C# Tutorial #1: A complete Windows Forms application (part 1)

The intention of this tutorial is to put together a simple Windows Forms application which actually does something useful. I will be writing these pages as I go along, learning as I go, so you will see all the pitfalls which I fall into and hopefully avoid doing the same thing.

So no comments about my coding, please! With this project, I'll be happy if it actually works, even if the code isn't the best, most concise or whatever.

Steps to take

1. Create a new Windows application

Create a new Windows Forms project in C# and give it the name SimpleEd. Incidentally, if you're following along with this, I strongly suggest you use the same object and variable names that I do. Otherwise, you may find in some future stage that I'm referring to some variable which doesn't seem to exist on your system.

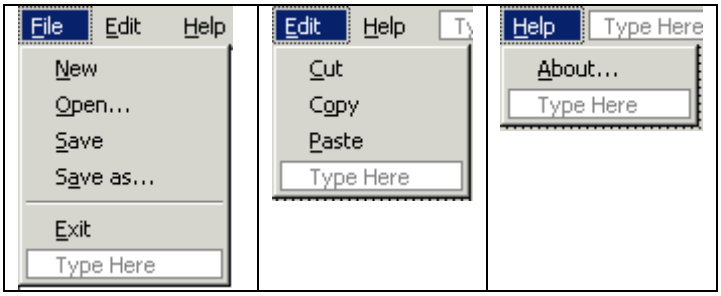
2. Set the form's properties

Select the form, currently named form1, and in the Properties window change the Name property to 'frmMain', the Text to 'Simple Editor', and the StartPosition to CenterScreen so that the window opens in the middle of the screen.

3. Add a menu

Next we'll add a basic menu structure. Click the MainMenu control in the toolbox and then click the form. The menu bar appears at the top of the form and a box labelled 'Type Here' may already be visible. If it isn't, click in the menu and one will appear. Just do what it says and type the string '&File' (without the quotation marks). The ampersand causes the menu accelerator key to be underlined.

You can now add more top level menus and menu items. You need to end up with a structure which looks like this:



This gives you three top-level menus each with one or more menu entries. You can add a separator, as in the File menu, by typing a hyphen as the menu text - Visual Studio expands this to the width of the menu. Note the accerlator keys for the menus.

In .NET all top-level menus and menu entries are objects of type System.Windows.Forms.MenuItem. When you create a menu, VS names each menu item with a standard name such as menuItem1, menuItem2, etc. This isn't terribly helpful and you need to rename each menu entry (except maybe the separators!).

To do this, click the menu item to be renamed in the design window, then rename it in the Properties window. You can also use the Properties window to add keyboard shortcuts such as Ctrl-C for copy: click the arrow in the righthand column of the Shortcut property and choose the shortcut from the dropdown list. I used the following names and shortcuts:

mnuFile
mnuFileNew (Ctrl-N)
mnuFileOpen (Ctrl-O)
mnuFileSave (Ctrl-S)
mnuFileSaveAs
mnuFileExit

mnuEdit
mnuEditCut (Ctrl-X)
mnuEditCopy (Ctrl-C)
mnuEditPaste (Ctrl-V)

mnuHelp
mnuHelpAbout

Note that you don't have to type 'Ctrl-N' to show this in the menu; this will appear automatically if the ShowShortcut property for the menu item is set to True.

4. Add a text box

For the text box, I prefer to use the RichTextBox control if for no other reason that it does not have the 64K text limit that the standard TextBox does under Windows 95/98. So add such a control from the toolbox and rename it rtfMain.

Position the box in the top left corner of the form's client area (i.e. under the menu bar). You can check to see if it's in the right place by checking its Location property - this should read 0,0. The size of the box doesn't matter as we have to handle the possibility of the user resizing the window, anyway.

You might also want to change the font used by the text box. I chose Arial 10 point, but you can select anything.

That's it for the initial design. In the next page of this tutorial, we'll add some code to get the program some functionality.

C# Tutorial #1: A complete Windows Forms application (part 2)

With the basic form and menus set up, we need to add some code to add functionality.

Firstly we'll add three basic sections:

1. Handle the form resize event.

The problem here is that when the user resizes the form, the text box will also have to be resized. We can catch this in frmMain's Resize event, so add an event handler for this event (see Adding event handlers to WinForms controls (part 2) for details of how to do this).

What we want to do is to set the size of the text box to be the same size as the client area of the form - i.e. all the space below the menu excluding the form borders. This needed some calculations in VB6 but fortunately it's easy in .NET as now we can use the form's ClientSize property. So we end up with:

```
private void frmMain_Resize(object sender, System.EventArgs e)
{
rtfMain.Size=this.ClientSize;
```

```
}
```

Now the text box's size is reset whenever the form's Resize event is triggered, and since this happens when the window is first drawn we can forget about setting the text box size in the design window.

2. Exit the application

This is straighforward. We add a Click event handler to the menu item mnuFileExit, in the usual way. To exit the program, we can just call the Exit method of the Application object (or indeed just close frmMain in this case).

But there is something else we need to do. If the user has changed the contents of the text box, we don't want to exit without the user having the chance to cancel the action, in case his/her work is lost. The RichTextBox control has a Modified property that we can use for this. If that property is True, the contents have been modified, otherwise they have not. So we can code the routine like this:

```
private void mnuFileExit_Click(object sender, System.EventArgs e)
{
// exit the application but query user if the text in the box has changed
if (rtfMain.Modified==true ) {
if (MessageBox.Show("Text has changed - really exit?", "Confirm exit",
MessageBoxButtons.YesNo, MessageBoxIcon.Question)==DialogResult.Yes) {
Application.Exit();
}
} else {
Application.Exit();
}
}
```

You can improve on that terse message if you like! Note the use of the message box return value. I will be looking at message boxes under C# in a later how-to. The code is self-explanatory and the program exits unless the text box content has been changed and the user confirms the action. You can compile the code, run the program and type a few characters in the box to confirm that this works.

3. Create a new document

This is very similar to exiting the application except that all we want to do is clear the existing text, and the RichTextBox control has a Clear() method which does this for us. All we need to do is create a Click event handler for the mnuFileNew menu item in the usual way, then copy and paste the body of the Click event function for the mnuFileExit handler into our new handler function.

We don't want the application to exit when we click File | New, so change both occurrences of Application.Exit() to rtfMain.Clear(). Finally, change the message in the MessageBox call to something more appropriate such as 'Text has changed - really create a new document?' and our handler is complete. Compile it and check that it works.

The next stage is to add code to load and save files, which the next page of this tutorial will do.

C# Tutorial #1: A complete Windows Forms application (part 3)

At this point we need to add some functionality to allow loading and saving of files. Without that, the editor is not much use to anyone.

Steps to take

1. Add an open file function

In Visual Basic 6, this was a slight pain because the File Open common dialog box was a little tricky to set up and use. In .NET it's easy. From the toolbox, add an OpenFile Dialog to the form. The dialog does not appear on the form, but in the space below it. Click on it to select it, and change its name to dlgOpen. All the other properties we will set from code.

Now add a click event handler to the File | Open menu item. We need to code the following:

checking to see if the current text has been changed and not saved, as opening a new file will overwrite the current text

setting the dialog box properties

opening the dialog box and checking for a valid filename when it returns

keeping the returned filename in case we have to save the file

opening the file

displaying the filename in the form's caption

First we need to be able to store the opened file's name. To do this, add a variable to the class definition (the bit which starts 'public class frmMain: System.Windows.Forms.Form'. I put the following declaration at the end of this block just above the line 'public frmMain':

```
private string m_sfileName="";
```

When the program first starts, we don't have an open file so there is no filename. We need to reflect that in the form's caption. Add a function to the frmMain class like so:

```
private void clearFormText()
{
this.Text="Simple Editor - [unnamed]";
}
```

This just reminds the user that the text hasn't been saved to a file yet. This function should be called when the form loads (i.e. in its constructor) so add a call to the function from the constructor (the bit where it says 'TODO: Add any constructor code after InitializeComponent call').

2. Add code to open a file

This is fairly straightforward. Here's the code to add to the Click event handler for mnuFileOpen:

```
private void mnuFileOpen_Click(object sender, System.EventArgs e)
{
bool bDoOpen = true;

// show an Open File dialog and open the file, but
// check that the current text has not changed first
if (rtfMain.Modified==true ) {
if (MessageBox.Show("Text has changed - really open another document?",
"Confirm file open", MessageBoxButtons.YesNo,
MessageBoxIcon.Question)==DialogResult.No)
bDoOpen=false;
}
```

```
if (bDoOpen==true) {
// get a file to open and open it
dlgOpen.Filter="Text files (*.txt)|*.txt|All files (*.*)|*.*";
dlgOpen.FilterIndex=1;
if (dlgOpen.ShowDialog()==DialogResult.OK && dlgOpen.FileName.Length>0) {
rtfMain.LoadFile(dlgOpen.FileName, RichTextBoxStreamType.PlainText);
rtfMain.Modified=false;
m_sfileName=dlgOpen.FileName;
setFormText(m_sfileName);
}
}
}
```

This should all be fairly obvious. We ask the user to confirm opening another file if the current text has been changed and not saved. A local variable is used to hold the user's response and we only open a file if the user clicked 'Yes' in the message box. We then set the OpenFile dialog's filter and filter index properties and call its Show method. If the user clicks OK and has selected a file, we open it using the RichTextBox's LoadFile method. Finally, we mark the file as unchanged, store the filename for later use, and call another function - setFormText, which sets the form's caption to show the filename. This function is explained later on this page.

3. Modify the File | New event handler

Remember that we set this handler just to clear the contents of the text box. We need to add two new statements to the handler to clear the form caption (because we don't have a filename with a brand new document) and clear the filename variable itself. Add these two lines after each occurrence of the line rtfMain.Clear():

```
m_sfileName=""; // clear the file name
clearFormText(); // and reset the form caption
```

4. Add Save and Save as... event handlers

Add Click event handlers for the Save and Save as... menu items.

When the user clicks File | Save, we may or may not have a filename. If we do then we can just save the file using that name, but if we don't, we'll have to prompt the user for a name. The Save handler is simple:

```
private void mnuFileSave_Click(object sender, System.EventArgs e)
{
// save the file if we have a filename for it
if (m_sfileName!="")
rtfMain.SaveFile(m_sfileName, RichTextBoxStreamType.PlainText);
else
mnuFileSaveAs_Click(sender, e);
}
```

The SaveFile method of the RichTextBox control actually saves the file using the filename we stored when it was opened (or last saved). If there is no filename to use, we just call the Save as... handler. This will require a SaveFile dialog, so add one to the form from the toolbox and rename it to dlgSave.

```
private void mnuFileSaveAs_Click(object sender, System.EventArgs e)
{
// if we have a filename, use that in the dialog, otherwise use
// a generic name
```

```
if (m_sfileName!="")
dlgSave.FileName=m_sfileName;
else
dlgSave.FileName="SimpleEd1";

dlgSave.Filter="Text files (*.txt)|*.txt|All files (*.*)|*.*";
dlgSave.FilterIndex=1;
dlgSave.DefaultExt="txt";
if (dlgSave.ShowDialog()==DialogResult.OK && dlgSave.FileName.Length>0) {
rtfMain.SaveFile(dlgSave.FileName,
RichTextBoxStreamType.PlainText);
rtfMain.Modified=false;
m_sfileName=dlgSave.FileName;
setFormText(m_sfileName);
}
}
```

Again, on entry to this function we may or may not have a filename to use. If we do, then we can display it in the filename box of the SaveFile dialog; if we don't, we'll use a generic name rather than leaving it blank. Then we set the filter and also the DefaultExt property, which will add the extension '.txt' to the filename if the user doesn't specify an extension. After that it's exactly the same as the File | Open event handler.

5. Set the form caption

Finally, there's that setFormText function, which takes the filename and changes the form's caption to reflect the current file. The problem is that we have the full path and file name, and all we really need is the actual file name without the path. We could parse this ourselves but the .NET framework has a class to do it. This is the FileInfo class and we just need to create an instance of this as follows:

```
private void setFormText(string sFile)
{
// create an instance of a FileInfo class and use
// it to get the file's name without its path
System.IO.FileInfo fInfo;
fInfo = new System.IO.FileInfo(sFile);
this.Text="Simple Editor - [" + fInfo.Name + "]";
}
```

The function takes the fully qualified filename as a parameter, and uses the FileInfo class to extract the file's base name. This is then displayed in the form's caption.

We now have some genuine functionality, but in the next instalment, we will start to add some actual usefulness to the program via the Edit menu.

C# Tutorial #1: A complete Windows Forms application (part 4)

We can now load and save files, but we need to add some editing functionality to the program. We'll work on the edit menu on this page.

Steps to take

1. Add word wrap capability

If you loaded a file into the program, you may have seen that by default the text is word-wrapped. We definitely want the ability to turn that on and off. The first thing is to add a word wrap menu item at the bottom of the edit menu, so do that now; you might also want a separator bar between the first three entries and the new word wrap item. Make the item text 'Word wrap' and its name mnuEditWordWrap.

Add a Click event handler for mnuEditWordWrap. One thing we must do is arrange for the menu item to be checked if word wrap is on, and unchecked if it is off. We therefore need to be sure what state word wrap is in when we start the program. In the frmMain constructor, add the following two lines:

```
rtfMain.WordWrap=false; // turn word wrap off
mnuEditWordWrap.Checked=false;
```

This means that when the program is first run, word wrap is always off, and the menu item is unchecked. To toggle word wrap, we'll use the Click event handler:

```
private void mnuEditWordWrap_Click(object sender, System.EventArgs e) {
// turn word wrap on and off
if (rtfMain.WordWrap==true) {
rtfMain.WordWrap=false;
mnuEditWordWrap.Checked=false;
} else {
rtfMain.WordWrap=true;
mnuEditWordWrap.Checked=true;
}
}
```

This just turns word wrap off if it's on, and on if it's off, altering the menu item Checked property accordingly.

2. Cut, Copy, & Paste

These three items are linked so we will consider them together. Add Click event handlers for the three menu items. The code for each is very simple:

```
private void mnuEditCut_Click(object sender, System.EventArgs e) {
// cut the selected text (if any) to the clipboard
rtfMain.Cut();
}

private void mnuEditCopy_Click(object sender, System.EventArgs e) {
// copy the selected text (if any) to the clipboard
rtfMain.Copy();
}

private void mnuEditPaste_Click(object sender, System.EventArgs e) {
// paste the text in the clipboard into the document
rtfMain.Paste();
}
```

We are just using methods of the RichTextControl to do this. But there is something else to do to complete this aspect of the program. If there is no selected text, the user shouldn't be able to cut or copy anything. Likewise, if there is no text in the clipboard, it shouldn't be possible to paste anything. We can enable or disable these menu items before the user sees them by adding an event handler to the mnuEdit menu item itself. I chose to use the Select event since that event is triggered even if the user navigates to the menu using the arrow keys instead of using a mouse. Add the Select event handler to the mnuEdit menu.

The code to add is as follows:

```
private void mnuEdit_Select(object sender, System.EventArgs e) {
// disable the Cut and Copy menu items if no text is selected
if (rtfMain.SelectionLength==0) {
mnuEditCut.Enabled=false;
mnuEditCopy.Enabled=false;
} else {
mnuEditCut.Enabled=true;
mnuEditCopy.Enabled=true;
}

// now check to see if we can paste anything
IDataObject iData=Clipboard.GetDataObject();
if (iData.GetDataPresent(DataFormats.Text)==true)
mnuEditPaste.Enabled=true;
else
mnuEditPaste.Enabled=false;
}
```

The first part is straightforward. The SelectionLength property of the RichTextBox control is checked; if it is more than zero, some text has been selected and we can cut or copy it; the menu entries are enabled. If the length is zero, there's nothing to cut or paste and we can disable the menu entries.

The second part is a little more complex. It uses an IDataObject, which is returned from the Clipboard by a GetDataObject() method call, to determine if there is any text on the clipboard. Since there may be data of several different types on the clipboard, we need to see if there is any text there; this is done using the GetDataPresent() method of the IDataObject, passing a parameter of DataFormats.Text to specify that we want to know if there is any text present. If there is, the Paste command is enabled.

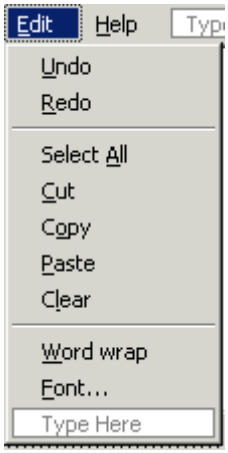
3. Undo & Redo functionality

An Undo and Redo commands are very useful. Add these two commands to the top of the Edit menu and add a separator after them.

While you're about it, add some more commands as well:

add a Select All command just above the Cut menu item
add a Clear command just below the Paste menu item
add a Font... command just below the Word wrap item

The Edit menu should now look like this:



Set shortcuts to Undo as Ctrl-Z, and Select All as Ctrl-A. Rename the menu items as follows:

Undo	mnuEditUndo
Redo	mnuEditRedo
Select All	mnuEditSelectAll
Clear	mnuEditClear
Font	mnuEditFont

Finally, add Click event handlers for these five new menu items.

In the next instalment, we will add the remainder of the code to the Edit menu items.

C# Tutorial #1: A complete Windows Forms application (part 5)

Having added some additional commands to the Edit menu, we now need to add code for these.

Steps to take

1. Undo & Redo

The code for these commands is simple:

```
private void mnuEditUndo_Click(object sender, System.EventArgs e) {
rtfMain.Undo();
}
private void mnuEditRedo_Click(object sender, System.EventArgs e) {
rtfMain.Redo();
}
```

However, the Undo and Redo commands may not be available, depending on whether there are any actions which can be undone or redone. We can check this, and enable or disable the commands appropriately, in the Select event handler of the Edit menu, as we did before. Add the following code fragment to the mnuEdit_Select function:

```
// enable/disable the Undo/Redo commands
if (rtfMain.CanUndo)
mnuEditUndo.Enabled=true;
else
mnuEditUndo.Enabled=false;
if (rtfMain.CanRedo)
mnuEditRedo.Enabled=true;
else
mnuEditRedo.Enabled=false;
```

The CanUndo and CanRedo properties of the text box are true if the last action (if any) can be undone or redone.

2. Select All

This just uses the SelectAll method of the RichTextBox:

```
private void mnuEditSelectAll_Click(object sender, System.EventArgs e) {
rtfMain.SelectAll();
```

```
}
```

3. Clear

Likewise, there is a Clear method as well. However, we should check that the user really wants to do this, in case data is lost:

```
private void mnuEditClear_Click(object sender, System.EventArgs e) {
if(MessageBox.Show("Do you really want to delete all the text?",
"Confirm text deletion", MessageBoxButtons.OKCancel,
MessageBoxIcon.Question)==DialogResult.OK)
rtfMain.Clear();
}
```

4. Font

To change the font, we need to add a FontDialog. This is exactly the same as adding an OpenFile or SaveFile dialog, so add it now and rename in dlgFont. In the Click event handler of the menu item, we need the following code:

```
private void mnuEditFont_Click(object sender, System.EventArgs e) {
// set the font dialog to the one we currently use
dlgFont.Font=rtfMain.Font;
if (dlgFont.ShowDialog()==DialogResult.OK)
rtfMain.Font=dlgFont.Font;
}
```

This just sets the font in the dialog to the one we already use, then waits for the user to click OK and sets the font in the text box.

5. Closing the form

This is nothing to do with the Edit menu, but it's got to appear somewhere! Although we take care of the user exiting the application through the File | Exit menu item, the user could close the window - which would, in this case, also exit the program. So we have to handle that, too.

The form can also trigger events, one of which is the Closing event. As you might imagine, this is triggered when the for is about to close, but before it has done so. The nice thing is that we can cancel closing the form inside this event. Add a Closing event handler to the form, and add the following code:

```
private void frmMain_Closing(object sender, System.ComponentModel.CancelEventArgs e) {
// closing this form will exit the application so we should
// query user if the text in the box has changed
if (rtfMain.Modified==true ) {
if (MessageBox.Show("Text has changed - really exit?", "Confirm exit",
MessageBoxButtons.YesNo, MessageBoxIcon.Question)==DialogResult.No)
e.Cancel=true;
}
}
```

This is very similar to the Exit Click event handler, but this time, if the user clicks Cancel in the message box, we set the Cancel property of the CancelEventArgs object to true, which aborts closing the form.

In the final instalment, we will add any remaining functionality, including the About box.

C# Tutorial #1: A complete Windows Forms application (part 6)

We are now in a position to finalise the program. Firstly, we need to add an About box.

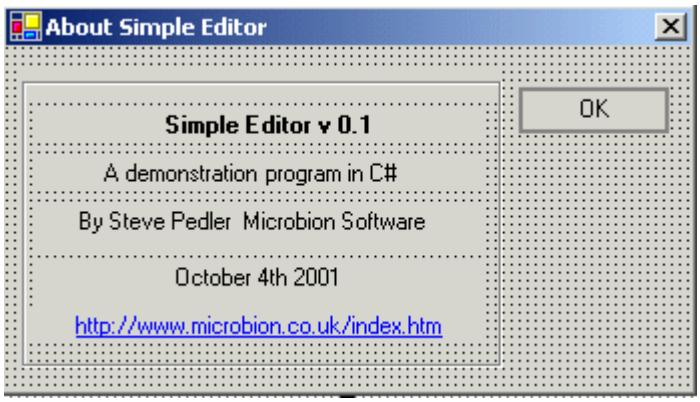
Steps to take

1. Add an About box

From the Project menu, select Add Windows Form... and add a new form. Name it frmAbout. Set its Text property to 'About Simple Editor' and its FormBorderStyle to FixedDialog. I set its StartPosition to CenterParent, the MinimizeBox and MaximizeBox properties to false, and the ShowInTaskbar property to false.

In the form, add a frame (what .NET calls a 'Group box' and delete its caption (Text property). Then add 4 labels inside the frame, plus a LinkLabel. This new control is in the toolbox under the Label control and apart from acting as a hyperlink, is otherwise a standard label control. Finally, add a button to the form and call it cmdAction (some reversion to VB there: in VB6, these are called Command Buttons!) with the Text set to 'OK'.

Your form should look like this (you'll need to alter the Text properties of the labels to what you see here, and set their TextAlign property to TopCenter):



You can also play about with the button. Try the FlatStyle property and the various values for it (I like the Popup setting). Also, you can make this button respond to the Enter key by setting the form's AcceptButton property.

The only other thing to mention is the LinkLabel control. You can display both ordinary text and a hyperlink in this control. If you just start typing into the Text property, it will all be treated as a hyperlink, which in this case is what we want. If that doesn't happen, go to the LinkArea property, click the button in the right hand column, and select the part of the text you want to be a hyperlink.

2. Add code to the About box

There are only two things we want this box to do. One is to close down when the user clicks the button; the other is to go to the URL in the LinkLabel when that control is clicked. Add a Click event handler to the button, and a LinkClicked event handler to the LinkLabel (not a Click event - this is important).

The button click code is very simple:

```
private void cmdAction_Click(object sender, System.EventArgs e) {
// just close the dialog box
this.Close();
}
```

For the LinkLabel we need to indicate that the link has been visited, and then navigate to the URL. The code is:

```
private void linkLabel1_LinkClicked(object sender,
System.Windows.Forms.LinkLabelLinkClickedEventArgs e) {
linkLabel1.LinkVisited=true;
System.Diagnostics.Process.Start("http://www.microbion.co.uk/index.htm");
}
```

Note the signature of the handler, which is different from the usual Click event handler, and the method used to run the browser and navigate to the URL.

3. Show the About box from the main window

We can show our new dialog by calling its ShowDialog() method. However, this is not a static method - it is a public instance method. This means we have to create an instance of the form before calling the method.

Add a Click event handler to the About... menu item in the Help menu of frmMain. In the event handler function, insert this code:

```
private void mnuHelpAbout_Click(object sender, System.EventArgs e) {
frmAbout fAb;

// show the About box
fAb = new frmAbout();
fAb.ShowDialog();
}
```

This just creates an instance of the form and displays it.

We now have a fully-functioning application. Of course, it isn't that remarkable, but it does work. There are several things we might want to add: a help system, a toolbar, a status bar, a most-recently-used (MRU) file list, and so on. And of course, it's missing a significant area of functionality: there's no way to print anything.

But for the moment, Simple Editor rests here. Over the next few weeks I will be adding some of the above functions to the program, as and when I find out how it's done! Until then, if you want to download the code to reconstruct the program without typing it in, use the link below. You'll need VS.NET beta 2 to compile it.

Download code for Simple Editor (42K)

Note: when you unzip this file, remember to turn on the option which recreates the subfolders!