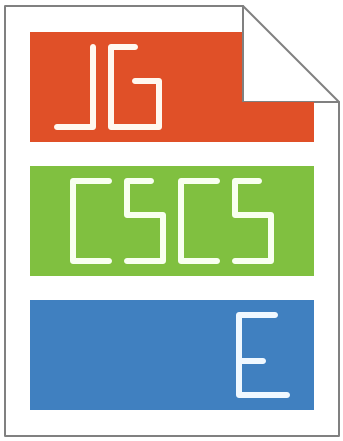

Just Great Software
Custom Syntax Coloring Scheme Editor
Manual

Version 5.2.0 — 9 February 2021

Published by Just Great Software Co. Ltd.

Table of Contents

1. Introduction

Welcome to the documentation for the Just Great Software Custom Syntax Coloring Scheme Editor.

This tool allows you to build custom syntax coloring schemes for EditPad and AceText . Version 5 of the scheme editor creates schemes compatible with EditPad Pro 8.0.0 and later and AceText 4.0.0 and later.

You can find the syntax coloring schemes that ship with EditPad and AceText in the folder where you installed the software. The default installation folder is a subfolder of C:\Program Files\Just Great Software. Each .jgcscs file contains one syntax coloring scheme. It is a good idea to study some of those schemes before attempting to create your own. You can download schemes created by other EditPad and AceText users at https://www.editpadpro.com/cscs.html. You can share your own schemes there too.

Schemes that you have downloaded are placed in product-specific folders under %APPDATA%\JGsoft under your Windows user profile. You can navigate to this folder by typing or pasting %APPDATA%\JGsoft in the address bar in Windows Explorer. You should place your own custom schemes in that folder too. You can do this easily by selecting EditPad Lite 8, EditPad Pro 8, or AceText 4 in the drop-down menu of the Save As button. If a .jgcscs file with the same file name exists in both the AppData folder and the Program Files folder, then the file in the AppData folder takes precedence.

The **tree view** at the left hand side of the scheme editor shows you the complete structure of the syntax coloring scheme. The settings presented in the right hand side of the scheme editor depend on the node that you have selected in the tree.

Select the topmost node in the tree to provide some general information about your syntax coloring scheme and the files it applies to. It's important to do this if you plan to share your scheme with other users.

The second node from the top is always labeled Main Coloring Scheme. There you create the actual syntax coloring scheme by adding scheme elements. The scheme elements appear as child nodes under the Main Coloring Scheme node. These are essentially a series of regular expressions and additional rules that are used to perform the actual syntax coloring.

You can add subschemes to create more complex syntax coloring schemes. Subschemes appear as sibling nodes below the Main Coloring Scheme node. Subschemes have their own scheme elements as child nodes.

The Legacy Brackets node at the bottom is only provided with backward compatibility for schemes created for older versions of EditPad Pro . New schemes should define brackets as part of the scheme elements.

The bottom half of the scheme editor provides an editor to preview and test your syntax coloring scheme. It has a toolbar with various buttons that correspond to similar buttons and menu items in EditPad Pro.

Before creating your first syntax coloring scheme, be sure to read the topic about Reliable Coloring Schemes topic. It's not enough to create a scheme that properly highlights a valid file. It also needs to work reliably with files in various stages of editing that do not adhere (yet) to the file format you're working with. If your scheme will be applied to very large files, you should also read the Fast Coloring Schemes topic. If your scheme needs to deal with files without line breaks, read the Breaking Schemes topic.

2. General Information

Select the topmost node in the tree to provide some general information about your scheme. This information is used to build the list of available schemes when you upload your scheme to https://www.editpadpro.com/cscs.html. Please fill out this page completely.

Scheme Name: The name of your syntax coloring scheme. This name is presented to the user when a syntax coloring scheme should be selected. Each scheme should have a unique scheme name. If two schemes have the same name they will appear right next to each other and the user won't be able to distinguish them. Changing the name of a scheme won't break any preferences or file type configurations that use the scheme. Those reference schemes by their file names.

File Extensions: The file extensions typically used by files to which your coloring scheme should be applied. Note that this item is purely informational. Any scheme can be applied to any file, no matter its extension. You should list extensions as full file masks, delimited by semicolons, e.g.: *.html;*.htm;*.shtml

Author Name: Your name.

Author Email: Email address that can be used to send you comments about your scheme. Your email address is not shown on the EditPad Pro web site when you upload your coloring scheme. It can be seen by people who open your scheme in the scheme editor.

Author Web Site: Complete URL of your web site. If you upload your scheme to EditPad Pro's web site then your name will be linked to this URL.

File Type Web Site: Complete URL of the web site that is the central source of information about the file type your scheme provides syntax coloring for. Try linking directly to a page with relevant information, rather than a top-level domain, unless the top-level domain is specific to the file type.

Block|Comment: The Block|Comment command in EditPad and the Format|Comment Block command in AceText use these two items to comment out a block of text. If single-line comments are supported by the file format you're creating a scheme for, enter the character that start a single-line comment in the left hand field, and leave the right hand field blank. If multi-line comments are supported, enter the characters that start a multi-line comment in the left hand field, and the characters that close it in the right hand field. If both single-line and multi-line comments are supported, enter the characters for the single-line comment only. The Block|Comment command is more flexible when used with single-line comments. If no comments are supported, leave both fields blank.

Comments shown to users: If there's anything important that people should know before using your scheme, you can enter that information into the edit box in the upper right corner. Particularly if you're creating a scheme for a file type for which one or more schemes are already available, explain what makes your scheme different. This text is displayed on the web site when people download syntax coloring schemes. So try to be brief in order to keep the length of the page with all the schemes reasonable. This is *not* the place for technical notes for people who may want to edit or build upon your scheme. The Main Coloring Scheme section has a space for developer notes.

Example: In the lower right corner edit box, enter a handful of lines of text as a small example file. Try to put as many different scheme elements (colors) in the example as you can. The color palette customization dialog in EditPad and AceText shows this example text so the user can preview your scheme with their

custom palette. In that dialog the user can also double-click text in the example to select the color that is applied to that text.

3. Reliable Coloring Schemes

It is very important to test your coloring scheme thoroughly. Make sure to test it on invalid text such as strings with a missing closing quote, etc. Test your scheme by performing normal and abnormal (typing nonsense) editing operations in the test editor to see if your scheme properly updates itself. This may require the addition of additional syntax elements that match unclosed strings and other stuff that is not part of the file format you are trying to color, but could appear while the user is editing his files.

Say you are creating a scheme for a language that has single-line double-quoted strings. If you specify `"[^"\v]*"` as the regular expression then the string does not get its color until both double quotes have been entered. But most text editors color everything up to the end of the line as a string while the second double quote is missing. The schemes supplied with EditPad and AceText do this as well. Therefore, `"[^"\v]*"?` is a better regex in this case. This colors everything after a double quote up to the next double quote or the end of the line as a string.

When a text file is being edited, it is only partially parsed again to update the syntax coloring. Only the paragraphs that have been modified are parsed again completely. The parser starts from the last syntax change before the start of the modified paragraphs. It continues until it reaches a point after the modified paragraphs where the newly applied syntax coloring is identical to the previous coloring.

This has consequences if your scheme contains elements that can span multiple paragraphs. There is no guarantee that such elements will not be chopped in pieces when only certain paragraphs are parsed again. Therefore, it is important that the scheme includes both the correctly terminated scheme element and the non-terminated element which continues until the end of the file. While only the former should appear in a file that adheres to the syntax you are trying to create a scheme for, the non-terminated version can appear while the user is editing the file. This makes sure that the syntax coloring is properly updated while the user's editing actions break and fix the proper syntax.

For example, to color double-quoted multi-line strings, use two scheme elements using the regexes `"[^"]*"` and `"[^"]*`. You need to use two different scheme elements so the software can differentiate between a properly terminated and an endless double-quoted string. This is important for the software to be able to determine if it needs to continue re-coloring the file beyond the last modified paragraph, and how far. The element with regex with the closing quote needs to be above the one without the closing quote so that it takes precedence. For syntax elements that cannot span multiple lines/paragraphs, you can use a single scheme element as described above, because the entire paragraph is always be re-colored.

Alternatively, you can use a toggle subscheme to color multi-line items. For example, for double-quoted multi-line strings, create an element in the main scheme that matches nothing but a double quote. Have that element toggle to a subscheme. The subscheme also needs have an element that matches a double quote. But this element toggles back to the main scheme. If quotes can be escaped inside strings, you can easily handle that with a second element in the subscheme using the regular expression `\\.` which matches a literal backslash and whatever character follows it. If the file has an unterminated string, the element in the subscheme that matches the closing quote never finds its match. The subscheme then highlights everything until the end of the file. This and the fact that the opening and closing quotes are matched by separate elements automatically ensures that colors are updated correctly when multi-line strings are edited.

Because of all of this, it is very important that you test your scheme while doing some heavy editing. Try to break the syntax rules and see if your scheme colors everything properly. Color should be correct when the file is syntactically valid. Colors should be reasonable when the file is syntactically invalid so that not too much flashing takes place while the file transitions between valid and invalid syntax as it is edited.

Finally, **do not try to do too much** in your syntax coloring scheme. The syntax coloring scheme is re-applied with each and every keystroke. If your scheme is too complex, EditPad will need a long time to apply it. If EditPad determines your scheme is taking too long, it displays the file without syntax coloring until your scheme has finished updating. This is not a problem when it happens when the user jumps to the end of a long file for the first time. But it is a problem if it happens with each and every keystroke.

The syntax coloring system is designed to provide visual cues to the user. Using a select set of different colors for different elements of the text makes it far easier for the human eye to navigate the text. But using too many colors for too many different things confuses the eye, rather than guiding it. The schemes are also not designed for complex syntax validation. Such validation would need to be done in the background, rather than with each key press, as is appropriate for syntax coloring. For example, if the file format you are designing a scheme for has a certain number of reserved words that can be used in a specific way, let your syntax coloring scheme color those reserved words without bothering to check if they were used appropriately. The latter is a job for a compiler or interpreter, not for a syntax coloring scheme.

4. Fast Coloring Schemes

There are two kinds of syntax coloring schemes. Most schemes require the whole file to be processed up to the part of the file that is visible on the screen. Some schemes only require the visible part to be processed. Obviously, the latter kind will be applied orders of magnitude faster when working with large files, since only a tiny part of the file needs to be processed.

The reason most schemes require the whole file to be parsed is that they have elements that can span multiple lines. For example, many programming languages have multi-line comments or strings. Therefore, everything before the visible part of the file must be scanned to check if there are any unclosed multi-line comments or strings that continue into the visible part of the file.

EditPad does cache syntax coloring information. So the whole file isn't reprocessed with every edit. However, the first time you scroll down a lengthy file, it may take a while for syntax coloring to appear.

If a scheme does not have elements that can span multiple lines, then EditPad can safely start applying it from the start of the first visible line. Whatever precedes the visible part cannot affect the colors of the visible part. Since the visible part is always very small, syntax coloring is applied instantly, even when the file is huge.

There are a few easy rules to make sure your scheme falls into the ultra-fast category. First, it cannot have any subschemes of type "toggle". Such subschemes may continue onto the next line. All other types of subschemes, including "toggle until end of line" and "toggle within detail" are permitted. These subscheme types guarantee the subscheme doesn't run beyond the end of the line or beyond the regex match of the element that uses the detail scheme.

Second, none of the regular expressions should be able to match line breaks. To do so, make sure "dot all" is turned off for all elements. None of the regular expressions should contain tokens such as `\r`, `\n`, `\s`, `\R` or `\v` that match line break characters. All negated character classes, however, should contain `\v` or `\s` to exclude all line breaks. Excluding only `\r` and `\n` is not sufficient. EditPad supports all Unicode line breaks. The shorthand for vertical whitespace `\v` matches them all. The shorthand for all whitespace `\s` does too. Your regular expressions should also not negated shorthands such as `\W` and `\D` as these too can match line breaks.

If your scheme uses bracket matching then each scheme element that defines any brackets must define both the opening and the closing bracket. Otherwise bracket matching may need to look on lines before or after the visible part of the file to look for the matching bracket. Schemes that meet all the rules except the bracket matching rule for fast schemes can still be used as fast schemes by EditPad. If the file is huge, bracket matching is automatically disabled. When bracket matching is disabled (regardless of file size) the scheme is applied to the visible part of the file only. When bracket matching is enabled (meaning the file isn't huge) the scheme is applied to the whole file like a non-fast scheme.

Help From The Scheme Editor

The scheme editor can tell you whether your scheme is fast or not. Click the **Fast Scheme? button** on the toolbar above the preview editor. You'll get a message box explaining the situation. The first paragraph explains what a fast scheme is. Then the message either says "This is a fast coloring scheme" or explains why your scheme is not fast. The first subscheme or regex that doesn't follow the rules is selected. Adjust it and click the button again to see if any others need adjustments.

If your scheme follows the rules for fast schemes then **EditPad's file type configuration** lists it in both the "syntax coloring scheme" drop-down list and the "syntax coloring scheme for huge files" drop-down list in the file type configuration. If not then your scheme is only listed in the "syntax coloring scheme" drop-down list.

Examples

It may be useful to create fast variants of regular coloring schemes. For example, the various SQL schemes supplied with EditPad also come in fast variants. Modern SQL variants support multi-line strings and comments. SQL files with data dumps can be incredibly large. The fast variants of the schemes don't recognize multi-line comments and strings. They can be used to provide syntax coloring for huge SQL files, with the only trade-off that multi-line comments and strings won't be colored properly. In EditPad's file type configuration you can select the full-featured SQL scheme from the "syntax coloring scheme" drop-down list and the fast scheme from the "syntax coloring scheme for huge files" drop-down list. This way you get the best syntax coloring when the file size is reasonable and still get some syntax coloring when the file size is huge. The limitations of the fast scheme may not even be an issue if you configure your data dump tool not to use escape characters for line breaks in strings.

The schemes for INI files, JSON, and Visual Basic included with EditPad are fast schemes when bracket matching is disabled. These schemes don't have any trade-offs. These file formats don't have any syntactic elements that can span multiple lines.

5. Breaking Schemes

Some files may not contain any line breaks at all. This is common for computer-generated XML and JSON files, for example. Such files can't be practically viewed or edited with a text editor. Fortunately, EditPad Pro is capable of automatically adding line breaks and indentation to such files. It can do this with a syntax coloring scheme that meets the rules for a "breaking" scheme.

A "breaking" scheme must have at least one element that has "break and indent" set to the rules for braces or for markup tags or to a custom rule that automatically breaks before or after. Those elements define the places where EditPad Pro can insert line breaks.

EditPad Pro can also add indentation when adding line breaks. The "indent by one step" and "outdent by one step" rules increase and decrease the level of indentation of the line after the line break. The user can configure the size of the indentation step on the Tabbing page in the file type configuration in EditPad Pro. These two rules and "keep indentation level" are the only indentation rules allowed in "breaking" schemes. Aligning with regex matches or brackets is not supported because column positions are not calculated while breaking an entire file.

For performance reasons, EditPad Pro does not use "detail" subschemes when breaking and indenting single line files. A "breaking" scheme can have "detail" subschemes. But those subschemes should not have any elements with "break and indent" set to anything other than "don't break; don't indent".

It does not matter whether the scheme meets the rules for "fast" schemes. Files without line breaks have to be processed entirely anyway.

Testing Breaking Schemes

The scheme editor can tell you whether your scheme is a "breaking" scheme or not. Click the **Breaking Scheme? button** on the toolbar above the preview editor. You'll get a message box explaining the situation. The first paragraph explains what a "breaking" scheme is. Then the message either says "This scheme can be used to add line breaks to files that have none" or explains it explains why that is not the case. Adjust the scheme and click the button again.

If you use the **Open button** or its drop-down menu to open a file that does not have any line breaks, then the scheme editor keeps an in-memory copy of the file as it existed (without line breaks) when you opened it. Whenever you preview a scheme, the scheme editor reloads the sample from that in-memory copy. If the scheme is a "breaking" scheme, line breaks are added to the sample according to the scheme's rules when the sample is reloaded. Otherwise, the sample is displayed without any line breaks. If you already previewed a "breaking" scheme before you opened the file without line breaks, then line breaks are added to the displayed sample immediately.

Examples

JSON.jgcscs is a "breaking" scheme. The JSON format is simple enough. No detail subschemes are needed. So the same scheme can be used for coloring JSON files as for adding line breaks to JSON files that don't have any.

XML.jgcscs and XML_fast.jgcscs both use detail subschemes to apply different colors to different parts of XML tags. The elements that match angle brackets in those detail subschemes specify the "break and indent" rules. This allows those schemes to provide automatic breaking and indentation while typing. But they cannot be used to add line breaks to XML files that have none.

XML_break.jgcscs was created specifically for adding indentation to XML files. This is a very simple scheme. It has only the bare minimum of elements to properly match XML tags. It has no subschemes. This makes the scheme very quick. The elements that match the XML tags define the "break and indent" rules.

6. Preview And Test

The bottom half of the scheme editor is a text editor control where you can preview and test your syntax coloring scheme. It has its own toolbar with various commands and options that allow you to test how the scheme behaves when you edit a file.

The **Open button** lets you open any text file. It has a drop-down menu with a history of recently opened files.

The **Preview button** applies your syntax coloring scheme to the sample file. It also applies it to the example that is part of your scheme's general information. If there is a problem with the scheme, such as an invalid regular expression, you'll get a popup message. The offending scheme element is selected.

Changes you make to your scheme are not automatically applied to the sample. You have to click the Preview button again each time you want to test the changes you've made. Schemes may be temporarily invalid while you edit them. So the sample editor keeps the most recent successful preview until you click the Preview button again.

The **Fast Scheme? button** explains the rules of "fast" syntax coloring schemes. It tells you whether your scheme meets those rules or why it does not. Such schemes only need to parse the visible part of the file. This makes them very fast and independent of the size of the file.

The **Breaking Scheme? button** explains the rules of "breaking" schemes. It tells you whether your scheme meets those rules or why it does not. EditPad Pro can use such schemes to add line breaks to files that have none.

The **Color Palette** button lets you pick the color palette for the preview. The color palette translates the named colors that you can select in the scheme editor into actual colors. The **Standard Palettes** are those that are built into our products. Your scheme should work well with those. The palette is not saved into the scheme. It only affects the preview. If you created **custom palettes** in EditPad Lite 8, EditPad Pro 8, or AceText 4 then you can select those from separate submenus.

The **Dark Theme** item at the bottom of the Color Palette menu toggles the entire user interface of the scheme editor between the standard Windows them and a dark theme.

The **Word Wrap** button toggles between wrapping long lines at the edge of the preview editor and not wrapping lines.

The next set of five buttons lets you test the bracket matching settings of your scheme. They are disabled if your scheme does not define any brackets. Bracket highlighting is always enabled in the preview. When a pair is highlighted, **Go to Matching Bracket** moves the cursor from one highlighted bracket to the other. **Go to Unmatched Bracket** moves the cursor to the nearest bracket that does not have a matching bracket. EditPad has these two commands in its Go menu.

Between Matching Brackets selects the text between the highlighted pair of matching brackets. If that text is already selected, the selection is expanded to include the brackets. If they are already included, the selection is expanded to the pair of matching brackets that the selected brackets are nested inside of. EditPad has the same command in its Select menu.

Insert Matching Brackets generates the matching bracket for the highlighted unmatched bracket. This only works if the scheme element that defines the unmatched bracket specifies text to generate a bracket. It works for unmatched opening brackets when the cursor is after them and for unmatched closing brackets when the cursor is before them. **Auto Match Brackets** automatically generates a closing bracket when you type an opening bracket if the scheme element that defines the opening bracket specifies a bracket to be generated automatically. EditPad has this command and option in its Edit menu.

The next two buttons allow you to the rules for breaking and indentation of your scheme. They are disabled if your scheme does not have any elements that specify automatic breaking or indentation. Turning on the **Auto Indent** option makes the preview editor use the scheme's rules to increase or decrease the amount of indentation on the newly entered line when you press the Enter key on the keyboard. Turning on the **Auto Break** option makes the preview editor automatically insert line breaks as specified by your scheme. Automatically inserted line breaks are always automatically indented too. EditPad Pro has the same options in the Options menu. The preview editor does not have a Keep Indent option. This is always enabled. When auto indent does not apply, pressing Enter indents the new line by the same amount as the previous line.

Scheme elements may set "breaking and indent" to treat the element as a **brace** or as a **markup tag**. The drop-down menu of the Auto Break button lets you choose from an number of styles for breaking and indenting braces. It also lets you toggle whether markup tags that contain mixed content should have automatic line breaks added to that mixed content. These settings are not saved into the syntax coloring scheme. They only affect the preview. In EditPad Pro, the user of the scheme can choose their own style for breaking and indenting braces and markup tags on the Brackets page in the file type configuration. EditPad Pro's help file explains the different styles for brackes and markup in detail.

The next three buttons allow you to test the quote conversion rules of your scheme. **Adjust Quotes** does a one-time adjustment of the quotes in the selection following the scheme's rules in the same way as the Convert|Quotes|Adjust Quotes command in EditPad Pro and AceText. **Auto Quotes** automatically adjusts quotes after you've typed them in the same way as the Convert|Quotes|Auto Adjust Quotes toggle in EditPad Pro and AceText. If all your elements and subschemes specify "don't convert quotes" then these two buttons are disabled in the scheme editor. But EditPad Pro and AceText only disables their Adjust Quotes command. They do not disable Auto Adjust Quotes. Instead Auto Quotes converts all straight quotes to smart quotes when the syntax coloring scheme offers no quote conversion rules. The **Quotation Style** button lets you choose the quotation style for the preview editor. This style is not saved into the scheme. In EditPad Pro and AceText users can choose their own style via Convert|Quotes|Quotation Style.

The last two buttons allow you to test the case conversion rules of your scheme. **Adjust Case** does a one-time adjustment of the case of the selected text following the scheme's rules in the same way as Convert|Case|Adjust Case does in EditPad Pro and AceText. **Auto Case** automatically adjusts the case of text after you've typed it in the same way as the Convert|Case|Auto Adjust Case menu item in EditPad Pro and AceText. If all your elements and subschemes specify "don't change case" then these two buttons are disabled.

Automatic adjustment of quotes and case only happens when you move the cursor away from the line you entered them on. This allows the adjustment to take whole lines and whole words into account so that it can differentiate between opening quotes and leading apostrophes and can properly capitalize words.

7. Coloring Scheme

Select "Main Coloring Scheme" near the top of the tree view to edit the main part of your syntax coloring scheme. You can specify a **default** color, spell check, quote conversion, and case adjustment rule for any text that is not matched by any element in your scheme. These are explained later in this topic. The "include" setting is explained in the help topic about subschemes.

You can use the **Developer Notes** box to enter any information that may be helpful when you or others want to edit the scheme in the future. These notes are only displayed in the scheme editor. Comments for people using the scheme should be entered with the general information on the scheme.

The main coloring scheme consists of one or more **scheme elements**. Use the **Add Element** and **Delete** buttons to add a new scheme element or to remove the selected one. The **Cut**, **Copy** and **Paste** buttons on the toolbar work on scheme elements (and also on whole subschemes).

Use the **Move Up** and **Move Down** buttons to move the active element up or down. **The order of the scheme elements is important.** If the regular expressions of two or more elements match *at the same position* in the file, the topmost element is be the one that is applied to the piece of text that starts at the regex match position. An example: if your scheme has both a "reserved word" element which matches a limited set of words, and an "identifier" element which matches any word, the "identifier" regex matches whatever the "reserved word" regex matches. Therefore, the "reserved word" element must be positioned above the "identifier" element so that "reserved word" takes precedence. Otherwise, all reserved words would be colored just like identifiers.

If two elements could potentially find overlapping matches starting at different positions in the file, then their order in the scheme is irrelevant. The leftmost match always takes precedence. If the same example scheme also has a "variable" regex that matches a dollar sign immediately followed by any word, then it doesn't matter that the same word could also be matched by the "reserved word" or "identifier" elements. The match of the "variable" regex would always be found first. Text matched by that regex is then no longer available to be matched by any other regex. If reserved words aren't permitted as variable names, then you need to adjust the "variable" regex to not match reserved words.

Scheme Elements

Name: Short, descriptive name for your own reference when editing the scheme. Elements can have the same name or even no name at all. But the scheme editor automatically assigns a generic name when pasting an element with no name or a duplicate name to encourage you to enter a unique name.

Comments: Comments on this scheme element. If your scheme has similar elements you should enter some comments to help understand the purpose of each element.

Regex: Regular expression that matches the text that this scheme element should apply its color and other settings to.

Case insensitive: When ticked, the regex is applied without regarding any difference to uppercase and lowercase letters.

Free-spacing: Turn this on to ignore spaces and line breaks in the regular expression, unless they're part of a character class or preceded by a backslash. You can use this to make complex regular expressions more readable by using multiple lines with indentation. In this mode, # starts a comment unless it is in a character class or preceded by a backslash.

Dot matches line breaks: In a regular expression, the dot . matches any character except line break characters. If the dot should match everything, including line breaks, tick "dot matches line breaks".

Color: The color that this scheme element applies to the text matched by its regular expression. You can only select specific named colors. Try to select the one that most closely describes the part that this scheme element highlights. The actual color (red-green-blue value) depends on the color palette selected by the user. In EditPad the palette is part of the file type configuration. In AceText the palette is a global setting in the Appearance Preferences. But AceText can use different palettes when printing or exporting collections and when sending clips as HTML or RTF to another application. The scheme editor allows you to preview your scheme with any palette available in EditPad Lite 8, EditPad Pro 8, and AceText 4. The drop-down list shows each named color in the actual colors of the preview palette. This makes it easier to select the correct named color. But it does not mean that the actual color is saved into the scheme.

Spell check: Choose whether the matched text should be included in a spell check operation. This applies to manual spell checks invoked with the Extra|Spell Check menu item. It also applies to live spelling (wavy red underlines) in EditPad Pro and AceText. Turn on "spell check" in the scheme editor if the regex most likely matches natural language. For anything else, turn it off. EditPad's and AceText's schemes for programming languages have this turned on for comments and strings, but not for anything else.

Color whitespace: Choose whether the background color, underline, and strikeout should be applied to any whitespace matched by the regex. You'll usually want to turn this on so highlighting and underlining runs uninterrupted. Note that this setting is relevant even if your chosen named color does not have a background color, underline, or strikeout. The user of your scheme may use it with a different palette that does give the named color a background, underline, or strikeout.

Color visualized space: Choose whether the text and background colors, underline, and strikeout should be applied to visualized whitespace. Turn this on if whitespace is an effective part of what is being colored. EditPad's and AceText's schemes turn this on for strings as the whitespace really is a part of the string. But they have it turned off for comments (and almost everything else) as there is no need to emphasize whitespace in comments. Then the palette's color for visualized whitespace is used. This is a muted color in the predefined palettes. If you turn this off but turn on "color whitespace", then the background color, underline, and strikeout are still applied to visualized whitespace.

The options for coloring whitespace also apply to **control characters** other than tabs and line breaks that your regex may match. Turning on either option applies the background color, underline, and strikeout to control characters. Turning on "color visualized spaces" also applies the text color to control characters. Otherwise, the palette's color for control characters is used.

Color line breaks: Choose whether the color should be applied to visualized line breaks. Turn this on if line breaks are an effective part of what is being colored. EditPad's and AceText's schemes turn this on for multi-line strings as the line breaks really are a part of the string. But they have it turned off for comments (and almost everything else) as there is no need to emphasize line breaks in comments. Then the palette's color for visualized whitespace is used for line breaks. This is a muted color in the predefined palettes.

Until right margin: This option affects scheme elements that match text at the end of a line. The effect is only visible if the palette applies a background color or underline style (other than "font") to the element's

named color. Turning this on makes the background color or underline run until the right hand margin of the editor control, instead of ending where the text ends. This can be useful to make section headings stand out and act as a divider. The "section" scheme element in INI.jgcscs uses this option to highlight the entire line that starts a [section] in .ini files.

Apply detail: Select (none) to apply all the settings of the scheme element. Select "RGB Preview" if you want the scheme element's color to be based on the regular expression match. The named color you selected above is then used as a fallback if the preview failed or if the user has disabled it. RBG preview can be disabled in the file type configuration in EditPad. It cannot be disabled in the scheme editor or in AceText. All of the other settings in the scheme element are applied normally.

If you added one or more "detail" subschemes to your syntax coloring scheme, then you can select them in the "apply detail" drop-down list. Doing so overrides all of the element's settings except the brackets and "toggle to". The help topic about subschemes explains this in more detail.

Toggle to: After applying this element's color or its detail subscheme, continue applying syntax coloring after this element's regex match using another subscheme. The help topic about subschemes explains this in more detail.

Next subscheme highlight: If "toggle to" is set to a subscheme that uses a highlight color then this option determines whether the subscheme's highlight color is also applied to this element. PHP.jgcscs, for example, uses this to apply the "code highlight" color that is used by the "PHP" subscheme also to the "PHP" element in the main coloring scheme that matches the opening <?php tag.

Action: Action to be taken when the user double-clicks on a piece of text this scheme element was applied to. The "tool errors" schemes included with EditPad use actions to make file names in error messages open the file at the correct location when double-clicked. It is customary, but not necessary, to select the "local link" or "internet link" colors for scheme elements that use actions.

- Open in editor: open the file specified in the first parameter in the application's built-in editor.
- Open in editor at line: like "open in editor", except that you can also specify a second parameter with the line number at which the file should be opened.
- Open in editor at line:col: like "open in editor", except that you can also specify a second parameter with the line number and column number delimited by a colon at which the file should be opened.
- Execute file: open the file specified in the first parameter with the default application for opening that kind of file. If the file is an executable program, you can specify command line parameters in the second parameter for the action.
- Start email: start a blank file and shows the email panel in EditPad Pro, using the email address specified in the first parameter as the To: address for the new mail.
- Open in browser: open the file or URL specified in the first parameter in the default browser.

Action parameters: If you select an action, you will get one or two extra boxes to provide parameters for that action. If you leave the first parameter blank, the entire text matched by the scheme element's regex is used as the first parameter. You can use part of the regex match by typing in a backreference such as `\1` as the parameter. See the regular expression tutorial in EditPad's or AceText's help file for more information on backreferences. The entire replacement string syntax, including backreferences and conditionals, is supported. You can also enter literal text and path placeholders. The path placeholders are expanded using the file name of the file being colored using your syntax coloring scheme. Path placeholders are expanded after the backreferences have been expanded.

Opening bracket: If this scheme element matches an opening bracket (and nothing else), select 0 (zero) in the drop-down list to specify that the whole regex match is an opening bracket. If the scheme element matches an opening bracket along with other text, add a capturing group to the regular expression that captures the opening bracket. Then select the group's number or name in the "opening bracket" drop-down list.

Closing bracket: Same as "opening bracket" but for elements that match a closing bracket. If the scheme element matches both an opening bracket and the closing bracket it pairs with, put a capturing group around each, and specify both "opening bracket" and "closing bracket". For example, the "string" element in Delphi.jgcscs specifies both "bracket open" and "bracket close" to pair the quotes that enclose a string.

If the capturing group referenced by "opening bracket" or "closing bracket" does not participate in the match or finds an empty match, then that bracket is treated as missing. Many of the schemes provided with EditPad define the opening and closing quotes of strings as brackets. They also make the closing quote optional so the scheme reliably handles unterminated strings. When an unterminated string is matched, its opening quote is treated as an unmatched bracket.

Bracket pairing: If you specified an opening bracket or a closing bracket but not both then you need to enter the same text in the "bracket pairing" box for the scheme element with the opening bracket and the one with the closing bracket. This allows the brackets to be paired. If your scheme has multiple pairs of brackets, then different bracket pairs need different text in the "bracket pairing" box. If one scheme element specifies both the opening and closing bracket, then you can leave "bracket pairing" empty as the two brackets are then paired automatically. In CPP.jgcscs, for example, has scheme elements that pair up braces, square brackets, and parentheses.

The text matched by the regular expression does not have to be the same for each opening or closing bracket. The "pairing" setting determines whether brackets are a pair or not. In Delphi.jgcscs, for example, the "Begin" scheme element matches one of the words "begin", "case", "record", "try", etc. It specifies 0 (the whole regex match) for "opening bracket". Pairing is set to "beginend". The "End" element matches just that word, specifying 0 for "closing bracket" and "beginend" for pairing. This causes the scheme to treat "begin", "case", "record", "try", etc. as opening brackets that can all be paired with "end".

You can use backreferences to capturing groups in the regular expression such as `\1` for bracket pairing. This allows you to use (part of) the regex match to determine what the bracket should be paired with. VB.jgcscs uses the "Block" element to define one of a list of keywords as an opening bracket. It has "bracket pairing" set to `\0` which makes the whole regex match (the keyword) the pairing text. The "End Block" element defines End followed by one of the same list of keywords as a closing bracket. The first capturing group in its regex captures the keyword. It has "bracket pairing" set to `\1` so that just the keyword is used for pairing. Together this allows the matches of the "Block" element to be paired with those of the "End Block" element, but only when they matched the same keyword. The same technique works when the brackets could be anything. XML.jgcscs matches entire XML tags with the "Opening tag" and "Closing tag" elements. The entire tags are defined as brackets. Both capture the name of the tag into the first backreference which is then used for bracket pairing.

Bracket pairing is case sensitive. If you want to pair up brackets case insensitively, you need to use backreferences that convert case. For example, `\L1` inserts the text matched by the first capturing group converted to lowercase. The HTML.jgcscs scheme matches opening and closing tags much like the XML scheme does. But it uses `\L1` to pair the tags case insensitively.

Within 1 subscheme: Turn this on if the bracket should only be paired with other brackets within the same block of text highlighted by the subscheme that contains the element defining the bracket. In other words,

the scheme stops looking for a matching bracket when a scheme element changes to another subscheme. Turn this off to find matching brackets throughout the file.

Bracket nesting: This setting is relevant if your scheme defines more than one pair of brackets. If you leave the "nesting" setting blank for a pair of brackets, then that pair of brackets cannot be nested in itself. For example, when matching parentheses without nesting, the inner pair in (()) is matched, while the outer brackets are highlighted as unmatched brackets. When "nesting" is blank, brackets from different pairs are ignored. For example, when matching parentheses and square brackets without nesting, the inner pair of parentheses in ([]) is still matched, while the outer pair is not. The pair of square brackets is also matched. Essentially, bracket pairs without nesting cannot be nested within themselves, and ignore all brackets that belong to other pairs.

If you specify something for "nesting", then you must always specify the same "nesting" for each "pairing". You can specify different nestings for different pairings. If "nesting" is specified for a pair, then bracket matching check whether that pair is correctly nested within itself. For example, when matching parentheses with nesting, both pairs in (()) are matched. Bracket pairs with a different "nesting" setting ignore each other. For example, when matching parentheses and square brackets using different "nesting" for the parentheses and for the brackets, then all 3 pairs in ([]) are considered to be correctly matched brackets.

If you require different pairs of brackets to be nested correctly, then you need to specify the same "nesting" for all those pairs. For example, when matching parentheses and square brackets using the same "nesting" for all, then only the inner pair of square brackets in ([][]) is highlitghed as correctly matched brackets. The other three pairs are highlighted as incorrectly matched brackets because the brackets between them are not nested correctly.

The PHP.jgcscs scheme is an example of a scheme that uses two sets of bracket nesting. The scheme elements that match HTML tags all have "nesting" set to "html". The scheme elements that match PHP code have "nesting" set to "php". This allows HTML tags to be paired and nested regardless of any PHP code in between and allows braces in PHP code to be paired and nested regardless of any HTML tags in between. The same scheme also has elements that define < and > in HTML tags as brackets. These elements do not specify anything for "nesting" as < and > cannot be nested.

You can use backreferences to specify bracket nesting as you can do for bracket pairing. But you have to make sure that after backreferences are substituted, same "nesting" text is always used for the same "pairing" text. If you don't, only the first "nesting" choice for each "pairing" is actually used.

Generate bracket: If a scheme element specifies only one of "opening" or "closing bracket", or both are specified but one refers to a group that is optional, then you can specify the text for the other bracket in the "generate" box. For example, specify) for a scheme element that matches (as an opening bracket. This text is inserted by Edit|Insert Matching Bracket in EditPad Pro when a (without a matching) is highlighted.

You can use backreferences to have the generated bracket adapt it to the bracket it needs to match. For example, the XML.jgcscs scheme has "Opening tag" and "Closing tag" elements that capture the tag's name in the first (and only) group. The "opening tag" elements use `</\1>` to generate the closing tag, and the "closing tag" element uses `<\1>` to generate the opening tag.

Auto: Tick the "Auto" checkbox if you want the Edit|Auto Match Brackets option in EditPad to automatically generate the closing bracket when you type the opening bracket. You should only turn this off if the closing bracket could be generated prematurely and cause it or other text to be interpreted incorrectly. The XML and HTML schemes, for example, do not automatically generate a > when you start typing a tag as that would interfere with automatically generating a closing tag after typing an opening tag.

Only after line break: If you tick "Auto", you can tick "Only after line break" to delay generating the matching bracket until a line break is entered by the user or automatically generated by the Auto Break option. This is necessary if the bracket could be matched prematurely. Delphi.jgcscs for example has a "Begin" element that automatically generates the "end" keyword to match the "begin" keyword. But "end" should not be generated immediately when the user types "begin" as the user may intend to continue typing a longer word which should not be paired with "end". By turning on "only after line break" the "end" keyword is only generated when the user presses Enter after typing "begin" which indicates "begin" is indeed intended to be the start of a block to be paired with "end".

Break and indent: This choice affects how the Auto Break and Auto Indent options in EditPad (and the preview in the scheme editor) automatically add line breaks before and after text matched by this element and how they indent the lines created by automatic or manual line breaks before and after this element.

The "opening brace" and "closing brace" are intended for elements that match the start or end of a code block, for which many programming languages use curly braces. In the scheme editor you can select a predefined block indentation styles in the drop-down menu under the Auto Break button on the preview toolbar to test this. In EditPad Pro the user can configure exactly how braces should be indented in the file type configuration. People can have strong opinions about this. So use the "opening brace" and "closing brace" options whenever possible, even if your language does not actually use braces. Delphi.jgcscs, for example, uses these options for the "begin" and "end" keywords.

The various "markup tag" options are indented for HTML and XML tags. If a scheme element matches tags entirely, select "opening markup tag" for `<tag>`, "closing markup tag" for `</tag>` and "empty markup tag" for `<tag/>`. If a scheme element matches `<` and `>` separately, select "start of opening or empty markup tag" for `<`, "start of closing markup tag" for `</`, "end of opening or closing markup tag" for `>` and "end of empty markup tag" for `/>`. If you have a scheme element that matches tags entirely and uses a detail subscheme then you need to make the detail subscheme match the angle brackets separately and set the "break and indent" option. Because the scheme element that matched the tag uses a detail subscheme, its "break and indent" setting is ignored. It's important that the "break and indent" rule corresponds with the role of the characters rather than the actual characters. HTML ends empty tags with `>` rather than `/>`. To handle that HTML.jgcscs has separate subschemes for opening tags and empty tags so that `>` can be set as "end of opening or closing markup tag" when inside an opening tag and as "end of empty markup tag" when inside a tag that is supposed to be empty. The difference matters because you'd expect automatic indentation to increase indentation after an opening tag but not after an empty tag.

The user can choose whether automatic line breaks are inserted before and after tags that are part of mixed content inside another tag. The scheme editor has an item in the drop-down menu under the Auto Break button on the preview toolbar. EditPad Pro has this option on the Brackets page in the file type configuration.

If you select "custom rules" then you get two pairs of settings to specify exactly how line breaks and indentation should be added before and after this element's matches. Then the user won't be able to customize auto breaking and indentation in EditPad, other than to completely turn off Auto Break and/or Auto Indent. Schemes can use any mixture of braces, markup tags, and custom rules. Automatic line breaks can never be inserted into the middle of an element's match. Indentation of the next line can be aligned to the middle of a match if the element defines an opening or closing bracket.

- Do not break or indent: Do not automatically insert a line break. Do not change the indentation of manual line breaks.

- Automatically break and indent: If Auto Break is on then automatically insert a line break and adjust the indentation of the new line. If Auto Break is off and Auto Indent is on, apply indentation rule to adjacent manual line breaks.
- Indent if at the start of a line: If Auto Indent is on and the user enters a line break and then immediately enters text matched by this element, adjust the indentation of that manual line break.
- Indent adjacent break: If Auto Indent is on then adjust the indentation of a new line created by manually entering a line break immediately after text matched by this element.
- Indent non-adjacent break: If Auto Indent is on then adjust the indentation of a new line created by manually entering a line break on the same line after text matched by this element if this element is the last one matched on that line that defines any breaking or indentation rules.
- Indent on the same line by one step: If Auto Break is on then insert a tab or spaces to move the cursor one indentation step further than the first non-whitespace character of the line. If the cursor is already at or beyond that column, then do nothing. If Auto Indent is on then add one step of indentation to a new line created by manually entering a line break on the same line after text matched by this element if this element is the last one matched on that line that defines any breaking or indentation rules.

- Keep indentation level: Indent the new line by the same amount as the previous non-blank line.
- Indent by one step: Indent the new line one step more than the previous non-blank line.
- Outdent by one step: Indent the new line one step less than the previous non-blank line.
- Outdent to previous level: Indent the new line by the same amount as the closest line above the previous non-blank line that is indented less than that previous non-blank line. Keep the indentation level if there is no such line.
- Same level as paired opening bracket: If this element's "closing bracket" setting defines a closing bracket at the start or end of the element's match (depending on whether the line break is inserted before or after the match) and that closing bracket can be paired with an opening bracket on a preceding line, then indent the new line by the same amount as the line containing the opening bracket. Keep the indentation level if there is no such line.
- Indent one step from paired bracket: As the previous rule. But indent the new line one step more than the line containing the opening bracket.
- Outdent one step from paired bracket: As the previous rule. But indent the new line one step less than the line containing the opening bracket.
- Outdent one level from paired bracket: Indent the new line by the same amount as the closest line above the line containing the opening bracket that is indented less than that previous non-blank line. Keep the indentation level if there is no such line.
- Align with start of match: Indent the new line by the exact amount needed to make text entered on the new line start at the same column as the text matched by this element.
- Align with end of match: Indent the new line by the exact amount needed to make text entered on the new line start one column after the column on which the text matched by this element ends.
- Align with start of opening bracket: Indent the new line by the exact amount needed to make text entered on the new line start at the same column as the text matched by this element's opening bracket. Align with the start of the match if there is no opening bracket.
- Align with end of opening bracket: Indent the new line by the exact amount needed to make text entered on the new line start one column after the column on which the text matched by this element's opening bracket ends. Align with the end of the match if there is no opening bracket.
- Align with start of closing bracket: Indent the new line by the exact amount needed to make text entered on the new line start at the same column as the text matched by this element's closing bracket. Align with the start of the match if there is no closing bracket.
- Align with end of closing bracket: Indent the new line by the exact amount needed to make text entered on the new line start one column after the column on which the text matched by this element's closing bracket ends. Align with the end of the match if there is no closing bracket.

In EditPad, the Tabbing page in the file type configuration allows the user to configure the size of an indentation step. There is also an option to detect the indentation size of lines around the line that needs to be indented. The preview in the scheme editor always detects indentation size. If a file has no indentation yet then it uses two spaces for indentation.

8. Subschemes

If your scheme is quite complex, you may need to use subschemes. Like the main part of the scheme, a subscheme is a series of scheme elements.

To create a new subscheme, click the **New SubScheme button** on the toolbar. To delete a subscheme, select it in the tree and click the **Delete button**. This button changes its appearance when a subscheme is selected to alert you that you will be deleting an entire subscheme rather than just one scheme element.

You can use the **Cut**, **Copy**, and **Paste** buttons to move or copy entire subschemes. When pasting a subscheme while another subscheme is selected, you'll get a drop-down menu that lets you choose whether the pasted subscheme should be inserted as a new subscheme after the selected one, or whether the pasted subscheme should replace the selected one.

You can use the **Move Up** and **Move Down** buttons to change the order of subschemes. The order is only relevant for "include" subschemes. The final section in this help topic explains that. For all other types of subschemes can be put in whatever order makes it easier for you to keep track of all the subschemes. All subschemes can reference all other subschemes.

Each subscheme must have a **unique name**. Scheme elements reference subschemes by name. Changing the name of a subscheme automatically updates all elements that reference it. Deleting a subscheme does not remove any references to it. So you can cut a subscheme and paste it back without breaking anything. You can make a scheme element reference a subscheme that does not yet exist by typing in the subscheme's name instead of selecting it from the element's "apply detail" or "toggle to" drop-down list. Pasting in an element form another syntax coloring scheme does not alter or remove subscheme references. So if you want to copy and paste multiple subschemes that each have elements referencing other subschemes, you do not need to worry about the order in which you copy and paste the subschemes. All references will be preserved. It is only when you click the Preview button that references must be resolved. (But if they aren't even doing that doesn't break any references to missing subschemes.)

Each subscheme is one of five **subscheme types**. Each type is explained in detail in separate sections below. Subschemes of type "include" have no settings other than their name and their type. All the other subscheme types have the following additional settings.

Subschemes have the same **default** color, spell check, quote conversion, and case adjustment rules for any text that is not matched by any element in the subscheme. These are described in the help topic for the main part of the syntax coloring scheme, which has these too.

Subschemes also have a **highlight color**. This color is applied to all text that is processed by the subscheme, regardless of whether that text is matched by any of the subscheme's elements. The color of the scheme elements then applied on top of this highlight. When the element color is overlaid on the highlight color, the editor checks which parts of the element color are set to "default" in the palette. For those parts, the highlight color is preserved. For other parts, the element color replaces the highlight color. The predefined palettes specify RGB values for the background of the various colors with "highlight" in their name, but not for colors that are named for programming language syntax. So a subscheme can use a color with "highlight" in its name as its highlight color and then use the programming syntax colors for its scheme elements to show programming code with a different highlight. Scala.jgcscs uses this effectively. The subschemes for XML use "markup highlight" as their highlight color and the various colors for markup tags for their elements. The "Scala in XML" subscheme uses "code highlight" as its highlight color while otherwise using pretty much the same elements as the main part of the scheme. This makes it easy to distinguish nested sections of Scala code.

But there is no restriction on which named color you can select as the highlight color for your subscheme. You could set to "Strong", for example, to make all the text processed by the subscheme bold.

Subschemes can override the Block|Comment setting. Leave this blank to use the setting from the general section of the scheme. PHP.jgcscs, for example, has its general Block|Comment set to HTML comments. Its subschemes that highlight PHP code change this to PHP comments for their parts of the file.

Detail

On its own, a scheme element can apply only one color to its regex matches. To apply multiple colors to one regex match, you can add a subscheme of type "detail". You can then select that subscheme in the element's "apply detail" drop-down list.

Doing so overrides all of the element's settings except the brackets and "toggle to". Set the subscheme's default color to the element's color. Then add elements to the subscheme that match the parts that need different colors.

The regular expressions of the scheme elements in a detail subscheme only see the part of the text matched by the scheme element that applies the detail subscheme. This means `\A`, `^`, `\z`, and `$` match at the start and the end of the element being detailed. Lookaround cannot see beyond the element being detailed.

Detail subschemes are **less efficient than toggle subschemes**. Much less so when the subscheme can span multiple lines. When an element applies a detail subscheme the text matched by the element's regex needs to be matched again by the subscheme's regexes. When an edit is made in text colored by a detail subscheme, the entire range colored by that detail subscheme needs to be parsed again. It has to because after the edit there is no guarantee that the regex of the element that applies the detail subscheme would still find the same match. So use detail subschemes only when you can't get the same result with a toggle subscheme.

One situation where a detail subscheme seems an obvious solution is coloring strings with interpolated variables. An element in the main scheme matches the whole string. A detail subscheme then highlights the variables. But this is not efficient. If the string supports escape characters it's even more complicated than using a "toggle" subscheme as explained in the next section.

Perl has very flexible ways of quoting strings. That makes it quite challenging to apply proper syntax coloring. Perl.jgcscs uses the toggle method for interpolated strings as much as possible. The "Double-quoted string" element, for example, just matches the opening double quote and then hands off to a "toggle" subscheme. But the "Interpolated double-slashed string" element is forced to use a "detail" subscheme. This element allows any non-word non-blank character as the string's opening delimiter. So it needs to use a backreference to match the same character as the closing delimiter. This means one regex has to match the entire string including both delimiters. A detail subscheme then picks out interpolated variables.

Another thing that can only be accomplished with detail subschemes to have **opening and closing brackets that have multiple colors**. A bracket must be defined by the match of a single scheme element (or a capturing group in a single scheme element). So then you need one element to match the whole bracket and apply a detail subscheme to give the bracket multiple colors. XML.jgcscs and HTML.jgcscs have "Opening tag" and "Closing tag" elements that match an entire XML and HTML tag and specify the entire tag as an opening or closing bracket. This allows bracket matching to pair up and highlight entire tags. A detail subscheme is then necessary to give attributes and attribute values different colors.

An element inside a “detail” subscheme can also use the “apply detail” to apply a further level of detail. There is no restriction on how deep “detail” subschemes can be nested inside each other.

An element’s “apply detail” setting makes the subscheme override all the element’s setting, except the brackets. This allows for two levels of brackets. HTML.jgcscs and XML.jgcscs make use of this. The subschemes used by the “Opening tag” and “Closing tag” elements separately match the < and > that start and end the tag and specify those as opening and closing brackets too. This gives the scheme two levels of brackets. XML and HTML tags are matched at the base level. < and > are matched at the second level.

Only two levels of brackets are supported. The second level is entered when an element defines a bracket and uses a detail subscheme. Any brackets defined by elements in the detail subscheme are then at the second level. This ensures that brackets added by the element can overlap with brackets added by the detail subscheme. It does not matter whether they actually overlap or not. The second level is entered regardless. It is exited when the detail subscheme finishes.

If the second level was already entered and another element is encountered that defines a bracket and uses another detail subscheme, then any brackets defined by the other detail subscheme are added at the second level if they occur in the text after any previous brackets at the second level. If they overlap or occur before any other brackets then they are ignored.

If an element does not define any brackets and uses a detail subscheme, then there is no possibility of overlap. So the second level is not entered. If the detail subscheme adds brackets, they are added at the base level.

An important restriction is that the two levels of brackets are segregated for bracket pairing and nesting. If brackets need to be paired, you need to make sure they are added at the same level. You cannot force brackets at the second level to be correctly nested inside brackets added at the first level. When looking for a matching bracket, brackets at the other level are totally ignored.

Toggle

With “toggle” subschemes you can switch your syntax coloring scheme between different contexts in the file. The previous section already mentioned that strings with variable interpolation, for example, are best handled with a “toggle” subscheme. In Perl.jgcscs the “Double-quoted string” element simply matches a double quote that starts an interpolated string. This element has “toggle to” set to the “Interpolated string” subscheme. This switches the syntax coloring scheme to the context of being inside an interpolated string. This subscheme has three elements. The element for the closing quote simply matches another double quote character. It has "toggle to" set to "Main coloring scheme". This switches the context back to regular Perl code.

The two other elements in the “Interpolated string” subscheme are included via the “Interpolation” subscheme. The “escape character” element matches any character that is escaped with a backslash, including an escaped double quote. By using a “toggle” subscheme, you can match a string that allows quotes to be escaped with three trivial regexes (opening quote, escaped character, and closing quote). Doing this with one regex is more complicated. Finally, the “variable” element colors the interpolated variables.

If the closing quote is missing, then the subscheme has no elements that toggle back to the main scheme. The subscheme then runs until the end of the file. In case of our Perl scheme, this is perfectly acceptable. In Perl double-quoted strings can span across lines.

When an edit is made to text colored by a "toggle" subscheme, the parser restarts at the last change of color before the start of the line that the edit was on. There is no need to back up further to where the subscheme was toggled to. The parser can see the context it is in from the existing colors.

Toggle subschemes can contain elements that toggle to other subschemes. They can have multiple elements that toggle to different subschemes. A "toggle" subscheme can even have an element that toggles to itself. This is not the same as an element that does not toggle at all. Only make a subscheme toggle to itself if you need the recursion, as explained below.

There is a special "toggle to" choice labeled "**toggle back to previous subscheme**". This makes the element toggle back to whichever subscheme that toggled to the present subscheme. This is very useful if you have elements in multiple subschemes all that toggle to the same subscheme. The subscheme toggled to can then toggle back to any of those multiple subschemes simply by having one element set to "toggle back to previous subscheme". Delphi.jgcscs uses this. The main part of the scheme, the "Property" subscheme and the "Asm" subscheme all include the "Common" subscheme. This means that the main part and the 2 subschemes all include the "(* comment" element which toggles to the "(*Comment*)" subscheme. That subscheme just has one element set to toggle back to the previous subscheme. This way Property and Asm blocks continue correctly after the comment.

This toggling and toggling back mechanism can be nested arbitrarily deep. Each time a subscheme toggles to another that toggle is added to a stack. When a subscheme wants to toggle back to the previous subscheme, the most recent toggle is popped from the stack. This is why an element toggling to its own subscheme is different to one that doesn't toggle at all. When a subscheme toggles to itself, that toggle is added to the stack too. This makes toggle subschemes a very efficient way to match recursive constructs.

In Scala, multi-line comments can be nested. Scala.jgcscs handles this with the "Multi-line comment" element in the main part of the scheme which toggles to the "Comment" subscheme. This subscheme has two elements. The "Nested comment" element matches another `/*` and toggles to the "Comment" subscheme. The "Closing */" element toggles back to the previous subscheme. This could be the main part of the scheme or the "Comment" subscheme itself. Three scheme elements with trivial regexes is all it takes to perfectly handle Scala's comments nested arbitrarily deep. Though you could handle this with a recursive regular expression, doing so would require a complicated regex that wouldn't perform nearly as well as the subscheme with two simple regexes.

Toggle Until End of Line

Having a "toggle" subscheme automatically disqualifies a syntax coloring scheme from being a "fast" scheme. As explained in the previous section, there's nothing to guarantee that a toggle subscheme actually toggles back at a certain point or at all.

The "toggle until end of line" subscheme type resolves this. These subschemes have an implied regex that matches a line break. When this regex matches, the subscheme toggles to the main part of the scheme. It always toggles to the main part of the scheme. It does not toggle back to whichever subscheme that toggled to the present subscheme.

This subscheme type is specifically intended to allow the toggle mechanism to be used in "fast" syntax coloring schemes. By always toggling to the main part of the syntax coloring scheme, the parser can know that it can begin with the main part of the scheme at the start of any line without having to take any preceding

lines into account. The requirement for “fast” schemes that none of the elements can have regexes that can match any line breaks does remain.

If a “toggle until end of line” subscheme contains a regex that matches line breaks, then the subscheme can span across lines. This is not a problem. It then simply toggles back to the main scheme when it encounters a line break that is not part of an element’s match that started before the line break. The implied regex takes precedence over all elements. But it does not force elements to stop at the first line break.

Toggle within Detail

This is another variation on the “toggle” subscheme type. A “detail” subscheme cannot have elements that toggle to a “toggle” or “toggle until end of line” subscheme. But “detail” subschemes can have elements that toggle to “toggle within detail” subschemes.

The only real difference is that a “toggle within detail” subscheme automatically stops where the “detail” subscheme was to stop. A “toggle within detail” subscheme can use “toggle back to previous subscheme” to go back to the “detail” subscheme. But it doesn’t need to. The run of the “detail” subscheme and anything it toggles to always ends where it should. The “toggle within detail” subscheme also inherits the positions where `\A` and `\z` matched in the “detail” subscheme.

A “toggle within detail” subscheme can toggle to other "toggle within detail" subschemes. These can toggle back and forth without restriction. They can also have elements that apply other “detail” subschemes.

Include

An “include” subscheme is never used directly. It is simply a reusable collection of scheme elements. Subschemes of all other types have an “include” box among the subscheme’s settings. This box lists all the “include” subschemes that you added to your syntax coloring scheme. Ticking any subschemes in the “include” box adds their elements to the present subscheme.

If multiple subschemes in your syntax coloring scheme need to use the exact same scheme elements, consider putting those elements in an "include" subscheme. Then have those multiple subschemes include that scheme. This makes your scheme easier to maintain when some of the shared elements need to be changed.

There is no performance benefit to using “include” subschemes. Internally, the scheme elements are duplicated into the subschemes that include the “include” subscheme.

The first section in the help topic about the main part of the scheme explained that the order of scheme elements in a subscheme matters when those elements can match the same text. Elements from included subschemes are added after the elements of the subscheme that includes them. So the elements in a subscheme that is not of type “include” always take precedence over elements from included subschemes. It does not matter whether the included subscheme is placed above or below the subscheme that includes it.

But if a subscheme includes mutiple subschemes that have elements that may match the same text, then the order of those included subschemes matters. Elements from an included subscheme that is higher in the list take precedence over elements from an included subscheme that is lower in the list.

The "toggle to" setting of a scheme element is restricted depending on the type of subscheme that uses the element. A "toggle within detail" subscheme can only be toggled to from elements in a "detail" or a "toggle within detail" subscheme. A "toggle" or "toggle until end of line subscheme can only be toggled to from elements in those types of subschemes or from the main scheme. If you need to color the same syntax inside and outside "detail" subschemes, then you will need two sets of subschemes with two sets of toggling elements. Only elements that don't toggle can be inside a shared "include" subscheme. If an element in an "include" subscheme doesn't follow the rules of the subscheme that includes it, then clicking the Preview button selects that element inside the "include" subscheme, because that's the only place it is visible in the scheme editor. But the error message will indicate the name of the subscheme that actually includes the element by including its subscheme.

PHP.jgcscs is an example of this restriction leading to some duplication of elements. It has two sets of subschemes for highlighting PHP code. One set is toggled to from the main part of the scheme. The other set is toggled to from the "detail" subschemes that highlight HTML tags. There is no other way to handle this if bracket matching needs to be able to match up entire HTML tags and to also highlight PHP code inside HTML tags. If bracket matching did not need to match up entire HTML tags, then HTML tags could be colored with a "toggle" subscheme which could then toggle to the same PHP subscheme for nested PHP code as the main scheme.

9. Regular Expressions

Regular expressions are used for searching through (usually textual) data. They allow you to search for pieces of text that match a certain form, instead of searching for a piece of text identical to the one you supply. For example, the regular expression `[0-9]+` allows you to search through a file for any integer number.

It takes some time to get used to the syntax used by regular expressions. In our example, we used square brackets to create the character set `[0-9]`. This character set matches any character that is a digit. The `+` means that the character set must be matched as many times as possible, and at least once. EditPad's help file and printable manual contain a detailed tutorial to regular expressions. The tutorial clearly explains the entire regular expression syntax. It explains the exact regex flavor used by EditPad and the scheme editor. AceText has the same documentation in its help file but not in its printable manual.

Regular expressions are not only useful for building your own syntax coloring schemes. You can also use them for powerful search and replace operations in EditPad. Make sure the Regex button is down and you can unleash all their power upon your files. In AceText you can search and replace with regular expressions if you click the Search Options button and then tick the Regular Expression checkbox. If you're a developer, your programming language very likely has its own regex library. It may even support regexes as a language feature.

If you want more assistance with creating and editing regular expressions, take a look at our product RegexBuddy at https://www.regexbuddy.com/. You can launch RegexBuddy right from within the scheme editor by clicking the RegexBuddy button on the main toolbar. This launches RegexBuddy and transfers the regular expression of the active scheme element and the text you have in the preview editor to RegexBuddy. This allows you to quickly test and debug this regex in isolation in RegexBuddy. After fixing your regex in RegexBuddy you can click the Send button in RegexBuddy to transfer your regex back to the scheme editor. EditPad's Search panel and AceText's Search toolbar have a similar button that allows you to prepare a regular expression for EditPad or AceText just as easily.

RegexBuddy can accurately emulate the regular expression flavors of a wide variety of applications and programming languages. It has specific support for the regex flavor used by the syntax coloring system. You can select EditPad 8 or AceText 4 as your application in RegexBuddy. In RegexBuddy's documentation, this regex flavor is known as "JGsoft V2".

10. Working with Named Colors

EditPad's and AceText's syntax coloring can be configured at two levels. The syntax coloring scheme, which you create in the syntax coloring scheme editor, you determine which parts of the file are colored as keywords, comments, strings, etc. You'll select named colors in the scheme editor, rather than actual red-green-blue colors.

The color palette assigns actual red-green-blue colors to the named colors. In the scheme editor, you can test with different palettes in the preview area. In EditPad, you can select a palette as part of the file type configuration. You can have a different palette fore ach file for each file type. In AceText, all clips are displayed using the palette you select it in the Appearance Preferences. But you can select another palette in AceText when printing or exporting a collection. You can configure AceText to use a different palette for each target application when sending clips as HTML or RTF.

The main reason for these two levels is that **color preferences** can be **highly personal** and depend on the **circumstances**, such as a dark palette for display and black on white for printing. But creating or even customizing syntax coloring schemes can be quite complicated. By separating color choice from syntax definition, anybody can easily create their own color palettes and use them with any type of file.

For this system to work well, it's important that your syntax coloring scheme uses the named colors as close to their intended purpose as possible. Do not select a particular named color for a scheme element just because it happens to be red, green, or blue in the default color palette. By using the appropriate named colors, people can create one color palette that makes all their files, including the one colored by your scheme, look nice and consistent.

If you think there aren't **enough named colors**, then you're probably trying to make your scheme way too detailed. Most of the schemes included with EditPad and AceText only use a subset of the available named colors. Remember that the purpose of syntax coloring is to make the text easier to navigate by the eye. Making certain things like keywords stand out helps to visualize the structure of the text. Giving everything a different color makes nothing stand out. It can even make the text harder to read than if no syntax coloring was applied at all. It's perfectly fine if non-structural parts of the file all have the same color. Most of the programming language schemes included with EditPad do not highlight any identifiers. Types, functions, variables, and constants are all highlighted as plain text. Though there are named colors available for all of these, using them all would not make the file easier to read.

Some of the predefined palettes even assign the exact same colors to multiple named colors. The JGsoft Classic and the Embarcadero palettes emulate older versions of EditPad and the Delphi/C++Builder IDE that support far fewer colors than the latest EditPad or AceText. These palettes may use the same colors for elements that your scheme tries to differentiate. Again, that's perfectly fine. People who choose to use those palettes are likely used to not seeing too many different colors in their text editor or IDE and may find the new default palette in EditPad or AceText makes your scheme far too "busy".

Your scheme should not require a specific color palette. It should work well with the palettes that are included with EditPad and AceText. It's not a problem if some elements get the same colors with some palettes as long as the more structural elements of the syntax stand out. Most of the programming language schemes included with EditPad and AceText have separate elements for integer and floating point numbers that use the corresponding named colors. But most of the palettes included with EditPad and AceText assign the exact same RGB values to these named colors.

One good reason to create a custom palette is to emulate the colors of an application that is commonly used to edit your type of file. But that palette should work as well as possible with the syntax coloring schemes included with EditPad. This way people who mainly use EditPad for your type of file can use your palette with your scheme and also use the same palette for other types of files.

Though EditPad allows a different palette to be selected for each file type, users shouldn't have to spend time customizing too many palettes. Ideally, they would spend time to create one palette that perfectly suits their personal tastes and then be able to use it for all their files.

These are the named colors available in the scheme editor:

- Plain text: Body text of the file, part of the file without special function, or identifiers in a programming language.
- 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.
- 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.
- Emphasis: Text in a document that will appear in italics when the document is printed or published.
- Strong: Text in a document that will appear bold when the document is printed or published.
- Strong emphasis: Text in a document that will appear bold and in italics when the document is printed or published.
- Underline: Text in a document that will appear underlined with the document is printed or published.
- Redacted: Text that will not appear at all when the document is printed or published.
- Label: An indicator that does not need emphasis, such as a fixed label before the actual data.
- Caption: An indicator that can use some emphasis, such as a subsection header or a title.
- Markup tag: Opening or closing tag in markup languages.
- Markup tag attribute: Attribute name in markup languages.
- Markup tag attribute value: Attribute value in markup languages.
- Markup tag delimiter: Punctuation that starts or ends a tag in markup languages.
- Markup entity: An entity name or a numeric character reference in markup languages.
- Comment: Human-readable text used for information only.
- Documentation comment: Human-readable text of particular importance, used for information only.
- Preprocessor statement: Any kind of meta-information, such as compiler and preprocessor directives.
- Reserved word: Words or character combinations with a specific meaning and specific use, such as keywords in a programming language.
- Function name: The name of a (built-in) function call in a programming language, or a reference in a document.
- 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.
- Type name: The name of a (built-in) type or class in a programming language.
- Constant name: The name of a (built-in) constant in a programming language, or a fixed placeholder in a document.
- Constant value: A literal value that does not fit one of the following literal types.
- Character: A single human-readable character (letter). Can also be used for escaped characters within character strings.
- Character string: Human-readable text to be processed by software.
- Text pattern: A text pattern such as wild cards or a regular expression.
- Integer number: A whole number. Could be decimal, hexadecimal, octal, or binary.
- Floating point number: A number with a fractional part and/or an exponent.

- Date and time: Any date or time.
- 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.
- Operator: Any kind of mathematical operator or other symbol with a specific meaning or effect.
- Bracket: Round, square, or curly brackets used in expressions, such as parentheses and square brackets in C-style languages.
- Structural brackets: Round, square, or curly brackets used to group statements or items, such as curly braces in C-style languages.
- Section header: Heading that starts a new section.
- Success message: Message in tool output, a log file, or a report indicating successful completion.
- Hint message: Informative hint message in tool output, a log file, or a report.
- Warning message: Non-fatal warning message in tool output, a log file, or a report.
- Error message: Fatal error message in tool output, a log file, or a report.
- Markup highlight: Highlight color for subschemes that highlight markup in files that do not predominantly consist of markup.
- Code highlight: Highlight color for subschemes that highlight procedural code in files that do not predominantly consist of procedural code.
- 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.

If you must create a custom palette because your file format uses concepts that can't be reasonably matched to these named colors, then the 10 fountain colors are the colors you should cannibalize first.

11. RGB Preview

You can set "apply detail" of any scheme element to "RGB preview" to make the regex match determine the color of the scheme element. Unlike actual "detail" subschemes, "RGB preview" only changes the element's color. It does not override or disable any other of the element's settings.

The element's named color is used as a fallback if the preview failed or if the user has disabled it. RGB preview can be disabled in the file type configuration in EditPad. It cannot be disabled in the scheme editor or in AceText.

For RGB preview to work, the scheme element's regular expression needs to contain **specific named capturing groups**. If it does not, clicking the Preview button in the scheme editor gives you an error message that tells you which named capturing groups are needed.

1. A group named "CSS" that matches a CSS color constant like #639 or #663399 or rebeccapurple.
2. A group named "RGB" that matches a 24-bit hexadecimal number like FFA000 for orange.
3. A group named "BGR" that matches a 24-bit hexadecimal number like 00A0FF for orange.
4. 3 groups named "R", "G", and "B" that match hexadecimal numbers between 0 and FF.
5. 3 groups named "Rd", "Gd", and "Bd" that match decimal numbers between 0 and 255.
6. 3 groups named "Rp", "Gp", and "Bp" that match decimal numbers between 0 and 100.

Remember that capturing group names are **case sensitive**. Leading zeros are optional when using 3 groups to separate the color components. Numbers between 0 and FF or 255 are interpreted as 8-bit RGB values. Numbers between 0 and 100 are interpreted as percentages.

The regular expression needs only needs one of the six groups or group trios in the list. If it has one of the groups from a trio, then it must have all three. The regular expression may have multiple of the six groups or trios in the list. In that case, for each regex match, the group(s) highest in the list that actually matched anything is used. If the match of that group or those 3 groups does not fit the expected format, then the RGB preview fails and the element's named color is used.

If the RGB preview succeeds, then the background color of the element is set to the RGB values retrieved from the regex match. The text color is set to a color that contrasts well with the background color to keep it readable. The shade of the text color will match the shade of the background color. This particularly helps to distinguish between different dark color shades that would otherwise be hard to distinguish behind the brighter text.

CSS.jgcscs uses RGB preview to highlight anything that looks like a CSS color reference in CSS properties. HTML.jgcscs does the same in HTML attribute values.

12. Path Placeholders

Path placeholders can be used for the action parameters in scheme elements. The full path they are expanded from is the full path to the file that is being colored using your syntax coloring scheme. The file C:\data\files\web\log\foo.bar.txt is used in this example.

It is possible that an unsaved file is being colored. In that case, all the placeholders below will be replaced with nothing. In EditPad Pro, the file can also be part of a project. In that case, you can use %PROJECTFILE%, %PROJECTPATH%, etc. to use part of the path to the project file. Again, if the project is unsaved, these placeholders are replaced with nothing. You can mix file and project placeholders as you like.

You can use these placeholders to convert relative paths to absolute paths. If you have a scheme element that uses the regex `include "([a-zA-Z0-9.]+)"`, you can specify `%PATH%\\\1` as the file to be opened. The first two backslashes are replaced by a single backlash, and `\1` is replaced by the text matched by the capturing group in the regex.

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.	*(nothing)*
%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.	*(nothing)*
%PATH1%	First folder in the path, without delimiters	data
%PATH2%	First two folders in the path, without	data\files

	leading or trailing delimiters	
(...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.	*(nothing)*
%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.	*(nothing)*

13. Legacy Brackets

For new syntax coloring schemes, you should define brackets as part of the coloring scheme instead of defining them separately in the Legacy Brackets section. The Legacy Brackets section is still available for backwards compatibility with old syntax coloring schemes. The settings in the Legacy Brackets section are only used if none of the elements in the coloring scheme or subschemes define any brackets. Legacy Brackets do not support automatic insertion of matching brackets.

Any pair of characters can be used as brackets. Brackets can consist of multiple characters. If two or more pairs of brackets start with the same character(s), enter the longest of the two first. The list of brackets is processed from top to bottom when the software tries to determine which bracket the text cursor points at.

Specify the character(s) that are the opening and closing brackets of a bracket pair. These character sequences are used as they are. You cannot use regular expressions to define legacy brackets. If you want to use regular expressions, define your brackets as part of the syntax coloring scheme instead of using the Legacy Brackets section.

The nesting checkbox determines whether the brackets nest or not. The nesting function only looks for opening and closing brackets of the pair it is trying to match. It does not take into account other kinds of brackets that you may have defined. If you want other kinds of brackets to be taken into account, define your brackets as part of the syntax coloring scheme instead of using the Legacy Brackets section and use the "bracket nesting" setting to define which bracket pairs should be correctly nested within each other.

Finally, select the range the software should search through when looking for the bracket that matches the one under the text cursor. If an opening bracket is under the cursor, the software only searches the text after the bracket. In case of a closing bracket, only the text before the closing bracket is searched. If you select "everywhere", all text before or after the bracket is searched. Otherwise, only text that has particular colors applied to it is searched.

- Within a single element: The software will only search inside the block of text that contains the bracket under the cursor, and was matched by a single scheme element. It will not search through other blocks of text matched by the same scheme element. For example, if you have a scheme element that matches a string, only brackets that can be matched inside a single string will be highlighted.
- At the edges of a single element: A pair of brackets will only be highlighted if the opening bracket can be found at the very start of a single block of text matched by a scheme element, and the closing bracket at the end of it. For example, if you have a scheme element that matches a string, you can use this option to match the quote characters at the edge of the string as brackets. In this situation, the opening and closing bracket are usually identical (i.e. a single quote or a double quote). The "at the edges" options are the only way to make sure the quotes are properly paired.
- Throughout identical elements: Both the opening bracket and the closing bracket must have been highlighted by the same scheme element. Unlike the "within a single element" option, the brackets do not need to be inside the same block of text. If the scheme element highlighted several blocks of text, all of them are searched.
- Within a single subscheme: The software will search through a single block of text that was colored by scheme elements in the same subscheme (or the main part of the scheme) as the element that colored the bracket under the cursor. The software stops searching when it encounters text colored by scheme elements from another subscheme.

- At the edges of a single subscheme: Like "within a single subscheme", except that the brackets must be found at the edges of the block of text colored by a single subscheme. There is one exception. The opening bracket may be colored by a scheme element of another subscheme (or the main scheme) that toggled to the subscheme highlighting the block. For example, if you have an element in the main scheme matching a quote character, and that element switches to a subscheme for highlighting a string until the next quote, you can match the opening and closing quote of a string with the "at the edges of a single subscheme" option.
- Throughout identical subschemes: The software will search through any text that was colored by any scheme element that is part of the same subscheme as the element that colored the bracket under the cursor.