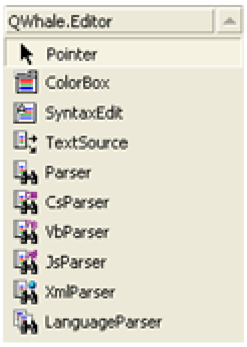
Introduction

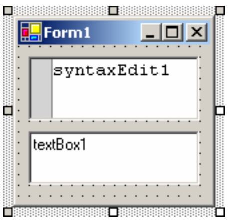
Lots of computer programs work with text in one way or another, and provide the user means to edit it. Despite the tendency in modern GUI applications to provide a multitude of dialog boxes, menus and wizards to enter the data, there are many cases when text remains the most flexible and convenient way of supplying the input to the program. However, this does not imply that editing text should not take advantages of modern achievements in the UI design; on the contrary, during the recent years text editors have become more user friendly and efficient tools than ever before. They have a set of standard features that users have come to expect from the text editor. Regretfully, the text editor control supplied with the *.NET* platform is severely lagging behind those expectations. The *Editor.NET SyntaxEdit* component has been specifically created to close that gap. Just to mention some of the most prominent features: support for large text, unlimited undo/redo, bookmarks, syntax highlighting, outlining, code completion, URL handling, gutter and margin indication, word-wrapping, fully customizable keyboard handling.

Installation



The installation procedure for the *Editor.NET* pack is very straightforward. Just unzip the archive into a location of your choice, open the toolbox in the Microsoft Development Studio IDE, create new page, then right click on it and select “Add/Remove items”. In the dialog, choose Browse and select *QWhale.Editor.dll*. Repeat the latest step for *QWhale.Common.dll*, *QWhale.Syntax.dll* and *QWhale.Syntax.Schemes.dll*. After doing this, you will have two new components on the toolbox that are directly related to the edit component: *SyntaxEditor* and *TextSource*, six components related to syntax parsing: *Parser*, *CsParser*, *VbParser*, *JsParser*, *XmlParser* and *LanguageParser*, and one unrelated bonus control: *ColorBox*.

Getting Started



Now, having installed the *Editor.NET* component pack, let us explore it step-by-step. The first thing to do after creating a new application is to place the *SyntaxEdit* component on the form. The *SyntaxEdit* component is the central component in the set, and in many cases it will be the only component 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.

Having dropped a new *SyntaxEdit* on the form you can go on and satisfy your curiosity by examining its properties. Some of them are similar to those of the *TextBox*, some of them will be new, however there probably will be many of intuitively understandable ones. Do not hesitate, try them out (the influence of some of them will be noticeable in design-time, some of them will have effect only in run-time) – most of them will work just the way you expect them to. After you go through this manual, you can look at the two supplied demo applications (some of you might have already done this before even opening this manual), which cover most of the functionality of the *SyntaxEdit*.

Now, to have some basic useful functionality, we have 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 *SyntaxEdit*’s *LoadFile* method. It can be called with one, two or three parameters. The first one specifies the name of the file to be loaded into the control. The optional second parameter describes the format filter to be used when loading and saving the text (plain text, rtf, html, and xml are supported). By default, the text is loaded as plain text. The third parameter specifies encoding.

E.g.:

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

Saving of the text is performed in a similar way:

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

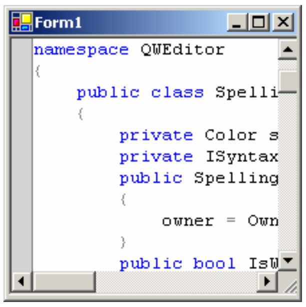
And, of course, you can use streams instead of files, by substituting the previous two functions by *LoadStream* and *SaveStream*.

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 fancy features of the *SyntaxEdit*, we must examine one of its fundamental concepts.

As you remember, there are three editor-related components in the pack. However, we have used only one of them, namely the *SyntaxEdit*. The other is non-visual components which can be explicitly created in the design time: *TextSource*. The *SyntaxEdit* component itself does not store the text it edits. This task are offloaded to the *TextSource* component. Even when the *Source* property of the *SyntaxEdit* is not assigned, the latter contains an internal instance of the *TextSource* component. At first, such a separation appears to be useless, however, it in fact allows easy implementing of 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 *SyntaxEdit*s 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* itself.

Syntax Parsing



```
namespace QWEditor
{
    public class Spelli
    {
        private Color s
        private ISyntax
        public Spelling
        {
            owner = Own
        }
        public bool IsW
```

Text parsing is performed by the *Parser* non-visual component. It controls the syntax highlighting and text outlining. If no parser is assigned, *SyntaxEdit* does not perform any parsing related functions, such as syntax highlighting and code outlining. 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*, or the *TextSource*. Parsing is performed using a finite-state automaton driven by regular expressions matching the parsed text. Although we must admit that creating a good parser for some complex language can be a tricky endeavor, you will rarely, if ever, have to do it yourself. The *SyntaxEdit* 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, however, unless you are the kind of person who is comfortable with writing them by hand in a plain-text editor, the built-in visual design-time parser editor will greatly simplify the process of their creation. In case you need most of these languages in your application, you can consider using *LanguageParser* component, which contains all these schemes in form of .dll resource.

Editor.NET package comes with few dedicated parsers for C#, J#, VB.NET and XML languages. These parsers do not use regular expressions when performing lexical analysis of the text, but use hard coded parsing algorithms instead. With this approach, dedicated parsers work much faster than regular-expression based ones. Also using these parsers it's possible to implement features found in *Visual Studio.NET* editor such as automatic code outlining, code completion, smart formatting and error underlying. These features will be discussed later in this manual.

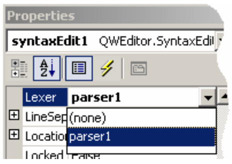
Now that we understand the purpose of all the components in the pack, let us go over the major features of the editor and learn how to use them.

Creating a New Parser

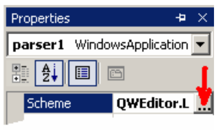
Although the *SyntaxEdit* is supplied with a collection of parsers, it may be sometimes necessary to create a new one. In this chapter we will develop a completely new parser for some trivial fictitious language.

At 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. Please note, that the syntax parser framework is not complex enough to do complete parsing of the text which might be required to catch every possible error, however, the level of detail achieved is the optimal balance between usefulness and complexity.

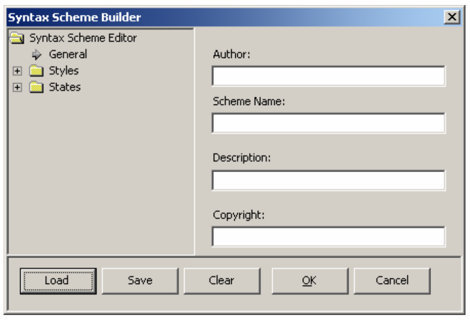
The first step is to create a new *Parser* object by dropping it from the toolbox, and assigning it to some *SyntaxEditor*, by picking the newly created parser from the list of choices appearing for the *Lexer* property of the editor.



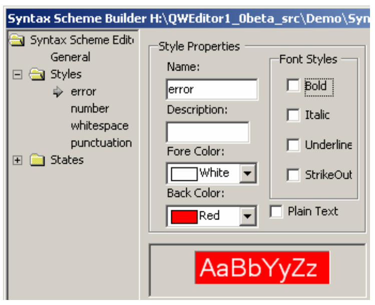
After having done this, we can start exploring the *SyntaxBuilder*. It is invoked by pressing the "…" button appearing for the *Scheme* property of the *parser* object.



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 pleasing your vanity and making your lawyers happy by entering the 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 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	*default*
	block	\{	punctuation	**block**
	error	.	error	*default*
Block				
	whitespace	\s+	Whitespace	*block*
	number	\d+	Number	*block*
	comma	,	Punctuation	*block*
	End	\}	Punctuation	**default**
	Error	.	Error	*block*

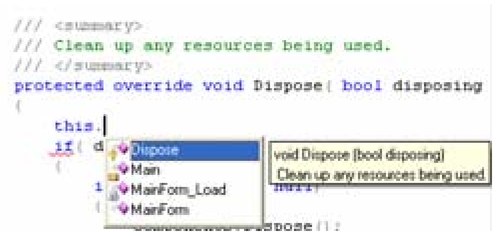
It would be possible to further elaborate the state machine to check that numbers are actually separated by commas, however in the real-life applications what we already have is good enough, so this possible improvement is left as an exercise for the inquisitive reader.

After this brief introduction to syntax parsers, you should have no problems with examining the existing parsers in search for solutions for your own parser.

Dedicated Parsers

Editor.NET component pack comes with several dedicated parsers, each one designed to perform syntax highlighting for certain language. Detailed explanation of how to create a new parser lies out of scope of this document, so we will discuss existing parsers instead. Currently there are four dedicated parsers: *CsParser* for *C#*, *JsParser* for *J#*, *VbParser* for *VB.NET* and *XmlParser* for *Xml*. All these parsers are derived from the *SyntaxParser* class implementing *ISyntaxParser* interface, which represents all properties and methods needed for automatic code outlining, code completion, smart formatting, and error highlighting. In fact these parsers do much more than just syntax highlighting. The first three parsers perform complete syntax parsing of the source code to build syntax tree, which is used to implement all mentioned features, while *XmlParser* supports only code outlining and smart formatting. We plan to extend the set of syntax parsers with parsers for *SQL*, *Jscript* and *VbScript* in the future releases.

Code Completion



```
/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing
{
    this.
    if( d
```

void Dispose (bool disposing)
Clean up any resources being used.

Although the main purpose of an editor is to be a convenient tool for the user to enter whatever text he or she desires, quite often a gentle guidance from the editor can significantly improve the effectiveness of the work process. When editing a text which has some structure (e.g. computer program in some language), there are often a 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, there is a key combination (*Ctrl+J* in default keymapping) to activate the 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 details in the following subsections.

Quick Info

To give the understanding of the *code completion* architecture of *SyntaxEdit* we will start from the simplest one. Please note that the name “*code completion*” can be somewhat misleading as it does not always implies 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*.

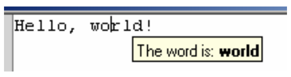
The members marked as "[in]" contain the information provided by the *SyntaxEdit*, and the "[out]" members can be filled by the handler.

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 after all this theoretical background it is time to see some working code.

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

    SyntaxEdit edit = (SyntaxEdit)sender;
    Point Pt = edit.Position;
    int Left, Right;
    edit.Lines.GetWord(Pt.Y, Pt.X, out Left, out Right);
    if(Left == 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 = message + word;
      p.BoldStart = message.Length;
      p.BoldEnd = p.Text.Length;

      e.StartPosition.X = Left;
      e.EndPosition.X = Right;
      e.EndPosition.Y = 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 *SyntaxEditor*.



```
Hello, world!
The word is: world
```

List Members

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

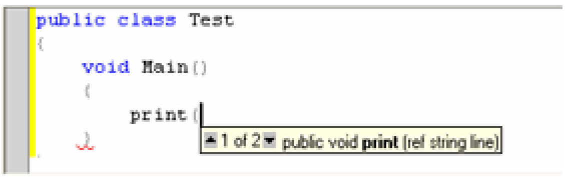
The purpose of the *members* is to permit the user to quickly choose one of the entities of the program (procedure, method, variable &c) relevant to the current context. This may mean listing all the available functions in the statement context of a procedural language, all the type compatible functions 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.

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

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

          IListMember m = p.AddMember();
          m.Name = "print";
          m.DataType = "void";
          m.Qualifier = "public";
          m.ImageIndex = 1;
          m.Description = "void print(ref string line)";

          m = p.AddMember();
          m.Name = "evaluate";
          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 *ParameterInfor* provider is used to respond to *ParameterMembers* code completion request. It can be activated by pressing *Ctrl+Shift+Space* key combination, or programmatically, by calling the *ParameterInfo* method of the *SyntaxEdit*.



```
public class Test
{
    void Main()
    {
        print(
```

▲1 of 2▼ public void **print** (ref string line)

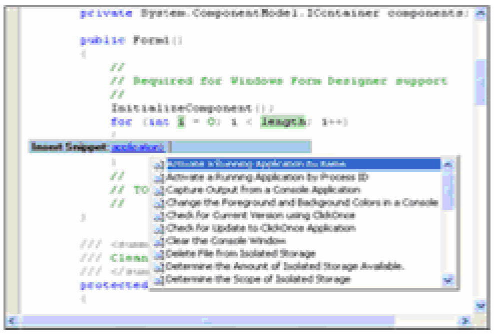
The purpose of the *parameter info* is to show a tooltip describing the parameters of the method call under cursor. Note that this only works for languages with traditional syntax for specifying parameters, i.e. method(arg1, arg2…).

```
private void syntaxEdit1_NeedCodeCompletion(object sender,
QWhale.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.AddMember();
          m.Name = "print";
          m.DataType = "void";
          m.Qualifier = "public";
          m.ImageIndex = 1;
          m.Parameters = new ParameterMember[1];
          m.Parameters[0] = new ParameterMember();
          m.Parameters[0].DataType = "string";
          m.Parameters[0].Name = "line";
          m.Parameters[0].Qualifier = "ref";
          m.ParamText = "(ref string line)";

          m = p.AddMember();
          m.Name = "print";
          m.DataType = "void";
          m.Qualifier = "protected";
          m.ImageIndex = 2;
          m.Parameters = new ParameterMember[1];
          m.Parameters[0] = new ParameterMember();
          m.Parameters[0].DataType = "string";
          m.Parameters[0].Name = "expression";
          m.ParamText = "(string expression)";

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

Code snippets

The code snippets is 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 pieces of text.



```
private System.ComponentModel.IContainer components;

public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();
    for (int i = 0; i < length; i++)
    {
Insert Snippet: application
    }
    //
    // TO
    //
}

/// <sum
/// Clean
/// </sum
protected
{
```

- Activate a Running Application by Name
- Activate a Running Application by Process ID
- Capture Output from a Console Application
- Change the Foreground and Background Colors in a Console
- Check for Current Version using ClickOnce
- Check for Update to ClickOnce Application
- Clear the Console Window
- Delete File from Isolated Storage
- Determine the Amount of Isolated Storage Available.
- Determine the Scope of Isolated Storage

If some fields are declared in the code snippets, editor allows modifying their values causing updating field value inside whole snippet.

The following picture illustrates the usage of the code snippets.

Automatic Code Completion Invocation

Recently, many users have become accustomed to having the *code completion* appear automatically as the type, for example after the type "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 is customary to show those popups only if the user stops input for some short period of time after typing the activating symbol ("." Or "(").

The easiest way to provide automatic code completion is to use dedicated parser already supporting this feature, for example *CsParser, JsParser, VbParser, VbScriptParser and JavaScriptParser.*. All these parsers are derived from the *NETSyntaxParser* class, which encapsulates common functionality for all .NET based languages. 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. We tried to make this feature behave as close as possible to the Visual Studio .NET editor, so it works in intuitively understandable way. When these parsers are used, you still can control some aspects of code completion, for example delay before code completion window appears, using *NeedCodeCompletion* event, which will be discussed later. Moreover you can register own types and objects, namespaces and assemblies for code completion using *CompletionReprository* property of *SyntaxParser*. For example, if you need to provide code completion for types declared in *System.Drawing* namespace, you can use the following code:

```
csParser1.RegisterNamespace("System.Drawing");
```

By default, *NETSyntaxParser* registers three assemblies for code completion: *System*, *System.Drawing* and *System.Windows.Forms*. If you need to provide code completion for assemblies declared in other assemblies, you need to register these assemblies this way:

```
csParser1.RegisterAssembly("System.Xml");
or
csParser1.RegisterAllAssemblies();
```

You may need to register types for code completion that are not declared in the assembly, but present in form of source code somewhere else. To register such types 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 CsParser();
parser.Strings = new SyntaxStrings();
((SyntaxStrings)parser.Strings).LoadFile("MyFile.cs");
parser.ReparseText();
csParser1.CompletionReprository.RegisterSyntaxTree(parser.SyntaxTree);
```

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

```
private void syntaxEdit1_NeedCodeCompletion(object sender,
QWhale.Syntax.CodeCompletionArgs e)
{
  if(e.CompletionType == CodeCompletionType.ListMembers ||
    e. CompletionType == CodeCompletionType.None &&
    e.KeyChar == '.')
  {
    // same as in the above example explaining how to implement list
members
    ...
    e.Interval = (e.CompletionType != CodeCompletionType.None)
      ? 0 : 500;
  }
  if(e.CompletionType == CodeCompletionType.ParameterInfo ||
    e. CompletionType == CodeCompletionType.None &&
    e.KeyChar == '(')
  {
    // same as in the above example explaining how to implement parameter
info
    ...

    e.Interval = (e.CompletionType != CodeCompletionType.None)
      ? 0 : 500;
  }
}
```

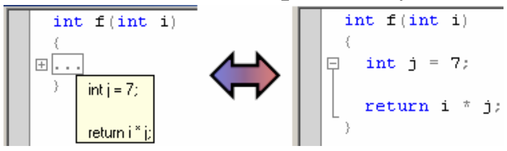
Depending on the kind of the language you are working with, and whether you are using some complete library to work with that language, or do everything yourself, the actual information on symbols will be retrieved in different ways:

- 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 you are doing and what exactly to do to acquire the information necessary for *code completion* to work;
- if you 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.

Outlining

The *SyntaxEdit* 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, e.g. 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



```
int f(int i)
{
...
}
```

```
int j = 7;
return i * j;
```

```
int f(int i)
{
  int j = 7;

  return i * j;
}
```

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* 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 *SyntaxEditor*. 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, e.g.:

```
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 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 synch with the text, which can be a non-trivial undertaking. Therefore, the second approach might be a better choice.

Indirect Definition of Outline Sections Using the Syntax Parser

To overcome the aforementioned drawback, the syntax parsing framework of the syntax editor 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 *QWhale.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 smooth editing experience, the implementation should be relatively fast.

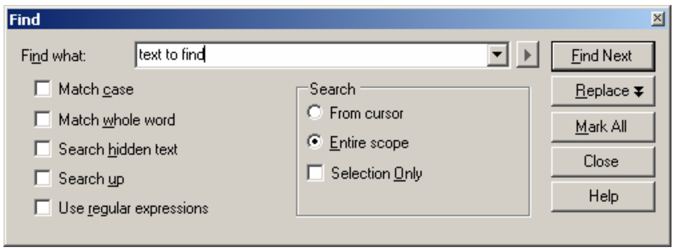
Editor.NET comes with four parsers already supporting automatic outlining for *C#, J#*, *VB.NET* and *Xml* languages.

The following example defines a parser class the 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 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;
            }
    }
```

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. This function is ready to use out-of-the-box: when the user presses *Ctrl+F* key combination, the familiar looking 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. ExecuteSearchDialog();
```

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.

E.g.:

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

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);
syntaxEdit1.Position = pos;
```

Selection

Just as almost any serious 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 later can be created by navigating the text with navigation keys holding *Shift* and *Alt* keys held together.

Selection.BackColor and *Selection.ForeColor* define the background and the foreground colors used to mark the currently selected text. *Selection.InActiveBackColor* and *Selection.InActiveForeColor* are used instead 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 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.
- *SelectLineOnDblClick* allows 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 overwrite the currently selected text.
- *SmartFormat* allows formatting block 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* casuses Edit control to draw border around selection
- *SelectLineOnTripleClick* allows to select whole line rather than single word by triple clicking the mouse

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 specified rectangular area.
- *SelectAll* selects 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.

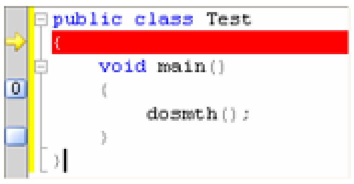
In fact, if some action can be performed by the user, it can also be performed programmatically. Just to give a better idea:

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

This code encloses the currently selected text in brackets.

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 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):



```
public class Test
{
    void main()
    {
        dosmth();
    }
}
```

```
new_image = syntaxEdit1.Gutter.Images.Images.Count;
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 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, 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:

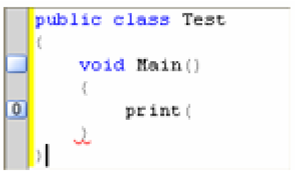
```
syntaxEdit1.Source.LineStyles.RemoveLineStyle(0);
```

Appearance of the gutter is controlled by the following properties: *Width*, *BrushColor*, *PenColor* and *Visible*. *Width* 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. *SyntaxEditor* 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 properties : *LineNumbersStart*, *LineNumbersForeColor*, *LineNumbersBackColor*, *LineNumbersAlignment*, *LineNumbersLeftIndent* and *LineNumbersRightIndent*, which are intuitively understandable.

Like Microsoft Visual Studio 2005 editor, *SyntaxEditor* implements ability to visually track modified lines. To enable this feature you need to turn *Gutter.Options. PaintLineModificators on.* When LineModificators 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 *LineModificatorChangedColor* and *LineModificatorSavedColor* properties.

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

Bookmarks



```
public class Test
{
    void Main()
    {
        print(
```

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't be more than 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)

Of course, like everything else 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 *syntaxEdit.Source.BookMarks.ClearAllBookmarks* method.

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

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

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

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

If you want to, you can change the bookmark indicator image like this:

```
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.)

Keyboard Mapping

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

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 then the *Tab*.

The state diagram can have arbitrary complexity (i.e. states can be nested), however, take your user in consideration: do not make things too confusing, there is probably always a clearer way.

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 editor the keymapping with 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,
((KeyList)syntaxEdit1.KeyList).Handlers.selectAllEvent);
```

Spellchecker Interface

The *SyntaxEdit* supports the spell-as-you-type spellchecker 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,
QWhale.Editor.WordSpellEventArgs e)
{
    e.Correct = e.Text.Length > 3;
}
...
this.syntaxEdit1.WordSpell += new
QWhale.Editor.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 the real life you would need to use some third-party software 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.

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

URL handling

The *SyntaxEdit* can be setup 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 (e.g. 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,
QWhale.Editor.UrlJumpEventArgs e)
{
    if(is_our_url(e.Text))
    {
          process_url(e.Text);
          e.Handled = true;
    }
}
```

Printing And Exporting

Although not so obvious at the first glance, but implementing the printing of formatted text from scratch is not a trivial undertaking. Same goes with the exporting text into such formats as RTF and HTML. The reason for this is the multitude of tiny details that need to be taken to consideration to make the result look good. The standard multi-line text box does not provide any support for this whatsoever. In contrast to this, *SyntaxEdit* comes with a complete support for printing and exporting.

Exporting can be performed as simple as this:

```
syntaxEdit.SaveFile(FileName, ExportFormat.Rtf);
```

Printing is performed and configured via the *Printing* property of the *SyntaxEditor*.

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

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

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 *GetPrintTag* event, for example:

```
private void syntaxEdit1_GetPrintTag(object sender,
QWhale.Editor.PrintTagEventArgs e)
{
    if(e.Tag == "\\[tag]")
    {
          e.Text = "tag replacement text";
          e.Handled = true;
    }
}
```

XML Serialization

When saving and loading using the XML format, using the code similar to the one seen in the previous section:

```
syntaxEdit.SaveFile(FileName, ExportFormat.Xml);
```

and

```
syntaxEdit.LoadFile(FileName, ExportFormat.Xml);
```

more than just saving or loading the text is performed. The usage of the XML format causes the entire state of the editor and all related components (text source and syntax parser) to be serialized/deserialized.

The typical use for this feature is preserving the state of the workspace between sessions (however, keep in mind that if the main storage format for your files is plain text, and the XML format is used just for state preservation, the state may become outdated if the plain text file gets modified by some external means), or to provide an automatic crash recovery by periodically saving the state (and deleting it when a normal save is performed), and checking for the presence of state files during the start up of the application.

Global Settings

If the application contains more than one instance of the editor, it is quite often desired for to share their UI settings, and to provide the user with a centralized facility to manage them. The *SyntaxSettings* class exists to aid fulfilling of this requirement.

This class 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

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

```
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:

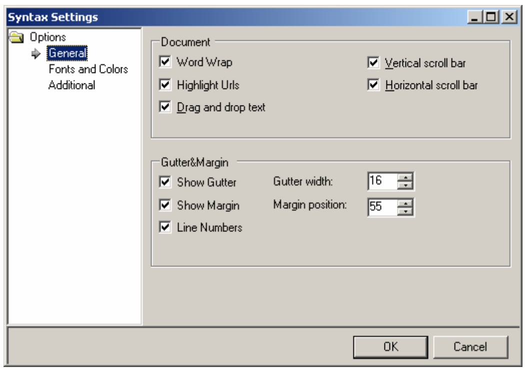
```
GlobalSettings.SaveFile("GlobalSettings.xml");
```

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 *SyntaxEditor* distribution includes a ready-to-use settings dialog.


Syntax Settings

Options
- General
- Fonts and Colors
- Additional

Document
- [✓] Word Wrap
- [✓] Highlight Urls
- [✓] Drag and drop text
- [✓] Vertical scroll bar
- [✓] Horizontal scroll bar

Gutter&Margin
- [✓] Show Gutter
- [✓] Show Margin
- [✓] Line Numbers
- Gutter width: 16
- Margin position: 55

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

```
using QWhale.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);
}
```

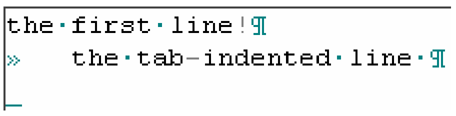
or just use the *SyntaxEdit.DisplayEditorSettingsDialog* method.

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.

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.


```
the·first·line!¶
»    the·tab-indented·line·¶
_
```

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 *Scrolling.Options* to allow *SyntaxEditor* to split it's content. Note that *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 options works only if *SystemScrollBars* is on.
- *AllowSplitHorz* – allows displaying horizontalsplitting button in the scroll area. This options works only if *SystemScrollBars* is off and control has *Dock* property set to *DockStyle.Fill.*
- *AllowSplitVert* – allows displaying vertical splitting button in the scroll area. This options 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.

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 line 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 usefull to display horizontal and vertical rulers, which will allow user 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 play back them later. Note that mouse input is not recorded.

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

Localization of dialogs

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

```
Using QWhale.Common;
...
StringConsts.Localize(new System.Globalization.CultureInfo(“de”));
```

Conclusion

We believe that this manual contains enough information to get you started with the SyntaxEdit and covers all of its major features, however, if you have some problems for which there is no solution in the manual, feel free to contact our technical support at support@qwhale.net.