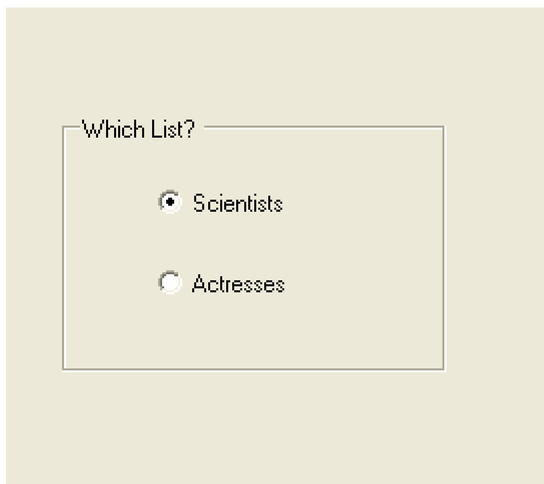
Radio Buttons and List Boxes

The plan for this demo is to have a group of two radio buttons and a list box. Which radio button is selected will determine what is displayed in the list box. It will either be a list of scientists, or a list of actresses and actors. Rather than using a modal dialog, this time we will add the controls directly on *Form1*.

Radio Buttons

Radio buttons are used to let the user select one of a group of choices. Usually you create a "group box" and put a group of radio buttons together inside the group box. Here's an example:



Note that each button has a label, specified by the *Text* property of the button. The *Text* property of the group box is also visible as its title.

Radio buttons are used to let (indeed force) the user to select exactly one item from a short list of alternatives, usually just two or three. If there are more items, consider using a list box instead. Exactly one button in the group should be selected. Sometimes, you might want to initialize a group of radio buttons with *no* button selected, with the

intention of forcing a user to make a choice, but usually it's better to have a default choice pre-selected.

In both the Windows API and MFC, there were some mistakes you could easily make with radio buttons, but now with .NET, programming radio buttons has been made completely idiot-proof. All you have to do is drag a group box onto your form, and drag the radio buttons into it, and set the *Checked* property of one of the buttons to be true, so it will be checked at startup.

Then you will observe that you can run your program and the following happen correctly:

- The buttons work automatically: only one at time is selected, and when you select a different one, the previous selection becomes unselected.
- The tab key moves you from other controls to the selected button within the group, and another tab key press moves you out of the group to the next control whose *TabStop* property is true.
- Within the group, the arrow keys (both horizontal and vertical) allow you to navigate between the buttons, and the arrow keys correctly cycle from the last button to the first.

Look at this automatically generated code in *InitializeComponent*:

```
this.groupBox1.Controls.Add(this.ActressesButton);
this.groupBox1.Controls.Add(this.ScientistButton);
```

These lines of code specify that these two buttons "belong" to this group box. This in turn enables the features listed above to work without further coding on your part. Try this experiment: in the form designer, drag one of the buttons outside the group box. Then look at the source code—yes, one of those lines of code is gone!

Then drag the button back into the group box—sure enough, the line of code appears again.

The code itself doesn't care whether the group box visually contains the buttons or not. You can manually remove one of those lines of code, and even though the button still appears in the group box, it is not logically "in" the group box any more. You could also have the button visually out of the group box, and if you manually inserted that line of code, it would be "logically in" the group box anyway. But if you let Visual Studio write the code, then "logically in" will correspond to "visually in".

The order in which the arrow keys visit the buttons is determined by the order of these *Add* lines, not by the visual layout in the form editor. Therefore, add the buttons to the form in the order you want the arrow keys to use. If you later change your design and insert another button, you should manually edit the code and move that *Add* line, to get the arrow keys to function correctly. This seems to be the only programming task about radio buttons that Visual Studio has not completely automated! It used to take a whole lecture to teach how to program radio buttons correctly, and many mistakes were made even after that.

In your code, if you want to find out whether a button was checked or not, for example when a modal dialog terminates, just examine that button's *Checked* property. In today's example, we need to change a member variable each time the selected radio button changes. Add a member variable as follows:

```
private String m_WhichList = "Scientists";
```

We want to change this value whenever the radio button selection changes. To do this, go to the property sheet of the radio button *ScientistsButton* and add a handler for the *CheckChanged* event:

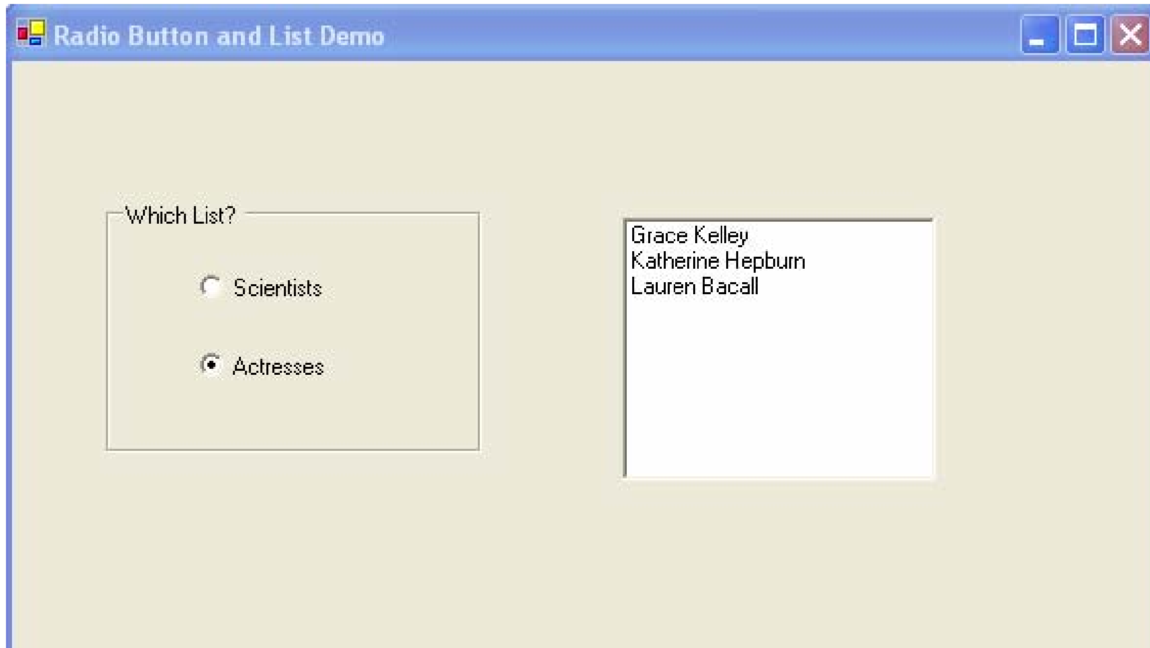
```
private void ScientistButton_CheckedChanged(object sender,
                                        System.EventArgs e)
{ m_WhichList = ((Control)sender).Text; }
```

By writing the code this way, instead of just as *m_Whichlist = "Scientists", we* can use this same handler for the other button in this group. Thus for the second button, you can just go to the *CheckChanged* event in the list of events, and choose this existing handler, instead of adding a new one. The cast to *Control* is necessary since *sender* is only of type *Object*; a cast to *Button* produces a run-time error, but a cast to *Control* works fine. Of course, you can't verify that *m_WhichList* is really changing correctly until we add some more code.

List Boxes

Now drag a list box from the Toolbox onto your dialog, and enlarge it a bit by dragging the corner. Initialize its contents using the *Items* property on its property sheet, as shown in the following screen shot:

Our aim is to make the contents of the list box change when the radio button selection changes.   Here is a screen shot showing the desired result:



This is accomplished by the following changes to the *CheckChanged* handler. (Aren't you glad we went to the trouble to ensure that both radio buttons had the same handler?)

```
private void ScientistButton_CheckedChanged(object sender,
                  System.EventArgs e)
{ m_WhichList = ((Control)sender).Text;
  listBox1.Items.Clear();
  if(m_WhichList == "Actresses")
      this.listBox1.Items.AddRange(new object[]
          {
           "Grace Kelley",
           "Katherine Hepburn",
           "Lauren Bacall"
          });

  else
    this.listBox1.Items.AddRange(new object[]
          {
            "Albert Einstein",
            "Madame Curie",
```

```
                "Grace Hopper",
                "Sonya Kovalevksy",
                "George Boole"});
            }
}
```

The *Items* member of the *ListBox* class is of type *ListBoxObjectCollection*. That class has a *Clear* method that enables you to remove all the items in one function call. The main data member of that class is an array of objects. Usually those "objects" are strings, but they also could be images, or anything else you like.

Put the data in member variables

We next want to give the user the ability to add another actress or scientist to the currently selected list. We will have to modify the handler above first, for the following reason. Let's say we add *Stephen Hawking* to the scientists list, and somehow we get him to show up in the list. Then we select *Actresses*, and the *Clear* call above erases all evidence of *Stephen Hawking*, and so when we select *Scientists* again, *Stephen Hawking* does not appear.

Evidently we need to maintain the two lists, of scientists and actors, as member data in our *Form1* class. That would have been a good idea in the first place, since those lists clearly constitute part of the program's data. Just cut and paste the *new object[]* code to the form's constructor. Now the handler looks like this:

```
private void ScientistButton_CheckedChanged(object sender,
                    System.EventArgs e)
{ m_WhichList = ((Control)sender).Text;
  this.listBox1.Items.Clear();
  if(m_WhichList == "Actresses")
      this.listBox1.Items.AddRange(m_Actresses);
  else
      this.listBox1.Items.AddRange(m_Scientists);
}
```

and the program functions just as before.

Warning: There is automatically generated code in *InitializeComponent* to initialize the strings in the list box. You may be tempted to replace it with

```
listBox1.Items.AddRange(m_Scientists);
```

You can do that, but this new line of code should go in the form's constructor, after the call to *InitializeComponent*. If you add new code by hand in *InitializeComponent*, it may be deleted the next time Visual Studio's form editor rewrites that code. Since that could be some time after you added the code, such a bug can be very confusing. Do not edit code in the "code regions" marked off as automatically generated.

The ArrayList Class

There is another improvement we need to make before we are ready to add strings. At present, *m_Scientists* and *m_Actresses* are just arrays of *Object*. Therefore they have a fixed length. Now, we could deal with that problem by making them "long enough", say 50 or so, and having a member variable to count how many of those places are actually used. But this problem comes up often, and it has been solved by the class *ArrayList* in C#. This is as good a time as any to learn *ArrayList*.

An *ArrayList* object functions more or less like an array, except that it has an *Add* method that permits you to add a new element. It has a default "capacity"; whenever you add too many elements for the capacity, then the capacity is automatically doubled. Without any effort on the programmer's part, new space is allocated and if necessary the old contents of the array are copied to the new space. (Whether this works properly when the array contains references to its own elements, I have not tested—probably not, but that situation

isn't common.) One constructor of an *ArrayList* takes an array as argument, so we can revise our form constructor as follows:

```
private ArrayList m_Scientists=new ArrayList( new object[]
{
"Albert Einstein",
"Madame Curie",
"Grace Hopper",
"Sonya Kovalevksy",
"George Boole"}
);
```

In our click handler and in the form constructor where we initialize the contents of *listBox1*, we could use a for-loop up to *m_Scientists.length*, but the *foreach* construct of C# is also handy:

```
foreach(Object x in m_Scientists)
   listBox1.Items.Add(x);
```

The revised click handler looks like this:

```
private void ScientistButton_CheckedChanged(object sender,
System.EventArgs e)
{ m_WhichList = ((Control)sender).Text;
  this.listBox1.Items.Clear();
  if(m_WhichList == "Actresses")
     { foreach(Object x in m_Actresses)
           this.listBox1.Items.Add(x);
     }
  else
     { foreach(Object x in m_Scientists)
           this.listBox1.Items.Add(x);
     }
 }
```

After these changes, the program should run as before; but now we are set up to add new strings.

Adding new strings to the list box

Drag a new TextBox and button onto the form, so the program looks like this when it runs:

Add a handler for the *Click* event for the *Add New Item* button, and put the following code there:

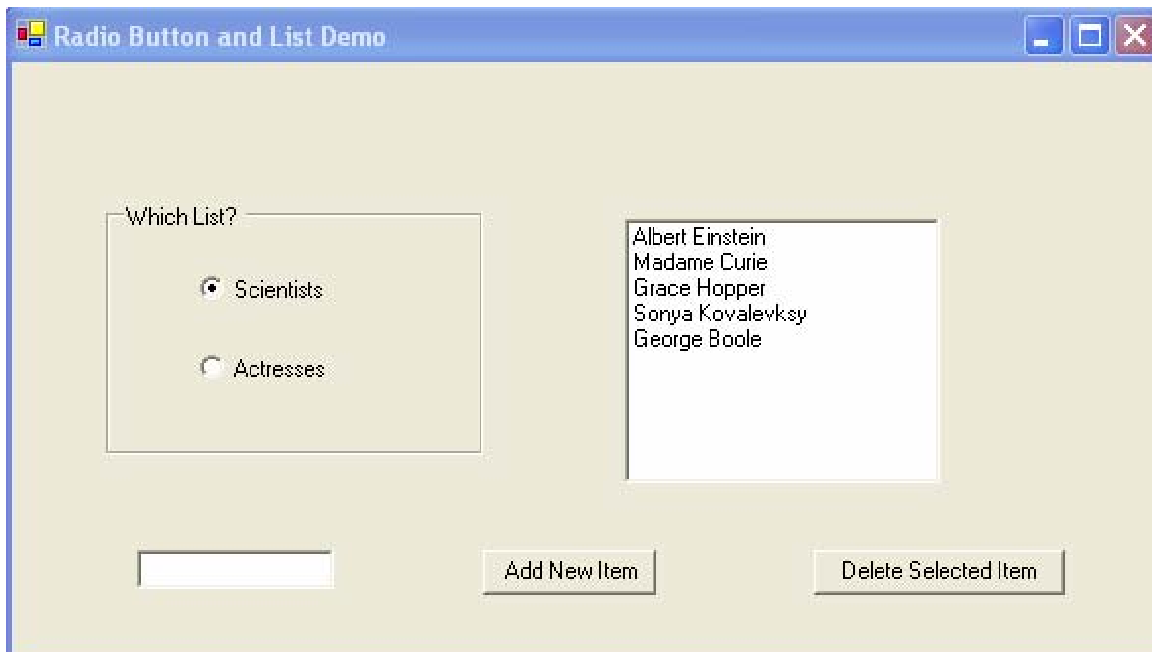
```
private void AddButton_Click(object sender,
System.EventArgs e)
      {
         if(m_WhichList == "Scientists")
            m_Scientists.Add(textBox1.Text.Trim());
         else
            m_Actresses.Add(textBox1.Text.Trim());
         listBox1.Items.Add(textBox1.Text.Trim());
      }
```

This code adds the new item to the correct list in the form's member data, *and* it adds the new item to the list box, so it will be immediately visible. The method *Trim* in the *String* class removes initial and final white space. You can see in the following screen shot that a string can be successfully added.

Now for the final test: when we select *Actresses*, and then select *Scientists* again, will *Stephen Kleene* show up in the list box? Try it and see!

What item is selected?

As a final exercise in list boxes, let's add a button to delete the selected item:



Here's the click handler for the new button:

```
private void DeleteButton_Click(object sender,
                                System.EventArgs e)
{ int k = listBox1.SelectedIndex;
  listBox1.Items.RemoveAt(k);
  if(m_WhichList == "Scientists")
      m_Scientists.RemoveAt(k);
  else
      m_Actresses.RemoveAt(k);
}
```

That makes it work. Test it by deleting an item, then switching lists and then switching back.