

AlteNET Code Editor for WinForms v7

Contents

| | |
|--|-----------|
| Introduction | 2 |
| What's included | 2 |
| Installation | 3 |
| Getting Started | 4 |
| <i>Syntax Parsing</i> | 7 |
| <i>C#/VisualBasic (Roslyn) and TypeScript/JavaScript parsers</i> | 8 |
| <i>LSP Parsers</i> | 8 |
| <i>Advanced Parsers</i> | 8 |
| <i>Generic parsers</i> | 9 |
| <i>Code Completion</i> | 9 |
| <i>Automatic Code Completion Invocation</i> | 9 |
| <i>Code snippets</i> | 10 |
| <i>Code Outlining</i> | 12 |
| <i>Direct Definition of Outline Sections</i> | 12 |
| <i>Indirect Definition of Outline Sections Using the Syntax Parser</i> | 12 |
| <i>Structure GuideLines</i> | 13 |
| <i>Selection</i> | 14 |
| <i>Searching and Replacing</i> | 15 |
| <i>Visual Themes</i> | 17 |
| <i>Gutter</i> | 17 |
| <i>Bookmarks</i> | 19 |
| <i>Keyboard Mapping</i> | 20 |
| <i>Spellchecker Interface</i> | 23 |
| <i>URL handling</i> | 23 |
| <i>Printing and Exporting</i> | 23 |
| <i>Miscellaneous Options</i> | 25 |
| <i>White-space Display</i> | 25 |
| <i>Line Separator</i> | 25 |
| <i>Scroll Bars and Split View</i> | 25 |
| Advanced Topics | 26 |
| <i>Automatic Code Completion for arbitrary programming language</i> | 26 |
| <i>Manual Code Completion</i> | 27 |
| <i>Quick Info</i> | 27 |
| <i>List Members</i> | 29 |
| <i>Parameter Info</i> | 31 |
| <i>Localization of dialogs</i> | 32 |
| <i>Page Layout mode</i> | 32 |
| <i>Marco Recording and PlayBack</i> | 32 |
| <i>Global Settings</i> | 32 |

| | |
|---|-----------|
| <i>Creating a New Generic Parser</i> | 34 |
| Integration with other AlterNET Studio components. | 36 |
| Licensing | 36 |

Introduction

AlterNET Code Editor is a component library that brings efficient code editing functionality to your .NET applications. It provides code editing capabilities such as syntax highlighting, code completion and code outlining, visual indicators for bookmarks, line styles, syntax errors and much more, matching speed and convenience of Microsoft Visual Studio editor.

What's included

The main component in the package is *SyntaxEdit*. This control provide text editing functionality and support almost all the features that can be found in the Visual Studio.NET code Editor, including customizable syntax highlighting, code outlining, code completion, unlimited undo/redo, bookmarks, word wrap, drag-n-drop, built-in search/replace dialogs, multiple view of the same text, displaying gutter, margin, line numbers and many more.

Roslyn-Based C# and Visual Basis parsers

C# and VB.NET parsers are based on .NET Compiler Platform ("Roslyn") which provides open-source C# and Visual Basic compilers with rich code analysis APIs:

<https://github.com/dotnet/roslyn>

Utilizing these APIs allows advanced code-editing features, such as code completion, finding references and declarations, code outlining and highlighting syntax warnings and errors, to be equivalent to Microsoft Visual Studio code editor.

TypeScript and JavaScript parsers

TypeScript is a superset of JavaScript that compiles to clean JavaScript output. TypeScript and JavaScript parsers are based on Microsoft TypeScript Compiler which provides open-source TypeScript and JavaScript syntax compilers and checkers with rich code analysis APIs:

<https://github.com/microsoft/TypeScript>

Utilizing these APIs allows advanced code-editing features, such as code completion, finding references and declarations, code outlining and highlighting syntax warnings and errors, to be equivalent to Microsoft Visual Studio Code text editor.

LSP Parsers

LSP code parsers implement syntax analysis are based on LangServer:

<https://langserver.org/>

LSP servers provide features like auto completion, go to definition, find all references and alike.

C/C++, Java ,Python, Lua XML, and PowerShell parsers are available in the package.

XAML parser

XAML parser implements full syntax analysis of the XAML code used for interface building in WPF applications. It provides basic code completion for most commonly used WPF controls.

Generic syntax highlighting parser

Generic *Parser* component, being based on a regular expression engine, is designed to perform syntax highlighting for an unlimited number of programming languages.

It comes with syntax schemes for more than 30 programming languages such as C, C#, VB, Java, Xml, Html, Python and many others.

Advanced Parsers

Advanced code parsers implement full syntax analysis for vast subset of modern programming languages, including Python, C#, VB.NET, Java, Ansi-C, JavaScript, VBScript, HTML and XML.

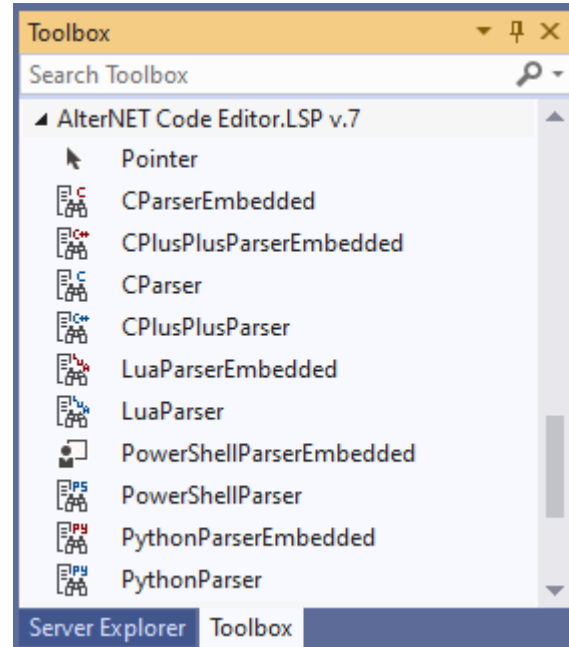
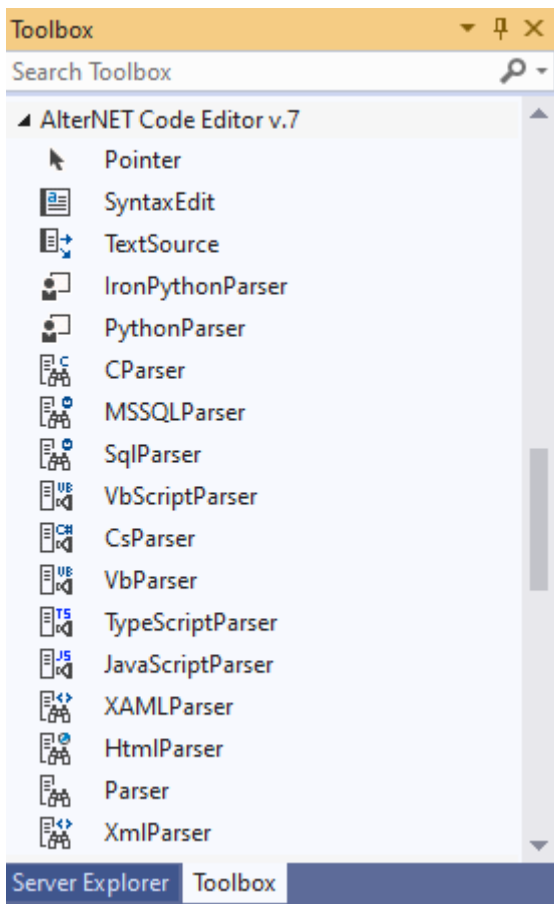
Demo and QuickStart projects – these projects show how to use various Code Editor features.

Full Source Code - which comes with Universal edition.

Installation

AlterNET Code Editor is installed as part of the AlterNET Studio installer program. Advanced installation options include platform selection (WinForms, WPF or both), and optional support for LangServer components.

Installation requires .NET Framework 4.6.1 + and Visual Studio 2015, 2017 or 2019 to be installed on the target machine. Installation program will register Visual Studio extensions and place SyntaxEdit, TextSource, and Parser components on the AlterNET Code Editor tab in the Visual Studio toolbox.



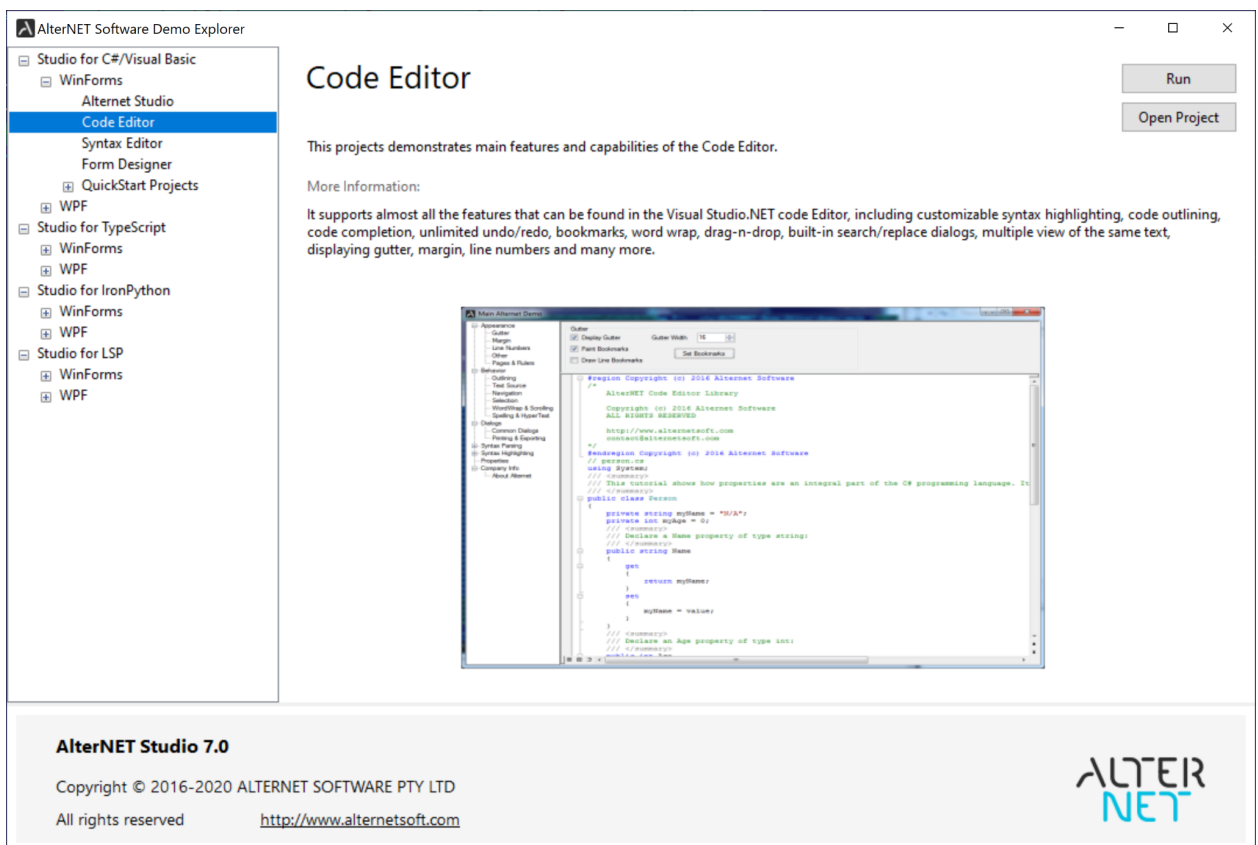
Other versions of .NET Framework 4.5.2,.NET Core 3.0,+ and Net 5.0 are available via NuGet packages. Complete list of NuGet packages can be found here:

<https://alternetsoft.com/download>

If you have a previous major version of AlterNET Studio, and decide to install the new one side-by-side, you will have two sets of Visual Studio Extensions and two sets of tabs, each one clearly displaying version number.

Getting Started

Once the product is installed, it might be a good idea to explore quick start projects and Code Editor demo first, either by compiling Alternet.Studio.AllDemos.sln projects or accessing these demos through the Demo Explorer tool which is added to Windows Start menu.



Below is brief overview of these projects:

Roslyn-Based Parsing – Shows how to link text edit control to Microsoft Roslyn-based parsers that perform full syntax and semantic analysis of the C# or Visual Basic code and provide features like code completion, code outlining and syntax/semantic error highlighting.

TypeScript Parsing – Shows how to link text edit control to Microsoft TypeScript-based parsers that perform full syntax and semantic analysis of the TypeScript or JavaScript code and provide features like code completion, code outlining and syntax/semantic error highlighting.

Advanced Syntax Parsing – Shows how to link text edit control to parsers that perform syntax analysis for a set of programming languages and provide features like code completion, code outlining and syntax error highlighting.

Snippet Parsers – Shows how to implement C# or Visual Basic syntax and semantic analysis for sub-set of the code, like class or method body.

SQL DOM Parser – Shows how to implement syntax analysis for Microsoft SQL.

XAML Parser – Shows how to implement syntax analysis for XAML.

LSP-based parsers (C/C++, Java ,Python, Lua XML, and PowerShell) – Shows how to implement syntax analysis for these languages using native servers.

Lsp Multiple Files – Shows how to combine multiple LSP documents into a single workspace.

Python Parsing – Shows how to implement syntax analysis for Python and IronPython

Scroll Bar Annotations – Shows how text edit control can display markers about current line, syntax errors, bookmarks, modified lines and search results on the vertical scrollbar area.

Syntax Highlighting – Shows how text edit control can highlight syntax when working with different programming languages.

Code Completion – Shows how to display code completion while you type; either by getting code completion information from parser, or programmatically.

Code Outlining – Shows how to use outlining; either provided by parser, or programmatically.

Selection – Shows how to use different options to control text selection appearance and behavior in the text editor.

Undo/Redo – Shows how to use various options to control undo/redo behavior.

Search and Replace – Shows how to use built-in Search and Replace dialogs, and how to implement search across multiple documents.

Gutter – Shows how to control the appearance of the gutter area and how to display various indicators on it.

Bookmarks – Shows how to set and navigate through numbered and though-loop bookmarks and how to set bookmarks navigation across multiple documents.

Word Warp – Shows how to configure text edit control to wrap words at the right-edge of the visible area or at a given position.

Line Styles – Shows how to display line indicators on text editor control area and associated images on the gutter.

Print and Preview – Shows how to print and preview text editor control content and how to set different printing options.

Code Snippets – Shows how to display and use predefined code templates to speed-up entering frequently used fragments of code

Multiple Views and Split View – Shows how to configure text editor windows to display and edit the same text content.

Margin – Shows how to use various options controlling the appearance and behavior of Margin line and UserMargin area next to gutter.

Hyper Text – Shows how to highlight hyperlinks in the text.

Page Layout – Shows how to configure text edit control to display its content as if it was positioned on the printed page.

Miscellaneous – Shows how to display white-space symbols and background images, as well as highlight matching brackets.

Customize – Shows example of options dialog that allows changing display settings of text edit control.

Once you've done with demo projects, it's time to create your own project with text editing capabilities.

The first thing to do after creating a new application is to place the *SyntaxEdit* control. This is the central component in the package, and in many cases it will be the only one that you explicitly create by placing it on the form. At the first glance, you can tell that it looks similar to the standard multi-line text box, with the exception of having a gray band on the left of its client area. This region is called gutter, and it will be discussed later.

Many properties of this control are similar to those of the *TextBox*, some of them will be new; however there probably will be many intuitively understandable ones. Now, to have some basic functionality, we need to learn how to load text into the editor and how to save it.

The simplest way to load text from the file is by using the *LoadFile* method. The first one specifies the name of the file to be loaded into the control. The optional second parameter specifies encoding.

```
syntaxEdit.LoadFile(openFileDialog1.FileName);
```

Saving of the text is performed in a similar way:

```
syntaxEdit.SaveFile(saveFileDialog1.FileName);
```

You can use streams instead of files, by substituting the previous two functions by *LoadStream* and *SaveStream*.

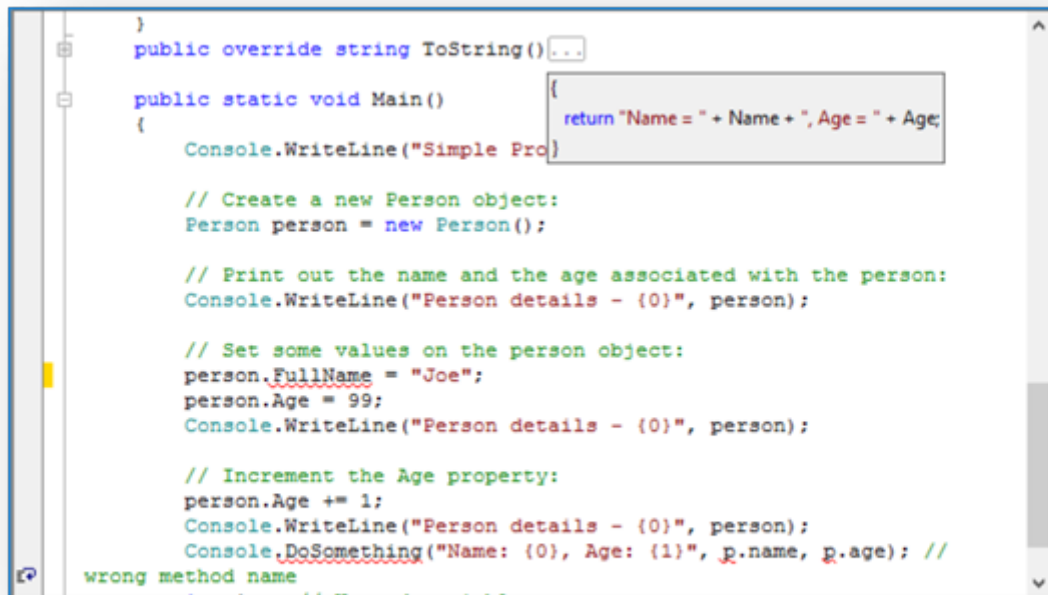
Now, if you add some UI to load and save files, you can have a simple functional text editor application.

Before we go any further exploring features of the text edit control, we must examine one of its fundamental concepts.

The other components in the package are non-visual ones: *TextSource* and various *Parsers*. The components are also accessible through Microsoft Visual Studio toolbox. The *SyntaxEdit* control itself does not store the text it edits. This task is offloaded to the *TextSource* component. Even when the *Source* property of the *SyntaxEdit* control is not assigned, the latter contains an internal instance of the *TextSource* component. It gives clear separation between visualization and data layers, and also makes it possible to implement a feature which would not be possible otherwise: multiple views of the same text. What it means is that by simply assigning a single *TextSource* to multiple text editors you can have them work with the same text. Visually, these editors can be either placed in a single window separated by a splitter control or in multiple windows. Most of the methods of *TextSource* components are also exposed via the *SyntaxEdit* control itself.

Syntax Parsing

Text parsing is performed by one of the *Parser* non-visual components. In the very basic version it controls the syntax highlighting, and in case of using more advanced parsers it enables additional features such as code completion, code outlining, code formatting and syntax error underlining.

A screenshot of a code editor window displaying C# code. The code includes a `ToString()` method, a `Main()` method, and various comments and console output statements. A tooltip is visible over the `ToString()` method, showing its implementation: `return "Name = " + Name + ", Age = " + Age;`. The code is color-coded: keywords in blue, comments in green, and strings in red. A small error message "wrong method name" is visible at the bottom left of the editor window.

```
    }  
    public override string ToString() {  
        return "Name = " + Name + ", Age = " + Age;  
    }  
  
    public static void Main()  
    {  
        Console.WriteLine("Simple Pro)  
  
        // Create a new Person object:  
        Person person = new Person();  
  
        // Print out the name and the age associated with the person:  
        Console.WriteLine("Person details - {0}", person);  
  
        // Set some values on the person object:  
        person.FullName = "Joe";  
        person.Age = 99;  
        Console.WriteLine("Person details - {0}", person);  
  
        // Increment the Age property:  
        person.Age += 1;  
        Console.WriteLine("Person details - {0}", person);  
        Console.DoSomething("Name: {0}, Age: {1}", p.name, p.age); //  
    }  
wrong method name
```

If no parser is assigned, *SyntaxEdit* does not perform any parsing related functions. To be able to use those features you need to explicitly create a *Parser* component and assign it via the *Lexer* property of either the *SyntaxEdit* control, or the *TextSource*.

C#/VisualBasic (Roslyn) and TypeScript/JavaScript parsers

The package includes parsers that are based on Microsoft Code Compiler technology (Roslyn) and Microsoft TypeScript compiler.

These parsers allows to have full syntax and semantic model of the text being edited by *SyntaxEdit* control, which enables additional features such as code completion, code outlining, code formatting, highlighting types in a different colors and underlying syntax and semantic errors and warnings to be identical to the ones found in Microsoft Visual Studio editor.

LSP Parsers

LangServer-based parsers rely on external servers to provide features like auto complete, go to definition, find all references and alike. The Code Editor package includes parsers based on this technology for C/C++, Java ,Python, Lua XML, and PowerShell. There are two variations of each parser - one that relies on the Language server to be installed on the target machine and one that includes all required payload (such as clang libraries for c/c++ or embedded python distribution) in the form of embedded resources. Java embedded parser contains LSP-server files, but not Java installation itself, which need to be installed on target machines independently.

Advanced Parsers

The Code Editor package comes with several advanced parsers, each one designed to perform syntax highlighting for certain languages. Each of these parsers is derived from the *SyntaxParser* class implementing *ISyntaxParser* interface and performs syntax analysis of the text in a specific programming language in order to provide advanced code editing features discussed above. These parsers use hard-coded parsing algorithms instead of generic regular-expression based rules, which makes them significantly faster compared to generic parsers (these parsers will be explained further). Currently we have advanced parsers for the following languages: Python, C#, J#, Visual Basic.NET, Ansi-C, VBScript, JavaScript, HTML SQL, T4 and XML. These parsers perform complete syntax parsing of the source code to

build the syntax tree, which is used to implement all mentioned features. Please note that these parsers might not support full language specification, especially language constructs which were added to these programming languages recently.

Python and XAML parsers are implemented in their own namespaces/assemblies, `Alternet.Syntax.Parsers.Advanced.Python` and `Alternet.Syntax.Parsers.Xaml` respectively.

Generic parsers

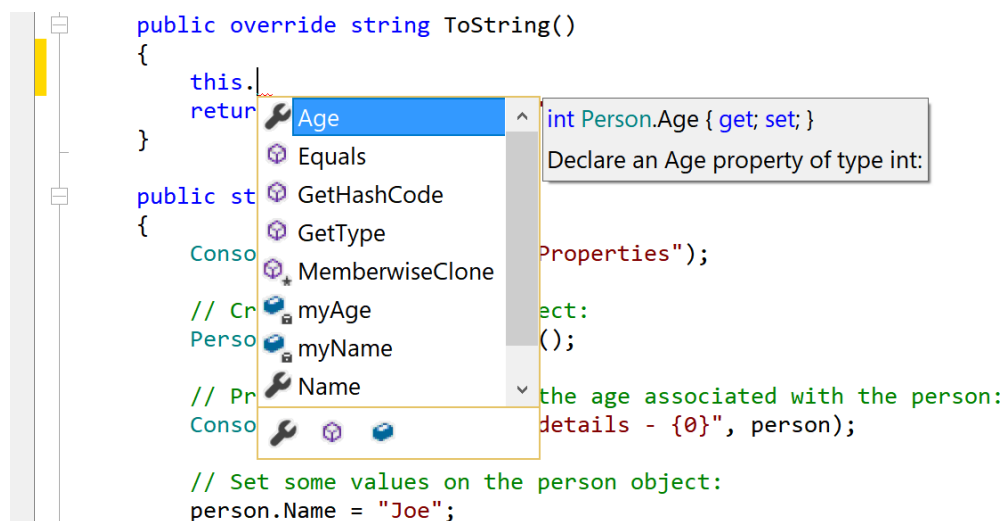
Writing your own parser to comply with a full specific programming language is non-trivial task, not to mention of keeping it up to date with full language specifications as the language evolves, however for a simple task of syntax highlighting it's normally not required. To perform syntax highlighting you can rely on a generic parsing engine, which is using finite-state automaton rules driven by regular expressions matching the parsed text. Although creating good syntax-highlighting rules for some complex language can still be tricky, you will rarely, if ever, have to do it yourself. The Code Editor is supplied with more than 30 ready-to-use syntax schemes for the most commonly used modern programming languages. In a rare case you need to implement a parser for some custom language, there is a big chance that one of these parsers can be a good starting point. Syntax schemes are stored on disk as .xml files and the built-in visual design-time parser editor is provided to simplify the process of their creation. In case you need most of these languages in your application, you can consider using one of the *SchemeParser* descendants which contains appropriate language schemes in the form of an assembly resource.

Creating a new generic parser is described in the Advanced section of this use guide.

Now that we understand the purpose of all the components in the package, let us go over the major features of the editor and learn how to use them.

Code Completion

Although the main purpose of an editor is to be a convenient tool for the user to enter the text, quite often a guidance from the editor can significantly improve the effectiveness of the work process. When editing a text which has some structure (i.e. computer program in some language), there are often well-defined sets of input possibilities in certain contexts. For example, for many programming languages, the sequence "someobject." should be followed by one of the existing field names. To assist the user in such situations, the text editor can activate popup list containing all the methods that can be accessed from the current scope.



If there is a partial word immediately to the left of the current cursor position, the first entry that starts with that word is highlighted. The user can then continue typing up until the method which he meant is selected or just use up and down arrow keys to navigate the list, and then insert the complete method

name by pressing the *Enter* key. Such a popup also automatically appears whenever the user types a period (‘.’) that follows some identifier and waits for some short period of time. There also are some other code completion methods which will be discussed in detail in the following subsections.

Automatic Code Completion Invocation

The task of code completion is to have the list of available choices to appear automatically as user types, for example after he types “someobject.” They expect the list of class members for that object to appear, and after they type “somemethod(“ they expect the tooltip showing the list of parameters for that function to appear. It can be customized to show those popups only if the user stops input for some short period of time after typing the activating symbol (“.” Or “(“).

The automatic code completion is implemented by Roslyn C# and Visual Basic parsers, TypeScript/JavaScript parsers, as well as by Advanced C#, J#, Visual Basic, VBScript, JavaScript and C parsers.

Automatic code completion is attempted after typing a period (‘.’) following a member (member access expression), typing an open brace (‘{’) following a member (invocation expression or object creation expression), typing a period (‘.’) inside *using* or *imports* section, typing *less sign* (‘<’) inside xml comments, etc. This feature is implemented as close as possible to the Visual Studio .NET editor, so it works in an intuitively understandable way. On top of that Roslyn-based parsers are configured to invoke code completion when the user starts typing identifiers.

When these parsers are used, you still can control some aspects of code completion, for example delay before code completion window appears, using the *NeedCodeCompletion* event, which will be discussed later. Moreover you can register your own types and objects, namespaces and assemblies for code completion using the *CompletionRepository* property of *SyntaxParser*.

To make types from most commonly used assemblies such as *System*, *System.Drawing*, and *System.Windows.Forms* to be available for code completion, you can call the following method

```
csParser1.Repository.RegisterDefaultAssemblies();
```

If you need to provide code completion for assemblies declared in other assemblies, you need to register these assemblies this way:

```
csParser1.Repository.RegisterAssembly ("System.Xml");
```

You may need to register types for code completion that are not declared in the assembly, but present in the form of source code somewhere else.

For Roslyn-based parsers you can rely on underlying solution/project/document object model:

```
csParser1.Repository.RegisterCodeFiles(new string[] { "MyFile.cs" });
```

For one of advanced parsers you need first to create *SyntaxParser*, load this file into the *SyntaxParser.Strings* object, and then add parsed *SyntaxTree* to the code completion repository. The following code demonstrates how it can be accomplished:

```
ISyntaxParser parser = new Alternet.Syntax.Parsers.Advanced.CsParser();  
parser.Strings = new TextStrings();  
parser.Strings.LoadFile("MyFile.cs");
```

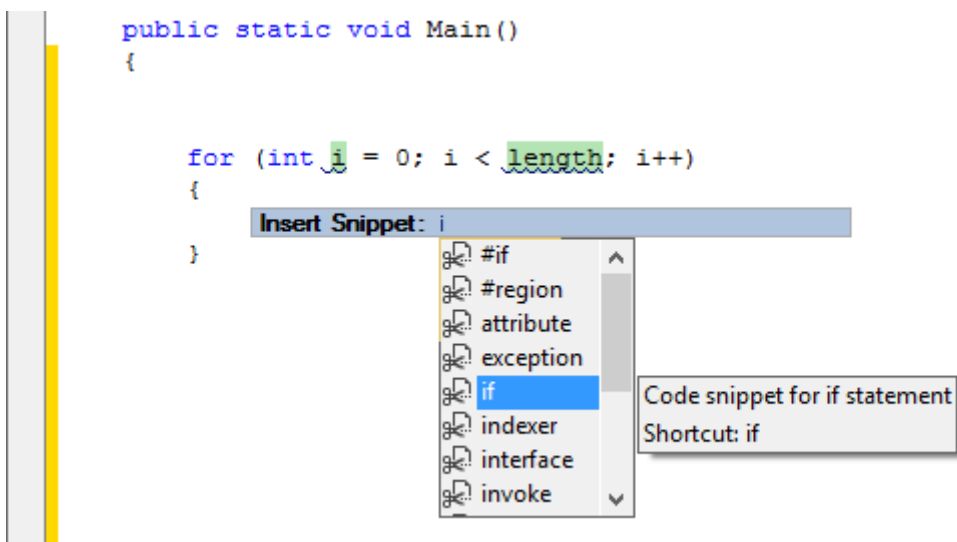
```
parser.ReparseText();  
csParser1.CompletionRepository.RegisterSyntaxTree(parser.SyntaxTree);
```

Code snippets

The code snippets are the next code completion provider, allowing to insert frequently used fragments of code. Code snippets can be inserted into the editor by pressing Tab key after snippet shortcut or by executing code snippet popup window with Ctrl + K + X key sequence, or activated programmatically, by calling the CodeSnippets method of the *SyntaxEdit*.

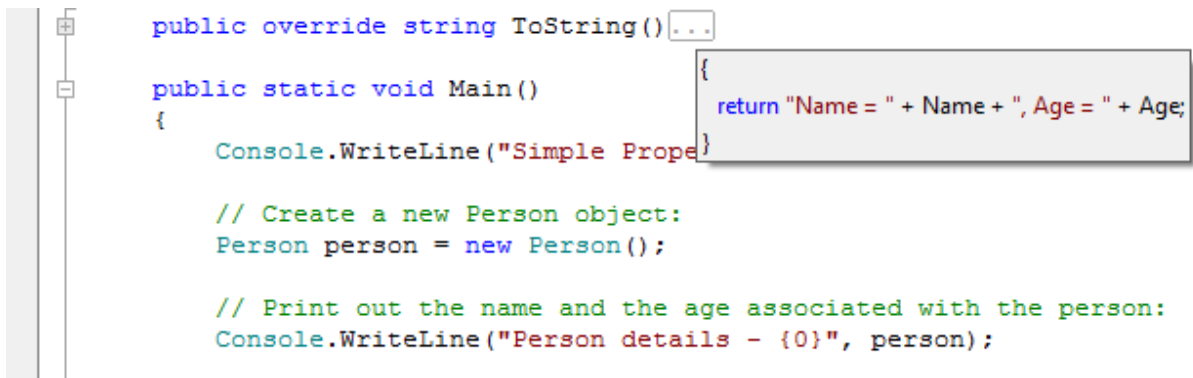
The purpose of the code snippets is to permit the user to quickly enter one of the predefined fragments of text. If the code snippet has fields declared, the editor allows modifying their values causing updating field values inside the whole snippet.

The following picture illustrates the usage of the code snippets.



Code Outlining

The *SyntaxEdit* control supports *outlining*, which is a text navigation feature that can make navigation of large structured texts more comfortable and effective. The essence of *outlining* lies in defining sections of the text as structural units and visually replacing them by some shorter representation, i.e. by ellipsis ("..."). During the text navigation the user can dynamically switch between the collapsed and complete representation of any particular section. Sections can be nested.



The section can be expanded by clicking on the "+" button, by double-clicking the proxy text, or by pressing the `Ctrl+M Ctrl+M` key sequence (in the default key mapping). The section can be collapsed by clicking on the "-" button, or by pressing the `Ctrl+M Ctrl+M` key sequence. All the sections can be globally collapsed or expanded using the `Ctrl+M Ctrl+L` key sequence.

Outlining is the property of the *SyntaxEdit* control itself, not of the *TextSource*, thus it is possible to have two views of the same text one with outlining and another without, or even to have completely different structural parts defined.

All the aspects of the *outlining* are controlled via the *Outlining* property of the *SyntaxEdit*. The *outlining* can be enabled or disabled using the *Outlining.AllowOutlining* property either in design time or at runtime. The look of the outline is controlled by the *Outlining.OutlineColor* and *Outlining.OutlineOptions* properties.

There are two approaches to defining outline sections.

Direct Definition of Outline Sections

Outline sections can be explicitly defined by calling the appropriate methods of the *Outlining* property, i.e.:

```
syntaxEdit1.Outlining.Outline(new Point(0, 0),  
    new Point(int.MaxValue, 0), 0, "...").Visible = false;
```

This code snippet defines the section of the first level consisting of the entire first line of the text, using ellipsis ("...") as the proxy text and being in a collapsed state.

While this approach is the simple one, it has one significant drawback: if sections represent structural units defined by the text itself, and the text can be edited by the user, sections have to be somehow constantly kept in sync with the text, which can be a non-trivial undertaking.

Indirect Definition of Outline Sections Using the Syntax Parser

To provide automatic code outlining, the syntax parsing framework has to be employed. This approach may seem to be more complex at the first look, however it provides consistent results. To implement this approach, a class descending from the *Alernet.Syntax.SyntaxParser* class needs to be defined, and the

Outline method needs to be implemented. This method will be frequently called by the *SyntaxEdit* whenever the text changes, so, to provide the user with a smooth editing experience, the implementation should be relatively fast.

Editor.NET comes with four parsers already supporting automatic outlining for *C#*, *Visual Basic*, *J#*, *JavaScript*, *VBScript*, *Ansi-C*, *SQL*, *HTML* and *XML* languages.

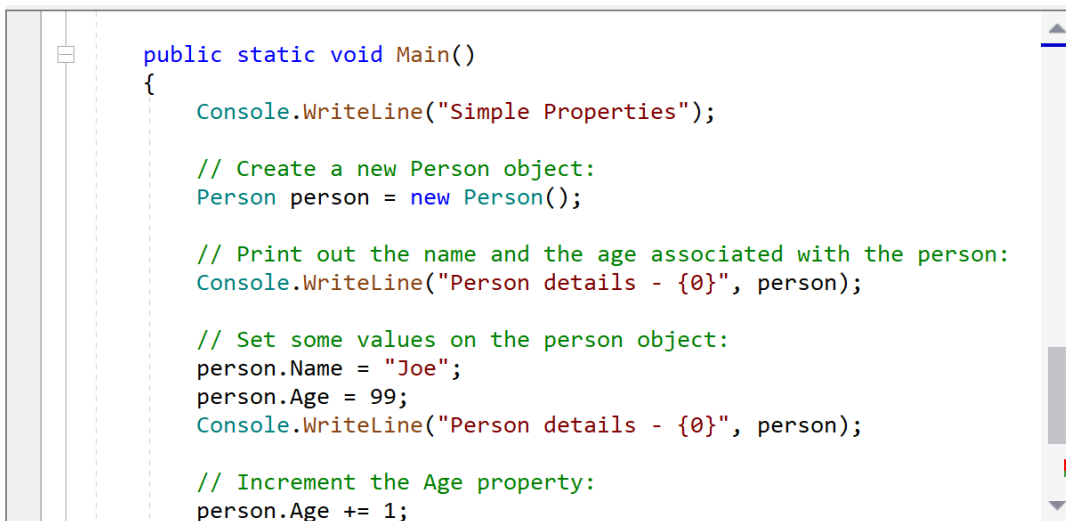
The following example defines a parser class that marks every line starting from the sharp (“#”) sign as a separate outline section.

```
private void InitializeComponent()
{
    ...
    this.parser1 = new XParser();
    ...
}

public class XParser: SyntaxParser
{
    public XParser()
    {
        Options = SyntaxOptions.Outline;
    }
    public override int Outline(IList<IRange> Ranges)
    {
        Ranges.Clear();
        for(int i = 0; i < Strings.Count; i++)
        {
            if(Strings[i].ToString().StartsWith("#"))
            {
                Ranges.Add(new OutlineRange(
                    new Point(0, i),
                    new Point(int.MaxValue, i),
                    0, "...", false));
            }
        }
        return Ranges.Count;
    }
}
```

Structure GuideLines

SyntaxEdit control can display dashed lines between syntax blocks for some parsers (Roslyn-based, TypeScript and some advanced parsers), helping the user to better understand the structure of the document being edited. This behavior is controlled by a Parser and can be switched off by the *StructureGuideLines* parser option.



```

public static void Main()
{
    Console.WriteLine("Simple Properties");

    // Create a new Person object:
    Person person = new Person();

    // Print out the name and the age associated with the person:
    Console.WriteLine("Person details - {0}", person);

    // Set some values on the person object:
    person.Name = "Joe";
    person.Age = 99;
    Console.WriteLine("Person details - {0}", person);

    // Increment the Age property:
    person.Age += 1;
}

```

Selection

Just like almost any text editor, the *SyntaxEdit* supports a concept of text selection and a wide range of operations on it. All the selection related aspects are controlled via the *Selection* property. Selections can be of two types: traditional stream-type selection, and block-type selection. The latter can be created by navigating the text with navigation keys holding *Shift* and *Alt* keys held together.

Selection.BackgroundColor and *Selection.ForegroundColor* define the background and the foreground colors used to mark the currently selected text. *Selection.InActiveBackColor* and *Selection.InActiveForeColor* are used when the editor is out of focus.

The *Selection.Options* controls different aspects of behavior of selections.

- *DisableSelection* completely disables selection support in the editor.
- *DisableDragging* disables drag-n-drop operations on selection.
- *SelectBeyondEol* allows selection in the virtual space (if the *NavigateOptions.BeyondEol* is enabled)
- *UseColors* instructs the editor to use the same foreground colors for selected text, as the ones used for unselected text (i.e. any syntax highlighting will be visible). Note for this to be useful, the section background color must be in contrast with all possible foreground colors.
- *HideSelection* causes the selection to become invisible when the editor loses focus.
- *SelectLineOnDbClick* allows the user to select the entire line by double-clicking on it.
- *DeselectOnCopy* causes selection to be removed after the user performs *copy selection to clipboard* operation.
- *PersistentBlocks* causes selection to be retained after the user has finished making it and has started other navigation.
- *OverwriteBlocks* causes the new input to overwrite the currently selected text.
- *SmartFormat* allows formatting blocks when pasting according to the rules defined by the syntax parser.
- *WordSelect* causes whole words to be selected rather than individual characters when using mouse selection.
- *DrawBorder* causes Edit control to draw border around selection
- *SelectLineOnTripleClick* allows to select whole line rather than single word by triple clicking the mouse
- *DeselectOnDbClick* causes selection to be cleared by dblclick.
- *ConvertToSpacesOnPaste* specifies that selection should convert all tabs to spaces in the text being pasted when *Lines.UseSpaces* is on.
- *RtfClipboard* causes selection to copy its content in text and rtf formats.
- *ClearOnDrag* causes selected text to be cleared after dragging from external source
- *CopyLineWhenEmpty* allows to copy whole line when selection is empty

- *DisableCodeSnippetOnTab* – disables code snippets insertion when pressing Tab key.
- *SelectWordOnCtrlClick* causes word under cursor to be selected when user holds Ctrl key
- *ExtendedBlockMode* causes text being typed to be inserted into the all selected lines within the rectangular block.

It is possible to programmatically select text by setting *Selection.SelectionStart* and *Selection.SelectionEnd* properties, or with the help of *SetSelection* method.

The selected text can be retrieved or set via the *SelectedText* property.

Various operations can be programmatically performed on the current selection. Some of them are:

- *IsEmpty* checks whether there is any text selected.
- *SetSelection* selects the specified rectangular area.
- *SelectAll* selects the whole document.
- *Copy/Cut/Paste* performs standard operations like copying text to the clipboard, cutting text to the clipboard and pasting text from the clipboard.
- *IsPosInSelection* checks if the specified position lies within selection.
- *Clear* clears selection (this does not affect the text itself).
- *Move* moves or copies the currently selected text to a new location.
- *SmartFormat* formats the selected text according to the rules defined by the syntax parser.
- *LowerCase/UpperCase/Capitalize* change the case of the currently selected text.
- *Indent/UnIndent* change the indent of the currently selected text.

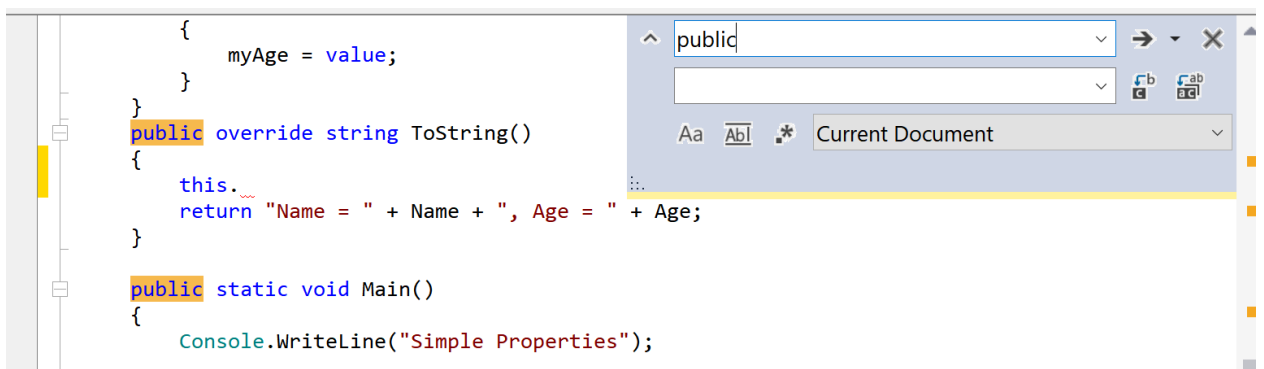
In fact, if some action can be performed by the user, it can also be performed programmatically. Just to give a better idea:

```
if(!SyntaxEdit1.Selection.IsEmpty)
    SyntaxEdit1.Selection.SelectedText =
        "(" + SyntaxEdit1.Selection.SelectedText + ")";
```

This code encloses the currently selected text in brackets.

Searching and Replacing

Among the operations that can be performed upon the text, there are operations of searching and replacing text strings. Unlike the standard multi-line text editor, which does not implement such a functionality, the *SyntaxEdit* control comes with the built-in support for them. It's ready to use out-of-the-box: when the user presses *Ctrl+F* key combination, the search dialog box appears:



The text-replace dialog can be activated by pressing *Ctrl+H*. Besides using the UI to control the process, all the operations can be executed programmatically by calling the corresponding methods of the *SyntaxEdit*.

For example, to find some string, you could use the following code:

```
syntaxEdit1.Find("some string");
```

Or, with regular expressions:

```
syntaxEdit1.Find(" ", SearchOptions.RegularExpressions, new  
System.Text.RegularExpressions.Regex("a.?z"));
```

To activate the Search Dialog:

```
syntaxEdit1.DisplaySearchDialog();
```

Moreover, the Search and Replace dialog box functionality is not hardwired: you can replace the dialog box by your own, by implementing the *ISearchDialog* interface, and assigning it to the editor by setting its *SearchDialog* property. The built-in dialog can serve as a good example and a starting point.

If you need to perform the Search and Replace operation without any user interaction, you can use the *ReplaceAll* method.

I.e.:

```
syntaxEdit1.ReplaceAll("bad", "good", SearchOptions.WholeWordsOnly |  
SearchOptions.EntireScope, out count);
```

After this, every occurrence of “bad” word in the entire text will be replaced by the “good”.

Note, that this would move the cursor position to the place where the last replacement has been made, so if you need it to be truly unnoticeable for the user, you need to enclose this call in the code which saves and restores the current cursor position:

```
System.Drawing.Point pos;  
pos = syntaxEdit1.Position;  
syntaxEdit1.ReplaceAll("bad", "good", SearchOptions.WholeWordsOnly |  
SearchOptions.EntireScope, out count);  
syntaxEdit1.Position = pos;
```

Search/Replace function can work across multiple documents. In order to allow search to find text in multiple editors, you will need to set *SearchManager.Shared* to true and provide list of editors to perform search in its *InitSearch* event handlers and return/navigate to the appropriate editor in *GetSearch* event handler:

```
SearchManager.SharedSearch.Shared = true;  
  
SearchManager.SharedSearch.InitSearch +=  
    new InitSearchEvent(DoInitSearch);  
SearchManager.SharedSearch.GetSearch +=  
    new GetSearchEvent(DoGetSearch);  
  
private void DoInitSearch(object sender, InitSearchEventArgs e)  
{  
    e.Search = GetActiveSyntaxEdit() as ISearch;  
    foreach (var edit in editors.Values)  
    {
```



```

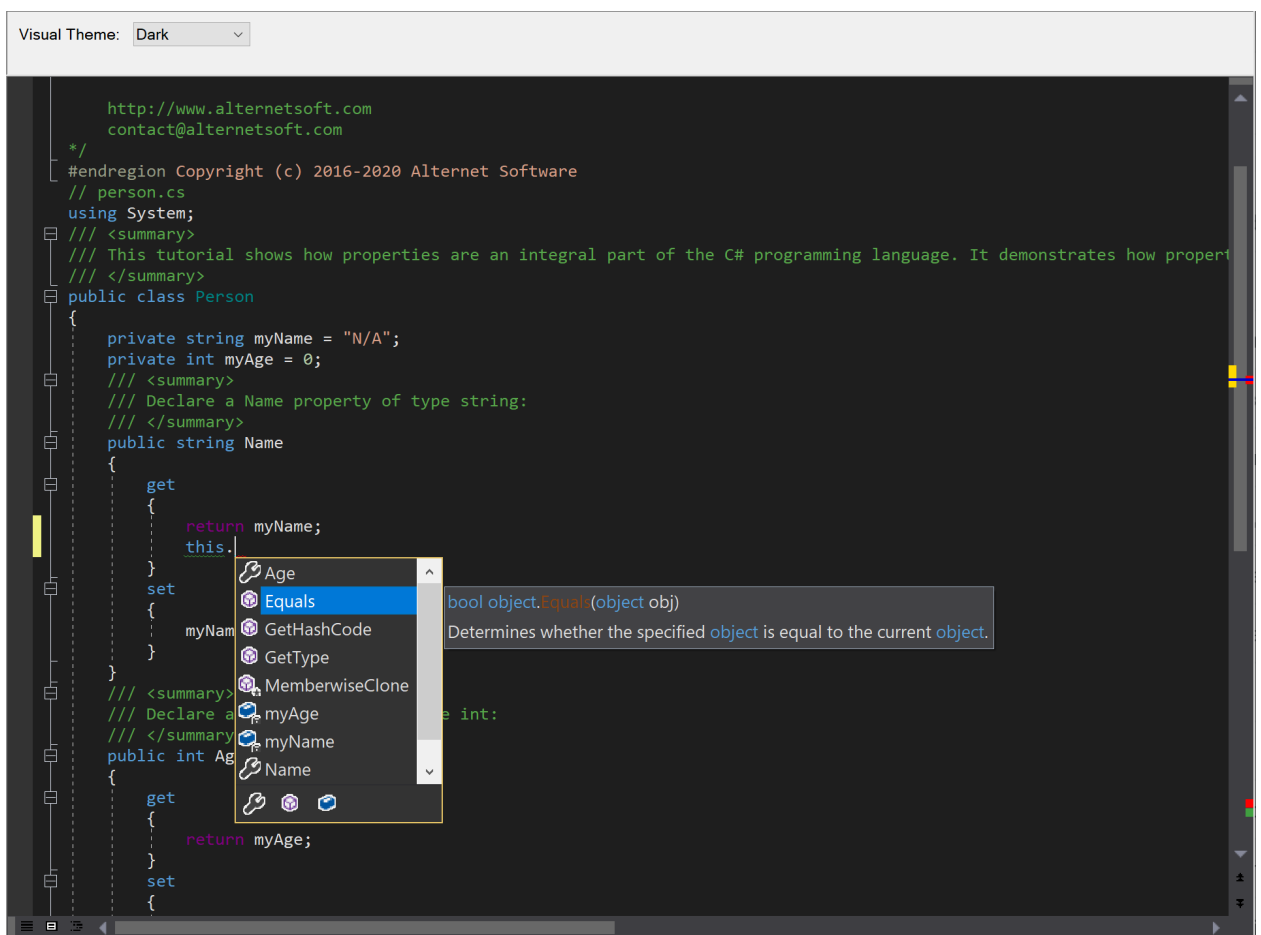
        edit.SearchGlobal = true;
        e.SearchList.Add(edit.Source.FileName);
    }
}

private void DoGetSearch(object sender, GetSearchEventArgs e)
{
    foreach (var edit in editors.Values)
    {
        if (edit.Source.FileName == e.FileName)
        {
            e.Search = edit as ISearch;
            break;
        }
    }
}
}

```

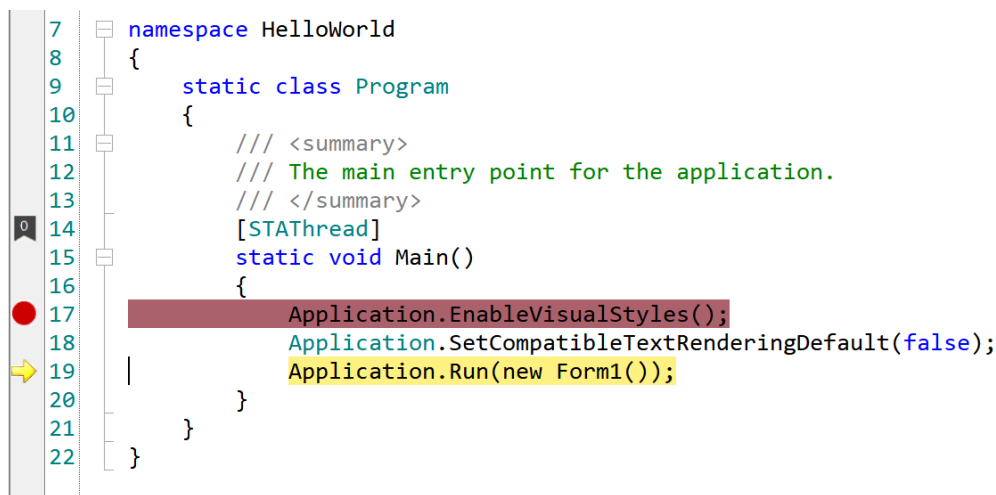
Visual Themes

Visual themes allow to change the appearance of all graphical elements in the editor by setting *VisualThemeType* or *VisualTheme* type properties. Light and Dark visual themes are included, and custom appearance can be configured via custom visual theme.



Gutter

The gutter is the area to the left of the text, the purpose of which is to display miscellaneous indicators for the corresponding lines of text. Among these indicators are bookmark indicators, line wrapping indicators, line styles icons, line numbers, outlining buttons and line modification markers.



All the images displayed in the gutter are contained in the gutters image list. The following code gives an example of how to add a custom icon to this list from another image list (for example, the one dropped on the form during design-time):

```
syntaxEdit1.Gutter.Images.Images.Add(imageList1.Images[0]);
```

The mechanism of the line styles icons allows you to define how certain lines of text will be displayed. The most common use for this is the indication of breakpoint lines and of the current execution point.

For example, the following code defines the style to be used for breakpoints.

```
style_id = syntaxEdit1.LineStyles.AddLineStyle("breakpoint",  
Color.White, Color.Red, Color.Gray, 11, LineStyleOptions.BeyondEol);
```

(Note, in the current version, image # 11 corresponds to the built-in breakpoint indicator image, and #12 corresponds to the current execution point image.

Later on, some line of the text can be assigned the style:

```
syntaxEdit1.Source.LineStyles.SetLineStyle(line_no, style_id);
```

(Note, that here and in the other places of this document line numbers start at 0.)

Note: at any given time, every line can have at most one style. If you need to remove line style for some particular line, call:

```
syntaxEdit1.Source.LineStyles.RemoveLineStyle(line_no);
```

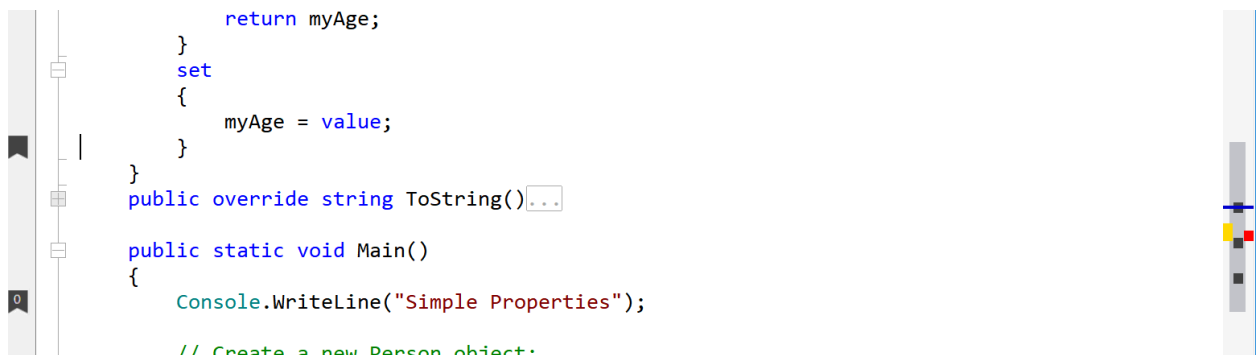
For *SyntaxEdit* control appearance of the gutter is controlled by the following properties: *Gutter.Width*, *Gutter.BrushColor*, *Gutter.PenColor* and *Gutter.Visible*. *Width* property specifies width of the gutter area, *BrushColor* specifies background color of the gutter area, *PenColor* specifies color of the gutter line, and *Visible* indicates whether or not to draw gutter. Note that gutter can adjust its width if line numbers or outlining is on and painted on the gutter. *SyntaxEdit* allows drawing line numbers to visually indicate position of the visible lines inside the document. To enable line numbers you need to set *Gutter.Options.PaintLineNumbers* to true. Turning *Gutter.Options.PaintLinesOnGutter* option on enables drawing line numbers on gutter area, turning it off causes line numbers to be painted immediately after gutter area. Appearance of line numbers are controlled by the *Gutter's* properties: *LineNumbersStart*, *LineNumbersForeColor*, *LineNumbersBackColor*, *LineNumbersAlignment*, *LineNumbersLeftIndent* and *LineNumbersRightIndent*, which are intuitively understandable.

Like Microsoft Visual Studio editor, *SyntaxEdit* provides the ability to visually track modified lines. To enable this feature you need to turn *Gutter.Options.PaintLineModifiers* on. When *LineModifiers* are on they indicate lines that were changed since last saving. New changes are marked with Yellow color; changes that were done before last saving are marked with Lime color. Colors can be customized using *LineModifierChangedColor* and *LineModifierSavedColor* properties.

Reaction to mouse clicks and double-clicks on the gutter area can be implemented by assigning handlers to the *GutterClick* and *GutterDbClick* events.

Bookmarks

Just as with often used reference books, the process of navigating the text can be made more comfortable and efficient with the usage of bookmarks. Two kinds of bookmarks are supported by the *SyntaxEdit*: plain and numbered. The former can be toggled for the current line using the *Ctrl+K Ctrl+K* key combination sequence, and can be navigated in cyclical manner using the *Ctrl+K Ctrl+N* (next bookmark) or *Ctrl+K Ctrl+P* (previous bookmark). The numbered bookmarks have a different flavor: there can be up to ten bookmarks, each having a number associated with it.



Toggling the numbered bookmark is performed using the *Ctrl+K Ctrl+#*, and navigation to the specific bookmark is performed by pressing the *Ctrl+#* key combination (where # is any of the digits from 0 to 9). There can be only one plain bookmark in any line. Numbered bookmarks do not have such a limitation, however, only the indicator for the first bookmark in the line will be displayed in the gutter area, if *Gutter.Options.PaintBookMarks* is set to true.

Like most other things in the editor, bookmarks can be manipulated programmatically. Note that the list of bookmarks belongs to the text source, so multiple views of the same source share the same set of bookmarks.

The following code snippet sets the plain bookmark at the current position:

```
System.Drawing.Point pos = syntaxEdit1.Position;
syntaxEdit1.Source.BookMarks.SetBookmark(pos, int.MaxValue);
```

To set the numbered bookmark, replace *int.MaxValue* by the bookmark number (0..9).

To clear all the bookmarks set in the text source, call the *ClearAllBookMarks* method:

```
syntaxEdit1.Source.BookMarks.ClearAllBookMarks();
```

Navigating to the location defined by a particular bookmark can be performed as follows:

```
syntaxEdit1.Source.BookMarks.GotoBookMark(index);
```

Code Editor supports named bookmarks with description and hyperlink. The user may see a description in a tooltip window when moving the cursor over the bookmark, and load the browser with specified url when clicking on the bookmark. Such bookmarks can be set using the following code:

```
syntaxEdit1.Source.BookMarks.SetBookmark(syntaxEdit1.Position, 0,  
"Bookmark1", "This is Named Bookmark", "www.alernetsoft.net");
```

If you need to have custom images, you can change the bookmark indicator images by assigning custom image list:

```
syntaxEdit1.Gutter.BookMarkImageIndex =  
syntaxEdit1.Gutter.Images.Images.Count;  
syntaxEdit1.Gutter.Images.Images.Add(imageList1.Images[0]);
```

(This code uses the first image from the *imageList1*, which you could, for example, create by just dropping a new Image List from the toolbox on the form. For more examples on working with the gutter, refer to the corresponding section of this manual.)

You can configure bookmarks navigation to work across multiple documents. These documents should be added to the *BookMarkManager* class, and every document should have the *FileName* property assigned.

```
BookMarkManager.Register(syntaxEdit1.Source);  
BookMarkManager.SharedBookMarks.Activate += new  
EventHandler<ActivateEventArgs>(DoActivate);  
  
private void DoActivate(object sender, ActivateEventArgs e)  
{  
    foreach (var edit in editors.Values)  
    {  
        if (edit.Source.FileName == e.FileName)  
        {  
            ActivateEditorTab(editor);  
            break;  
        }  
    }  
}
```

In this mode all bookmarks will be stored in a global list inside *BookmarkManager* instead of every individual *SyntaxEdit* control allowing global navigation through them.

Keyboard Mapping

While the *SyntaxEdit* closely mimics the key-mapping common to most of Microsoft's products, it is completely customizable: you can add or change behavior of certain keys or even define an entirely different key-mapping.

To assign an action to some key combination, use the following code:

```
private void syntaxEdit1_Action()  
{  
    ...  
}  
...  
syntaxEdit1.KeyList.Add(Keys.W | Keys.Control | Keys.Alt, new  
KeyEvent(syntaxEdit1_Action));
```

This would make the *Ctrl+Alt+W* key combination execute the *syntaxEdit1_Action* method.

Or, to pass some object to the key handler:

```
private void syntaxEdit1_Action(object o)  
{  
    ...  
}  
...  
syntaxEdit1.KeyList.Add(Keys.W | Keys.Control | Keys.Alt, new  
KeyEventEx(syntaxEdit1_Action), some_object);
```

To remove some key handler, regardless of whether you have added it yourself, or it is the default one, call:

```
syntaxEdit1.KeyList.Remove(Keys.A | Keys.Control);
```

The code described before is used to manage the key handling in the default state. In fact, the key handling is slightly more complex than that: the *SyntaxEdit*'s key handling mechanism can be in different states, other than the default one. Every state has its own key mapping table. Key mapping for bookmark operations can serve as a good example: after the user presses the *Ctrl+K* key combination, combinations *Ctrl+K*, *Ctrl+N*, *Ctrl+P*, *Ctrl+L* (the list is incomplete) obtain the new meaning. If a key combination is pressed for which there is no assignment in some non-default state, then the state is changed to default, and the combination is evaluated in the new context. *SyntaxEdit* defines four different non-default states, but you can implement your own:

```
syntaxEdit1.KeyList.Add(Keys.W | Keys.Control, null, 0, 5);  
syntaxEdit1.KeyList.Add(Keys.Tab, new KeyEvent(syntaxEdit1_Action), 5,  
5);
```

This code creates a state that is activated by pressing the *Ctrl+W* key combination, and in which the *Tab* key causes the *syntaxEdit1_Action* to be executed. The state is changed back to default when the user presses some key other than the *Tab*.

Up until now we have only examined the cases where you add some new functionality, or suppress some existing one. There also might be a case, when you want to use an entirely different key mapping, for example, to simulate some other environment your users are familiar with. To accomplish this, it is necessary to completely clear the current key mapping, and then to assign every function performed by the editor to some key. Note, that this really means every function: even such trivial things as cursor navigation and insertion of a new line are performed according to the key mapping.

For example, the following code assigns the editor's key-mapping to a single action defined: "Select All", which is assigned to the *Ctrl+X* key combination.

```
syntaxEdit1.KeyList.Clear();  
syntaxEdit1.KeyList.Add(Keys.X | Keys.Control,  
((EventHandlers)syntaxEdit1.KeyList.Handlers).SelectAllEvent);
```

Spellchecker Interface

The *SyntaxEdit* supports the spell-as-you-type spell checker integration. To enable spelling for the editor, set its *Spelling.CheckSpelling* property to true and assign the *WordSpell* event handler.

The following artificial example considers any word longer than 3 characters to be correct:

```
private void syntaxEdit1_WordSpell(object sender,
Alternet.Editor.TextSource.WordSpellEventArgs e)
{
    e.Correct = e.Text.Length > 3;
}
...
this.syntaxEdit1.WordSpell += new
Alternet.Editor.TextSource.WordSpellEvent(this.syntaxEdit1_WordSpell);
```

Incorrect words are displayed with the wiggly underline (the default color is red, but it can be changed using the *Spelling.SpellColor* property). In real-life scenarios you would need to use some third-party software/dictionary to really check the text. Another alternative would be using some word-list file, many of them, including Public Domain or free ones, can be found on the Internet. Take a look at Miscellaneous quick start project, which has one of these dictionaries.

Another useful feature supported by *SyntaxEdit* is AutoCorrect, allowing you to auto correct words when typing. To enable this feature you need to set property *AutoCorrection* to true and handle the *AutoCorrect* event to provide replacements for words that were typed incorrectly.

URL handling

The *SyntaxEdit* can be set up to handle pieces of text that look like some kind of an URL by setting the *HyperText.HighlightUrls* property to *true*. The handling consists of highlighting those pieces of text, and of processing clicks on them. By default, clicking the URL causes the operating system default action to be performed (i.e. launching a browser or an email client), however, you can override this behavior by assigning the *JumpToUrl* event handler.

```
private void syntaxEdit1_JumpToUrl(object sender,
Alternet.Editor.UrlJumpEventArgs e)
{
    if(is_our_url(e.Text))
    {
        process_url(e.Text);
        e.Handled = true;
    }
}
```

Printing and Exporting

SyntaxEdit includes support for printing, print previewing, and exporting to RTF and HTML.

Exporting can be performed as simple as this:

```
syntaxEdit.SaveFile(FileName, new RtfExport());
```

Printing tasks are performed and configured via the *Printing* property of the *SyntaxEdit*.

For example, to show the print preview dialog, call:

```
syntaxEdit1.Printing.ExecutePrintPreviewDialog();
```

SyntaxEdit control supports adding user-defined information while printing.

To add some text to the footer:

```
syntaxEdit1.Printing.Footer.CenterText = "draft";
```

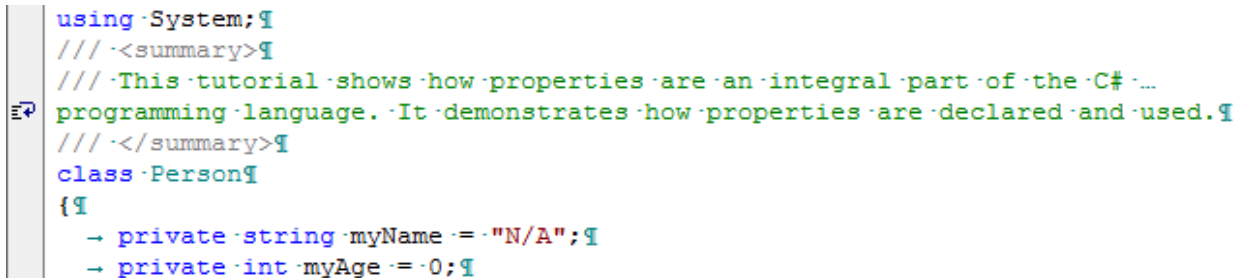
Text in headers and footers can contain substitution tags. The standard ones are: \[page], \[pages], \[date], \[time] and \[username]. It is possible to add custom tags by assigning a handler for the *DrawHeader* event, for example:

```
private void syntaxEdit1_DrawHeader(object sender, Alternet.Editor.  
DrawHeaderEventArgs e)  
{  
    if(e.Tag == "\\[tag]")  
    {  
        e.Text = "tag replacement text";  
        e.Handled = true;  
    }  
}
```


Miscellaneous Options

White-space Display

It is sometimes desirable for the user to see the codes which influence the layout of the text and are normally invisible themselves. These codes are space, tab, end-of-line, and the end-of-file (not really a code), and are often collectively referred to as the white-space. The *SyntaxEdit* has the option to display them, and to control their appearance.



```
using System;
/// <summary>
/// This tutorial shows how properties are an integral part of the C# ...
programming language. It demonstrates how properties are declared and used.
/// </summary>
class Person
{
    private string myName = "N/A";
    private int myAge = 0;
```

The display of the white-space is enabled using the *WhiteSpace.Visible* property. The color used to display white-space codes is determined by the *WhiteSpace.SymbolColor* property, and the characters used to display those codes are determined by *EofSymbol*, *EolSymbol*, *SpaceSymbol*, and *TabSymbol* properties.

Line Separator

It is possible to have lines of the editor to be separated by thin horizontal lines, and to have the current line highlighted. This behavior is controlled by the *LineSeparator* property.

The following options are available:

- *HighlightCurrentLine* specifies that the current line in the editor will be highlighted using the *HighlightColor* for background.
- *HideHighlighting* specifies that the highlighting of the current line should be hidden when the editor loses focus.
- *SeparateLines* specifies that a thin horizontal line of *LineColor* should be drawn between each line of text.
- *SeparateWrapLines* specifies that each visual line of text produced as a result of word-wrap should be separated in the same manner as separate lines (works only if the *SeparateLines* option is also specified).
- *SeparateContent* specifies that line separator will be drawn between sections of the code (for example between methods), if *SyntaxEdit* control is associated with *SyntaxParser* supporting this feature.

Scroll Bars and Split View

The appearance and behavior of scrollbars is controlled by the *Scrolling* property.

The *Scrolling.ScrollBars* property determines which scrollbars and under what conditions appear on the *SyntaxEdit*. It can take one of the following values:

- *None* – neither horizontal, nor vertical scrollbar ever appear
- *Horizontal* – horizontal scrollbar appears if necessary, vertical one never appears
- *Vertical* – vertical scrollbar appears if necessary, horizontal one never appears
- *Both* – both horizontal and vertical scrollbars appear if necessary
- *ForcedHorizontal* – horizontal scrollbar is always visible, vertical one never appears
- *ForcedVertical* – vertical scrollbar is always visible, horizontal one never appears
- *ForcedBoth* – both horizontal and vertical scrollbars are always visible

Behavior of the scrollbars is controlled by *ScrollingOptions*.

You can also use *ScrollingOptions* to allow *SyntaxEdit* to split its content. Note that *SyntaxEdit*'s *Dock* must be set to *DockStyle.Fill*, otherwise this feature will not work. Splitters are displayed in the left-bottom corner for vertical splitting and in the right-top corner for horizontal splitting.

- *SmoothScroll* – if set, the display is updated as the user drags the scrollbar, otherwise the display is updated only when the user releases the scrollbar thumb. Disabling this option may improve performance on slow machines.
- *ShowScrollHint* – if set, a hint window, showing the new number of the topmost string, is displayed whenever the user drags the scrollbar.
- *UseScrollDelta* – if set, editor window content is scrolled by several characters when caret becomes invisible rather than one character
- *SystemScrollbars* – if set, system scroll bars are displayed, otherwise custom scrollbars are used.
- *FlatScrollbars* – if set, scroll bars are displayed in flat style. This option works only if *SystemScrollBars* is on.
- *AllowSplitHorz* – allows displaying horizontal splitting buttons in the scroll area. This option works only if *SystemScrollBars* is off and control has *Dock* property set to *DockStyle.Fill*.
- *AllowSplitVert* – allows displaying a vertical splitting button in the scroll area. This option works only if *SystemScrollBars* is off and control has *Dock* property set to *DockStyle.Fill*.
- *HorzButtons* – allows displaying additional buttons in the horizontal scrolling area. This option works only if *SystemScrollBars* is off.
- *VertButtons* – allows displaying additional buttons in the vertical scrolling area. This option works only if *SystemScrollBars* is off.
- *VerticalScrollBarAnnotations* – allows displaying scroll bar annotations that show special items such as line modifications, syntax errors, search results bookmarks and the caret position, throughout the entire document within the scroll bar. Individual annotation kinds are controlled by *Annotations* property.

Advanced Topics

Automatic Code Completion for arbitrary programming language

If there is no *SyntaxParser* for your language, you can consider implementing automatic code completion using *NeedCodeCompletion* event:

```
private void syntaxEdit1_NeedCodeCompletion(object sender,
Alternet.Syntax.CodeCompletionArgs e)
{
    if ((e.CompletionType == CodeCompletionType.ListMembers) ||
        (e.CompletionType == CodeCompletionType.CompleteWord) ||
        ((e.CompletionType == CodeCompletionType.None) && (e.KeyChar == '.')))
    {
        // Look at Manual Code Completion, list members section
        ...
        e.Interval = (e.CompletionType != CodeCompletionType.None)
            ? 0 : 500;
    }
    if(e.CompletionType == CodeCompletionType.ParameterInfo ||
        e.CompletionType == CodeCompletionType.None &&
        e.KeyChar == '(')
    {
        // Look at Manual Code Completion, parameter info section
        ...

        e.Interval = (e.CompletionType != CodeCompletionType.None)
```

```
? 0 : 500;  
}  
}
```

Depending on the kind of the language you are working with, and whether you are using some complete library to work with that language, or do everything yourself, the actual information on symbols will be retrieved in different ways:

- if you are using some third party library, look for something that resembles the name “Symbolic Information API” or like in the manual for that library;
- if you are developing your own language, or at least your own engine for some existing language, you probably already know what exactly to do to acquire the information necessary for *code completion* to work;
- If you are working with the .NET family of languages, *CLR Reflection API* should probably be of use for this purpose. The sample program supplied with the package provides a good starting point on working with it.

Manual Code Completion

If you use Code Editor with the parser that does not fully support automatic code completion, you can still provide some guidance to the users as he types by implementing some of the code completion logic manually.

Quick Info

To give the understanding of the *code completion* architecture we will start from the simplest one. Please note that the name “*code completion*” can be somewhat misleading as it does not always imply the ability to enter some code from the list. Sometimes it just means providing the user with some information on the current context.

The first thing to do to start implementing code completion is to assign the handler for the editor’s *NeedCodeCompletion* event. This event occurs whenever the code completion is explicitly requested by the user by means of one of the key combinations (the default one for the quick info is the *Ctrl+K Ctrl+I* sequence), and also whenever some key press has caused the text to be input (this is to permit automatic activation after entering period (“.”) or like. Because this event gets called rather often, the handler has to be reasonably fast to avoid slowing the things down.

The handler receives two arguments: first is the *SyntaxEdit* object that sends the event, and the second is the *CodeCompletionArgs* object which contains the information about the request and receives the results provided by the handler.

The *CodeCompletionArgs* contains the following members which are of interest to us:

- [in] *CompletionType* takes one of the following values: *None*, *CompleteWord*, *ListMembers*, *ParameterInfo*, *QuickInfo*, and specifies the type of *code completion* request. *None* means that this event has occurred as the result of text input.
- [in] *KeyChar* contains the character that has been just entered, if the *CompletionType* is *None*.
- [in, out] *StartPosition* and *EndPosition* determine the range of cursor positions in which the *code completion* window will remain visible. Whenever the user navigates out of this range the *code completion window* is closed. Initially the *StartPosition* is set to the current cursor position and the *EndPosition* is set to *(-1, -1)* which means to the end of this line. The handler can modify these values.
- [out] *Handled* specifies whether the event has been handled. If this value is *true* the *code completion* popup does not appear.
- [in, out] *NeedShow* specifies whether the *code completion* popup is to be shown.

- [out] *Interval* specifies the delay, expressed in milliseconds, before showing the *code completion* popup. Zero value means to show immediately. If the popup is delayed it can be cancelled during the delay interval as a result of further user input or navigation.
- [out] *ToolTip* specifies whether *code completion* popup should appear as a tooltip window (*true*) or as a list (*false*). When the popup appears as the tooltip it just provides the user some information, and cannot cause any text to be entered.
- [out] *Provider* is the object supplied by the handler, which is used to hold the data required by the *code completion*.

The members marked as “[in]” contain the information provided by the *SyntaxEdit*, and the “[out]” members can be filled by the handler.

There are different types of *providers* each suitable for different *code completion* types. The one to be used to provide the *Quick Info* is named *QuickInfo*. It is the simplest of *providers*, and it holds just a single string to be displayed in the tooltip window and optionally a range of characters in that string to be shown in bold font.

Now let's try to write some working code.

```
private void syntaxEdit1_NeedCodeCompletion(object sender,
Alternet.Syntax.CodeCompletionArgs e)
{
    if (e.CompletionType == CodeCompletionType.QuickInfo)
    {
        IQuickInfo p = new QuickInfo();

        SyntaxEdit edit = (SyntaxEdit)sender;
        Point Pt = edit.PointToClient(Cursor.Position);
        Pt = edit.ScreenToText(Pt.X, Pt.Y);
        int Left, Right;
        if (!edit.Lines.GetWord(Pt.Y, Pt.X, out Left, out Right))
        {
            p.Text = "No word under cursor";
            e.EndPosition = e.StartPosition;
        }
        else
        {
            string word = edit.Lines[Pt.Y].Substring(Left, Right - Left + 1);
            string message = "The word is: ";

            p.Text = string.Format("{0}<b>{1}</b>", message, word);

            e.StartPosition = new Point(Left, e.StartPosition.Y);
            e.EndPosition = new Point(Right, e.StartPosition.Y);

            e.NeedShow = true;
            e.Provider = p;
            e.ToolTip = true;
        }
    }
}
```

This example provides the *Quick Info* tooltip (activated by the *Ctrl+K Ctrl+I* sequence) displaying the word which is currently under cursor. *Quick Info* can also be activated programmatically, using the *QuickInfo* method of the *SyntaxEdit*.

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

The word is: **Name**

List Members

The *ListMembers* provider is used to respond to *ListMembers* code completion requests. It can be activated by pressing *Control+J* key combination, or programmatically, by calling the *ListMembers* method of the *SyntaxEdit*.

The purpose of the *members* is to permit the user to quickly choose one of the entities of the program (procedure, method, variable, etc.) relevant to the current context. This may mean listing all the available functions in the statement context of a procedural language, all the type compatible functions and variables in expression context. For object-oriented languages in addition to functions and variables there would also be methods and fields of the current class. In a qualifier context (like “someobject.”, or “somestructure.”, or “somepointer->”) the list should consist of methods and fields of the corresponding class/type. Detailed explanation of ways to acquire such an information lies out of scope of this document, however, please take a look at the end of this chapter for some references.

```
public static void Main()
{
    Console.  

    Console.  

    Console.  

}
```

The following code snippet illustrates the usage of the *list members*.

```
private void syntaxEdit1_NeedCodeCompletion(object sender,
Alternet.Syntax.CodeCompletionArgs e)
{
    if (e.CompletionType == CodeCompletionType.ListMembers)
    {
        IListMembers p = new ListMembers();
        p.ShowDescriptions = true;
        p.ShowResults = false;
        p.ShowQualifiers = false;

        IListMember m = p.AddListMember();
        m.Name = "print";
        m.DisplayText = "<b>print</b>";
    }
}
```

```
m.DataType = "void";
m.Qualifier = "public";
m.ImageIndex = 1;
m.Description = "void print(ref string line)";

m = p.AddListMember();
m.Name = "evaluate";
m.DisplayText = "<b>evaluate</b>";
m.DataType = "double";
m.Qualifier = "protected";
m.ImageIndex = 2;
m.Description = "double evaluate(string expression)";

e.NeedShow = true;
e.Provider = p;
e.ToolTip = false;
    }
}
```

Parameter Info

The *ParameterInfo* provider is used to respond to *ParameterMembers* code completion requests. It can be activated by pressing *Ctrl+Shift+Space* key combination, or programmatically, by calling the *ParameterInfo* method of the *SyntaxEdit*.

```
public static void Main()
{
    Console.Beep (
        ▲1 of 2 ▼ public void print (ref string line)
```

The purpose of the *parameter info* is to show a tooltip describing the parameters of the method call under cursor. Note that this only works for languages with traditional syntax for specifying parameters, i.e. method (arg1, arg2...).

```
private void syntaxEdit1_NeedCodeCompletion(object sender,
Alternet.Syntax.CodeCompletionArgs e)
{
    if (e.CompletionType == CodeCompletionType.ParameterInfo)
    {
        IListMembers p = new ListMembers();
        p.ShowDescriptions = true;
        p.ShowResults = true;
        p.ShowQualifiers = true;
        p.ShowParams = true;

        IListMember m = p.AddListMember();
        m.Name = "print";
        m.DataType = "void";
        m.Qualifier = "public";
        m.ImageIndex = 1;
        m.Parameters = new ParameterMembers();
        IParameterMember param = m.Parameters.AddParameterMember();
        param.DataType = "string";
        param.Name = "line";
        param.Qualifier = "ref";
        m.ParamText = "(ref string line)";

        m = p.AddListMember();
        m.Name = "print";
        m.DataType = "void";
        m.Qualifier = "protected";
        m.ImageIndex = 2;
        m.Parameters = new ParameterMembers();
        param = m.Parameters.AddParameterMember();
        param.DataType = "string";
        param.Name = "expression";
        m.ParamText = "(string expression)";
        e.NeedShow = true;
        e.Provider = p;
        e.ToolTip = true;
    }
}
```

Localization of dialogs

All string constants used in dialogs are localized to a few foreign languages. So far *CodeEditor* supports German, French, Spanish, Russian and Ukrainian languages. The following code demonstrates how to switch to German language:

```
Using Altnet.Common;
...
CultureInfo oldcInfo = Thread.CurrentThread.CurrentUICulture;
Thread.CurrentThread.CurrentUICulture = new CultureInfo("de");
try
{
    StringConsts.Localize();
}
finally
{
    Thread.CurrentThread.CurrentUICulture = oldcInfo;
}
```

Page Layout mode

SyntaxEdit has different ways to get a good view of the editor content. Use normal mode for typing, editing, and formatting text. Working in page layout mode making it easy to see how text will be positioned on the printed page. Page breaks mode is similar to normal mode, but allows to visually separate pages by displaying dotted lines between individual pages. Current mode is controlled by the *Pages.PageType* property. Use the *Pages.DefaultPage* property to change bounds and margins of the default page. In Page Layout mode it may be useful to display horizontal and vertical rulers, which will allow users to visually change margins of the current page or selected range of pages. Rulers can be turned on or off using *Pages.Rulers* property.

Marco Recording and PlayBack

SyntaxEdit has macro recording and playback capabilities. It allows recording sequences of keyboard commands and playing them later. Note that mouse input is not recorded.

This feature enables you to store a set of frequently used editing commands. Set *MacroRecording* property to start/finish macro recording. Use the *PlayBack* method to repeat the stored command sequence.

Global Settings

If the application contains more than one instance of the editor, it is quite often desired to share their UI settings, and to provide the user with a centralized facility to manage them. Code Editor is shipped with *Customize* quick start project that demonstrates how this can be accomplished.

It includes *SyntaxSettings* class which is a holder for the following set of settings:

- The font used to display the text in the editor
- Syntax highlighting styles (i.e. foreground and background colors, font style)
- Whether the following features are enabled or not:
 - Show margin
 - Show gutter
 - URL highlighting
 - Outlining
 - Word wrapping
 - Use of spaces instead of tabs for indents
- The width of the gutter area

- The position of the margin
- Tab-stop positions
- Navigation options
- Selection options
- Outline options
- Scrollbar options
- Color Themes

To use this class, its instance must be created, i.e.:

```
private SyntaxSettings GlobalSettings;
...
GlobalSettings = new SyntaxSettings();
```

The settings can be retrieved from some particular *SyntaxEdit* controls as follows:

```
GlobalSettings.LoadFromEdit(syntaxEdit1);
```

And then assigned to some other editor like this:

```
GlobalSettings.ApplyToEdit(syntaxEdit2);
```

Settings can be easily stored to some file:

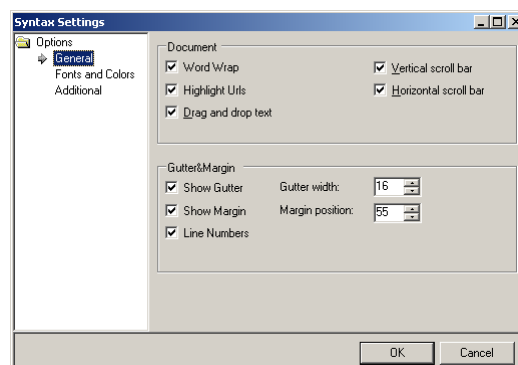
```
GlobalSettings.SaveFile("GlobalSettings.xml");
```

And later on, loaded from that file:

```
GlobalSettings.LoadFile("GlobalSettings.xml");
```

As the name of the file hints, settings are stored in the XML format. Note, that in the real application you would check for the existence of that file, and also, this file should probably be located somewhere down the user's *Application Data* folder.

To make the handling of the global settings even easier, the *Customize* demo project includes an example settings dialog.



All you need to do to use it, is to declare and construct its instance:

```
using Alternet.Editor.Dialogs    ;
...
private DlgSyntaxSettings Options;
...
Options = new DlgSyntaxSettings();
```

And later on, when the user requests the editor settings dialog perform something similar to the following:

```
Options.SyntaxSettings.Assign(GlobalSettings);
if(Options.ShowDialog() == DialogResult.OK)
{
    GlobalSettings.Assign(Options.SyntaxSettings);
    // for each syntaxEdit used in the application do
    GlobalSettings.ApplyToEdit(syntaxEdit);
}
```

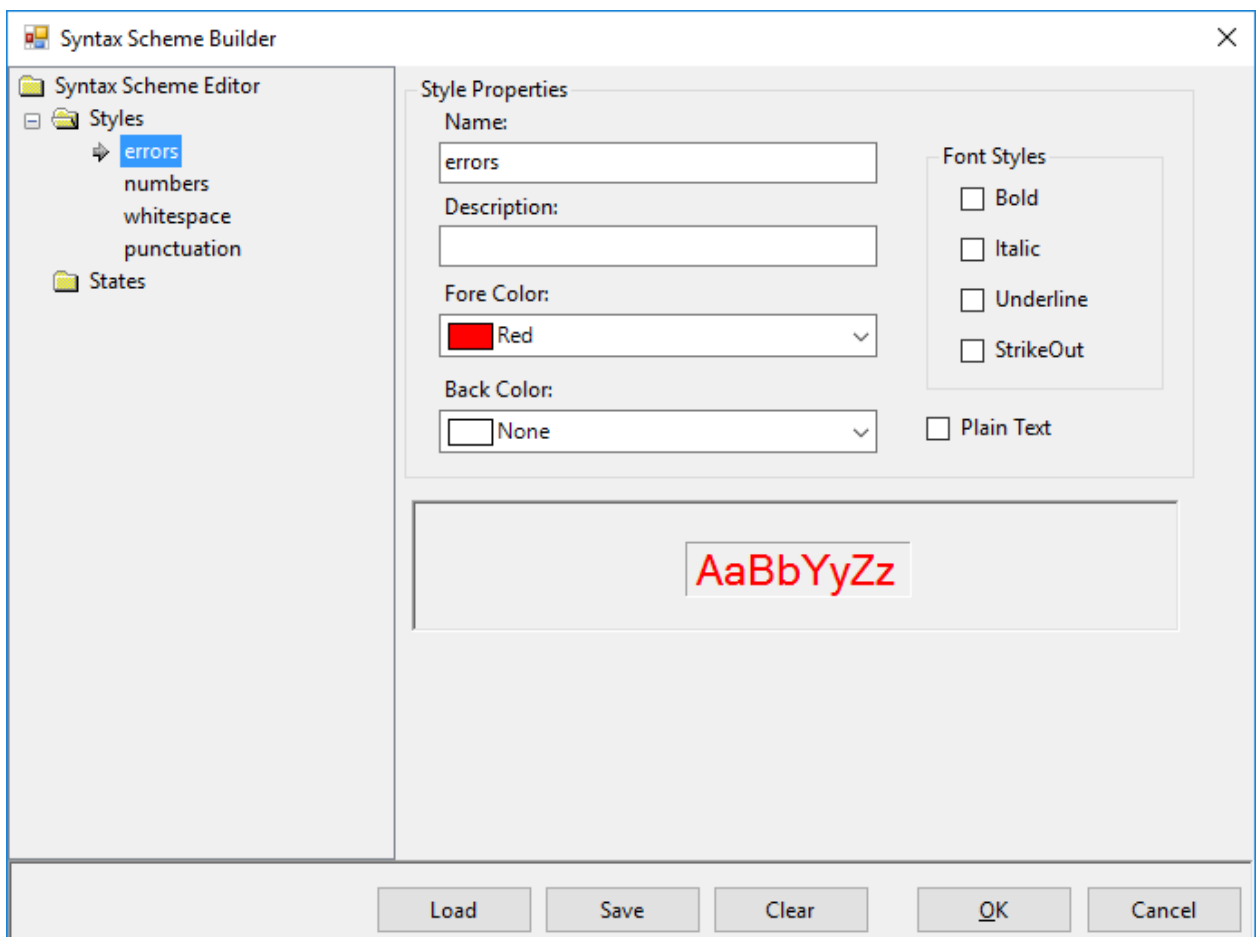
Creating a New Generic Parser

Although the Code Editor is supplied with a collection of parsers, it may be sometimes necessary to create a new one. In this chapter we will develop a completely new parser for some trivial fictitious language. Feel free to skip this section if you

First, let us informally describe the language we are willing to parse. The valid text in this language consists of zero or more groups, enclosed in curly brackets (“{”, “}”), each containing zero or more numbers separated by commas. We want to distinctly highlight punctuation symbols and numbers, and to highlight erroneous input.

The first step is to create a new *Parser* object by dropping it from the toolbox, and assigning it to some *SyntaxEdit* by picking the newly created parser from the list of choices appearing for the *Lexer* property of the editor.

After having done this, we can start exploring the *SyntaxBuilder*. It is invoked by pressing the “...” button appearing for the *Scheme* property of the *parser* object.



Now you should see the *SyntaxBuilder* window appear. If you wanted to use some existing scheme, you would have pressed the *Load* button, however, this time we are going to create a new scheme completely from scratch. After completing it, you can press the *Save* button to make it possible to use this new scheme in other projects.

The next thing to do, after entering optional information about the author and the copyright, is to define syntax highlighting styles used in the scheme. This is accomplished by clicking the right mouse button on the *Styles* node to bring the context menu, and choosing the *Add Style* command.

After creating a style you should give it a name and define its visual attributes. For this example we will need four three styles: *number*, *punctuation*, *whitespace*, and *error*. Let us define numbers to have olive color and italic text style, punctuation symbols to be blue, and errors to have red background and white foreground. The *whitespace* style is defined as having no distinct markup at all.

Then we define the states of the parser. For our example language there will be two states: *default* and *block*. States are defined similarly to styles, by choosing the *Add State* command in the context menu appearing to the *States* node. In turn, states contain syntax blocks, created by the *Add Syntax Block* command from the context menu of a state.

The syntax parser is essentially a state machine, driven by the text. Transition conditions are expressed in terms of regular expressions which are checked against the parsed text at the current position up to the next end of line. Expressions are tried in the syntax block definition order. The first successful match determines the syntax block. The text position is advanced by the length of the match, and the text is assigned the style specified for that syntax block. The matched text is additionally matched against the list of the reserved words associated with this syntax block, and if a match occurs, the style defined by

the *ResWord Style* is used instead of the one defined by the *Style* property. The state of the state machine is changed according to the *Leave State* property of the syntax block, which can specify any of the states, including the same state, in which the syntax block resides, meaning no state transition is to take place.

The state machine for the language we are parsing is described in the following table, and deserves some comments.

The *whitespace* syntax block is only necessary because of the presence of a match all *error* syntax block. In the more common case where no error highlighting is used, no style (which is the same as the *whitespace* style that we have defined) would be used for the text that does not match any of the syntax blocks. The *error* syntax block is the last in the sequence and matches a single character which has not been matched by any of the preceding rules. The *block* syntax block is matched when the opening curly bracket is met. The bracket itself is assigned the *punctuation* style, and the state machine changes its state into the *block* state (note that state name, style name and syntax block style name coincidences are not required).

In the *block* state, the *whitespace*, and *error* syntax blocks serve the same purpose as in the *default* state. *Number* and *comma* syntax blocks cause numbers and commas to have the corresponding styles, and the *end* syntax block, which matches the closing curly bracket, causes the transition back to the default state.

| State | Syntax Block | Regular Expression | Style | Leave State |
|---------|--------------|--------------------|-------------|----------------|
| Default | | | | |
| | whitespace | \s+ | whitespace | <i>default</i> |
| | block | \{ | punctuation | block |
| | error | . | error | <i>default</i> |
| Block | | | | |
| | whitespace | \s+ | Whitespace | <i>block</i> |
| | number | \d+ | Number | <i>block</i> |
| | comma | , | Punctuation | <i>block</i> |
| | End | \} | Punctuation | default |
| | Error | . | Error | <i>block</i> |

Integration with other AlterNET Studio components.

Code Editor can be used as a stand-alone package in the applications that require text editing functionality; it also can be used in conjunction with Form Designer and Scripter packages to provide code editing functionality for code-behind files, writing event handlers and script editing.

Please refer to our AlterNET Studio demo to see how Code Editor, Form Designer and Scripter/Debugger work together.

Licensing

Code Editor requires a valid license to be installed for developing a .NET application that uses its components. We supply evaluation-licenses upon AlterNET Studio installation; these licenses are based on licx files technology provided by Microsoft and are valid for 30 days since first use.

Upon ordering a paid version of our product, you will receive a License key and will be able to activate it on your computer. This key will support a number of activations and it's not transferable between development machines otherwise.

Below is more information about Microsoft license compiler and some discussions related to intent and purpose of licx files, using them with source code control systems, etc.

[https://msdn.microsoft.com/en-us/library/ha0k3c9f\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ha0k3c9f(v=vs.110).aspx)

<http://stackoverflow.com/questions/5628969/how-licenses-licx-file-is-used>

<http://stackoverflow.com/questions/51363/how-does-the-licenses-licx-based-net-component-licensing-model-work>

In a nut-shell, once you drag *SyntaxEdit* on your form, the licx file with the following content will be added to your project under Properties folder:

```
Alternet.Editor.SyntaxEdit, Alternet.Editor.v7, Version=7.1.0.0, Culture=neutral,  
PublicKeyToken=8032721e70924a63
```

In case you create a SyntaxEdit component from code, make sure licx file with such content is added to your project – you can use one from one of our demo projects if needed.

The design-time license is being checked when you work with this component at design-time, or when you compile your project; and the nag screen reminding you about the evaluation mode will appear once in a while. If you run a project compiled with an evaluation version of Code Editor (without launching it from Microsoft Visual Studio debugger), you will see a screen suggesting that the application was created with an evaluation version of Code Editor. Once you activate a paid license using LicenseActivation tool, nag screen will no longer appear.

When the evaluation period expires, you will still be able to compile and run your application from within Visual Studio, however applications created with expired license will not be run in standalone mode.