

## **Toolbars and Status Bars**

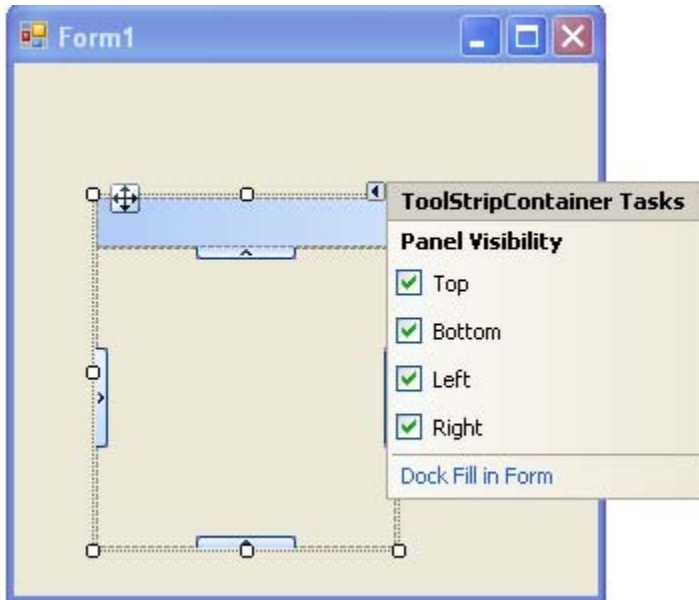
Toolbars started out as a graphical alternative to menus. A toolbar was (and is) a row of buttons, usually with graphical icons on the buttons to represent common tasks in the program. These buttons normally duplicate certain commands that can be issued through the menu of the program, but the idea is to put the more common commands in easy one-click reach. In the years since that time, menus have gotten more graphical and more flexible in their appearance and placement, and in Visual Studio 2005, they are treated almost the same from the programmer's point of view. That's a change, since until 2005, it was assumed that every program would have a menu but a toolbar was optional, so the "client area" of the window didn't include the menu, but it did include the toolbar.

A status bar is a window, just one line of text high, across the bottom of a window, used for informative messages. Status bars existed before "tool tips" were invented, and they served (and still can serve) a similar purpose; but in my opinion they should really be replaced by tool tips. In my experience people often don't even see the information at the bottom of the window in a status bar. That, of course, is why tool tips were invented—to make the extra information pop up where people are already looking. Nevertheless status bars sometimes may be useful, and they're easy to program.

## **ToolStripContainer**

Menus and toolbars both occupy part of your form, below the title bar. That creates a certain annoyance in that coordinates in your form do include the area covered by the menu and title bar. The remedy for this is to create a separate window that just fills the area not used by the menu and title bar. The FCL provides an easy way of doing this: the *ToolStripContainer*.

When you're starting out, instead of first adding a *MenuStrip* to your form, add a *ToolStripContainer*. You'll see something like this:



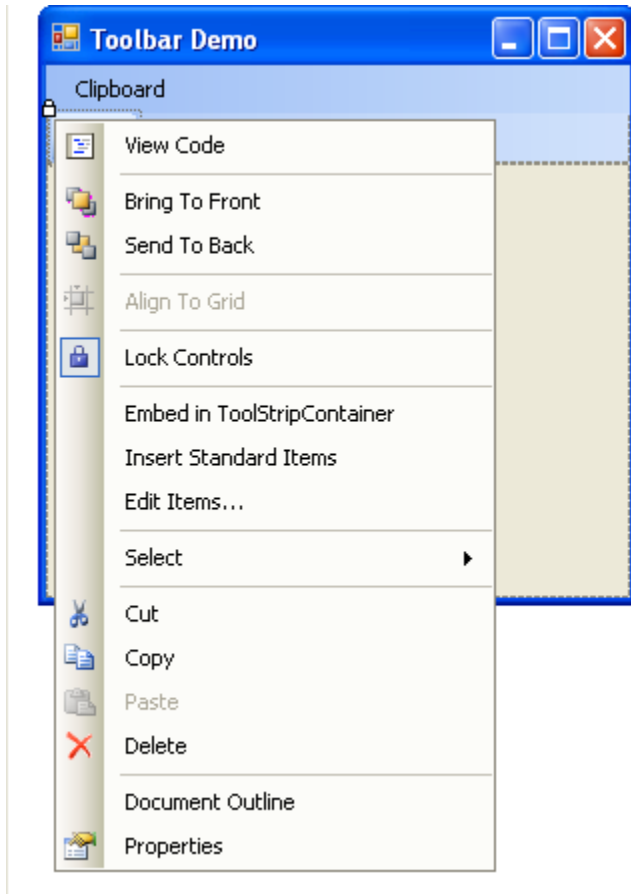
Remove the checks from all but “top”, and then drag your *MenuStrip* onto the container’s top panel. Now right-click in what appears to be the main window, and choose *Properties*. Observe that you’re not in *Form1* but in a *Panel* object. What has happened is that the *ToolStripContainer* window contains five smaller windows, called “panels”. The top panel contains your menu, and the center one, with the name *ContentPanel*, is available for you to draw in, as you formerly drew in *Form1*. Now, you won’t have to adjust coordinates for the height of the menu.

## The ClipboardDemo program

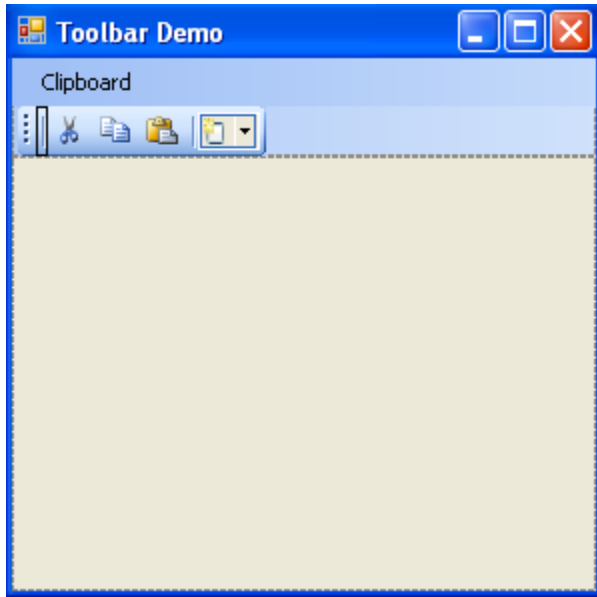
Add a *Clipboard* menu with three menu items, *Cut*, *Copy*, and *Paste*. We’ll add a toolstrip with three buttons to duplicate those menu commands.

## Adding a ToolStrip

Add a *ToolStrip* item into that top panel. (You can click that little tab below the menu to make it open in the design windows enough to be able to drag-and-drop a toolstrip there.) Right-click your toolstrip and choose *Insert Standard Items*:



Then delete all but the three items shown here:



Observe that Visual Studio was kind and smart enough to enter something in the *ToolTipText* property of those buttons. You could change it, and if this were a custom button, you would want to enter it. This should usually be the same text as the corresponding menu item. Observe also the *AutoToolTip* property, which controls the automatic appearance of tool tips when the user mouse over a button on the toolbar. It's true by default, so you don't need to change it.

## **Adding an ImageList**

If, however, you wanted some custom items, you would do something more complex. You would add an *ImageList*. Here's how (but we don't do that in this demo):

Drag and drop an *ImageList* to your form. We want the *ImageList* to contain a list of bitmaps to be used for the buttons. On the property sheet of the image list, you will see the *Images* property. Press the button labeled "...". Then the *Add* button. Then you can browse for image files. Browse to where your custom image

files are, and add them one by one. In Visual Studio 2003, you had to employ this method even to add the standard items, but now this is easier, as shown above. Still, a serious commercial program will need its own custom toolbar, so in your future work you may need to know about *ImageList*.

## Examining the source code

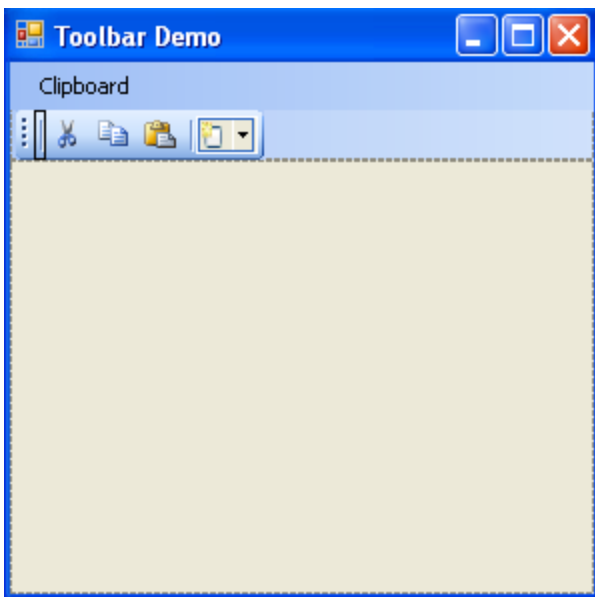
Examine the code in *InitializeComponent* that the form editor wrote for you. Find the code that causes the bitmaps for your buttons to be incorporated in your executable file, so the .bmp files won't need to be there at run time. It looks something like this:

```
// copyToolStripButton
this.copyToolStripButton.DisplayStyle =
    System.Windows.Forms.ToolStripItemDisplayStyle.Image;
this.copyToolStripButton.Image =
    ((System.Drawing.Image)(resources.GetObject(
        "copyToolStripButton.Image")));
this.copyToolStripButton.ImageTransparentColor =
    System.Drawing.Color.Magenta;
this.copyToolStripButton.Name = "copyToolStripButton";
this.copyToolStripButton.Size = new System.Drawing.Size(23,
    22);
this.copyToolStripButton.Text = "&Copy";
```

In the *Tag* property of each button, put the name of the corresponding menu item. This is how we will tell the program what menu item corresponds to what toolbar button. Specifically, these tags should be *cutToolStripMenuItem*, *copyToolStripMenuItem*, and *pasteToolStripMenuItem*, unless you changed what Visual Studio wrote for you. Actually, the *Tag* property can be anything at all—its type is just *Object*. Just entering a *Tag* property doesn't make anything at all happen, by itself, but we are going to use those *Tag* properties in our code.

Now check the source code. You will find that the *Tag* properties have *not* been set as you intended—they have been set to strings, for example “cutToolStripMenuItem” instead of the menu item itself. Both qualify as objects, and the form editor didn’t know which you meant. Fine, delete the quote marks in your source code, and it will still compile fine, and the form editor seems to retain this code rather than writing over it.

Now you can build and run the program. Both menu and toolbar should be visible:



Also, the tooltips should be working when you mouse over the toolbar.

## **Handlers for the Toolbar Buttons**

We don’t want to give each individual button on the toolbar its own handler. On the property sheet of each toolbar button, select

“events” (the lightning bolt) and look at the *Click* event. Select the following handler (which should already be there, but contains no code) and add the following two lines of code:

```
private void toolStripContainer1_TopToolStripPanel_Click(object sender, EventArgs e)
{
    ToolStripMenuItem mi =
        (ToolStripMenuItem)((ToolStripButton)sender).Tag;
    mi.PerformClick();
}
```

These two lines, together with our assignment of the *Tag* properties to the toolbar buttons, will ensure that the buttons duplicate the corresponding menu items. Of course, since we didn't put in any code for the menu items yet, we can't test that.

## Finishing the program

Add a *TextBox* to your form. Set its properties as follows: *Multiline* should be true; *ScrollBars* should be *Both*; *AcceptsTab* should be true. *Dock* should be *Fill* (you set that by clicking the middle box). The *Text* property should be blank. Check that you can build and run.

Add handlers for your menu items, and write code for the *Cut*, *Paste*, and *Copy* items as follows:

```
private void CutItem_Click(object sender, System.EventArgs e)
{
    textBox1.Cut();
}

private void PasteItem_Click(object sender, System.EventArgs e)
{
    textBox1.Paste();
}

private void CopyItem_Click(object sender, System.EventArgs e)
{
    textBox1.Copy();
}
```

Now, at last you can test something: both your menu items and your toolbar buttons for *Copy*, *Cut*, and *Paste* should be working. Type some text into the textbox and cut and paste it to Notepad to see if it works. Change it, and paste the changed text back from Notepad to your program.

## Writing your own code

In Chapter 20 of Petzold, a similar program is created, without using the design editor. You should, after class, compare the two approaches. On page 987 there is a discussion of how to disable the Cut and Copy buttons when no text is selected, and to disable the Paste button when there is nothing on the clipboard.

## Status Bars

Now, let's put a status bar on this program. Select your *toolStripContainer* object in the form designer (by right-clicking in the main part of the form ). Make the bottom panel's *Visible* property *True* and click to make it visible in the form editor. Then you can drag a *Status Strip* onto that panel.

On the property sheet of the status strip, go to the *Items* property and add one item. Give that panel the text *menu help*. Set the *AutoSize* property to *False* and the *Size* property to give it a width of 250. Run the program: you should see the status bar and the text *menu help*.

The status bar is often used to provide a fuller explanation of the selected menu item. To do that we need to use the *Select* event that a menu item can generate. Pick one of your menu items, bring up its property sheet, click the lightning bolt to look at events, and double-click the *MouseHover* event to add a handler. I chose the

*Cut* item, so I got *CutToolStripItem\_MouseHover* as my handler. Then select that *same* handler for the other two menu items.

14. Here's the code for *MenuItemSelect*: (don't cut and paste because there are line breaks inserted in quoted strings for readability).

```
private void cutToolStripMenuItem_MouseHover(
    object sender, EventArgs e)
{ ToolStripMenuItem mi = (ToolStripMenuItem)sender;
  switch (mi.Text)
  { case "Cut":
    toolStripStatusLabel1.Text = "Copy selected
text to the clipboard and delete it from the
textbox.";
    break;
    case "Copy":
    toolStripStatusLabel1.Text = "Copy selected
text to the clipboard.";
    break;
    case "Paste":
    toolStripStatusLabel1.Text = "Paste selected
text from the clipboard at the current
cursor location.";
    break;
    default:
    toolStripStatusLabel1.Text = "oops!";
    break;
  }
}
```

After this, test the program: When the menu is in use, the status line text should change.

One problem with this code is that the status line text remains in place after the menu is closed, instead of resetting to the initial text. For the solution to that problem, see Petzold, p. 970.

Another problem with this code is that it will be really hard to maintain when the menu item text is edited, because then the menu item text has to be duplicated in two places, and you'll never be able to keep them in sync. In MathXpert, I solved this problem by using the position of the menu item in an array, but it was still difficult to keep them in sync. Petzold suggests another solution on page 971.

*Further explorations:* Look at the properties of your toolbar buttons, and add some text to the *Text* property of the buttons (it can be the same as the menu text). Run the program and see how this looks. Then change the *TextAlign* property of the toolbar to *Right* and see how that looks. As usual in Visual Studio, there are many, many properties in the classes we've studied here; at least look at their names.