special, very useful kind of clip. You cannot create such a clip by dragging and dropping some text onto the Clip Collection. You can only create them by editing a clip, or by clicking the New Clip button when no text is selected. These clips are very useful to insert frequently used bits of text that consist of an opening and a closing part, such as HTML tags. The before part is placed before the cursor or selection, and the after part is placed behind it.

Indent: Turn on the indent option for “plain text” and “before and after text” clips, if the clip consists of multiple lines, and you want the second and subsequent lines to be lined up with the first line. E.g. source code is often indented to indicate block structure. Turn on the “indent” option for clips holding source code snippets, and they will be properly indented when you use them.

Syntax: The syntax coloring scheme for the clip. Only the Edit Clip dialog uses this syntax coloring scheme. It does not affect how the clip is pasted into the file you’re editing.

Text: The text stored by the clip. You can use all text editing keyboard shortcuts and mouse actions to edit the text of a clip.

URL: Website address or full path to a file on your computer or Windows network. Double-clicking the URL opens the URL according to your settings in the System Preferences. Double-clicking a file path opens the file in EditPad if the path matches the file mask of one of your file types other than the “unspecified file type” and the file type that it matches uses the option “never open files of this type in hexadecimal mode”. Otherwise, double-clicking the file path opens the file in the application that is associated with it in Windows or executes the file if it is executable.

View | Character Map

Select Character Map in the view menu to show a grid with all the characters available in a particular text encoding. Read the help topic for Convert | Text Encoding to learn more about text encodings. By default, the character map shows the characters supported by the encoding used by the active file. You can select a different encoding in the drop-down list on the character map’s toolbar. Selecting an encoding in the character map only affects the character map. It does not affect the file you’re editing.

If the selected encoding is an 8-bit encoding, the character map displays all characters in the encoding except non-printable control characters. The Windows, Mac, and DOS encodings, except those for Far East languages, are encodings with 224 characters and 32 control characters. The ISO-8859 and EBCDIC encodings are encodings with 192 characters and 64 control characters. Some character maps have holes in them indicated by crossed-out grid cells. These indicate positions in the encoding that do not define any character.

The various Unicode transformations define thousands of characters. So do the Windows, Mac, and EUC encodings for Korean, Japanese and Chinese. EditPad’s character map displays the characters in the order

given to them by the Unicode standard, even for the Windows, Mac, and EUC encodings. To make it easier for you to find the character you want, you can filter the map to show only characters of a certain type. You can filter by Unicode category, Unicode block, and/or Unicode script. To enable a filter, simply select a choice from the drop-down list. To disable it, select (all categories), (all blocks), or (all scripts) at the top of the list. If you enable multiple filters, all of them apply at the same time. If you select the “decimal digits” category and the Thai block, the map will show Thai digits only (or nothing at all if the encoding doesn’t support Thai).

If certain characters appear as squares, that means the character cannot be displayed using the current font. You can select a different font with Options|Font. Particularly when showing all Unicode characters, you may see a lot of squares. While Unicode tries to define characters for all human languages and scripts, most fonts only support one script, or one group of closely related scripts. Microsoft supplies a font called “Arial Unicode” with recent versions of Windows. This font can display nearly all Unicode characters.

When you hover the mouse over a character in the map, its decimal and hexadecimal number in the encoding will be shown in the status bar. Double-click on a character to insert it into the file.

Because you can make the character map show any encoding, it is possible that the character map shows characters that cannot be represented by the encoding used by the active file. If you try to insert an unsupported character, it will show up as a question mark in the file, and EditPad will show a warning that the character could not be inserted. The question mark is not a placeholder but a permanent question mark. Press Backspace or use Edit|Undo to remove the question mark. To insert the proper character, change the file’s encoding first, and then insert the character again.

To change the file’s encoding, click the button next to the drop-down list with the encodings. A menu will pop up. The Text Encoding item at the bottom is identical to the Text Encoding item in the Convert menu. It pops up a window that allows you to change the encoding. If you have already selected the encoding you want in the drop-down list on the character map, you can use the Display File with Encoding item or the Convert File to Encoding item to change the file’s encoding without using the Convert|Text Encoding popup window. The Display File with Encoding item corresponds to the “interpret the data as being encoded with another character set” choice in the Text Encoding window. The Convert File to Encoding item corresponds to the “encode the original data with another character set” choice. Both choices are explained in detail in the Convert|Text Encoding help topic.

The left-hand side of the character map toolbar has several buttons to insert various representations of the character you selected in the character map. The leftmost button inserts the character itself. The next two buttons insert the character’s code page index in decimal and hexadecimal notation. These buttons are only available when the character map shows an 8-bit code page. They insert the character’s position in the code page being shown by the character map. This is not necessarily the character’s index in the code page being used by the file you’re editing.

The remaining three buttons insert the Unicode code point of the selected character in three different notations: a Unicode escape, a decimal numeric character reference, or a hexadecimal numeric character reference. They always insert the Unicode code point, regardless of the encoding used by the file or the encoding shown in the character map. For example, if you have the euro symbol selected in the character map, these buttons insert `\u20AC`, `€`, and `€`; because the euro symbol occupies code point U+20AC in the Unicode standard.

The edit box at the right-hand side of the character map toolbar allows you to look up characters in the character map. To look up a character, type in the character itself or its representation, and press Enter. The “look up” box supports various notations. If you type in a single character, that character is selected in the

character map. If you type in a Unicode escape such as `\u20AC` or a numeric character reference such as `€` or `€` then the character represented by that Unicode code point is selected in the character map. This works even when the character map is showing a non-Unicode encoding. You can also type in just a decimal number such as `169` or just a hexadecimal number such as `A9`. If you want `80` to be taken as a hexadecimal number, type in `0x80` or `80h`. How the number is interpreted depends on whether the character map is displaying an 8-bit encoding or not. If the encoding is 8-bit, the drop-down lists with Unicode categories, blocks, and scripts will be invisible. Then the number is interpreted as an index to that 8-bit code page. In the Windows code pages, for example, `0x80` is the euro symbol. If the character map encoding is not 8-bit, meaning the three Unicode drop-down lists are visible, then the number is taken as a Unicode code point, even if the encoding is not Unicode. If you have the multi-byte Windows 932 code page selected, `20AC` is interpreted as the euro symbol, while `0x80` is interpreted as the control character represented by code point `U+0080`, even though the euro symbol is represented by the single byte `0x80` in code page 932.

View | Hexadecimal

Click on the Hexadecimal item in the View menu or press `Ctrl+H` to switch between text and hexadecimal editing. The undo and redo lists are cleared whenever you switch.

In hexadecimal mode, the editing area is split into three zones. The left zone is a column with byte offsets. It indicates the offset relative to the start of the file of the first byte on each row in the hex editor. The first file in the byte has offset zero.

The middle area shows all the bytes in the file represented as a hexadecimal number. You can type in the numbers 0 through 9 and the letters A through F to insert bytes. To change some bytes without changing the length of the file, press the Insert key on the keyboard to toggle between Insert and Overwrite mode. In Overwrite mode, the bytes you type in will replace the bytes already present in the file.

The right-hand column shows the textual representation of each byte. It translates bytes into characters using the file type's default text encoding or the non-Unicode encoding if either of those is a single byte encoding. Otherwise, it uses the encoding you selected for the ASCII section in the Editor Preferences. You can select a different 8-bit encoding for the active file using `Convert | Text Encoding`. You can type characters into the text column. EditPad Pro uses the same encoding to translate the characters into bytes.

If you want to see only the hexadecimal representation or only the textual representation, select Hexadecimal Only or ASCII only in the submenu of the Hexadecimal item in the View menu. Select Hexadecimal and ASCII to restore the default. Whichever mode you last selected in the submenu is used as the default next time you click the `View | Hexadecimal` item directly.

If you have used `View | Split Editor` to split the editor in two, then you can select `Split Hexadecimal and ASCII` in the `View | Hexadecimal` submenu to have the first half of the editor show only the hexadecimal representation, and the second half of the editor only the ASCII representation. If you didn't split the editor, then the editor shows both the hexadecimal and ASCII representations.

By default, the hexadecimal view groups bytes in blocks of 8 bytes and shows as many blocks of 8 bytes on one row as will fit within the width of EditPad's window. If your binary files use a specific record size, it may be more convenient to display one record on one row, even if that requires horizontal scrolling or leaves blank space. Use `Options | Record Size` to specify the number of bytes EditPad Pro should display per row in hexadecimal mode.

If your binary files store integer or floating point numbers that are larger than one byte, you can easily edit those numbers in decimal representation using EditPad Pro's Byte Value Editor while you're in hexadecimal mode.

Working With Text Inside Binary Files

If your binary files store text using an 8-bit encoding, you can work with that directly in the right-hand ASCII section of the hex editor. First use **Convert|Text Encoding** to select the proper encoding if it is not the file type's default. In hexadecimal mode, **Convert|Text Encoding** never changes the bytes that are already in your file. It only changes how the ASCII section displays those bytes and how characters you type into the ASCII section are translated into bytes.

If you copy and paste between different files in hexadecimal mode in EditPad Pro then you will always paste the exact bytes that you copied, regardless of whether you are using the same or different encodings for the ASCII section of those files. If you copy from a file in text mode and paste into another file in hexadecimal mode, then the text you copied is first converted from the text file's encoding to the 8-bit encoding you're using for the hexadecimal file. The resulting bytes are then pasted.

If you need to work with text inside a binary file that is stored as Unicode or using a multi-byte code page, then use **File|New** to open an additional tab to use as a scratch space. To interpret bytes in the binary file as text, copy them from the binary file. Switch the scratch file to hexadecimal mode and then paste. You'll paste the exact bytes. Now switch the scratch file to text mode. Use **Convert|Text Encoding** with the "interpret" option and the encoding you want to interpret the bytes with. This does not change the bytes in the scratch file. It only makes EditPad display them using the selected encoding. To insert new text into the binary file, switch the scratch file to text mode, set its encoding if needed, and enter your text. Then switch the scratch file to hex mode. This gives you the bytes that represent your text in the encoding that you were using in text mode. It doesn't matter if the ASCII section shows garbage. It may be interpreting those same bytes with a different 8-bit encoding. Simply copy and paste the bytes into your actual file.

If you are using Unicode for your scratch file (or any file) and its file type specifies that a byte order marker should be written, then the BOM is added at the start of the file in hexadecimal mode. If you don't want the BOM then you may want to use a file type that doesn't write one for your scratch file.

In hexadecimal mode, the Search and Replace boxes work exactly the same as the main editor. They use the same encoding for their ASCII sections. So you can search for text using the same 8-bit encoding inside binary files by entering it into the ASCII section of the Search box. You can search for text using other encodings by first preparing it in a scratch file as explained above and then pasting into the Search box.

View|Byte Value Editor

When editing a file in hexadecimal mode, you can open EditPad Pro's Byte Value Editor. With the Byte Value Editor, you can edit the byte that the text cursor points to in the hexadecimal editor, and up to 7 bytes that follow it. Instead of entering hexadecimal values directly in the hex editor, you can enter decimal numbers in various formats. When you enter a number in one of the formats, EditPad Pro automatically converts the decimal number into 1 to 8 bytes. EditPad Pro uses those bytes to replace the byte pointed to by the cursor and as many following bytes as needed.

The Byte Value Editor works with the main file editing area, as well as with the search and replace boxes. The edit box that was the last one to have keyboard focus is the one the Byte Value Editor is connected to.

The first row of values are unsigned integers of 1, 2, 3, 4, 6 or 8 bytes. Values are directly converted between decimal and hexadecimal.

The second row of values are signed integers of the same sizes. Hexadecimal values are negated using two's complement, like modern microprocessors do. E.g. if you enter -1 in any of the signed integer boxes, all bytes will be set to FF.

The binary box allows you to enter up to 8 bits to edit the current byte as a binary number. This is useful if the byte represents a bit field.

Three floating point boxes allow you to enter floating point numbers of various precisions. The 4-byte floating point number defines a 32-bit single precision floating point number following the IEEE standard. The 8-byte floating point number defines a 64-bit double precision floating point number, also following the IEEE standard. The single precision and double precision floating point numbers are identical to the single and double floating point types in most modern programming languages. You can type in floating point numbers using scientific notation. E.g. 1.5e6 is one and a half million. Positive infinity, negative infinity and “not a number” are represented by Inf, -Inf and NaN respectively.

The 6-byte floating point number is a bit of an odd-ball. It defines a 48-bit floating point number which corresponds with the “Real” type in Turbo Pascal and other Pascal variants. In Delphi, the “Real” type is actually a 64-bit double precision floating point number. The “Real48” type is the Delphi equivalent of the Real type in Turbo Pascal.

The little and big endian options switch the order of the bytes. The decimal value 300 is 0x12C in hexadecimal. In little endian format, the two bytes are ordered 2C 01, starting with the least significant byte. In big endian, the two bytes are ordered 01 2C, starting with the most significant byte. You can see the difference if you type 300 in the “word” field in EditPad Pro. The Intel and AMD processors that Windows runs on all use the little endian format natively. The PowerPC processors that the Apple Macintosh used to run on use the big endian format natively. Recent Macs use little endian Intel processors.

View | Split Editor

Click on the Split Editor item in the View menu to split the editor into two, giving you two independent views of the active file. In the submenu of the Split Editor item you can choose whether the editor should be split horizontally or vertically, or whether the second editor should be placed in a floating window. The floating window is useful if your PC has multiple monitors and you want the second view to be on another monitor. Clicking the Split Editor item directly toggles between not splitting and splitting using the method you last selected in the submenu.

You can switch keyboard focus between the two halves without altering their selections or cursor positions by pressing the keyboard shortcut for the View | Focus Editor command. F9 is the default.

When you switch between files while the editor is split, the editor remains split in the same way, and both halves of the editor will show the file that you switched to. EditPad cannot show more than one file at the same time in a single EditPad instance. Use View | New Editor to start a second instance.

By default, the two halves of the split editor scroll independently. Scrolling one half of the editor does not scroll the other half. If you want both halves to respond to your scrolling simultaneously, use the `View|Joint Scrolling` command.

Both editor halves maintain their own text cursor position and their own selection. Selecting text in one half does not select it in the other half. If you make a different selection in both halves, you can use `Block|Swap Selections` to swap the two blocks of text that you selected. All other commands that work on selections only work on the selection in the editor half that has keyboard focus, or that had keyboard focus most recently if neither half does.

Both editor halves do show the same file. Any changes you make to the file in one half automatically show up in the other half. Only the selections, cursor positions, and scrolling positions are independent.

View|Joint Scrolling

If you split the editor, then by default the two halves of the split editor scroll independently. This makes it easy to copy and paste between different parts of the same file, or to look at one part of the file while typing into another part of the file.

If the two parts of the file you are working consist of (roughly) the same number of lines but are too long to fit on the screen, select `Joint with Fixed Gap` in the `Joint Scrolling` submenu of the `View` menu. From then on, scrolling one half of the editor automatically scrolls the other half an equal number of lines. EditPad Pro remembers the gap between the line numbers of the first line that was visible in each half of the editor when you selected the `Joint with Fixed Gap` joint scrolling option. If you scroll to the top or bottom of the file, EditPad Pro may not be able to maintain the gap. But it will still remember the gap. If you scroll back to the middle of the file, the scrolling gap between the two halves will be restored.

If you are concentrating on only one part of a file, and that part is longer than fits on the screen, you can split the editor vertically and then select `Joint without Gap` in the `Joint Scrolling` submenu. Then the right-hand half of the editor will show the lines immediately following the lines shown in the left-hand half. Essentially you get two newspaper columns of your file. This mode is particularly useful when you have a wide screen monitor and the lines in your file are short, or you prefer your lines to be word wrapped to a short length, making them easier to read and edit. This view is maintained when you scroll either half of the editor. If you scroll the left half to the bottom or the right half to the top, then both halves will show the bottom or top of the file. Once you scroll back to the middle of the file, the right half will again show the lines immediately following the left half.

If you want to turn off joint scrolling, either click the `Joint Scrolling` item in the `View` menu directly, or select `Independent` in the submenu. Clicking the item directly toggles between independent scrolling and the previously selected joint scrolling mode.

The `Joint with Fixed Gap`, `Joint without Gap`, and `Independent` options only affect the active file. When you switch between files, the view remains split in the same way, but joint scrolling reverts to what it was for the file you're switching to. If you want to use the same scrolling method for all files, select `Joint without Gap for All Files` or `Independent for All Files` in the `Joint Scrolling` submenu. Then the scrolling method won't change when you switch between files. The `Joint with Fixed Gap` option does not have an "all files" equivalent. The scrolling gap is always remembered independently for each file.

View | New Editor

When you try to start a new instance of EditPad through the Windows Start menu or a desktop shortcut, EditPad will activate the instance that was already running instead of starting a new instance. Any files that the new instance should have opened will be opened by the existing instance. Since EditPad is designed to make it easy to work with large numbers of files, you can easily work with all your documents in one location, without cluttering your desktop.

If you do want to have multiple instances of EditPad running, simply select the New Editor item in the View menu. This can be useful if you want to view two files side by side.

In the submenu of the New Editor item, you can select how the two instance should be positioned relative to each other. Clicking the New Editor item directly opens the instance using the option you selected last time.

- On top: Opens the new instance at the same size and position as the current instance.
- Horizontal split: The new instance takes up the bottom half of the area occupied by the current instance. The current instance shrinks to take up the top half of its area.
- Vertical split: The new instance takes up the right-hand half of the area occupied by the current instance. The current instance shrinks to take up the left-hand half of its area.
- Cascade: Opens the new instance at the same size as the current instance, but positioned a bit lower and a bit more to the right.
- Second monitor: Opens the new instance at the same size as the current instance, but positioned on the next monitor. If you have more than two monitors, repeating this command as many times as you have monitors minus one (to account for the running instance) will put an EditPad instance on each of your monitors.

The Open Active File item in the New Editor submenu has no immediate effect other than toggling the check mark next to the item. When checked, starting a new editor makes the new instance open the file that is active in the current instance.

The Horizontal Split and Vertical Split options change the size of the current instance. If you don't resize the instance that was split after you launched the new instance, then closing the new instance automatically makes the split instance resize itself again to take back the space it occupied before it was split in half by the new instance.

If you want to start a second EditPad instance from a shortcut, script or application, you can do so with the `/newinstance` command line parameter. Then the new instance opens all the files you specify on the command line. It will not affect the size or position of any running instances, or set its own position based on any running instances. You can use an additional parameter such as `/brl100t200r300b400` to set the new instance's bounding rectangle to left 100, top 200, right 300 and bottom 400, counting pixels from the top left corner of the screen.

View | Other Editor Joint Scrolling

You can use the Other Editor Joint Scrolling command in the View menu whenever you have multiple instances of EditPad Pro running. It works whether you started them with the View | New Editor menu item or via the `/newinstance` command line parameter. Clicking the Other Editor Joint Scrolling item has no immediate effect other than putting a box around its icon to indicate it is active.

If you turn on Other Editor Joint Scrolling in the EditPad Pro instance that has keyboard focus, then it will broadcast a scrolling message to all other EditPad Pro instances whenever you scroll the active file. If Other Editor Joint Scrolling is turned on in any other running EditPad Pro instance, then those instances will scroll their active files to the same line whenever they receive the scrolling messages. The result is that if you turn on Other Editor Joint Scrolling in two or more EditPad Pro instances, then all those instances will scroll simultaneously. This makes it easy to compare two or more files side by side.

View | Browser

Select Browser in the View menu to open the active file in a web browser. If the file is modified, EditPad will save it first.

If this feature does not work, either EditPad is unable to detect your computer's default web browser, or you've incorrectly configured the browser setting in the System Preferences.

If the file you're editing was opened via FTP then EditPad Pro uses the "translate FTP folders into HTTP URLs" setting from the FTP connection dialog to determine the URL that represents the file on your server. This URL is then passed to your browser.

If you want to use multiple browsers you can add them to the Tools menu in EditPad Pro. If you're using another application to upload your file to a server, or if you have the server software running on your own PC, then you can add a tool with an URL as its command line.

View | Close Panels

Press the Esc key on the keyboard to close the active side panels. The active panel is the one that has keyboard focus. When the panel is closed, keyboard focus is automatically given to the main editor.

If none of the open panels has keyboard focus, all of them are closed. To quickly close all panels at any time, simply press the Esc key twice.

View | Focus Editor

The keyboard shortcut associated with the Focus Editor item in the View menu is a quick way to set the keyboard focus to the main editor. Doing so makes the text cursor (blinking vertical bar) visible in the main editor allowing you to continue editing the file. The default keyboard shortcut is F9.

Using the Focus Editor keyboard shortcut allows you to quickly switch back and forth between the main editor and other panels. E.g. while working with the search and replace panel, you can press Tab on the keyboard to cycle through the various controls on the search and replace panel. To switch to the main editor, press F9. To switch back to the search and replace panel, press Ctrl+F.

If you have split the editor, then you can use this keyboard shortcut to switch keyboard focus between the two halves of the editor when one of them already has focus. When another part of EditPad has focus, this keyboard shortcut moves focus back to the editor half that had focus most recently.

View | Dark Theme

Toggles EditPad's user interface (toolbars, menus, dialog boxes, etc.) between the default Windows theme (which is a light theme) and EditPad's own dark theme.

If you regularly switch between themes you should select color palettes that have companion palettes on the Colors and Syntax page in the file type configuration. Then switching between themes also switches the file types between light and dark palettes. Switching between themes also swaps the text and background colors on the Panels page in the Preferences to match the theme if one color is very dark and the other color is very light.

View | Restore Default Layout

The Restore Default Layout item in the View menu restores all side panels, all toolbars, all context menus, and the main menu to their default configurations and locations. All menu and toolbar customizations you made are undone.

View | Custom Layouts

If you have spent time rearranging EditPad's toolbars and/or side panels, or if you have customized the toolbars or menus, you should use the Custom Layouts item in the View menu to save those settings. Simply select Save Layouts and type in a name. That name then appears under the Custom Layouts submenu. If you click on the name in the Custom Layouts submenu, EditPad restores all side panels, all toolbars, all context menus, and the main menu to the configurations and locations they had when you saved the layout.

You can save as many layouts as you like. If you want to update a layout, save it again under the same name. To delete a layout, select the Delete Layout command and then select the layout you want to delete.

How to Rearrange EditPad's Panels

To move a panel, use the mouse to drag and drop its caption bar (for a panel docked to the side, or a floating panel) or its tab (for a tabbed panel). While you drag the panel, squares appear at the four edges of EditPad's window. While dragging over another panel, five squares appear in the center of that panel. Drop the panel onto one of the four squares at the edges of EditPad's window to dock the panel to that edge. Drop the panel onto one of the four outer squares in the center of another panel to dock the dragged panel to one of the four sides of the panel you're dropping it onto. Drop the panel on the center square of another panel to arrange the two panels inside a tabbed container.

To make a panel float freely, drag it away from EditPad or simply double-click its caption or tab. Floating a panel is very useful if your computer has more than one monitor. Move the floating panel to your second monitor to take full advantage of your multi-monitor system. If you drag a second panel onto the floating panel, you can dock both panels together in a single floating container. This way you can conveniently display several panels on the second monitor.

Panels that are docked to the edge of EditPad's window can be pinned to that side by clicking the pin button on the panel's caption bar. You can also pin a panel that is inside a tabbed container that is docked to the

edge. Doing so pins all panels in that tab container. Pinned panels appear as a small strip showing only the panel's icon and caption. When you hover the mouse over the panel's icon or caption on that strip, the panel slides into view. It remains visible while the mouse pointer is over the panel. When the mouse pointer leaves the panel it slides out of view again. Click the pin button again to make the panel permanently visible again. You cannot drag a panel to a different location while it is pinned (in auto-hide mode).

17. View | Files Panel

The Files Panel is a panel that sits docked at the left side of EditPad's window. You can make it visible by selecting Files Panel in the View menu. You can dock the panel elsewhere by dragging its caption bar or its tab.

The Files Panel shows a collapsible tree of the projects and files you have open in EditPad Pro. All project nodes in the tree sit under the EditPad Workspace root node. Each project node contains nodes for the files in the project. Depending on the options you enabled under the View button (see below), the file nodes may be grouped under folder and file type nodes under project nodes too.

Files that have unsaved changes are highlighted in bold. Closed files that are still part of managed projects are dimmed if you choose to show them at all. Outside files in managed projects are shown in italics.

The Files Panel offers several commands that you can execute via the panel's toolbar, or via the right-click menu. The key difference between these commands and their equivalents that you find in EditPad Pro's main menu is that these commands work on the files and projects that you have selected in the Files Panel, rather than the active project or active file.

Edit

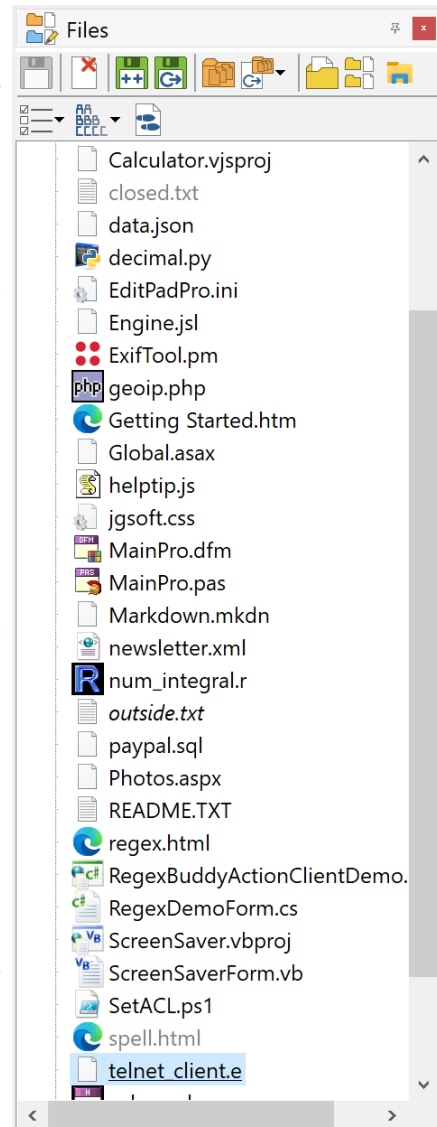
Double-click on a file node to activate the file for editing. If the file's project isn't active yet, it will be activated too. Double-click on a project node to activate the project, and the file last active in that project.

Save

If you've selected one or more files, those files will be saved just like File | Save would do. If you've selected a project, that project is saved like Project | Save Project As would do. Saving a project does not save the files in that project. If you've selected a folder, all files listed under that folder in the Files Panel are saved.

Close

Closes the selected files or projects like File | Close and Project | Close Project would do. Closing files does not remove them from managed projects. If you've selected a folder, all the files listed under that folder are closed, and the folder will disappear from the Files Panel.



Remove from Project

Closes the selected files and removes them from managed projects like Project|Remove From Project would do. If you've selected a folder, all the files listed under that folder are removed from the project, and the folder will disappear from the Files Panel.

If a project is selected, the project is simply closed as Project|Close Project would do.

Save Copy As

If you selected one file or one project, you will be asked in which folder and under which name you want to save the copy of the file or project. This works just like File|Save Copy As and Project|Save Copy of Project As do for the current file or project.

If you selected several files, you will be asked for a folder to copy the files into. All selected files will be copied into that folder under the names they already have. This way you can quickly copy a bunch of files.

If you selected several projects, you can choose whether to copy the .epp files into which the projects themselves are saved, or whether you want to copy the document files held by the projects, or both. All those files will be copied into the folder you select under the names they already have.

Rename / Move

If you selected one file or one project, you will be asked for a new name and folder for that file or project, just like the File|Rename / Move and Project|Rename / Move Project do for the active file or project.

If you selected several files, you will be asked for a folder to move the files into. All selected files will be moved into that folder, without being renamed. This way you can quickly move a bunch of files to a different location.

If you selected several projects, you can choose whether to move the .epp files into which the projects themselves are saved, or whether you want to move the document files held by the projects, or both. All those files will be moved into the folder you select under the names they already have.

New Project

Creates a new untitled project. If one or more files are selected, they will be moved into the newly created project. This is a quick way to split a project with many files into multiple projects with a manageable number of files. If no files are selected, the new project will have one untitled blank file.

Move to Project

Removes the selected files from the project(s) they're in, and adds them to the project that you select in the Move to Project submenu. You need to open the project you want to move the files into before you can move them. The submenu only lists open projects.

This command does not move the files on disk. It only changes which project the files are part of. If you move closed files or outside files from one managed project to another, those files remain closed files or outside files in the project you've moved them to.

Add Outside Files

If you have one or more outside files selected in a managed project, those files are made part of the project as open files.

Open Files from This Folder

Opens the File|Open dialog showing the folder that you selected in the Files Panel, or the folder containing the file or project that you selected in the Files Panel. Other than the default folder for the file selection dialog, there is no difference between this command and File|Open.

Explore in EditPad

Opens the Explorer Panel and selects the folder that you selected in the Files Panel, or the folder containing the file or project that you selected in the Files Panel.

Explore in Windows Explorer

Launches Windows Explorer showing the folder that you selected in the Files Panel. If you selected a file or a project, then Windows Explorer is launched showing the folder containing that file or project, with that file or project selected in Windows Explorer.

View

The View button on the Files Panel toolbar has a submenu with options that determine what the Files Panel displays and how the Files Panel is organized.

- **Closed Files in Managed Projects:** Turn on to make the Files Panel show closed files that are still part of managed projects. These will appear dimmer than files that are open. Turn off to hide all closed files, even if they are still part of managed projects. Files that were closed from unmanaged projects are always removed from the project, and never show up on the Files Panel.
- **Collapse Automatically:** Turn on to collapse all nodes in the tree on the Files Panel, except those containing the active file. Turn off to expand all nodes, except those that you collapsed manually. If you manually expand a node, that node will stay expanded regardless of this option.
- **Flat List of Files:** Turn on to show a flat list of files directly under the project nodes, without any folder nodes, even if the files are in different folders. Turn off to show folder nodes under the project nodes to indicate the folder hierarchy that the files are in.
- **Group by File Type:** Turn on to put nodes for all the file types that are used in the project under the project node. The file and/or folder nodes are then put under the file type nodes. If you turn this off, the Files Panel does not add nodes for file types at all.

- **Paths Relative to Projects:** Turn on to add the nodes directly under the project node for files that are in the same folder as the .epp project file. If the project has files stored in subfolders of the folder containing the .epp file, then only nodes for the subfolders are added. For all other files, a drive letter node is added under the project node and the files and folder nodes are added to the drive letter node. If you turn off this option, drive letter nodes and folder nodes for the full path are added for all files in the Project. This option is not available when showing a flat list of files.
- **Subfolders first:** Turn on to first list subfolders under project and folder nodes, and then file nodes under the same project and folder nodes. Turn off to list files first, then subfolders. This option is not available when showing a flat list of files.

Sort

When there are multiple files under the same project node or folder node, the sort option determines how the files are ordered. The option does not affect the order of any other nodes.

- **Alphabetically:** Sort files alphabetically by their file names.
- **Tab Order:** List files in the same order as they have in the row of file tabs. This order is determined by the order in which you open files, and which file was active when you opened more files. Dragging and dropping tabs or Files Panel nodes changes the order of both the tabs and the Files Panel nodes.
- **Time Last Edited:** List files in the order which you last edited them, from most recently edited to least recently edited. If you edit a file in EditPad, it is moved to the top of the list immediately. All files that you have edited in EditPad since you last opened them are ordered by the moment of the last change, regardless of whether those changes were saved. Undoing an action is also considered a change and will move the file to the top of the list. Files that you have opened but not edited in EditPad are ordered by the last modification time stamp the files have on disk.

18. View | Explorer Panel

The Explorer Panel is a panel that sits docked at the left side of EditPad's window. You can make it visible by selecting Explorer Panel in the View menu. You can dock the panel elsewhere by dragging its caption bar or its tab.

The Explorer Panel functions as a miniature version of Windows Explorer, the file manager that is part of Windows. With EditPad Pro's Explorer panel, you can easily open files without having to go through File | Open. You can also copy, move and delete files, even if you didn't open those files in EditPad Pro. The copy, move and delete commands in the Explorer Panel work on the files that you've selected in the Explorer panel, rather than the files you have opened in EditPad Pro.

By default, the Explorer Panel shows a tree of all drives, folders and files on your computer. Since the list of files may be too long to work comfortably, you can apply various filters to make the Explorer Panel show only those files you're interested in.

The Explorer panel provides various commands in its right-click menu. Some of these commands also have their own buttons on the toolbar at the top of the Explorer panel.

Open

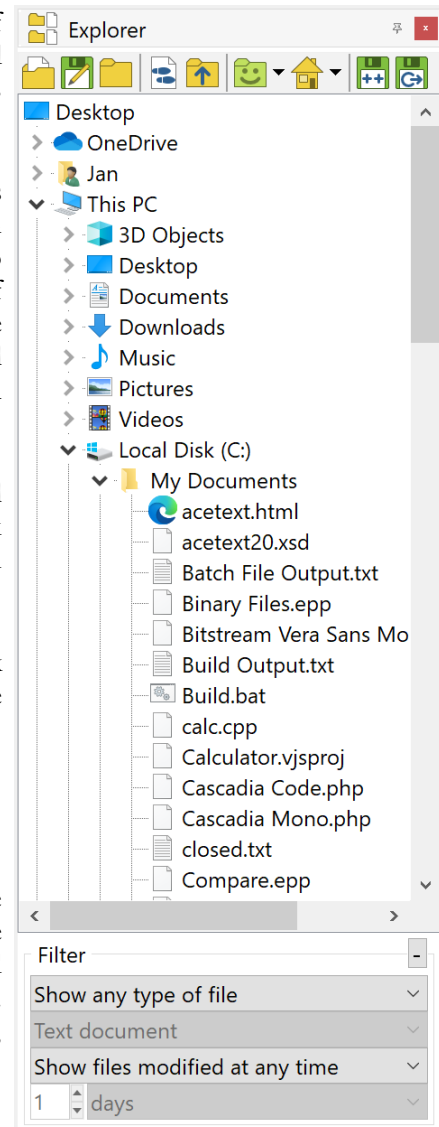
If you've selected one or more files, the Open button will open those files into the current project, just like File | Open would do. If you've selected one or more folders, the Open button will open all files in all the selected folders and all the subfolders of the selected folders. However, only files passing the Explorer Panel's filter will be opened, since those are the only files visible in the Explorer Panel.

Save into Folder

This command is identical to the File | Save As command, except that it defaults to the folder that you have selected in the Explorer Panel, or the folder containing the file that you have selected. The file that is saved is the file that you are editing, just like File | Save As does.

Create Folder

Create a new folder under the folder that you have selected in the Explorer Panel. You will be prompted for the folder's name.



Select Active File

If the file you're editing in EditPad Pro has a file name (i.e. you've saved it), you can click the Select Active File button to select that file in the Explorer Panel. Drives and folders in the Explorer Panel's tree will be expanded as needed. If the file you're editing is not visible in the Explorer Panel because you've filtered it out, then the folder containing the file will be selected instead.

Up Folder

The Up Folder button is a quick way to select the folder that contains the currently selected file or folder. If the home folder is selected when you click the Up Folder button, EditPad Pro will make the home folder's parent folder the new home folder. If the home folder is a drive, then the desktop space becomes the new home folder.

Favorite Folders

To add a folder to your favorite folders, click on the downward pointing arrow next to the Favorite Folders button, and click on the Add Selected Folder item. You can then quickly select that folder in the Explorer Panel by clicking on the downward pointing arrow next to the Favorite Folders button, and clicking on the folder you want to select.

Set Home Folder

Click the Set Home Folder button to make the currently selected folder the root folder of the Explorer Panel. Any folders above the home folder will be invisible. If you save all your text files on a particular drive or in a particular folder, setting that drive or folder as the home folder will make the Explorer Panel easier to navigate. EditPad Pro will remember your home folder next time you run it. The Explorer Panel will then load much faster, as it only has to scan the home folder that you specified, rather than all drives and network resources connected to your PC.

Explore in Windows Explorer

Launches Windows Explorer showing the folder that you selected in the Explorer Panel. If you selected a file or a project, then Windows Explorer is launched showing the folder containing that file or project, with that file or project selected in Windows Explorer.

Copy

If you selected one file, the Copy button will ask you for a folder where to save the file, and a name to save it under. With one file, the Copy command works like File|Save Copy As, except that it copies the file you selected in the Explorer Panel rather than the active file.

If you selected more than one file, or you selected one or more folders, the Copy button will ask you for a folder to copy all the selected files and folders into. Folders will be copied entirely, including all their files and

all their subfolders. All files will be copied, including files that are invisible in the Explorer Panel because of the file filter.

Rename / Move

If you selected one file, the Rename / Move button will ask you for the folder you want to move the file into, and the new name you want to give it. With one file, the Rename / Move command works like File|Rename / Move, except that it renames and/or moves the file you selected in the Explorer Panel rather than the active file.

If you selected more than one file, or you selected one or more folders, the Rename / Move button will ask you for a folder to move all the selected files and folders into. The files and folders will be moved only, not renamed. Folders will be moved entirely, including all their files and all their subfolders. All files in the selected folders will be moved, including files that are invisible in the Explorer Panel because of the file filter.

Delete

Click the Delete button to delete the selected files and folders. When deleting files and folders from the hard disk, they will be moved into the Windows Recycle Bin.

Refresh

Reload the list of files and folders under the selected folder.

19. File Filter

EditPad Pro's Explorer Panel and FTP Panel allow you to filter the files they display. Showing only the files you're interested in makes it easier to work with folders containing large numbers of files.

The Explorer and FTP panels are not updated instantly while you edit the filters. To update the panel, either click on it with the mouse or press Enter while one of the filter drop-down lists has keyboard focus.

You can filter files based on their names in four ways:

- Show any type of file: Do not filter files based on their names.
- Show files of type: Shows the files that match the file mask of the file types you've configured. Select the file type you want from the drop-down list.
- Files matching a file mask: Shows the files that match the file mask you type into the drop-down list. You can enter multiple masks separated by semicolons.
- Show files matching a regex: A file will only be shown if the regular expression you type into the drop-down list matches the file's name. If you enter multiple regular expressions separated by semicolons, a file will be shown if at least one of the regular expressions matches.

In addition to filtering files based on their names, you can also filter files based on the date and time they were last modified.

- Show files modified at any time: Do not filter files based on their last modification date.
- Last modified during the past...: Only show files that have been last modified in a certain number of past hours, days, weeks, months or years.
- Not last modified during the past...: Only show files that were not last modified in a certain number of past hours, days, weeks, months or years.
- Last modified since...: Only show files last modified on or after a specific date. Files last modified on the date you specify are shown.
- Not last modified since...: Only show files that were last modified before a specific date. Files last modified on the date you specify are shown.

When specifying a time period, EditPad Pro starts counting from the start of the current period. E.g. if you tell EditPad Pro to show files modified during the last two hours at half past three, EditPad Pro will show files modified at or after one o'clock, two hours before the start of the current hour.

Weeks start on Monday. If you tell EditPad Pro to show files modified during the last week on Wednesday the 14th, EditPad Pro will show files modified on or after Monday the 5th. The only exception to this rule is when you filter the files by a number of weeks on a Sunday. Then, EditPad Pro starts counting from the next Monday. The same filter on Sunday the 18th will have EditPad Pro show files modified on or after Monday the 12th.

20. View | FTP Panel

The FTP Panel is a panel that sits docked at the left side of EditPad's window. You can make it visible by selecting FTP Panel in the View menu. You can dock the panel elsewhere by dragging its caption bar or its tab.

FTP Panel Layout

The tree view at the top of the FTP Panel shows all the FTP servers EditPad Pro is currently connected to. EditPad Pro can be connected to any number of FTP servers at the same time. You can transfer files between your PC and all the connected servers simultaneously. Obviously, this will only be practical with a high-speed Internet connection. The tree at the top also shows an upload and download queue for each server below the server's node. You can resize the server tree by dragging the splitter bar immediately below the server tree, above the toolbar with the open, download and upload buttons.

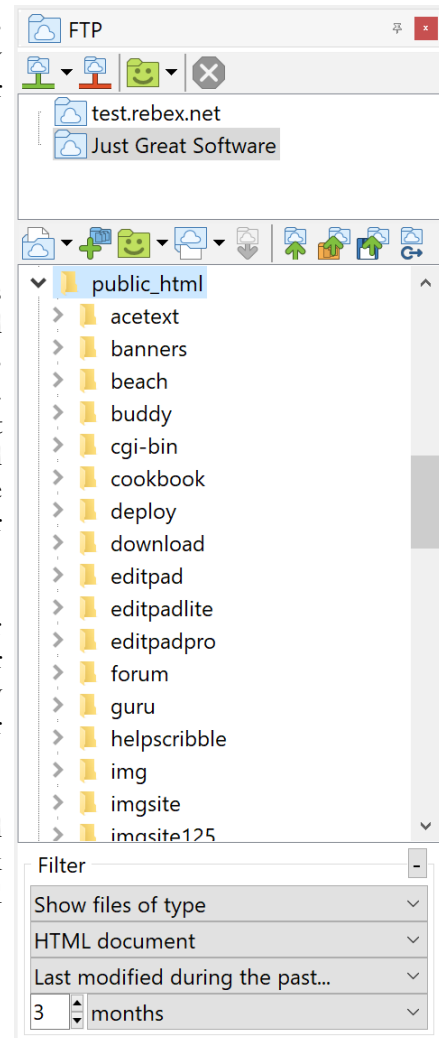
The tree at the bottom shows the folder structure of the FTP server you selected in the list of servers at the top. If you click on another server, the folder view will be replaced instantly. You can do so at any time, even when EditPad Pro is still retrieving part of the folder structure. The retrieval will continue in the background.

By default, the FTP Panel shows all folders and files on the selected FTP server. Since the list of files may be too long to work comfortably, you can apply various filters to make the FTP Panel show only those files you're interested in.

ASCII and Binary Mode

Many FTP client have a switch that lets you choose whether to upload and download files in ASCII or binary mode. In binary mode, the FTP client uploads or downloads an identical copy of the file. In ASCII mode, the FTP client and server will attempt to convert the file between the client and server operating systems. When using a Windows FTP client to upload a text file to a Linux server, for example, Windows-style line breaks will be converted to UNIX-style line breaks when uploading a file, and the other way around when downloading a file.

EditPad Pro always uploads and downloads files in binary mode. You'll always get an exact copy of what's on the server and you'll always put an exact copy of the file back onto the server. Unlike many other applications, EditPad Pro transparently handles Windows, UNIX, and classic Macintosh text files. Still, you should be careful when uploading files to Linux or UNIX servers. A CGI script, for example, may abort with a mysterious 500 server error if you upload it with Windows-style line breaks onto a web server running Linux. The solution is to select **Convert | Line Break Style | To UNIX (LF only)** in EditPad Pro's menu before uploading the file. You can also save the file on your Windows PC with UNIX-style line breaks, so you need to do the conversion only once. You can set the default for new files in the File Type Encoding settings.



Connect to FTP

Click the Connect to FTP button to connect to an FTP server. Enter the server's host name (e.g. ftp.mydomain.com), port, your login or user name and your password. You can also choose whether EditPad Pro should remember your password. EditPad Pro supports all the popular methods for securing FTP connections. You can select the one your server uses from the Encryption drop-down list. If you're not sure which method your server uses, the port number is often a good indication.

- No encryption: plain old unsecure FTP. The default port is 21. Choose this if your server doesn't support any encryption at all. If the server requires encryption, then EditPad won't be able to connect if you select "no encryption".
- Explicit TLS, fully encrypted: this is the recommended method to require a fully encrypted FTP connection. The connection starts with a plain old unsecure connection on port 21. Before logging in, EditPad requests TLS encryption from the server. If the server supports TLS then the connection is secured before your login and password are sent. If the server does not support TLS then the FTP session is aborted and EditPad shows an error message. In that case, try "implicit TLS" before trying "no encryption". EditPad Pro will encrypt file transfers. If the server does not support encrypted file transfers then EditPad Pro will be able to connect to the server but will then fail to retrieve any directory listings and will fail to transfer any files. In that case, switch to "TLS, files unencrypted".
- TLS, files unencrypted: use this method if your server supports TLS but not encrypted file transfers. The connection starts with a plain old unsecure connection on port 21. Before logging in, EditPad requests TLS encryption from the server. If the server supports TLS then the connection is secured before your login and password are sent. If the server does not support TLS then the FTP session is aborted and EditPad shows an error message. EditPad Pro will not encrypt file transfers. If the server does not allow unencrypted file transfers then EditPad Pro will be able to connect to the server but will then fail to retrieve any directory listings and will fail to transfer any files. In that case, switch to "TLS, fully encrypted".
- Implicit TLS, fully encrypted: use this method if your FTP server supports encryption but does not support negotiating for the encryption. EditPad will establish an FTP connection that is encrypted with TLS from the start. The default port is 990. If the server does not support SSL then the FTP

Connect to FTP Server [X]

Server: ftp.just-great-software.com

Encryption: Explicit TLS, fully encrypted

FTP Port: 21 IP version: Auto

Login: jgsoft

Password: ••••••••

☐ Remember password

☐ Keyboard-interactive authentication

Private key: [dropdown] ...

☐ Proxy Server...

Description: Just Great Software

Initial directory: [empty]

☒ Use initial directory as root

☐ Cache directory listings

☒ Passive FTP

☒ Keep connection alive with NOOP

SFTP Block Size: 128 KB

Backups on server: Single backup appending ~

Local backups: G:\temp\FTP Cache\ ...

☒ Load unchanged files from local backups

Translate FTP folders into HTTP URLs:

[Text area with scrollbars]

[OK] [Cancel] [Help]

session is aborted and EditPad will show an error message. In that case, try the more modern “explicit TLS” option before trying “no encryption”. EditPad Pro will encrypt file transfers. If the server does not support encrypted file transfers then EditPad Pro will be able to connect to the server but will then fail to retrieve any directory listings and will fail to transfer any files.

- Implicit TLS, files unencrypted: use this method if your FTP server supports non-negotiated encryption, but does not support encrypted file transfers. EditPad will establish an FTP connection encrypted with TLS from the start. The default port is 990. If the server does not support SSL then the FTP session is aborted and EditPad will show an error message. EditPad Pro will not encrypt file transfers. If the server does not allow unencrypted file transfers then EditPad Pro will be able to connect to the server but will then fail to retrieve any directory listings and will fail to transfer any files.
- SFTP (fully encrypted): select this method if your server is an SSH server rather than an FTP server. EditPad Pro will connect via SSH and use the SFTP protocol to transfer files directly via the SSH connection. The default port is 22. SFTP connections are always fully encrypted. The FTP protocol is not used at all. (SFTP and FTP are totally unrelated, even though they are similarly named and serve a similar purpose. EditPad Pro handles the differences transparently.)

Select IPv4 or IPv6 if you know which Internet Protocol version your FTP server runs on. Select “Auto” if you don’t. But “Auto” isn’t guaranteed to work. It can fail if the server’s domain publishes both IPv4 and IPv6 addresses but the FTP server can only be reached via IPv6. If EditPad can’t connect to your server at all and you’ve already tried all the encryption options then try selecting IPv4 or IPv6 and retry the various encryption options.

To connect to an FTP server, you need to enter a username and password. To connect to an SSH (SFTP) server, you need to enter a username and use one of three possible authentication methods. The first option is to enter a password. The second option is to tick the “keyboard-interactive authentication” checkbox. The third option is to provide a private key. Which method you need depends on whether your SSH server requires password authentication, keyboard-interactive authentication, or private key authentication.

Keyboard-interactive authentication results in a prompt during the connection. Usually this prompt asks for a password. The actual prompt depends on what the server asks from EditPad.

Private key authentication requires you to import a private key or select a previously imported one. Click the (...) button next to the “private key” setting to access EditPad’s private key storage. Click the Import button to import a private key. The first time you do this you will be prompted for a password (and again to avoid typos). This password is used to encrypt all the keys in EditPad’s private key storage. After importing the key, click the Select button to select that key and close the key storage dialog. You can connect to another server using the same key simply by selecting the key in the “private key” drop-down list in the FTP connection dialog. You can connect to another server using a different key by clicking the (...) button and importing another key. EditPad will ask for the private key storage password once per EditPad session at the moment when it needs to encrypt a key you’re importing or decrypt a key you need to connect to a server. The keys are stored with the rest of EditPad’s settings, which is in %APPDATA%\JGsoft\EditPad Pro 8 for normal installs, and EditPad’s installation folder for portable installs. If you forget your password then the EditPad’s private key storage becomes inaccessible. In that case, exit EditPad Pro, delete all the key files, and restart EditPad Pro to start with a fresh private key storage. You will be prompted for a new password when you import the first key.

Click the Proxy Server button if your PC can only connect to FTP or SSH servers via a proxy. Proxy settings are shared between all FTP connection settings. Proxy settings for FTP connections are separate from proxy settings for HTTP connections (such as for downloading schemes in the file type configuration). EditPad supports HTTP, SOCKSv4, and SOCKSv5 proxy servers. If you select “SOCKSv5 + DNS” then EditPad

sends your server's domain name to the proxy server, leaving it to the proxy server to resolve the domain. Otherwise EditPad lets your PC connect to its DNS server to resolve the domain and then sends the IP address to the proxy to make the connection.

Optionally, you can also enter a description for the FTP server. This description will appear in history lists instead of the default `username@hostname`.

The initial directory is the directory that EditPad Pro navigates to when you first connect to the FTP server. If you don't specify an initial directory then EditPad Pro lets the FTP server choose the initial directory. If you turn on the option "use initial directory as root" then you can only access the initial directory and its subdirectories. You won't be able to access the directory's parent or sibling directories. This option is available whether you specified the initial directory or not.

If you have a slow internet connection then you can tell EditPad Pro to keep a cached copy of the FTP server's directory listings. This speeds up future connections to the same server. You may want to turn off this feature if you often modify files on the FTP server outside of EditPad Pro, as that will cause the cached listings to be out of date, which may be confusing.

If EditPad Pro cannot retrieve an FTP server's directory listings, try toggling the "passive FTP" option. While passive FTP is usually preferred, not all servers support it.

If a server times out the connection quickly then you can tell EditPad Pro to send "no operation (NOOP)" commands to the server every 30 seconds. Not all servers reset the time-out in response to NOOP commands.

Setting the SFTP block size to 128 KB speeds up SFTP connections. But this may prevent EditPad Pro from transferring files to an older SFTP server if it does not support 128 KB blocks. 32 KB is the largest block size that every SFTP server must support.

You can use the "backups on server" setting to have EditPad Pro rename the existing file to a backup file whenever it is about to overwrite a file on your FTP server. If the backup file already exists, it is replaced by the new backup. The History panel does not track this backup copy on the server.

You can speed up working with files on an FTP server by specifying a folder for local backups. Whenever you open a file from the FTP server, EditPad Pro writes a copy of that file to the local backup folder. When you save the file, EditPad Pro saves both the local backup and uploads the new file to the FTP server. If you enabled backup copies on the Save Files page in the Preferences then EditPad Pro keeps backups of the local copy of the file when you save it to FTP. This allows you to use the History panel with files opened from FTP. It will show the local backup copies. If you enabled working copies in the Save Files Preferences then the working copies are saved in the local backup folder of each FTP server. Working copies are never uploaded to the server.

EditPad Pro does not delete the local backup copy when you close a file you opened via FTP. Turn on "load unchanged files from local backups" to read the local backup copy of a file instead of downloading the file from the server if the file's size and time stamp on the server is still the same as that of the local backup copy.

In the "translate FTP folders into HTTP URLs" box, you can enter a list of FTP folder paths and their corresponding HTTP URLs. Enter one folder=URL pair per line, separating the folder and URL with an equals sign. The folder must be a full path. When you select the View|Browser command when you're editing a file in that folder, EditPad Pro will replace the folder path in the file's path with the URL. E.g. if you

specify `/usr/home/me/public_html/=http://www.mydomain.com` and you're editing the file `/usr/home/me/public_html/subfolder/file.html` then EditPad Pro will open the URL `http://www.mydomain.com/subfolder/file.html`

When you want to connect to the same server again, you can select it from the Connect button's drop-down menu. If the password was remembered, EditPad Pro will connect to the server instantly. Otherwise, it will ask you for the password.

You can connect to as many servers as you like. Connecting to another server does not break previous connections. Simply click on a connection in the top half of the FTP panel to see the file listings. You can also connect to the same server more than once if you use a different login name. Since a server may restrict the number of connections from a single computer, you may have to disconnect from the server before reconnecting with a different login. EditPad Pro does not impose any restrictions on the number of simultaneous connections to an FTP server.

Disconnect from FTP

Click the Disconnect from FTP button to break EditPad Pro's connection with the selected FTP server. Any pending uploads or downloads will be aborted. The server will be removed from the list of servers in the top half of the FTP panel.

If the FTP server or a network outage breaks the connection, the server will remain listed in the FTP panel. EditPad Pro won't notice that the connection was broken until you try to upload or download another file. When you do, EditPad Pro will automatically try to reconnect to the server every 10 seconds until either it succeeds, or you click the Disconnect button.

Favorite Servers

The Connect to FTP button will automatically keep a history of the last 16 FTP servers that you connected to. If you connect to lots of FTP servers, and use some more frequently than others, you can add the FTP servers you use frequently to your list of favorite FTP servers. To add a server to the list, connect to it. Then click on the downward pointing arrow next to the Favorites button, and click on the Add Connected FTP Server item. To connect to a favorite FTP server, select it from the Favorites button's drop-down menu.

Abort

Aborts all uploads and downloads to and from the selected server that are currently pending or in progress.

Open from FTP

Click the Open from FTP button to open all selected files for editing. EditPad Pro will instantly create new, empty tabs for the files. The files are added to the FTP server's download queue. While a file downloads, its text will appear in EditPad Pro bit by bit. You can start editing the downloaded portion right away. The remainder of the file will be appended as it is downloaded. If you close a tab before the file is downloaded, that file's download will be aborted.

You cannot open entire folders from FTP like you can open entire folders in the Explorer Panel. If you click Open from FTP with a folder selected, that folder's node will simply be expanded. EditPad Pro will retrieve the list of subfolders and files, but won't actually download the files themselves.

EditPad Pro will remember that you opened the files from FTP. When you use File|Save or File|Save All, or you confirm the save question asked by File|Close and similar commands, EditPad Pro will automatically upload the file back to the FTP server, overwriting the original file. To save the file on your computer, use File|Save As instead. Once you save the file on your own computer, EditPad Pro will *not* automatically upload it to the FTP server when you save.

The drop-down menu of the open button shows a list of files that you have recently opened from this server. Select a file from the drop-down menu to open it again.

Add to Project from FTP

The Add to Project from FTP button opens a file from FTP just like the Open from FTP button does. If the active project is unmanaged then there is no difference between the two. The file then becomes part of the project regardless of which command you use.

If the active project is a managed project then Add to Project from FTP makes the newly opened file a part of the project. When you open the project again, EditPad Pro will try to download the file from FTP again. It will prompt to reconnect to the server if you're not connected when reopening the project. Open from FTP opens the file as an outside file. Then the file is not opened when you close and reopen the managed project.

Favorite Files

If you often edit particular files on this server, you can add them to the list of favorite files via the Favorites button next to the Open button. EditPad Pro keeps a list of favorite files for each server that you added to your favorite servers, and each server that is remembered by the Connect to FTP button. These lists are separate. To open a particular favorite file, first connect to the server the file is on, and then select the file from the favorite files on that server.

Tail on FTP

Tailing a file on FTP is very convenient for dealing with files that may be slow to download entirely, that may be updated continuously, or that have the information you want at the end of the file. Log files are a prime example. Tailing these files on FTP instead of opening them can save you a lot of download time by downloading only the end of the file. It can keep you up-to-date by continuously reloading that tail end.

The Tail on FTP command shows the same dialog box as the File|Tail command. All the options work in the same way via FTP. Depending on the speed of your FTP connection, you may choose a smaller tail end to load and a larger reload interval. The interval is paused while the updated tail end is downloaded. So if you set a 10-second interval but it takes 2 seconds to download the new tail end of the file then you'll see the file update every 12 seconds. This ensures that the file remains workable if the download time were to exceed the interval.

Download

Click the Download button to download the selected files to disk, without opening them for editing in EditPad Pro. If you selected one file, EditPad Pro will ask you where and with which name you want to save it. If you selected more than one file, EditPad Pro will ask you for the folder where you want to save the selected files. All files will be downloaded with the same name as they have on the FTP server.

Upload File

Click the Upload File button to upload the file you're currently editing to the FTP server. If you selected a folder in the FTP panel, the file will be uploaded into that folder. If you selected a file in the FTP panel, the file you're editing will be uploaded into the same folder as the selected file. EditPad Pro will ask you which name you want the uploaded file to have on the FTP server.

If the file you're uploading was untitled, EditPad Pro will change the tab's caption to the file name you used to upload the file. If you later modify the file and use File|Save or File|Save All, or you confirm the save question asked by File|Close and similar commands, EditPad Pro will automatically upload the file back to the FTP server, overwriting the original file. To save the file on your computer, use File|Save As instead. Once you save the file on your own computer, or if it already had been saved on your own computer before you uploaded it, EditPad Pro will *not* automatically upload it to the FTP server when you save.

Upload Project

Click the Upload Project button to upload all files in the current project. If you selected a folder in the FTP panel, the files will be uploaded into that folder. If you selected a file in the FTP panel, the project's files will be uploaded into the same folder as the selected file.

If one or more of the files in the project are untitled, EditPad Pro will ask you for a name for those files. Files that were previously uploaded or saved will be uploaded with the name they already have. Files that were untitled when you uploaded them will be automatically uploaded when you save them, just like when uploading a single untitled file.

Upload from Disk

Click the Upload Project button to upload files that you aren't editing in EditPad Pro. EditPad Pro will show an open file dialog to select the files to upload. You can select multiple files to be uploaded. If you selected a folder in the FTP panel, the files will be uploaded into that folder. If you selected a file in the FTP panel, the files will be uploaded into the same folder as the selected file.

Should you use the Upload from Disk file to upload a file that you're editing in EditPad Pro, then the copy of the file on disk will be uploaded. If the file has unsaved changes in EditPad Pro, those changes won't appear in the uploaded file.

Create Folder

Click the Create Folder button to create a new folder on the FTP server. If you selected a folder in the FTP panel, the new folder will become a subfolder of the selected folder. If you selected a file in the FTP panel, the folder will become a subfolder of the folder that holds the selected file.

Refresh Folder

EditPad Pro automatically updates the contents of a folder shown in the files and folders tree on the FTP panel whenever you use EditPad to upload, rename, or delete a file via FTP in that folder. If you use another way to add or delete files on the FTP server, then EditPad won't show those changes on the FTP panel. To make EditPad Pro check for changes in a particular folder, select that folder or a file in that folder and invoke the Refresh Folder command via the context menu.

If you regularly use another FTP client, you may want to turn off the option to cache directory listings when you connect to the FTP server in EditPad. Then EditPad will refresh all folder listings each time you connect to the FTP server.

Rename/Move

Rename or move the selected file or folder. You will be prompted for the new name or path.

File Permissions

If your FTP server is running on a UNIX or Linux host, select the File Permissions item in the context menu if you want to change the permissions on the selected file. These are the permissions that you can set via the `chmod` command in UNIX or Linux.

Delete

To delete a file or folder, right-click on it and select Delete in the context menu. Most FTP servers will not allow you to delete a folder until you've deleted all of its files and subfolders. You can delete multiple files and folders by selecting them all, and then right-clicking on one of the selected items.

21. View | File History

The File History is a panel that sits docked at the left side of EditPad's window. You can make it visible by selecting File History in the View menu. You can dock the panel elsewhere by dragging its caption bar or its tab.

The File History shows a list of all backup files that exist for the file you're currently editing in EditPad Pro. To determine which files are backups, the File History uses the "backup copies" setting in the Save Files Preferences. The File History will only show files that follow the naming method you selected in the Save Files Preferences. To make the File History useful, you should select one of the "multi backup" options or the "hidden history" option. These are the only backup methods that can keep more than one backup copy of any given file.

The File History will display two dates for each backup file: the modification date and the backup date. When creating a backup copy, EditPad Pro sets the backup's last modification time stamp to that of the original file. E.g. Today, December 1st, you open a file last modified on November 27th. When you save the file, EditPad Pro will create a backup copy with a modification date of November 27th. The original file will get a modification date of December 1st. When you view the contents of the backup folder in Windows Explorer, the backup file's date will show the November 27 modification date.

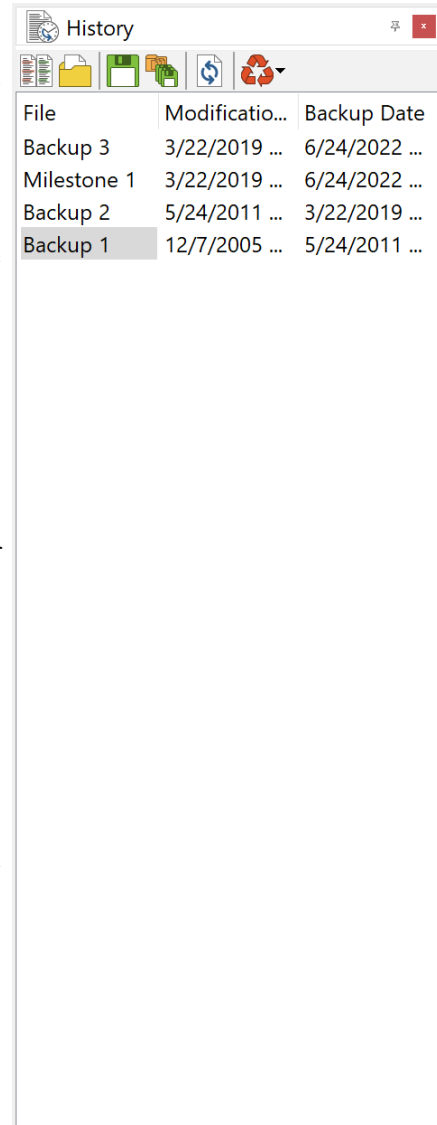
The backup date indicates the date that the backup copy was made. In the example above, that would be December 1st. If you right-click on the file in Windows Explorer and select properties, the file's creation date will indicate the date the backup copy was made.

Compare

Select one of the backup copies and click the Compare button to show the differences between the backup copy and the state the file currently has in EditPad Pro. You will be asked for the same file comparison options used by Extra | Compare Files. The backup copy will be highlighted as the "old" file, and the file you're editing in EditPad Pro as the "new" file.

To compare two backup copies with each other, first open the younger copy of the two. Then, select the older copy and click the Compare button. The backup copy you've opened in EditPad Pro will always be highlighted as the "new" file, because it takes the role of the file you're currently editing. Therefore, you should open the younger copy of the two you want to compare. This will avoid confusion as to which is which.

When the active tab has file comparison highlights, the File History will show the full paths of the files that were compared along with their colors. The color for the original file, or the "old" file, is used to highlight lines only present in that file. The color for the edited file, or the "new" file, is used to highlight lines only present in that file. Lines present in both files are not highlighted.



Open

Select a backup file and click the Open button to open the file in a tab for viewing or editing. The tab will indicate the file name of the backup file. If you try to save the backup file with File|Save, EditPad Pro will prompt you for a file name. You should save the file under a new name rather than overwriting a backup file. If you overwrite a backup file, EditPad Pro will make a backup of the backup, which can get confusing.

If you open a backup copy via the File History, the File History will continue to display the backup copies of the original file. You should not use File|Open or other commands to open backup copies outside the File History, because then the link between the backup and the original won't be established.

Save Milestone

The Save Milestone button saves the file you're editing in EditPad Pro. However, it does not overwrite the original file. Instead, it creates a separate "milestone" copy. The milestone copy will appear among the backup files in the File History. The difference between a milestone copy and a backup copy is that EditPad Pro will automatically delete backup copies when they exceed the number and age limits you specified in the Open Files Preferences. Milestone copies are never automatically deleted. They function as a more permanent backup. You should make a milestone copy whenever you're embarking on a more involved editing task that you might want to undo entirely.

If you forgot to save a milestone copy, you can do so afterwards as long as there's still a backup copy of the file in the state it had when you should have saved the milestone. Simply open that backup copy via the File History, and click the Save Milestone button. When you save a milestone of a backup file that you've opened via the File History, EditPad Pro will make a copy of that backup file, and turn it into a milestone of the original file.

If you save a milestone of a file that has unsaved changes, the milestone will be identical to the in-memory copy of the file rather than a copy of the file on disk. The milestone will include your unsaved changes. The file on disk is not affected. So you could use the milestone feature to save unsaved changes without actually saving the file, in order to make a backup of changes you plan to abandon.

Save Milestone for All Files in Project

The Save Milestone for All Files in Project button will create a milestone copy of each file in the current project, just like the Save Milestone button does for the current file.

Revert

Click the Revert button to replace the original file with the selected backup file. EditPad Pro will first create a new backup copy from the original file, and then replace the original file with the selected backup file. The last modification date on the reverted file will be that of the backup file, rather than that of the moment you clicked the Revert button. The selected backup file is deleted in the process.

If you change your mind after reverting a file, select the backup file with the most recent backup date, which is the backup created by the reversion. Click the Revert button again to restore the file and its backups to the

way they were before the first reversion. If you are using the numbered backups, the numbers of the backups will have shifted, but their modification dates and their contents will be as they were.

If you edit and save a file after reverting it, the backup made when saving will have the same modification date as the backup that you reverted to and will thus appear at that position in the list of backups rather than at the top.

Delete

The Delete button has 3 subitems. You can choose to delete only the selected backup file, all backup files for the file that is active in EditPad, or all backup files for all files in the project that is active in EditPad.

23. Help Menu

Keyboard Navigation and Editing Shortcuts

The shortcuts in this list are the shortcuts that are recognized by every full text editor control in EditPad, such as the main editor, the search box, the replace box, the clip editor, etc.

Some of these shortcuts are also used by menu items. If you select the command from the menu, it will always work on the main editor. If you select the command by pressing the shortcut key combination, it will work on the editor that has input focus. This is the editor that shows the text cursor, the blinking vertical bar. This is true for all menu items listed below, but not for any other menu items.

All shortcuts that are used by menu items can be configured in Options|Preferences|Keyboard. All other shortcuts cannot be changed.

When a key combination only works in text mode, this is indicated by [text]. If it only works in hexadecimal mode, the indication is [hex].

Cursor navigation keys

Arrow key: Moves the text cursor (blinking vertical bar).

Ctrl+Arrow Left [text]: Moves the text cursor to the start of the previous word or the end of the previous line, whichever is closer.

Ctrl+Arrow Right [text]: Moves the text cursor to the start of the next word or the end of the current line, whichever is closer.

Ctrl+Arrow Up or Down: Scrolls the text one line up or down, or jumps to the next paragraph (your preference). When scrolling, the cursor moves along unless it's at the top or bottom (configurable).

Ctrl+Alt+Arrow Up or Down: Moves the text cursor to the previous or next occurrence of the word under the text cursor.

Page Up or Down: Moves the text cursor up or down an entire screen.

Ctrl+Page Up or Down: Scrolls the text one screen up or down.

Alt+Page Up or Down: Scrolls the text to make the line with the cursor the last or the first visible line if the cursor is above or below the last or first visible line. Scrolls the text one screen up or down otherwise.

Home: Moves the text cursor to the beginning of the line. Can be configured to move to the first non-whitespace character of the line, and to the very start of the line after a second press.

Ctrl+Home: Moves the text cursor to the start of the entire text.

End: Moves the text cursor to the end of the line. Can be configured to move to the last non-whitespace character of the line, and to the very end of the line after a second press.

Ctrl+End: Moves the text cursor to the end of the entire text.

Shift+Navigation key: Moves the text cursor and expands or shrinks the selection towards the new text cursor position. If there was no selection, one is started. Pressing Ctrl as well moves the text cursor correspondingly.

Alt+Shift+Navigation [text]: The same as when Alt is not pressed, except that the selection will be rectangular instead of flowing along with the text.

Ctrl+]: Go | Go to Matching Bracket

Shift+Ctrl+[: Go | Go to Unmatched Bracket

Ctrl+[: Block | Between Matching Brackets

Any of the above key combinations that do not select part of the text remove the current selection (the selection, not the selected text) unless selections are persistent. Any key combination that moves the text cursor also causes the text to scroll to keep the text cursor in view if the move makes the text cursor invisible. Any key combination that scrolls the text lets the text cursor keep its position relative to the text, unless you chose to keep the text cursor in view while scrolling.

Editing commands

Enter: In the main editor: inserts a line break. In the search or replace box: searches for the next occurrence.

Shift+Enter: Inserts a line break.

Ctrl+Enter: Edit | Insert Page Break

Delete: Deletes the current selection if there is one and selections are not persistent. Otherwise, the character to the right of the text cursor is deleted.

Ctrl+Delete: Deletes the current selection if there is one. In text mode, if there is no selection, the part of the current word to the right of the text cursor is deleted. If the cursor is not on a word, all characters to the right of the cursor up to the start of the next word are deleted.

Shift+Ctrl+Delete: In text mode, all the text on the current line to the right of the text cursor is deleted. In hexadecimal mode, the selection is deleted.

Backspace: Deletes the current selection if there is one and selections are not persistent. Otherwise, the character to the left of the text cursor is deleted.

Ctrl+Backspace: Deletes the current selection if there is one and selections are not persistent. In text mode, if there is no selection, the part of the current word to the left of the text cursor is deleted. If the cursor is not on a word, all characters to the left of the cursor up to the start of the next word are deleted.

Shift+Ctrl+Backspace: Deletes the current selection if there is one and selections are not persistent. In text mode, if there is no selection, all the text on the current line to the left of the text cursor is deleted.

Alt+Backspace: Alternative shortcut for Edit|Undo

Alt+Shift+Backspace: Alternative shortcut for Edit|Redo

Ctrl+Z: Edit|Undo

Ctrl+Y: Edit|Redo

Insert: Toggles between insert and overwrite mode.

Tab: In text mode, if there is a selection, the entire selection is indented. If there is no selection, a tab is inserted. In hexadecimal mode, pressing Tab makes the text cursor switch between the hexadecimal side and text side.

Shift+Tab: In text mode, if there is a selection, the entire selection is unindented (outdented). If there is no selection and there is a tab, or a series of spaces the size of a tab, to the left of the text cursor, that tab or spaces are deleted.

Ctrl+A: Edit|Select All

Shift+F5: Go|Previous Editing Position

Ctrl+Alt+Y: Edit|Delete Line

Shift+Ctrl+Alt+Y: Edit|Duplicate Line

Shift+Ctrl+B: Block|Begin Selection

Shift+Ctrl+E: Block|End Selection

Shift+Ctrl+D: Block|Expand Selection

Ctrl+D: Block|Duplicate

Ctrl+M: Block|Move

Ctrl+Alt+B: Block|Go to Beginning

Ctrl+Alt+E: Block|Go to End

Shift+Ctrl+]: Edit|Insert Matching Bracket

Clipboard commands

Ctrl+X: Edit|Cut

Shift+Ctrl+X: Edit|Cut Append

Ctrl+C: Edit|Copy

Shift+Ctrl+C: Edit|Copy Append

Ctrl+V: Edit|Paste

Shift+Ctrl+V: Edit|Swap with Clipboard

Shift+Delete: Alternative shortcut for Edit|Cut

Ctrl+Insert: Alternative shortcut for Edit|Copy

Shift+Insert: Alternative shortcut for Edit|Paste

Search Toolbar

Some of the search options on the search toolbar have Alt+letter access key shortcuts that are also used by main menu items. When the search toolbar or search panel has keyboard focus, the search options take precedence when you use an Alt+letter shortcut. Otherwise, the main menu takes precedence.

Characters with Diacritics

While you can use the Character Map to insert any character that you can't type on your keyboard, you can type many characters with diacritics in EditPad even if they don't appear on your keyboard. First, hold down the Ctrl key and press a punctuation key. If your keyboard uses the Shift key to type a particular punctuation character then hold down the Shift key too. Release all keys. Then type in a letter from A to Z, holding down Shift if you want a capital letter.

In EditPad Pro, Ctrl+/_ is a shortcut for Block|Toggle Comment by default. If you want to use Ctrl+/_ to type characters with strokes, remove the shortcut from Block|Toggle Comment in the keyboard preferences.

Punctuation	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Ctrl+'	á	á	ć	đ	é		ġ	í		í	í	í	í	í	ó	ó	ó	ó	ó	ó	ú	ú	ú	ú	ú	ú
Ctrl+`	à			è				ì						ò							ù	ù	ù	ù	ù	ù
Ctrl+:	ä			ë			ĥ	ï						ö						ť	ü	ü	ü	ü	ü	ü
Ctrl+^	â	â	â	ê	ê	ê	ĥ	î	î	î	î	î	î	ô	ô	ô	ô	ô	ô	ř	š	š	š	š	š	š
Ctrl+~	ã			ẽ				ĩ						õ	õ	õ	õ	õ	õ		û	û	û	û	û	û
Ctrl+,			ç	đ	ę		ğ	ĥ			ķ	ķ	ķ	ņ				ř	š	š	š					
Ctrl+@	â	©															®	™	™	™	ü	ü	ü	ü	ü	ü
Ctrl+.	à	â	â	â	ê	ê	ĥ	î	î	î	î	î	î	ô	ô	ô	ô	ô	ô	ř	š	š	š	š	š	š
Ctrl+/_		â	â	â	ê	ê	ĥ	î	î	î	î	î	î	ô	ô	ô	ô	ô	ô	ř	š	š	š	š	š	š
Ctrl+&	æ				ff			fi	ij		fl			œ				ß	st							

Punctuation	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Ctrl+'	Á		Ć	Đ	É		Ġ		Í		Ķ	Ļ	Ņ	Ō	Ó	Ŕ	Ŝ		Ú		Ŵ		Ý	Ž		
Ctrl+`	À				È				Ì					Ñ	Ò						Û		Ŵ		Ỳ	
Ctrl+:	Ä				Ë			Ĥ	İ						Ö						Ü		Ẁ	Ẃ	Ỳ	
Ctrl+^	Â		Ĉ	Ď	Ê		Ĝ	Ĥ	Î	Ĵ	Ķ	Ļ		Ņ	Ō			Ř	Ŝ	Ť	Û		Ẁ		Ỳ	Ž
Ctrl+~	Ã				Ẽ				Ĩ					Ñ	Õ						Û	Ỳ			Ỳ	
Ctrl+,			Ç	Đ	È		Ġ	Ĥ			Ķ	Ļ		Ņ				Ř	Ŝ	Ť						
Ctrl+@	À		©															®	™	Û						
Ctrl+.	À	Â	Ĉ	Ď	Ê	Ĝ	Ĥ	Î					Ņ	Ō	Ó	Ŕ		Ř	Ŝ	Ť			Ẁ	Ẃ	Ỳ	Ž
Ctrl+ /			ç	đ	è	ğ	ĥ	ı			ķ				ø					ť					Ẃ	Ž
Ctrl+&	Æ								ffi	Ij		ffl			Œ					§	ft					

Punctuation	1	2	3	4	5	6	7	8
Ctrl+ /		1/2	3/4	1/4	5/8	3/8	7/8	1/8
Ctrl+:	5/6	2/3	1/3	4/5	1/5	1/6	2/5	3/5

Tip of The Day



The first time EditPad starts without any prompts (such as it not being the default text editor), it shows the tip of the day screen at startup. The tips highlight some of EditPad's more interesting features. Click the More Info button to open EditPad's help file to learn more about the feature described in the tip. If you're in the mood for reading tips, click the Next Tip button to read more tips.

Each time you start EditPad it shows a different tip. If you dismiss the tip of the day window within 3 seconds, such as by pressing the Esc key on the keyboard, EditPad assumes you didn't read the tip, and shows the same tip next time. If you want to go back to a tip that you read in the past, click the Previous Tip button.

If you don't want the tip of the day to be shown each time you start EditPad, clear the "show tip of the day at startup" checkbox before you dismiss the tip of the day window. If you want to show the tips again, select Tip of the Day in the Help menu, and tick the checkbox.

Help | EditPad Web Site

Launches your web browser to open <https://www.editpadpro.com/>

Help | Check for New Version

Launches your web browser to show a page that tells you whether the version of EditPad you are presently using is the most recent one or not.

If a newer version is available then you will be able to download it immediately.

Share Experiences and Get Help on The User Forums

When you click the Login button you will be asked for a name and email address. The name you enter is what others see when you post a message to the forum. It is polite to enter your real, full name. The forums are private, friendly and spam-free, so there's no need to hide behind a pseudonym. While you can use an anonymous handle, you'll find that people (other EditPad users) are more willing to help you if you let them know who you are. Support staff from Just Great Software will answer technical support questions anyhow.

The email address you enter is used to email you whenever others participate in one of your discussions. The email address is never displayed to anyone, and is never used for anything other than the automatic notifications. EditPad's forum system does not have a function to respond privately to a message. If you don't want to receive automatic email notifications, there's no need to enter an email address.

If you select "never email replies", you'll never get any email. If you select "email replies to conversations you start", you'll get an email whenever somebody replies to a conversation that you started. If you select "email replies to conversations that you participate in", you'll get an email whenever somebody replies to a conversation that you started or replied to. The From address on the email notifications is forums@jgsoft.com. You can filter the messages based on this address in your email software.

EditPad's forum system uses the standard HTTP protocol which is also used for regular web browsing. If your computer is behind an HTTP proxy, click the Proxy button to configure the proxy connection.

If you prefer to be notified of new messages via an RSS feed instead of email, log in first. After EditPad has connected to the forums, you can click the Feeds button to select RSS feeds that you can add to your favorite feed reader.

Various Forums

Below the Login button, there is a list where you can select which particular forum you want to participate in. The “EditPad” forum is for discussing anything related to the EditPad software itself. This is the place for technical support questions, feature requests and other feedback regarding the functionality and use of EditPad.

The “regular expressions” forum is for discussing regular expressions in general. Here you can talk about creating regular expressions for particular tasks, and exchange ideas on how to implement regular expressions with whatever application or programming language you work with.

Searching The Forums

Before starting a new conversation, please check first if there’s already a conversation going on about your topic. If you find one, start with reading all the messages in that conversation. If you have any further comments or questions on that conversation, reply to the existing conversation instead of starting a new one. That way, the thread of the conversation stays together, and others can instantly see what you’re talking about. It doesn’t matter whether the conversation is many years old. If you reply to it, it becomes an active conversation that moves to the top automatically.

In the top right corner of the Forum window, there is a box on the toolbar that you can use to search for messages. When you enter something into that box, only conversations that include at least one message containing the word or phrase you entered are shown.

The filtering happens in real time as you type in your word or phrase. It includes all conversation subjects, all message summaries, and all author names. It also includes the message bodies of conversations that have been downloaded. EditPad automatically downloads the 20 most recent conversations when you connect to the forum. Other conversations are downloaded only if you click on them to view them. Downloaded messages are cached between EditPad sessions.

This means that the real time filtering does not search through the body text of older messages that you’ve never viewed. To search through those messages, click the Search Forum button that sits immediately to the left of the search box. EditPad then shows all conversations with messages containing the search term in their summaries, author names, or body texts, including messages that haven’t been downloaded yet.

You can enter only one search term, which is searched for literally. If you enter “find me”, only conversations containing the two words “find me” next to each other and in that order are shown. You cannot use boolean operators like “or” or “and”. Since the filtering is instant, you can quickly try various keywords.

If your search term can be found in the subject of a conversation, then all messages in that conversation are always shown. If the search term cannot be found in the subject of a conversation, but it can be found in the summary or body text of a message in that conversation, then the Show Complete Conversations button determines which messages are shown. When this button is up, only the messages in which the search term was found are shown for that conversation. When this button is down, all messages are shown for that conversation.

If you have previously participated in the forums, you can use the Show My Conversations button to show only conversations that you have participated in and conversations that have a message that you gave a +1. If

you did not enter a search term, this shows all messages in all those conversations. If you did enter a search term, this reduces the search results to conversations that you participated in.

Conversations and Messages

The left-hand half of the Forum window shows two lists. The one at the top shows conversations. The bottom one shows the messages in the selected conversation. You can change the order of the conversations and messages by clicking on the column headers in the lists. A conversation talks about one specific topic. In other forums, a conversation is sometimes called a thread.

If you want to talk about a topic that doesn't have a conversation yet, click the New button to start a new conversation. A new entry appears in the list of conversations with an edit box. Type in a brief subject for your conversation (up to 100 characters) and press Enter. Please write a clear subject such as "scraping an HTML table in Perl" rather than "need help with HTML" or just "help". A clear subject significantly increases the odds that somebody who knows the answer will actually click on your conversation, read your question and reply. A generic scream for help only gives the impression you're too lazy to type in a clear subject, and most forum users don't like helping lazy people.

After typing in your subject and pressing Enter, the keyboard focus moves to the empty box where you can enter the body text of your message. Please try to be as clear and descriptive as you can. The more information you provide, the more likely you'll get a timely and accurate answer. If your question is about a particular regular expression, don't forget to attach your regular expression or test data. Use the forum's attachment system rather than copying and pasting stuff into your message text.

If you want to reply to an existing conversation, select the conversation and click the Reply button. It doesn't matter which message in the conversation you selected. Replies are always to the whole conversation rather than to a particular message in a conversation. EditPad doesn't thread messages like newsgroup software tends to do. This prevents conversations from veering off-topic. If you want to respond to somebody and bring up a different subject, you can start a new conversation, and mention the new conversation in a short reply to the old one.

When starting a reply, a new entry appears in the list of messages. Type in a summary of your reply (up to 100 characters) and press Enter. Then you can type in the full text of your reply, just like when you start a new conversation. However, doing so is optional. If your reply is very brief, simply leave the message body blank. When you send a reply without any body text, the forum system uses the summary as the body text, and automatically prepends [nt] to your summary. The [nt] is an abbreviation for "no text", meaning the summary is all there is. If you see [nt] on a reply, you don't need to click on it to see the rest of the message. This way you can quickly respond with "Thank you" or "You're welcome" and other brief courtesy messages that are often sadly absent from online communication.

When you're done with your message, click the Send button to publish it. There's no need to hurry clicking the Send button. EditPad forever keeps all your messages in progress, even if you close and restart EditPad, or refresh the forums. Sometimes it's a good idea to sleep on a reply if the discussion gets a little heated. You can have as many draft conversations and replies as you want. You can read other messages while composing your reply. If you're replying to a long question, you can switch between the message with the question and your reply while you're writing.

Dates and Times

The Started column indicates how long ago each conversation was started. The Last Reply column indicates how long ago the last reply was made, if any. For older conversations, it indicates how much later that reply came after the conversation was started. If you sort conversations by last reply, conversations without replies are sorted by their starting date. The sort order is always based on absolute dates.

The Date Posted column indicates how long ago each message was posted. For replies to older conversations, this date indicates how much later that message was posted after the conversation was started. The Date Edited column indicates how long ago the message was edited, if at all. For older messages, it indicates how much later it was edited after it was posted. If you sort messages by the date they were edited, messages that weren't edited are sorted by their posting date. The sort order is always based on absolute dates.

Directly Attach Files and Screen Shots

One of the greatest benefits of EditPad's built-in forums is that you can attach files to your messages. Simply click the Attach button and select the item you want to attach.

To attach a screen shot, press the Print Screen button on the keyboard to capture your whole desktop. Or, press Alt+Print Screen to just capture the active window (e.g. EditPad's window). Then switch to the Forum window, click the Attach button, and select Clipboard. You can also attach text you copied to the clipboard this way.

It's best to add your attachments while you're still composing your message. The attachments appear with the message, but won't be uploaded until you click the Send button to post your message. If you add an attachment to a message you've written previously, it is uploaded immediately. If you send a message and later notice you forgot an attachment then you can attach it directly. You shouldn't click the Edit button unless you want to edit the body text of the message.

You cannot attach anything to messages written by others. Write your own reply, and attach your data to that.

To check out an attachment uploaded by somebody else, click the Use or Save button. The Use button loads the attachment directly into a new tab in EditPad, even if the attachment is not a plain text file. If you click the Save button, EditPad prompts for a location to save the attachment. EditPad does not automatically open attachments you save.

EditPad automatically compresses attachments in memory before uploading them. So if you want to attach an external file, there's no need to compress it using a zip program first. If you compress the file manually, everybody who wants to open it has to decompress it manually. If you let EditPad compress it automatically, decompression is also automatic.

Saying Thanks or Me Too and Starring Conversations

The +1 button lets you say "thanks" or "me too" for the message that you are presently reading. This is a quick way to show your appreciation or agreement without having to post a reply. Everybody can see how many people gave a +1 to a particular message. But nobody can see who gave those +1. You can only see

whether one of those +1 came from you or not. The +1 column in the list of messages shows a number to indicate the total number of people (possibly including you) that gave a +1 to that message. The +1 column shows a + before the number if one of those +1 came from you.

The list of conversations also has a +1 column. The number in this column indicates the total number of different people who gave a +1 to one or more messages in the conversation. The number of +1 for the conversation will be less than the sum of the +1 of all messages in the conversation if one person gave a +1 to multiple messages in the conversation. The +1 column for conversations shows a + before the number if you gave a +1 to any of the messages in that conversation.

You can click the +1 column header to sort conversations or messages by their +1. Conversations or messages that you gave a +1 are placed above conversations or messages that you did not give a +1. So you can also use the +1 feature to star or bookmark messages as sorting by +1 puts yours at the top. The conversations that you gave +1 are sorted among themselves by decreasing number of total +1. Below all those, the remaining conversations are sorted by their total +1.

Another way to save a conversation for later is to click the Reply button without clicking the Send button. Conversations with unsent replies are always sorted at the top. They are never hidden when filtering the forum. Nobody but you can see your unsent reply. Replies don't touch the server until you click the Send button. Unsent replies persist when you close and restart EditPad.

Taking Back Your Words

If you regret anything you wrote, simply delete it. There are three Delete buttons. The one above the list of conversations deletes the whole conversation. You can only delete a conversation if nobody else participated in it. The Delete button above the edit box for the message body deletes that message, if you wrote it. It is labeled Cancel if you have not yet sent your message. The Delete button above the list of attachments deletes the selected attachment, if it belongs to a message that you wrote.

If somebody already downloaded your message before you got around to deleting it, it won't vanish instantly. The message will disappear from their view of the forums the next time they log onto the forums or click Refresh. If you see messages disappear when you refresh your own view of the forums, that means the author of the message deleted it. If you replied to a conversation and the original question disappears, leaving your reply as the only message, you should delete your reply too. Otherwise, your reply will look silly all by itself. When you delete the last reply to a conversation, the conversation itself is also deleted, whether you started it or not.

Changing Your Opinion

If you think you could better phrase something you wrote earlier, select the message and then click the Edit button above the message text. You can then edit the subject and/or body text of the message. Click the Send button to publish the edited message. It will replace the original. If you change your mind about editing the message, click the Cancel button. Make sure to click it only once! When editing a message, the Delete button changes its caption to Cancel and when clicked reverts the message to what it was before you started editing it. If you click Delete a second time (i.e. while the message is no longer being edited), you'll delete the message from the forum.

If other people have already downloaded your message, their view of the message will magically change when they click Refresh or log in again. Since things may get confusing if people respond to your original message before they see the edited message, it's best to restrict your edits to minor errors like spelling mistakes. If you change your opinion, click the Reply button to add a new message to the same conversation.

Updating Your View

When you click the Login button, EditPad automatically downloads all new conversations and message summaries. Message bodies are downloaded one conversation at a time as you click on the conversations. Attachments are downloaded individually when you click the Use or Save button.

EditPad keeps a cache of conversations and messages that persists when you close EditPad. Attachments are cached while EditPad is running, and discarded when you close EditPad. By caching conversations and messages, EditPad improves the responsiveness of the forum while reducing the stress on the forum server.

If you keep EditPad running for a long time, EditPad does not automatically check for new conversations and messages. To do so, click the Refresh button.

Whenever you click Login or Refresh, all conversations and messages are marked as “read”. They won't have any special indicator in the list of conversations or messages. If the refresh downloads new conversation and message summaries, those are marked as “unread”. This is indicated with the same “people in the cloud” icon as shown next to the Login button.

Forum RSS Feeds

When you're connected to the user forum, you can click the Feeds button to select RSS feeds that you can add to your favorite feed reader. This way, you can follow EditPad's discussion forums as part of your regular reading, without having to start EditPad. To participate in the discussions, simply click on a link in the RSS feed. All links in EditPad's RSS feeds will start EditPad and present the forum login screen. After you log in, wait a few moments for EditPad to download the latest conversations. EditPad will automatically select the conversation or message that the link points to. If EditPad was already running and you were already logged onto the forums, the conversation or message that the link points to is selected immediately.

You can choose which conversations should be included in the RSS feed:

- All conversations in all groups: show all conversations in all the discussion groups that you can access in EditPad.
- All conversations in the selected group: show the list of conversations that EditPad is presently showing on the Forum window.
- All conversations you participated in: show all conversations that you started or replied to in all the discussion groups that you can access in EditPad.
- All conversations you started: show all conversations that you started in all the discussion groups that you can access in EditPad.
- Only the selected conversation: show only the conversation that you are presently reading on the Forum window in EditPad.

In addition, you can choose how the conversations that you want in your RSS feed should be arranged into items or entries in the feed:

- One item per group, with a list of conversations: Entries link to groups as a whole. The entry titles show the names of the groups. The text of each entry shows a list of conversation subjects and dates. You can click the subjects to participate in the conversation in EditPad. Choose this option if you prefer to read discussions in EditPad itself (with instant access to attachments), and you only want your RSS reader to tell you if there's anything new to be read.
- One item per conversation, without messages: Entries link to conversations. The entry titles show the subjects of the conversations. The entry text shows the date the conversation was started and last replied to. If your feed has conversations from multiple groups, those will be mixed among each other.
- One item per conversation, with a list of messages: Entries link to conversations. The entry titles show the subjects of the conversations. The entry text shows the list of replies with their summaries, author names, and dates. You can click a message summary to read the message in EditPad. If your feed has conversations from multiple groups, those will be mixed among each other.
- One item per conversation, with a list of messages: Entries link to conversations. The entry titles show the subjects of the conversations. The entry text shows the list of replies, each with their full text. If your feed has conversations from multiple groups, those will be mixed among each other. This is the best option if you want to read full discussions in your RSS reader.
- One item per message with its full text: Entries link to messages (responses to conversations). The entry titles show the message summary. The entry text shows the full text of the reply, and the conversation subject that you can click on to open the conversation in EditPad. If your feed lists multiple conversations, replies to different conversations are mixed among each other. Choose this option if you want to read full discussions in your RSS reader, but your RSS reader does not mark old entries as unread when they are updated in the RSS feed.

Help | Support and Feedback

Before requesting technical support, please use the version history to see whether you are using the latest version of EditPad. We take pride in quickly fixing bugs and resolving problems in free minor updates. If you encounter a problem with EditPad, it is quite possible that we have already released a new version that no longer has this problem.

EditPad has a built-in user forum that allows you to easily communicate with other EditPad users. If you're having a technical problem with EditPad, you're likely not the only one. The problem may have already been discussed on the forums. So search there first and you may get an immediate answer. If you don't see your issue discussed, feel free to start a new conversation in the forum. Other EditPad users will soon chime in, probably even before a Just Great Software technical support person sees it.

However, if you have purchased EditPad, you are entitled to free technical support via email. To request technical support, send an email to support@editpadlite.com if you're using EditPad Lite, or support@editpadpro.com if you're using EditPad Pro. You can expect to receive a reply by the next business day. For instant gratification, try the forums.

Feature Requests and Other Feedback

If you have any comments about EditPad, good or bad, suggestions for improvements, please do not hesitate to send them to our technical support department. Or better yet: post them to the forum so other EditPad users can add their vote. While we cannot implement each and every user wish, we do take all feedback into account when developing new versions of our software. Customer feedback is an essential part of Just Great Software.

Help | Create Portable Installation

The Create Portable Installation item in the Help menu makes it easy to create a portable copy of EditPad Pro on a removable disk, USB stick or flash memory card. You can run EditPad off the removable device on any computer, without leaving any traces of EditPad on that computer.

If the capacity of the device you want to install EditPad onto is limited, you can choose not to install certain parts of EditPad on the device. The only part that is not optional is the “main application” part, which is EditPad itself. You can select if you want to copy over the help file, the syntax coloring schemes, and the file navigation schemes. If you installed one or more spell checker dictionaries, you can choose which ones you want to make available on the device.

Note that the Create Portable Installation command will not delete any files from the device. If you use the Create Portable Installation command a second time, and select fewer parts than the first time, EditPad will *not* remove the deselected parts from the device.

EditPad will show you the complete list of devices that Windows reports as removable devices. External hard disks often report themselves as being hard disks rather than being removable devices. If you want to install EditPad in a portable manner on such a device, turn on the option to treat all drives as removable drives. EditPad will then create a file RemovableDrive.sys on the destination drive. This file act as a token to tell EditPad not to touch the host computer.

To proceed with the installation, click on the device you want to install onto, and click the Install button. The label showing the amount of disk space needed will indicate the parts being copied to the removable drive. Once the operation is complete, the Install onto Removable Drive screen will close automatically.

EditPad will copy itself to a fixed EditPadLite8 or EditPadPro8 subfolder off the root on the device. If you like more complex folder structures, you can move the complete EditPadLite8 or EditPadPro8 folder into a different parent folder on the device.

Creating Portable Installations Using The Installer

If you have not yet installed EditPad Pro onto your hard disk, you can create a portable installation by running EditPad’s installer. On the welcome screen, click the Portable Installation button. Follow the steps. This method works even if you do not have the necessary permissions to install software onto the computer you’re using.

If you run the installer on 32-bit Windows the portable install will be 32-bit. It will work on both 32-bit and 64-bit Windows. If you run the installer on 64-bit Windows, then the portable install will be 64-bit. It will

only work on 64-bit Windows. You can pass the `/32` command line parameter to the installer to force it to create a 32-bit installation on 64-bit Windows.

Restrictions on Portable Installs

Portable installs are designed to not leave any trace on the host PC, other than files that you explicitly save to the host PC. All settings and history are stored in the same folder as EditPad itself.

This means that a few of EditPad's features are disabled in portable installs. The Preferences dialog does not have its Shortcuts page. In the Configure File Types dialog, on the Definition page, the buttons to create and remove file associations are disabled.

Changing Drive Letters

When you connect a portable drive to different PCs, it may be assigned different drive letters on those PCs. A portable install of EditPad automatically handles a change of drive letter each time you run it. All file paths in your settings and history are automatically updated to the new drive letter when you start EditPad. So when saving files, configuring tools, or changing other settings that refer to files, simply use absolute paths using your portable drive's current drive letter to refer to tools or files on that drive.

24. Customizing Toolbars and Menus

EditPad's main menu, all toolbars, and most context menus are fully configurable. You can move toolbars, hide them, add and remove items, and even change item captions.

EditPad's interface is modular. All functionality other than the main editor is placed on side panels that you can rearrange and even drag off as floating windows.

Use the Custom Layouts item in the View menu to save and load your customizations.

Menu and Toolbar Appearance

You can switch between small, medium, and large toolbar icons by right-clicking any toolbar and selecting the Small Icons, Medium Icons, or Large Icons option. Toolbars increase their height and, if needed, wrap buttons across multiple rows to accommodate larger icons. The main menu always displays small icons next to menu items.

You can a different font face or font size for the menus by right-clicking the main menu or any toolbar and selecting Menu Font. Your chosen font is used by the main menu as well as all drop-down menus and all right-click menus. It is also used by toolbar buttons with text labels, such as the search option buttons.

Moving The Menu and Toolbars

Moving the menu and toolbars can be done directly by dragging them with the mouse. First, you need to make sure the toolbars aren't locked. Right-click on the main menu or any toolbar and make sure Lock Toolbars is not ticked.

To start dragging a toolbar, click on the dotted line at the left-hand edge of the toolbar. While holding down the mouse button, move the mouse pointer to the location where you want to place the toolbar. The toolbar will automatically snap into place when the mouse pointer is at a position where you can dock the toolbar. If you release the mouse button, the toolbar will stay docked. If you don't release the mouse button, the toolbar will continue to be dragged. If the mouse pointer is not at a position where you can dock the toolbar, the toolbar will float on top of EditPad's window.

When a toolbar is floating, you can resize it like any window. If you double-click its caption, the toolbar will snap back to the place where it was docked last time.

You can dock the main menu and all toolbars at various positions. You can dock them at all 4 edges of EditPad's window. If side panels are visible, you can also dock them below the caption or tab of the visible side panels. If tabs or side panels are visible that take up space between the edge of EditPad's window and the main editor (the area where you edit your files), then you can also dock toolbars at any of the 4 edges of the editor that are not adjacent to the edges of EditPad's window.

You can dock multiple toolbars at the same position and stack them horizontally or vertically or both ways. Simply drag one toolbar to the same position as another toolbar. The dragged toolbar will automatically stack itself with the other one. Exactly how it stacks itself depends on whether you move the mouse pointer near the top or bottom of the other toolbar, or over the blank space at the end of the toolbar.

If you find yourself accidentally moving toolbars around, right-click on the main menu or any toolbar and select Lock Toolbars. Then you will no longer be able to move docked toolbars around. Floating toolbars can still be moved. Select Lock Toolbars again if you want to be able to move the toolbars again.

By default, the main menu and the main toolbar are docked at the top. The search toolbar is docked at the bottom, or at the top of the search panel when the search panel is visible. Most side panels have their own toolbars that are docked below the tab or caption of the panel.

Showing and Hiding The Menu and Toolbars

To show or hide the main menu or any toolbar, right-click on the main menu bar or any visible toolbar. The context menu will give you a list of all available toolbars. Click one to show or hide it. Toolbars that belong to side panels won't be available unless the side panel is visible. Use the View menu to open the side panels.

The main menu and the main toolbar cannot both be hidden. If you hide one while the other is hidden, the other one will become visible again. All other toolbars can be shown or hidden independently.

The Tools toolbar is a special toolbar. You can show or hide it and move it around, but you cannot directly customize its contents like you can with the other toolbars. The Tools toolbar lists the tools that you have selected to appear on the Tools toolbar via Tools | Configure Tools.

Adding and Removing items from The Main Menu, Toolbars, and Context Menus

In EditPad, the main menu is really just another toolbar. The only difference is that by default it contains menu items rather than buttons. But you can place buttons directly on the main menu, and you can place menu items on toolbars and even in context menus. There is no difference between menu items and toolbar buttons. Each command can be a menu item or a toolbar button, depending on whether you place it into a menu or onto a toolbar.

To edit any toolbar or menu, right-click on any toolbar or the main menu and select Customize. The Customization screen will pop up.

To move a menu item or button from one toolbar to another, left-click on the item, hold down the mouse button, and drag the item to its new location. If you want to move an item into a submenu, move it to the menu, wait for the menu to expand while holding down the mouse button, and then move the button you're dragging to its location. To duplicate a menu item or button, hold down the Control key on the keyboard and then drag the item to its new location.

To remove a menu item or button, left-click the item and drag it to any position on the screen that is not an EditPad toolbar or menu. Then release the mouse pointer. Or, right-click the item and select Delete.

To add a separator line between menu items or toolbar buttons, first add the items that you want to separate to the menu or toolbar. Then right-click on the item that goes after the separator, and select Begin a Group.

If you want to bring back a menu item or button that you previously removed, click on the Commands tab in the Customization screen. The commands are organized into categories that correspond to the menus that

contain those commands by default. The categories in the Customization screen and the items they contain never change.

Pay attention to similarly named commands in different categories. For example, you can find an Open command in the File category, the FTP Panel category, and the History Panel category. These are 3 unique commands. The File, Open command sits in the file menu and on the main toolbar by default. It shows a dialog box for opening files. The FTP Panel, Open command sits on the middle toolbar on the FTP panel. It opens the file that you have selected in the FTP panel. The File History, Open command sits on the toolbar at the top of the History Panel. It opens the file that you have selected in the History panel. You can technically place these 3 commands anywhere. But it's probably not a good idea to use the Open items for the side panels anywhere but the proper side panel toolbar.

While customizing the toolbars and menus, an extra toolbar appears docked at the top edge of EditPad's window, below the main menu and main toolbar if they're still in their default positions. This toolbar has 6 drop-down menus (8 in EditPad Pro) that hold the items for EditPad's most important context menus. Do not customize the toolbar itself. EditPad will automatically reset it. The purpose of this toolbar is to give you access to the context menus while customizing the toolbars and menus, so you can customize the context menus too. When you're not customizing, the context menus appear in the following situations:

- Editor (outside selection): Right-click on any text in the main editor that is not selected. This menu lists commands that affect the whole file by default. If you find it confusing that you can't copy text when clicking outside the selection, add the copy commands to this menu too.
- Editor (inside selection): Right-click on selected text in the main editor. This menu lists commands for copying and editing the selected text.
- Editor (margin): Right-click on the left-hand margin that shows line numbers, bookmarks, and/or folding ranges in the main editor. This menu has commands for bookmarks, folding, and line numbers.
- File tabs: Right-click on any tab for a file. This menu lists commands for saving and closing the file and setting some of its options.
- File tabs background: Right-click on the blank space after the last file tab. If there is no blank space, right-click on the scrolling arrows after the rightmost visible tab. This menu lists commands for saving and closing all files, as well as opening more files.
- Project tabs: Right-click on any tab for a project. This menu lists commands for saving and closing the project and managing the files in it.
- Project tabs background: Right-click on the blank space after the last project tab. If there is no blank space, right-click on the scrolling arrows after the rightmost visible tab. This menu lists commands for saving and closing all projects, as well as opening more projects.
- Notification icon: If you have enabled the notification icon in the System Preferences, right-clicking that icon shows this menu. It has commands for restoring and exiting EditPad, and for opening files and projects.

Editing Menu Item and Toolbar Button Captions

Before you can edit menu items and toolbar buttons, you need to be customizing the toolbars and menus. To do so, right-click any toolbar or menu and select Customize.

To change the caption of any menu item, right-click on the menu item. In the context menu, left-click on the Name item. Then type in the new name. Press Enter to confirm the change. If you don't press Enter, the change won't stick.

To change the Alt+letter access key shortcut of a menu item, edit its name. In the new name, change the position of the ampersand. The letter after the last ampersand in the name becomes the Alt+letter access key shortcut. Use two ampersands if you want to have a literal ampersand in the name.

If you place a command that has an icon associated with it on a toolbar, the toolbar button will only show the icon by default. If you find some of the icons confusingly similar, right-click on the button and select “image and text” or “text only (always)”. Then the button will show the command’s caption, with or without the image. Commands that don’t have an icon can also be placed on toolbars. They will show their caption. The default captions are intended for the menus. They’re probably longer than you want for toolbars. You can change the caption of a button by right-clicking it, left-clicking Name, typing in the new caption, and pressing Enter.

If you want to restore the default caption of an item or button, right-click it and select Reset.

If you create multiple toolbar buttons and/or menu items for the same command, editing the caption of one of its items or buttons only affects that item or button. By default, the various search options are listed with their full captions in the Search Options submenu of the Search menu. They also appear as buttons with one-word captions on the Search toolbar.

Adding Toolbars

In the customization window, click on the Toolbars tab. This tab lists all available toolbars. Toolbars that belong to side panels won’t be available unless the side panel is visible. Use the View menu to open the side panels before customizing the toolbars. Tick or clear the checkbox next to a toolbar to show or hide it.

To add a toolbar, click the New button. The name you give the toolbar will appear in the context menu when you right-click any toolbar. The new toolbar will be blank initially. You can place buttons on it like you can on any other toolbar. Adding new toolbars can be useful if you want to have toolbar buttons in different places. If you have so many buttons on the main toolbar that it wraps into multiple lines, you may prefer to move some of the buttons onto additional toolbars and dock them left or right instead of having one fat toolbar at the top.

The default toolbars can be hidden but cannot be deleted. Only toolbars you’ve added yourself can be deleted.

25. Command Line Parameters

You can specify as many files as you want on the command line. EditPad will open all files. You should put double quotes around file names that contain spaces. Putting double quotes around file names without spaces makes no difference. If you specify a file that does not exist on the command line, EditPad creates a blank tab with that file name. The file itself is not created until you save it. You can use wildcards such as `*.txt` to make EditPad open all files that match the wildcard. You can use paths relative to the current directory.

Example: `EditPadPro8.exe "C:\My Documents\text.txt" C:\Development\source.c`

If you start EditPad while another copy of EditPad is already running then the newly run copy sends the command line parameters to the existing copy. The newly run copy closes itself as soon as the existing copy has processed the command line parameters and opened all files.

If you want to force a second EditPad window to appear, specify the `/newinstance` parameter on the command line. This parameter can appear anywhere on the command line. It can be used in combination with any other command line parameter.

When using `/newinstance`, you can also specify the location of the new EditPad window. `/br100t200r300b400` sets the window's bounding rectangle to left 100, top 200, right 300 and bottom 400, counting pixels from the top left corner of the screen.

Some applications that launch a text editor wait for the editor to close as a signal that you're done editing the file and that the application can proceed with whatever it was doing. Such applications won't behave correctly when EditPad reuses an existing instance and then closes as soon as the file has been opened. To avoid this problem you can specify the `/newinstance` parameter to start a new EditPad instance that the application can wait for. Alternatively, you can specify the `/wait` parameter to reuse the existing window as usual, but to make the newly run copy wait until you have closed the file in the existing instance. While waiting, the newly run instance appears as a separate process in the task manager, but is otherwise invisible.

Parameters Specific to One File

All of the parameters in this section require a single file to be specified on the command line. Each of them can appear only once on the command line. EditPad does not show any error messages if you do not respect these rules, but the result will not be what you expected. The order of the parameters does not matter.

`/p` tells EditPad to show the print preview immediately after opening the file, so you can print it with one click.

`/l123` (slash el one two three) tells EditPad to place the text cursor on line 123 of the file. The first line in the file is number one. The lines are counted as if word wrap were off. Negative numbers count from the end of the file. `/l-1` (slash el minus one) places the text cursor at the very end of the file.

`/c45` tells EditPad to place the text cursor at byte offset 45 in the file. Byte offset 0 is at the first character in the file. If the file is opened in text mode and the file has a byte order marker then the byte order marker is not included in the count.

If you use `/c` in combination with `/l`, then the meaning of `/c` changes. `/c7` places the text cursor at the 7th column on the line indicated by `/l`. The first column on the line is number 1.

Finally, `/s123-145` selects bytes 123 through 144 (inclusive) and places the text cursor at byte offset 145. You can select ranges in reverse. `/s145-123` also selects bytes 123 through 144 (inclusive) but places the cursor at byte offset 123. The high number in the range (145 in the examples) is always exclusive.

In combination with `/l`, `/s4-7` select columns 4 through 6 and puts the text cursor on the 7th column on the line indicated by `/l`. Column 1 is the first column on the line.

`/c` and `/s` cannot be used in combination with each other. If you specify `/s` then `/c` is ignored.

Some text file encodings use multiple bytes per character. The cursor cannot be placed between bytes that are part of the same character. Selections must include either none or all of the bytes that a character consist of. If you specify a byte offset in the middle of a character using `/c` or `/s` then EditPad places the cursor or ends the selection at the nearest valid byte offset between two characters.

Additional EditPad Pro Parameters

The parameters described above are supported by both EditPad Lite and Pro. EditPad Pro supports several additional parameters.

`/newproject` tells EditPad Pro to use Project|New Project to start with a new project to open all the files specified on the command line. This parameter only makes a difference when reusing an existing instance that already has open files.

`/newprojectcombine` does the same as `/newproject`, except that `/newprojectcombine` is ignored if another file was opened with this parameter less than one second ago. If you have an application or script that launches multiple EditPad Pro instances at the same time to open multiple files at the same time, the result of the `/newprojectcombine` parameter is that all those files will be opened into one new project.

`/hex` tells EditPad Pro to open the files specified on the command line in hexadecimal mode, regardless of whether you set this option in the file type configuration or not.

`/readonly` tells EditPad Pro to open the files specified on the command line in read-only mode. Use this parameter if you want to make sure you don't accidentally modify the files in EditPad Pro. You can click the read-only indicator on the status bar to turn off read-only mode later.

`/import` shows the Project|Import File Listing screen. All the files specified on the command line are treated as file listings to be imported rather than as files to be opened.

`/folder` followed by a separate parameter with the full path to a folder makes EditPad Pro show the Project|Open Folder window with that folder preselected. You can suppress the dialog with `/masks` followed by a file mask. EditPad Pro then automatically opens all files in the folder that match the file mask. Specify `/masks *.*` to open all files in the folder. You can pass `/recurse` to include subfolders. You can pass `/closed` to add the files to the project without opening them. The `/newproject` and `/readonly` parameters also affect files opened from a folder.

Tailing Files

If you pass any of the following three parameters on the command line then EditPad Pro opens all the files on the command line in the same way that the File|Tail menu item does. The three parameters correspond with the three options on the Tail Files dialog box.

`/tailsize` followed by a positive number loads only that number of megabytes from the tail end of the file. Do not place a space before the number. `/tailsize7` loads the last 7 MB of the file. Omit this parameter to load the file entirely.

`/tailinterval` followed by a positive number continuously reloads the file when it is modified on disk while you are looking at the file at an interval of the given number of seconds. Do not place a space before the number. `/tailinterval60` reloads the file once a minute. Omit this parameter or pass `/tailinterval0` to avoid automatically reloading the file while you are looking at it.

`/tailappend` tells EditPad Pro the file is only ever appended to, which optimizes automatic and manual reloads. Omit this parameter to reload the entire file.

Opening Projects

You can pass the path to an `.epp` project file on the command line just like any other file. EditPad Pro recognizes `.epp` files as project files and opens them as a project. Command line parameters such as `/hex` or `/readonly` or the tailing parameters that determine how files are opened on the command line are not applied to files opened as part of a project. They are not applied to the project itself either.

Part 2

Regular Expression Tutorial

1. Regular Expressions Tutorial

This tutorial teaches you all you need to know to be able to craft powerful time-saving regular expressions. It starts with the most basic concepts, so that you can follow this tutorial even if you know nothing at all about regular expressions yet.

The tutorial doesn't stop there. It also explains how a regular expression engine works on the inside and alerts you to the consequences. This helps you to quickly understand why a particular regex does not do what you initially expected. It will save you lots of guesswork and head scratching when you need to write more complex regexes.

What Regular Expressions Are Exactly - Terminology

Basically, a regular expression is a pattern describing a certain amount of text. Their name comes from the mathematical theory on which they are based. But we will not dig into that. You will usually find the name abbreviated to "regex" or "regexp". This tutorial uses "regex", because it is easy to pronounce the plural "regexes". In this book, regular expressions are shaded gray as `regex`.

This first example is actually a perfectly valid regex. It is the most basic pattern, simply matching the literal text `regex`. A “match” is the piece of text, or sequence of bytes or characters that pattern was found to correspond to by the regex processing software. Matches are highlighted in blue in this book.

`\b[A-Z0-9._%+-]+\@[A-Z0-9.-]+\.[A-Z]{2,}\b` is a more complex pattern. It describes a series of letters, digits, dots, underscores, percentage signs and hyphens, followed by an at sign, followed by another series of letters, digits and hyphens, finally followed by a single dot and two or more letters. In other words: this pattern describes an email address. This also shows the syntax highlighting applied to regular expressions in this book. Word boundaries and quantifiers are blue, character classes are orange, and escaped literals are gray. You'll see additional colors like green for grouping and purple for meta tokens later in the tutorial.

With the above regular expression pattern, you can search through a text file to find email addresses, or verify if a given string looks like an email address. This tutorial uses the term “string” to indicate the text that the regular expression is applied to. This book highlights them in `green`. The term “string” or “character string” is used by programmers to indicate a sequence of characters. In practice, you can use regular expressions with whatever data you can access using the application or programming language you are working with.

2. Literal Characters

The most basic regular expression consists of a single literal character, such as `a`. It matches the first occurrence of that character in the string. If the string is `Jack is a boy`, it matches the `a` after the `J`. The fact that this `a` is in the middle of the word does not matter to the regex engine. If it matters to you, you will need to tell that to the regex engine by using word boundaries. We will get to that later.

This regex can match the second `a` too. It only does so when you tell the regex engine to start searching through the string after the first match. In a text editor, you can do so by using its “Find Next” or “Search Forward” function. In a programming language, there is usually a separate function that you can call to continue searching through the string after the previous match.

Similarly, the regex `cat` matches `cat` in `About cats and dogs`. This regular expression consists of a series of three literal characters. This is like saying to the regex engine: find a `c`, immediately followed by an `a`, immediately followed by a `t`.

Note that regex engines are case sensitive by default. `cat` does not match `Cat`, unless you tell the regex engine to ignore differences in case.

Special Characters

Because we want to do more than simply search for literal pieces of text, we need to reserve certain characters for special use. In EditPad there are 12 characters with special meanings: the backslash `\`, the caret `^`, the dollar sign `$`, the period or dot `.`, the vertical bar or pipe symbol `|`, the question mark `?`, the asterisk or star `*`, the plus sign `+`, the opening parenthesis `(`, the closing parenthesis `)`, the opening square bracket `[`, and the opening curly brace `{`. These special characters are often called “metacharacters”. Most of them are errors when used alone.

If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash. If you want to match `1+1=2`, the correct regex is `1\\+1=2`. Otherwise, the plus sign has a special meaning.

Note that `1+1=2`, with the backslash omitted, is a valid regex. So you won’t get an error message. But it doesn’t match `1+1=2`. It would match `111=2` in `123+111=234`, due to the special meaning of the plus character.

If you forget to escape a special character where its use is not allowed, such as in `+1`, then you will get an error message.

EditPad treats the brace `{` as a literal character, unless it is part of a repetition operator like `a{1,3}`. So you generally do not need to escape it with a backslash, though you can do so if you want.

`]` is a literal outside character classes. Different rules apply inside character classes. Those are discussed in the topic about character classes.

All other characters should not be escaped with a backslash. That is because the backslash is also a special character. The backslash in combination with a literal character can create a regex token with a special meaning. E.g. `\\d` is a shorthand that matches a single digit from `0` to `9`.

EditPad also supports the `\Q... \E` escape sequence. All the characters between the `\Q` and the `\E` are interpreted as literal characters. E.g. `\Q*\d+*\E` matches the literal text `*\d+*`. The `\E` may be omitted at the end of the regex, so `\Q*\d+*` is the same as `\Q*\d+*\E`.

3. Non-Printable Characters

You can use special character sequences to put non-printable characters in your regular expression. Use `\t` to match a tab character (ASCII 0x09), `\r` for carriage return (0x0D) and `\n` for line feed (0x0A). More exotic non-printables are `\a` (bell, 0x07), `\e` (escape, 0x1B), and `\f` (form feed, 0x0C). Remember that Windows text files use `\r\n` to terminate lines, while UNIX text files use `\n`.

You can use `\uFFFF` or `\x{FFFF}` to insert a Unicode character. The euro currency sign occupies Unicode code point U+20AC. If you cannot type it on your keyboard, you can insert it into a regular expression with `\u20AC` or `\x{20AC}`. See the tutorial section on Unicode for more details on matching Unicode code points.

When working with files in 8-bit code pages in EditPad, you can include any character in your regular expression if you know its position in the character set that you are working with. In the Latin-1 character set, the copyright symbol is character 0xA9. So to search for the copyright symbol, you can use `\xA9`. Another way to search for a tab is to use `\x09`. Note that the leading zero is required.

Line Breaks

`\R` is a special escape that matches any line break, including Unicode line breaks. What makes it special is that it treats CRLF pairs as indivisible. If the match attempt of `\R` begins before a CRLF pair in the string, then a single `\R` matches the whole CRLF pair. `\R` will not backtrack to match only the CR in a CRLF pair. So while `\R` can match a lone CR or a lone LF, `\R{2}` or `\R\R` cannot match a single CRLF pair. The first `\R` matches the whole CRLF pair, leaving nothing for the second one to match.

Note that `\R` only looks forward to match CRLF pairs. The regex `\r\R` can match a single CRLF pair. After `\r` has consumed the CR, the remaining lone LF is a valid line break for `\R` to match.

Octal Escapes

EditPad supports `\o{377}` for octal escapes. You can have any number of octal digits between the curly braces, with or without leading zero. There is no confusion with backreferences and literal digits that follow are cleanly separated by the closing curly brace. Do be careful to only put octal digits between the curly braces.

4. First Look at How a Regex Engine Works Internally

Knowing how the regex engine works enables you to craft better regexes more easily. It helps you understand quickly why a particular regex does not do what you initially expected. This saves you lots of guesswork and head scratching when you need to write more complex regexes.

After introducing a new regex token, this tutorial explains step by step how the regex engine actually processes that token. This inside look may seem a bit long-winded at certain times. But understanding how the regex engine works enables you to use its full power and help you avoid common mistakes.

While there are many implementations of regular expressions that differ sometimes slightly and sometimes significantly in syntax and behavior, there are basically only two kinds of regular expression engines: text-directed engines, and regex-directed engines. Nearly all modern regex flavors are based on regex-directed engines. This is because certain very useful features, such as lazy quantifiers and backreferences, can only be implemented in regex-directed engines.

A regex-directed engine walks through the regex, attempting to match the next token in the regex to the next character. If a match is found, the engine advances through the regex and the subject string. If a token fails to match, the engine *backtracks* to a previous position in the regex and the subject string where it can try a different path through the regex. This tutorial will talk a lot more about backtracking later on. Modern regex flavors using regex-directed engines have lots of features such as atomic grouping and possessive quantifiers that allow you to control this backtracking.

A text-directed engine walks through the subject string, attempting all permutations of the regex before advancing to the next character in the string. A text-directed engine never backtracks. Thus, there isn't much to discuss about the matching process of a text-directed engine. In most cases, a text-directed engine finds the same matches as a regex-directed engine.

When this tutorial talks about regex engine internals, the discussion assumes a regex-directed engine. It only mentions text-directed engines in situations where they find different matches. And that only really happens when your regex uses alternation with two alternatives that can match at the same position.

The Regex Engine Always Returns the Leftmost Match

This is a very important point to understand: a regex engine always returns the leftmost match, even if a “better” match could be found later. When applying a regex to a string, the engine starts at the first character of the string. It tries all possible permutations of the regular expression at the first character. Only if all possibilities have been tried and found to fail, does the engine continue with the second character in the text. Again, it tries all possible permutations of the regex, in exactly the same order. The result is that the regex engine returns the *leftmost* match.

When applying `cat` to `He captured a catfish for his cat.`, the engine tries to match the first token in the regex `c` to the first character in the match `H`. This fails. There are no other possible permutations of this regex, because it merely consists of a sequence of literal characters. So the regex engine tries to match the `c` with the `e`. This fails too, as does matching the `c` with the space. Arriving at the 4th character in the string, `c` matches `c`. The engine then tries to match the second token `a` to the 5th character, `a`. This succeeds too. But then, `t` fails to match `p`. At that point, the engine knows the regex cannot be matched starting at the 4th character in the string. So it continues with the 5th: `a`. Again, `c` fails to match here and the engine carries on.

At the 15th character in the string, **c** again matches **c**. The engine then proceeds to attempt to match the remainder of the regex at character 15 and finds that **a** matches **a** and **t** matches **t**.

The entire regular expression could be matched starting at character 15. The engine is “eager” to report a match. It therefore reports the first three letters of catfish as a valid match. The engine never proceeds beyond this point to see if there are any “better” matches. The first match is considered good enough.

In this first example of the engine’s internals, our regex engine simply appears to work like a regular text search routine. However, it is important that you can follow the steps the engine takes in your mind. In following examples, the way the engine works has a profound impact on the matches it finds. Some of the results may be surprising. But they are always logical and predetermined, once you know how the engine works.

5. Character Classes or Character Sets

With a “character class”, also called “character set”, you can tell the regex engine to match only one out of several characters. Simply place the characters you want to match between square brackets. If you want to match an a or an e, use `[ae]`. You could use this in `gr[ae]y` to match either `gray` or `grey`. Very useful if you do not know whether the document you are searching through is written in American or British English.

A character class matches only a single character. `gr[ae]y` does not match `graay`, `graey` or any such thing. The order of the characters inside a character class does not matter. The results are identical.

You can use a hyphen inside a character class to specify a range of characters. `[0-9]` matches a *single* digit between 0 and 9. You can use more than one range. `[0-9a-fA-F]` matches a single hexadecimal digit, case insensitively. You can combine ranges and single characters. `[0-9a-fxA-FX]` matches a hexadecimal digit or the letter X. Again, the order of the characters and the ranges does not matter.

Character classes are one of the most commonly used features of regular expressions. You can find a word, even if it is misspelled, such as `sep[ae]r[ae]te` or `li[cs]en[cs]e`. You can find an identifier in a programming language with `[A-Za-z_][A-Za-z_0-9]*`. You can find a C-style hexadecimal number with `0[xX][A-Fa-f0-9]+`.

Negated Character Classes

Typing a caret after the opening square bracket negates the character class. The result is that the character class matches any character that is *not* in the character class. Unlike the dot, negated character classes also match (invisible) line break characters. If you don’t want a negated character class to match line breaks, you need to include the line break characters in the class. `[^0-9\r\n]` matches any character that is not a digit or a line break.

It is important to remember that a negated character class still must match a character. `q[^u]` does *not* mean: “a q not followed by a u”. It means: “a q followed by a character that is not a u”. It does not match the q in the string `Iraq`. It does match the q and the space after the q in `Iraq is a country`. Indeed: the space becomes part of the overall match, because it is the “character that is not a u” that is matched by the negated character class in the above regexp. If you want the regex to match the q, and only the q, in both strings, you need to use negative lookahead: `q(?:!u)`. But we will get to that later.

Metacharacters Inside Character Classes

The only special characters or metacharacters inside a character class are the closing bracket `]`, the backslash `\`, the caret `^`, and the hyphen `-`. The usual metacharacters are normal characters inside a character class, and do not need to be escaped by a backslash. To search for a star or plus, use `[+*]`. Your regex will work fine if you escape the regular metacharacters inside a character class, but doing so significantly reduces readability.

To include a backslash as a character without any special meaning inside a character class, you have to escape it with another backslash. `[\\x]` matches a backslash or an x. The closing bracket `]`, the caret `^` and the hyphen `-` can be included by escaping them with a backslash, or by placing them in a position where they do not take on their special meaning.

To include an unescaped caret as a literal, place it anywhere except right after the opening bracket. `[x^]` matches an x or a caret.

You can include an unescaped closing bracket by placing it right after the opening bracket, or right after the negating caret. `[]x]` matches a closing bracket or an x. `[^]x]` matches any character that is not a closing bracket or an x.

The hyphen can be included right after the opening bracket, or right before the closing bracket, or right after the negating caret. Both `[-x]` and `[x-]` match an x or a hyphen. `[^ -x]` and `[^x -]` match any character that is not an x or a hyphen.

Many regex tokens that work outside character classes can also be used inside character classes. This includes character escapes, octal escapes, and hexadecimal escapes for non-printable characters. It also includes Unicode character escapes and Unicode properties. `[$\u20AC]` matches a dollar or euro sign.

Repeating Character Classes

If you repeat a character class by using the `?`, `*`, or `+` operators, you're repeating the entire character class. You're not repeating just the character that it matched. The regex `[0-9]+` can match `837` as well as `222`.

If you want to repeat the matched character, rather than the class, you need to use backreferences. `([0-9])\1+` matches `222` but not `837`. When applied to the string `833337`, it matches `3333` in the middle of this string. If you do not want that, you need to use lookahead.

Looking Inside The Regex Engine

As was mentioned earlier: the order of the characters inside a character class does not matter. `gr[ae]y` matches `grey` in `Is his hair grey or gray?`, because that is the *leftmost match*. We already saw how the engine applies a regex consisting only of literal characters. Now we'll see how it applies a regex that has more than one permutation. That is: `gr[ae]y` can match both `gray` and `grey`.

Nothing noteworthy happens for the first twelve characters in the string. The engine fails to match `g` at every step, and continues with the next character in the string. When the engine arrives at the 13th character, `g` is matched. The engine then tries to match the remainder of the regex with the text. The next token in the regex is the literal `r`, which matches the next character in the text. So the third token, `[ae]` is attempted at the next character in the text (`e`). The character class gives the engine two options: match `a` or match `e`. It first attempts to match `a`, and fails.

But because we are using a regex-directed engine, it must continue trying to match all the other permutations of the regex pattern before deciding that the regex cannot be matched with the text starting at character 13. So it continues with the other option, and finds that `e` matches `e`. The last regex token is `y`, which can be matched with the following character as well. The engine has found a complete match with the text starting at character 13. It returns `grey` as the match result, and looks no further. Again, the *leftmost match* is returned, even though we put the `a` first in the character class, and `gray` could have been matched in the string. But the engine simply did not get that far, because another equally valid match was found to the left of it. `gray` is only matched if you tell the regex engine to continue looking for a second match in the remainder of the subject string after the first match.

6. Character Class Subtraction

Character class subtraction makes it easy to match any single character present in one list (the character class), but not present in another list (the subtracted class). The syntax for this is `[class-[subtract]]`. If the character after a hyphen is an opening bracket, these flavors interpret the hyphen as the subtraction operator rather than the range operator. You can use the full character class syntax within the subtracted character class.

The character class `[a-z-[aeiou]]` matches a single letter that is not a vowel. In other words: it matches a single consonant. Without character class subtraction or intersection, the only way to do this would be to list all consonants: `[b-df-hj-np-tv-z]`.

The character class `[\p{Nd}-[^\p{IsThai}]]` matches any single Thai digit. The base class matches any Unicode digit. All non-Thai characters are subtracted from that class. `[\p{Nd}-[P{IsThai}]]` does the same. `[\p{IsThai}-[^\p{Nd}]]` and `[\p{IsThai}-[P{Nd}]]` also match a single Thai digit by subtracting all non-digits from the Thai characters.

Nested Character Class Subtraction

Since you can use the full character class syntax within the subtracted character class, you can subtract a class from the class being subtracted. `[0-9-[0-6-[0-3]]]` first subtracts 0-3 from 0-6, yielding `[0-9-[4-6]]`, or `[0-37-9]`, which matches any character in the string `0123789`.

The class subtraction must always be the last element in the character class. `[0-9-[4-6]a-f]` is not a valid regular expression. It should be rewritten as `[0-9a-f-[4-6]]`. The subtraction works on the whole class. E.g. `[\p{Ll}\p{Lu}-[\p{IsBasicLatin}]]` matches all uppercase and lowercase Unicode letters, except any ASCII letters. The `\p{IsBasicLatin}` is subtracted from the combination of `\p{Ll}\p{Lu}` rather than from `\p{Lu}` alone. This regex will not match `abc`.

While you can use nested character class subtraction, you cannot subtract two classes sequentially. To subtract ASCII characters and Greek characters from a class with all Unicode letters, combine the ASCII and Greek characters into one class, and subtract that, as in `[\p{L}-[\p{IsBasicLatin}\p{IsGreek}]]`.

Negation Takes Precedence over Subtraction

The character class `[^1234-[3456]]` is both negated and subtracted from. In all flavors that support character class subtraction, the base class is negated before it is subtracted from. This class should be read as “(not 1234) minus 3456”. Thus this character class matches any character other than the digits 1, 2, 3, 4, 5, and 6.

7. Character Class Intersection

Character class intersection makes it easy to match any single character that must be present in two sets of characters. The syntax for this is `[class&&intersect]`. You can use the full character class syntax within the intersected character class.

You cannot omit the nested square brackets in PowerGREP. If you do, PowerGREP interprets the ampersands as literals. So in PowerGREP `[class&&intersect]` is a character class containing only literals, just like `[clas&inter]`.

The character class `[a-z&&[^aeiou]]` matches a single letter that is not a vowel. In other words: it matches a single consonant. Without character class subtraction or intersection, the only way to do this would be to list all consonants: `[b-df-hj-np-tv-z]`.

The character class `[\p{Nd}&&[\p{IsThai}]]` matches any single Thai digit. `[\p{IsThai}&&[\p{Nd}]]` does exactly the same.

Intersection of Multiple Classes

You can intersect the same class more than once. `[0-9&&[0-6&&[4-9]]]` is the same as `[4-6]` as those are the only digits present in all three parts of the intersection.

PowerGREP does not allow anything after the nested `]`. The characters `56` in `[0-9&&[12]56]` are an error.

You also shouldn't put `&&` at the very start or very end of the regex. PowerGREP treats leading and trailing `&&` as literal ampersands.

Intersection in Negated Classes

The character class `[^1234&&[3456]]` is both negated and intersected. In EditPad, negation takes precedence over intersection. EditPad reads this regex as “(not 1234) and 3456”. Thus in EditPad this class is the same as `[56]` and matches the digits 5 and 6.

9. The Dot Matches (Almost) Any Character

In regular expressions, the dot or period is one of the most commonly used metacharacters. Unfortunately, it is also the most commonly misused metacharacter.

The dot matches a single character, without caring what that character is. The only exception are line break characters. In all regex flavors discussed in this tutorial, the dot does *not* match line breaks by default.

This exception exists mostly because of historic reasons. The first tools that used regular expressions were line-based. They would read a file line by line, and apply the regular expression separately to each line. The effect is that with these tools, the string could never contain line breaks, so the dot could never match them.

Line Break Characters

In EditPad the dot recognizes the newline `\n`, the carriage return `\r`, the form feed `\f`, the vertical tab `\x0B`, the Latin-1 next line control character `\u0085`, the Unicode line separator `\u2028` and the Unicode paragraph separator `\u2029` as line break characters. The dot does not match any of these characters unless you turn on the “dot matches line breaks” option. The dot treats a CRLF pair as two separate characters.

`\N` Never Matches Line Breaks

`\N` matches any single character that is not a line break, just like the dot does. Unlike the dot, `\N` is not affected by the “dot matches line breaks” option. `(?s)\N.` turns on single-line mode and then matches any character that is not a line break followed by any character regardless of whether it is a line break.

Use The Dot Sparingly

The dot is a very powerful regex metacharacter. It allows you to be lazy. Put in a dot, and everything matches just fine when you test the regex on valid data. The problem is that the regex also matches in cases where it should not match. If you are new to regular expressions, some of these cases may not be so obvious at first.

Let’s illustrate this with a simple example. Say we want to match a date in mm/dd/yy format, but we want to leave the user the choice of date separators. The quick solution is `\d\d.\d\d.\d\d`. Seems fine at first. It matches a date like `02/12/03` just fine. Trouble is: `02512703` is also considered a valid date by this regular expression. In this match, the first dot matched `5`, and the second matched `7`. Obviously not what we intended.

`\d\d[- /.]\d\d[- /.]\d\d` is a better solution. This regex allows a dash, space, dot and forward slash as date separators. Remember that the dot is not a metacharacter inside a character class, so we do not need to escape it with a backslash.

This regex is still far from perfect. It matches `99/99/99` as a valid date. `[01]\d[- /.][0-3]\d[- /.]\d\d` is a step ahead, though it still matches `19/39/99`. How perfect you want your regex to be depends on what you want to do with it. If you are validating user input, it has to be perfect. If you are parsing data files from a known source that generates its files in the same way every time, our last attempt is

probably more than sufficient to parse the data without errors. You can find a better regex to match dates in the example section.

Use Negated Character Classes Instead of the Dot

A negated character class is often more appropriate than the dot. The tutorial section that explains the repeat operators star and plus covers this in more detail. But the warning is important enough to mention it here as well. Again let's illustrate with an example.

Suppose you want to match a double-quoted string. Sounds easy. We can have any number of any character between the double quotes, so `".*"` seems to do the trick just fine. The dot matches any character, and the star allows the dot to be repeated any number of times, including zero. If you test this regex on `Put a "string" between double quotes`, it matches `"string"` just fine. Now go ahead and test it on `Houston, we have a problem with "string one" and "string two". Please respond.`

Ouch. The regex matches `"string one"` and `"string two"`. Definitely not what we intended. The reason for this is that the star is *greedy*.

In the date-matching example, we improved our regex by replacing the dot with a character class. Here, we do the same with a negated character class. Our original definition of a double-quoted string was faulty. We do not want any number of *any character* between the quotes. We want any number of characters that are not double quotes or newlines between the quotes. So the proper regex is `"[^\r\n]*"`. If your flavor supports the shorthand `\v` to match any line break character, then `"[^\v]*"` is an even better solution.

10. Start of String and End of String Anchors

Thus far, we have learned about literal characters, character classes, and the dot. Putting one of these in a regex tells the regex engine to try to match a single character.

Anchors are a different breed. They do not match any character at all. Instead, they match a position before, after, or between characters. They can be used to “anchor” the regex match at a certain position. The caret `^` matches the position before the first character in the string. Applying `^a` to `abc` matches `a`. `^b` does not match `abc` at all, because the `b` cannot be matched right after the start of the string, matched by `^`. See below for the inside view of the regex engine.

Similarly, `$` matches right after the last character in the string. `c$` matches `c` in `abc`, while `a$` does not match at all.

A regex that consists solely of an anchor can only find zero-length matches. This can be useful, but can also create complications that are explained near the end of this tutorial.

Using `^` and `$` as Start of Line and End of Line Anchors

If you have a string consisting of multiple lines, like `first line\nsecond line` (where `\n` indicates a line break), it is often desirable to work with lines, rather than the entire string. Therefore, most regex engines have the option to expand the meaning of both anchors. `^` can then match at the start of the string (before the `f` in the above string), as well as after each line break (between `\n` and `s`). Likewise, `$` still matches at the end of the string (after the last `e`), and also before every line break (between `e` and `\n`).

In EditPad, the caret and dollar always match at the start and end of each line. This makes sense because EditPad is designed to work with entire files, rather than short strings.

Line Break Characters

Just like the dot, anchors in EditPad recognize the newline `\n`, the carriage return `\r`, the form feed `\f`, the vertical tab `\x0B`, the Latin-1 next line control character `\u0085`, the Unicode line separator `\u2028` and the Unicode paragraph separator `\u2029` as line break characters. In addition, the anchors treat CRLF as an indivisible pair. `^` matches after CRLF and `$` matches before CRLF, but neither match in the middle of a CRLF pair.

Permanent Start of String and End of String Anchors

`\A` only ever matches at the start of the string. Likewise, `\Z` only ever matches at the end of the string. These two tokens never match at line breaks. This is true in all regex flavors discussed in this tutorial, even when you turn on “multiline mode”. In EditPad Pro and PowerGREP, where the caret and dollar always match at the start and end of lines, `\A` and `\Z` only match at the start and the end of the entire file.

Strings Ending with a Line Break

Because Perl returns a string with a newline at the end when reading a line from a file, Perl's regex engine matches `$` at the position before the line break at the end of the string even when multi-line mode is turned off. Perl also matches `$` at the very end of the string, regardless of whether that character is a line break. So `^\d+$` matches `123` whether the subject string is `123` or `123\n`.

Most modern regex flavors have copied this behavior. That includes EditPad.

`\Z` also matches before the final line break. If you only want a match at the absolute very end of the string, use `\z` (lowercase z instead of uppercase Z). `\A\d+\Z` does not match `123\n`. `\z` matches after the line break, which is not matched by the shorthand character class.

Strings Ending with Multiple Line Breaks

If a string ends with multiple line breaks and multi-line mode is off then `$` only matches before the last of those line breaks in all flavors where it can match before the final break. The same is true for `\Z` regardless of multi-line mode.

Boost is the only exception. In Boost, `\Z` can match before any number of trailing line breaks as well as at the very end of the string. So if the subject string ends with three line breaks, Boost's `\Z` has four positions that it can match at. Like in all other flavors, Boost's `\Z` is independent of multi-line mode. Boost's `$` only matches at the very end of the string when you turn off multi-line mode (which is on by default in Boost).

Looking Inside The Regex Engine

Let's see what happens when we try to match `^4$` to `749\n486\n4` (where `\n` represents a newline character) in multi-line mode. As usual, the regex engine starts at the first character: `7`. The first token in the regular expression is `^`. Since this token is a zero-length token, the engine does not try to match it with the character, but rather with the position before the character that the regex engine has reached so far. `^` indeed matches the position before `7`. The engine then advances to the next regex token: `4`. Since the previous token was zero-length, the regex engine does *not* advance to the next character in the string. It remains at `7`. `4` is a literal character, which does not match `7`. There are no other permutations of the regex, so the engine starts again with the first regex token, at the next character: `4`. This time, `^` cannot match at the position before the `4`. This position is preceded by a character, and that character is not a newline. The engine continues at `9`, and fails again. The next attempt, at `\n`, also fails. Again, the position before `\n` is preceded by a character, `9`, and that character is not a newline.

Then, the regex engine arrives at the second `4` in the string. The `^` can match at the position before the `4`, because it is preceded by a newline character. Again, the regex engine advances to the next regex token, `4`, but does not advance the character position in the string. `4` matches `4`, and the engine advances both the regex token and the string character. Now the engine attempts to match `$` at the position before (indeed: before) the `8`. The dollar cannot match here, because this position is followed by a character, and that character is not a newline.

Yet again, the engine must try to match the first token again. Previously, it was successfully matched at the second **4**, so the engine continues at the next character, **8**, where the caret does not match. Same at the **6** and the newline.

Finally, the regex engine tries to match the first token at the third **4** in the string. With success. After that, the engine successfully matches **4** with **4**. The current regex token is advanced to **\$**, and the current character is advanced to the very last position in the string: the void after the string. No regex token that needs a character to match can match here. Not even a negated character class. However, we are trying to match a dollar sign, and the mighty dollar is a strange beast. It is zero-length, so it tries to match the position before the current character. It does not matter that this “character” is the void after the string. In fact, the dollar checks the current character. It must be either a newline, or the void after the string, for **\$** to match the position before the current character. Since that is the case after the example, the dollar matches successfully.

Since **\$** was the last token in the regex, the engine has found a successful match: the last **4** in the string.

11. Word Boundaries

The metacharacter `\b` is an anchor like the caret and the dollar sign. It matches at a position that is called a “word boundary”. This match is zero-length.

There are three different positions that qualify as word boundaries:

- Before the first character in the string, if the first character is a word character.
- After the last character in the string, if the last character is a word character.
- Between two characters in the string, where one is a word character and the other is not a word character.

Simply put: `\b` allows you to perform a “whole words only” search using a regular expression in the form of `\bword\b`. A “word character” is a character that can be used to form words. All characters that are not “word characters” are “non-word characters”.

In EditPad, characters that are matched by the short-hand character class `\w` are the characters that are treated as word characters by word boundaries.

Since digits are considered to be word characters, `\b4\b` can be used to match a 4 that is not part of a larger number. This regex does not match `44 sheets of a4`. So saying “`\b` matches before and after an alphanumeric sequence” is more exact than saying “before and after a word”.

`\B` is the negated version of `\b`. `\B` matches at every position where `\b` does not. Effectively, `\B` matches at any position between two word characters as well as at any position between two non-word characters.

Looking Inside The Regex Engine

Let’s see what happens when we apply the regex `\b is\b` to the string `This island is beautiful`. The engine starts with the first token `\b` at the first character `T`. Since this token is zero-length, the position before the character is inspected. `\b` matches here, because the `T` is a word character and the character before it is the void before the start of the string. The engine continues with the next token: the literal `i`. The engine does not advance to the next character in the string, because the previous regex token was zero-length. `i` does not match `T`, so the engine retries the first token at the next character position.

`\b` cannot match at the position between the `T` and the `h`. It cannot match between the `h` and the `i` either, and neither between the `i` and the `s`.

The next character in the string is a space. `\b` matches here because the space is not a word character, and the preceding character is. Again, the engine continues with the `i` which does not match with the space.

Advancing a character and restarting with the first regex token, `\b` matches between the space and the second `i` in the string. Continuing, the regex engine finds that `i` matches `i` and `s` matches `s`. Now, the engine tries to match the second `\b` at the position before the `l`. This fails because this position is between two word characters. The engine reverts to the start of the regex and advances one character to the `s` in `island`. Again, the `\b` fails to match and continues to do so until the second space is reached. It matches there, but matching the `i` fails.

But `\b` matches at the position before the third `i` in the string. The engine continues, and finds that `i` matches `i` and `s` matches `s`. The last token in the regex, `\b`, also matches at the position before the third space in the string because the space is not a word character, and the character before it is.

The engine has successfully matched the word `is` in our string, skipping the two earlier occurrences of the characters `i` and `s`. If we had used the regular expression `is`, it would have matched the `is` in `This`.

12. Alternation with The Vertical Bar or Pipe Symbol

I already explained how you can use character classes to match a single character out of several possible characters. Alternation is similar. You can use alternation to match a single regular expression out of several possible regular expressions.

If you want to search for the literal text `cat` or `dog`, separate both options with a vertical bar or pipe symbol: `cat|dog`. If you want more options, simply expand the list: `cat|dog|mouse|fish`.

The alternation operator has the lowest precedence of all regex operators. That is, it tells the regex engine to match either everything to the left of the vertical bar, or everything to the right of the vertical bar. If you want to limit the reach of the alternation, you need to use parentheses for grouping. If we want to improve the first example to match whole words only, we would need to use `\b(cat|dog)\b`. This tells the regex engine to find a word boundary, then either `cat` or `dog`, and then another word boundary. If we had omitted the parentheses then the regex engine would have searched for a word boundary followed by `cat`, or, `dog` followed by a word boundary.

Remember That The Regex Engine Is Eager

I already explained that the regex engine is eager. It stops searching as soon as it finds a valid match. The consequence is that in certain situations, the order of the alternatives matters. Suppose you want to use a regex to match a list of function names in a programming language: `Get`, `GetValue`, `Set` or `SetValue`. The obvious solution is `Get|GetValue|Set|SetValue`. Let's see how this works out when the string is `SetValue`.

The regex engine starts at the first token in the regex, `G`, and at the first character in the string, `S`. The match fails. However, the regex engine studied the entire regular expression before starting. So it knows that this regular expression uses alternation, and that the entire regex has not failed yet. So it continues with the second option, being the second `G` in the regex. The match fails again. The next token is the first `S` in the regex. The match succeeds, and the engine continues with the next character in the string, as well as the next token in the regex. The next token in the regex is the `e` after the `S` that just successfully matched. `e` matches `e`. The next token, `t` matches `t`.

At this point, the third option in the alternation has been successfully matched. Because the regex engine is eager, it considers the entire alternation to have been successfully matched as soon as one of the options has. In this example, there are no other tokens in the regex outside the alternation, so the entire regex has successfully matched `Set` in `SetValue`.

Contrary to what we intended, the regex did not match the entire string. There are several solutions. One option is to take into account that the regex engine is eager, and change the order of the options. If we use `GetValue|Get|SetValue|Set`, `SetValue` is attempted before `Set`, and the engine matches the entire string. We could also combine the four options into two and use the question mark to make part of them optional: `Get(Value)?|Set(Value)?`. Because the question mark is greedy, `SetValue` is attempted before `Set`.

The best option is probably to express the fact that we only want to match complete words. We do not want to match `Set` or `SetValue` if the string is `SetValueFunction`. So the solution is

`\b(Get|GetValue|Set|SetValue)\b` or `\b(Get(Value)?|Set(Value?))\b`. Since all options have the same end, we can optimize this further to `\b(Get|Set)(Value)?\b`.

13. Optional Items

The question mark makes the preceding token in the regular expression optional. `colou?r` matches both `colour` and `color`. The question mark is called a quantifier.

You can make several tokens optional by grouping them together using parentheses, and placing the question mark after the closing parenthesis. E.g.: `Nov(ember)?` matches `Nov` and `November`.

You can write a regular expression that matches many alternatives by including more than one question mark. `Feb(ruary)? 23(rd)?` matches `February 23rd`, `February 23`, `Feb 23rd` and `Feb 23`.

You can also use curly braces to make something optional. `colou{0,1}r` is the same as `colou?r`.

Important Regex Concept: Greediness

The question mark is the first metacharacter introduced by this tutorial that is *greedy*. The question mark gives the regex engine two choices: try to match the part the question mark applies to, or do not try to match it. The engine always tries to match that part. Only if this causes the entire regular expression to fail, will the engine try ignoring the part the question mark applies to.

The effect is that if you apply the regex `Feb 23(rd)?` to the string `Today is Feb 23rd, 2003`, the match is always `Feb 23rd` and not `Feb 23`. You can make the question mark *lazy* (i.e. turn off the greediness) by putting a second question mark after the first.

The discussion about the other repetition operators has more details on greedy and lazy quantifiers.

Looking Inside The Regex Engine

Let's apply the regular expression `colou?r` to the string `The colonel likes the color green`.

The first token in the regex is the literal `c`. The first position where it matches successfully is the `c` in `colonel`. The engine continues, and finds that `o` matches `o`, `l` matches `l` and another `o` matches `o`. Then the engine checks whether `u` matches `n`. This fails. However, the question mark tells the regex engine that failing to match `u` is acceptable. Therefore, the engine skips ahead to the next regex token: `r`. But this fails to match `n` as well. Now, the engine can only conclude that the entire regular expression cannot be matched starting at the `c` in `colonel`. Therefore, the engine starts again trying to match `c` to the first `o` in `colonel`.

After a series of failures, `c` matches the `c` in `color`, and `o`, `l` and `o` match the following characters. Now the engine checks whether `u` matches `r`. This fails. Again: no problem. The question mark allows the engine to continue with `r`. This matches `r` and the engine reports that the regex successfully matched `color` in our string.

14. Repetition with Star and Plus

One repetition operator or quantifier was already introduced: the question mark. It tells the engine to attempt to match the preceding token zero times or once, in effect making it optional.

The asterisk or star tells the engine to attempt to match the preceding token zero or more times. The plus tells the engine to attempt to match the preceding token once or more. `<[A-Za-z][A-Za-z0-9]*>` matches an HTML tag without any attributes. The angle brackets are literals. The first character class matches a letter. The second character class matches a letter or digit. The star repeats the second character class. Because we used the star, it's OK if the second character class matches nothing. So our regex will match a tag like ``. When matching `<HTML>`, the first character class will match `H`. The star will cause the second character class to be repeated three times, matching `T`, `M` and `L` with each step.

I could also have used `<[A-Za-z0-9]+>`. I did not, because this regex would match `<1>`, which is not a valid HTML tag. But this regex may be sufficient if you know the string you are searching through does not contain any such invalid tags.

Limiting Repetition

There's an additional quantifier that allows you to specify how many times a token can be repeated. The syntax is `{min,max}`, where *min* is zero or a positive integer number indicating the minimum number of matches, and *max* is an integer equal to or greater than *min* indicating the maximum number of matches. If the comma is present but *max* is omitted, the maximum number of matches is infinite. So `{0,1}` is the same as `?`, `{0,}` is the same as `*`, and `{1,}` is the same as `+`. Omitting both the comma and *max* tells the engine to repeat the token exactly *min* times.

You could use `\b[1-9][0-9]{3}\b` to match a number between 1000 and 9999. `\b[1-9][0-9]{2,4}\b` matches a number between 100 and 99999. Notice the use of the word boundaries.

Watch Out for The Greediness!

Suppose you want to use a regex to match an HTML tag. You know that the input will be a valid HTML file, so the regular expression does not need to exclude any invalid use of sharp brackets. If it sits between sharp brackets, it is an HTML tag.

Most people new to regular expressions will attempt to use `<.+>`. They will be surprised when they test it on a string like `This is a first test`. You might expect the regex to match `` and when continuing after that match, ``.

But it does not. The regex will match `first`. Obviously not what we wanted. The reason is that the plus is *greedy*. That is, the plus causes the regex engine to repeat the preceding token as often as possible. Only if that causes the entire regex to fail, will the regex engine *backtrack*. That is, it will go back to the plus, make it give up the last iteration, and proceed with the remainder of the regex. Let's take a look inside the regex engine to see in detail how this works and why this causes our regex to fail. After that, I will present you with two possible solutions.

Like the plus, the star and the repetition using curly braces are greedy.

Looking Inside The Regex Engine

The first token in the regex is `<`. This is a literal. As we already know, the first place where it will match is the first `<` in the string. The next token is the dot, which matches any character except newlines. The dot is repeated by the plus. The plus is *greedy*. Therefore, the engine will repeat the dot as many times as it can. The dot matches `E`, so the regex continues to try to match the dot with the next character. `M` is matched, and the dot is repeated once more. The next character is the `>`. You should see the problem by now. The dot matches the `>`, and the engine continues repeating the dot. The dot will match all remaining characters in the string. The dot fails when the engine has reached the void after the end of the string. Only at this point does the regex engine continue with the next token: `>`.

So far, `<.+` has matched `first test` and the engine has arrived at the end of the string. `>` cannot match here. The engine remembers that the plus has repeated the dot more often than is required. (Remember that the plus *requires* the dot to match only once.) Rather than admitting failure, the engine will *backtrack*. It will reduce the repetition of the plus by one, and then continue trying the remainder of the regex.

So the match of `.+` is reduced to `EM>first tes`. The next token in the regex is still `>`. But now the next character in the string is the last `t`. Again, these cannot match, causing the engine to backtrack further. The total match so far is reduced to `first te`. But `>` still cannot match. So the engine continues backtracking until the match of `.+` is reduced to `EM>first`. Now, `>` can match the next character in the string. The last token in the regex has been matched. The engine reports that `first` has been successfully matched.

Remember that the regex engine is *eager* to return a match. It will not continue backtracking further to see if there is another possible match. It will report the first valid match it finds. Because of greediness, this is the leftmost longest match.

Laziness Instead of Greediness

The quick fix to this problem is to make the plus *lazy* instead of greedy. Lazy quantifiers are sometimes also called “ungreedy” or “reluctant”. You can do that by putting a question mark after the plus in the regex. You can do the same with the star, the curly braces and the question mark itself. So our example becomes `<.+?>`. Let’s have another look inside the regex engine.

Again, `<` matches the first `<` in the string. The next token is the dot, this time repeated by a lazy plus. This tells the regex engine to repeat the dot as few times as possible. The minimum is one. So the engine matches the dot with `E`. The requirement has been met, and the engine continues with `>` and `M`. This fails. Again, the engine will *backtrack*. But this time, the backtracking will force the lazy plus to expand rather than reduce its reach. So the match of `.+` is expanded to `EM`, and the engine tries again to continue with `>`. Now, `>` is matched successfully. The last token in the regex has been matched. The engine reports that `` has been successfully matched. That’s more like it.

An Alternative to Laziness

In this case, there is a better option than making the plus lazy. We can use a greedy plus and a negated character class: `<[!>]+>`. The reason why this is better is because of the backtracking. When using the lazy plus, the engine has to backtrack for each character in the HTML tag that it is trying to match. When using

the negated character class, no backtracking occurs at all when the string contains valid HTML code. Backtracking slows down the regex engine. You will not notice the difference when doing a single search in a text editor. But you will save plenty of CPU cycles when using such a regex repeatedly in a tight loop in a script that you are writing, or perhaps in a custom syntax coloring scheme for EditPad Pro.

Only regex-directed engines backtrack. Text-directed engines don't and thus do not get the speed penalty. But they also do not support lazy quantifiers.

Repeating \Q...\E Escape Sequences

The \Q...\E sequence escapes a string of characters, matching them as literal characters. The escaped characters are treated as individual characters. If you place a quantifier after the \E, it will only be applied to the last character. E.g. if you apply `\Q*\d+*\E+` to `*\d+**\d+*`, the match will be `*\d+**`. Only the asterisk is repeated.

15. Use Parentheses for Grouping and Capturing

By placing part of a regular expression inside round brackets or parentheses, you can group that part of the regular expression together. This allows you to apply a quantifier to the entire group or to restrict alternation to part of the regex.

Only parentheses can be used for grouping. Square brackets define a character class, and curly braces are used by a quantifier with specific limits.

Parentheses Create Numbered Capturing Groups

Besides grouping part of a regular expression together, parentheses also create a numbered capturing group. It stores the part of the string matched by the part of the regular expression inside the parentheses.

The regex `Set(Value)?` matches `Set` or `SetValue`. In the first case, the first (and only) capturing group remains empty. In the second case, the first capturing group matches `Value`.

Non-Capturing Groups

If you do not need the group to capture its match, you can optimize this regular expression into `Set(?:Value)?`. The question mark and the colon after the opening parenthesis are the syntax that creates a non-capturing group. The question mark after the opening parenthesis is unrelated to the question mark at the end of the regex. The final question mark is the quantifier that makes the previous token optional. This quantifier cannot appear after an opening parenthesis, because there is nothing to be made optional at the start of a group. Therefore, there is no ambiguity between the question mark as an operator to make a token optional and the question mark as part of the syntax for non-capturing groups, even though this may be confusing at first. There are other kinds of groups that use the `(?:` syntax in combination with other characters than the colon that are explained later in this tutorial.

`color=(?:red|green|blue)` is another regex with a non-capturing group. This regex has no quantifiers.

Using Text Matched By Capturing Groups

Capturing groups make it easy to extract part of the regex match. You can reuse the text inside the regular expression via a backreference. Backreferences can also be used in replacement strings. Please check the replacement text tutorial for details.

16. Using Backreferences To Match The Same Text Again

Backreferences match the same text as previously matched by a capturing group. Suppose you want to match a pair of opening and closing HTML tags, and the text in between. By putting the opening tag into a backreference, we can reuse the name of the tag for the closing tag. Here's how: `<([A-Z][A-Z0-9]*)\b(?:>|/|1)>`. This regex contains only one pair of parentheses, which capture the string matched by `[A-Z][A-Z0-9]*`. This is the opening HTML tag. (Since HTML tags are case insensitive, this regex requires case insensitive matching.) The backreference `\1` (backslash one) references the first capturing group. `\1` matches the exact same text that was matched by the first capturing group. The `/` before it is a literal character. It is simply the forward slash in the closing HTML tag that we are trying to match.

To figure out the number of a particular backreference, scan the regular expression from left to right. Count the opening parentheses of all the numbered capturing groups. The first parenthesis starts backreference number one, the second number two, etc. Skip parentheses that are part of other syntax such as non-capturing groups. This means that non-capturing parentheses have another benefit: you can insert them into a regular expression without changing the numbers assigned to the backreferences. This can be very useful when modifying a complex regular expression.

You can reuse the same backreference more than once. `([a-c])x\1x\1` matches `axaxa`, `bxbxb` and `cxcxc`.

Most regex flavors support up to 99 capturing groups and double-digit backreferences. So `\99` is a valid backreference if your regex has 99 capturing groups.

Looking Inside The Regex Engine

Let's see how the regex engine applies the regex `<([A-Z][A-Z0-9]*)\b(?:>|/|1)>` to the string `Testing <I>bold italic</I> text`. The first token in the regex is the literal `<`. The regex engine traverses the string until it can match at the first `<` in the string. The next token is `[A-Z]`. The regex engine also takes note that it is now inside the first pair of capturing parentheses. `[A-Z]` matches `B`. The engine advances to `[A-Z0-9]` and `>`. This match fails. However, because of the star, that's perfectly fine. The position in the string remains at `>`. The word boundary `\b` matches at the `>` because it is preceded by `B`. The word boundary does not make the engine advance through the string. The position in the regex is advanced to `[^>]`.

This step crosses the closing bracket of the first pair of capturing parentheses. This prompts the regex engine to store what was matched inside them into the first backreference. In this case, `B` is stored.

After storing the backreference, the engine proceeds with the match attempt. `[^>]` does not match `>`. Again, because of another star, this is not a problem. The position in the string remains at `>`, and position in the regex is advanced to `>`. These obviously match. The next token is a dot, repeated by a lazy star. Because of the laziness, the regex engine initially skips this token, taking note that it should backtrack in case the remainder of the regex fails.

The engine has now arrived at the second `<` in the regex, and the second `<` in the string. These match. The next token is `/`. This does not match `I`, and the engine is forced to backtrack to the dot. The dot matches the second `<` in the string. The star is still lazy, so the engine again takes note of the available backtracking position and advances to `<` and `I`. These do not match, so the engine again backtracks.

The backtracking continues until the dot has consumed `<I>bold italic`. At this point, `<` matches the third `<` in the string, and the next token is `/` which matches `/`. The next token is `\1`. Note that the token is the backreference, and not `B`. The engine does not substitute the backreference in the regular expression. Every time the engine arrives at the backreference, it reads the value that was stored. This means that if the engine had backtracked beyond the first pair of capturing parentheses before arriving the second time at `\1`, the new value stored in the first backreference would be used. But this did not happen here, so `B` it is. This fails to match at `I`, so the engine backtracks again, and the dot consumes the third `<` in the string.

Backtracking continues again until the dot has consumed `<I>bold italic</I>`. At this point, `<` matches `<` and `/` matches `/`. The engine arrives again at `\1`. The backreference still holds `B`. `\1` matches `B`. The last token in the regex, `>` matches `>`. A complete match has been found: `<I>bold italic</I>>`.

Backtracking Into Capturing Groups

You may have wondered about the word boundary `\b` in the `<([A-Z][A-Z0-9]*)\b[^>]*>.*?</\1>` mentioned above. This is to make sure the regex won't match incorrectly paired tags such as `<boo>bold`. You may think that cannot happen because the capturing group matches `boo` which causes `\1` to try to match the same, and fail. That is indeed what happens. But then the regex engine backtracks.

Let's take the regex `<([A-Z][A-Z0-9]*)\b[^>]*>.*?</\1>` without the word boundary and look inside the regex engine at the point where `\1` fails the first time. First, `.*` continues to expand until it has reached the end of the string, and `</\1>` has failed to match each time `.*` matched one more character.

Then the regex engine backtracks into the capturing group. `[A-Z0-9]*` has matched `oo`, but would just as happily match `o` or nothing at all. When backtracking, `[A-Z0-9]*` is forced to give up one character. The regex engine continues, exiting the capturing group a second time. Since `[A-Z][A-Z0-9]*` has now matched `bo`, that is what is stored into the capturing group, overwriting `boo` that was stored before. `[^>]*` matches the second `o` in the opening tag. `>.*?</` matches `>bold</`. `\1` fails again.

The regex engine does all the same backtracking once more, until `[A-Z0-9]*` is forced to give up another character, causing it to match nothing, which the star allows. The capturing group now stores just `b`. `[^>]*` now matches `oo`. `>.*?</` once again matches `>bold<`. `\1` now succeeds, as does `>` and an overall match is found. But not the one we wanted.

There are several solutions to this. One is to use the word boundary. When `[A-Z0-9]*` backtracks the first time, reducing the capturing group to `bo`, `\b` fails to match between `o` and `o`. This forces `[A-Z0-9]*` to backtrack again immediately. The capturing group is reduced to `b` and the word boundary fails between `b` and `o`. There are no further backtracking positions, so the whole match attempt fails.

The reason we need the word boundary is that we're using `[^>]*` to skip over any attributes in the tag. If your paired tags never have any attributes, you can leave that out, and use `<([A-Z][A-Z0-9]*)>.*?</\1>`. Each time `[A-Z0-9]*` backtracks, the `>` that follows it fails to match, quickly ending the match attempt.

If you don't want the regex engine to backtrack into capturing groups, you can use an atomic group. The tutorial section on atomic grouping has all the details.

Repetition and Backreferences

As I mentioned in the above inside look, the regex engine does not permanently substitute backreferences in the regular expression. It will use the last match saved into the backreference each time it needs to be used. If a new match is found by capturing parentheses, the previously saved match is overwritten. There is a clear difference between `([abc]+)` and `([abc])+`. Though both successfully match `cab`, the first regex will put `cab` into the first backreference, while the second regex will only store `b`. That is because in the second regex, the plus caused the pair of parentheses to repeat three times. The first time, `c` was stored. The second time, `a`, and the third time `b`. Each time, the previous value was overwritten, so `b` remains.

This also means that `([abc]+)=\1` will match `cab=cab`, and that `([abc])+=\1` will not. The reason is that when the engine arrives at `\1`, it holds `b` which fails to match `c`. Obvious when you look at a simple example like this one, but a common cause of difficulty with regular expressions nonetheless. When using backreferences, always double check that you are really capturing what you want.

Useful Example: Checking for Doubled Words

When editing text, doubled words such as “the the” easily creep in. Using the regex `\b(\w+)\s+\1\b` in your text editor, you can easily find them. To delete the second word, simply type in `\1` as the replacement text and click the Replace button.

Parentheses and Backreferences Cannot Be Used Inside Character Classes

Parentheses cannot be used inside character classes, at least not as metacharacters. When you put a parenthesis in a character class, it is treated as a literal character. So the regex `[(a)b]` matches `a`, `b`, `(`, and `)`.

Backreferences, too, cannot be used inside a character class. The `\1` in a regex like `(a)[\1b]` is a needlessly escaped literal 1.

17. Backreferences to Failed Groups

The previous topic on backreferences applies to all regex flavors, except those few that don't support backreferences at all. Flavors behave differently when you start doing things that don't fit the "match the text matched by a previous capturing group" job description.

There is a difference between a backreference to a capturing group that matched nothing, and one to a capturing group that did not participate in the match at all. The regex `(q?)b\1` matches `b`. `q?` is optional and matches nothing, causing `(q?)` to successfully match and capture nothing. `b` matches `b` and `\1` successfully matches the nothing captured by the group.

The regex `(q)?b\1` fails to match `b`. `(q)` fails to match at all, so the group never gets to capture anything at all. Because the whole group is optional, the engine does proceed to match `b`. The engine now arrives at `\1` which references a group that did not participate in the match attempt at all. This causes the backreference to fail to match at all, mimicking the result of the group. Since there's no `?` making `\1` optional, the overall match attempt fails.

Backreferences to Non-Existent Capturing Groups

Backreferences to groups that do not exist, such as `(one)\7`, are an error. A backslash followed by two digits forms a single-digit backreference followed by a literal digit if there are fewer capturing groups than the two-digit number. So `(one)\12` matches `oneone2`.

Forward References

EditPad allows you to use a backreference to a group that appears later in the regex. Forward references are obviously only useful if they're inside a repeated group. Then there can be situations in which the regex engine evaluates the backreference after the group has already matched. Before the group is attempted, the backreference fails like a backreference to a failed group does.

The regex `(\2two|(one))+` matches `oneonetwo`. At the start of the string, `\2` fails. Trying the other alternative, `one` is matched by the second capturing group, and subsequently by the first group. The first group is then repeated. This time, `\2` matches `one` as captured by the second group. `two` then matches `two`. With two repetitions of the first group, the regex has matched the whole subject string.

Nested References

A nested reference is a backreference inside the capturing group that it references. Like forward references, nested references are only useful if they're inside a repeated group, as in `(\1two|(one))+`. When nested references are supported, this regex also matches `oneonetwo`. At the start of the string, `\1` fails. Trying the other alternative, `one` is matched by the second capturing group, and subsequently by the first group. The first group is then repeated. This time, `\1` matches `one` as captured by the last iteration of the first group. It doesn't matter that the regex engine has re-entered the first group. The text matched by the group was stored into the backreference when the group was previously exited. `two` then matches `two`. With two repetitions of the first group, the regex has matched the whole subject string. If you retrieve the text from the capturing

groups after the match, the first group stores `onetwo` while the second group captured the first occurrence of `one` in the string.

18. Named Capturing Groups and Backreferences

EditPad supports numbered capturing groups and numbered backreferences. Long regular expressions with lots of groups and backreferences may be hard to read. They can be particularly difficult to maintain as adding or removing a capturing group in the middle of the regex upsets the numbers of all the groups that follow the added or removed group.

Python's `re` module was the first to offer a solution: named capturing groups and named backreferences. `(?P<name>group)` captures the match of `group` into the backreference “name”. `name` must be an alphanumeric sequence starting with a letter. `group` can be any regular expression. You can reference the contents of the group with the named backreference `(?P=name)`. The question mark, P, angle brackets, and equals signs are all part of the syntax. Though the syntax for the named backreference uses parentheses, it's just a backreference that doesn't do any capturing or grouping. The HTML tags example can be written as `<(P<tag>[A-Z][A-Z0-9]*)\b[^>]*>.*?</(?P=tag)>`.

.NET also supports named capture. Microsoft's developers invented their own syntax, rather than follow the one pioneered by Python and copied by PCRE (the only two regex engines that supported named capture at that time). `(?<name>group)` or `(?'name'group)` captures the match of `group` into the backreference “name”. The named backreference is `\k<name>` or `\k'name'`. Compared with Python, there is no P in the syntax for named groups. The syntax for named backreferences is more similar to that of numbered backreferences than to what Python uses. You can use single quotes or angle brackets around the name. This makes absolutely no difference in the regex. You can use both styles interchangeably. The syntax using angle brackets is preferable in programming languages that use single quotes to delimit strings, while the syntax using single quotes is preferable when adding your regex to an XML file, as this minimizes the amount of escaping you have to do to format your regex as a literal string or as XML content.

Because Python and .NET introduced their own syntax, we refer to these two variants as the “Python syntax” and the “.NET syntax” for named capture and named backreferences. EditPad supports both.

Numbers for Named Capturing Groups

Mixing named and numbered capturing groups is not recommended because flavors are inconsistent in how the groups are numbered. If a group doesn't need to have a name, make it non-capturing using the `(?:group)` syntax. You can make all unnamed groups non-capturing by putting the `(?n)` mode modifier at the start of your regex. If you make all unnamed groups non-capturing, you can skip this section and save yourself a headache.

Most flavors number both named and unnamed capturing groups by counting their opening parentheses from left to right. Adding a named capturing group to an existing regex still upsets the numbers of the unnamed groups. In .NET, however, unnamed capturing groups are assigned numbers first, counting their opening parentheses from left to right, skipping all named groups. After that, named groups are assigned the numbers that follow by counting the opening parentheses of the named groups from left to right.

EditPad copied the Python and the .NET syntax at a time when only Python and PCRE used the Python syntax, and only .NET used the .NET syntax. Therefore it also copied the numbering behavior of both Python and .NET, so that regexes intended for Python and .NET would keep their behavior. It numbers Python-style named groups along unnamed ones, like Python does. It numbers .NET-style named groups afterward, like .NET does. These rules apply even when you mix both styles in the same regex.

As an example, the regex `(a)(?P<x>b)(c)(?P<y>d)` matches `abcd` as expected. If you do a search-and-replace with this regex and the replacement `\1\2\3\4` or `$1$2$3$4`, you will get `abcd`. All four groups were numbered from left to right, from one till four.

Things are a bit more complicated with .NET. The regex `(a)(?<x>b)(c)(?<y>d)` again matches `abcd`. However, if you do a search-and-replace with `$1$2$3$4` as the replacement, you will get `acbd`. First, the unnamed groups `(a)` and `(c)` got the numbers 1 and 2. Then the named groups “x” and “y” got the numbers 3 and 4.

In all other flavors that copied the .NET syntax the regex `(a)(?<x>b)(c)(?<y>d)` still matches `abcd`. But in all those flavors, except the EditPad flavor, the replacement `\1\2\3\4` or `$1$2$3$4` gets you `abcd`. All four groups were numbered from left to right.

In PowerGREP named capturing groups play a special role. Groups with the same name are shared between all regular expressions and replacement texts in the same PowerGREP action. This allows captured by a named capturing group in one part of the action to be referenced in a later part of the action. Because of this, PowerGREP does not allow numbered references to named capturing groups at all. When mixing named and numbered groups in a regex, the numbered groups are still numbered following the Python and .NET rules, like the JGsoft flavor always does.

Multiple Groups with The Same Name

EditPad allows multiple groups in the regular expression to have the same name. All groups with the same name share the same storage for the text they match. Thus, a backreference to that name matches the text that was matched by the group with that name that most recently captured something. A reference to the name in the replacement text inserts the text matched by the group with that name that was the last one to capture something.

19. Branch Reset Groups

Alternatives inside a branch reset group share the same capturing groups. The syntax is `(?|regex)` where `(?|` opens the group and `regex` is any regular expression. If you don't use any alternation or capturing groups inside the branch reset group, then its special function doesn't come into play. It then acts as a non-capturing group.

The regex `(?|(a)|(b)|(c))` consists of a single branch reset group with three alternatives. This regex matches either `a`, `b`, or `c`. The regex has only a single capturing group with number 1 that is shared by all three alternatives. After the match, `$1` holds `a`, `b`, or `c`.

Compare this with the regex `(a)|(b)|(c)` that lacks the branch reset group. This regex also matches `a`, `b`, or `c`. But it has three capturing groups. After the match, `$1` holds `a` or nothing at all, `$2` holds `b` or nothing at all, while `$3` holds `c` or nothing at all.

Backreferences to capturing groups inside branch reset groups work like you'd expect. `(?|(a)|(b)|(c))\1` matches `aa`, `bb`, or `cc`. Since only one of the alternatives inside the branch reset group can match, the alternative that participates in the match determines the text stored by the capturing group and thus the text matched by the backreference.

The alternatives in the branch reset group don't need to have the same number of capturing groups. `(?|abc|(d)(e)(f)|g(h)i)` has three capturing groups. When this regex matches `abc`, all three groups are empty. When `def` is matched, `$1` holds `d`, `$2` holds `e` and `$3` holds `f`. When `ghi` is matched, `$1` holds `h` while the other two are empty.

You can have capturing groups before and after the branch reset group. Groups before the branch reset group are numbered as usual. Groups in the branch reset group are numbered continued from the groups before the branch reset group, which each alternative resetting the number. Groups after the branch reset group are numbered continued from the alternative with the most groups, even if that is not the last alternative. So `(x)(?|abc|(d)(e)(f)|g(h)i)(y)` defines five capturing groups. `(x)` is group 1, `(d)` and `(h)` are group 2, `(e)` is group 3, `(f)` is group 4, and `(y)` is group 5.

Named Capturing Groups in Branch Reset Groups

You can use named capturing groups inside branch reset groups. You have to use the same group names in the same order in each alternative. Mismatched group names are an error.

`(?'before'x)(?|abc|(?'left'd)(?'middle'e)(?'right'f)|g(?'left'h)i)(?'after'y)` is the same as the previous regex. It names the five groups “before”, “left”, “middle”, “right”, and “after”. Notice that because the 3rd alternative has only one capturing group, that must be the name of the first group in the other alternatives.

If you omit the names in some alternatives, the groups will still share the names with the other alternatives. In the regex `(?'before'x)(?|abc|(?'left'd)(?'middle'e)(?'right'f)|g(h)i)(?'after'y)` the group `(h)` is still named “left” because the branch reset group makes it share the name and number of `(?'left'd)`.

In PowerGREP, groups with the same name are always treated as one and the same group. So you don't really need to use a branch reset group in PowerGREP when using named capturing groups.

Day and Month with Accurate Number of Days

It's time for a more practical example. These two regular expressions match a date in m/d or mm/dd format. They exclude invalid dates such as 2/31.

```
^(?:((0?[13578])1[02])|(0?[469])11|(0?2)/([12][0-9])0?[1-9])$ # 31 days
|((0?[13578])1[02])|((0?[469])11|((30|([12][0-9])0?[1-9])) # 30 days
|((0?2)/([12][0-9])0?[1-9]) # 29 days
```

The first version uses a non-capturing group (?:...) to group the alternatives. It has six separate capturing groups. \$1 and \$2 hold the month and the day for months with 31 days. \$3 and \$4 hold them for months with 30 days. \$5 and \$6 are only used for February.

```
^(?:|((0?[13578])1[02])|((0?[469])11|((30|([12][0-9])0?[1-9])) # 31 days
|((0?2)/([12][0-9])0?[1-9]) # 29 days
```

The second version uses a branch reset group (?|...) to group the alternatives and merge their capturing groups. The 4th character is the only difference between these two regexes. Now there are only two capturing groups. These are shared between the three alternatives. When a match is found \$1 always holds the month and 2 always holds the day, regardless of the number of days in the month.

20. Free-Spacing Regular Expressions

Most modern regex flavors support a variant of the regular expression syntax called free-spacing mode. This mode allows for regular expressions that are much easier for people to read. Of the flavors discussed in this tutorial, only XML Schema and the POSIX and GNU flavors don't support it. Plain JavaScript doesn't either, but XRegExp does. The mode is usually enabled by setting an option or flag outside the regex. With flavors that support mode modifiers, you can put `(?x)` the very start of the regex to make the remainder of the regex free-spacing.

In free-spacing mode, whitespace between regular expression tokens is ignored. Whitespace includes spaces, tabs, and line breaks. Note that only whitespace *between* tokens is ignored. `a b c` is the same as `abc` in free-spacing mode. But `\ d` and `\d` are not the same. The former matches `d`, while the latter matches a digit. `\d` is a single regex token composed of a backslash and a "d". Breaking up the token with a space gives you an escaped space (which matches a space), and a literal "d".

Likewise, grouping modifiers cannot be broken up. `(?>atomic)` is the same as `(?> ato mic)` and as `(?>ato mic)`. They all match the same atomic group. They're not the same as `(? >atomic)`. The latter is a syntax error. The `?>` grouping modifier is a single element in the regex syntax, and must stay together. This is true for all such constructs, including lookahead, named groups, etc.

EditPad ignores all Unicode spaces and line breaks in free-spacing mode. Most other regex flavors, even if they normally support Unicode, only ignore the ASCII space, tab, line feed, carriage return, and form feed characters.

Free-Spacing in Character Classes

A character class is also treated as a single token. `[abc]` is not the same as `[a b c]`. The former matches one of three letters, while the latter matches those three letters or a space. In other words: free-spacing mode has no effect inside character classes. Spaces and line breaks inside character classes will be included in the character class. This means that in free-spacing mode, you can use `\` or `[]` to match a single space. Use whichever you find more readable. The hexadecimal escape `\x20` also works, of course.

Comments in Free-Spacing Mode

Another feature of free-spacing mode is that the `#` character starts a comment. The comment runs until the end of the line. Everything from the `#` until the line break character is ignored.

Putting it all together, the regex to match a valid date can be clarified by writing it across multiple lines:

```
# Match a 20th or 21st century date in yyyy-mm-dd format
((?:19|20)\d\d)      # year (group 1)
[- /.]              # separator
(0[1-9]|1[012])     # month (group 2)
[- /.]              # separator
(0[1-9]|1[12]|0-9]|3[01]) # day (group 3)
```

Java 8 Helpful Match Replace Split Copy Paste History

Case sensitive Free-spacing [...] Dot doesn't match line breaks ^\$ don't match at line breaks

Default line breaks Reset

Match a 20th or 21st century date in yyyy-mm-dd format

```
(?:(?:19|20)\d\d|(?!(?:19|20)\d\d)[0-9]{4})[-./](?!(?:19|20)\d\d)[0-9]{1,2}[-./](?!(?:19|20)\d\d)[0-9]{1,2}
```

History

Date yyyy-mm-dd

Create Convert Test Debug Use Library GREP Forum

Detailed Explain Token Insert Token Compare (no comparison) Export RegExMagic

Comment: Match a 20th or 21st century date in yyyy-mm-dd format

- Match the regex below and capture its match into backreference number 1
 - Match the regular expression below
 - Match this alternative (attempting the next alternative only if this one fails)
 - Match the character string "19" literally
 - Or match this alternative (the entire group fails if this one fails to match)
 - Match the character string "20" literally
 - Match a single character that is a "digit" (ASCII 0-9 only)
 - Match a single character that is a "digit" (ASCII 0-9 only)
 - Comment: year (group 1)
 - Match a single character from the list "-./"
 - Comment: separator
 - Match the regex below and capture its match into backreference number 2
 - Match this alternative (attempting the next alternative only if this one fails)
 - Match the character "0" literally
 - Match a single character in the range between "1" and "9"
 - Or match this alternative (the entire group fails if this one fails to match)
 - Match the character "1" literally
 - Match a single character from the list "012"
 - Comment: month (group 2)
 - Match a single character from the list "-./"
 - Comment: separator
 - Match the regex below and capture its match into backreference number 3
 - Match this alternative (attempting the next alternative only if this one fails)
 - Match the character "0" literally
 - Match a single character in the range between "1" and "9"
 - Or match this alternative (attempting the next alternative only if this one fails)

Comments Without Free-Spacing

EditPad also allows you to add comments to your regex without using free-spacing mode. The syntax is `(?#comment)` where "comment" can be whatever you want, as long as it does not contain a closing parenthesis. The regex engine ignores everything after the `(?#` until the first closing parenthesis.

21. Unicode Regular Expressions

Unicode is a character set that aims to define all characters and glyphs from all human languages, living and dead. With more and more software being required to support multiple languages, or even just *any* language, Unicode has been strongly gaining popularity in recent years. Using different character sets for different languages is simply too cumbersome for programmers and users.

Characters, Code Points, and Graphemes or How Unicode Makes a Mess of Things

Most people would consider `à` a single character. Unfortunately, it need not be depending on the meaning of the word “character”.

EditPad treats any single Unicode *code point* as a single character. When this tutorial tells you that the dot matches any single character, this translates into Unicode parlance as “the dot matches any single Unicode code point”. In Unicode, `à` can be encoded as two code points: U+0061 (a) followed by U+0300 (grave accent). In this situation, `.` applied to `à` will match `a` without the accent. `^.$` will fail to match, since the string consists of two code points. `^..$` matches `à`.

The Unicode code point U+0300 (grave accent) is a *combining mark*. Any code point that is not a combining mark can be followed by any number of combining marks. This sequence, like U+0061 U+0300 above, is displayed as a single *grapheme* on the screen.

Unfortunately, `à` can also be encoded with the single Unicode code point U+00E0 (a with grave accent). The reason for this duality is that many historical character sets encode “a with grave accent” as a single character. Unicode’s designers thought it would be useful to have a one-on-one mapping with popular legacy character sets, in addition to the Unicode way of separating marks and base letters (which makes arbitrary combinations not supported by legacy character sets possible).

How to Match a Single Unicode Grapheme

Matching a single grapheme, whether it’s encoded as a single code point, or as multiple code points using combining marks, is easy: simply use `\X`. You can consider `\X` the Unicode version of the dot. There is one difference, though: `\X` always matches line break characters, whereas the dot does not match line break characters unless you enable the dot matches newline matching mode.

Matching a Specific Code Point

To match a specific Unicode code point, use `\uFFFF` where FFFF is the hexadecimal number of the code point you want to match. You must always specify 4 hexadecimal digits. E.g. `\u00E0` matches `à`, but only when encoded as a single code point U+00E0.

Unicode Categories

In addition to complications, Unicode also brings new possibilities. One is that each Unicode character belongs to a certain category. You can match a single character belonging to the “letter” category with `\p{L}`. You can match a single character *not* belonging to that category with `\P{L}`.

Again, “character” really means “Unicode code point”. `\p{L}` matches a single code point in the category “letter”. If your input string is `ä` encoded as U+0061 U+0300, it matches `a` without the accent. If the input is `ä` encoded as U+00E0, it matches `ä` with the accent. The reason is that both the code points U+0061 (a) and U+00E0 (ä) are in the category “letter”, while U+0300 is in the category “mark”.

In addition to the standard notation, EditPad allows you to use the shorthand `\pL`. The shorthand only works with single-letter Unicode properties. `\pLl` is *not* the equivalent of `\p{Ll}`. It is the equivalent of `\p{L}l` which matches `Al` or `äl` or any Unicode letter followed by a literal `l`.

EditPad also supports the longhand `\p{Letter}`. You can find a complete list of all Unicode properties below. You may omit the underscores or use hyphens or spaces instead.

- `\p{L}` or `\p{Letter}`: any kind of letter from any language.
 - `\p{Ll}` or `\p{Lowercase_Letter}`: a lowercase letter that has an uppercase variant.
 - `\p{Lu}` or `\p{Uppercase_Letter}`: an uppercase letter that has a lowercase variant.
 - `\p{Lt}` or `\p{Titlecase_Letter}`: a letter that appears at the start of a word when only the first letter of the word is capitalized.
 - `\p{L&}` or `\p{Cased_Letter}`: a letter that exists in lowercase and uppercase variants (combination of Ll, Lu and Lt).
 - `\p{Lm}` or `\p{Modifier_Letter}`: a special character that is used like a letter.
 - `\p{Lo}` or `\p{Other_Letter}`: a letter or ideograph that does not have lowercase and uppercase variants.
- `\p{M}` or `\p{Mark}`: a character intended to be combined with another character (e.g. accents, umlauts, enclosing boxes, etc.).
 - `\p{Mn}` or `\p{Non_Spacing_Mark}`: a character intended to be combined with another character without taking up extra space (e.g. accents, umlauts, etc.).
 - `\p{Mc}` or `\p{Spacing_Combining_Mark}`: a character intended to be combined with another character that takes up extra space (vowel signs in many Eastern languages).
 - `\p{Me}` or `\p{Enclosing_Mark}`: a character that encloses the character it is combined with (circle, square, keycap, etc.).
- `\p{Z}` or `\p{Separator}`: any kind of whitespace or invisible separator.
 - `\p{Zs}` or `\p{Space_Separator}`: a whitespace character that is invisible, but does take up space.
 - `\p{Zl}` or `\p{Line_Separator}`: line separator character U+2028.
 - `\p{Zp}` or `\p{Paragraph_Separator}`: paragraph separator character U+2029.
- `\p{S}` or `\p{Symbol}`: math symbols, currency signs, dingbats, box-drawing characters, etc.
 - `\p{Sm}` or `\p{Math_Symbol}`: any mathematical symbol.
 - `\p{Sc}` or `\p{Currency_Symbol}`: any currency sign.
 - `\p{Sk}` or `\p{Modifier_Symbol}`: a combining character (mark) as a full character on its own.
 - `\p{So}` or `\p{Other_Symbol}`: various symbols that are not math symbols, currency signs, or combining characters.
- `\p{N}` or `\p{Number}`: any kind of numeric character in any script.

- `\p{Nd}` or `\p{Decimal_Digit_Number}`: a digit zero through nine in any script except ideographic scripts.
- `\p{Nl}` or `\p{Letter_Number}`: a number that looks like a letter, such as a Roman numeral.
- `\p{No}` or `\p{Other_Number}`: a superscript or subscript digit, or a number that is not a digit 0–9 (excluding numbers from ideographic scripts).
- `\p{P}` or `\p{Punctuation}`: any kind of punctuation character.
 - `\p{Pd}` or `\p{Dash_Punctuation}`: any kind of hyphen or dash.
 - `\p{Ps}` or `\p{Open_Punctuation}`: any kind of opening bracket.
 - `\p{Pe}` or `\p{Close_Punctuation}`: any kind of closing bracket.
 - `\p{Pi}` or `\p{Initial_Punctuation}`: any kind of opening quote.
 - `\p{Pf}` or `\p{Final_Punctuation}`: any kind of closing quote.
 - `\p{Pc}` or `\p{Connector_Punctuation}`: a punctuation character such as an underscore that connects words.
 - `\p{Po}` or `\p{Other_Punctuation}`: any kind of punctuation character that is not a dash, bracket, quote or connector.
- `\p{C}` or `\p{Other}`: invisible control characters and unused code points.
 - `\p{Cc}` or `\p{Control}`: an ASCII or Latin-1 control character: 0x00–0x1F and 0x7F–0x9F.
 - `\p{Cf}` or `\p{Format}`: invisible formatting indicator.
 - `\p{Co}` or `\p{Private_Use}`: any code point reserved for private use.
 - `\p{Cs}` or `\p{Surrogate}`: one half of a surrogate pair in UTF-16 encoding.
 - `\p{Cn}` or `\p{Unassigned}`: any code point to which no character has been assigned.

Unicode Scripts

The Unicode standard places each assigned code point (character) into one script. A script is a group of code points used by a particular human writing system. Some scripts like Thai correspond with a single human language. Other scripts like Latin span multiple languages.

Some languages are composed of multiple scripts. There is no Japanese Unicode script. Instead, Unicode offers the Hiragana, Katakana, Han, and Latin scripts that Japanese documents are usually composed of.

A special script is the Common script. This script contains all sorts of characters that are common to a wide range of scripts. It includes all sorts of punctuation, whitespace and miscellaneous symbols.

All assigned Unicode code points (those matched by `\p{Cn}`) are part of exactly one Unicode script. All unassigned Unicode code points (those matched by `\p{Cn}`) are not part of any Unicode script at all.

Here's a list:

1. `\p{Common}`
2. `\p{Arabic}`
3. `\p{Armenian}`
4. `\p{Bengali}`
5. `\p{Bopomofo}`
6. `\p{Braille}`
7. `\p{Buhid}`
8. `\p{Canadian_Aboriginal}`
9. `\p{Cherokee}`

10. `\p{Cyrillic}`
11. `\p{Devanagari}`
12. `\p{Ethiopic}`
13. `\p{Georgian}`
14. `\p{Greek}`
15. `\p{Gujarati}`
16. `\p{Gurmukhi}`
17. `\p{Han}`
18. `\p{Hangul}`
19. `\p{Hanunoo}`
20. `\p{Hebrew}`
21. `\p{Hiragana}`
22. `\p{Inherited}`
23. `\p{Kannada}`
24. `\p{Katakana}`
25. `\p{Khmer}`
26. `\p{Lao}`
27. `\p{Latin}`
28. `\p{Limbu}`
29. `\p{Malayalam}`
30. `\p{Mongolian}`
31. `\p{Myanmar}`
32. `\p{Ogham}`
33. `\p{Oriya}`
34. `\p{Runic}`
35. `\p{Sinhala}`
36. `\p{Syriac}`
37. `\p{Tagalog}`
38. `\p{Tagbanwa}`
39. `\p{TaiLe}`
40. `\p{Tamil}`
41. `\p{Telugu}`
42. `\p{Thaana}`
43. `\p{Thai}`
44. `\p{Tibetan}`
45. `\p{Yi}`

EditPad allows you to use `\p{IsLatin}` instead of `\p{Latin}`. The “Is” syntax is useful for distinguishing between scripts and blocks, as explained in the next section.

Unicode Blocks

The Unicode standard divides the Unicode character map into different blocks or ranges of code points. Each block is used to define characters of a particular script like “Tibetan” or belonging to a particular group like “Braille Patterns”. Most blocks include unassigned code points, reserved for future expansion of the Unicode standard.

Note that Unicode blocks do not correspond 100% with scripts. An essential difference between blocks and scripts is that a block is a single contiguous range of code points, as listed below. Scripts consist of characters taken from all over the Unicode character map. Blocks may include unassigned code points (i.e. code points matched by `\p{Cn}`). Scripts never include unassigned code points. Generally, if you’re not sure whether to use a Unicode script or Unicode block, use the script.

For example, the Currency block does not include the dollar and yen symbols. Those are found in the Basic_Latin and Latin-1_Supplement blocks instead, even though both are currency symbols, and the yen symbol is not a Latin character. This is for historical reasons, because the ASCII standard includes the dollar sign, and the ISO-8859 standard includes the yen sign. You should not blindly use any of the blocks listed below based on their names. Instead, look at the ranges of characters they actually match. A tool like RegexpBuddy can be very helpful with this. The Unicode property `\p{Sc}` or `\p{Currency_Symbol}` would be a better choice than the Unicode block `\p{InCurrency_Symbols}` when trying to find all currency symbols.

1. `\p{InBasic_Latin}`: U+0000–U+007F
2. `\p{InLatin-1_Supplement}`: U+0080–U+00FF
3. `\p{InLatin_Extended-A}`: U+0100–U+017F
4. `\p{InLatin_Extended-B}`: U+0180–U+024F
5. `\p{InIPA_Extensions}`: U+0250–U+02AF
6. `\p{InSpacing_Modifier_Letters}`: U+02B0–U+02FF
7. `\p{InCombining_Diacritical_Marks}`: U+0300–U+036F
8. `\p{InGreek_and_Coptic}`: U+0370–U+03FF
9. `\p{InCyrillic}`: U+0400–U+04FF
10. `\p{InCyrillic_Supplementary}`: U+0500–U+052F
11. `\p{InArmenian}`: U+0530–U+058F
12. `\p{InHebrew}`: U+0590–U+05FF
13. `\p{InArabic}`: U+0600–U+06FF
14. `\p{InSyriac}`: U+0700–U+074F
15. `\p{InThaana}`: U+0780–U+07BF
16. `\p{InDevanagari}`: U+0900–U+097F
17. `\p{InBengali}`: U+0980–U+09FF
18. `\p{InGurmukhi}`: U+0A00–U+0A7F
19. `\p{InGujarati}`: U+0A80–U+0AFF
20. `\p{InOriya}`: U+0B00–U+0B7F
21. `\p{InTamil}`: U+0B80–U+0BFF
22. `\p{InTelugu}`: U+0C00–U+0C7F
23. `\p{InKannada}`: U+0C80–U+0CFF
24. `\p{InMalayalam}`: U+0D00–U+0D7F
25. `\p{InSinhala}`: U+0D80–U+0DFF
26. `\p{InThai}`: U+0E00–U+0E7F
27. `\p{InLao}`: U+0E80–U+0EFF
28. `\p{InTibetan}`: U+0F00–U+0FFF
29. `\p{InMyanmar}`: U+1000–U+109F
30. `\p{InGeorgian}`: U+10A0–U+10FF
31. `\p{InHangul_Jamo}`: U+1100–U+11FF
32. `\p{InEthiopic}`: U+1200–U+137F
33. `\p{InCherokee}`: U+13A0–U+13FF
34. `\p{InUnified_Canadian_Aboriginal_Syllabics}`: U+1400–U+167F
35. `\p{InOgham}`: U+1680–U+169F
36. `\p{InRunic}`: U+16A0–U+16FF
37. `\p{InTagalog}`: U+1700–U+171F
38. `\p{InHanunoo}`: U+1720–U+173F
39. `\p{InBuhid}`: U+1740–U+175F
40. `\p{InTagbanwa}`: U+1760–U+177F
41. `\p{InKhmer}`: U+1780–U+17FF
42. `\p{InMongolian}`: U+1800–U+18AF

43. `\p{InLimbu}`: U+1900–U+194F
44. `\p{InTai_Le}`: U+1950–U+197F
45. `\p{InKhmer_Symbols}`: U+19E0–U+19FF
46. `\p{InPhonetic_Extensions}`: U+1D00–U+1D7F
47. `\p{InLatin_Extended_Additional}`: U+1E00–U+1EFF
48. `\p{InGreek_Extended}`: U+1F00–U+1FFF
49. `\p{InGeneral_Punctuation}`: U+2000–U+206F
50. `\p{InSuperscripts_and_Subscripts}`: U+2070–U+209F
51. `\p{InCurrency_Symbols}`: U+20A0–U+20CF
52. `\p{InCombining_Diacritical_Marks_for_Symbols}`: U+20D0–U+20FF
53. `\p{InLetterlike_Symbols}`: U+2100–U+214F
54. `\p{InNumber_Forms}`: U+2150–U+218F
55. `\p{InArrows}`: U+2190–U+21FF
56. `\p{InMathematical_Operators}`: U+2200–U+22FF
57. `\p{InMiscellaneous_Technical}`: U+2300–U+23FF
58. `\p{InControl_Pictures}`: U+2400–U+243F
59. `\p{InOptical_Character_Recognition}`: U+2440–U+245F
60. `\p{InEnclosed_Alphanumerics}`: U+2460–U+24FF
61. `\p{InBox_Drawing}`: U+2500–U+257F
62. `\p{InBlock_Elements}`: U+2580–U+259F
63. `\p{InGeometric_Shapes}`: U+25A0–U+25FF
64. `\p{InMiscellaneous_Symbols}`: U+2600–U+26FF
65. `\p{InDingbats}`: U+2700–U+27BF
66. `\p{InMiscellaneous_Mathematical_Symbols-A}`: U+27C0–U+27EF
67. `\p{InSupplemental_Arrows-A}`: U+27F0–U+27FF
68. `\p{InBraille_Patterns}`: U+2800–U+28FF
69. `\p{InSupplemental_Arrows-B}`: U+2900–U+297F
70. `\p{InMiscellaneous_Mathematical_Symbols-B}`: U+2980–U+29FF
71. `\p{InSupplemental_Mathematical_Operators}`: U+2A00–U+2AFF
72. `\p{InMiscellaneous_Symbols_and_Arrows}`: U+2B00–U+2BFF
73. `\p{InCJK_Radicals_Supplement}`: U+2E80–U+2EFF
74. `\p{InKangxi_Radicals}`: U+2F00–U+2FDF
75. `\p{InIdeographic_Description_Characters}`: U+2FF0–U+2FFF
76. `\p{InCJK_Symbols_and_Punctuation}`: U+3000–U+303F
77. `\p{InHiragana}`: U+3040–U+309F
78. `\p{InKatakana}`: U+30A0–U+30FF
79. `\p{InBopomofo}`: U+3100–U+312F
80. `\p{InHangul_Compatibility_Jamo}`: U+3130–U+318F
81. `\p{InKanbun}`: U+3190–U+319F
82. `\p{InBopomofo_Extended}`: U+31A0–U+31BF
83. `\p{InKatakana_Phonetic_Extensions}`: U+31F0–U+31FF
84. `\p{InEnclosed_CJK_Letters_and_Months}`: U+3200–U+32FF
85. `\p{InCJK_Compatibility}`: U+3300–U+33FF
86. `\p{InCJK_Unified_Ideographs_Extension_A}`: U+3400–U+4DBF
87. `\p{InYijing_Hexagram_Symbols}`: U+4DC0–U+4DFF
88. `\p{InCJK_Unified_Ideographs}`: U+4E00–U+9FFF
89. `\p{InYi_Syllables}`: U+A000–U+A48F
90. `\p{InYi_Radicals}`: U+A490–U+A4CF
91. `\p{InHangul_Syllables}`: U+AC00–U+D7AF
92. `\p{InHigh_Surrogates}`: U+D800–U+DB7F
93. `\p{InHigh_Private_Use_Surrogates}`: U+DB80–U+DBFF
94. `\p{InLow_Surrogates}`: U+DC00–U+DFFF

- 95. \p{InPrivate_Use_Area}: U+E000–U+F8FF
- 96. \p{InCJK_Compatibility_Ideographs}: U+F900–U+FAFF
- 97. \p{InAlphabetic_Presentation_Forms}: U+FB00–U+FB4F
- 98. \p{InArabic_Presentation_Forms-A}: U+FB50–U+FDFF
- 99. \p{InVariation_Selectors}: U+FE00–U+FE0F
- 100. \p{InCombining_Half_Marks}: U+FE20–U+FE2F
- 101. \p{InCJK_Compatibility_Forms}: U+FE30–U+FE4F
- 102. \p{InSmall_Form_Variants}: U+FE50–U+FE6F
- 103. \p{InArabic_Presentation_Forms-B}: U+FE70–U+FEFF
- 104. \p{InHalfwidth_and_Fullwidth_Forms}: U+FF00–U+FFEF
- 105. \p{InSpecials}: U+FFF0–U+FFFF

22. Specifying Modes Inside The Regular Expression

Mode modifiers allow you to specify matching options in the regular expression itself. If you use a mode modifier to toggle an option then that overrides the same option in EditPad's user interface.

- `(?i)` makes the regex case insensitive.
- `(?x)` turn on free-spacing mode.
- `(?s)` for “single line mode” makes the dot match all characters, including line breaks.
- `(?m)` for “multi-line mode” makes the caret and dollar match at the start and end of each line in the subject string.
- `(?n)` turns all unnamed groups into non-capturing groups.

Turning Modes On and Off for Only Part of The Regular Expression

EditPad allows you to apply modifiers to only part of the regular expression. If you insert the modifier `(?ism)` in the middle of the regex then the modifier only applies to the part of the regex to the right of the modifier. You can turn off modes by preceding them with a minus sign. All modes after the minus sign will be turned off. E.g. `(?i-sm)` turns on case insensitivity, and turns off both single-line mode and multi-line mode.

Modifier Spans

Instead of using two modifiers, one to turn an option on, and one to turn it off, you use a modifier span. `(?i)caseless(?-i)cased(?i)caseless` is equivalent to `(?i)caseless(?-i:cased)caseless`. This syntax resembles that of the non-capturing group `(?:group)`. You could think of a non-capturing group as a modifier span that does not change any modifiers. Like a non-capturing group, the modifier span does not create a backreference.

23. Atomic Grouping

An atomic group is a group that, when the regex engine exits from it, automatically throws away all backtracking positions remembered by any tokens inside the group. Atomic groups are non-capturing. The syntax is `(?>group)`. Lookaround groups are also atomic. Atomic grouping is supported by most modern regular expression flavors, including EditPad. EditPad also supports possessive quantifiers, which are essentially a notational convenience for atomic grouping.

An example will make the behavior of atomic groups clear. The regular expression `a(bc|b)c` (capturing group) matches `abcc` and `abc`. The regex `a(?>bc|b)c` (atomic group) matches `abcc` but not `abc`.

When applied to `abc`, both regexes will match `a` to `a`, `bc` to `bc`, and then `c` will fail to match at the end of the string. Here their paths diverge. The regex with the capturing group has remembered a backtracking position for the alternation. The group will give up its match, `b` then matches `b` and `c` matches `c`. Match found!

The regex with the atomic group, however, exited from an atomic group after `bc` was matched. At that point, all backtracking positions for tokens inside the group are discarded. In this example, the alternation's option to try `b` at the second position in the string is discarded. As a result, when `c` fails, the regex engine has no alternatives left to try.

Of course, the above example isn't very useful. But it does illustrate very clearly how atomic grouping eliminates certain matches. Or more importantly, it eliminates certain match attempts.

Regex Optimization Using Atomic Grouping

Consider the regex `\b(integer|insert|in)\b` and the subject `integers`. Obviously, because of the word boundaries, these don't match. What's not so obvious is that the regex engine will spend quite some effort figuring this out.

`\b` matches at the start of the string, and `integer` matches `integer`. The regex engine makes note that there are two more alternatives in the group, and continues with `\b`. This fails to match between the `r` and `s`. So the engine backtracks to try the second alternative inside the group. The second alternative matches `in`, but then fails to match `s`. So the engine backtracks once more to the third alternative. `in` matches `in`. `\b` fails between the `n` and `t` this time. The regex engine has no more remembered backtracking positions, so it declares failure.

This is quite a lot of work to figure out `integers` isn't in our list of words. We can optimize this by telling the regular expression engine that if it can't match `\b` after it matched `integer`, then it shouldn't bother trying any of the other words. The word we've encountered in the subject string is a longer word, and it isn't in our list.

We can do this by turning the capturing group into an atomic group: `\b(?>integer|insert|in)\b`. Now, when `integer` matches, the engine exits from an atomic group, and throws away the backtracking positions it stored for the alternation. When `\b` fails, the engine gives up immediately. This savings can be significant when scanning a large file for a long list of keywords. This savings will be vital when your alternatives contain repeated tokens (not to mention repeated groups) that lead to catastrophic backtracking.

Don't be too quick to make all your groups atomic. As we saw in the first example above, atomic grouping can exclude valid matches too. Compare how `\b(?:integer|insert|in)\b` and `\b(?:in|integer|insert)\b` behave when applied to `insert`. The former regex matches, while the latter fails. If the groups weren't atomic, both regexes would match. Remember that alternation tries its alternatives from left to right. If the second regex matches `in`, it won't try the two other alternatives due to the atomic group.

24. Possessive Quantifiers

The topic on repetition operators or quantifiers explains the difference between greedy and lazy repetition. Greediness and laziness determine the order in which the regex engine tries the possible permutations of the regex pattern. A greedy quantifier first tries to repeat the token as many times as possible, and gradually gives up matches as the engine backtracks to find an overall match. A lazy quantifier first repeats the token as few times as required, and gradually expands the match as the engine backtracks through the regex to find an overall match.

Because greediness and laziness change the order in which permutations are tried, they can change the overall regex match. However, they do not change the fact that the regex engine will backtrack to try all possible permutations of the regular expression in case no match can be found.

Possessive quantifiers are a way to prevent the regex engine from trying all permutations. This is primarily useful for performance reasons. You can also use possessive quantifiers to eliminate certain matches.

How Possessive Quantifiers Work

Like a greedy quantifier, a possessive quantifier repeats the token as many times as possible. Unlike a greedy quantifier, it does *not* give up matches as the engine backtracks. With a possessive quantifier, the deal is all or nothing. You can make a quantifier possessive by placing an extra + after it. * is greedy, *? is lazy, and *+ is possessive. ++, ?+ and {n,m}+ are all possessive as well.

Let's see what happens if we try to match "[^"]*" against "abc". The " matches the ". [^"] matches a, b and c as it is repeated by the star. The final " then matches the final " and we found an overall match. In this case, the end result is the same, whether we use a greedy or possessive quantifier. There is a slight performance increase though, because the possessive quantifier doesn't have to remember any backtracking positions.

The performance increase can be significant in situations where the regex fails. If the subject is "abc" (no closing quote), the above matching process happens in the same way, except that the second " fails. When using a possessive quantifier, there are no steps to backtrack to. The regular expression does not have any alternation or non-possessive quantifiers that can give up part of their match to try a different permutation of the regular expression. So the match attempt fails immediately when the second " fails.

Had we used "[^"]*" with a greedy quantifier instead, the engine would have backtracked. After the " failed at the end of the string, the [^"]* would give up one match, leaving it with ab. The " would then fail to match c. [^"]* backtracks to just a, and " fails to match b. Finally, [^"]* backtracks to match zero characters, and " fails a. Only at this point have all backtracking positions been exhausted, and does the engine give up the match attempt. Essentially, this regex performs as many needless steps as there are characters following the unmatched opening quote.

When Possessive Quantifiers Matter

The main practical benefit of possessive quantifiers is to speed up your regular expression. In particular, possessive quantifiers allow your regex to fail faster. In the above example, when the closing quote fails to match, we *know* the regular expression couldn't possibly have skipped over a quote. So there's no need to

backtrack and check for the quote. We make the regex engine aware of this by making the quantifier possessive. In fact, some engines, including EditPad's engine, detect that `[^"]*` and `"` are mutually exclusive when compiling your regular expression, and automatically make the star possessive.

Now, linear backtracking like a regex with a single quantifier does is pretty fast. It's unlikely you'll notice the speed difference. However, when you're nesting quantifiers, a possessive quantifier may save your day. Nesting quantifiers means that you have one or more repeated tokens inside a group, and the group is also repeated. That's when catastrophic backtracking often rears its ugly head. In such cases, you'll depend on possessive quantifiers and/or atomic grouping to save the day.

Possessive Quantifiers Can Change The Match Result

Using possessive quantifiers can change the result of a match attempt. Since no backtracking is done, and matches that would require a greedy quantifier to backtrack will not be found with a possessive quantifier. For example, `".*"` matches `"abc"` in `"abc"x`, but `".*+"` does not match this string at all.

In both regular expressions, the first `"` matches the first `"` in the string. The repeated dot then matches the remainder of the string `abc"x`. The second `"` then fails to match at the end of the string.

Now, the paths of the two regular expressions diverge. The possessive dot-star wants it all. No backtracking is done. Since the `"` failed, there are no permutations left to try, and the overall match attempt fails. The greedy dot-star, while initially grabbing everything, is willing to give back. It will backtrack one character at a time. Backtracking to `abc"`, `"` fails to match `x`. Backtracking to `abc`, `"` matches `"`. An overall match `"abc"` is found.

Essentially, the lesson here is that when using possessive quantifiers, you need to make sure that whatever you're applying the possessive quantifier to should not be able to match what should follow it. The problem in the above example is that the dot also matches the closing quote. This prevents us from using a possessive quantifier. The negated character class in the previous section cannot match the closing quote, so we can make it possessive.

25. Lookahead and Lookbehind Zero-Length Assertions

Lookahead and lookbehind, collectively called “lookaround”, are zero-length assertions just like the start and end of line, and start and end of word anchors explained earlier in this tutorial. The difference is that lookaround actually matches characters, but then gives up the match, returning only the result: match or no match. That is why they are called “assertions”. They do not consume characters in the string, but only assert whether a match is possible or not. Lookaround allows you to create regular expressions that are impossible to create without them, or that would get very longwinded without them.

Positive and Negative Lookahead

Negative lookahead is indispensable if you want to match something not followed by something else. When explaining character classes, this tutorial explained why you cannot use a negated character class to match a `q` not followed by a `u`. Negative lookahead provides the solution: `q(?:u)`. The negative lookahead construct is the pair of parentheses, with the opening parenthesis followed by a question mark and an exclamation point. Inside the lookahead, we have the trivial regex `u`.

Positive lookahead works just the same. `q(?:u)` matches a `q` that is followed by a `u`, without making the `u` part of the match. The positive lookahead construct is a pair of parentheses, with the opening parenthesis followed by a question mark and an equals sign.

You can use any regular expression inside the lookahead (but not lookbehind, as explained below). Any valid regular expression can be used inside the lookahead. If it contains capturing groups then those groups will capture as normal and backreferences to them will work normally, even outside the lookahead. The lookahead itself is not a capturing group. It is not included in the count towards numbering the backreferences. If you want to store the match of the regex inside a lookahead, you have to put capturing parentheses around the regex inside the lookahead, like this: `(?: (regex))`. The other way around will not work, because the lookahead will already have discarded the regex match by the time the capturing group is to store its match.

Regex Engine Internals

First, let’s see how the engine applies `q(?:!u)` to the string `Iraq`. The first token in the regex is the literal `q`. As we already know, this causes the engine to traverse the string until the `q` in the string is matched. The position in the string is now the void after the string. The next token is the lookahead. The engine takes note that it is inside a lookahead construct now, and begins matching the regex inside the lookahead. So the next token is `u`. This does not match the void after the string. The engine notes that the regex inside the lookahead failed. Because the lookahead is negative, this means that the lookahead has successfully matched at the current position. At this point, the entire regex has matched, and `q` is returned as the match.

Let’s try applying the same regex to `quit`. `q` matches `q`. The next token is the `u` inside the lookahead. The next character is the `u`. These match. The engine advances to the next character: `i`. However, it is done with the regex inside the lookahead. The engine notes success, and discards the regex match. This causes the engine to step back in the string to `u`.

Because the lookahead is negative, the successful match inside it causes the lookahead to fail. Since there are no other permutations of this regex, the engine has to start again at the beginning. Since `q` cannot match anywhere else, the engine reports failure.

Let's take one more look inside, to make sure you understand the implications of the lookahead. Let's apply `q(?:u)i` to `quit`. The lookahead is now positive and is followed by another token. Again, `q` matches `q` and `u` matches `u`. Again, the match from the lookahead must be discarded, so the engine steps back from `i` in the string to `u`. The lookahead was successful, so the engine continues with `i`. But `i` cannot match `u`. So this match attempt fails. All remaining attempts fail as well, because there are no more `q`'s in the string.

The regex `q(?:u)i` can never match anything. It tries to match `u` and `i` at the same position. If there is a `u` immediately after the `q` then the lookahead succeeds but then `i` fails to match `u`. If there is anything other than a `u` immediately after the `q` then the lookahead fails.

Positive and Negative Lookbehind

Lookbehind has the same effect, but works backwards. It tells the regex engine to temporarily step backwards in the string, to check if the text inside the lookbehind can be matched there. `(?<!a)b` matches a "b" that is not preceded by an "a", using negative lookbehind. It doesn't match `cab`, but matches the `b` (and only the `b`) in `bed` or `debt`. `(?<=a)b` (positive lookbehind) matches the `b` (and only the `b`) in `cab`, but does not match `bed` or `debt`.

The construct for positive lookbehind is `(?<=text)`: a pair of parentheses, with the opening parenthesis followed by a question mark, "less than" symbol, and an equals sign. Negative lookbehind is written as `(?<!\text)`, using an exclamation point instead of an equals sign.

More Regex Engine Internals

Let's apply `(?<=a)b` to `thingamabob`. The engine starts with the lookbehind and the first character in the string. In this case, the lookbehind tells the engine to step back one character, and see if `a` can be matched there. The engine cannot step back one character because there are no characters before the `t`. So the lookbehind fails, and the engine starts again at the next character, the `h`. (Note that a negative lookbehind would have succeeded here.) Again, the engine temporarily steps back one character to check if an "a" can be found there. It finds a `t`, so the positive lookbehind fails again.

The lookbehind continues to fail until the regex reaches the `m` in the string. The engine again steps back one character, and notices that the `a` can be matched there. The positive lookbehind matches. Because it is zero-length, the current position in the string remains at the `m`. The next token is `b`, which cannot match here. The next character is the second `a` in the string. The engine steps back, and finds out that the `m` does not match `a`.

The next character is the first `b` in the string. The engine steps back and finds out that `a` satisfies the lookbehind. `b` matches `b`, and the entire regex has been matched successfully. It matches one character: the first `b` in the string.

Important Notes About Lookbehind

You can use lookbehind anywhere in the regex, not only at the start. If you want to find a word not ending with an “s”, you could use `\b\w+(?<!s)\b`. This is definitely not the same as `\b\w+[^s]\b`. When applied to `John's`, the former matches `John` and the latter matches `John'` (including the apostrophe). I will leave it up to you to figure out why. (Hint: `\b` matches between the apostrophe and the `s`). The latter also doesn’t match single-letter words like “a” or “I”. The correct regex without using lookbehind is `\b\w*[^s\W]\b` (star instead of plus, and `\W` in the character class). Personally, I find the lookbehind easier to understand. The last regex, which works correctly, has a double negation (the `\W` in the negated character class). Double negations tend to be confusing to humans. Not to regex engines, though.

EditPad allows you to use a full regular expression inside lookbehind, including infinite repetition and backreferences. EditPad really applies the regex inside the lookbehind backwards, going through the regex inside the lookbehind and through the subject string from right to left. EditPad only needs to evaluate the lookbehind once, regardless of how many different possible lengths it has.

Lookaround Is Atomic

The fact that lookaround is zero-length automatically makes it atomic. As soon as the lookaround condition is satisfied, the regex engine forgets about everything inside the lookaround. It will not backtrack inside the lookaround to try different permutations.

The only situation in which this makes any difference is when you use capturing groups inside the lookaround. Since the regex engine does not backtrack into the lookaround, it will not try different permutations of the capturing groups.

For this reason, the regex `(?=(\d+))\w+\1` never matches `123x12`. First the lookahead captures `123` into `\1`. `\w+` then matches the whole string and backtracks until it matches only `1`. Finally, `\w+` fails since `\1` cannot be matched at any position. Now, the regex engine has nothing to backtrack to, and the overall regex fails. The backtracking steps created by `\d+` have been discarded. It never gets to the point where the lookahead captures only `12`.

Obviously, the regex engine does try further positions in the string. If we change the subject string, the regex `(?=(\d+))\w+\1` does match `56x56` in `456x56`.

If you don’t use capturing groups inside lookahead, then all this doesn’t matter. Either the lookahead condition can be satisfied or it cannot be. In how many ways it can be satisfied is irrelevant.

26. Testing The Same Part of a String for More Than One Requirement

Lookaround, which was introduced in detail in the previous topic, is a very powerful concept. Unfortunately, it is often underused by people new to regular expressions, because lookaround is a bit confusing. The confusing part is that the lookaround is zero-length. So if you have a regex in which a lookahead is followed by another piece of regex, or a lookbehind is preceded by another piece of regex, then the regex traverses part of the string twice.

A more practical example makes this clear. Let's say we want to find a word that is six letters long and contains the three consecutive letters `cat`. Actually, we can match this without lookaround. We just specify all the options and lump them together using alternation: `cat\w{3}|\wcat\w{2}|\w{2}cat\w|\w{3}cat`. Easy enough. But this method gets unwieldy if you want to find any word between 6 and 12 letters long containing either “cat”, “dog” or “mouse”.

Lookaround to The Rescue

In this example, we basically have two requirements for a successful match. First, we want a word that is 6 letters long. Second, the word we found must contain the word “cat”.

Matching a 6-letter word is easy with `\b\w{6}\b`. Matching a word containing “cat” is equally easy: `\b\w*cat\w*\b`.

Combining the two, we get: `(?=\b\w{6}\b)\b\w*cat\w*\b`. Easy! Here's how this works. At each character position in the string where the regex is attempted, the engine first attempts the regex inside the positive lookahead. This sub-regex, and therefore the lookahead, matches only when the current character position in the string is at the start of a 6-letter word in the string. If not, the lookahead fails and the engine continues trying the regex from the start at the next character position in the string.

The lookahead is zero-length. So when the regex inside the lookahead has found the 6-letter word, the current position in the string is still at the beginning of the 6-letter word. The regex engine attempts the remainder of the regex at this position. Because we already know that a 6-letter word can be matched at the current position, we know that `\b` matches and that the first `\w*` matches 6 times. The engine then backtracks, reducing the number of characters matched by `\w*`, until `cat` can be matched. If `cat` cannot be matched, the engine has no other choice but to restart at the beginning of the regex, at the next character position in the string. This is at the second letter in the 6-letter word we just found, where the lookahead will fail, causing the engine to advance character by character until the next 6-letter word.

If `cat` can be successfully matched, the second `\w*` consumes the remaining letters, if any, in the 6-letter word. After that, the last `\b` in the regex is guaranteed to match where the second `\b` inside the lookahead matched. Our double-requirement-regex has matched successfully.

Optimizing Our Solution

While the above regex works just fine, it is not the most optimal solution. This is not a problem if you are just doing a search in a text editor. But optimizing things is a good idea if this regex will be used repeatedly and/or on large chunks of data in an application you are developing.

You can discover these optimizations by yourself if you carefully examine the regex and follow how the regex engine applies it, as we did above. The third and last `\b` are guaranteed to match. Since word boundaries are zero-length, and therefore do not change the result returned by the regex engine, we can remove them, leaving: `(?=\b\b{6}\b)\w*cat\b`. Though the last `\w*` is also guaranteed to match, we cannot remove it because it adds characters to the regex match. Remember that the lookahead discards its match, so it does not contribute to the match returned by the regex engine. If we omitted the `\w*`, the resulting match would be the start of a 6-letter word containing “cat”, up to and including “cat”, instead of the entire word.

But we can optimize the first `\w*`. As it stands, it will match 6 letters and then backtrack. But we know that in a successful match, there can never be more than 3 letters before “cat”. So we can optimize this to `\w{0,3}`. Note that making the asterisk lazy would not have optimized this sufficiently. The lazy asterisk would find a successful match sooner, but if a 6-letter word does not contain “cat”, it would still cause the regex engine to try matching “cat” at the last two letters, at the last single letter, and even at one character beyond the 6-letter word.

So we have `(?=\b\b{6}\b)\w{0,3}cat\b`. One last, minor, optimization involves the first `\b`. Since it is zero-length itself, there’s no need to put it inside the lookahead. So the final regex is: `\b(?:\b{6}\b)\w{0,3}cat\b`.

You could replace the final `\w*` with `\w{0,3}` too. But it wouldn’t make any difference. The lookahead has already checked that we’re at a 6-letter word, and `\w{0,3}cat` has already matched 3 to 6 letters of that word. Whether we end the regex with `\w*` or `\w{0,3}` doesn’t matter, because either way, we’ll be matching all the remaining word characters. Because the resulting match and the speed at which it is found are the same, we may just as well use the version that is easier to type.

A More Complex Problem

So, what would you use to find any word between 6 and 12 letters long containing either “cat”, “dog” or “mouse”? Again we have two requirements, which we can easily combine using a lookahead: `\b(?:\w{6,12}\b)\w{0,9}(cat|dog|mouse)\w*`. Very easy, once you get the hang of it. This regex will also put “cat”, “dog” or “mouse” into the first backreference.

27. Keep The Text Matched So Far out of The Overall Regex Match

`\K` keeps the text matched so far out of the overall regex match. `h\Kd` matches only the second `d` in `adhd`.

This feature is not really needed in EditPad because it supports unrestricted lookbehind, which is much more flexible than `\K`. Still, EditPad supports `\K` if you prefer this way of working.

Looking Inside The Regex Engine

Let's see how `h\Kd` works. The engine begins the match attempt at the start of the string. `h` fails to match `a`. There are no further alternatives to try. The match attempt at the start of the string has failed.

The engine advances one character through the string and attempts the match again. `h` fails to match `d`.

Advancing again, `h` matches `h`. The engine advances through the regex. The regex has now reached `\K` in the regex and the position between `h` and the second `d` in the string. `\K` does nothing other than to tell that if this match attempt ends up succeeding, the regex engine should pretend that the match attempt started at the present position between `h` and `d`, rather than between the first `d` and `h` where it really started.

The engine advances through the regex. `d` matches the second `d` in the string. An overall match is found. Because of the position saved by `\K`, the second `d` in the string is returned as the overall match.

`\K` only affects the position returned after a successful match. It does not move the start of the match attempt during the matching process. The regex `hhh\Kd` matches the `d` in `hhhd`. This regex first matches `hhh` at the start of the string. Then `\K` notes the position between `hhh` and `hd` in the string. Then `d` fails to match the fourth `h` in the string. The match attempt at the start of the string has failed.

Now the engine must advance one character in the string before starting the next match attempt. It advances from the actual start of the match attempt, which was at the start of the string. The position stored by `\K` does not change this. So the second match attempt begins at the position after the first `h` in the string. Starting there, `hhh` matches `hhh`, `\K` notes the position, and `d` matches `d`. Now, the position remembered by `\K` is taken into account, and `d` is returned as the overall match.

`\K` Can Be Used Anywhere

You can use `\K` pretty much anywhere in any regular expression. You should only avoid using it inside lookbehind. You can use it inside groups, even when they have quantifiers. You can have as many instances of `\K` in your regex as you like. `(ab\Kc|d\Ke)f` matches `cf` when preceded by `ab`. It also matches `ef` when preceded by `d`.

`\K` does not affect capturing groups. When `(ab\Kc|d\Ke)f` matches `cf`, the capturing group captures `abc` as if the `\K` weren't there. When the regex matches `ef`, the capturing group stores `de`.

Limitations of \K

Lookbehind really goes backwards through the string. This allows lookbehind check for a match before the start of the match attempt. When the match attempt was started at the end of the previous match, lookbehind can match text that was part of the previous match. `\K` cannot do this, because it does not affect the way the regex engine goes through the matching process.

If you iterate over all matches of `(?<=a)a` in the string `aaaa`, you will get three matches: the second, third, and fourth `a` in the string. The first match attempt begins at the start of the string and fails because the lookbehind fails. The second match attempt begins between the first and second `a`, where the lookbehind succeeds and the second `a` is matched. The third match attempt begins after the second `a` that was just matched. Here the lookbehind succeeds too. It doesn't matter that the preceding `a` was part of the previous match. Thus the third match attempt matches the third `a`. Similarly, the fourth match attempt matches the fourth `a`. The fifth match attempt starts at the end of the string. The lookbehind still succeeds, but there are no characters left for `a` to match. The match attempt fails. The engine has reached the end of the string and the iteration stops. Five match attempts have found three matches.

Things are different when you iterate over `a\Ka` in the string `aaaa`. You will get only two matches: the second and the fourth `a`. The first match attempt begins at the start of the string. The first `a` in the regex matches the first `a` in the string. `\K` notes the position. The second `a` matches the second `a` in the string, which is returned as the first match. The second match attempt begins after the second `a` that was just matched. The first `a` in the regex matches the third `a` in the string. `\K` notes the position. The second `a` matches the fourth `a` in the string, which is returned as the first match. The third match attempt begins at the end of the string. `a` fails. The engine has reached the end of the string and the iteration stops. Three match attempts have found two matches.

Basically, you'll run into this issue when the part of the regex before the `\K` can match the same text as the part of the regex after the `\K`. If those parts can't match the same text, then a regex using `\K` will find the same matches than the same regex rewritten using lookbehind. In that case, you should use `\K` instead of lookbehind as that will give you better performance in Perl, PCRE, and Ruby.

Another limitation is that while lookbehind comes in positive and negative variants, `\K` does not provide a way to negate anything. `(?!a)b` matches the string `b` entirely, because it is a "b" not preceded by an "a". `[^a]\Kb` does not match the string `b` at all. When attempting the match, `[^a]` matches `b`. The regex has now reached the end of the string. `\K` notes this position. But now there is nothing left for `b` to match. The match attempt fails. `[^a]\Kb` is the same as `(?<=[^a])b`, which are both different from `(?!a)b`.

28. If-Then-Else Conditionals in Regular Expressions

A special construct (`?ifthen|else`) allows you to create conditional regular expressions. If the *if* part evaluates to true, then the regex engine will attempt to match the *then* part. Otherwise, the *else* part is attempted instead. The syntax consists of a pair of parentheses. The opening bracket must be followed by a question mark, immediately followed by the *if* part, immediately followed by the *then* part. This part can be followed by a vertical bar and the *else* part. You may omit the *else* part, and the vertical bar with it.

For the *if* part, you can use the lookahead and lookbehind constructs. Using positive lookahead, the syntax becomes `(?(?=regex)then|else)`. Because the lookahead has its own parentheses, the *if* and *then* parts are clearly separated.

Remember that the lookahead constructs do not consume any characters. If you use a lookahead as the *if* part, then the regex engine will attempt to match the *then* or *else* part (depending on the outcome of the lookahead) at the same position where the *if* was attempted.

Alternatively, you can check in the *if* part whether a capturing group has taken part in the match thus far. Place the number of the capturing group inside parentheses, and use that as the *if* part. Note that although the syntax for a conditional check on a backreference is the same as a number inside a capturing group, no capturing group is created. The number and the parentheses are part of the if-then-else syntax started with `(?`.

For the *then* and *else*, you can use any regular expression. If you want to use alternation, you will have to group the *then* or *else* together using parentheses, like in `(?(?=condition)(then1|then2|then3)|else1|else2|else3))`. Otherwise, there is no need to use parentheses around the *then* and *else* parts.

Looking Inside The Regex Engine

The regex `(a)?b(?(1)c|d)` consists of the optional capturing group `(a)?`, the literal `b`, and the conditional `(?(1)c|d)` that tests the capturing group. This regex matches `bd` and `abc`. It does not match `bc`, but does match `bd` in `abd`. Let's see how this regular expression works on each of these four subject strings.

When applied to `bd`, `a` fails to match. Since the capturing group containing `a` is optional, the engine continues with `b` at the start of the subject string. Since the whole group was optional, the group did not take part in the match. Any subsequent backreference to it like `\1` will fail. Note that `(a)?` is very different from `(a?)`. In the former regex, the capturing group does not take part in the match if `a` fails, and backreferences to the group will fail. In the latter group, the capturing group always takes part in the match, capturing either `a` or nothing. Backreferences to a capturing group that took part in the match and captured nothing always succeed. Conditionals evaluating such groups execute the “then” part. In short: if you want to use a reference to a group in a conditional, use `(a)?` instead of `(a?)`.

Continuing with our regex, `b` matches `b`. The regex engine now evaluates the conditional. The first capturing group did not take part in the match at all, so the “else” part or `d` is attempted. `d` matches `d` and an overall match is found.

Moving on to our second subject string `abc`, `a` matches `a`, which is captured by the capturing group. Subsequently, `b` matches `b`. The regex engine again evaluates the conditional. The capturing group took part in the match, so the “then” part or `c` is attempted. `c` matches `c` and an overall match is found.

Our third subject `bc` does not start with `a`, so the capturing group does not take part in the match attempt, like we saw with the first subject string. `b` still matches `b`, and the engine moves on to the conditional. The first capturing group did not take part in the match at all, so the “else” part or `d` is attempted. `d` does not match `c` and the match attempt at the start of the string fails. The engine does try again starting at the second character in the string, but fails since `b` does not match `c`.

The fourth subject `abd` is the most interesting one. Like in the second string, the capturing group grabs the `a` and the `b` matches. The capturing group took part in the match, so the “then” part or `c` is attempted. `c` fails to match `d`, and the match attempt fails. Note that the “else” part is not attempted at this point. The capturing group took part in the match, so only the “then” part is used. However, the regex engine isn’t done yet. It restarts the regular expression from the beginning, moving ahead one character in the subject string.

Starting at the second character in the string, `a` fails to match `b`. The capturing group does not take part in the second match attempt which started at the second character in the string. The regex engine moves beyond the optional group, and attempts `b`, which matches. The regex engine now arrives at the conditional in the regex, and at the third character in the subject string. The first capturing group did not take part in the current match attempt, so the “else” part or `d` is attempted. `d` matches `d` and an overall match `bd` is found.

If you want to avoid this last match result, you need to use anchors. `^(a)?b(?:c|d)$` does not find any matches in the last subject string. The caret fails to match before the second and third characters in the string.

Named and Relative Conditionals

You can use the name of a capturing group instead of its number as the *if* test. The syntax is slightly inconsistent between regex flavors. Simply specify the name of the group between parentheses. `(?<test>a)?b(?:c|d)` is the regex from the previous section using named capture.

EditPad also supports relative conditionals. The syntax is the same as that of a conditional that references a numbered capturing group with an added plus or minus sign before the group number. The conditional then counts the opening parentheses to the left (minus) or to the right (plus) starting at the `(?(` that opens the conditional. `(a)?b(?:(-1)c|d)` is another way of writing the above regex. The benefit is that this regex won’t break if you add capturing groups at the start or the end of the regex.

Conditionals Referencing Non-Existent Capturing Groups

Conditionals that reference a non-existent capturing group are not an error. They always attempt the “else” part.

Example: Extract Email Headers

The regex `^((From|To|Subject): ((?(2)\w+@\w+\.[a-z]+\.[a-z]+))` extracts the From, To, and Subject headers from an email message. The name of the header is captured into the first backreference. If the header is the From or To header, it is captured into the second backreference as well.

The second part of the pattern is the if-then-else conditional `((?(2)\w+@\w+\.[a-z]+\.[a-z]+))`. The *if* part checks whether the second capturing group took part in the match thus far. It will have taken part if the header is the From or To header. In that case, the *then* part of the conditional `\w+@\w+\.[a-z]+\.[a-z]+` tries to match an email address. To keep the example simple, we use an overly simple regex to match the email address, and we don't try to match the display name that is usually also part of the From or To header.

If the second capturing group did not participate in the match this far, the *else* part `.` is attempted instead. This simply matches the remainder of the line, allowing for any test subject.

Finally, we place an extra pair of parentheses around the conditional. This captures the contents of the email header matched by the conditional into the third backreference. The conditional itself does not capture anything. When implementing this regular expression, the first capturing group will store the name of the header ("From", "To", or "Subject"), and the third capturing group will store the value of the header.

You could try to match even more headers by putting another conditional into the "else" part. E.g. `^((From|To|Date|Subject): ((?(2)\w+@\w+\.[a-z]+\.[a-z]+)(?(3)mm/dd/yyyy\.[a-z]+\.[a-z]+))` would match a "From", "To", "Date" or "Subject", and use the regex `mm/dd/yyyy` to check whether the date is valid. Obviously, the date validation regex is just a dummy to keep the example simple. The header is captured in the first group, and its validated contents in the fourth group.

As you can see, regular expressions using conditionals quickly become unwieldy. I recommend that you only use them if one regular expression is all your tool allows you to use. When programming, you're far better off using the regex `^((From|To|Date|Subject): (.+))` to capture one header with its unvalidated contents. In your source code, check the name of the header returned in the first capturing group, and then use a second regular expression to validate the contents of the header returned in the second capturing group of the first regex. Though you'll have to write a few lines of extra code, this code will be much easier to understand and maintain. If you precompile all the regular expressions, using multiple regular expressions will be just as fast, if not faster, than the one big regex stuffed with conditionals.

29. Matching Nested Constructs with Balancing Groups

EditPad has a special feature called balancing groups. The main purpose of balancing groups is to match balanced constructs or nested constructs, which is where they get their name from. A technically more accurate name for the feature would be capturing group subtraction. That’s what the feature really does. This feature was first invented by the .NET regex flavor, which does not support regular expression recursion.

(?<capture-subtract>regex) or (?’capture-subtract’regex) is the basic syntax of a balancing group. It’s the same syntax used for named capturing groups but with two group names delimited by a minus sign. The name of this group is “capture”. You can omit the name of the group. (?<-subtract>regex) or (?’-subtract’regex) is the syntax for a non-capturing balancing group.

The name “subtract” must be the name of another group in the regex. When the regex engine enters the balancing group, it subtracts one match from the group “subtract”. If the group “subtract” did not match yet, or if all its matches were already subtracted, then the balancing group fails to match. You could think of a balancing group as a conditional that tests the group “subtract”, with “regex” as the “if” part and an “else” part that always fails to match. The difference is that the balancing group has the added feature of subtracting one match from the group “subtract”, while a conditional leaves the group untouched.

If the balancing group succeeds and it has a name (“capture” in this example), then the group captures the text between the end of the match that was subtracted from the group “subtract” and the start of the match of the balancing group itself (“regex” in this example).

Looking Inside The Regex Engine

Let’s apply the regex `(?'open'o)+(?'between-open'c)+` to the string `ooccc`. `(?'open'o)` matches the first `o` and stores that as the first capture of the group “open”. The quantifier `+` repeats the group. `(?'open'o)` matches the second `o` and stores that as the second capture. Repeating again, `(?'open'o)` fails to match the first `c`. But the `+` is satisfied with two repetitions.

The regex engine advances to `(?'between-open'c)`. Before the engine can enter this balancing group, it must check whether the subtracted group “open” has captured something. It has captured the second `o`. The engine enters the group, subtracting the most recent capture from “open”. This leaves the group “open” with the first `o` as its only capture. Now inside the balancing group, `c` matches `c`. The engine exits the balancing group. The group “between” captures the text between the match subtracted from “open” (the second `o`) and the `c` just matched by the balancing group. This is an empty string but it is captured anyway.

The balancing group too has `+` as its quantifier. The engine again finds that the subtracted group “open” captured something, namely the first `o`. The regex enters the balancing group, leaving the group “open” without any matches. `c` matches the second `c` in the string. The group “between” captures `oc` which is the text between the match subtracted from “open” (the first `o`) and the second `c` just matched by the balancing group.

The balancing group is repeated again. But this time, the regex engine finds that the group “open” has no matches left. The balancing group fails to match. The group “between” is unaffected, retaining its most recent capture.

The `+` is satisfied with two iterations. The engine has reached the end of the regex. It returns `ooCC` as the overall match. `Match.Groups['open'].Success` will return `false`, because all the captures of that group were subtracted. `Match.Groups['between'].Value` returns `"oc"`.

Matching Balanced Pairs

We need to modify this regex if we want it to match a balanced number of o's and c's. To make sure that the regex won't match `ooCCC`, which has more c's than o's, we can add anchors: `^(?'open'o)+(?'-open'c)+$`. This regex goes through the same matching process as the previous one. But after `(?'-open'c)+` fails to match its third iteration, the engine reaches `$` instead of the end of the regex. This fails to match. The regex engine will backtrack trying different permutations of the quantifiers, but they will all fail to match. No match can be found.

But the regex `^(?'open'o)+(?'-open'c)+$` still matches `ooC`. The matching process is again the same until the balancing group has matched the first `c` and left the group `'open'` with the first `o` as its only capture. The quantifier makes the engine attempt the balancing group again. The engine again finds that the subtracted group `"open"` captured something. The regex enters the balancing group, leaving the group `"open"` without any matches. But now, `c` fails to match because the regex engine has reached the end of the string.

The regex engine must now backtrack out of the balancing group. When backtracking a balancing group, EditPad also backtracks the subtraction. Since the capture of the first `o` was subtracted from `"open"` when entering the balancing group, this capture is now restored while backtracking out of the balancing group. The repeated group `(?'-open'c)+` is now reduced to a single iteration. But the quantifier is fine with that, as `+` means "once or more" as it always does. Still at the end of the string, the regex engine reaches `$` in the regex, which matches. The whole string `ooC` is returned as the overall match. `Match.Groups['open'].Captures` will hold the first `o` in the string as the only item in the `CaptureCollection`. That's because, after backtracking, the second `o` was subtracted from the group, but the first `o` was not.

To make sure the regex matches `oc` and `ooCC` but not `ooC`, we need to check that the group `"open"` has no captures left when the matching process reaches the end of the regex. We can do this with a conditional. `(?(open)(?!))` is a conditional that checks whether the group `"open"` matched something. In EditPad, having matched something means still having captures on the stack that weren't backtracked or subtracted. If the group has captured something, the "if" part of the conditional is evaluated. In this case that is the empty negative lookahead `(?!)`. The empty string inside this lookahead always matches. Because the lookahead is negative, this causes the lookahead to always fail. Thus the conditional always fails if the group has captured something. If the group has not captured anything, the "else" part of the conditional is evaluated. In this case there is no "else" part. This means that the conditional always succeeds if the group has not captured something. This makes `(?(open)(?!))` a proper test to verify that the group `"open"` has no captures left.

The regex `^(?'open'o)+(?'-open'c)+(?(open)(?!))$` fails to match `ooC`. When `c` fails to match because the regex engine has reached the end of the string, the engine backtracks out of the balancing group, leaving `"open"` with a single capture. The regex engine now reaches the conditional, which fails to match. The regex engine will backtrack trying different permutations of the quantifiers, but they will all fail to match. No match can be found.

The regex `^(?'open'o)+(?'-open'c)+(?(open)(?!))$` does match `ooCC`. After `(?'-open'c)+` has matched `CC`, the regex engine cannot enter the balancing group a third time, because `"open"` has no captures

left. The engine advances to the conditional. The conditional succeeds because “open” has no captures left and the conditional does not have an “else” part. Now `$` matches at the end of the string.

Matching Balanced Constructs

`^(?: (?: 'open' o)+ (?: ' -open' c)+)+ (?: (open) (?!)) $` wraps the capturing group and the balancing group in a non-capturing group that is also repeated. This regex matches any string like `000c00cc0cc0c` that contains any number of perfectly balanced o’s and c’s, with any number of pairs in sequence, nested to any depth. The balancing group makes sure that the regex never matches a string that has more c’s at any point in the string than it has o’s to the left of that point. The conditional at the end, which must remain outside the repeated group, makes sure that the regex never matches a string that has more o’s than c’s.

`^(?>(?: 'open' o)+ (?: ' -open' c)+)+ (?: (open) (?!)) $` optimizes the previous regex by using an atomic group instead of the non-capturing group. The atomic group, which is also non-capturing, eliminates nearly all backtracking when the regular expression cannot find a match, which can greatly increase performance when used on long strings with lots of o’s and c’s but that aren’t properly balanced at the end. The atomic group does not change how the regex matches strings that do have balanced o’s and c’s.

`^m*(?>(?: 'open' o)m*)+ (?: ' -open' c)m*)+ (?: (open) (?!)) $` allows any number of letters `m` anywhere in the string, while still requiring all o’s and c’s to be balanced. `m*` at the start of the regex allows any number of m’s before the first o. `(?: 'open' o)+` was changed into `(?>(?: 'open' o)m*)+` to allow any number of m’s after each o. Similarly, `(?: ' -open' c)+` was changed into `(?>(?: ' -open' c)m*)+` to allow any number of m’s after each c.

This is the generic solution for matching balanced constructs using EditPad’s balancing groups or capturing group subtraction feature. You can replace `o`, `m`, and `c` with any regular expression, as long as no two of these three can match the same text.

`^[^()]* (?>(?: 'open' \([^()]*)+ (?: ' -open' \) [^()]*)+)+ (?: (open) (?!)) $` applies this technique to match a string in which all parentheses are perfectly balanced.

Backreferences To Subtracted Groups

You can use backreferences to groups that have their matches subtracted by a balancing group. The backreference matches the group’s most recent match that wasn’t backtracked or subtracted. The regex `(?: 'x' [ab]) {2} (?: ' -x') \k'x'` matches `aaa`, `aba`, `bab`, or `bbb`. It does not match `aab`, `abb`, `baa`, or `bba`. The first and third letters of the string have to be the same.

Let’s see how `(?: 'x' [ab]) {2} (?: ' -x') \k'x'` matches `aba`. The first iteration of `(?: 'x' [ab])` captures `a`. The second iteration captures `b`. Now the regex engine reaches the balancing group `(?: ' -x')`. It checks whether the group “x” has matched, which it has. The engine enters the balancing group, subtracting the match `b` from the stack of group “x”. There are no regex tokens inside the balancing group. It matches without advancing through the string. Now the regex engine reaches the backreference `\k'x'`. The match at the top of the stack of group “x” is `a`. The next character in the string is also an `a` which the backreference matches. `aba` is found as an overall match.

When you apply this regex to `abb`, the matching process is the same, except that the backreference fails to match the second `b` in the string. Since the regex has no other permutations that the regex engine can try, the match attempt fails.

Matching Palindromes

`^(? 'letter' [a-z])+[a-z]?(?:\k'letter'(?'-letter'))+(?(letter)(?!))$` matches palindrome words of any length. This regular expression takes advantage of the fact that backreferences and capturing group subtraction work well together. It also uses an empty balancing group as the regex in the previous section.

Let's see how this regex matches the palindrome `radar`. `^` matches at the start of the string. Then `(? 'letter' [a-z])+` iterates five times. The group "letter" ends up with five matches on its stack: `r`, `a`, `d`, `a`, and `r`. The regex engine is now at the end of the string and at `[a-z]?` in the regex. It doesn't match, but that's fine, because the quantifier makes it optional. The engine now reaches the backreference `\k'letter'`. The group "letter" has `r` at the top of its stack. This fails to match the void after the end of the string.

The regex engine backtracks. `(? 'letter' [a-z])+` is reduced to four iterations, leaving `r`, `a`, `d`, and `a` on the stack of the group "letter". `[a-z]?` matches `r`. The backreference again fails to match the void after the end of the string. The engine backtracks, forcing `[a-z]?` to give up its match. Now "letter" has `a` at the top of its stack. This causes the backreference to fail to match `r`.

More backtracking follows. `(? 'letter' [a-z])+` is reduced to three iterations, leaving `d` at the top of the stack of the group "letter". The engine again proceeds with `[a-z]?`. It fails again because there is no `d` for the backreference to match.

Backtracking once more, the capturing stack of group "letter" is reduced to `r` and `a`. Now the tide turns. `[a-z]?` matches `d`. The backreference matches `a` which is the most recent match of the group "letter" that wasn't backtracked. The engine now reaches the empty balancing group `(?'-letter')`. This matches, because the group "letter" has a match `a` to subtract.

The backreference and balancing group are inside a repeated non-capturing group, so the engine tries them again. The backreference matches `r` and the balancing group subtracts it from "letter"'s stack, leaving the capturing group without any matches. Iterating once more, the backreference fails, because the group "letter" has no matches left on its stack. This makes the group act as a non-participating group. Backreferences to non-participating groups always fail in EditPad.

`(?:\k'letter'(?'-letter'))+` has successfully matched two iterations. Now, the conditional `(?(letter)(?!))` succeeds because the group "letter" has no matches left. The anchor `$` also matches. The palindrome `radar` has been matched.

30. Regular Expression Recursion

`(?R)`, `(?0)`, and `\g<0>` are three different ways to specify regular expression recursion. EditPad supports these syntactic variations for maximum compatibility with other regular expression flavors. As we'll see later, there are differences in how EditPad deals with backreferences and backtracking during recursion, depending on the syntax you choose. But these differences do not come into play in the basic example on this page.

Simple Recursion

The regexes `a(?R)?z`, `a(?0)?z`, and `a\g<0>?z` all match one or more letters `a` followed by exactly the same number of letters `z`. Since these regexes are functionally identical, we'll use the syntax with `R` for recursion to see how this regex matches the string `aaazzz`.

First, `a` matches the first `a` in the string. Then the regex engine reaches `(?R)`. This tells the engine to attempt the whole regex again at the present position in the string. Now, `a` matches the second `a` in the string. The engine reaches `(?R)` again. On the second recursion, `a` matches the third `a`. On the third recursion, `a` fails to match the first `z` in the string. This causes `(?R)` to fail. But the regex uses a quantifier to make `(?R)` optional. So the engine continues with `z` which matches the first `z` in the string.

Now, the regex engine has reached the end of the regex. But since it's two levels deep in recursion, it hasn't found an overall match yet. It only has found a match for `(?R)`. Exiting the recursion after a successful match, the engine also reaches `z`. It now matches the second `z` in the string. The engine is still one level deep in recursion, from which it exits with a successful match. Finally, `z` matches the third `z` in the string. The engine is again at the end of the regex. This time, it's not inside any recursion. Thus, it returns `aaazzz` as the overall regex match.

Matching Balanced Constructs

The main purpose of recursion is to match balanced constructs or nested constructs. The generic regex is `b(?:m|(?R))*e` where `b` is what begins the construct, `m` is what can occur in the middle of the construct, and `e` is what can occur at the end of the construct. For correct results, no two of `b`, `m`, and `e` should be able to match the same text. You can use an atomic group instead of the non-capturing group for improved performance: `b(?:>m|(?R))*e`.

A common real-world use is to match a balanced set of parentheses. `\((?>[^\(\)]|(?R))*\)` matches a single pair of parentheses with any text in between, including an unlimited number of parentheses, as long as they are all properly paired. If the subject string contains unbalanced parentheses, then the first regex match is the leftmost pair of balanced parentheses, which may occur after unbalanced opening parentheses. If you want a regex that does not find any matches in a string that contains unbalanced parentheses, then you need to use a subroutine call instead of recursion. If you want to find a sequence of multiple pairs of balanced parentheses as a single match, then you also need a subroutine call.

If what may appear in the middle of the balanced construct may also appear on its own without the beginning and ending parts then the generic regex is `b(?R)*e|m`. Again, `b`, `m`, and `e` all need to be mutually exclusive. `\((?R)*\)|[^\(\)]+` matches a pair of balanced parentheses like the regex in the previous section. But it also matches any text that does not contain any parentheses at all.

31. Regular Expression Subroutines

Subroutine calls are very similar to regular expression recursion. Instead of matching the entire regular expression again, a subroutine call only matches the regular expression inside a capturing group. You can make a subroutine call to any capturing group from anywhere in the regex. If you place a call inside the group that it calls, you'll have a recursive capturing group.

As with regex recursion, there is a wide variety of syntax that you can use for exactly the same thing. One set of syntax, borrowed from Perl, uses `(?1)` to call a numbered group, `(?+1)` to call the next group, `(?-1)` to call the preceding group, and `(?&name)` to call a named group. You can use all of these to reference the same group. `(?+1)(?'name'[abc])(?1)(?-1)(?&name)` matches a string that is five letters long and consists only of the first three letters of the alphabet. This regex is exactly the same as `[abc](?'name'[abc])[abc][abc][abc]`.

The second set of syntax, borrowed from PCRE and expanded upon, uses `(?P>1)` to call a numbered group, `(?P>+1)` to call the next group, `(?P>-1)` to call the preceding group, and `(?P>name)` to call a named group. You can use all of these to reference the same group. `(?P>+1)(?P<name>[abc])(?P>1)(?P>-1)(?P>name)` matches a string that is five letters long and consists only of the first three letters of the alphabet. This regex is exactly the same as `[abc](?P<name>[abc])[abc][abc][abc]`.

The third set of syntax, borrowed from Ruby, looks more like that of backreferences. `\g<1>` and `\g'1'` call a numbered group, `\g<name>` and `\g'name'` call a named group, while `\g<-1>` and `\g'-1'` call the preceding group. `\g<+1>` and `\g'+1'` call the next group. `\g<+1>(?<name>[abc])\g<1>\g<-1>\g<name>` and `\g'+1'(?'name'[abc])\g'1'\g'-1'\g'name'` match the same 5-letter string as the preceding examples. The syntax with angle brackets and with quotes can be used interchangeably.

EditPad not only has three sets of syntax for subroutine calls, it also has three different behaviors for subroutine calls that mimic the behavior of Perl, PCRE, or Ruby, depending on the syntax you choose. This changes how EditPad deals with capturing, backreferences, and backtracking during subroutine calls. But these differences do not come into play in the basic examples on this page.

Matching Balanced Constructs

Recursion into a capturing group is a more flexible way of matching balanced constructs than recursion of the whole regex. We can wrap the regex in a capturing group, recurse into the capturing group instead of the whole regex, and add anchors outside the capturing group. `\A(b(?>m|(?1))*e)\Z` is the generic regex for checking that a string consists entirely of a correctly balanced construct. Again, `b` is what begins the construct, `m` is what can occur in the middle of the construct, and `e` is what can occur at the end of the construct. For correct results, no two of `b`, `m`, and `e` should be able to match the same text. You can use an atomic group instead of the non-capturing group for improved performance: `\A(b(?>m|(?1))*e)\Z`.

Similarly, `\Ao*(b(?>m|(?1))*eo*)++\Z` and the optimized `\Ao*(b(?>m|(?1))*eo*)++\Z` match a string that consists of nothing but a sequence of one or more correctly balanced constructs, with possibly other text in between. Here, `o` is what can occur outside the balanced constructs. It will often be the same as `m`. `o` should not be able to match the same text as `b` or `e`.

`\A(\((?>[^\(\)]|(?1))*\))\Z` matches a string that consists of nothing but a correctly balanced pair of parentheses, possibly with text between them. `\A[^\(\)]*+(\((?>[^\(\)]|(?1))*+\)[^\(\)]*++)\Z` also allows text before the first opening parenthesis and after the last closing parenthesis in the string.

Matching The Same Construct More Than Once

A regex that needs to match the same kind of construct (but not the exact same text) more than once in different parts of the regex can be shorter and more concise when using subroutine calls. Suppose you need a regex to match patient records like these:

```
Name: John Doe
Born: 17-Jan-1964
Admitted: 30-Jul-2013
Released: 3-Aug-2013
```

Further suppose that you need to match the date format rather accurately so the regex can filter out valid records, leaving invalid records for human inspection. In most regex flavors you could easily do this with this regex, using free-spacing syntax:

```
^Name: \ (.*?)\r?\n
Born: \ (?:3[01]||[12][0-9]||[1-9])
      - (?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
      - (?:19|20)[0-9][0-9]\r?\n
Admitted: \ (?:3[01]||[12][0-9]||[1-9])
           - (?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
           - (?:19|20)[0-9][0-9]\r?\n
Released: \ (?:3[01]||[12][0-9]||[1-9])
           - (?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
           - (?:19|20)[0-9][0-9]$
```

With subroutine calls you can make this regex much shorter, easier to read, and easier to maintain:

```
^Name: \ (.*?)\r?\n
Born: \ (? 'date' (?:3[01]||[12][0-9]||[1-9])
      - (?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
      - (?:19|20)[0-9][0-9])\r?\n
Admitted: \ \g'date'\r?\n
Released: \ \g'date'$
```

Separate Subroutine Definitions

You can take this one step further using the special `DEFINE` group: `(?(DEFINE)(?'subroutine'regex))`. While this looks like a conditional that references the non-existent group `DEFINE` containing a single named group “subroutine”, the `DEFINE` group is a special syntax. The fixed text `(?(DEFINE)` opens the group. A parenthesis closes the group. This special group tells the regex engine to ignore its contents, other than to parse it for named and numbered capturing groups. You can put as many capturing groups inside the `DEFINE` group as you like. The `DEFINE` group itself never matches anything, and never fails to match. It is completely ignored. The regex

foo(? (DEFINE) (? 'subroutine' skipped)) bar matches foobar. The DEFINE group is completely superfluous in this regex, as there are no calls to any of the groups inside it.

With a DEFINE group, our regex becomes:

```
(? (DEFINE) (? 'date' (? 3 [01] [12] [0-9] [1-9])
- (? : Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)
- (? : 19 20 [0-9] [0-9]) ) )
^Name: \ (.* ) \r? \n
Born: \ ( ?P>date ) \r? \n
Admitted: \ ( ?P>date ) \r? \n
Released: \ ( ?P>date ) $
```

Quantifiers On Subroutine Calls

Quantifiers on subroutine calls work just like a quantifier on recursion. The call is repeated as many times in sequence as needed to satisfy the quantifier. ([abc])(?1){3} matches abcb and any other combination of four-letter combination of the first three letters of the alphabet. First the group matches once, and then the call matches three times. This regex is equivalent to ([abc])[abc]{3}.

Quantifiers on the group are ignored by the subroutine call. ([abc]){3}(?1) also matches abcb. First, the group matches three times, because it has a quantifier. Then the subroutine call matches once, because it has no quantifier. ([abc]){3}(?1){3} matches six letters, such as abbcab, because now both the group and the call are repeated 3 times. These two regexes are equivalent to ([abc]){3}[abc] and ([abc]){3}[abc]{3}.

32. Infinite Recursion

Regular expressions such as `(?R)?z` or `a?(?R)?z` or `a|(?R)z` that use recursion without having anything that must be matched in front of the recursion can result in infinite recursion. If the regex engine reaches the recursion without having advanced through the text then the next recursion will again reach the recursion without having advanced through the text. With the first regex this happens immediately at the start of the match attempt. With the other two this happens as soon as there are no further letters `a` to be matched.

EditPad treats the first two regexes as a syntax error because they always lead to infinite recursion. It allows the third regex because that one can match `a`.

Circular Infinite Subroutine Calls

Subroutine calls can also lead to infinite recursion. EditPad handles the potentially infinite recursion in `((?1)?z)` or `(a?(?1)?z)` or `(a|(?1)z)` in the same way as they handle potentially infinite recursion of the entire regex.

But subroutine calls that are not recursive by themselves may end up being recursive if the group they call has another subroutine call that calls a parent group of the first subroutine call. When subroutine calls are forced to go around in a circle that too leads to infinite recursion. Detecting such circular calls when compiling a regex is more complicated than checking for straight infinite recursion. Unlike most other applications, EditPad is able to detect this and treat it as a syntax error.

When infinite recursion does occur, whether it's straight recursion or subroutine calls going in circles, EditPad treats it as a matching error that aborts the entire match attempt.

Endless Recursion

A regex such as `a(?R)z` that has a recursion token that is not optional and is not have an alternative without the same recursion leads to endless recursion. Such a regular expression can never find a match. When `a` matches the regex engine attempts the recursion. If it can match another `a` then it has to attempt the recursion again. Eventually `a` will run out of letters to match. The recursion then fails. Because it's not optional the regex fails to match.

EditPad detects this situation when compiling your regular expression. It flags endless recursion as a syntax error.

33. Quantifiers On Recursion

The introduction to recursion shows how `a(?R)?z` matches `aaazzz`. The quantifier `?` makes the preceding token optional. In other words, it repeats the token between zero or one times. In `a(?R)?z` the `(?R)` is made optional by the `?` that follows it. You may wonder why the regex attempted the recursion three times, instead of once or not at all.

The reason is that upon recursion, the regex engine takes a fresh start in attempting the whole regex. All quantifiers and alternatives behave as if the matching process prior to the recursion had never happened at all, other than that the engine advanced through the string. The regex engine restores the states of all quantifiers and alternatives when it exits from a recursion, whether the recursion matched or failed. Basically, the matching process continues normally as if the recursion never happened, other than that the engine advanced through the string.

If you're familiar with procedural programming languages, regex recursion is basically a recursive function call and the quantifiers are local variables in the function. Each recursion of the function gets its own set of local variables that don't affect and aren't affected by the same local variables in recursions higher up the stack.

Let's see how `a(?R){3}z|q` behaves. The simplest possible match is `q`, found by the second alternative in the regex.

The simplest match in which the first alternative matches is `aqqqz`. After `a` is matched, the regex engine begins a recursion. `a` fails to match `q`. Still inside the recursion, the engine attempts the second alternative. `q` matches `q`. The engine exits from the recursion with a successful match. The engine now notes that the quantifier `{3}` has successfully repeated once. It needs two more repetitions, so the engine begins another recursion. It again matches `q`. On the third iteration of the quantifier, the third recursion matches `q`. Finally, `z` matches `z` and an overall match is found.

This regex does not match `aqgz` or `aqqqqz`. `aqgz` fails because during the third iteration of the quantifier, the recursion fails to match `z`. `aqqqqz` fails because after `a(?R){3}` has matched `aqqq`, `z` fails to match the fourth `q`.

The regex can match longer strings such as `aaqqqqzzz`. With this string, during the second iteration of the quantifier, the recursion matches `aqqqz`. Since each recursion tracks the quantifier separately, the recursion needs three consecutive recursions of its own to satisfy its own instance of the quantifier. This can lead to arbitrarily long matches such as `aaaqqqqzzzaqqqzqzqaqqaaqqqzqqzzz`.

Boost's issues with quantifiers on recursion also affect quantifiers on parent groups of the recursion token. They also affect quantifiers on subroutine calls and quantifiers on groups that contain a subroutine call to a parent group of the group with the quantifier.

Quantifiers on Other Tokens in The Recursion

Quantifiers on other tokens in the regex behave normally during recursion. They track their iterations separately at each recursion. So `a{2}(?R)z|q` matches `aaqz`, `aaaaqzz`, `aaaaaaqzzz`, and so on. `a` has to match twice during each recursion.

Quantifiers like these that are inside the recursion but do not repeat the recursion itself do work correctly in Boost.

34. Subroutine Calls May or May Not Capture

This tutorial introduced regular expression subroutines with this example that we want to match accurately:

```
Name: John Doe
Born: 17-Jan-1964
Admitted: 30-Jul-2013
Released: 3-Aug-2013
```

You can use this regular expression with syntax borrowed from Ruby. You can remember this by the fact that the `\g` syntax is a Ruby invention.

```
^Name: \ (.*?)\n
Born: \ (? 'date' (? : 3 [01] [12] [0-9] [1-9] )
      - (? : Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec )
      - (? : 19 20 [0-9] [0-9] )\n
Admitted: \ \g'date'\n
Released: \ \g'date'$
```

Or you can use this regular expression with syntax borrowed from Perl. You can remember this by the fact that Perl uses ampersands for subroutine calls in procedural code too.

```
^Name: \ (.*?)\n
Born: \ (? 'date' (? : 3 [01] [12] [0-9] [1-9] )
      - (? : Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec )
      - (? : 19 20 [0-9] [0-9] )\n
Admitted: \ (?&date)\n
Released: \ (?&date)$
```

Finally, you can use this regular expression with syntax borrowed from PCRE. You can remember this by the fact that PCRE 4.0 invented subroutine calls using this syntax.

```
^Name: \ (.*?)\n
Born: \ (?P<date> (? : 3 [01] [12] [0-9] [1-9] )
      - (? : Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec )
      - (? : 19 20 [0-9] [0-9] )\n
Admitted: \ (?P>date)\n
Released: \ (?P>date)$
```

A subroutine call using the Ruby syntax in EditPad makes the capturing group store the text matched during the subroutine call, just like subroutine calls always do in Ruby. A subroutine call using the Perl or PCRE syntax in EditPad does not affect the group that is called, just like subroutine calls never affect the called group in Perl or PCRE.

If you want to extract the dates from the match, the best solution is to add another capturing group for each date. Then you can ignore the text stored by the “date” group and this particular difference between these flavors. Then it doesn’t matter which syntax you use for subroutine calls in EditPad.

```
^Name: \ (.*?)\n
Born: \ (? 'born' (? 'date' (? : 3 [01] [12] [0-9] [1-9] )
      - (? : Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec )
      - (? : 19 20 [0-9] [0-9] )\n
```



```
Admitted:\ (? 'admitted' \g'date' ) \n
Released:\ (? 'released' \g'date' ) $
```

Capturing Groups Inside Recursion or Subroutine Calls

There are further differences between the Ruby syntax versus the Perl or PCRE syntax for subroutine calls in EditPad when your regex makes a subroutine call or recursive call to a capturing group that contains other capturing groups. The same issues also affect recursion of the whole regular expression if it contains any capturing groups. For the remainder of this topic, the term “recursion” applies equally to recursion of the whole regex, recursion into a capturing group, or a subroutine call to a capturing group.

When you use the Perl or PCRE syntax, EditPad backs up and restores capturing groups when entering and exiting recursion, just like PCRE and Perl 5.20 and later do. When the regex engine enters recursion, it internally makes a copy of all capturing groups. This does not affect the capturing groups. Backreferences inside the recursion match text captured prior to the recursion unless and until the group they reference captures something during the recursion. After the recursion, all capturing groups are replaced with the internal copy that was made at the start of the recursion. Text captured during the recursion is discarded. This means you cannot use capturing groups to retrieve parts of the text that were matched during recursion.

When you use the Ruby syntax, EditPad’s behavior is completely different. When the regex engine enters or exits recursion, it makes no changes to the text stored by capturing groups at all. Backreferences match the text stored by the capturing group during the group’s most recent match, irrespective of any recursion that may have happened. After an overall match is found, each capturing group still stores the text of its most recent match, even if that was during a recursion. This means you can use capturing groups to retrieve part of the text matched during the last recursion.

Odd Length Palindromes Using The Perl or PCRE Syntax

Using the Perl syntax, you can use `\b(? 'word' (? 'letter' [a-z]) (? &word) \k'letter' [a-z]) \b` to match palindrome words such as `a`, `dad`, `radar`, `racecar`, and `redivider`. This regex only matches palindrome words that are an odd number of letters long. This covers most palindrome words in English. To extend the regex to also handle palindrome words that are an even number of characters long we have to worry about differences in how Perl and PCRE backtrack after a failed recursion attempt which is discussed later in this tutorial. We gloss over these differences here because they only come into play when the subject string is not a palindrome and no match can be found.

Let’s see how this regex matches `radar`. The word boundary `\b` matches at the start of the string. The regex engine enters the two capturing groups. `[a-z]` matches `r` which is then stored in the capturing group “letter”. Now the regex engine enters the first recursion of the group “word”. At this point, Perl forgets that the “letter” group matched `r`. PCRE does not. But this does not matter. `(? 'letter' [a-z])` matches and captures `a`. The regex enters the second recursion of the group “word”. `(? 'letter' [a-z])` captures `d`. During the next two recursions, the group captures `a` and `r`. The fifth recursion fails because there are no characters left in the string for `[a-z]` to match. The regex engine must backtrack.

Because `(? &word)` failed to match, `(? 'letter' [a-z])` must give up its match. The group reverts to `a`, which was the text the group held at the start of the recursion. Again, this does not matter because the regex engine must now try the second alternative inside the group “word”, which contains no backreferences. The second `[a-z]` matches the final `r` in the string. The engine now exits from a successful recursion. The text

stored by the group “letter” is restored to what it had captured prior to entering the fourth recursion, which is `a`.

After matching `(?&word)` the engine reaches `\k'letter'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more, making the capturing group give up the `a`. The second alternative now matches the `a`. The regex engine exits from the third recursion. The group “letter” is restored to the `d` matched during the second recursion.

The regex engine has again matched `(?&word)`. The backreference fails again because the group stores `d` while the next character in the string is `r`. Backtracking again, the second alternative matches `d` and the group is restored to the `a` matched during the first recursion.

Now, `\k'letter'` matches the second `a` in the string. That’s because the regex engine has arrived back at the first recursion during which the capturing group matched the first `a`. The regex engine exits the first recursion. The capturing group is restored to the `r` which it matched prior to the first recursion.

Finally, the backreference matches the second `r`. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` matches at the end of the string. The end of the regex is reached and `radar` is returned as the overall match. If you query the groups “word” and “letter” after the match you’ll get `radar` and `r`. That’s the text matched by these groups outside of all recursion.

Why This Regex Does Not Work Using The Ruby Syntax

To match palindromes this way using the Ruby syntax, you need to use a special backreference that specifies a recursion level. If you use a normal backreference as in `\b(?:'word'(?'letter'[a-z]))\g'word'\k'letter'|[a-z])\b`, EditPad will not complain. But it will not match palindromes longer than three letters either. Instead this regex matches things like `a`, `dad`, `radaa`, `raceccc`, and `rediviii`.

Let’s see why this regex does not match `radar` using the Ruby syntax. The regex starts out like the one with the Perl syntax, entering the recursions until there are no characters left in the string for `[a-z]` to match.

Because `\g'word'` failed to match, `(?'letter'[a-z])` must give up its match. EditPad reverts it to `a`, which was the text the group most recently matched. The second `[a-z]` matches the final `r` in the string. The engine now exits from a successful recursion. The group “letter” continues to hold its most recent match `a`.

After matching `\g'word'` the engine reaches `\k'letter'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more, reverting the group to the previously matched `d`. The second alternative now matches the `a`. The regex engine exits from the third recursion.

The regex engine has again matched `\g'word'`. The backreference fails again because the group stores `d` while the next character in the string is `r`. Backtracking again, the group reverts to `a` and the second alternative matches `d`.

Now, `\k'letter'` matches the second `a` in the string. The regex engine exits the first recursion which successfully matched `ada`. The capturing group continues to hold `a` which is its most recent match that wasn’t backtracked.

The regex engine is now at the last character in the string. This character is **r**. The backreference fails because the group still holds **a**. The engine can backtrack once more, forcing **(?'letter'[a-z])\g'word'\k'letter'** to give up the **rada** it matched so far. The regex engine is now back at the start of the string. It can still try the second alternative in the group. This matches the first **r** in the string. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. **\b** fails to match after the first **r**. The regex engine has no further permutations to try. The match attempt has failed.

If the subject string is **radaa**, EditPad's engine goes through nearly the same matching process as described above. Only the events described in the last paragraph change. When the regex engine reaches the last character in the string, that character is now **a**. This time, the backreference matches. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. **\b** matches at the end of the string. The end of the regex is reached and **radaa** is returned as the overall match. If you query the groups "word" and "letter" after the match you'll get **radaa** and **a**. Those are the most recent matches of these groups that weren't backtracked.

Basically, this regex matches any word that is an odd number of letters long and in which all the characters to the right of the middle letter are identical to the character just to the left of the middle letter. That's because EditPad only restores capturing groups when they backtrack, but not when it exits from recursion using the Ruby syntax.

The solution, when using the Ruby syntax, is to use a backreference that specifies a recursion level instead of the normal backreference used in the regex on this page.

35. Backreferences That Specify a Recursion Level

Earlier topics in this tutorial explain regular expression recursion and regular expression subroutines. In this topic the word “recursion” refers to recursion of the whole regex, recursion of capturing groups, and subroutine calls to capturing groups. The previous topic also explained that these features handle capturing groups differently in EditPad when using the Ruby syntax than when using the Perl or PCRE syntax.

When you use the Ruby `\g` syntax for recursion, backreferences in EditPad can match the same text as was matched by a capturing group at any recursion level relative to the recursion level that the backreference is evaluated at. You can do this with the same syntax for named backreferences by adding a sign and a number after the name. In most situations you will use `+0` to specify that you want the backreference to reuse the text from the capturing group at the same recursion level. You can specify a positive number to reference the capturing group at a deeper level of recursion. This would be a recursion the regex engine has already exited from. You can specify a negative number to reference the capturing group at a level that is less deep. This would be a recursion that is still in progress.

Odd Length Palindromes The Ruby Way

You can use `\b(?:'word'(?'letter'[a-z])\g'word'\k'letter+0'|[a-z])\b` to match palindrome words such as `a`, `dad`, `radar`, `racecar`, and `redivider`. To keep this example simple, this regex only matches palindrome words that are an odd number of letters long.

Let's see how this regex matches `radar`. The word boundary `\b` matches at the start of the string. The regex engine enters the capturing group “word”. `[a-z]` matches `r` which is then stored in the stack for the capturing group “letter” at recursion level zero. Now the regex engine enters the first recursion of the group “word”. `(?'letter'[a-z])` matches and captures `a` at recursion level one. The regex enters the second recursion of the group “word”. `(?'letter'[a-z])` captures `d` at recursion level two. During the next two recursions, the group captures `a` and `r` at levels three and four. The fifth recursion fails because there are no characters left in the string for `[a-z]` to match. The regex engine must backtrack.

The regex engine must now try the second alternative inside the group “word”. The second `[a-z]` in the regex matches the final `r` in the string. The engine now exits from a successful recursion, going one level back up to the third recursion.

After matching `\g'word'` the engine reaches `\k'letter+0'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more. The second alternative now matches the `a`. The regex engine exits from the third recursion.

The regex engine has again matched `\g'word'` and needs to attempt the backreference again. The backreference specifies `+0` or the present level of recursion, which is 2. At this level, the capturing group matched `d`. The backreference fails because the next character in the string is `r`. Backtracking again, the second alternative matches `d`.

Now, `\k'letter+0'` matches the second `a` in the string. That's because the regex engine has arrived back at the first recursion during which the capturing group matched the first `a`. The regex engine exits the first recursion.

The regex engine is now back outside all recursion. At this level, the capturing group stored `r`. The backreference can now match the final `r` in the string. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` matches at the end of the string. The end of the regex is reached and `radar` is returned as the overall match.

Backreferences to Other Recursion Levels

Backreferences to other recursion levels can be easily understood if we modify our palindrome example. `abcdefedcba` is also a palindrome matched by the previous regular expression. Consider the regular expression `\b(?:'word'(?:'letter'[a-z])\g'word'(?;\k'letter-1'|z)|[a-z])\b`. The backreference now wants a match the text one level less deep on the capturing group's stack. It is alternated with the letter `z` so that something can be matched when the backreference fails to match.

The new regex matches things like `abcdefdcbaz`. After a whole bunch of matching and backtracking, the second `[a-z]` matches `f`. The regex engine exits from a successful fifth recursion. The capturing group “letter” has stored the matches `a`, `b`, `c`, `d`, and `e` at recursion levels zero to four. Other matches by that group were backtracked and thus not retained.

Now the engine evaluates the backreference `\k'letter-1'`. The present level is 4 and the backreference specifies -1. Thus the engine attempts to match `d`, which succeeds. The engine exits from the fourth recursion.

The backreference continues to match `c`, `b`, and `a` until the regex engine has exited the first recursion. Now, outside all recursion, the regex engine again reaches `\k'letter-1'`. The present level is 0 and the backreference specifies -1. Since recursion level -1 never happened, the backreference fails to match. This is not an error but simply a backreference to a non-participating capturing group. But the backreference has an alternative. `z` matches `z` and `\b` matches at the end of the string. `abcdefdcbaz` was matched successfully.

You can take this as far as you like. The regular expression `\b(?:'word'(?:'letter'[a-z])\g'word'(?;\k'letter-2'|z)|[a-z])\b` matches `abcdefcbazz`. `\b(?:'word'(?:'letter'[a-z])\g'word'(?;\k'letter-99'|z)|[a-z])\b` matches `abcdefzzzzzz`.

Going in the opposite direction, `\b(?:'word'(?:'letter'[a-z])\g'word'(?;\k'letter+1'|z)|[a-z])\b` matches `abcdefzedcb`. Again, after a whole bunch of matching and backtracking, the second `[a-z]` matches `f`, the regex engine is back at recursion level 4, and the group “letter” has `a`, `b`, `c`, `d`, and `e` at recursion levels zero to four on its stack.

Now the engine evaluates the backreference `\k'letter+1'`. The present level is 4 and the backreference specifies +1. The capturing group was backtracked at recursion level 5. This means we have a backreference to a non-participating group, which fails to match. The alternative `z` does match. The engine exits from the fourth recursion.

At recursion level 3, the backreference points to recursion level 4. Since the capturing group successfully matched at recursion level 4, it still has that match on its stack, even though the regex engine has already exited from that recursion. Thus `\k'letter+1'` matches `e`. Recursion level 3 is exited successfully.

The backreference continues to match `d` and `c` until the regex engine has exited the first recursion. Now, outside all recursion, the regex engine again reaches `\k'letter+1'`. The present level is 0 and the backreference specifies +1. The capturing group still retains all its previous successful recursion levels. So the

backreference can still match the `b` that the group captured during the first recursion. Now `\b` matches at the end of the string. `abcdefzdcb` was matched successfully.

You can take this as far as you like in this direction too. The regular expression `\b(?:'word'(? 'letter' [a-z])\g'word'(?:\k'letter'+2'[z]| [a-z]))\b` matches `abcdefzzedc`.
`\b(?:'word'(? 'letter' [a-z])\g'word'(?:\k'letter'+99'[z]| [a-z]))\b` matches `abcdefzzzzzz`.

36. Recursion and Subroutine Calls May or May Not Be Atomic

Earlier topics in this tutorial explain regular expression recursion and regular expression subroutines. In this topic the word “recursion” refers to recursion of the whole regex, recursion of capturing groups, and subroutine calls to capturing groups.

EditPad lets you choose whether recursion should be atomic or not. Atomic recursion gives better performance, but may exclude certain matches or find different matches as illustrated above. `(?P>0)` is atomic in EditPad. You can remember this because this syntax for recursion uses an angle bracket just like an atomic group. You can use a number or a name instead of zero for an atomic subroutine call to a numbered or named capturing group. Any other syntax for recursion is not atomic in EditPad.

Consider the regular expressions `aa$a(?:R)a|a` and `aa$a(?:P>0)a|a`. The first uses non-atomic recursion, while the second uses atomic recursion. Let's see how EditPad goes through the matching process of these regexes when `aaa` is the subject string.

The first alternative `aa$a` fails because the anchor cannot be matched between the second and third `a` in the string. Attempting the second alternative at the start of the string, `a` matches `a`. Now the regex engine enters the first recursion.

Inside the recursion, the first alternative matches the second and third `a` in the string. The regex engine exits a successful recursion. But now, the `a` that follows `(?R)` or `(?P>0)` in the regex fails to match because the regex engine has already reached the end of the string. Thus the regex engine must backtrack. Here is where the two regexes behave differently.

For the first regex, EditPad remembers that inside the recursion the regex matched the second alternative and that there are three possible alternatives. EditPad backtracks *into* the recursion. The second alternative inside the recursion is backtracked, reducing the match so far to the first `a` in the string. Now the third alternative is attempted. `a` matches the second `a` in the string. The regex engine again exits successfully from the same recursion. This time, the `a` that follows `(?R)` in the regex matches the third `a` in the string. `aaa` is found as the overall match.

For the second regex, on the other hand, EditPad remembers nothing about the recursion other than that it matched `aa` at the end of the string. EditPad does backtrack *over* the recursion, reducing the match so far to the first `a` in the string. But this leaves the second alternative in the regex without any further permutations to try. Thus the `a` at the start of the second alternative is also backtracked, reducing the match so far to nothing. EditPad continues the match attempt at the start of the string with the third alternative and finds that `a` matches `a` at the start of the string. For the second regex, this is the overall match.

Palindromes of Any Length with Backtracking into Recursion

The topic about recursion and capturing groups explains a regular expression to match palindromes that are an odd number of characters long. The solution seems trivial. `\b(?:'word'(? 'letter'[a-z]))(?:&word)\k'letter'|[a-z]?)\b` does the trick Perl-style. The quantifier `?` makes the `[a-z]` that matches the letter in the middle of the palindrome optional. Ruby-style we can use `\b(?:'word'(? 'letter'[a-z])\g'word'\k'letter'+0'|[a-z]?)\b` which adds the same quantifier to the solution that specifies the recursion level for the backreference. PCRE-style, with atomic subroutine calls,

`\b(?:'word'(? 'letter'[a-z])(?P>word)\k'letter'|[a-z]?)\b` still matches odd-length palindromes, but not even-length palindromes.

Let's see how these regexes match or fail to match `deed`. All 3 regexes start off the same. The group “letter” matches `d`. During three consecutive recursions, the group captures `e`, `e`, and `d`. The fourth recursion fails, because there are no characters left to match. Back in the third recursion, the first alternative is backtracked and the second alternative matches `d` at the end of the string. The engine exits the third recursion with a successful match. Back in the second recursion, the backreference fails because there are no characters left in the string.

Here the behavior diverges. The first 2 regexes backtrack *into* the third recursion and backtrack the quantifier `?` that makes the second alternative optional. In the third recursion, the second alternative gives up the `d` that it matched at the end of the string. The engine exits the third recursion again, this time with a successful zero-length match. Back in the second recursion, the backreference still fails because the group stored `e` for the second recursion but the next character in the string is `d`. Thus the first alternative is backtracked and the second alternative matches the second `e` in the string. The second recursion is exited with success.

In the first recursion, the backreference again fails. The group stored `e` for the first recursion but the next character in the string is `d`. Again, EditPad backtracks into the second recursion to try the permutation where the second alternative finds a zero-length match. Back in the first recursion again, the backreference now matches the second `e` in the string. The engine leaves the first recursion with success. Back in the overall match attempt, the backreference matches the final `d` in the string. The word boundary succeeds and an overall match is found.

The third regex, however, does not backtrack into the third recursion. It does backtrack *over* the third recursion when it backtracks the first alternative in the second recursion. Now, the second alternative in the second recursion matches the second `e` in the string. The second recursion is exited with success.

In the first recursion, the backreference again fails. The group stored `e` for the first recursion but the next character in the string is `d`. Again, PCRE does not backtrack into the second recursion, but immediately fails the first alternative in the first recursion. The second alternative in the first recursion now matches the first `e` in the string. EditPad exits the first recursion with success. Back in the overall match attempt, the backreference fails, because the group captured `d` prior to the recursion, and the next character is the second `e` in the string. Backtracking again, the second alternative in the overall regex match now matches the first `d` in the string. Then the word boundary fails. The regex with the atomic subroutine call did not find any matches.

Palindromes of Any Length with Atomic Recursion

To match palindromes of any length with atomic subroutine calls, we need a regex that matches words of an even number of characters and of an odd number of characters separately. Free-spacing mode makes this regex easier to read:

```
\b(?:'word'
  (? 'oddword' (? 'oddletter' [a-z])(?P>oddword) \k'oddletter' |[a-z])
  | (? 'evenword' (? 'evenletter' [a-z])(?P>evenword)?\k'evenletter')
)\b
```


Basically, this is two copies of the original regex combined with alternation. The first alternative has the groups “word” and “letter” renamed to “oddword” and “oddletter”. The second alternative has the groups “word” and “letter” renamed to “evenword” and “evenletter”. The call `(?P>evenword)` is now made optional with a question mark instead of the alternative `[a-z]`. A new group “word” combines the two groups “oddword” and “evenword” so that the word boundaries still apply to the whole regex.

The first alternative “oddword” in this regex matches a palindrome of odd length like `radar` in exactly the same way as the regex discussed in the topic about recursion and capturing groups does. The second alternative in the new regex is never attempted.

When the string is a palindrome of even length like `deed`, the new regex first tries all permutations of the first alternative. The second alternative “evenword” is attempted only after the first alternative fails to find a match.

The second alternative starts off in the same way as the original regex. The group “evenletter” matches `d`. During three consecutive recursions, the group captures `e`, `e`, and `d`. The fourth recursion fails, because there are no characters left to match. Back in the third recursion, the regex engine notes that recursive call `(?P>evenword)?` is optional. It proceeds to the backreference `\k'evenletter'`. The backreference fails because there are no characters left in the string. Since the recursion has no further alternatives to try, it is backtracked. The group “evenletter” must give up its most recent match and EditPad exits from the failed third recursion.

In the second recursion, the backreference fails because the capturing group matched `e` during that recursion but the next character in the string is `d`. The group gives up another match and EditPad exits from the failed second recursion.

Back in the first recursion, the backreference succeeds. The group matched the first `e` in the string during that recursion and the backreference matches the second. EditPad exits from the successful first recursion.

Back in the overall match attempt, the backreference succeeds again. The group matched the `d` at the start of the string during the overall match attempt, and the backreference matches the final `d`. Exiting the groups “evenword” and “word”, the word boundary matches at the end of the string. `deed` is the overall match.

37. POSIX Character Classes

Don't confuse the POSIX term "character class" with what is normally called a regular expression character class. `[x-z0-9]` is an example of what this tutorial calls a "character class" and what POSIX calls a "bracket expression". `[:digit:]` is a POSIX character class, used inside a bracket expression like `[x-z[:digit:]]`. The POSIX character class names must be written all lowercase.

When used on ASCII strings, these two regular expressions find exactly the same matches: a single character that is either `x`, `y`, `z`, or a digit. When used on strings with non-ASCII characters, the `[:digit:]` class may include digits in other scripts, depending on the locale.

The POSIX standard defines 12 character classes. The table below lists all 12, plus the `[:ascii:]` and `[:word:]` classes that EditPad also supports. The table also shows equivalent character classes that you can use in ASCII and Unicode regular expressions if the POSIX classes are unavailable. The ASCII equivalents correspond exactly what is defined in the POSIX standard. The Unicode equivalents correspond to what EditPad matches. The POSIX standard does not define a Unicode locale. Some classes also have Perl-style shorthand equivalents.

Java does not support POSIX bracket expressions, but does support POSIX character classes using the `\p` operator. The class names are case sensitive. Unlike the POSIX syntax which can only be used inside a bracket expression, Java's `\p` can be used inside and outside bracket expressions.

EditPad supports both the POSIX and Java syntax. It matches only ASCII characters when using the POSIX syntax, and Unicode characters when using the Java syntax.

POSIX	Description	ASCII	Unicode	Shorthand	Java
<code>[:alnum:]</code>	Alphanumeric characters	<code>[a-zA-Z0-9]</code>	<code>[\p{L}\p{Nl}\p{Nd}]</code>		<code>\p{Alnum}</code>
<code>[:alpha:]</code>	Alphabetic characters	<code>[a-zA-Z]</code>	<code>\p{L}\p{Nl}</code>		<code>\p{Alpha}</code>
<code>[:ascii:]</code>	ASCII characters	<code>[\x00-\x7F]</code>	<code>\p{InBasicLatin}</code>		<code>\p{ASCII}</code>
<code>[:blank:]</code>	Space and tab	<code>[\t]</code>	<code>[\p{Zs}\t]</code>	<code>\h</code>	<code>\p{Blank}</code>
<code>[:cntrl:]</code>	Control characters	<code>[\x00-\x1F\x7F]</code>	<code>\p{Cc}</code>		<code>\p{Cntrl}</code>
<code>[:digit:]</code>	Digits	<code>[0-9]</code>	<code>\p{Nd}</code>	<code>\d</code>	<code>\p{Digit}</code>
<code>[:graph:]</code>	Visible characters (anything except spaces and control characters)	<code>[\x21-\x7E]</code>	<code>[\p{Z}\p{C}]</code>		<code>\p{Graph}</code>
<code>[:lower:]</code>	Lowercase letters	<code>[a-z]</code>	<code>\p{Ll}</code>	<code>\l</code>	<code>\p{Lower}</code>
<code>[:print:]</code>	Visible characters and spaces (anything except control characters)	<code>[\x20-\x7E]</code>	<code>\p{C}</code>		<code>\p{Print}</code>
<code>[:punct:]</code>	Punctuation symbols). (and symbols).	<code>[!\"#\$%&'()*+,-./:;<=>?@[\^_`{ }~]</code>	<code>[\p{P}\$+<=>^` ~]</code>		<code>\p{Punct}</code>
<code>[:space:]</code>	All whitespace characters, including line breaks	<code>[\t\r\n\v\f]</code>	<code>[\p{Z}\t\r\n\v\f]</code>	<code>\s</code>	<code>\p{Space}</code>

[:upper :]	Uppercase letters	[A-Z]	\p{Lu}	\u	\p{Upper}
[:word :]	Word characters (letters, numbers and underscores)	[A-Za-z0-9_]	[\p{L}\p{Nl}\p{Nd}\p{Pc}]	\w	\p{IsWord}
[:xdigit :]	Hexadecimal digits	[A-Fa-f0-9]	[A-Fa-f0-9]		\p{XDigit}
POSIX	Description	ASCII	Unicode	Shorthand	Java

38. Zero-Length Regex Matches

We saw that anchors, word boundaries, and lookahead match at a position, rather than matching a character. This means that when a regex only consists of one or more anchors, word boundaries, or lookarounds, it can result in a zero-length match. Depending on the situation, this can be very useful or undesirable.

In email, for example, it is common to prepend a “greater than” symbol and a space to each line of the quoted message. We can easily do this by replacing all matches of the regex `>` with `>` . Though this regex does not match any characters, it does find a position. The replacement string is inserted there, just like we want it.

Using `^\d*$` to test if the user entered a number would give undesirable results. It causes the script to accept an empty string as a valid input. Let’s see why.

There is only one “character” position in an empty string: the void after the string. The first token in the regex is `^`. It matches the position before the void after the string, because it is preceded by the void before the string. The next token is `\d*`. One of the star’s effects is that it makes the `\d`, in this case, optional. The engine tries to match `\d` with the void after the string. That fails. But the star turns the failure of the `\d` into a zero-length success. The engine proceeds with the next regex token, without advancing the position in the string. So the engine arrives at `$`, and the void after the string. These match. At this point, the entire regex has matched the empty string, and the engine reports success.

The solution is to use the regex `^\d+$` with the proper quantifier to require at least one digit to be entered. If you always make sure that your regexes cannot find zero-length matches, other than special cases such as matching the start or end of each line, then you can save yourself the headache you’ll get from reading the remainder of this topic.

Advancing After a Zero-Length Regex Match

If a regex can find zero-length matches at any position in the string, then it will. The regex `\d*` matches zero or more digits. If the subject string does not contain any digits, then this regex finds a zero-length match at every position in the string. It finds 4 matches in the string `abc`, one before each of the three letters, and one at the end of the string.

Things get tricky when a regex can find zero-length matches at any position as well as certain non-zero-length matches. Say we have the regex `\d*|x`, the subject string `x1`, and a regex engine allows zero-length matches. Which and how many matches do we get when iterating over all matches?

The first match attempt begins at the start of the string. `\d` fails to match `x`. But the `*` makes `\d` optional. The first alternative finds a zero-length match at the start of the string.

Now the regex engine is in a tricky situation. We’re asking it to go through the entire string to find all non-overlapping regex matches. The first match ended at the start of the string, where the first match attempt began. The regex engine needs a way to avoid getting stuck in an infinite loop that forever finds the same zero-length match at the start of the string.

EditPad’s solution, which is used by most regex engines, is to start the next match attempt one character after the end of the previous match, if the previous match was zero-length. In this case, the second match attempt

begins at the position between the **x** and the **1** in the string. **\d** matches **1**. The end of the string is reached. The quantifier ***** is satisfied with a single repetition. **1** is returned as the overall match.

EditPad has an extra rule to skip zero-length matches at the position where the previous match ended, so you can never have a zero-length match immediately adjacent to a non-zero-length match. In our example EditPad only finds two matches: the zero-length match at the start of the string, and **1**.

39. Continuing at The End of The Previous Match

The anchor `\G` matches at the position where the previous match ended. During the first match attempt, `\G` matches at the start of the string in the way `\A` does.

Applying `\G\w` to the string `test string` matches `t`. Applying it again matches `e`. The 3rd attempt yields `s` and the 4th attempt matches the second `t` in the string. The fifth attempt fails. During the fifth attempt, the only place in the string where `\G` matches is after the second `t`. But that position is not followed by a word character, so the match fails.

40. Replacement Strings Tutorial

A replacement string, also known as the replacement text, is the text that each regular expression match is replaced with during a search-and-replace. In most applications, the replacement text supports special syntax that allows you to reuse the text matched by the regular expression or parts thereof in the replacement. This tutorial explains this syntax. While replacement strings are fairly simple compared with regular expressions, there is still great variety between the syntax used by various applications and their actual behavior.

In this book, replacement strings are shown as **replace** like you would enter them in the Replace box of an application. Literal text in the replacement is highlighted in yellow. As `$&\$` shows, special tokens are highlighted in blue and escaped characters in gray.

Table of Contents

Literal Characters and Special Characters

The simplest replacement text consists of only literal characters. Certain characters have special meanings in replacement strings and have to be escaped. Escaping rules may get a bit complicated when using replacement strings in software source code.

Non-Printable Characters

Non-printable characters such as control characters and special spacing or line break characters are easier to enter using control character escapes or hexadecimal escapes.

Matched Text

Reinserting the entire regex match into the replacement text allows a search-and-replace to insert text before and after regular expression matches without really replacing anything.

Backreferences

Backreferences to named and numbered capturing groups in the regular expression allow the replacement text to reuse parts of the text matched by the regular expression.

Match Context

EditPad supports special tokens in replacement strings that allow you to insert the subject string or the part of the subject string before or after the regex match. This can be useful when the replacement text syntax is used to collect search matches and their context instead of making replacements in the subject string.

Case Conversion

EditPad can insert the text matched by the regex or by capturing groups converted to uppercase or lowercase.

Conditionals

EditPad can use one replacement or another replacement depending on whether a capturing group participated in the match. This allows you to use different replacements for different matches of the regular expression.

41. Special Characters

The most basic replacement string consists only of literal characters. The replacement `replacement` simply replaces each regex match with the text `replacement`.

Because we want to be able to do more than simply replace each regex match with the exact same text, we need to reserve certain characters for special use. In most replacement text flavors, two characters tend to have special meanings: the backslash `\` and the dollar sign `$`. Whether and how to escape them depends on the application you're using. In some applications, you always need to escape them when you want to use them as literal characters. In other applications, you only need to escape them if they would form a replacement text token with the character that follows.

In EditPad, you can use a backslash to escape the backslash and the dollar, and you can use a dollar to escape the dollar. `\\` replaces with a literal backslash, while `\$` and `$$` replace with a literal dollar sign. You only need to escape them to suppress their special meaning in combination with other characters. In `\!` and `$!`, the backslash and dollar are literal characters because they don't have a special meaning in combination with the exclamation point. You can't and needn't escape the exclamation point or any other character except the backslash and dollar, because they have no special meaning in EditPad replacement strings.

42. Non-Printable Characters

EditPad allows you to use special escape sequences to enter a few common control characters. Use `\t` to replace with a tab character (ASCII 0x09), `\r` for carriage return (0x0D), and `\n` for line feed (0x0A). Remember that Windows text files use `\r\n` to terminate lines, while UNIX text files use `\n`.

EditPad also supports hexadecimal escapes. You can use `\uFFFF` or `\x{FFFF}` to insert a Unicode character. The euro currency sign occupies Unicode code point U+20AC. If you cannot type it on your keyboard, you can insert it into the replacement text with `\x{20AC}`. For the 127 ASCII characters, you can use `\x00` through `\x7F`. If you are working with files using 8-bit code pages in EditPad you can also use `\x80` through `\xFF` to insert characters from those 8-bit code pages.

43. Matched Text

Reinserting the entire regex match into the replacement text allows a search-and-replace to insert text before and after regular expression matches without really replacing anything. It also allows you to replace the regex match with something that contains the regex match. For example, you could replace all matches of the regex `https?://\S+` with `$&` to turn all URLs in a file into HTML anchors that link to those URLs.

`$&` is substituted with the whole regex match in the replacement text. `\&` works too.

The overall regex match is treated as an implied capturing group number zero. You can use the syntax for backreferences in the replacement text to reference group number zero and thus insert the whole regex match in the replacement text.

44. Numbered and Named Backreferences

If your regular expression has named or numbered capturing groups, then you can reinsert the text matched by any of those capturing groups in the replacement text. Your replacement text can reference as many groups as you like, and can even reference the same group more than once. This makes it possible to rearrange the text matched by a regular expression in many different ways. As a simple example, the regex `*(\w+)*` matches a single word between asterisks, storing the word in the first (and only) capturing group. The replacement text `\1` replaces each regex match with the text stored by the capturing group between bold tags. Effectively, this search-and-replace replaces the asterisks with bold tags, leaving the word between the asterisks in place. This technique using backreferences is important to understand. Replacing `*word*` as a whole with `word` is far easier and far more efficient than trying to come up with a way to correctly replace the asterisks separately.

The `\1` syntax for backreferences in the replacement text is borrowed from the syntax for backreferences in the regular expression. EditPad supports `\1` through `\99`. If there are not enough capturing groups in the regex for the double-digit backreference to be valid, then EditPad treats `\10` through `\99` as a single-digit backreference followed by a literal digit.

`$1` through `$99` for single-digit and double-digit backreferences are also supported by EditPad. If there are not enough capturing groups in the regex for a double-digit backreference to be valid, then `$10` through `$99` are treated as a single-digit backreference followed by a literal digit.

Putting curly braces around the digit `${1}` isolates the digit from any literal digits that follow.

If your regular expression has named capturing groups, then you should use named backreferences to them in the replacement text. The regex `(?'name'group)` has one group called “name”. You can reference this group with `${name}` in EditPad.

Backreferences to Non-Existent Capturing Groups

An invalid backreference is a reference to a number greater than the number of capturing groups in the regex or a reference to a name that does not exist in the regex. Such a backreference is a syntax error.

Backreferences to Non-Participating Capturing Groups

A non-participating capturing group is a group that did not participate in the match attempt at all. This is different from a group that matched an empty string. The group in `a(b?)c` always participates in the match. Its contents are optional but the group itself is not optional. The group in `a(b)?c` is optional. It participates when the regex matches `abc`, but not when the regex matches `ac`.

In EditPad, there is no difference between a backreference in the replacement string to a group that matched the empty string or a group that did not participate. Both are replaced with an empty string.

Backreference to The Highest-Numbered Group

In EditPad, `$+` inserts the text matched by the highest-numbered group that actually participated in the match. When `(a)(b)|(c)(d)` matches `ab`, `$+` is substituted with `b`. When the same regex matches `cd`, `$+` inserts `d`. `\+` does the same.

45. Match Context

EditPad supports special tokens in replacement strings that allow you to insert the subject string or the part of the subject string before or after the regex match. This can be useful when the replacement text syntax is used to collect search matches and their context instead of making replacements in the subject string.

In the replacement text, `$`` (dollar backtick) is substituted with the part of the subject string to the left of the regex match. `\`` (backslash backtick) works too.

You can use `$'` (dollar quote) or `\'` (backslash quote) to insert the part of the subject string to the right of the regex match.

In the replacement text, `$_` is substituted with the entire subject string. `_` is just an escaped underscore.

46. Replacement Text Case Conversion

PowerGREP allows you to prefix the matched text token `\0` and the backreferences `\1` through `\99` with a letter that changes the case of the inserted text. U is for uppercase, L for lowercase, I for initial capitals (first letter of each word is uppercase, rest is lowercase), and F for first capital (first letter in the inserted text is uppercase, rest is lowercase). The letter only affects the case of the backreference that it is part of.

When the regex `(?i)(Hello)(World)` matches `HeLLó WóRlD` the replacement text `\U1\L2\I0\F0` becomes `HELLó wóRld Helló WóRld Helló wóRld`.

47. Replacement String Conditionals

Replacement string conditionals allow you to use one replacement when a particular capturing group participated in the match and another replacement when that capturing group did not participate in the match. EditPad supports the syntax invented by Boost as well as the syntax invented by PCRE2.

Boost Replacement String Conditionals

The syntax borrowed from Boost is `(?1matched:unmatched)` where `1` is a number between 1 and 99 referencing a numbered capturing group. `matched` is used as the replacement for matches in which the capturing group participated. `unmatched` is used for matches in which the group did not participate. The colon `:` delimits the two parts. If you want a literal colon in the `matched` part, then you need to escape it with a backslash. If you want a literal closing parenthesis anywhere in the conditional, then you need to escape that with a backslash too.

The parentheses delimit the conditional from the remainder of the replacement string. `start(?1matched:unmatched)finish` replaces with `startmatchedfinish` when the group participates and with `startunmatchedfinish` when it doesn't.

The `matched` and `unmatched` parts can be blank. You can omit the colon if the `unmatched` part is blank. So `(?1matched:)` and `(?1matched)` replace with `matched` when the group participates. They replace the match with nothing when the group does not participate.

You can use the full replacement string syntax in `matched` and `unmatched`. This means you can nest conditionals inside other conditionals. So `(?1one(?2two):(?2two:none))` replaces with `onetwo` when both groups participate, with `one` or `two` when group 1 or 2 participates and the other doesn't, and with `none` when neither group participates. With Boost `?1one(?2two):?2two:none` does exactly the same but omits parentheses that aren't needed.

EditPad treats conditionals that reference non-existing capturing groups as an error. If there are two digits after the question mark but not enough capturing groups for a two-digit conditional to be valid, then only the first digit is used for the conditional and the second digit is a literal. So when there are less than 12 capturing groups in the regex, `(?12matched)` replaces with `2matched` when capturing group 1 participates in the match.

You can avoid the ambiguity between single digit and double digit conditionals by placing curly braces around the number. `(?{1}1:0)` replaces with `1` when group 1 participates and with `0` when it doesn't, even if there are 11 or more capturing groups in the regex. `(?{12}twelve:not twelve)` is always a conditional that references group 12, even if there are fewer than 12 groups in the regex (which would make the conditional invalid).

The syntax with curly braces also allows you to reference named capturing groups by their names. `(?{name}matched:unmatched)` replaces with `matched` when the group "name" participates in the match and with `unmatched` when it doesn't. If the group does not exist, EditPad treats the conditionals as an error.

PCRE2 Replacement String Conditional

The syntax borrowed from PCRE2 is `${1:+matched:unmatched}` where `1` is a number between 1 and 99 referencing a numbered capturing group. If your regex contains named capturing groups then you can reference them in a conditional by their name: `${name:+matched:unmatched}`.

`matched` is used as the replacement for matches in which the capturing group participated. `unmatched` is used for matches in which the group did not participate. `:+` delimits the group number or name from the first part of the conditional. The second colon delimits the two parts. If you want a literal colon in the `matched` part, then you need to escape it with a backslash. If you want a literal closing curly brace anywhere in the conditional, then you need to escape that with a backslash too. Plus signs have no special meaning beyond the `:+` that starts the conditional, so they don't need to be escaped.

You can use the full replacement string syntax in `matched` and `unmatched`. This means you can nest conditionals inside other conditionals. So `${1:+one${2:+two}:${2:+two:none}}` replaces with `onetwo` when both groups participate, with `one` or `two` when group 1 or 2 participates and the other doesn't, and with `none` when neither group participates.

`${1:-unmatched}` and `${name:-unmatched}` are shorthands for `${1:+${1}:unmatched}` and `${name:+${name}:unmatched}`. They insert the text captured by the group if it participated in the match. They insert `unmatched` if the group did not participate. When using this syntax, `:-` delimits the group number or name from the contents of the conditional. The conditional has only one part in which colons and minus signs have no special meaning.

Escaping Question Marks, Colons, Parentheses, and Curly Braces

As explained above, you need to use backslashes to escape colons that you want to use as literals when used in the `matched` part of the conditional. You also need to escape literal closing parentheses (Boost) or curly braces (PCRE2) with backslashes inside conditionals.

You can escape colons, parentheses, curly braces, and even question marks with backslashes to make sure they are interpreted as literals anywhere in the replacement string. But generally there is no need to.

The colon does not have any special meaning in the `unmatched` part or outside conditionals. So you don't need to escape it there. The question mark does not have any special meaning if it is not followed by a digit or a curly brace. So you only need to escape question marks with backslashes if you want to use a literal question mark followed by a literal digit or curly brace as the replacement.

In EditPad, opening parentheses are part of the syntax for conditionals. The first unescaped closing parenthesis that follows it then ends the conditional. All other unescaped opening and closing parentheses are literals.

Part 3

Regular Expression Examples

1. Sample Regular Expressions

Below, you will find many example patterns that you can use for and adapt to your own purposes. Key techniques used in crafting each regex are explained, with links to the corresponding pages in the tutorial where these concepts and techniques are explained in great detail.

Grabbing HTML Tags

`<TAG\b[^>]*>(.*?)</TAG>` matches the opening and closing pair of a specific HTML tag. Anything between the tags is captured into the first backreference. The question mark in the regex makes the star lazy, to make sure it stops before the first closing tag rather than before the last, like a greedy star would do. This regex will not properly match tags nested inside themselves, like in `<TAG>one<TAG>two</TAG>one</TAG>`.

`<([A-Z][A-Z0-9]*)\b[^>]*>(.*?)</\1>` will match the opening and closing pair of any HTML tag. Be sure to turn off case sensitivity. The key in this solution is the use of the backreference `\1` in the regex. Anything between the tags is captured into the second backreference. This solution will also not match tags nested in themselves.

Trimming Whitespace

You can easily trim unnecessary whitespace from the start and the end of a string or the lines in a text file by doing a regex search-and-replace. Search for `^[\t]+` and replace with nothing to delete leading whitespace (spaces and tabs). Search for `[\t]+$` to trim trailing whitespace. Do both by combining the regular expressions into `^[\t]+|[\t]+$`. Instead of `[\t]` which matches a space or a tab, you can expand the character class into `[\t\r\n]` if you also want to strip line breaks. Or you can use the shorthand `\s` instead.

More Detailed Examples

Numeric Ranges. Since regular expressions work with text rather than numbers, matching specific numeric ranges requires a bit of extra care.

Matching a Floating Point Number. Also illustrates the common mistake of making everything in a regular expression optional.

Matching an Email Address. There's a lot of controversy about what is a proper regex to match email addresses. It's a perfect example showing that you need to know exactly what you're trying to match (and what not), and that there's always a trade-off between regex complexity and accuracy.

Matching an IP Address.

Matching Valid Dates. A regular expression that matches 31-12-1999 but not 31-13-1999.

Finding or Verifying Credit Card Numbers. Validate credit card numbers entered on your order form. Find credit card numbers in documents for a security audit.

Matching Complete Lines. Shows how to match complete lines in a text file rather than just the part of the line that satisfies a certain requirement. Also shows how to match lines in which a particular regex does *not* match.

Removing Duplicate Lines or Items. Illustrates simple yet clever use of capturing parentheses or backreferences.

Regex Examples for Processing Source Code. How to match common programming language syntax such as comments, strings, numbers, etc.

Two Words Near Each Other. Shows how to use a regular expression to emulate the “near” operator that some tools have.

Common Pitfalls

Catastrophic Backtracking. If your regular expression seems to take forever, or simply crashes your application, it has likely contracted a case of catastrophic backtracking. The solution is usually to be more specific about what you want to match, so the number of matches the engine has to try doesn't rise exponentially.

Making Everything Optional. If all the parts in your regex are optional, it will match a zero-length string anywhere. Your regex will need to express the facts that different parts are optional depending on which parts are present.

Repeating a Capturing Group vs. Capturing a Repeated Group. Repeating a capturing group will capture only the last iteration of the group. Capture a repeated group if you want to capture all iterations.

Mixing Unicode and 8-bit Character Codes. Using 8-bit character codes like `\x80` with a Unicode engine and subject string may give unexpected results.

2. Matching Numeric Ranges with a Regular Expression

Since regular expressions deal with text rather than with numbers, matching a number in a given range takes a little extra care. You can't just write `[0-255]` to match a number between 0 and 255. Though a valid regex, it matches something entirely different. `[0-255]` is a character class with three elements: the character range 0-2, the character 5 and the character 5 (again). This character class matches a single digit 0, 1, 2 or 5, just like `[0125]`.

Since regular expressions work with text, a regular expression engine treats `0` as a single character, and `255` as three characters. To match all characters from 0 to 255, we'll need a regex that matches between one and three characters.

The regex `[0-9]` matches single-digit numbers 0 to 9. `[1-9][0-9]` matches double-digit numbers 10 to 99. That's the easy part.

Matching the three-digit numbers is a little more complicated, since we need to exclude numbers 256 through 999. `1[0-9][0-9]` takes care of 100 to 199. `2[0-4][0-9]` matches 200 through 249. Finally, `25[0-5]` adds 250 till 255.

As you can see, you need to split up the numeric range in ranges with the same number of digits, and each of those ranges that allow the same variation for each digit. In the 3-digit range in our example, numbers starting with 1 allow all 10 digits for the following two digits, while numbers starting with 2 restrict the digits that are allowed to follow.

Putting this all together using alternation we get: `[0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]`. This matches the numbers we want, with one caveat: regular expression searches usually allow partial matches, so our regex would match `123` in `12345`. There are two solutions to this.

If you're searching for these numbers in a larger document or input string, use word boundaries to require a non-word character (or no character at all) to precede and to follow any valid match. The regex then becomes `\b([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\b`. Since the alternation operator has the lowest precedence of all, the parentheses are required to group the alternatives together. This way the regex engine will try to match the first word boundary, then try all the alternatives, and then try to match the second word boundary after the numbers it matched. Regular expression engines consider all alphanumeric characters, as well as the underscore, as word characters.

If you're using the regular expression to validate input, you'll probably want to check that the entire input consists of a valid number. To do this, replace the word boundaries with anchors to match the start and end of the string: `^([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])$`.

Here are a few more common ranges that you may want to match:

- 000..255: `^(0[01][0-9][0-9]|2[0-4][0-9]|25[0-5])$`
- 0 or 000..255: `^(0|[01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])$`
- 0 or 000..127: `^(0|[0-9]?[0-9]|1[01][0-9]|12[0-7])$`
- 0..999: `^(0|[1-9][0-9]|1[0-9][0-9]|1[0-9][0-9])$`
- 000..999: `^(0[0-9]{3})$`
- 0 or 000..999: `^(0|[0-9]{1,3})$`
- 1..999: `^(1|[1-9][0-9]|1[0-9][0-9])$`

- 001..999: `^(00[1-9]|0[1-9][0-9]|[1-9][0-9][0-9])$`
- 1 or 001..999: `^(0{0,2}[1-9]|0?[1-9][0-9]|[1-9][0-9][0-9])$`
- 0 or 00..59: `^[0-5]?[0-9]$`
- 0 or 000..366: `^([012]?[0-9]?[0-9]|3[0-5][0-9]|36[0-6])$`

3. Matching Floating Point Numbers with a Regular Expression

This example shows how you can avoid a common mistake often made by people inexperienced with regular expressions. As an example, we will try to build a regular expression that can match any floating point number. Our regex should also match integers and floating point numbers where the integer part is not given. We will not try to match numbers with an exponent, such as 1.5e8 (150 million in scientific notation).

At first thought, the following regex seems to do the trick: `[-+]?[0-9]*\.[0-9]*`. This defines a floating point number as an optional sign, followed by an optional series of digits (integer part), followed by an optional dot, followed by another optional series of digits (fraction part).

Spelling out the regex in words makes it obvious: everything in this regular expression is optional. This regular expression considers a sign by itself or a dot by itself as a valid floating point number. In fact, it even considers an empty string as a valid floating point number. If you tried to use this regex to find floating point numbers in a file, you'd get a zero-length match at every position in the string where no floating point number occurs.

Not escaping the dot is also a common mistake. A dot that is not escaped matches any character, including a dot. If we had not escaped the dot, both `4.4` and `4X4` would be considered floating point numbers.

When creating a regular expression, it is more important to consider what it should *not* match, than what it should. The above regex indeed matches a proper floating point number, because the regex engine is greedy. But it also matches many things we do not want, which we have to exclude.

Here is a better attempt: `[-+]?([0-9]*\.[0-9]+|[0-9]+)`. This regular expression matches an optional sign, that is either followed by zero or more digits followed by a dot and one or more digits (a floating point number with optional integer part), or that is followed by one or more digits (an integer).

This is a far better definition. Any match must include at least one digit. There is no way around the `[0-9]+` part. We have successfully excluded the matches we do not want: those without digits.

We can optimize this regular expression as: `[-+]?[0-9]*\.[0-9]+`.

If you also want to match numbers with exponents, you can use: `[-+]?[0-9]*\.[0-9]+([eE][-+]?[0-9]+)?`. Notice how I made the entire exponent part optional by grouping it together, rather than making each element in the exponent optional.

Finally, if you want to validate if a particular string holds a floating point number, rather than finding a floating point number within longer text, you'll have to anchor your regex: `^[-+]?[0-9]*\.[0-9]+$` or `^[-+]?[0-9]*\.[0-9]+([eE][-+]?[0-9]+)?$`. You can find additional variations of these regexes in RegxBuddy's library.

4. How to Find or Validate an Email Address

The regular expression I receive the most feedback, not to mention “bug” reports on, is the one you’ll find right in the tutorial’s introduction: `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b`. This regular expression, I claim, matches any email address. Most of the feedback I get refutes that claim by showing one email address that this regex doesn’t match. Usually, the “bug” report also includes a suggestion to make the regex “perfect”.

As I explain below, my claim only holds true when one accepts my definition of what a valid email address really is, and what it’s not. If you want to use a different definition, you’ll have to adapt the regex. Matching a valid email address is a perfect example showing that (1) before writing a regex, you have to know exactly what you’re trying to match, and what not; and (2) there’s often a trade-off between what’s exact, and what’s practical.

The virtue of my regular expression above is that it matches 99% of the email addresses in use today. All the email addresses it matches can be handled by 99% of all email software out there. If you’re looking for a quick solution, you only need to read the next paragraph. If you want to know all the trade-offs and get plenty of alternatives to choose from, read on.

If you want to use the regular expression above, there are two things you need to understand. First, long regexes make it difficult to nicely format paragraphs. So I didn’t include `a-z` in any of the three character classes. This regex is intended to be used with your regex engine’s “case insensitive” option turned on. (You’d be surprised how many “bug” reports I get about that.) Second, the above regex is delimited with word boundaries, which makes it suitable for extracting email addresses from files or larger blocks of text. If you want to check whether the user typed in a valid email address, replace the word boundaries with start-of-string and end-of-string anchors, like this: `^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$`.

The previous paragraph also applies to all of the following examples. You may need to change word boundaries into start/end-of-string anchors, or vice versa. And you have to turn on the case insensitive matching option.

Trade-Offs in Validating Email Addresses

Before ICANN made it possible for any well-funded company to create their own top-level domains, the longest top-level domains were the rarely used `.museum` and `.travel` which are 6 letters long. The most common top-level domains were 2 letters long for country-specific domains, and 3 or 4 letters long for general-purpose domains like `.com` and `.info`. A lot of regexes for validating email addresses you’ll find in various regex tutorials and references still assume the top-level domain to be fairly short. Older editions of this regex tutorial mentioned `\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b` as the regex for email addresses in its introduction. There’s only one little difference between this regex and the one at the top of this page. The `4` at the end of the regex restricts the top-level domain to 4 characters. If you use this regex with anchors to validate the email address entered on your order form, `fabio@disapproved.solutions` has to do his shopping elsewhere. Yes, the `.solutions` TLD exists and when I write this, `disapproved.solutions` can be yours for \$16.88 per year.

If you want to be more strict than `[A-Z]{2,}` for the top-level domain, `^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,63}$` is as far as you can practically go. Each part of a domain name can be no longer than 63 characters. There are no single-digit top-level domains and none contain digits. It doesn’t look like ICANN will approve such domains either.

Email addresses can be on servers on a subdomain as in `john@server.department.company.com`. All of the above regexes match this email address, because I included a dot in the character class after the `@` symbol. But the above regexes also match `john@aol...com` which is not valid due to the consecutive dots. You can exclude such matches by replacing `[A-Z0-9.-]+\.` with `(?:[A-Z0-9-]+\.)+` in any of the above regexes. I removed the dot from the character class and instead repeated the character class and the following literal dot. E.g. `^[A-Z0-9._%+-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,}$` matches `john@server.department.company.com` but not `john@aol...com`.

If you want to avoid your system choking on arbitrarily large input, you can replace the infinite quantifiers with finite ones. `^[A-Z0-9._%+-]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,125}[A-Z]{2,63}$` takes into account that the local part (before the `@`) is limited to 64 characters and that each part of the domain name is limited to 63 characters. There's no direct limit on the number of subdomains. But the maximum length of an email address that can be handled by SMTP is 254 characters. So with a single-character local part, a two-letter top-level domain and single-character sub-domains, 125 is the maximum number of sub-domains.

The previous regex does not actually limit email addresses to 254 characters. If each part is at its maximum length, the regex can match strings up to 8129 characters in length. You can reduce that by lowering the number of allowed sub-domains from 125 to something more realistic like 8. I've never seen an email address with more than 4 subdomains. If you want to enforce the 254 character limit, the best solution is to check the length of the input string before you even use a regex. Though this requires a few lines of procedural code, checking the length of a string is near-instantaneous. If you can only use regexes, `^[A-Z0-9@._%+-]{6,254}$` can be used as a first pass to make sure the string doesn't contain invalid characters and isn't too short or too long. If you need to do everything with one regex, you'll need a regex flavor that supports lookahead. The regular expression `^(?=[A-Z0-9@._%+-]{6,254}$)[A-Z0-9._%+-]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,8}[A-Z]{2,63}$` uses a lookahead to first check that the string doesn't contain invalid characters and isn't too short or too long. When the lookahead succeeds, the remainder of the regex makes a second pass over the string to check for proper placement of the `@` sign and the dots.

All of these regexes allow the characters `._%+-` anywhere in the local part. You can force the local part to begin with a letter by using `^[A-Z0-9][A-Z0-9._%+-]{0,63}` instead of `^[A-Z0-9._%+-]{1,64}` for the local part: `^[A-Z0-9][A-Z0-9._%+-]{0,63}@(?:[A-Z0-9-]{1,63}\.){1,125}[A-Z]{2,63}$`. When using lookahead to check the overall length of the address, the first character can be checked in the lookahead. We don't need to repeat the initial character check when checking the length of the local part. This regex is too long to fit the width of the page, so let's turn on free-spacing mode:

```
^(?=[A-Z0-9][A-Z0-9@._%+-]{5,253}$)
[A-Z0-9._%+-]{1,64}@(?:[A-Z0-9-]{1,63}\.){1,8}[A-Z]{2,63}$
```

Domain names can contain hyphens. But they cannot begin or end with a hyphen. `[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?` matches a domain name between 1 and 63 characters long that starts and ends with a letter or digit. The non-capturing group makes the middle of the domain and the final letter or digit optional as a whole to ensure that we allow single-character domains while at the same time ensuring that domains with two or more characters do not end with a hyphen. The overall regex starts to get quite complicated:

```
^[A-Z0-9][A-Z0-9._%+-]{0,63}@
(?:[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?\.){1,8}[A-Z]{2,63}$
```

Domain names cannot contain consecutive hyphens. `[A-Z0-9]+(?:-[A-Z0-9]+)*` matches a domain name that starts and ends with a letter or digit and that contains any number of non-consecutive hyphens. This is the most efficient way. This regex does not do any backtracking to match a valid domain name. It matches all letters and digits at the start of the domain name. If there are no hyphens, the optional group that

follows fails immediately. If there are hyphens, the group matches each hyphen followed by all letters and digits up to the next hyphen or the end of the domain name. We can't enforce the maximum length when hyphens must be paired with a letter or digit, but letters and digits can stand on their own. But we can use the lookahead technique that we used to enforce the overall length of the email address to enforce the length of the domain name while disallowing consecutive hyphens: `(?=[A-Z0-9-]{1,63}\.)([A-Z0-9-]+(?:-[A-Z0-9-]+)*)\.` Notice that the lookahead also checks for the dot that must appear after the domain name when it is fully qualified in an email address. This is important. Without checking for the dot, the lookahead would accept longer domain names. Since the lookahead does not consume the text it matches, the dot is not included in the overall match of this regex. When we put this regex into the overall regex for email addresses, the dot will be matched as it was in the previous regexes:

```
^[A-Z0-9]([A-Z0-9-]{0,63})@
(?: (?:[A-Z0-9-]{1,63}\.)([A-Z0-9-]+(?:-[A-Z0-9-]+)*)\.) {1,8} [A-Z]{2,63}$
```

If we include the lookahead to check the overall length, our regex makes two passes over the local part, and three passes over the domain names to validate everything:

```
^(?=[A-Z0-9]([A-Z0-9-]{0,63})@. {5,253}$)([A-Z0-9-]{1,64})@
(?: (?:[A-Z0-9-]{1,63}\.)([A-Z0-9-]+(?:-[A-Z0-9-]+)*)\.) {1,8} [A-Z]{2,63}$
```

On a modern PC or server this regex will perform just fine when validating a single 254-character email address. Rejecting longer input would even be faster because the regex will fail when the lookahead fails during first pass. But I wouldn't recommend using a regex as complex as this to search for email addresses through a large archive of documents or correspondence. You're better off using the simple regex at the top of this page to quickly gather everything that looks like an email address. Deduplicate the results and then use a stricter regex if you want to further filter out invalid addresses.

And speaking of backtracking, none of the regexes on this page do any backtracking to match valid email addresses. But particularly the latter ones may do a fair bit of backtracking on something that's not quite a valid email address. If your regex flavor supports possessive quantifiers, you can eliminate all backtracking by making all quantifiers possessive. Because no backtracking is needed to find matches, doing this does not change what is matched by these regexes. It only allows them to fail faster when the input is not a valid email address. The simplest regex that correctly handles subdomains then becomes `^[A-Z0-9-]{1,64}++@(?:[A-Z0-9-]{1,63}++\.)([A-Z0-9-]{1,64}++(?:-[A-Z0-9-]{1,63}++)*\.) {1,8} [A-Z]{2,63}++$` with an extra `+` after each quantifier. We can do the same with our most complex regex:

```
^(?=[A-Z0-9]([A-Z0-9-]{0,63})@. {5,253}++$)([A-Z0-9-]{1,64}++)@
(?: (?:[A-Z0-9-]{1,63}++\.)([A-Z0-9-]{1,64}++(?:-[A-Z0-9-]{1,63}++)*\.) {1,8} [A-Z]{2,63}++$
```

An important trade-off in all these regexes is that they only allow English letters, digits, and the most commonly used special symbols. The main reason is that I don't trust all my email software to be able to handle much else. Even though `John.O'Haras@theoharas.com` is a syntactically valid email address, there's a risk that some software will misinterpret the apostrophe as a delimiting quote. Blindly inserting this email address into an SQL query, for example, will at best cause it to fail when strings are delimited with single quotes and at worst open your site up to SQL injection attacks.

And of course, it's been many years already that domain names can include non-English characters. But most software still sticks to the 37 characters Western programmers are used to. Supporting internationalized domains opens up a whole can of worms of how the non-ASCII characters should be encoded. So if you use any of the regexes on this page, anyone with an `@ทีโอช.นิค.ไทย` address will be out of luck. But perhaps it is

telling that `http://ที่เอสเน็ต.ไทย` simply redirects to `http://thnic.co.th` even though they're in the business of selling `.ไทย` domains.

The conclusion is that to decide which regular expression to use, whether you're trying to match an email address or something else that's vaguely defined, you need to start with considering all the trade-offs. How bad is it to match something that's not valid? How bad is it not to match something that is valid? How complex can your regular expression be? How expensive would it be if you had to change the regular expression later because it turned out to be too broad or too narrow? Different answers to these questions will require a different regular expression as the solution. My email regex does what I want, but it may not do what you want.

Regexes Don't Send Email

Don't go overboard in trying to eliminate invalid email addresses with your regular expression. The reason is that you don't really know whether an address is valid until you try to send an email to it. And even that might not be enough. Even if the email arrives in a mailbox, that doesn't mean somebody still reads that mailbox. If you really need to be sure an email address is valid, you'll need to send an email to it that contains a code or link for the recipient to perform a second authentication step. And if you're doing that, then there is little point in using a regex that may reject valid email addresses.

The same principle applies in many situations. When trying to match a valid date, it's often easier to use a bit of arithmetic to check for leap years, rather than trying to do it in a regex. Use a regular expression to find potential matches or check if the input uses the proper syntax, and do the actual validation on the potential matches returned by the regular expression. Regular expressions are a powerful tool, but they're far from a panacea.

The Official Standard: RFC 5322

Maybe you're wondering why there's no "official" fool-proof regex to match email addresses. Well, there is an official definition, but it's hardly fool-proof.

The official standard is known as RFC 5322. It describes the syntax that valid email addresses must adhere to. You can (but you shouldn't—read on) implement it with the following regular expression. RFC 5322 leaves the domain name part open to implementation-specific choices that won't work on the Internet today. The regex implements the "preferred" syntax from RFC 1035 which is one of the recommendations in RFC 5322:

```
\A(?:[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*/+=?^_`{|}~-]+)*)
|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]
\\[\x01-\x09\x0b\x0c\x0e-\x7f])*"
@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\.[a-z0-9](?:[a-z0-9-]*[a-z0-9])?
|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5a\x53-\x7f]
\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")+)z
```

This regex has two parts: the part before the `@`, and the part after the `@`. There are two alternatives for the part before the `@`. The first alternative allows it to consist of a series of letters, digits and certain symbols,

including one or more dots. However, dots may not appear consecutively or at the start or end of the email address. The other alternative requires the part before the @ to be enclosed in double quotes, allowing any string of ASCII characters between the quotes. Whitespace characters, double quotes and backslashes must be escaped with backslashes.

The part after the @ also has two alternatives. It can either be a fully qualified domain name (e.g. regular-expressions.info), or it can be a literal Internet address between square brackets. The literal Internet address can either be an IP address, or a domain-specific routing address.

The reason you shouldn't use this regex is that it is overly broad. Your application may not be able to handle all email addresses this regex allows. Domain-specific routing addresses can contain non-printable ASCII control characters, which can cause trouble if your application needs to display addresses. Not all applications support the syntax for the local part using double quotes or square brackets. In fact, RFC 5322 itself marks the notation using square brackets as obsolete.

We get a more practical implementation of RFC 5322 if we omit IP addresses, domain-specific addresses, the syntax using double quotes and square brackets. It will still match 99.99% of all email addresses in actual use today.

```
\A[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*/+=?^_`{|}~-]+)*@
(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.[a-z0-9](?:[a-z0-9-]*[a-z0-9])?)?\z
```

Neither of these regexes enforce length limits on the overall email address or the local part or the domain names. RFC 5322 does not specify any length limitations. Those stem from limitations in other protocols like the SMTP protocol for actually sending email. RFC 1035 does state that domains must be 63 characters or less, but does not include that in its syntax specification. The reason is that a true regular language cannot enforce a length limit and disallow consecutive hyphens at the same time. But modern regex flavors aren't truly regular, so we can add length limit checks using lookahead like we did before:

```
\A(?:[a-z0-9@. !#$%&'*/+=?^_`{|}~-]{6,254}\z)
(?:[a-z0-9 !#$%&'*/+=?^_`{|}~-]{1,64}@)
[a-z0-9 !#$%&'*/+=?^_`{|}~-]+(?:\.[a-z0-9 !#$%&'*/+=?^_`{|}~-]+)*
@ (?:[a-z0-9-]{1,63}\.[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\z
(?:[a-z0-9-]{1,63}\z)[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\z
```

So even when following official standards, there are still trade-offs to be made. Don't blindly copy regular expressions from online libraries or discussion forums. Always test them on your own data and with your own applications.

5. How to Find or Validate an IP Address

Matching an IP address is another good example of a trade-off between regex complexity and exactness. `\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b` will match any IP address just fine. But will also match `999.999.999.999` as if it were a valid IP address. If your regex flavor supports Unicode, it may even match `١٢٣.٩٢٣.٠٩٩.٠٩٩`. Whether this is a problem depends on the files or data you intend to apply the regex to.

Restricting and Capturing The Four IP Address Numbers

To restrict all 4 numbers in the IP address to 0..255, you can use the following regex. It stores each of the 4 numbers of the IP address into a capturing group. You can use these groups to further process the IP number. Free-spacing mode allows this to fit the width of the page.

```
\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

The above regex allows one leading zero for numbers 10 to 99 and up to two leading zeros for numbers 0 to 9. Strictly speaking, IP addresses with leading zeros imply octal notation. So you may want to disallow leading zeros. This requires a slightly longer regex:

```
\b(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|1[0-9]?[0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|1[0-9]?[0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|1[0-9]?[0-9])\.(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|1[0-9]?[0-9])\b
```

Restricting The Four IP Address Numbers Without Capturing Them

If you don't need access to the individual numbers, you can shorten above 3 regexes with a quantifier to:

```
\b(?:\d{1,3}\.){3}\d{1,3}\b
```

```
\b(?: (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. ){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

```
\b(?: (?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|1[0-9]?[0-9])\. ){3}(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|1[0-9]?[0-9])\b
```

Checking User Input

The above regexes use word boundaries to make sure the first and last number in the IP address aren't part of a longer sequence of alphanumeric characters. These regexes are appropriate for finding IP addresses in longer strings.

If you want to validate user input by making sure a string consists of nothing but an IP address then you need to replace the word boundaries with start-of-string and end-of-string anchors. You can use the dedicated anchors `\A` and `\Z` if your regex flavor supports them:

```
\A(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\Z
```

If not, you'll have to use `^` and `$` and make sure that the option for them to match at line breaks is off:

```
^(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```


6. Matching a Valid Date

`^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|1[12][0-9]|3[01])$` matches a date in yyyy-mm-dd format from 1900-01-01 through 2099-12-31, with a choice of four separators. The anchors make sure the entire variable is a date, and not a piece of text containing a date. The year is matched by `(19|20)\d\d`. I used alternation to allow the first two digits to be 19 or 20. The parentheses are mandatory. Had I omitted them, the regex engine would go looking for 19 or the remainder of the regular expression, which matches a date between 2000-01-01 and 2099-12-31. Parentheses are the only way to stop the vertical bar from splitting up the entire regular expression into two options.

The month is matched by `0[1-9]|1[012]`, again enclosed by parentheses to keep the two options together. By using character classes, the first option matches a number between 01 and 09, and the second matches 10, 11 or 12.

The last part of the regex consists of three options. The first matches the numbers 01 through 09, the second 10 through 29, and the third matches 30 or 31.

Smart use of alternation allows us to exclude invalid dates such as 2000-00-00 that could not have been excluded without using alternation. To be really perfectionist, you would have to split up the month into various options to take into account the length of the month. The above regex still matches 2003-02-31, which is not a valid date. Making leading zeros optional could be another enhancement.

If you want to require the delimiters to be consistent, you could use a backreference. `^(19|20)\d\d([- /.])(0[1-9]|1[012])\2(0[1-9]|1[12][0-9]|3[01])$` will match `1999-01-01` but not `1999/01-01`.

Again, how complex you want to make your regular expression depends on the data you are using it on, and how big a problem it is if an unwanted match slips through. If you are validating the user's input of a date in a script, it is probably easier to do certain checks outside of the regex. For example, excluding February 29th when the year is not a leap year is far easier to do in a scripting language. It is far easier to check if a year is divisible by 4 (and not divisible by 100 unless divisible by 400) using simple arithmetic than using regular expressions.

To match a date in mm/dd/yyyy format, rearrange the regular expression to `^(0[1-9]|1[012])[- /.](0[1-9]|1[12][0-9]|3[01])[- /.](19|20)\d\d$`. For dd-mm-yyyy format, use `^(0[1-9]|1[12][0-9]|3[01])[- /.](0[1-9]|1[012])[- /.](19|20)\d\d$`. You can find additional variations of these regexes in RegxBuddy's library.

7. Replacing Numerical Dates with Textual Dates

This example shows how you can replace numerical dates from 1/1/50 or 01/01/50 through 12/31/49 with their textual equivalents from January 1st, 1950 through December 31st, 2049. This is only possible with a single regular expression if you can vary the replacement based on what was matched. One way to do this is to build each replacement in procedural code. This example shows how you can do it using replacement string conditionals.

To be able to use replacement string conditionals, the regular expression needs a separate capturing group for each part of the match that needs a different replacement. Each month needs to be replaced with its own name, so we need a separate capturing group to match each month number. Cardinal numbers ending with 1, 2, and 3 have unique suffixes. So we need four groups to match day numbers ending with 1, 2, 3, or another digit. We're assuming year numbers 50 to 99 to be 1950 to 1999, and year numbers 00 to 49 to be 2000 to 2049. So we need two more groups to match each half century.

Putting this all together results in a rather long regular expression. Free-spacing helps to keep it readable. The structure of the regex is the same as what you would use for matching valid dates. It's only more long-winded because we need 12 alternatives to match the month, 4 alternatives to match the day, and 2 alternatives to match the year.

```
\b
(?: # Month
  ('jan'|'0?1')|('feb'|'0?2')|('mar'|'0?3')|('apr'|'0?4')|('may'|'0?5')|('jun'|'0?6')
  |('jul'|'0?7')|('aug'|'0?8')|('sep'|'0?9')|('oct'|'10')|('nov'|'11')|('dec'|'12')
) /
0?(?: # Day
  ('1st'|'23?1')|('2nd'|'2?2')|('3rd'|'2?3')|('nth'|'30'1[123]|'12'?[4-90])
) /
(?: # Year
  ('19xx'|'5-9' '[0-9]')|('20xx'|'0-4' '[0-9]')
)
\b
```

The replacement string will use backreferences to reinsert the date numbers. Since we want to omit leading zeros from the replacements, we placed `0?` outside the capturing groups for date numbers. This means that our regex also allows leading zeros for days 10 to 31. Since our goal is to replace dates rather than validate them, we can live with this. Otherwise, we would need two sets of four alternatives to match the day of the month. One set for single digit days, and one set for double digit days.

Unfortunately, free-spacing does not work with replacement strings. So the replacement consists of one very long line. It is broken into multiple lines here to fit the width of the page. This is the replacement using Boost syntax:

```
(?{jan}January)(?{feb}February)(?{mar}March)(?{apr}April)(?{may}May)(?{jun}June)
(?{jul}July)(?{aug}August)(?{sep}September)(?{oct}October)(?{nov}November)
(?{dec}December) (?{1st}${1st}st)(?{2nd}${2nd}nd)(?{3rd}${3rd}rd)(?{nth}${nth}th)
, (?{19xx}19${19xx})(?{20xx}20${20xx})
```

This is the replacement using PCRE2 syntax:

```
${jan:+January}${feb:+February}${mar:+March}${apr:+April}${may:+May}${jun:+June}
${jul:+July}${aug:+August}${sep:+September}${oct:+October}${nov:+November}
```

```

${dec:+December} ${1st:+${1st}st} ${2nd:+${2nd}nd} ${3rd:+${3rd}rd} ${nth:+${nth}th}
, ${19xx:+19${19xx}} ${20xx:+20${20xx}}

```

First we have 12 conditionals that reference the 12 capturing groups for the months. Each conditional inserts the month's name when their group participates. They insert nothing when their group does not participate. Since only one of these groups participates in any match, only one of these conditionals actually inserts anything into the replacement.

Then we have a literal space and 4 more conditionals that reference the 4 capturing groups for the days. When the group participates, the conditional uses a backreference to the same group to reinsert the day number matched by the group. The backreference is followed by a literal suffix.

Finally, we have a literal comma, a literal space, and 2 more conditionals for the year. The conditionals again use literal text and a backreference to expand the year from 2 to 4 digits.

8. Finding or Verifying Credit Card Numbers

With a few simple regular expressions, you can easily verify whether your customer entered a valid credit card number on your order form. You can even determine the type of credit card being used. Each card issuer has its own range of card numbers, identified by the first 4 digits.

You can use a slightly different regular expression to find credit card numbers, or number sequences that might be credit card numbers, within larger documents. This can be very useful to prove in a security audit that you're not improperly exposing your clients' financial details.

We'll start with the order form.

Stripping Spaces and Dashes

The first step is to remove all non-digits from the card number entered by the customer. Physical credit cards have spaces within the card number to group the digits, making it easier for humans to read or type in. So your order form should accept card numbers with spaces or dashes in them.

To remove all non-digits from the card number, simply use the “replace all” function in your scripting language to search for the regex `[\^0-9]+` and replace it with nothing. If you only want to replace spaces and dashes, you could use `[-]+`. If this regex looks odd, remember that in a character class, the hyphen is a literal when it occurs right before the closing bracket (or right after the opening bracket or negating caret).

If you're wondering what the plus is for: that's for performance. If the input has consecutive non-digits, such as `1===2`, then `[\^0-9]+` matches the three equals signs at once and deletes them in one replacement. Without the plus, three replacements would be required. In this case, the savings are only a few microseconds. But it's a good habit to keep regex efficiency in the back of your mind. Though the savings are minimal here, so is the effort of typing the extra plus.

Validating Credit Card Numbers on Your Order Form

Validating credit card numbers is the ideal job for regular expressions. They're just a sequence of 13 to 16 digits, with a few specific digits at the start that identify the card issuer. You can use the specific regular expressions below to alert customers when they try to use a kind of card you don't accept, or to route orders using different cards to different processors. All these regexes were taken from RegxBuddy's library.

- Visa: `^4[0-9]{12}(?:[0-9]{3})?.$` All Visa card numbers start with a 4. New cards have 16 digits. Old cards have 13.
- MasterCard: `^(?:5[1-5][0-9]{2}||222[1-9]||22[3-9][0-9]||2[3-6][0-9]{2}||27[01][0-9]||2720)[0-9]{12}.$` MasterCard numbers either start with the numbers 51 through 55 or with the numbers 2221 through 2720. All have 16 digits.
- American Express: `^3[47][0-9]{13}.$` American Express card numbers start with 34 or 37 and have 15 digits.
- Diners Club: `^3(?:0[0-5]||[68][0-9])[0-9]{11}.$` Diners Club card numbers begin with 300 through 305, 36 or 38. All have 14 digits. There are Diners Club cards that begin with 5 and have 16 digits. These are a joint venture between Diners Club and MasterCard, and should be processed like a MasterCard.

- Discover: `^6(?:011|5[0-9]){2}[0-9]{12}$` Discover card numbers begin with 6011 or 65. All have 16 digits.
- JCB: `^(?:2131|1800|35\d{3})\d{11}$` JCB cards beginning with 2131 or 1800 have 15 digits. JCB cards beginning with 35 have 16 digits.

If you just want to check whether the card number looks valid, without determining the brand, you can combine the above six regexes using alternation. A non-capturing group puts the anchors outside the alternation. Free-spacing allows for comments and for the regex to fit the width of this page.

```
^(?:4[0-9]{12}(?:[0-9]{3})?|
| (?:5[1-5][0-9]{2}
| 222[1-9]|22[3-9][0-9]|2[3-6][0-9]{2}|27[01][0-9]|2720)[0-9]{12}|
| 3[47][0-9]{13}|
| 3[2]0[0-5][68][0-9][0-9]{11}|
| 6(?:011|5[0-9]{2})[0-9]{12}|
| (?:2131|1800|35\d{3})\d{11})$
# Visa
# MasterCard
# American Express
# Diners Club
# Discover
# JCB
```

These regular expressions will easily catch numbers that are invalid because the customer entered too many or too few digits. They won't catch numbers with incorrect digits. For that, you need to follow the Luhn algorithm, which cannot be done with a regex. And of course, even if the number is mathematically valid, that doesn't mean a card with this number was issued or that there's money in the account. The benefit or the regular expression is that you can put it in a bit of JavaScript to instantly check for obvious errors, instead of making the customer wait 30 seconds for your credit card processor to fail the order. And if your card processor charges for failed transactions, you'll really want to implement both the regex and the Luhn validation.

Finding Credit Card Numbers in Documents

With two simple modifications, you could use any of the above regexes to find card numbers in larger documents. Simply replace the caret and dollar with a word boundary as in `\b4[0-9]{12}(?:[0-9]{3})?\b`.

If you're planning to search a large document server, a simpler regular expression will speed up the search. Unless your company uses 16-digit numbers for other purposes, you'll have few false positives. The regex `\b\d{13,16}\b` will find any sequence of 13 to 16 digits.

When searching a hard disk full of files, you can't strip out spaces and dashes first like you can when validating a single card number. To find card numbers with spaces or dashes in them, use `\b(?:\d[-]*?){13,16}\b`. This regex allows any amount of spaces and dashes anywhere in the number. This is really the only way. Visa and MasterCard put digits in sets of 4, while Amex and Discover use groups of 4, 5 and 6 digits. People entering the numbers may have different ideas yet.

9. Matching Whole Lines of Text

Often, you want to match complete lines in a text file rather than just the part of the line that satisfies a certain requirement. This is useful if you want to delete entire lines in a search-and-replace in a text editor, or collect entire lines in an information retrieval tool.

To keep this example simple, let's say we want to match lines containing the word "John". The regex `John` makes it easy enough to locate those lines. But the software will only indicate `John` as the match, not the entire line containing the word.

The solution is fairly simple. To specify that we need an entire line, we will use the caret and dollar sign and turn on the option to make them match at embedded newlines. In software aimed at working with text files like EditPad Pro and PowerGREP, the anchors always match at embedded newlines. To match the parts of the line before and after the match of our original regular expression `John`, we simply use the dot and the star. Be sure to turn *off* the option for the dot to match newlines.

The resulting regex is: `^.*John.*$`. You can use the same method to expand the match of any regular expression to an entire line, or a block of complete lines. In some cases, such as when using alternation, you will need to group the original regex together using parentheses.

Finding Lines Containing or Not Containing Certain Words

If a line can meet any out of series of requirements, simply use alternation in the regular expression. `^.*\b(one|two|three)\b.*$` matches a complete line of text that contains any of the words "one", "two" or "three". The first backreference will contain the word the line actually contains. If it contains more than one of the words, then the last (rightmost) word will be captured into the first backreference. This is because the star is greedy. If we make the first star lazy, like in `^.*?\b(one|two|three)\b.*$`, then the backreference will contain the first (leftmost) word.

If a line must satisfy all of multiple requirements, we need to use lookahead. `^(?=.*?\bone\b)(?=.*?\btwo\b)(?=.*?\bthree\b).*$` matches a complete line of text that contains *all* of the words "one", "two" and "three". Again, the anchors must match at the start and end of a line and the dot must not match line breaks. Because of the caret, and the fact that lookahead is zero-length, all of the three lookaheads are attempted at the start of the each line. Each lookahead will match any piece of text on a single line (`.*?`) followed by one of the words. All three must match successfully for the entire regex to match. Note that instead of words like `\bword\b`, you can put any regular expression, no matter how complex, inside the lookahead. Finally, `.*$` causes the regex to actually match the line, after the lookaheads have determined it meets the requirements.

If your condition is that a line should *not* contain something, use negative lookahead. `^(?!.*regex).*$` matches a complete line that does *not* match `regex`. Notice that unlike before, when using positive lookahead, I repeated both the negative lookahead and the dot together. For the positive lookahead, we only need to find one location where it can match. But the negative lookahead must be tested at each and every character position in the line. We must test that `regex` fails everywhere, not just somewhere.

Finally, you can combine multiple positive and negative requirements as follows: `^(?=.*?\bmust-have\b)(?=.*?\bmandatory\b)(?!.*\bavoid\b).*$`. When checking multiple positive

requirements, the `.*` at the end of the regular expression full of zero-length assertions made sure that we actually matched something. Since the negative requirement must match the entire line, it is easy to replace the `.*` with the negative test.

10. Deleting Duplicate Lines From a File

If you have a file in which all lines are sorted (alphabetically or otherwise), you can easily delete (consecutive) duplicate lines. Simply open the file in your favorite text editor, and do a search-and-replace searching for `^(.*)\r?\n\1]+$` and replacing with `\1`. For this to work, the anchors need to match before and after line breaks (and not just at the start and the end of the file or string), and the dot must *not* match newlines.

Here is how this works. The caret will match only at the start of a line. So the regex engine will only attempt to match the remainder of the regex there. The dot and star combination simply matches an entire line, whatever its contents, if any. The parentheses store the matched line into the first backreference.

Next we will match the line separator. I put the question mark into `\r?\n` to make this regex work with both Windows (`\r\n`) and UNIX (`\n`) text files. So up to this point we matched a line and the following line break.

Now we need to check if this combination is followed by a duplicate of that same line. We do this simply with `\1`. This is the first backreference which holds the line we matched. The backreference will match that very same text.

If the backreference fails to match, the regex match and the backreference are discarded, and the regex engine tries again at the start of the next line. If the backreference succeeds, the plus symbol in the regular expression will try to match additional copies of the line. Finally, the dollar symbol forces the regex engine to check if the text matched by the backreference is a complete line. We already know the text matched by the backreference is preceded by a line break (matched by `\r?\n`). Therefore, we now check if it is also followed by a line break or if it is at the end of the file using the dollar sign.

The entire match becomes `line\nline` (or `line\nline\nline` etc.). Because we are doing a search and replace, the line, its duplicates, and the line breaks in between them, are all deleted from the file. Since we want to keep the original line, but not the duplicates, we use `\1` as the replacement text to put the original line back in.

Removing Duplicate Items From a String

We can generalize the above example to `afterseparator(item)(separator\1)+beforeseparator`, where `afterseparator` and `beforeseparator` are zero-length. So if you want to remove consecutive duplicates from a comma-delimited list, you could use `(?<=,[^,]) ([^,]*) (,\1)+ (?=,|$)`.

The positive lookbehind `(?<=,[^,])` forces the regex engine to start matching at the start of the string or after a comma. `([^,]*)` captures the item. `(,\1)+` matches consecutive duplicate items. Finally, the positive lookahead `(?=,|$)` checks if the duplicate items are complete items by checking for a comma or the end of the string.

11. Example Regexes to Match Common Programming Language Constructs

Regular expressions are very useful to manipulate source code in a text editor or in a regex-based text processing tool. Most programming languages use similar constructs like keywords, comments and strings. But often there are subtle differences that make it tricky to use the correct regex. When picking a regex from the list of examples below, be sure to read the description with each regex to make sure you are picking the correct one.

Unless otherwise indicated, all examples below assume that the dot does *not* match newlines and that the caret and dollar *do* match at embedded line breaks. In many programming languages, this means that single-line mode must be off, and multi-line mode must be on.

When used by themselves, these regular expressions may not have the intended result. If a comment appears inside a string, the comment regex will consider the text inside the string as a comment. The string regex will also match strings inside comments. The solution is to use more than one regular expression and to combine those into a simple parser, like in this pseudo-code:

```
GlobalStartPosition := 0;
while GlobalStartPosition < LengthOfText do
  GlobalMatchPosition := LengthOfText;
  MatchedRegex := NULL;
  foreach Regex in RegexList do
    Regex.StartPosition := GlobalStartPosition;
    if Regex.Match and Regex.MatchPosition < GlobalMatchPosition then
      MatchedRegex := Regex;
      GlobalMatchPosition := Regex.MatchPosition;
    endif
  endforeach
  if MatchedRegex <> NULL then
    // At this point, MatchedRegex indicates which regex matched
    // and you can do whatever processing you want depending on
    // which regex actually matched.
  endif
  GlobalStartPosition := GlobalMatchPosition;
endwhile
```

If you put a regex matching a comment and a regex matching a string in `RegexList`, then you can be sure that the comment regex will not match comments inside strings, and vice versa. Inside the loop you can then process the match according to whether it is a comment or a string.

An alternative solution is to combine regexes: `(comment)|(string)`. The alternation has the same effect as the code snippet above. Iterate over all the matches of this regex. Inside the loop, check which capturing group found the regex match. If group 1 matched, you have a comment. If group two matched, you have a string. Then process the match according to that.

You can use this technique to build a full parser. Add regular expressions for all lexical elements in the language or file format you want to parse. Inside the loop, keep track of what was matched so that the following matches can be processed according to their context. For example, if curly braces need to be balanced, increment a counter when an opening brace is matched, and decrement it when a closing brace is matched. Raise an error if the counter goes negative at any point or if it is nonzero when the end of the file is reached.

Comments

`#.*$` matches a single-line comment starting with a `#` and continuing until the end of the line. Similarly, `//.*$` matches a single-line comment starting with `//`.

If the comment must appear at the start of the line, use `^#.*$`. If only whitespace is allowed between the start of the line and the comment, use `^\s*#.*$`. Compiler directives or pragmas in C can be matched this way. Note that in this last example, any leading whitespace will be part of the regex match. Use capturing parentheses to separate the whitespace and the comment.

`/*.*?*/` matches a C-style multi-line comment if you turn on the option for the dot to match newlines. The general syntax is `begin.*?end`. C-style comments do not allow nesting. If the “begin” part appears inside the comment, it is ignored. As soon as the “end” part is found, the comment is closed.

If your programming language allows nested comments, there is no straightforward way to match them using a regular expression, since regular expressions cannot count. Additional logic is required.

Strings

`"[^"\\r\\n]*"` matches a single-line string that does not allow the quote character to appear inside the string. Using the negated character class is more efficient than using a lazy dot. `"["]*"` allows the string to span across multiple lines.

`"[^"\\\\r\\n]*(?:\\\[^\\"\\\\r\\n]*\\")*"` matches a single-line string in which the quote character can appear if it is escaped by a backslash. Though this regular expression may seem more complicated than it needs to be, it is much faster than simpler solutions which can cause a whole lot of backtracking in case a double quote appears somewhere all by itself rather than part of a string. `"[^"\\]*(?:\\\[^\\"\\\\r\\n]*\\")*"` allows the string to span multiple lines.

You can adapt the above regexes to match any sequence delimited by two (possibly different) characters. If we use `b` for the starting character, `e` and the end, and `x` as the escape character, the version without escape becomes `b[^\r\n]*e`, and the version with escape becomes `b[^\r\n]*(?:x[^\r\n]*e)`.

Numbers

`\b\d+\b` matches a positive integer number. Do not forget the word boundaries! `[-+]? \b\d+\b` allows for a sign.

`\b0[xX][0-9a-fA-F]+\b` matches a C-style hexadecimal number.

`((\b[0-9]+)?\.\b[0-9]+\b)` matches an integer number as well as a floating point number with optional integer part. `(\b[0-9]+\.\b[0-9]+\b)?\.\b[0-9]+\b` matches a floating point number with optional integer as well as optional fractional part, but does not match an integer number.

`((\b[0-9]+)?\.\b[0-9]+\b)+([eE] [-+]?[0-9]+\b)` matches a number in scientific notation. The mantissa can be an integer or floating point number with optional integer part. The exponent is optional.

`\b[0-9]+(\.[0-9]+)?(e[+-]?[0-9]+)?\b` also matches a number in scientific notation. The difference with the previous example is that if the mantissa is a floating point number, the integer part is mandatory.

If you read through the floating point number example, you will notice that the above regexes are different from what is used there. The above regexes are more stringent. They use word boundaries to exclude numbers that are part of other things like identifiers. You can prepend `[-+]?` to all of the above regexes to include an optional sign in the regex. I did not do so above because in programming languages, the `+` and `-` are usually considered operators rather than signs.

Reserved Words or Keywords

Matching reserved words is easy. Simply use alternation to string them together: `\b(first|second|third|etc)\b` Again, do not forget the word boundaries.

12. Find Two Words Near Each Other

Some search tools that use boolean operators also have a special operator called “near”. Searching for “term1 near term2” finds all occurrences of term1 and term2 that occur within a certain “distance” from each other. The distance is a number of words. The actual number depends on the search tool, and is often configurable.

You can easily perform the same task with the proper regular expression.

Emulating “near” with a Regular Expression

With regular expressions you can describe almost any text pattern, including a pattern that matches two words near each other. This pattern is relatively simple, consisting of three parts: the first word, a certain number of unspecified words, and the second word. An unspecified word can be matched with the shorthand character class `\w+`. The spaces and other characters between the words can be matched with `\W+` (uppercase W this time).

The complete regular expression becomes `\bword1\W+(?:\w+\W+){1,6}?word2\b`. The quantifier `{1,6}?` makes the regex require at least one word between “word1” and “word2”, and allow at most six words.

If the words may also occur in reverse order, we need to specify the opposite pattern as well:

```
\b(?:word1\W+(?:\w+\W+){1,6}?word2|word2\W+(?:\w+\W+){1,6}?word1)\b
```

If you want to find any pair of two words out of a list of words, you can use:

```
\b(word1|word2|word3)(?:\W+\w+){1,6}?\W+(word1|word2|word3)\b
```

The final regex also finds a word near itself. It will match `word2 near word2`, for example.

13. Runaway Regular Expressions: Catastrophic Backtracking

Consider the regular expression `(x+x+)+y`. Before you scream in horror and say this contrived example should be written as `xx+y` or `x{2,}y` to match exactly the same without those terribly nested quantifiers: just assume that each “x” represents something more complex, with certain strings being matched by both “x”. See the section on HTML files below for a real example.

Let’s see what happens when you apply this regex to `xxxxxxxxxy`. The first `x+` will match all 10 `x` characters. The second `x+` fails. The first `x+` then backtracks to 9 matches, and the second one picks up the remaining `x`. The group has now matched once. The group repeats, but fails at the first `x+`. Since one repetition was sufficient, the group matches. `y` matches `y` and an overall match is found. The regex is declared functional, the code is shipped to the customer, and his computer explodes. Almost.

The above regex turns ugly when the `y` is missing from the subject string. When `y` fails, the regex engine backtracks. The group has one iteration it can backtrack into. The second `x+` matched only one `x`, so it can’t backtrack. But the first `x+` can give up one `x`. The second `x+` promptly matches `xx`. The group again has one iteration, fails the next one, and the `y` fails. Backtracking again, the second `x+` now has one backtracking position, reducing itself to match `x`. The group tries a second iteration. The first `x+` matches but the second is stuck at the end of the string. Backtracking again, the first `x+` in the group’s first iteration reduces itself to 7 characters. The second `x+` matches `xxx`. Failing `y`, the second `x+` is reduced to `xx` and then `x`. Now, the group can match a second iteration, with one `x` for each `x+`. But this (7,1),(1,1) combination fails too. So it goes to (6,4) and then (6,2)(1,1) and then (6,1),(2,1) and then (6,1),(1,2) and then I think you start to get the drift.

If you try this regex on a 10x string in RegexBuddy’s debugger, it’ll take 2558 steps to figure out the final `y` is missing. For an 11x string, it needs 5118 steps. For 12, it takes 10238 steps. Clearly we have an exponential complexity of $O(2^n)$ here. At 19x the debugger bows out because it won’t show more than one million steps.

RegexBuddy is forgiving in that it detects it’s going in circles and aborts the match attempt. Other regex engines (like .NET) will keep going forever, while others will crash with a stack overflow (like Perl, before version 5.10). Stack overflows are particularly nasty on Windows, since they tend to make your application vanish without a trace or explanation. Be very careful if you run a web service that allows users to supply their own regular expressions. People with little regex experience have surprising skill at coming up with exponentially complex regular expressions.

Possessive Quantifiers and Atomic Grouping to The Rescue

In the above example, the sane thing to do is obviously to rewrite it as `xx+y` which eliminates the nested quantifiers entirely. Nested quantifiers are repeated or alternated tokens inside a group that is itself repeated or alternated. These almost always lead to catastrophic backtracking. About the only situation where they don’t is when the start of each alternative inside the group is not optional, and mutually exclusive with the start of all the other alternatives, and mutually exclusive with the token that follows it (inside its alternative inside the group). E.g. `(a+b+|c+d+)+y` is safe. If anything fails, the regex engine will backtrack through the whole regex, but it will do so linearly. The reason is that all the tokens are mutually exclusive. None of them can match any characters matched by any of the others. So the match attempt at each backtracking position

will fail, causing the regex engine to backtrack linearly. If you test this on `aaaabbbbccccdddd`, RegexBuddy needs only 13 steps rather than millions of steps to figure it out.

However, it's not always possible or easy to rewrite your regex to make everything mutually exclusive. So we need a way to tell the regex engine not to backtrack. When we've grabbed all the x's, there's no need to backtrack. There couldn't possibly be a y in anything matched by either `x+`. Using a possessive quantifier, our regex becomes `(x+x+)+y`. This fails the `21x` string in merely 7 steps. That's 6 steps to match all the x's, and 1 step to figure out that y fails. Almost no backtracking is done. Using an atomic group, the regex becomes `(?>(x+x+)+)y` with the exact same results.

A Real Example: Matching CSV Records

Here's a real example from a technical support case I once handled. The customer was trying to find lines in a comma-delimited text file where the 12th item on a line started with a P. He was using the innocently-looking regexp `^(.*?,){11}P`.

At first sight, this regex looks like it should do the job just fine. The lazy dot and comma match a single comma-delimited field, and the `{11}` skips the first 11 fields. Finally, the P checks if the 12th field indeed starts with P. In fact, this is exactly what will happen when the 12th field indeed starts with a P.

The problem rears its ugly head when the 12th field does not start with a P. Let's say the string is `1,2,3,4,5,6,7,8,9,10,11,12,13`. At that point, the regex engine will backtrack. It will backtrack to the point where `^(.*?,){11}` had consumed `1,2,3,4,5,6,7,8,9,10,11`, giving up the last match of the comma. The next token is again the dot. The dot matches a comma. *The dot matches the comma!* However, the comma does not match the `1` in the 12th field, so the dot continues until the 11th iteration of `.*?`, has consumed `11,12,`. You can already see the root of the problem: the part of the regex (the dot) matching the contents of the field also matches the delimiter (the comma). Because of the double repetition (star inside `{11}`), this leads to a catastrophic amount of backtracking.

The regex engine now checks whether the 13th field starts with a P. It does not. Since there is no comma after the 13th field, the regex engine can no longer match the 11th iteration of `.*?`. But it does not give up there. It backtracks to the 10th iteration, expanding the match of the 10th iteration to `10,11,`. Since there is still no P, the 10th iteration is expanded to `10,11,12,`. Reaching the end of the string again, the same story starts with the 9th iteration, subsequently expanding it to `9,10,, 9,10,11,, 9,10,11,12,`. But between each expansion, there are more possibilities to be tried. When the 9th iteration consumes `9,10,`, the 10th could match just `11`, as well as `11,12,`. Continuously failing, the engine backtracks to the 8th iteration, again trying all possible combinations for the 9th, 10th, and 11th iterations.

You get the idea: the possible number of combinations that the regex engine will try for each line where the 12th field does not start with a P is huge. All this would take a long time if you ran this regex on a large CSV file where most rows don't have a P at the start of the 12th field.

Preventing Catastrophic Backtracking

The solution is simple. When nesting repetition operators, make absolutely sure that there is only one way to match the same match. If repeating the inner loop 4 times and the outer loop 7 times results in the same overall match as repeating the inner loop 6 times and the outer loop 2 times, you can be sure that the regex engine will try all those combinations.

In our example, the solution is to be more exact about what we want to match. We want to match 11 comma-delimited fields. The fields must not contain comma's. So the regex becomes: `^([\^,\r\n]*,){11}P`. If the P cannot be found, the engine will still backtrack. But it will backtrack only 11 times, and each time the `[\^,\r\n]` is not able to expand beyond the comma, forcing the regex engine to the previous one of the 11 iterations immediately, without trying further options.

See the Difference with RegexBuddy

The screenshot shows the RegexBuddy application interface. The top toolbar includes buttons for Match, Replace, Split, Copy, Paste, and other functions. The main window displays the regex `^([\^,\r\n]*,){11}P` and the test subject `1,2,3,4,5,6,7,8,9,10,11,12,13`. The History panel on the right shows the match attempt. The bottom panel shows the step-by-step execution of the regex engine, with the final message: "Match attempt failed after 52149 steps".

Step	Execution
52121	1,2,3,4,5,6,7,8,9,10,11,12, backtrack
52122	1,2,3,4,5,6,7,8,9,10,11,12,
52123	1,2,3,4,5,6,7,8,9,10,11,12, backtrack
52124	1,2,3,4,5,6,7,8,9,10,11,12,1
52125	1,2,3,4,5,6,7,8,9,10,11,12,1 backtrack
52126	1,2,3,4,5,6,7,8,9,10,11,12,13
52127	1,2,3,4,5,6,7,8,9,10,11,12,13 backtrack
52128	1,2,3,4,5,6,7,8,9,10,11, backtrack
52129	1,2,3,4,5,6,7,8,9,10,11,
52130	1,2,3,4,5,6,7,8,9,10,11, backtrack
52131	1,2,3,4,5,6,7,8,9,10,11,1
52132	1,2,3,4,5,6,7,8,9,10,11,1 backtrack
52133	1,2,3,4,5,6,7,8,9,10,11,12
52134	1,2,3,4,5,6,7,8,9,10,11,12,
52135	1,2,3,4,5,6,7,8,9,10,11,12, ok
52136	1,2,3,4,5,6,7,8,9,10,11,12, backtrack
52137	1,2,3,4,5,6,7,8,9,10,11,12,1
52138	1,2,3,4,5,6,7,8,9,10,11,12,1 backtrack
52139	1,2,3,4,5,6,7,8,9,10,11,12,13
52140	1,2,3,4,5,6,7,8,9,10,11,12,13 backtrack
52141	1,2,3,4,5,6,7,8,9,10,11,12, backtrack
52142	1,2,3,4,5,6,7,8,9,10,11,12,
52143	1,2,3,4,5,6,7,8,9,10,11,12, backtrack
52144	1,2,3,4,5,6,7,8,9,10,11,12,1
52145	1,2,3,4,5,6,7,8,9,10,11,12,1 backtrack
52146	1,2,3,4,5,6,7,8,9,10,11,12,13
52147	1,2,3,4,5,6,7,8,9,10,11,12,13 backtrack
52148	backtrack
52149	backtrack

Match attempt failed after 52149 steps

If you try this example with RegexBuddy's debugger, you will see that the original regex `^([\^,\r\n]*,){11}P` needs 25,593 steps to conclude there regex cannot match `1,2,3,4,5,6,7,8,9,10,11,12`. If the string is `1,2,3,4,5,6,7,8,9,10,11,12,13`, just 3 characters more, the number of steps doubles to 52,149. It's

not too hard to imagine that at this kind of exponential rate, attempting this regex on a large file with long lines could easily take forever.

Our improved regex `^([^\r\n]*,){11}P`, however, needs just 52 steps to fail, whether the subject string has 12 numbers, 13 numbers, 16 numbers or a billion. While the complexity of the original regex was exponential, the complexity of the improved regex is constant with respect to whatever follows the 11th field. The reason is the regex fails almost immediately when it discovers the 12th field doesn't start with a P. It backtracks the 11 iterations of the group without expanding again.

RegxBuddy

Ruby 2.4-2.6 | Helpful | Match | Replace | Split | Copy | Paste | History

Case sensitive | Exact spacing | Dot doesn't match line breaks | Reset

Regex: `^([^\r\n]*,){11}P`

History:

- 12th field starts with P (lazy dot)
- 12th field starts with P (negated class)
- 12th field starts with P (atomic)

Test | GREP | Forum

Test subject:

```

10 1,2,3,4,5
11 1,2,3,4,5,
12 1,2,3,4,5,6
13 1,2,3,4,5,6,
14 1,2,3,4,5,6,7
15 1,2,3,4,5,6,7,
16 1,2,3,4,5,6,7,8
17 1,2,3,4,5,6,7,8,
18 1,2,3,4,5,6,7,8,9
19 1,2,3,4,5,6,7,8,9,
20 1,2,3,4,5,6,7,8,9,10
21 1,2,3,4,5,6,7,8,9,10,
22 1,2,3,4,5,6,7,8,9,10,11
23 1,2,3,4,5,6,7,8,9,10,11,
24 1,2,3,4,5,6,7,8,9,10,11,
25 1,2,3,4,5,6,7,8,9,10,11,backtrack
26 1,2,3,4,5,6,7,8,9,10,1backtrack
27 1,2,3,4,5,6,7,8,9,10,1backtrack
28 1,2,3,4,5,6,7,8,9,10,ok
29 1,2,3,4,5,6,7,8,9,10,backtrack
30 1,2,3,4,5,6,7,8,9,1backtrack
31 1,2,3,4,5,6,7,8,9,1backtrack
32 1,2,3,4,5,6,7,8,9,ok
33 1,2,3,4,5,6,7,8,9,backtrack
34 1,2,3,4,5,6,7,8,ok
35 1,2,3,4,5,6,7,8,backtrack
36 1,2,3,4,5,6,7,ok
37 1,2,3,4,5,6,7,backtrack
38 1,2,3,4,5,6,ok
39 1,2,3,4,5,6,backtrack

```

Test results:

Whole file | CRLF pairs

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

The regular expression does not match the test subject

The complexity of the improved regex is linear to the length of the first 11 fields. 24 steps are needed in our example to match the 11 fields. It takes only 1 step to discover that P can't be matched. Another 27 steps are then needed to backtrack all the iterations of the two quantifiers. That's the best we can do, since the engine

does have to scan through all the characters of the first 11 fields to find out where the 12th one begins. Our improved regex is a perfect solution.

Alternative Solution Using Atomic Grouping

In the above example, we could easily reduce the amount of backtracking to a very low level by better specifying what we wanted. But that is not always possible in such a straightforward manner. In that case, you should use atomic grouping to prevent the regex engine from backtracking.

The screenshot shows the RegexpBuddy application interface. The main text area contains the regex pattern `^(>([^\r\n]*+,){11})P`. The History panel on the right shows three matches: "12th field starts with P (lazy dot)", "12th field starts with P (negated class)", and "12th field starts with P (atomic)". The Test panel at the bottom shows a list of 27 test cases, with the last two failing due to backtracking. The bottom status bar indicates "Match attempt failed after 27 steps".

Using atomic grouping, the above regex becomes `^(>([^\r\n]*+,){11})P`. Everything between `(?>)` is treated as one single token by the regex engine, once the regex engine leaves the group. Because the entire group is

one token, no backtracking can take place once the regex engine has found a match for the group. If backtracking is required, the engine has to backtrack to the regex token before the group (the caret in our example). If there is no token before the group, the regex must retry the entire regex at the next position in the string.

Let's see how `^(?>(.*?),){11}P` is applied to `1,2,3,4,5,6,7,8,9,10,11,12,13`. The caret matches at the start of the string and the engine enters the atomic group. The star is lazy, so the dot is initially skipped. But the comma does not match `1`, so the engine backtracks to the dot. That's right: backtracking is allowed here. The star is not possessive, and is not immediately enclosed by an atomic group. That is, the regex engine did not cross the closing parenthesis of the atomic group. The dot matches `1`, and the comma matches too. `{11}` causes further repetition until the atomic group has matched `1,2,3,4,5,6,7,8,9,10,11,`.

Now, the engine leaves the atomic group. Because the group is atomic, all backtracking information is discarded and the group is now considered a single token. The engine now tries to match `P` to the `1` in the 12th field. This fails.

So far, everything happened just like in the original, troublesome regular expression. Now comes the difference. `P` fails to match, so the engine backtracks. But the atomic group made it forget all backtracking positions. The match attempt at the start of the string fails immediately.

That is what atomic grouping and possessive quantifiers are for: efficiency by disallowing backtracking. The most efficient regex for our problem at hand would be `^(?>((?>[^\r\n]*)).){11}P`, since possessive, greedy repetition of the star is faster than a backtracking lazy dot. If possessive quantifiers are available, you can reduce clutter by writing `^(?>([^\r\n]*)+,){11}P`.

If you test the final solution in RegxBuddy's debugger, you'll see that it needs the same 24 steps to match the 11 fields. Then it takes 1 step to exit the atomic group and throw away all the backtracking information. Discovering that `P` can't be matched still takes one step. But because of the atomic group, backtracking it all takes only a single step.

Quickly Matching a Complete HTML File

Another common situation where catastrophic backtracking occurs is when trying to match "something" followed by "anything" followed by "another something" followed by "anything", where the lazy dot `.*` is used. The more "anything", the more backtracking. Sometimes, the lazy dot is simply a symptom of a lazy programmer. `".*"` is not appropriate to match a double-quoted string, since you don't really want to allow anything between the quotes. A string can't have (unescaped) embedded quotes, so `"[^\r\n]*"` is more appropriate, and won't lead to catastrophic backtracking when combined in a larger regular expression. However, sometimes "anything" really is just that. The problem is that "another something" also qualifies as "anything", giving us a genuine `x+x+` situation.

Suppose you want to use a regular expression to match a complete HTML file, and extract the basic parts from the file. If you know the structure of HTML files, it's very straightforward to write the regular expression

```
<html>.*<head>.*<title>.*</title>.*</head>.*<body>[^>]*.*</body>.*</html>.
```

With the "dot matches newlines" or "single line" matching mode turned on, it will work just fine on valid HTML files.

Unfortunately, this regular expression won't work nearly as well on an HTML file that misses some of the tags. The worst case is a missing `</html>` tag at the end of the file. When `</html>` fails to match, the regex engine backtracks, giving up the match for `</body>.*?`. It will then further expand the lazy dot before `</body>`, looking for a second closing `</body>` tag in the HTML file. When that fails, the engine gives up `<body[^>]*>.*?`, and starts looking for a second opening `<body[^>]*>` tag all the way to the end of the file. Since that also fails, the engine proceeds looking all the way to the end of the file for a second closing head tag, a second closing title tag, etc.

If you run this regex in RegexBuddy's debugger, the output will look like a sawtooth. The regex matches the whole file, backs up a little, matches the whole file again, backs up some more, backs up yet some more, matches everything again, etc. until each of the 7 `.*` tokens has reached the end of the file. The result is that this regular has a worst case complexity of N^7 . If you double the length of the HTML file with the missing `<html>` tag by appending text at the end, the regular expression will take 128 times (2^7) as long to figure out the HTML file isn't valid. This isn't quite as disastrous as the 2^N complexity of our first example, but will lead to very unacceptable performance on larger invalid files.

In this situation, we know that each of the literal text blocks in our regular expression (the HTML tags, which function as delimiters) will occur only once in a valid HTML file. That makes it very easy to package each of the lazy dots (the delimited content) in an atomic group.

`<html>(?!.*<head>)(?!.*<title>)(?!.*</title>)(?!.*</head>)(?!.*<body[^>]*>)(?!.*</body>).*</html>` will match a valid HTML file in the same number of steps as the original regex. The gain is that it will fail on an invalid HTML file almost as fast as it matches a valid one. When `</html>` fails to match, the regex engine backtracks, giving up the match for the last lazy dot. But then, there's nothing further to backtrack to. Since all of the lazy dots are in an atomic group, the regex engines has discarded their backtracking positions. The groups function as a "do not expand further" roadblock. The regex engine is forced to announce failure immediately.

You've no doubt noticed that each atomic group also contains an HTML tag after the lazy dot. This is critical. We do allow the lazy dot to backtrack until its matching HTML tag was found. E.g. when `.*/body>` is processing `Last paragraph</p></body>`, the `</` regex tokens will match `</` in `</p>`. However, `b` will fail `p`. At that point, the regex engine will backtrack and expand the lazy dot to include `</p>`. Since the regex engine hasn't left the atomic group yet, it is free to backtrack inside the group. Once `</body>` has matched, and the regex engine leaves the atomic group, it discards the lazy dot's backtracking positions. Then it can no longer be expanded.

Essentially, what we've done is to bind a repeated regex token (the lazy dot to match HTML content) to the non-repeated regex token that follows it (the literal HTML tag). Since anything, including HTML tags, can appear between the HTML tags in our regular expression, we cannot use a negated character class instead of the lazy dot to prevent the delimiting HTML tags from being matched as HTML content. But we can and did achieve the same result by combining each lazy dot and the HTML tag following it into an atomic group. As soon as the HTML tag is matched, the lazy dot's match is locked down. This ensures that the lazy dot will never match the HTML tag that should be matched by the literal HTML tag in the regular expression.

14. Runaway Regular Expressions: Too Many Repetitions

When a regular expression contains a repeated group such as `^(one|two)*done$` then it has two alternatives to try for each repetition of the group. In theory this regex should match a string with an arbitrarily long sequence of `oneoneone...done`. In practice, a backtracking regex engine will have to give up at some point.

If the regex engine uses a recursive algorithm then each repetition of the group adds a call to the engine's call stack. The engine will give up or even crash when the number of repetitions it actually needs to make exceeds the available stack space.

Even if the engine is non-recursive, it still has to keep track of where the group's match attempt started with each repetition. This is needed so the engine can backtrack if the remainder of the regex fails to match. When trying to match `oneoneone` the engine repeats the group 3 times. It stores a backtracking position for the quantifier before each repetition. The alternation operator also stores a backtracking position at each attempt. After the 3 repetitions `done` fails to match at the end of the string. Now the engine goes back through the 6 backtracking positions in reverse order. It attempts `two` at each backtracking position of the alternation operator. It attempts `done` at each backtracking position of the quantifier. The process is entirely linear. Even with a string that repeats `one` thousands of times the regex will match or fail to match instantly, depending on whether there's a `done` at the end of the string.

But if you use this regex on a string that repeats `one` millions of times then you may run into limitations of the regex engine designed to stop it from running out of memory trying to remember all those backtracking positions. This can prevent the engine from finding extremely long matches.

The above example makes matters worse by using a capturing group. At the end of a successful match, the capturing group will hold the last occurrence of `one` or `two` in the string. But during the match attempt, it also holds the most recent match of `one` or `two` with each iteration. This causes extra work for the regex engine with each iteration of the group and each time the group is backtracked into.

Optimize with Non-Capturing and Atomic Groups

We can optimize this regex to reduce the amount of work the regex engine has to do. These are good techniques to apply to all your regexes. Even if you'll only use your regexes on shorter strings where the performance difference is hardly measurable, you should treat them as good coding habits.

First of all, only use capturing groups if you really want to capture part of the regex match. Otherwise you can always use a non-capturing group. Turning a capturing group that doesn't have a backreference into a non-capturing group never changes what the regex is supposed to match. So `^(?:one|two)*done$` is our first optimization.

In this case, the two alternatives are mutually exclusive. `two` can never match at a position where `one` has already matched. So all that backtracking is unnecessary. We can tell the regex engine that by using an atomic group: `^(?>one|two)*done$`. Now the regex engine throws away the backtracking position of the alternation operator each time it repeats the group. It no longer attempts `two` at a position where `one` has already matched.

Optimize with Possessive Quantifiers

But the quantifier `*` still backtracks. `^(?>one|two)*done$` still attempts `done` at every position where `one` has matched as the quantifier backtracks. To stop this we can make the quantifier possessive: `^(?>one|two)*+done$`. This regex does not backtrack at all.

Whether this regex can really match an arbitrarily long sequence of `oneoneone...done` depends on how the regex engine implements possessive quantifiers. If the possessive quantifier does not store backtracking positions at all, then it can. But in some regex flavors the possessive quantifier is another way of writing `^(?>(?(?>one|two))*)*done$`. In that case, the quantifier still stores all its backtracking positions, only to throw them away when the regex engine exits the outer atomic group. This does improve performance when `done` fails to match. But it doesn't allow longer successful matches if the regex engine is limited by the number of backtracking positions that the quantifier can store.

Eliminate Needless Groups

Even better than turning capturing groups into non-capturing or atomic groups is to eliminate unnecessary groups. People sometimes needlessly group regex tokens because they do not understand the precedence of operators in a regex. The alternation operator has the lowest precedence of all. It alternates between everything to the left of it and everything to the right of it within the regex or group that contains it. A quantifier has high precedence. It repeats just the token or group in front of it.

So `one|two*` would match `one`, `tw`, `two`, `twoo`, `twooo`, etc. We needed the group to repeat the alternation instead of just the final `o`. But we don't need extra groups around the alternatives. The two nested groups in `(?: (?:one) | (?:two))*` are unnecessary.

`(?:[ab])+` also has a needless group. The character class is treated as a single regex token. The quantifier can repeat it directly: `[ab]+`.

How much an impact these unnecessary groups have depends on the regex engine. If you followed the advice to use non-capturing groups then the engine may be able to optimize away the unnecessary groups. But it can't do that if you have unnecessary capturing groups. The regex engine can't know whether you'll need to retrieve the text matched by the capturing groups afterwards, so it can't remove them.

Repeat Single Tokens

In theory, `^(?:a|b|c)+$` and `^[abc]+$` are the same. In practice, most backtracking engines execute the latter much faster. The character class can attempt both characters at the same time. It doesn't need to backtrack at all to try the other characters like the alternation operator does. Each iteration of the character class matches exactly one character. While the quantifier may need to backtrack, it doesn't need backtracking positions to do so. It just needs to remember the number of iterations. It can backtrack simply by stepping backwards one character in the string and decrementing the number of iterations. This enables `^[abc]+$` to match a string of any length.

`"(?:[^\"]|\\.)*"` is a simplistic solution to match a double-quoted string that may contain double quotes if they are escaped by a backslash. We allow line breaks and assume the dot matches them.

The above regex is correct in the sense that it matches all double-quoted strings and nothing else. But it is simplistic because it performs poorly. We could use an atomic group if we flipped the two alternatives (remember the regex engine is eager). But unless the regex engine has possessive quantifiers that don't store backtracking positions at all, we're not going to be able to make this regex match double-quoted strings that are millions of characters long as long as we're repeating the group for each character in the string.

To optimize this regex we need to repeat the negated character class: `"(?:[^\\"]+|\\.)"`. This allows the regex to quickly match runs of non-escaped characters within the string. Since those are far more common than escaped characters, this significantly reduces the number of backtracking positions the regex engine needs to remember. The outer group only repeats once for each run of non-escaped characters and once for each escaped character. The two quantifiers will still backtrack if the closing quote fails to match. But most iterations will be of the inner quantifier which can backtrack much faster.

Note that the negated character class now includes the backslash. This ensures the two alternatives are mutually exclusive. This is essential. If you paid attention to the catastrophic backtracking topic then you'll notice a similar pattern of nested quantifiers. Though our regex will backtrack when the closing quote fails to match, it does so linearly because the second alternative can never match a character that was matched by the first alternative.

We can take this optimization one step further. We don't need to repeat the group for runs of non-escaped characters and we don't need it to alternate it between escaped and non-escaped characters. We only need the group to handle escaped characters. `"[^\\"]*(?:\\.[^\\"])*"` treats a double-quoted string as a series of zero or more non-escaped characters followed by zero or more escaped characters that are each followed by zero or more non-escaped characters. Now the group only remembers one backtracking position for each non-escaped character in the string. This enables the regex to match strings of pretty much any length. It'll only run into regex engine limitations if a string should contain millions of escaped characters. It will backtrack if the closing quote fails to match. But all the backtracking attempts will immediately fail because the group starts with `\\.` which is mutually exclusive with `[^\\"]*`.

If the regex engine does support atomic grouping or possessive quantifiers then we can put the icing on the cake with `"(?:>[^\\"]*(?:>\\.[^\\"]*+))"` or `"[^\\"]*+(?:\\.[^\\"]*+)*"`. Both these regexes throw away all backtracking positions when attempting to match the closing double quote. So they never backtrack at all.

15. Preventing Regular Expression Denial of Service (ReDoS)

The previous topic explains catastrophic backtracking with practical examples from the perspective of somebody trying to get their regular expressions to work and perform well on their own PC. You should understand those examples before reading this topic.

It's annoying when catastrophic backtracking happens on your PC. But when it happens in a server application with multiple concurrent users, it can really be catastrophic. Too many users running regexes that exhibit catastrophic backtracking will bring down the whole server. And "too many" need only be as few as the number of CPU cores in the server.

If the server accepts regexes from the user, then the user can easily provide one that exhibits catastrophic backtracking on any subject. If the server accepts subject data from the user, then the user may be able to provide subjects that trigger catastrophic backtracking in regexes used by the server, if those regexes are predisposed to catastrophic backtracking. When the user can do either of those things, the server is susceptible to regular expression denial of service (ReDoS). When enough users (or one actor masquerading as many users) provide malicious regexes and/or subjects to match against, the server will be spending nearly all its CPU cycles on trying to match those regexes.

Handling Regexes Provided by The User

If your application allows the user to provide their own regexes, then your only real defense is to use a text-directed regex engine. Those engines don't backtrack. Their performance depends on the length of the subject string, not the complexity of the regular expression. But they also don't support features like backreferences that depend on backtracking and that many users expect.

If your application uses a backtracking engine with user-provided regexes, then you can only mitigate the consequences of catastrophic backtracking. And you'll really need to do so. It's very easy for people with limited regex skills to accidentally craft one that degenerates into catastrophic backtracking.

You'll need to use a regex engine that aborts the match attempt when catastrophic backtracking occurs rather than running until the script crashes or the OS kills it. You can easily test this. When the regex `(x\w{1,10})+y` is attempted on an ever growing string of `x`'s there should be a reasonable limit on how long it takes for the regex engine to give up. Ideally your engine will allow you to configure this limit for your purposes. The .NET engine, for example, allows you to pass a timeout to the `Regex()` constructor. The PCRE engine allows you to set recursion limits. The lower your limits the better the protection against ReDoS, but higher the risk of aborting legitimate regexes that would find a valid match given slightly more time. Low recursion limits may prevent long regex matches. Low timeouts may abort searches through large files too early.

If your regex engine has no such features, you could implement your own timeout. Spawn a separate thread to execute the regular expression. Wait on the thread with a timeout. If the thread finishes before the wait times out, process its result. Otherwise, kill the thread and tell the user the regex is too complex. The `safe_regex` package implements this for Ruby.

Reviewing Regexes in The Application

If the server only uses regexes that are hard-coded in your application, then you can prevent regex-based denial of service attacks entirely. You need to make sure that your regexes won't exhibit catastrophic backtracking regardless of the subjects they're used on. This isn't particularly difficult for somebody with a solid grasp of regular expressions. But it does require care and attention. It's not enough to just test that the regex matches valid subjects. You need to make sure, by looking at the regex independently of any subject data, that it is not possible for multiple permutations of the same regex to match the same thing.

Permutations occur when you give the regular expression a choice. You can do this with alternation and with quantifiers. So these are the regex tokens you need to inspect. Possessive quantifiers are excepted, because they never backtrack.

Alternation

Alternatives must be mutually exclusive. If multiple alternatives can match the same text then the engine will try both if the remainder of the regex fails. If the alternatives are in a group that is repeated, you have catastrophic backtracking.

A classic example is `(.|\s)*` to match any amount of any text when the regex flavor does not have a “dot matches line breaks” mode. If this is part of a longer regex then a subject string with a sufficiently long run of spaces will break the regex. The engine will try every possible combination of the spaces being matched by `.` or `\s`. For example, 3 spaces could be matched as `...`, `..\s`, `.\s.`, `.\s\s`, `\s.`, `\s.\s`, `\s\s.`, or `\s\s\s`. That's 2^N permutations. The fix is to use `(.|\n)*` to make the alternatives mutually exclusive. Even better to be more specific about which characters are really allowed, such as `[\r\n\t\x20-\x7E]*` for ASCII printables, tabs, and line breaks.

It is acceptable for two alternatives to partially match the same text. `[0-9]*\.[0-9]+|[0-9]+` is perfectly fine to match a floating point number with optional integer part and optional fraction. Though a subject that consists of only digits is initially matched by `[0-9]*` and does cause some backtracking when `\.` fails, this backtracking never becomes catastrophic. Even if you put this inside a group in a longer regex, the group only does a minimal amount of backtracking. (But the group mustn't have a quantifier or it will fall foul of the rule for nested quantifiers.)

Quantifiers in Sequence

Quantified tokens that are in sequence must either be mutually exclusive with each other or be mutually exclusive with what comes between them. Otherwise both can match the same text and all combinations of the two quantifiers will be tried when the remainder of the regex fails to match. A token inside a group with alternation is still in sequence with any token before or after the group.

A classic example is `a.*?b.*?c` to match 3 things with “anything” between them. When `c` can't be matched the first `.*?` expands character by character until the end of the line or file. For each expansion the second `.*?` expands character by character to match the remainder of the line or file. The fix is to realize that you can't have “anything” between them. The first run needs to stop at `b` and the second run needs to stop at `c`. With single characters `a[^b]*b[^c]*c` is an easy solution. The negated character classes guarantee the

repetition stops at the delimiter. If your regex flavor supports possessive quantifiers then you can use `a+b+c+` to further increase performance.

For a more complex example and solution, see matching a complete HTML file in the previous topic. This explains how you can use atomic grouping to prevent backtracking in more complex situations.

Nested Quantifiers

A group that contains a token with a quantifier must not have a quantifier of its own unless the quantified token inside the group can only be matched with something else that is mutually exclusive with it. That ensures that there is no way that fewer iterations of the outer quantifier with more iterations of the inner quantifier can match the same text as more iterations of the outer quantifier with fewer iterations of the inner quantifier.

The regex `(x+w{1,10})+y` matches a sequence of one or more codes that start with an `x` followed by 1 to 10 word characters, all followed by a `y`. All is well as long as the `y` can be matched. When the `y` is missing, backtracking occurs. If the string doesn't have too many `x`'s then backtracking happens very quickly. Things only turn catastrophic when the subject contains a long sequence of `x`'s. `x` and `x` are not mutually exclusive. So the repeated group can match `xxxx` in one iteration as `x+w{1,10}` or in two iterations as `x+w{1,10}x+w{1,10}`.

To solve this, you first need to consider whether `x` and `y` should be allowed in the 1 to 10 characters that follow it. Excluding the `x` eliminates most backtracking. What's left won't be catastrophic. You could exclude it with character class subtraction as in `(x+[+w-+x]{1,10})+y` or with character class intersection as in `(x+[+w&&+[+x]+]{1,10})+y`. If you don't have those features you'll need to spell out the characters you want to allow: `(x+[a-wyz0-9_]{1,10})+y`.

If the `x` should be allowed then your only solution is to disallow the `y` in the same way. Then you can make the group atomic or the quantifier possessive to eliminate the backtracking.

If both `x` and `y` should be allowed in the sequences of 1 to 10 characters, then there is no regex-only solution. You can't make the group atomic or the quantifier possessive as then `w{1,10}` matches the final `y` which causes `y` to fail.

Other Defensive Techniques

In addition to preventing catastrophic backtracking as explained above, you should make your regular expressions as strict as possible. The stricter the regex, the less backtracking it does and thus the better it performs. Even if you can't measure the performance difference because the regex is used infrequently on short strings, proper technique is a habit. It also reduces the chance that a less experienced developer introduces catastrophic backtracking when they extend your regex later.

Make groups that contain alternatives atomic as much as you can. Use `\b(?:one|two|three)\b` to match a list of words.

Make quantifiers possessive as much as you can. If a repeated token is mutually exclusive with what follows, enforce that with a possessive quantifier.

Use (negated) character classes instead of the dot. It's rare that you really want to allow "anything". A double-quoted string, for example, can't contain "anything". It can't contain unescaped double quotes. So use `"[^\n"]*"` instead of `".*?"`. Though both find exactly the same matches when used on their own, the latter can lead to catastrophic backtracking when pasted into a longer regex. The former never backtracks regardless of anything else the regex needs to match.

Why Use Regexes at All?

Some would certainly argue that the above only shows that regexes are dangerous and that they should not be used. They'll then force developers to do the job with procedural code. Procedural code to match non-trivial patterns quickly becomes long and complicated, increasing the chance of bugs and the cost to develop and maintain the code. Many pattern matching problems are naturally solved with recursion. And when a large subject string can't be matched, runaway recursion leads to stack overflows that crash the application.

Developers need to learn to correctly use their tools. This is no different for regular expressions than for anything else.

16. Repeating a Capturing Group vs. Capturing a Repeated Group

When creating a regular expression that needs a capturing group to grab part of the text matched, a common mistake is to repeat the capturing group instead of capturing a repeated group. The difference is that the repeated capturing group will capture only the last iteration, while a group capturing another group that's repeated will capture all iterations. An example will make this clear.

Let's say you want to match a tag like `!abc!` or `!123!`. Only these two are possible, and you want to capture the `abc` or `123` to figure out which tag you got. That's easy enough: `!(abc|123)!` will do the trick.

Now let's say that the tag can contain multiple sequences of `abc` and `123`, like `!abc123!` or `!123abcabc!`. The quick and easy solution is `!(abc|123)+!`. This regular expression will indeed match these tags. However, it no longer meets our requirement to capture the tag's label into the capturing group. When this regex matches `!abc123!`, the capturing group stores only `123`. When it matches `!123abcabc!`, it only stores `abc`.

This is easy to understand if we look at how the regex engine applies `!(abc|123)+!` to `!abc123!`. First, `!` matches `!`. The engine then enters the capturing group. It makes note that capturing group #1 was entered when the engine reached the position between the first and second character in the subject string. The first token in the group is `abc`, which matches `abc`. A match is found, so the second alternative isn't tried. (The engine does store a backtracking position, but this won't be used in this example.) The engine now leaves the capturing group. It makes note that capturing group #1 was exited when the engine reached the position between the 4th and 5th characters in the string.

After having exited from the group, the engine notices the plus. The plus is greedy, so the group is tried again. The engine enters the group again, and takes note that capturing group #1 was entered between the 4th and 5th characters in the string. It also makes note that since the plus is not possessive, it may be backtracked. That is, if the group cannot be matched a second time, that's fine. In this backtracking note, the regex engine also saves the entrance and exit positions of the group during the previous iteration of the group.

`abc` fails to match `123`, but `123` succeeds. The group is exited again. The exit position between characters 7 and 8 is stored.

The plus allows for another iteration, so the engine tries again. Backtracking info is stored, and the new entrance position for the group is saved. But now, both `abc` and `123` fail to match `!`. The group fails, and the engine backtracks. While backtracking, the engine restores the capturing positions for the group. Namely, the group was entered between characters 4 and 5, and existed between characters 7 and 8.

The engine proceeds with `!`, which matches `!`. An overall match is found. The overall match spans the whole subject string. The capturing group spans characters 5, 6 and 7, or `123`. Backtracking information is discarded when a match is found, so there's no way to tell after the fact that the group had a previous iteration that matched `abc`.

The solution to capturing `abc123` in this example should be obvious now: the regex engine should enter and leave the group only once. This means that the plus should be inside the capturing group rather than outside. Since we do need to group the two alternatives, we'll need to place a second capturing group around the repeated group: `!((abc|123)+)!`. When this regex matches `!abc123!`, capturing group #1 will store

`abc123`, and group #2 will store `123`. Since we're not interested in the inner group's match, we can optimize this regular expression by making the inner group non-capturing: `!(?:abc|123)+!`.

17. Mixing Unicode and 8-bit Character Codes

Internally, computers deal with numbers, not with characters. When you save a text file, each character is mapped to a number, and the numbers are stored on disk. When you open a text file, the numbers are read and mapped back to characters. When processing text with a regular expression, the regular expression needs to use the same mapping as you used to create the file or string you want the regex to process.

When you simply type in all the characters in your regular expression, you normally don't have anything to worry about. The application or programming library that provides the regular expression functionality will know what text encodings your subject string uses, and process it accordingly. So if you want to search for the euro currency symbol, and you have a European keyboard, just press AltGr+E. Your regex € will find all euro symbols just fine.

But you can't press AltGr+E on a US keyboard. Or perhaps you like your source code to be 7-bit clean (i.e. plain ASCII). In those cases, you'll need to use a character escape in your regular expression.

In EditPad, simply use the Unicode escape \u20AC. U+20AC is the Unicode code point for the euro symbol. It will always match the euro symbol, whether your file is encoded in UTF-8, UTF-16, UCS-2 or whatever. Even when your file is encoded with a legacy 8-bit code page, there's no confusion. \u20AC is always the euro symbol.

EditPad also supports the 8-bit character escape \xFF. However, its use is not recommended. For characters \x00 through \x7F, there's usually no trouble. The first 128 Unicode code points are identical to the ASCII table that most 8-bit code pages are based on.

Just Great Software applications treat \x80 as \u0080 when searching through a Unicode text file, but as \u20AC when searching through a Windows 1252 text file. There's no magic here. It matches the character with index 80h in the text file, regardless of the text file's encoding. Unicode code point U+0080 is a Latin-1 control code, while Windows 1252 character index 80h is the euro symbol. In reverse, if you type in the euro symbol in a text editor, saving it as UTF-16LE will save two bytes AC 20, while saving as Windows 1252 will give you one byte 80.

If you find the above confusing, simply don't use \x80 through \xFF with EditPad.

Part 4

Regular Expression Reference

1. Special and Non-Printable Characters

Feature:	Literal character
Syntax:	Any character except <code>[\\^\$. ?*+()]</code>
Description:	All characters except the listed special characters match a single instance of themselves
Example:	<code>a</code> matches <code>a</code>
Feature:	Literal curly braces
Syntax:	<code>{</code> and <code>}</code>
Description:	<code>{</code> and <code>}</code> are literal characters, unless they're part of a valid regular expression token such as a quantifier <code>{3}</code>
Example:	<code>{</code> matches <code>{</code>
Feature:	Backslash escapes a metacharacter
Syntax:	<code>\</code> followed by any of <code>[\\^\$. ?*+()]</code>
Description:	A backslash escapes special characters to suppress their special meaning
Example:	<code>*</code> matches <code>*</code>
Feature:	Escape sequence
Syntax:	<code>\Q... \E</code>
Description:	Matches the characters between <code>\Q</code> and <code>\E</code> literally, suppressing the meaning of special characters
Example:	<code>\Q+ - * / \E</code> matches <code>+ - * /</code>
Feature:	Hexadecimal escape
Syntax:	<code>\xFF</code> where FF are 2 hexadecimal digits
Description:	Matches the character at the specified position in the code page
Example:	<code>\xA9</code> matches <code>©</code> when using the Latin-1 code page
Feature:	Character escape
Syntax:	<code>\n</code> , <code>\r</code> and <code>\t</code>
Description:	Match an LF character, CR character and a tab character respectively
Example:	<code>\r\n</code> matches a Windows CRLF line break
Feature:	Line break
Syntax:	<code>\R</code>
Description:	Matches any line break, including CRLF as a pair, CR only, LF only, form feed, vertical tab, and any Unicode line break
Example:	
Feature:	Line break
Syntax:	<code>\R</code>
Description:	Matches the next line control character U+0085
Example:	
Feature:	Line break
Syntax:	<code>\R</code>
Description:	CRLF line breaks are indivisible
Example:	<code>\R{2}</code> and <code>\R\R</code> cannot match <code>\r\n</code>

Feature: Line break
Syntax: Literal CRLF, LF, or CR line break
Description: Matches CRLF as a pair, CR only, and LF only regardless of the line break style used in the regex
Example:

Feature: Character escape
Syntax: `\a`
Description: Match the “alert” or “bell” control character (ASCII 0x07)
Example:

Feature: Character escape
Syntax: `\e`
Description: Match the “escape” control character (ASCII 0x1B)
Example:

Feature: Character escape
Syntax: `\f`
Description: Match the “form feed” control character (ASCII 0x0C)
Example:

Feature: Octal escape
Syntax: `\o{7777}` where 7777 is any octal number
Description: Matches the character at the specified position in the active code page
Example: `\o{20254}` matches € when using Unicode

2. Basic Features

Feature:	Dot
Syntax:	<code>.</code> (dot)
Description:	Matches any single character except line break characters. Most regex flavors have an option to make the dot match line break characters too.
Example:	<code>.</code> matches <code>x</code> or (almost) any other character
Feature:	Not a line break
Syntax:	<code>\N</code>
Description:	Matches any single character except line break characters, like the dot, but is not affected by any options that make the dot match all characters including line breaks.
Example:	<code>\N</code> matches <code>x</code> or any other character that is not a line break
Feature:	Alternation
Syntax:	<code> </code> (pipe)
Description:	Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of alternatives.
Example:	<code>abc def xyz</code> matches <code>abc</code> , <code>def</code> or <code>xyz</code>
Feature:	Alternation is eager
Syntax:	<code> </code>
Description:	Alternation returns the first alternative that matches.
Example:	<code>a ab</code> matches <code>a</code> in <code>ab</code>

3. Character Classes

Feature:	Character class
Syntax:	[
Description:	When used outside a character class, [begins a character class. Inside a character class, different rules apply. Unless otherwise noted, the syntax on this page is only valid inside character classes, while the syntax on all other reference pages is not valid inside character classes.
Example:	
Feature:	Literal character
Syntax:	Any character except ^ -] \
Description:	All characters except the listed special characters are literal characters that add themselves to the character class.
Example:	[abc] matches a , b or c
Feature:	Backslash escapes a metacharacter
Syntax:	\ (backslash) followed by any of ^ -] \
Description:	A backslash escapes special characters to suppress their special meaning.
Example:	[\^\]] matches ^ or]
Feature:	Range
Syntax:	- (hyphen) between two tokens that each specify a single character.
Description:	Adds a range of characters to the character class.
Example:	[a-zA-Z0-9] matches any ASCII letter or digit
Feature:	Negated character class
Syntax:	^ (caret) immediately after the opening [
Description:	Negates the character class, causing it to match a single character <i>not</i> listed in the character class.
Example:	[^a-d] matches x (any character except a, b, c or d)
Feature:	Literal opening bracket
Syntax:	[
Description:	An opening square bracket is a literal character that adds an opening square bracket to the character class.
Example:	[ab[cd]ef] matches aef] , bef] , [ef] , cef] , and def]
Feature:	Character class subtraction
Syntax:	[base-[subtract]]
Description:	Removes all characters in the “subtract” class from the “base” class.
Example:	[a-z-[aeiou]] matches a single letter that is not a vowel.
Feature:	Character class intersection
Syntax:	[base&&[intersect]]
Description:	Reduces the character class to the characters present in both “base” and “intersect”.
Example:	[a-z&&[^aeiou]] matches a single letter that is not a vowel.

Feature: Character escape
 Syntax: `\n`, `\r` and `\t`
 Description: Add an LF character, a CR character, or a tab character to the character class, respectively.
 Example: `[\n\r\t]` a line feed, a carriage return, or a tab.

Feature: Character escape
 Syntax: `\a`
 Description: Add the “alert” or “bell” control character (ASCII 0x07) to the character class.
 Example: `[\a\t]` matches a bell or a tab character.

Feature: Character escape
 Syntax: `\b`
 Description: Add the “backspace” control character (ASCII 0x08) to the character class.
 Example: `[\b\t]` matches a backspace or a tab character.

Feature: Character escape
 Syntax: `\e`
 Description: Add the “escape” control character (ASCII 0x1B) to the character class.
 Example: `[\e\t]` matches an escape or a tab character.

Feature: Character escape
 Syntax: `\f`
 Description: Add the “form feed” control character (ASCII 0x0C) to the character class.
 Example: `[\f\t]` matches a form feed or a tab character.

Feature: POSIX class
 Syntax: `[:alpha:]`
 Description: Matches one character from a POSIX character class. Can only be used in a bracket expression.
 Example: `[[[:digit:]][[:lower:]]]` matches one of `0` through `9` or `a` through `z`

Feature: POSIX shorthand class
 Syntax: `[:d:]`, `[:s:]`, `[:w:]`
 Description: Matches one character from the POSIX character classes “digit”, “space”, or “word”. Can only be used in a bracket expression.
 Example: `[[[:s:]][[:d:]]]` matches a space, a tab, a line break, or one of `0` through `9`

Feature: POSIX shorthand class
 Syntax: `[:l:]` and `[:u:]`
 Description: Matches one character from the POSIX character classes “lower” or “upper”. Can only be used in a bracket expression.
 Example: `[[[:u:]][[:l:]]]` matches `Aa` but not `aA`.

Feature: POSIX shorthand class
 Syntax: `[:h:]`
 Description: Matches one character from the POSIX character classes “blank”. Can only be used in a bracket expression.
 Example: `[[[:h:]]]` matches a space.

Feature: POSIX shorthand class
Syntax: `[:V:]`
Description: Matches a vertical whitespace character. Can only be used in a bracket expression.
Example: `[[:v:]]` match any single vertical whitespace character.

Feature: POSIX class
Syntax: Any supported `\p{...}` syntax
Description: `\p{...}` syntax can be used inside character classes.
Example: `[\p{Digit}\p{Lower}]` matches one of `0` through `9` or `a` through `z`

Feature: POSIX class
Syntax: `\p{Alpha}`
Description: Matches one character from a POSIX character class.
Example: `\p{Digit}` matches any single digit.

Feature: POSIX class
Syntax: `\p{IsAlpha}`
Description: Matches one character from a POSIX character class.
Example: `\p{IsDigit}` matches any single digit.

4. Shorthand Character Classes

Feature:	Shorthand
Syntax:	Any shorthand outside character classes
Description:	Shorthands can be used outside character classes.
Example:	<code>\w</code> matches a single word character
Feature:	Shorthand
Syntax:	Any shorthand inside a character class
Description:	Shorthands can be used inside character classes.
Example:	<code>[\w]</code> matches a single word character
Feature:	Shorthand
Syntax:	Any negated shorthand inside a character class
Description:	Negated shorthands can be used inside character classes.
Example:	<code>[\W]</code> matches a single character that is not a word character
Feature:	Shorthand
Syntax:	<code>\d</code>
Description:	Adds all digits to the character class. Matches a single digit if used outside character classes.
Example:	<code>[\d]</code> and/or <code>\d</code> match a character that is a digit
Feature:	Shorthand
Syntax:	<code>\w</code>
Description:	Adds all word characters to the character class. Matches a single word character if used outside character classes.
Example:	<code>[\w]</code> and/or <code>\w</code> match any single word character
Feature:	Shorthand
Syntax:	<code>\s</code>
Description:	Adds all whitespace to the character class. Matches a single whitespace character if used outside character classes.
Example:	<code>[\s]</code> and/or <code>\s</code> match any single whitespace character
Feature:	Shorthand
Syntax:	<code>\l</code> and <code>\u</code>
Description:	Adds all lowercase letters or all uppercase letters to the character class. Matches a single lowercase or uppercase letter if used outside character classes.
Example:	<code>\u\l</code> matches <code>Aa</code> but not <code>aA</code> .
Feature:	Shorthand
Syntax:	<code>\v</code>
Description:	Adds all vertical whitespace to the character class. Matches a single vertical whitespace character if used outside character classes.
Example:	<code>[\v]</code> and/or <code>\v</code> match any single vertical whitespace character

Feature: Shorthand
Syntax: `\h`
Description: Adds all horizontal whitespace to the character class. Matches a single horizontal whitespace character if used outside character classes.
Example: `[\h]` and/or `\h` match any single horizontal whitespace character

Feature: XML Shorthand
Syntax: `\i`
Description: Adds all characters that are allowed as the initial character in XML names to the character class. Matches one such character if used outside character classes.
Example: `\i\c*` matches an XML name

Feature: XML Shorthand
Syntax: `\c`
Description: Adds all characters that are allowed as the second and following characters in XML names to the character class. Matches one such character if used outside character classes.
Example: `\i\c*` matches an XML name

5. Anchors

Feature:	String anchor
Syntax:	<code>^</code> (caret)
Description:	Matches at the start of the string the regex pattern is applied to.
Example:	<code>^.</code> matches <code>a</code> in <code>abc\ndef</code>
Feature:	String anchor
Syntax:	<code>\$</code> (dollar)
Description:	Matches at the end of the string the regex pattern is applied to.
Example:	<code>.\$</code> matches <code>f</code> in <code>abc\ndef</code>
Feature:	String anchor
Syntax:	<code>\$</code> (dollar)
Description:	Matches before the final line break in the string (if any) in addition to matching at the very end of the string.
Example:	<code>.\$</code> matches <code>f</code> in <code>abc\ndef\n</code>
Feature:	Line anchor
Syntax:	<code>^</code> (caret)
Description:	Matches after each line break in addition to matching at the start of the string, thus matching at the start of each line in the string.
Example:	<code>^.</code> matches <code>a</code> and <code>d</code> in <code>abc\ndef</code>
Feature:	Line anchor
Syntax:	<code>\$</code> (dollar)
Description:	Matches before each line break in addition to matching at the end of the string, thus matching at the end of each line in the string.
Example:	<code>.\$</code> matches <code>c</code> and <code>f</code> in <code>abc\ndef</code>
Feature:	String anchor
Syntax:	<code>\A</code>
Description:	Matches at the start of the string the regex pattern is applied to.
Example:	<code>\A\w</code> matches only <code>a</code> in <code>abc</code>
Feature:	Attempt anchor
Syntax:	<code>\G</code>
Description:	Matches at the start of the match attempt.
Example:	<code>\G\w</code> matches <code>a</code> , <code>b</code> , and <code>c</code> when iterating over all matches in <code>abc def</code>
Feature:	String anchor
Syntax:	<code>\z</code>
Description:	Matches at the end of the string the regex pattern is applied to.
Example:	<code>\w\z</code> matches <code>f</code> in <code>abc\ndef</code> but fails to match <code>abc\ndef\n</code>
Feature:	String anchor
Syntax:	<code>\Z</code>
Description:	Matches at the end of the string as well as before the final line break in the string (if any).
Example:	<code>.\Z</code> matches <code>f</code> in <code>abc\ndef</code> and in <code>abc\ndef\n</code> but fails to match <code>abc\ndef\n\n</code>

6. Word Boundaries

Feature:	Word boundary
Syntax:	<code>\b</code>
Description:	Matches at a position that is followed by a word character but not preceded by a word character, or that is preceded by a word character but not followed by a word character.
Example:	<code>\b.</code> matches <code>a</code> , <code>,</code> , and <code>d</code> in <code>abc def</code>
Feature:	Word boundary
Syntax:	<code>\B</code>
Description:	Matches at a position that is preceded and followed by a word character, or that is not preceded and not followed by a word character.
Example:	<code>\B.</code> matches <code>b</code> , <code>c</code> , <code>e</code> , and <code>f</code> in <code>abc def</code>
Feature:	Tcl word boundary
Syntax:	<code>\y</code>
Description:	Matches at a position that is followed by a word character but not preceded by a word character, or that is preceded by a word character but not followed by a word character.
Example:	<code>\y.</code> matches <code>a</code> , <code>,</code> , and <code>d</code> in <code>abc def</code>
Feature:	Tcl word boundary
Syntax:	<code>\Y</code>
Description:	Matches at a position that is preceded and followed by a word character, or that is not preceded and not followed by a word character.
Example:	<code>\Y.</code> matches <code>b</code> , <code>c</code> , <code>e</code> , and <code>f</code> in <code>abc def</code>
Feature:	Tcl word boundary
Syntax:	<code>\m</code>
Description:	Matches at a position that is followed by a word character but not preceded by a word character.
Example:	<code>\m.</code> matches <code>a</code> and <code>d</code> in <code>abc def</code>
Feature:	Tcl word boundary
Syntax:	<code>\M</code>
Description:	Matches at a position that is preceded by a word character but not followed by a word character.
Example:	<code>.\M</code> matches <code>c</code> and <code>f</code> in <code>abc def</code>

7. Quantifiers

Feature:	Greedy quantifier
Syntax:	<code>?</code> (question mark)
Description:	Makes the preceding item optional. Greedy, so the optional item is included in the match if possible.
Example:	<code>abc?</code> matches <code>abc</code> or <code>ab</code>
Feature:	Lazy quantifier
Syntax:	<code>??</code>
Description:	Makes the preceding item optional. Lazy, so the optional item is excluded in the match if possible.
Example:	<code>abc??</code> matches <code>ab</code> or <code>abc</code>
Feature:	Possessive quantifier
Syntax:	<code>?+</code>
Description:	Makes the preceding item optional. Possessive, so if the optional item can be matched, then the quantifier won't give up its match even if the remainder of the regex fails.
Example:	<code>abc?+c</code> matches <code>abcc</code> but not <code>abc</code>
Feature:	Greedy quantifier
Syntax:	<code>*</code> (star)
Description:	Repeats the previous item zero or more times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is not matched at all.
Example:	<code>".*"</code> matches <code>"def" "ghi"</code> in <code>abc "def" "ghi" jkl</code>
Feature:	Lazy quantifier
Syntax:	<code>*?</code>
Description:	Repeats the previous item zero or more times. Lazy, so the engine first attempts to skip the previous item, before trying permutations with ever increasing matches of the preceding item.
Example:	<code>".*?"</code> matches <code>"def"</code> and <code>"ghi"</code> in <code>abc "def" "ghi" jkl</code>
Feature:	Possessive quantifier
Syntax:	<code>*+</code>
Description:	Repeats the previous item zero or more times. Possessive, so as many items as possible will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
Example:	<code>".*+"</code> can never match anything
Feature:	Greedy quantifier
Syntax:	<code>+</code> (plus)
Description:	Repeats the previous item once or more. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only once.
Example:	<code>".+"</code> matches <code>"def" "ghi"</code> in <code>abc "def" "ghi" jkl</code>

Feature:	Lazy quantifier
Syntax:	<code>+?</code>
Description:	Repeats the previous item once or more. Lazy, so the engine first matches the previous item only once, before trying permutations with ever increasing matches of the preceding item.
Example:	<code>" .+?"</code> matches <code>"def"</code> and <code>"ghi"</code> in <code>abc "def" "ghi" jkl</code>
Feature:	Possessive quantifier
Syntax:	<code>++</code>
Description:	Repeats the previous item once or more. Possessive, so as many items as possible will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
Example:	<code>" .++"</code> can never match anything
Feature:	Fixed quantifier
Syntax:	<code>{n}</code> where <code>n</code> is an integer ≥ 1
Description:	Repeats the previous item exactly <code>n</code> times.
Example:	<code>a{3}</code> matches <code>aaa</code>
Feature:	Greedy quantifier
Syntax:	<code>{n,m}</code> where $n \geq 0$ and $m \geq n$
Description:	Repeats the previous item between <code>n</code> and <code>m</code> times. Greedy, so repeating <code>m</code> times is tried before reducing the repetition to <code>n</code> times.
Example:	<code>a{2,4}</code> matches <code>aaaa</code> , <code>aaa</code> or <code>aa</code>
Feature:	Greedy quantifier
Syntax:	<code>{n,}</code> where $n \geq 0$
Description:	Repeats the previous item at least <code>n</code> times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only <code>n</code> times.
Example:	<code>a{2,}</code> matches <code>aaaaa</code> in <code>aaaaa</code>
Feature:	Greedy quantifier
Syntax:	<code>{,m}</code> where $m \geq 1$
Description:	Repeats the previous item between zero and <code>m</code> times. Greedy, so repeating <code>m</code> times is tried before reducing the repetition to zero times.
Example:	<code>a{,4}</code> matches <code>aaaa</code> , <code>aaa</code> , <code>aa</code> , <code>a</code> , or the empty string
Feature:	Lazy quantifier
Syntax:	<code>{n,m}?</code> where $n \geq 0$ and $m \geq n$
Description:	Repeats the previous item between <code>n</code> and <code>m</code> times. Lazy, so repeating <code>n</code> times is tried before increasing the repetition to <code>m</code> times.
Example:	<code>a{2,4}?</code> matches <code>aa</code> , <code>aaa</code> or <code>aaaa</code>
Feature:	Lazy quantifier
Syntax:	<code>{n,}?</code> where $n \geq 0$
Description:	Repeats the previous item <code>n</code> or more times. Lazy, so the engine first matches the previous item <code>n</code> times, before trying permutations with ever increasing matches of the preceding item.
Example:	<code>a{2,}?</code> matches <code>aa</code> in <code>aaaaa</code>

Feature: Lazy quantifier
 Syntax: $\{, m\}?$ where $m \geq 1$
 Description: Repeats the previous item between zero and m times. Lazy, so repeating zero times is tried before increasing the repetition to m times.
 Example: `a{,4}?` matches the empty string, `a`, `aa`, `aaa` or `aaaa`

Feature: Possessive quantifier
 Syntax: $\{n, m\}+$ where $n \geq 0$ and $m \geq n$
 Description: Repeats the previous item between n and m times. Possessive, so as many items as possible up to m will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
 Example: `a{2,4}+a` matches `aaaaa` but not `aaaa`

Feature: Possessive quantifier
 Syntax: $\{n, \}+$ where $n \geq 0$
 Description: Repeats the previous item n or more times. Possessive, so as many items as possible will be matched, without trying any permutations with less matches even if the remainder of the regex fails.
 Example: `a{2,}+a` never matches anything

8. Unicode Syntax Reference

This reference page explains what the Unicode tokens do when used outside character classes. All of these except `\X` can also be used inside character classes. Inside a character class, these tokens add the characters that they normally match to the character class.

Feature: Grapheme
 Syntax: `\X`
 Description: Matches a single Unicode grapheme, whether encoded as a single code point or multiple code points using combining marks. A grapheme most closely resembles the everyday concept of a “character”.
 Example: `\X` matches `ä` encoded as U+0061 U+0300, `ä` encoded as U+00E0, `©`, etc.

Feature: Code point
 Syntax: `\uFFFF` where FFFF are 4 hexadecimal digits
 Description: Matches a specific Unicode code point.
 Example: `\u00E0` matches `ä` encoded as U+00E0 only. `\u00A9` matches `©`

Feature: Code point
 Syntax: `\u{FFFF}` where FFFF are 1 to 4 hexadecimal digits
 Description: Matches a specific Unicode code point.
 Example: `\u{E0}` matches `ä` encoded as U+00E0 only. `\u{A9}` matches `©`

Feature: Code point
 Syntax: `\x{FFFF}` where FFFF are 1 to 4 hexadecimal digits
 Description: Matches a specific Unicode code point.
 Example: `\x{E0}` matches `ä` encoded as U+00E0 only. `\x{A9}` matches `©`

Feature: Unicode category
 Syntax: `\pL` where L is a Unicode category
 Description: Matches a single Unicode code point in the specified Unicode category.
 Example: `\pL` matches `ä` encoded as U+00E0; `\pS` matches `©`

Feature: Unicode category
 Syntax: `\PL` where L is a Unicode category
 Description: Matches a single Unicode code point that is *not* in the specified Unicode category.
 Example: `\pS` matches `ä` encoded as U+00E0; `\PL` matches `©`

Feature: Unicode category
 Syntax: `\p{L}` where L is a Unicode category
 Description: Matches a single Unicode code point in the specified Unicode category.
 Example: `\p{L}` matches `ä` encoded as U+00E0; `\p{S}` matches `©`

Feature: Unicode category
 Syntax: `\p{IsL}` where L is a Unicode category
 Description: Matches a single Unicode code point in the specified Unicode category.
 Example: `\p{IsL}` matches `ä` encoded as U+00E0; `\p{IsS}` matches `©`

Feature:	Unicode category
Syntax:	<code>\p{Category}</code>
Description:	Matches a single Unicode code point in the specified Unicode category.
Example:	<code>\p{Letter}</code> matches <code>ä</code> encoded as U+00E0; <code>\p{Symbol}</code> matches <code>©</code>
Feature:	Unicode category
Syntax:	<code>\p{IsCategory}</code>
Description:	Matches a single Unicode code point in the specified Unicode category.
Example:	<code>\p{IsLetter}</code> matches <code>ä</code> encoded as U+00E0; <code>\p{IsSymbol}</code> matches <code>©</code>
Feature:	Unicode script
Syntax:	<code>\p{Script}</code>
Description:	Matches a single Unicode code point that is part of the specified Unicode script. Each Unicode code point is part of exactly one script. Scripts never contain unassigned code points.
Example:	<code>\p{Greek}</code> matches <code>Ω</code>
Feature:	Unicode script
Syntax:	<code>\p{IsScript}</code>
Description:	Matches a single Unicode code point that is part of the specified Unicode script. Each Unicode code point is part of exactly one script. Scripts never contain unassigned code points.
Example:	<code>\p{IsGreek}</code> matches <code>Ω</code>
Feature:	Unicode block
Syntax:	<code>\p{Block}</code>
Description:	Matches a single Unicode code point that is part of the specified Unicode block. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points.
Example:	<code>\p{Arrows}</code> matches any of the code points from U+2190 until U+21FF (<code>←</code> until <code>↔</code>)
Feature:	Unicode block
Syntax:	<code>\p{InBlock}</code>
Description:	Matches a single Unicode code point that is part of the specified Unicode block. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points.
Example:	<code>\p{InArrows}</code> matches any of the code points from U+2190 until U+21FF (<code>←</code> until <code>↔</code>)
Feature:	Unicode block
Syntax:	<code>\p{IsBlock}</code>
Description:	Matches a single Unicode code point that is part of the specified Unicode block. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points.
Example:	<code>\p{IsArrows}</code> matches any of the code points from U+2190 until U+21FF (<code>←</code> until <code>↔</code>)
Feature:	Negated Unicode property
Syntax:	<code>\P{Property}</code>
Description:	Matches a single Unicode code point that does <i>not</i> have the specified property (category, script, or block).
Example:	<code>\P{L}</code> matches <code>©</code>

Feature: Negated Unicode property
Syntax: `\p{^Property}`
Description: Matches a single Unicode code point that does *not* have the specified property (category, script, or block).
Example: `\p{^L}` matches ©

Feature: Unicode property
Syntax: `\P{^Property}`
Description: Matches a single Unicode code point that *does have* the specified property (category, script, or block). Double negative is taken as positive.
Example: `\P{^L}` matches q

9. Capturing Groups and Backreferences

Feature:	Capturing group
Syntax:	(regex)
Description:	Parentheses group the regex between them. They capture the text matched by the regex inside them into a numbered group that can be reused with a numbered backreference. They allow you to apply regex operators to the entire grouped regex.
Example:	<code>(abc){3}</code> matches <code>abcabcabc</code> . First group matches <code>abc</code> .
Feature:	Non-capturing group
Syntax:	(?: regex)
Description:	Non-capturing parentheses group the regex so you can apply regex operators, but do not capture anything.
Example:	<code>(?:abc){3}</code> matches <code>abcabcabc</code> . No groups.
Feature:	Backreference
Syntax:	\1 through \9
Description:	Substituted with the text matched between the 1st through 9th numbered capturing group.
Example:	<code>(abc def)=\1</code> matches <code>abc=abc</code> or <code>def=def</code> , but not <code>abc=def</code> or <code>def=abc</code> .
Feature:	Backreference
Syntax:	\10 through \99
Description:	Substituted with the text matched between the 10th through 99th numbered capturing group.
Example:	
Feature:	Backreference
Syntax:	\k<1> through \k<99>
Description:	Substituted with the text matched between the 1st through 99th numbered capturing group.
Example:	<code>(abc def)=\k<1></code> matches <code>abc=abc</code> or <code>def=def</code> , but not <code>abc=def</code> or <code>def=abc</code> .
Feature:	Backreference
Syntax:	\k'1' through \k'99'
Description:	Substituted with the text matched between the 1st through 99th numbered capturing group.
Example:	<code>(abc def)=\k'1'</code> matches <code>abc=abc</code> or <code>def=def</code> , but not <code>abc=def</code> or <code>def=abc</code> .
Feature:	Backreference
Syntax:	(?P=1) through (?P=99)
Description:	Substituted with the text matched between the 1st through 99th numbered capturing group.
Example:	<code>(abc def)=(?P=1)</code> matches <code>abc=abc</code> or <code>def=def</code> , but not <code>abc=def</code> or <code>def=abc</code> .
Feature:	Relative Backreference
Syntax:	\k<-1>, \k<-2>, etc.
Description:	Substituted with the text matched by the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the backreference.
Example:	<code>(a)(b)(c)(d)\k<-3></code> matches <code>abcbd</code> .

Feature:	Relative Backreference
Syntax:	<code>\k&apos; - 1&apos; ; , \k&apos; - 2&apos; ; , etc.</code>
Description:	Substituted with the text matched by the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the backreference.
Example:	<code>(a)(b)(c)(d)\k' - 3'</code> matches <code>abcbd</code> .
Feature:	Failed backreference
Syntax:	Any numbered backreference
Description:	Backreferences to groups that did not participate in the match attempt fail to match.
Example:	<code>(a)?\1</code> matches <code>aa</code> but fails to match <code>b</code> .
Feature:	Nested backreference
Syntax:	Any numbered backreference
Description:	Backreferences can be used inside the group they reference.
Example:	<code>(a\1?){3}</code> matches <code>aaaaaa</code> .
Feature:	Forward reference
Syntax:	Any numbered backreference
Description:	Backreferences can be used before the group they reference.
Example:	<code>(\2?(a)){3}</code> matches <code>aaaaaa</code> .

10. Named Groups and Backreferences

Feature:	Named capturing group
Syntax:	(?<name>regex)
Description:	Captures the text matched by “regex” into the group “name”. The name can contain letters and numbers but must start with a letter.
Example:	(?<x>abc){3} matches abcabcabc . The group x matches abc .
Feature:	Named capturing group
Syntax:	(?'name'regex)
Description:	Captures the text matched by “regex” into the group “name”. The name can contain letters and numbers but must start with a letter.
Example:	(?'x'abc){3} matches abcabcabc . The group x matches abc .
Feature:	Named capturing group
Syntax:	(?P<name>regex)
Description:	Captures the text matched by “regex” into the group “name”. The name can contain letters and numbers but must start with a letter.
Example:	(?P<x>abc){3} matches abcabcabc . The group x matches abc .
Feature:	Duplicate named group
Syntax:	Any named group
Description:	Two named groups can share the same name.
Example:	(?<x>a) (?<x>b) matches a or b .
Feature:	Duplicate named group
Syntax:	Any named group
Description:	Named groups that share the same name are treated as one and the same group, so there are no pitfalls when using backreferences to that name.
Example:	
Feature:	Named backreference
Syntax:	\k<name>
Description:	Substituted with the text matched by the named group “name”.
Example:	(?<x>abc)\k<x> matches abc=abc or def=def , but not abc=def or def=abc .
Feature:	Named backreference
Syntax:	\k'name'
Description:	Substituted with the text matched by the named group “name”.
Example:	(?'x'abc)\k'x' matches abc=abc or def=def , but not abc=def or def=abc .
Feature:	Named backreference
Syntax:	(?P=name)
Description:	Substituted with the text matched by the named group “name”.
Example:	(?P<x>abc)\k<x> matches abc=abc or def=def , but not abc=def or def=abc .
Feature:	Failed backreference
Syntax:	Any named backreference
Description:	Backreferences to groups that did not participate in the match attempt fail to match.
Example:	(?<x>a)?\k<x> matches aa but fails to match b .

- Feature: Nested backreference
 Syntax: Any named backreference
 Description: Backreferences can be used inside the group they reference.
 Example: `(?<x>a\k<x>?){3}` matches `aaaaaa`.
- Feature: Forward reference
 Syntax: Any named backreference
 Description: Backreferences can be used before the group they reference.
 Example: `(\k<x>?(?<x>a)){3}` matches `aaaaaa`.
- Feature: Named capturing group
 Syntax: Any named capturing group
 Description: A number is a valid name for a capturing group.
 Example: `(?<17>abc){3}` matches `abcabcabc`. The group named “17” matches `abc`.
- Feature: Named backreference
 Syntax: Any named backreference
 Description: A number is a valid name for a backreference which then points to a group with that number as its name.
 Example: `(?<17>abc\k<17>=def)` matches `abc=abc` or `def=def`, but not `abc=def` or `def=abc`.

11. Special Groups

Feature:	Comment
Syntax:	(?#comment)
Description:	Everything between (?# and) is ignored by the regex engine.
Example:	a(?#foobar)b matches ab
Feature:	Branch reset group
Syntax:	(? regex)
Description:	If the regex inside the branch reset group has multiple alternatives with capturing groups, then the capturing group numbers are the same in all the alternatives.
Example:	(x)(? (a) (bc) (def))\2 matches xaa, xbcbc, or xdefdef with the first group capturing x and the second group capturing a, bc, or def
Feature:	Atomic group
Syntax:	(?>regex)
Description:	Atomic groups prevent the regex engine from backtracking back into the group after a match has been found for the group. If the remainder of the regex fails, the engine may backtrack over the group if a quantifier or alternation makes it optional. But it will not backtrack into the group to try other permutations of the group.
Example:	a(?>bc b)c matches abcc but not abc
Feature:	Positive lookahead
Syntax:	(?=regex)
Description:	Matches at a position where the pattern inside the lookahead can be matched. Matches only the position. It does not consume any characters or expand the match. In a pattern like one(?=two)three, both two and three have to match at the position where the match of one ends.
Example:	t(?=s) matches the second t in streets.
Feature:	Negative lookahead
Syntax:	(?!regex)
Description:	Similar to positive lookahead, except that negative lookahead only succeeds if the regex inside the lookahead fails to match.
Example:	t(?!s) matches the first t in streets.
Feature:	Positive lookbehind
Syntax:	(?<=regex)
Description:	Matches at a position if the pattern inside the lookbehind can be matched ending at that position.
Example:	(?<=s)t matches the first t in streets.
Feature:	Negative lookbehind
Syntax:	(?<!=regex)
Description:	Matches at a position if the pattern inside the lookbehind cannot be matched ending at that position.
Example:	(?<!=s)t matches the second t in streets.

- Feature: Lookbehind
 Syntax: `(?<=regex|longer regex)`
 Description: Alternatives inside lookbehind can differ in length.
 Example: `(?<=is|e)t` matches the second and fourth `t` in `twisty streets`.
- Feature: Lookbehind
 Syntax: `(?<=x{n,m})`
 Description: Quantifiers with a finite maximum number of repetitions can be used inside lookbehind.
 Example: `(?<=s\w{1,7})t` matches only the fourth `t` in `twisty streets`.
- Feature: Lookbehind
 Syntax: `(?<=regex)`
 Description: The full regular expression syntax can be used inside lookbehind.
 Example: `(?<=s\w+)t` matches only the fourth `t` in `twisty streets`.
- Feature: Lookbehind
 Syntax: `(group)(?<=\1)`
 Description: Backreferences can be used inside lookbehind. Syntax prohibited in lookbehind is also prohibited in the referenced capturing group.
 Example: `(\w).+(?<=\1)` matches `twisty street` in `twisty streets`.
- Feature: Keep text out of the regex match
 Syntax: `\K`
 Description: The text matched by the part of the regex to the left of the `\K` is omitted from the overall regex match. Other than that the regex is matched normally from left to right. Capturing groups to the left of the `\K` capture as usual.
 Example: `s\Kt` matches only the first `t` in `streets`.
- Feature: Lookaround conditional
 Syntax: `(?(?=regex)then|else)` where `(?=regex)` is any valid lookaround and `then` and `else` are any valid regexes
 Description: If the lookaround succeeds, the “then” part must match for the overall regex to match. If the lookaround fails, the “else” part must match for the overall regex to match. The lookaround is zero-length. The “then” and “else” parts consume their matches like normal regexes.
 Example: `(?(?<=a)b|c)` matches the second `b` and the first `c` in `babxcac`
- Feature: Named conditional
 Syntax: `(?(name)then|else)` where `name` is the name of a capturing group and `then` and `else` are any valid regexes
 Description: If the capturing group with the given name took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
 Example: `(?<one>a)?(? (one)b|c)` matches `ab`, the first `c`, and the second `c` in `babxcac`

Feature:	Named conditional
Syntax:	<code>(?(<name>)then else)</code> where <code>name</code> is the name of a capturing group and <code>then</code> and <code>else</code> are any valid regexes
Description:	If the capturing group with the given name took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
Example:	<code>(?<one>a)?(?(<one>)b c)</code> matches <code>ab</code> , the first <code>c</code> , and the second <code>c</code> in <code>babxcac</code>
Feature:	Named conditional
Syntax:	<code>(?(&apos;name&apos;;)then else)</code> where <code>name</code> is the name of a capturing group and <code>then</code> and <code>else</code> are any valid regexes
Description:	If the capturing group with the given name took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
Example:	<code>(?'one'a)?(?('one')b c)</code> matches <code>ab</code> , the first <code>c</code> , and the second <code>c</code> in <code>babxcac</code>
Feature:	Conditional
Syntax:	<code>(?(1)then else)</code> where <code>1</code> is the number of a capturing group and <code>then</code> and <code>else</code> are any valid regexes
Description:	If the referenced capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
Example:	<code>(a)?(? (1) b c)</code> matches <code>ab</code> , the first <code>c</code> , and the second <code>c</code> in <code>babxcac</code>
Feature:	Relative conditional
Syntax:	<code>(?(-1)then else)</code> where <code>-1</code> is a negative integer and <code>then</code> and <code>else</code> are any valid regexes
Description:	Conditional that tests the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting immediately before the conditional. If the referenced capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
Example:	<code>(a)?(? (-1) b c)</code> matches <code>ab</code> , the first <code>c</code> , and the second <code>c</code> in <code>babxcac</code>
Feature:	Forward conditional
Syntax:	<code>(?(+1)then else)</code> where <code>+1</code> is a positive integer and <code>then</code> and <code>else</code> are any valid regexes
Description:	Conditional that tests the capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the “then” part of conditional. If the referenced capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group did not take part in the match thus far, the “else” part must match for the overall regex to match.
Example:	<code>((?(+1)b c)(d)?){2}</code> matches <code>cc</code> and <code>cdb</code> in <code>bdbdcccxcdb</code>

12. Mode Modifiers

Mode modifier syntax consists of two elements that differ among regex flavors. Parentheses and a question mark are used to add the modifier to the regex. Depending on its position in the regex and the regex flavor it may affect the whole regex or part of it. If a flavor supports at least one modifier syntax, then it will also support one or more letters that can be used inside the modifier to toggle specific modes. If it doesn't, "n/a" is indicated for all letters for that flavors.

If a flavor supports mode modifiers but does not support a particular letter, it will be indicated as "no". That does not mean that the flavor doesn't have this mode at all. The flavor may still have the mode, but no option to turn it off. Modes are also not necessarily off by default. For example, in most regex flavors, `^` and `$` match at the start and end of the string only by default. But the Just Great Software applications and Ruby, they match at the start and end of each line by default. In the JGsoft applications, you can turn off this mode with `(?-m)` while in Ruby you cannot turn off this mode at all. `(?-m)` affects the dot rather than the anchors in Ruby.

The table below only indicates whether each flavor supports a particular letter to toggle a particular mode. It does not indicate the defaults.

Feature:	Mode modifier
Syntax:	<code>(?letters)</code> at the start of the regex
Description:	A mode modifier at the start of the regex affects the whole regex and overrides any options set outside the regex.
Example:	<code>(?i)a</code> matches <code>a</code> and <code>A</code> .
Feature:	Mode modifier
Syntax:	<code>(?letters)</code> in the middle of the regex
Description:	A mode modifier in the middle of the regex affects only the part of the regex to the right of the modifier. If the modifier is used inside a group, it only affects the part of the regex inside that group to the right of the modifier. If the regex or group uses alternation, all alternatives to the right of the modifier are affected.
Example:	<code>te(?i)st</code> matches <code>test</code> and <code>teST</code> but not <code>TEst</code> or <code>TEST</code> .
Feature:	Modifier group
Syntax:	<code>(?letters:regex)</code>
Description:	Non-capturing group with modifiers that affect only the part of the regex inside the group.
Example:	<code>te(?i:st)</code> matches <code>test</code> and <code>teST</code> but not <code>TEst</code> or <code>TEST</code> .
Feature:	Negative modifier
Syntax:	<code>(?on-off)</code> and <code>(?on-off:regex)</code>
Description:	Modifier letters (if any) before the hyphen are turned on, while modifier letters after the hyphen are turned off.
Example:	<code>(?i)te(?-i)st</code> matches <code>test</code> and <code>TEst</code> but not <code>teST</code> or <code>TEST</code> .
Feature:	Case insensitive
Syntax:	<code>(?i)</code>
Description:	Turn on case insensitivity.
Example:	<code>(?i)a</code> matches <code>a</code> and <code>A</code> .

Feature:	Free-spacing
Syntax:	(?x)
Description:	Turn on free-spacing mode to ignore whitespace between regex tokens and allow # comments.
Example:	(?x) a#b matches a
Feature:	Single-line
Syntax:	(?s)
Description:	Make the dot match all characters including line break characters.
Example:	(?s) .* matches ab\n\ndef in ab\n\ndef
Feature:	Multi-line
Syntax:	(?m)
Description:	Make ^ and \$ match at the start and end of each line.
Example:	(?m) ^. matches a and d in ab\n\ndef
Feature:	Explicit capture
Syntax:	(?n)
Description:	Plain parentheses are non-capturing groups instead of numbered capturing groups. Only named capturing groups actually capture.
Example:	(?n) (a b)c is the same as (? :a b)c

13. Balancing Groups, Recursion, and Subroutines

- Feature:** Balancing group
Syntax: (?<capture-subtract>regex) where “capture” and “subtract” are group names and “regex” is any regex
Description: The name “subtract” must be used as the name of a capturing group elsewhere in the regex. If this group has captured matches that haven’t been subtracted yet, then the balancing group subtracts one capture from “subtract”, attempts to match “regex”, and stores its match into the group “capture”. If “capture” is omitted, the same happens without storing the match. If “regex” is omitted, the balancing group succeeds without advancing through the string. If the group “subtract” has no matches to subtract, then the balancing group fails to match, regardless of whether “regex” is specified or not.
Example: `^(?<1>\w)+\w?`
`(\k<1>(?<-1>))+`
`(?(1)(?!))$` matches any palindrome word
- Feature:** Balancing group
Syntax: (?&capture-subtract®ex) where “capture” and “subtract” are group names and “regex” is any regex
Description: The name “subtract” must be used as the name of a capturing group elsewhere in the regex. If this group has captured matches that haven’t been subtracted yet, then the balancing group subtracts one capture from “subtract”, attempts to match “regex”, and stores its match into the group “capture”. If “capture” is omitted, the same happens without storing the match. If “regex” is omitted, the balancing group succeeds without advancing through the string. If the group “subtract” has no matches to subtract, then the balancing group fails to match, regardless of whether “regex” is specified or not.
Example: `^(?'1'\w)+\w?`
`(\k'1'(?'-1'))+`
`(?(1)(?!))$` matches any palindrome word
- Feature:** Recursion
Syntax: (?R)
Description: Recursion of the entire regular expression.
Example: `a(?R)?z` matches `az`, `aazz`, `aaazzz`, etc.
- Feature:** Recursion
Syntax: (?0)
Description: Recursion of the entire regular expression.
Example: `a(?0)?z` matches `az`, `aazz`, `aaazzz`, etc.
- Feature:** Recursion
Syntax: \g<0>
Description: Recursion of the entire regular expression.
Example: `a\g<0>?z` matches `az`, `aazz`, `aaazzz`, etc.
- Feature:** Recursion
Syntax: \g'0'
Description: Recursion of the entire regular expression.
Example: `a\g'0'?z` matches `az`, `aazz`, `aaazzz`, etc.

Feature:	Subroutine call
Syntax:	(?1) where 1 is the number of a capturing group
Description:	Recursion of a capturing group or subroutine call to a capturing group.
Example:	<code>a(b(?1)?y)z</code> matches <code>abyz</code> , <code>abbyyz</code> , <code>abbbyyyz</code> , etc.
Feature:	Subroutine call
Syntax:	<code>\g<1></code> where 1 is the number of a capturing group
Description:	Recursion of a capturing group or subroutine call to a capturing group.
Example:	<code>a(b\g<1>?y)z</code> matches <code>abyz</code> , <code>abbyyz</code> , <code>abbbyyyz</code> , etc.
Feature:	Subroutine call
Syntax:	<code>\g&apos;1&apos;</code> where 1 is the number of a capturing group
Description:	Recursion of a capturing group or subroutine call to a capturing group.
Example:	<code>a(b\g'1'?y)z</code> matches <code>abyz</code> , <code>abbyyz</code> , <code>abbbyyyz</code> , etc.
Feature:	Relative subroutine call
Syntax:	(?-1) where -1 is a negative integer
Description:	Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the subroutine call.
Example:	<code>a(b(?-1)?y)z</code> matches <code>abyz</code> , <code>abbyyz</code> , <code>abbbyyyz</code> , etc.
Feature:	Relative subroutine call
Syntax:	<code>\g<-1></code> where -1 is a negative integer
Description:	Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the subroutine call.
Example:	<code>a(b\g<-1>?y)z</code> matches <code>abyz</code> , <code>abbyyz</code> , <code>abbbyyyz</code> , etc.
Feature:	Relative subroutine call
Syntax:	<code>\g&apos;-1&apos;</code> where -1 is a negative integer
Description:	Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from right to left starting at the subroutine call.
Example:	<code>a(b\g'-1'?y)z</code> matches <code>abyz</code> , <code>abbyyz</code> , <code>abbbyyyz</code> , etc.
Feature:	Forward subroutine call
Syntax:	(?+1) where +1 is a positive integer
Description:	Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the subroutine call.
Example:	<code>(?+1)x([ab])</code> matches <code>axa</code> , <code>axb</code> , <code>bxa</code> , and <code>bx b</code>
Feature:	Forward subroutine call
Syntax:	<code>\g<+1></code> where +1 is a positive integer
Description:	Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the subroutine call.
Example:	<code>\g<+1>x([ab])</code> matches <code>axa</code> , <code>axb</code> , <code>bxa</code> , and <code>bx b</code>

- Feature: Forward subroutine call
 Syntax: `\g'+1'`; where +1 is a positive integer
 Description: Recursion of or subroutine call to a capturing group that can be found by counting as many opening parentheses of named or numbered capturing groups as specified by the number from left to right starting at the subroutine call.
 Example: `\g'+1'x([ab])` matches `axa`, `axb`, `bxa`, and `bxb`
- Feature: Named subroutine call
 Syntax: `(?&name)` where “name” is the name of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(?<x>b(?&x)?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
- Feature: Named subroutine call
 Syntax: `(?P>name)` where “name” is the name of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(?P<x>b(?P>x)?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
- Feature: Named subroutine call
 Syntax: `\g<name>` where “name” is the name of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(?<x>b\g<x>?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
- Feature: Named subroutine call
 Syntax: `\g'name'`; where “name” is the name of a capturing group
 Description: Recursion of a capturing group or subroutine call to a capturing group.
 Example: `a(?'x'b\g'x'?y)z` matches `abyz`, `abbyyz`, `abbbyyyz`, etc.
- Feature: Subroutine definitions
 Syntax: `(?(DEFINE) regex)` where “regex” is any regex
 Description: The DEFINE group does not take part in the matching process. Subroutine calls can be made to capturing groups inside the DEFINE group.
 Example: `?(DEFINE)([ab])`
`x(?1)y(?1)z` matches `xayaz`, `xaybz`, `xbyaz`, and `xbybz`
- Feature: Subroutine calls capture
 Syntax: Subroutine call using Ruby-style `\g` syntax
 Description: A subroutine call to a capturing group makes that capturing group store the text matched during the subroutine call.
 Example: When `([ab])\g'1'` matches `ab` the first capturing group holds `b` after the match.
- Feature: Recursion reverts capturing groups
 Syntax: Recursion or subroutine call using syntax other than `\g`
 Description: When the regex engine exits from recursion or a subroutine call, it reverts all capturing groups to the text they had matched prior to entering the recursion or subroutine call.
 Example: When `(a)<([bc])\1(?:)` matches `abaca` the third group stores `b` after the match

- Feature: Recursion does not isolate or revert capturing groups
- Syntax: Recursion or subroutine call using Ruby-style `\g` syntax
- Description: Capturing groups are not given any special treatment by recursion and subroutine calls, except perhaps that subroutine calls capture. Backreferences always see the text most recently matched by each capturing group, regardless of whether they are inside the same level of recursion or not.
- Example: When `(a)(\g'2')[bc]\1` matches `abaca` the third group stores `c` after the match
-
- Feature: Recursion is atomic
- Syntax: Recursion or subroutine call using `(?P>...)`
- Description: Recursion and subroutine calls are atomic. Once the regex engine exits from them, it will not backtrack into it to try different permutations of the recursion or subroutine call.
- Example: `(a+)(?P>1)(?P>1)` can never match anything because the first `(?P>1)` matches all remaining a's and the regex engine won't backtrack into the first `(?P>1)` when the second one fails

14. Replacement String Characters

Feature: Backslash
 Syntax: A backslash that does not form a token
 Description: A backslash that is not part of a replacement string token is a literal backslash.
 Example: Replacing with `\!` yields `\!`

Feature: Backslash
 Syntax: Trailing backslash
 Description: A backslash at the end of the replacement string is a literal backslash.
 Example: Replacing with `\` yields `\`

Feature: Backslash
 Syntax: `\\`
 Description: A backslash escapes itself.
 Example: Replacing with `\\` yields `\`

Feature: Dollar
 Syntax: A dollar that does not form a token
 Description: A dollar sign that does not form a replacement string token is a literal dollar sign.
 Example: Replacing with `$!` yields `$!`

Feature: Dollar
 Syntax: Trailing dollar
 Description: A dollar sign at the end of the replacement string is a literal dollar sign.
 Example: Replacing with `$` yields `$`

Feature: Dollar
 Syntax: `$$`
 Description: A dollar sign escapes itself.
 Example: Replacing with `$$` yields `$`

Feature: Dollar
 Syntax: `\$`
 Description: A backslash escapes a dollar sign.
 Example: Replacing with `\$` yields `$`

Feature: Hexadecimal escape
 Syntax: `\xFF` where FF are 2 hexadecimal digits
 Description: Inserts the character at the specified position in the code page
 Example: `\xA9` inserts `©` when using the Latin-1 code page

Feature: Unicode escape
 Syntax: `\uFFFF` where FFFF are 4 hexadecimal digits
 Description: Inserts a specific Unicode code point.
 Example: `\u00E0` inserts `à` encoded as U+00E0 only. `\u00A9` inserts `©`

Feature:	Unicode escape
Syntax:	<code>\u{FFFF}</code> where FFFF are 1 to 4 hexadecimal digits
Description:	Inserts a specific Unicode code point.
Example:	<code>\u{E0}</code> inserts <code>à</code> encoded as U+00E0 only. <code>\u{A9}</code> inserts <code>©</code>
Feature:	Unicode escape
Syntax:	<code>\x{FFFF}</code> where FFFF are 1 to 4 hexadecimal digits
Description:	Inserts a specific Unicode code point.
Example:	<code>\x{E0}</code> inserts <code>à</code> encoded as U+00E0 only. <code>\x{A9}</code> inserts <code>©</code>
Feature:	Character escape
Syntax:	<code>\n</code> , <code>\r</code> and <code>\t</code>
Description:	Insert an LF character, CR character and a tab character respectively
Example:	<code>\r\n</code> inserts a Windows CRLF line break
Feature:	Octal escape
Syntax:	<code>\o{7777}</code> where 7777 is any octal number
Description:	Inserts the character at the specified position in the active code page
Example:	<code>\o{20254}</code> inserts <code>€</code> when using Unicode

15. Matched Text and Backreferences in Replacement Strings

Feature: Whole match

Syntax: `\&`

Description: Insert the whole regex match.

Example: Replacing `\d+` with `[\&]` in `1a2b` yields `[1]a[2]b`

Feature: Whole match

Syntax: `$&`

Description: Insert the whole regex match.

Example: Replacing `\d+` with `[$&]` in `1a2b` yields `[1]a[2]b`

Feature: Whole match

Syntax: `\0`

Description: Insert the whole regex match.

Example: Replacing `\d+` with `[\0]` in `1a2b` yields `[1]a[2]b`

Feature: Whole match

Syntax: `$0`

Description: Insert the whole regex match.

Example: Replacing `\d+` with `[$0]` in `1a2b` yields `[1]a[2]b`

Feature: Whole match

Syntax: `\g<0>`

Description: Insert the whole regex match.

Example: Replacing `\d+` with `[\g<0>]` in `1a2b` yields `[1]a[2]b`

Feature: Backreference

Syntax: `\1` through `\9`

Description: Insert the text matched by one of the first 9 capturing groups.

Example: Replacing `(a)(b)(c)` with `\3\3\1` in `abc` yields `cca`

Feature: Backreference

Syntax: `\10` through `\99`

Description: Insert the text matched by capturing groups 10 through 99.

Example:

Feature: Backreference and literal

Syntax: `\10` through `\99`

Description: When there are fewer capturing groups than the 2-digit number, treat this as a single-digit backreference followed by a literal number instead of as an invalid backreference.

Example: Replacing `(a)(b)(c)` with `\39\38\17` in `abc` yields `c9c8a7`

Feature: Backreference

Syntax: `$1` through `$9`

Description: Insert the text matched by one of the first 9 capturing groups.

Example: Replacing `(a)(b)(c)` with `$3$3$1` in `abc` yields `cca`

Feature:	Backreference
Syntax:	<code>\$10</code> through <code>\$99</code>
Description:	Insert the text matched by capturing groups 10 through 99.
Example:	
Feature:	Backreference and literal
Syntax:	<code>\$10</code> through <code>\$99</code>
Description:	When there are fewer capturing groups than the 2-digit number, treat this as a single-digit backreference followed by a literal number instead of as an invalid backreference.
Example:	Replacing <code>(a)(b)(c)</code> with <code>\$39\$38\$17</code> in <code>abc</code> yields <code>c9c8a7</code>
Feature:	Backreference
Syntax:	<code>\${1}</code> through <code>\${99}</code>
Description:	Insert the text matched by capturing groups 1 through 99.
Example:	Replacing <code>(a)(b)(c)</code> with <code>\${3}\${3}\${1}</code> in <code>abc</code> yields <code>cca</code>
Feature:	Backreference
Syntax:	<code>\g<1></code> through <code>\g<99></code>
Description:	Insert the text matched by capturing groups 1 through 99.
Example:	Replacing <code>(a)(b)(c)</code> with <code>\g<3>\g<3>\g<1></code> in <code>abc</code> yields <code>cca</code>
Feature:	Named backreference
Syntax:	<code>\${name}</code>
Description:	Insert the text matched by the named capturing group “name”.
Example:	Replacing <code>(?'one' a)(?'two' b)</code> with <code>\${two}\${one}</code> in <code>ab</code> yields <code>ba</code>
Feature:	Named backreference
Syntax:	<code>\g<name></code>
Description:	Insert the text matched by the named capturing group “name”.
Example:	Replacing <code>(?P<one>a)(?P<two>b)</code> with <code>\g<two>\g<one></code> in <code>ab</code> yields <code>ba</code>
Feature:	Backreference to non-participating group
Syntax:	Any supported backreference syntax
Description:	A backreference to a non-participating capturing group is replaced with the empty string.
Example:	
Feature:	Last backreference
Syntax:	<code>\+</code>
Description:	Insert the text matched by the highest-numbered capturing group that actually participated in the match.
Example:	Replacing <code>(a)(z)?</code> with <code>[\+]</code> in <code>ab</code> yields <code>[a]b</code>
Feature:	Last backreference
Syntax:	<code>\$+</code>
Description:	Insert the text matched by the highest-numbered capturing group that actually participated in the match.
Example:	Replacing <code>(a)(z)?</code> with <code>[\$+]</code> in <code>ab</code> yields <code>[a]b</code>

16. Context and Case Conversion in Replacement Strings

Feature: Match Context

Syntax: `\`` (backslash backtick)

Description: Insert the part of the subject string to the left of the regex match

Example: Replacing `b` with `\`` in `abc` yields `aac`

Feature: Match Context

Syntax: `$`` (dollar backtick)

Description: Insert the part of the subject string to the left of the regex match

Example: Replacing `b` with `$`` in `abc` yields `aac`

Feature: Match Context

Syntax: `\'` (backslash quote)

Description: Insert the part of the subject string to the right of the regex match

Example: Replacing `b` with `\'` in `abc` yields `acc`

Feature: Match Context

Syntax: `$'` (dollar quote)

Description: Insert the part of the subject string to the right of the regex match

Example: Replacing `b` with `$'` in `abc` yields `acc`

Feature: Match Context

Syntax: `$_`

Description: Insert the whole subject string

Example: Replacing `b` with `$_` in `abc` yields `aabcc`

Feature: Case Conversion

Syntax: `\U0` and `\U1` through `\U99`

Description: Insert the whole regex match or the 1st through 99th backreference with all letters in the matched text converted to uppercase.

Example: Replacing `.+` with `\U0` in `HeLL0 WoRLD` yields `HELLO WORLD`

Feature: Case Conversion

Syntax: `\L0` and `\L1` through `\L99`

Description: Insert the whole regex match or the 1st through 99th backreference with all letters in the matched text converted to lowercase.

Example: Replacing `.+` with `\L0` in `HeLL0 WoRLD` yields `hello world`

Feature: Case Conversion

Syntax: `\F0` and `\F1` through `\F99`

Description: Insert the whole regex match or the 1st through 99th backreference with the first letter in the matched text converted to uppercase and the remaining letters converted to lowercase.

Example: Replacing `.+` with `\F0` in `HeLL0 WoRLD` yields `Hello world`

Feature: Case Conversion
Syntax: `\I0` and `\I1` through `\I99`
Description: Insert the whole regex match or the 1st through 99th backreference with the first letter of each word in the matched text converted to uppercase and the remaining letters converted to lowercase.
Example: Replacing `.+` with `\I0` in `HeLL0 WoRLD` yields `Hello World`

17. Conditionals in Replacement Strings

Feature:	Conditional
Syntax:	<code>(?1yes:no)</code> through <code>(?99yes:no)</code>
Description:	Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
Example:	Replacing all matches of <code>(y)?\n</code> in <code>yyn!</code> with <code>(?1yes:no)</code> yields <code>yesyesno!</code>
Feature:	Conditional
Syntax:	<code>(?10yes:no)</code> through <code>(?99yes:no)</code>
Description:	When there are fewer capturing groups than the 2-digit number, treat this as a single-digit conditional with the “yes” part starting with a literal number instead of as an invalid conditional.
Example:	Replacing all matches of <code>(y)?\n</code> in <code>yyn!</code> with <code>(?19yes:no)</code> yields <code>9yes9yesno!</code>
Feature:	Conditional
Syntax:	<code>(?{1}yes:no)</code> through <code>(?{99}yes:no)</code>
Description:	Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
Example:	Replacing all matches of <code>(y)?\n</code> in <code>yyn!</code> with <code>(?{1}yes:no)</code> yields <code>yesyesno!</code>
Feature:	Conditional
Syntax:	<code>\${1:+yes:no}</code> through <code>\${99:+yes:no}</code>
Description:	Conditional referencing a numbered capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
Example:	Replacing all matches of <code>(y)?\n</code> in <code>yyn!</code> with <code>\${1:+yes:no}</code> yields <code>yesyesno!</code>
Feature:	Conditional
Syntax:	<code>\${1:-no}</code> through <code>\${99:-no}</code>
Description:	Conditional referencing a numbered capturing group. Inserts the text captured by the group if it participated or the contents of the conditional if it didn’t.
Example:	Replacing all matches of <code>(y)?\n</code> in <code>yyn!</code> with <code>\${1:-no}</code> yields <code>yynno!</code>
Feature:	Conditional
Syntax:	<code>(?{name}yes:no)</code>
Description:	Conditional referencing a named capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
Example:	Replacing all matches of <code>(?'one'y)?\n</code> in <code>yyn!</code> with <code>(?{one}yes:no)</code> yields <code>yesyesno!</code>
Feature:	Conditional
Syntax:	<code>\${name:+yes:no}</code>
Description:	Conditional referencing a named capturing group. Inserts the “yes” part if the group participated or the “no” part if it didn’t.
Example:	Replacing all matches of <code>(?'one'y)?\n</code> in <code>yyn!</code> with <code>\${one:+yes:no}</code> yields <code>yesyesno!</code>
Feature:	Conditional
Syntax:	<code>\${name:-no}</code>
Description:	Conditional referencing a named capturing group. Inserts the text captured by the group if it participated or the contents of the conditional if it didn’t.
Example:	Replacing all matches of <code>(?'one'y)?\n</code> in <code>yyn!</code> with <code>\${one:-no}</code> yields <code>yynno!</code>

Index

- \$. *see* dollar sign, *see* dollar sign
- \ . *see* backslash
- ^ . *see* caret
- .. *see* dot
- | . *see* vertical bar
- ? . *see* question mark
- * . *see* star
- + . *see* plus
- (. *see* parenthesis
-) . *see* parenthesis
- [. *see* square brackets
-] . *see* square brackets
- { . *see* curly braces
- } . *see* curly braces
- \t . *see* tab
- \r . *see* carriage return
- \n . *see* line feed
- \a . *see* bell
- \e . *see* escape
- \f . *see* form feed
- \d . *see* digit
- \D . *see* digit
- \s . *see* whitespace
- \S . *see* whitespace
- \w . *see* word character
- \W . *see* word character
- \c . *see* control characters *or* XML names
- \C . *see* control characters *or* XML names
- \i . *see* XML names
- \I . *see* XML names
- \b . *see* word boundary
- { . *see* curly braces
- \1 . *see* backreference
- \K . *see* keep
- \G . *see* previous match
- \ . *see* backslash
- \t . *see* tab
- \n . *see* line feed
- #65535 and #xffff => characters, 146
- .bak, 232
- .epp files, 33
- .jgcses, 183, 193, 195, 213
- .jgfn, 202, 213
- .tcl files, 259
- 32-bit, 303
- 64-bit, 303
- 8859, 127
- accents, 291
- active line highlight, 215
- adapt, 60
- adapt case, 60
- add active file, 38
- add outside files, 39
- add to project, 37
- add to project unopened, 38
- adjust case, 132, 176
- adjust quotes, 133, 176
- all files, 61
- all projects, 61
- Allman, 196
- alnum, 394
- alpha, 394
- alphabetic sort, 115
- alphanumeric sort, 115, 116
- alternation, 333
- anchor, 328, 363, 398
- any character, 326
- apostrophe, 135
- append, 87
- ASCII, 318, 394
- ascii only, 263
- assertion, 363
- associations, 169
- asterisk. *see* star
- atomic
 - recursion, 391
- attachments, 16
- auto adjust case, 132
- auto adjust quotes, 134
- auto break, 151, 196
- auto indent, 150, 167, 177, 196
- auto match brackets, 28
- auto trim trailing whitespace, 119
- automatic reload, 228
- automatic save, 232
- autosave, 232
- \b . *see* word boundary
- back in edited files, 73
- back in editing positions, 71
- backreference
 - in a character class, 342
 - number, 340
 - recursion, 388

- repetition, 451
- backreferences, 65
- backslash, 316, 401
 - in a character class, 321
- backtick, 135
- backtracking, 337, 437
 - recursion, 391
- backup, 17
- backup files, 232, 287
- balanced constructs, 373, 377
- balancing groups, 373
- begin file, 328
- begin line, 328
- begin selection, 82
- begin string, 328
- bell, 318
- between matching brackets, 83
- bitmapped fonts, 159
- blank, 394
- blank lines, 142, 143
- block, 76, 77, 78, 79, 82, 83, 84, 85, 86, 87, 88, 90, 167
- bom, 170
- BOM, 243
- bookmark, 89, 90
- Borland colors, 186
- braces. *see* curly braces
- bracket matching, 27, 28
- brackets, 71, 196, *see* square brackets *or* parenthesis
- breaking long lines, 167
- browser, 268
- byte offset, 70, 309
- byte order marker, 170
- Byte Order Marker, 243
- byte value editor, 264
- \c. *see* control characters *or* XML names
- \C. *see* control characters *or* XML names
- capital letters, 133
- caps lock, 133
- capturing group, 339
 - recursion, 378, 384
- capturing group subtraction, 373
- caret, 316, 328
 - in a character class, 321
- carriage return, 130, 318
- case, 132, 133
- case conversion, 66
- case sensitive, 49, 60
- catastrophic backtracking, 437
- character class, 321
 - intersection, 324
 - negated, 321
 - negated shorthand, 325
 - repeating, 322
 - shorthand, 325
 - special characters, 321
 - subtract, 323
 - XML names, 325
- character map, 261
- character range, 321
- character set. *see* character class
- character width, 161
- characters, 316, 401
 - ASCII, 318
 - categories, 352
 - digit, 325
 - in a character class, 321
 - invisible, 318
 - metacharacters, 316
 - non-printable, 318
 - non-word, 325, 331
 - special, 316, 401
 - Unicode, 318, 351
 - whitespace, 325
 - word, 325, 331
- characters => #65535, 145
- characters => #xffff, 146
- characters => htmlchar, 147
- characters => \uFFFF, 144
- check for new version, 296
- check spelling, 112, 113, 114
- choice, 333
- class, 321
- clip collection, 259
- clip editor, 260
- clipboard, 21, 22, 23, 24, 25, 26, 93, 259
- close, 18
- close all, 19
- close all but current, 19
- close all but current project, 44
- close all files, 40
- close outside files, 40
- close panels, 268
- close project, 44
- closed files, 36, 62
- closing bracket, 353
- closing files, 232
- closing quote, 353
- cntrl, 394
- code folding, 202
- code page, 127, 170, 453
- code point, 351

- color palette, 11
- colors, 183, 186
- column, 78, 79, 154, 174
- column numbers, 175
- combining character, 352
- combining mark*, 351
- combining multiple regexes, 333, 433
- command line, 96
- command line parameters, 309
- command line placeholders, 98
- command line utilities, 95
- comment, 434
- comment out, 84, 85
- comments, 84, 85, 86, 349
- commonly used files, 8, 10
- compare changes, 124
- compare files, 119, 122, 124
- complex scripts, 157
- conditional, 370
 - replacement, 408
- conditions
 - many in one regex, 366
- configure, 95, 166
- consolidate blank lines, 118
- contents, 3
- context menu, 305
- continue
 - from previous match, 398
- control characters, 353
- convert, 127, 130, 132, 133, 134, 135, 137, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148
- copy, 17, 22
- copy append, 23
- copy as HTML, 23
- copy as rich text, 24
- copy as RTF, 24
- copy matches, 58
- Copy Path to Clipboard, 239
- copy visible lines, 93
- count, 58, 125, 126, 154
- create portable installation, 303
- CRLF, 130
- CRLF pair, 318
- cross. *see* plus
- CSCS, 193, 195, 213
- CSV, 177
- Ctrl+Enter, 30, 130
- curly braces, 316, 336
- currency sign, 352
- cursor, 219
- custom layouts, 269
- custom syntax coloring schemes, 183, 193, 195, 213
- customize, 305
- cut, 21
- cut append, 21
- cut matches, 58
- \d. *see* digit
- \D. *see* digit
- dark theme, 269
- dash, 353
- data, 315
- date, 28, 215, 425, 426
 - to text, 426
 - validate, 425
- date format, 215
- default text editor, 169
- delete, 18, 30
- delete blank lines, 118
- delete duplicate lines, 116
- delete folded lines, 94
- delete line, 31
- delete project, 44
- denial of service, 447
- desktop, 254
- diacritics, 291
- diff, 119
- difference, 119
- digit, 325, 353, 394
- distance, 436
- dollar sign, 316, 328, 401
- DOS, 127, 170
- dot, 316, 326
 - misuse, 438
- dot matches line breaks, 60
- double click, 222
- double spacing, 142, 143
- download from FTP, 279
- downloading custom syntax coloring schemes, 193
- drag tabs, 74, 75
- drive letter, 304
- duplicate items, 432
- duplicate line, 30
- duplicate lines, 116, 432
- duplicate selection, 76
- eager, 320, 333
- EBCDIC, 127, 170
- edit, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 31, 70
- edit clip, 260
- edit columns, 79
- editor, 215
- EditPad web site, 296

- elastic tab stops, 177
- else, 370
 - replacement, 408
- email, 15, 251
- email address, 418
- email as attachment, 16
- enclosing mark, 352
- encoding, 127, 170
- end file, 328
- End key, 221
- end line, 328
- end of line, 318, 402
- end selection, 82
- end string, 328
- endless recursion, 381
- engine, 319
- Enter, 130
- environment variables, 98
- epp files, 33
- escape, 316, 318
 - in a character class, 321
- EUC, 127
- euro, 453
- example
 - combining multiple regexes, 433
 - date, 425, 426
 - duplicate items, 432
 - duplicate lines, 432
 - exponential number, 417, 434
 - floating point number, 417
 - HTML tags, 413
 - integer number, 434
 - keywords, 435
 - multi-line comment, 434
 - not meeting a condition, 430
 - number, 434
 - numeric ranges, 415
 - prepend lines, 396
 - programming languages, 433
 - quoted string, 327, 434
 - reserved words, 435
 - scientific number, 417, 434
 - single-line comment, 434
 - source code, 433
 - trimming whitespace, 413
 - whole line, 430
- exit, 19
- expand selection, 83
- explorer panel, 275
- export file listing, 43
- export preferences, 164
- export to HTML or RTF, 18, 87
- external commands, 95
- extra, 112, 113, 115, 116, 118, 119, 122, 124, 125, 126
- fallback fonts, 160
- favorites, 8, 33, 63
- feedback, 302
- feeds, 301
- file, 4, 6, 8, 10, 11, 15, 16, 17, 18, 19, 232
- file associations, 169, 304
- file history, 287
- file locks, 228
- file mask, 167, 278
- file navigator, 202, 290
- file size, 228
- file tabs, 237, 239
- file types, 149, 166, 169
- files, 36
- files matching a file mask, 278
- files matching a regex, 278
- files panel, 271
- filter, 278
- find, 46, 47, 48, 49, 58, 60, 61, 62, 63
- find first, 50
- find last, 52
- find next, 50
- find on disk, 55, 57
- find previous, 51
- fixed line lengths, 141
- flash card, 303
- floating point number, 417
- flow paragraphs, 142, 149
- focus editor, 268
- fold, 91, 92, 93, 94, 202
- fold all, 93
- fold lines, 57, 91
- fold unselected, 92
- folders, 34, 228
- font, 154, 156, 159, 164, 174, 241
- font size, 216
- form feed, 130, 318
- format paragraphs, 142
- formatted text, 18, 23, 24, 87
- forum, 296
- forward in edited files, 73
- forward in editing positions, 72
- free-spacing, 60, 349
- FTP panel, 279
- FTPS, 279
- full stop. *see* dot
- go, 70, 71, 72, 73, 74, 75
- go to, 70, 84, 89
- go to matching bracket, 71

- go to next bookmark, 88
- go to next fold, 92
- go to previous bookmark, 88
- go to previous fold, 92
- go to unmatched bracket, 71
- graph, 394
- grapheme, 351
- grave accent, 135
- greedy*, 335, 336
- green EditPad icon, 169
- group, 339
 - capturing, 339
 - in a character class, 342
 - named, 345
 - nested, 437
 - repetition, 451
- guillemet, 135
- gutter, 186
- hanging indent, 149
- hard returns, 141
- help, 295, 296, 302, 303
- hexadecimal mode, 167, 263, 264
- hibernation, 235
- hide lines, 91
- highlight active line, 215
- highlight all, 53
- highlight misspelled words, 113
- highlighting, 183
- history, 63, 287
- hollow rectangles, 127
- home key, 215
- Home key, 221
- Horstmann, 196
- HTML, 18, 23, 87
- HTML tags, 413
- htmlchar => characters, 147
- hue, 189
- huge files, 228
- hyphen, 353
 - in a character class, 321
- \i. *see* XML names
- \I. *see* XML names
- icon for text files, 169
- icon size, 305
- if-then-else, 370
 - replacement, 408
- import file listing, 41
- import preferences, 165
- incremental search, 54
- indent, 77, 149, 150, 167
- indent wrapped lines, 149, 174

- indentation size, 177
- indentation spaces => tabs, 144
- infinite recursion, 381
- initial caps, 133
- Insert, 243
- insert date and time, 28
- insert file, 86
- insert page break, 30
- install onto removable drive, 303
- installing custom syntax coloring schemes, 193
- instances, 267
- instant find next, 48
- instant find previous, 48
- instant highlight, 46
- instant incremental search, 47
- instant replace, 48
- instant search options, 49
- integer number, 434
- internet explorer, 268
- intersect character classes, 324
- invert case, 133
- inverted line by line, 63
- invisible characters, 318
- ISO-8859, 127
- jgcs, 183, 193, 195, 213
- jgfs, 202, 213
- joint scrolling, 266
- K&R, 196
- keep, 368
- keep indent, 150, 177
- keep instant macro, 111
- keyboard, 247, 268, 291
- keystroke macros, 107, 109, 110, 111
- keywords, 435
- KOI8, 127
- lazy, 337
 - better alternative, 337
- leading whitespace, 118
- leftmost match, 319
- left-to-right, 156, 157, 164
- letter, 352, *see* word character
- light theme, 269
- line, 328
 - begin, 328
 - duplicate, 432
 - end, 328
 - not meeting a condition, 430
 - prepend, 396
- line break, 318, 326, 402
- line break indicators, 153
- line break style, 130, 170, 243
- line breaks, 60, 142, 176, 183

- line breaks => wrapping, 142
- line by line, 63
- line feed, 130, 318, 402
- line height, 161
- line highlight, 215
- line length, 175
- line lengths, 141
- line number, 70, 309
- line numbers, 167, 174, 175
- line separator, 130, 352
- line spacing, 142, 143
- line terminator, 318, 402
- lines, 30, 31, 70, 78, 91, 94, 116, 125, 154
- Linux, 170
- list all matches, 54, 57
- literal characers, 316, 401
- live spell check, 113
- live spelling, 183
- lock toolbars, 305
- locking files, 228
- log viewer, 6
- lookahead, 363
- lookaround, 363
 - many conditions in one regex, 366
- lookbehind, 364
 - limitations, 365
- loop automatically, 49, 62
- lower, 394
- lowercase, 66, 133
- lowercase letter, 352
- luminance, 189
- Macintosh, 170
- macros, 107, 109, 110, 111
- macros menu, 107
- mail, 15, 251
- mail as attachment, 16
- managed project, 36
- many conditions in one regex, 366
- margins, 167
- mark, 89, 90, 352
- match, 315
- match placeholders, 65
- matching bracket, 27, 28
- matching brackets, 71, 196
- mathematical symbol, 352
- menu, 305
- menu font, 305
- merge, 119
- merging files, 119
- metacharacters, 316
 - in a character class, 321
- milestones, 287
- misspelled words, 113
- mode span, 358
- modifier span, 358
- monitor files, 6
- monospaced, 157
- mouse, 222
- mouse pointer, 219
- mouse wheel, 216
- move file, 17
- move project, 41
- move selection, 76
- move tabs, 74, 75
- mru, 4
- MSIE, 268
- multi-line comment, 434
- multi-line search panel, 46
- multiple instances, 267
- multiple regexes, 433
- multiple regexes combined, 333
- named group, 345
- natural sort, 115, 116
- navigation, 202
- navigator, 290
- near, 436
- negated character class, 321
- negated shorthand, 325
- negative line numbers, 154
- negative lookahead, 363
- negative lookbehind, 364
- nested constructs, 373, 377
- nested grouping, 437
- Netscape, 268
- new, 4
- new editor, 215, 267
- new instance, 267, 309
- new page, 30
- new project, 32, 309
- new version, 296
- newline, 60, 130
- news feeds, 301
- next comparison mark, 124
- next edit, 72
- next file, 73
- next line, 130
- next project, 75
- next unsaved edit, 72
- non-capturing group, 339
- non-printable characters, 318
- non-spacing mark, 352
- notes, 259
- number, 325, 352, 434
 - backreference, 340

- exponential, 417, 434
- floating point, 417
- range, 415
- scientific, 417, 434
- numbered capturing group, 339
- numbers, 154, 174, 215
- numeric ranges, 415
- offset, 70, 243, 309
- often-used files, 8, 10
- okina, 135
- once or more, 336
- open, 4, 6, 33, 34, 279
- open closed files, 39
- open files, 36
- open files at startup, 232
- opening bracket, 353
- opening quote, 353
- OpenType, 159
- option, 333, 335, 336
- options, 149, 150, 151, 152, 153, 154, 156, 164, 165, 214
- or
 - one character or another, 321
 - one regex or another, 333
- organize favorites, 8
- OS X, 170
- other editor joint scrolling, 267
- outdent, 77
- outside files, 36
- overwrite, 25
- Overwrite, 243
- padding, 65
- page break, 130
- page breaks, 30
- pages, 30
- palette, 11, 186
- palindrome, 376, 385, 391
- panel preferences, 241
- paragraph count, 125, 126
- paragraph separator, 130, 352
- paragraph symbols, 174
- parameters, 309
- parentheses, 71, 339
- parser, 433
- paste, 25, 127
- pastebook, 259
- path placeholders, 66, 98
- pattern, 64, 315
- period. *see* dot
- persistent selections, 78, 215
- Pilcrow symbol, 153
- pipe symbol. *see* vertical bar
- placeholders, 65, 66, 96, 98
- play instant macro, 110
- plus, 316, 336
 - possessive quantifiers, 361
- pointer, 219
- portable installation, 303
- position, 70
- positive lookahead, 363
- positive lookbehind, 364
- possessive, 361
- precedence, 333, 339
- preferences, 95, 164, 165, 166, 167, 169, 186, 214, 215, 219, 228, 232, 241, 243, 247, 251, 254, 256
- prefix block, 86
- prepare to search, 45
- prepend lines, 396
- previous comparison mark, 125
- previous edit, 72
- previous editing position, 71, 72
- previous file, 74
- previous match, 398
- previous project, 75
- previous unsaved edit, 72
- previously edited file, 73
- primes, 140
- print, 11, 87, 394
- programming languages, 433
- project, 32, 33, 34, 36, 37, 38, 39, 40, 41, 43, 44, 75, 89, 113, 126, 232, 309
- project tabs, 237, 239
- prompt on replace, 53
- prompt to save, 232
- properties
 - Unicode, 352
- punct, 394
- punctuation, 353
- quantifier
 - backreference, 451
 - backtracking, 337
 - curly braces, 336
 - greedy, 336
 - group, 451
 - lazy, 337
 - nested, 437
 - once or more, 336
 - once-only, 361
 - plus, 336
 - possessive, 361
 - question mark, 335
 - reluctant, 337
 - specific amount, 336

- star, 336
- ungreedy, 337
- zero or more, 336
- zero or once, 335
- question mark, 316, 335
 - common mistake, 417
 - lazy quantifiers, 337
- question marks instead of text, 127
- quit, 19
- quotation style, 141
- quoted string, 327, 434
- quotes, 133, 134, 135, 137, 139, 140, 141
- quotes => primes, 140
- range of characters, 321
- Ratliff, 196
- read only, 8
- reboot, 232
- record instant macro, 110
- record macro, 107
- record size, 153
- rectangular selections, 79
- recursion
 - atomic, 391
 - backreference, 388
 - backtracking, 391
 - capturing group, 378, 384
- recursively opening files from folders, 34
- redo, 21
- ReDoS, 447
- reflow paragraphs, 142
- regex, 59, 60, 64, 167, 278, 315
- regex engine, 319
- RegexBuddy, 64
- regex-directed engine, 319
- RegexMagic, 64
- registry, 256
- regular expression, 59, 315
- regular expressions, 59, 64
- reload files, 228
- reload files from disk, 41
- reload from disk, 17
- reluctant, 337
- removable drive, 303
- remove all bookmarks, 90
- remove all folding points, 93
- remove closed files, 40
- remove fold, 92
- remove from project, 39
- remove obsolete files, 40
- rename file, 17
- rename project, 41
- reopen files at startup, 232
- reopen menu, 4
- repetition
 - backreference, 451
 - backtracking, 337
 - curly braces, 336
 - greedy, 336
 - group, 451
 - lazy, 337
 - nested, 437
 - once or more, 336
 - once-only, 361
 - plus, 336
 - possessive, 361
 - question mark, 335
 - reluctant, 337
 - specific amount, 336
 - star, 336
 - ungreedy, 337
 - zero or more, 336
 - zero or once, 335
- replace, 48, 65
- replace all, 53
- replace and find next, 52
- replace and find previous, 52
- replace current, 52
- replace smart quotes, 139
- requirements
 - many in one regex, 366
- reserved characters, 316, 401
- reserved characters => xml entities, 148
- reset, 347
- restore default layout, 269
- reuse
 - part of the match, 340
- rich text, 18, 24, 87
- right margin, 167
- right-to-left, 156, 157, 164
- ROT-13, 148
- round bracket, 316
- round brackets, 339
- row numbers, 175
- rss, 301
- RSS feeds, 301
- RTF, 18, 24, 87
- ruler, 175
- running programs, 95
- \s. *see* whitespace
- \S. *see* whitespace
- saturation, 189
- save, 10, 279
- save all, 11

- save all files in project, 33
- save as, 10
- save as HTML or RTF, 18, 87
- save copy as, 17
- save copy of project as, 41
- save project, 33
- sawtooth, 443
- scratch pad, 259
- script, 156, 164, 353
- scroll wheel, 222
- search, 45, 46, 48, 49, 50, 51, 52, 53, 54, 55, 57, 58, 59, 63, 65
- search options, 58, 59, 60, 61, 62, 63
- search pattern, 64
- search range, 61
- second instance, 267
- select all, 26
- select fold, 92
- selection only, 61
- selections, 76, 77, 78, 79, 82, 83, 84, 87, 113
- send to, 254
- separator, 352
- session, 32, 33, 232
- set any bookmark, 88
- set bookmark, 90
- several conditions in one regex, 366
- SFTP, 279
- shortcut keys, 247
- shortcuts, 254, 291
- shorthand character class, 325
 - negated, 325
 - XML names, 325
- show any type of file, 278
- show files of type, 278
- show indentation, 152
- show list of all matches, 57
- show paragraphs, 174
- show spaces, 174
- show spaces and tabs, 153
- side by side, 267
- side panel preferences, 241
- single spacing, 142, 143
- single-line comment, 434
- sleep, 235
- smart => straight quotes, 137
- smart End key, 221
- smart home key, 215
- smart Home key, 221
- smart quotes, 135
- SMTP, 251
- snippet storage, 259
- sort alphabetically, 115
- sort alphanumerically, 115, 116
- sort tabs alphanumerically, 74, 75
- source code, 433
- space, 394
- space separator, 352
- spaces, 143, 153, 174
- spaces and tabs, 176
- spacing, 161
- spacing combining mark, 352
- special characters, 316, 401
 - in a character class, 321
- specific amount, 336
- spell check, 112, 114
- spell check all, 113
- spell check highlight, 113
- spell check live, 113
- spell check project, 113
- spell check selection, 113
- spelling, 183
- split editor, 265
- split hexadecimal and ascii, 263
- split screen, 267
- square bracket, 321
- square brackets, 316
- squares instead of text, 127
- SSL, 279
- star, 316, 336
 - common mistake, 417
- start file, 328
- start line, 328
- start menu, 254
- start string, 328
- statistics, 125, 126
- status bar, 243
- stay on top, 164
- straight => smart quotes, 135
- straight quotes, 137
- string, 315, 434
 - begin, 328
 - end, 328
 - quoted, 327
- Stroustrup, 196
- subtract character class, 323
- suffix block, 86
- support, 302
- surrogate, 353
- swap selections, 77
- swap with clipboard, 26
- symbol, 352
- syntax coloring, 183, 193, 195, 213
- syntax colors, 186
- syntax highlighting, 183

- tab, 318, 402
- tab size, 167, 177
- tabs, 74, 75, 143, 153, 237, 239
- tab-separated values, 177
- tail, 6, 309
- TCL, 259
- template files, 10
- terminate lines, 318, 402
- text, 315
- text cursor, 219
- text direction, 174
- text encoding, 127, 170, 453
- text file icon, 169
- text layout, 156, 174
- text-directed engine, 319
- TextPad Collection Library, 259
- theme, 269
- time, 28, 215
- tip of the day, 295
- title case, 133
- titlecase letter, 352
- TLS, 279
- toggle all folds, 93
- toggle comment, 86
- toggle fold, 91
- toggle search panel, 46
- tool placeholders, 98
- toolbar, 305
- toolbar icon size, 305
- tools, 95, 96, 101, 103
- tools menu, 95
- trailing whitespace, 118, 119, 176
- trim all trailing whitespace upon save, 176
- trim trailing whitespace, 118, 119, 176
- trim whitespace, 118, 176
- trimming whitespace, 413
- triple click, 222
- TrueType, 159
- TSV, 177
- tutorial, 315
- two instances, 267
- uFFFF => characters, 145
- uncomment block, 85
- undo, 20, 21
- unfold all, 93
- unfold lines, 91
- ungreedy, 337
- unicode, 127
- Unicode, 170, 243, 351
 - blocks, 354
 - categories, 352
 - characters, 351
 - code point, 351
 - combining mark*, 351
 - grapheme, 351
 - properties, 352
 - ranges, 354
 - scripts, 353
- unindent, 77
- UNIX, 170
- unmatched bracket, 71
- unreadable file, 127
- unsaved changes, 232
- unsaved edit, 72
- unselect, 84
- unspecified file type, 166
- untitled files, 232
- unwrapping text, 142
- update, 296
- upgrade, 296
- upload to FTP, 279
- upper, 395
- uppercase, 66, 133
- uppercase letter, 352
- usb stick, 303
- using custom syntax coloring schemes, 195, 213
- UTF-16, 170
- UTF-8, 127, 170
- version, 296
- vertical bar, 316, 333
- vertical lines, 152
- vertical rulers, 175
- vertical tab, 130
- view, 215, 261, 263, 264, 265, 266, 267, 268, 269, 271, 275, 279, 287, 290
- view logs, 6
- Visual Studio colors, 186
- visualize line breaks, 153
- visualize paragraphs, 174
- visualize spaces, 153, 174
- \w. *see* word character
- \W. *see* word character
- wait, 309
- web browser, 268
- web site of EditPad, 296
- wheel, 216, 222
- Whitesmiths, 196
- whitespace, 118, 119, 153, 325, 352, 413
- whole line, 328, 430
- whole word, 331
- whole words only, 49, 60
- Windows, 170
- Windows code page, 127

- word, 331, 395
- word boundary, 331
- word character, 325, 331
- word count, 125, 126
- word wrap, 141, 142, 149, 167, 174, 175
- words, 49, 60
 - keywords, 435
- WordStar, 249
- working copies, 232
- workspace, 32, 232
- wrapping, 142

- write, 87
- www, 268, 296
- xdigit, 395
- xml entities => reserved characters, 148
- XML names, 325
- zero or more, 336
- zero or once, 335
- zero-based line numbers, 215
- zero-column selection, 79
- zero-length, 328, 363
- zero-length match, 396