Methods

One of the basic principles of structured programming is that individual behaviours should be implemented as individual program modules or **methods**. Complex code sequences should be encapsulated in methods and called (invoked) elsewhere in the program as a single command. This **modularity** results in code that is robust, readable and easy to maintain.

A **method** is just a group of statements placed together under a single name. A method encapsulates a particular behaviour, and can be invoked at any point in the program.

```
private void button1_Click(object sender, EventArgs e)

public int SumAllValues()

private double halveThisValue(int value)
```

Access Modifiers

The modifiers can be:

`private`	The method is limited to the containing `class.`
`public`	No restrictions within the namespace
`protected`	Limited to the containing `class`, or classes derived from the containing `class`.
`internal`	Limited to current project.
`protected internal`	Limited to current project or classes derived from `class`.

Return type

The return type identifies the *type of value that is returned* when a method is completed. Any of the pre-defined or valid user-defined types can be used. If a method does not return a value, the keyword `void` is placed at the return type location.

Method name

The method name should start with a verb and describe what the method does. Starts with uppercase letter for public methods and starts with lower case for private methods.

Input Parameters

Methods can accept **parameters**, values that are passed as input to the method. The parameter is listed in the parentheses (round brackets) of the method heading by its data type *and* then its local name. Note, an **argument** is the value of a parameter.

Method body

Code statements written within curly brackets.

Interface Controls: Responding to User Input

- Review flow of control structures (*if … else* statements, *case* statements, *for* loops, and *while* loops) that determine the order in which program instructions are executed.
- Use Visual Studio interface controls (*ListBoxes, RadioButtons* and *CheckBoxes)* to implement complex branching behaviours in response to user input.
- Create the *Pizza Parlour* application.

1. Making Decisions

If … else Selection Statement

```
if (examScore > 95)
{
        MessageBox.Show("You scored an A+ in Programming 2!");
}
else
{
        MessageBox.Show("You didn't score an A+ in programming 2.");
}
```

Case Statement

If you're testing for more than two possible values, you should use a *case* statement.

```
switch (num)
{
        case 1:
                <do something>;
                break;
        case 2:
                <do something>;
                break;
        case 3:
                <do something>;
                break;
        …..
        case n:
                <do something>;
                break;
        default
                <do something>;
                break;
}
```

The "do somethings" can be of unlimited complexity; the usual rules for use of curly brackets apply.
Put the most frequently occurring value at the top, as the case statement is executed in the listed order.
The default case is executed when none of the preceding statements are true (in this case, when A doesn't equal any of value1, value2,…, value*n*).

---

2. Repeating Instructions

You will recall that the difference between *for* loops and *while* loops is that *for* loops are determinate (they run for a specified number of cycles), and *while* loops are indeterminate (they run until some condition or conditions are met, and this takes different numbers of cycles for different executions of the program). *While* loops often involve a Boolean flag that is set to *false* before the loop begins, and set to *true* inside the loop when the termination conditions are satisfied. The loop runs until the flag is *true*.

For **loop:**

```
for (int i = <initial value>;  i <= <final value> ; i++)
{
      <do something>
}
```

Always use a *for* loop in preference to a *while* loop.
The count variable needs to be an ordinal type and is declared inside the loop. It is initialised on the first pass of the loop and is incremented at each pass.

While **loop:**

```
while (some condition holds)
{
      <do something>
}
```

DoWhile **loop:**

```
do
{
      <the something>
}
while (some conditon holds)
```

The condition is evaluated at either the beginning or end of the loop.
Any condition variables must be declared outside the loop and need to be initialised and incremented manually.
Beware of infinite loops. If the condition may never be met, add an OR statement to allow escape.

The TextBox and ListBox Controls

In the first class, we used a single-lined *TextBox* to read and write text. A *TextBox* can be also be multi-lined, selected from the drop down box at the top right corner of the *TextBox* control.

Each control has its own set of properties, events and methods, events. We used the *TextBox* control's **Properties** (e.g. *Width*) and control **Events** (e.g. *Click*). There is a third part to the interface of a control: its **Methods**. *Methods* are commands that the control understands. You use these commands when writing event handlers for the control. For example, *TextBox* controls have a Method called *Clear()*. This method tells a *TextBox* to erase all the text it contains. To invoke a method, you use the same dot notation that you used for properties. Thus, the statement *textBox1.Clear()*; will erase all the text in a *TextBox* named *textBox1*. To add text to a *TextBox*, use the *AppendText(s)* method, where the parameter *s* is the string that is to be added to the *TextBox*.

A *ListBox* is another control for reading and writing text. It is multi-lined by definition.

One of the things that makes C# so powerful is that a control's properties can themselves be complex objects that have their own methods, events and sub-properties. For example, a *ListBox* contains a property called *Items*, which stores the text contents of the *ListBox* (just as the Width property of a *Button* stores the *Width* of the *Button*). The Items property is itself a complex data object with methods and properties. For example, Items has a method called *Clear()* to clear all items in the *ListBox*. It also has a method called *Add()*, which adds a new line of text to *Items* (and thus to the *ListBox*). To call a submethod like *Add()*, just extend the dot notation in the logical way, adding the method name after the property name. For example, the following statement puts the string "*This is a new line of text*" after the current contents of the *ListBox* called *listBox1*:

```
listBox1.Items.Add("This is new line of text");
```

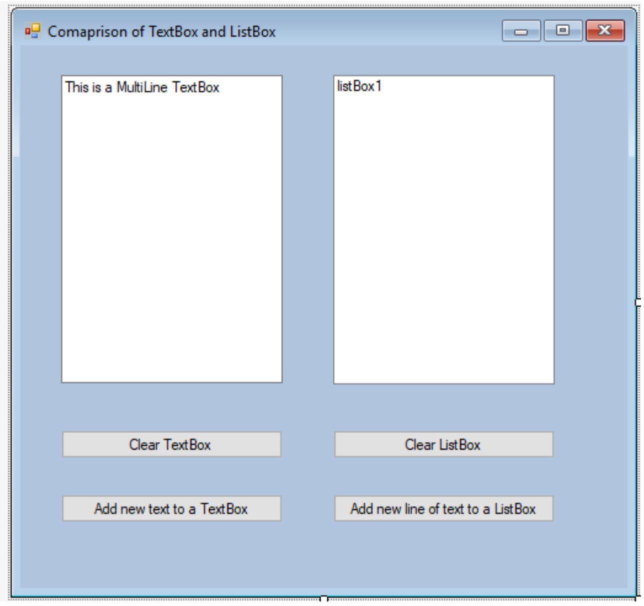
The *Items* property also has properties of its own. For example, *Items* has a property *Count*, which stores the number of lines of text the *Items* property contains (this is of course equal to the number of lines of text in the *ListBox*).

```
listBox1.Items.Count
```

Note that you do not need to know how the *Clear* method, the *Lines* property, the *Add* method or the *Count* property are implemented. You only need to know how to use them. For example, you must know that the *Add* method requires a string parameter, and that *Count* is an integer value. You need to know the interface, but not the implementation. This is called **Information Hiding** and is an important concept in Object Oriented programming.

Task 1: Comparing a TextBox and a ListBox

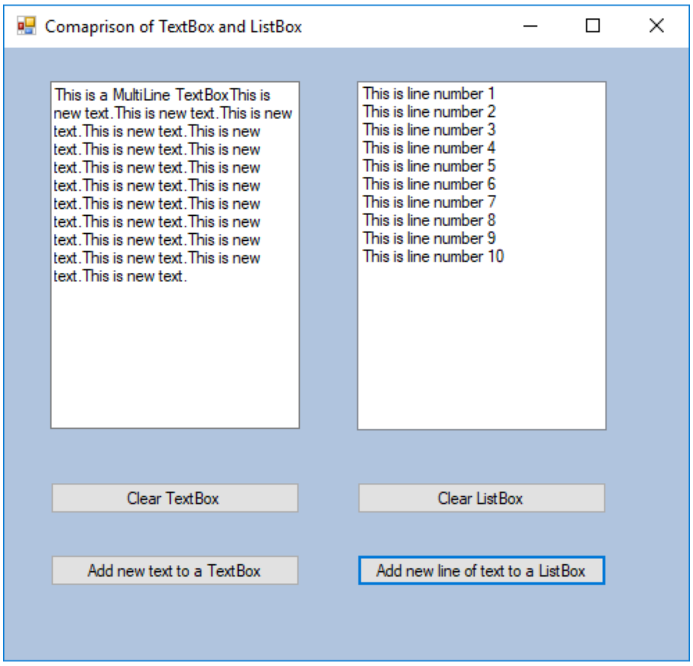
1. Start a new project.
2. Add a *TextBox*, *ListBox* and four *Buttons* to your Form:



3. Write the code, so that when the user clicks the "Clear the TextBox" *Button*, the text is cleared.
4. Write the code, so that when the user clicks the "Add a line to a TextBox" *Button*, it adds a new line to the contents of the *TextBox*. The syntax for adding a line to a *TextBox* is:

```
textBox1.AppendText("This is new text");
```

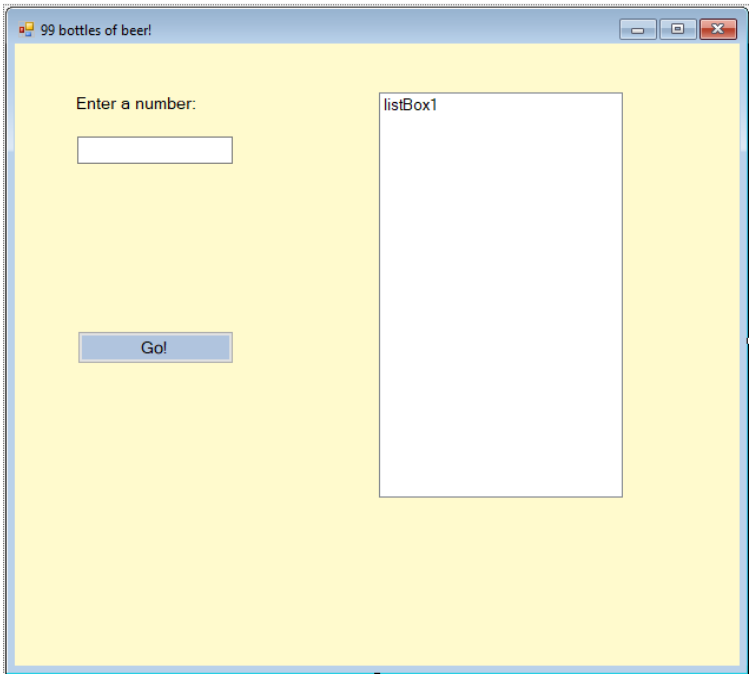
5. Now add a new line of text to the *ListBox*. Recall that the *Count* property (*listBox1.Items.Count*) equals the number of lines of text in the *ListBox*. Using the *Count* property, modify your code so that on each click of the button, a new line is added to your *ListBox* that says "This is line number *n*", where *n* equals the line number. The output should look like this:



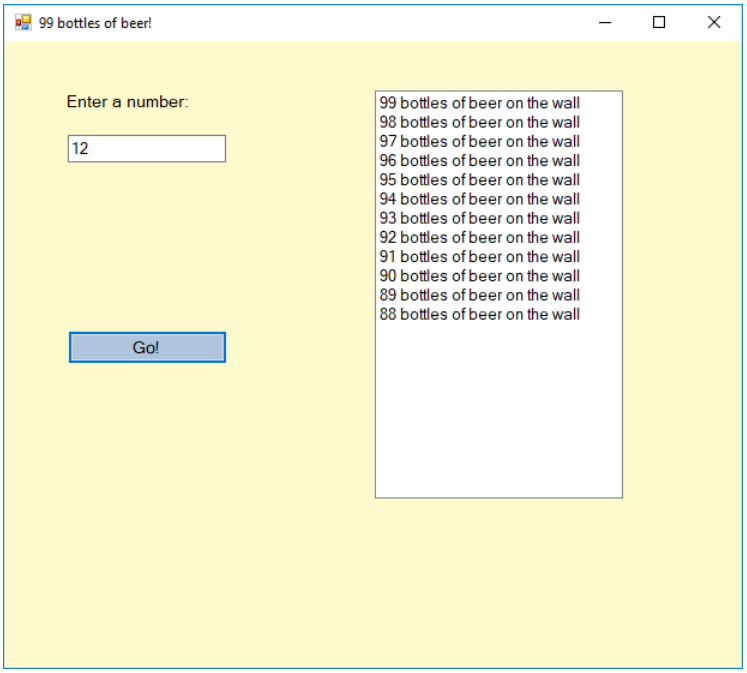
Note: you will probably find that the first line your program writes is 'This is line number 0". Why does this happen? How can you fix it?

Task 2: 99 Bottles of Beer on the Wall with a For Loop

1. Create a new project. Place a *Label*, *TextBox*, *ListBox* and a *Button* on your *Form.*



2. Your program is going to display the lyrics to the classic song “99 Bottles of Beer on the Wall” in the *ListBox*. The user enters a number into the *TextBox*, then clicks the button. Display the first *n* lines of the song, where *n* is the number the user entered into the *TextBox*.



3. This program requires very little code. The whole thing can be done with a single statement inside a *for* loop.

Interface Elements for Branching

C# provides two familiar user interface elements that can be used to collect user input to determine branching conditions. They are *RadioButtons* and *CheckBoxes*.

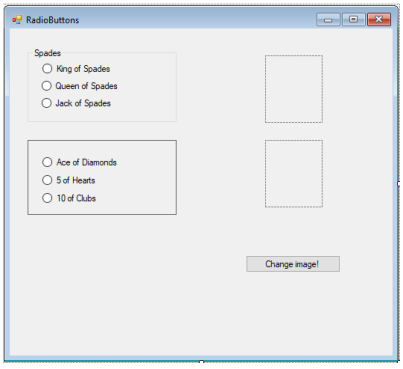
Radio Buttons

RadioButtons are Yes/No (or On/Off) controls that occur in mutually exclusive sets. That is, only one member of a set of radio buttons can be selected (on) at a time. Selecting any member of a set of radio buttons causes all the other members to be deselected.

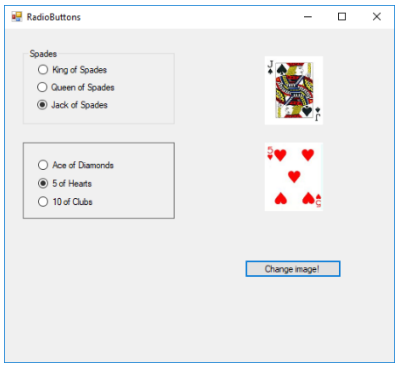
If you place a number of radio buttons directly on the *Form*, they are automatically members of a mutually exclusive set (i.e. only one of them can be selected at a time). If you want more than one set of radio buttons, you must place each set in a separate container. C# provides two types of containers. They can be used to group any C# controls, but are most frequently used to define a set of *RadioButtons*. They are a *GroupBox* and a *Panel*. You must manually place each button on the container, and must manually change the caption of each button.

Task 3: Using *RadioButtons* to Collect User Input

In this exercise, the user will choose a new card by selecting the appropriate *RadioButton*. The card image will be loaded when the *Button* is clicked.



Design time



Run time

1. Create a new project.
2. Place two *PictureBoxes* on your *Form.*
3. Add the image files in *CardsVersion1* to the *Resources* folder using the *Project/RadioButons Properties/Resources path.*
4. Create two sets of *RadioButtons*. Each set will control the choice of card image to be displayed. Make one set using a *GroupBox* and one using a *Panel*.
5. The user selects one *RadioButton* from each group, then clicks a button called “Change image!” (or equivalent). Each PictureBox changes to the card image indicated by the user’s selection.
6. To determine the status of Radio Buttons in different containers, you will need this statement (as well as many others, of course):

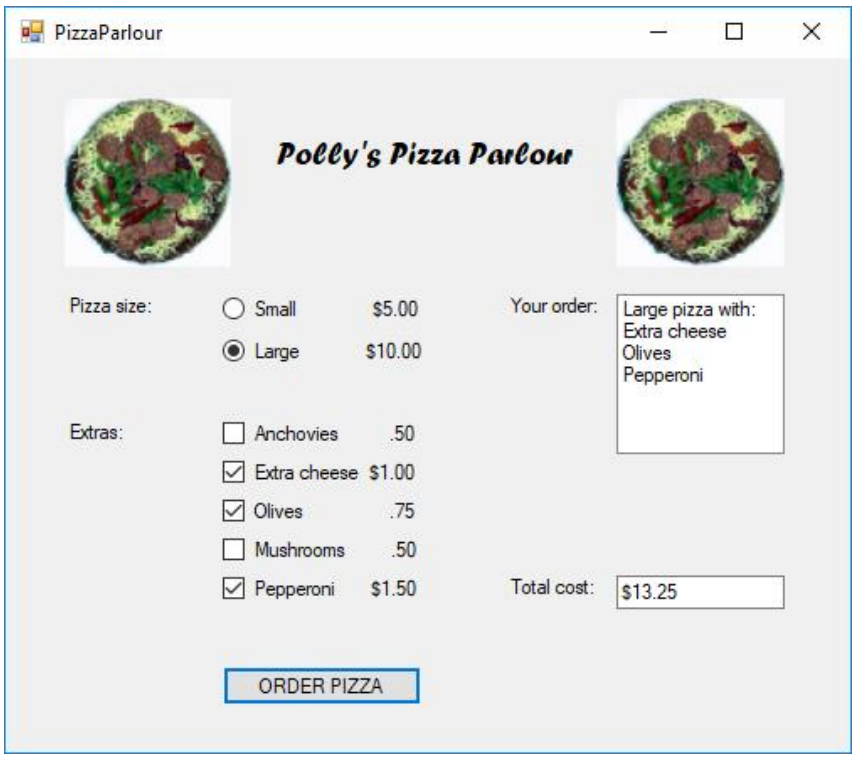
```
if (radioButton1.Checked)
{
        pictureBox1.Image = Properties.Resources.S13;
}
```

Check Boxes

CheckBoxes are also on/off controls for getting user input, but they are not mutually exclusive. That is, when you have multiple check boxes on a form, any number of them may be selected at the same time.
To determine if a *CheckBox* is selected, look at *checkBox1.Checked* (as for *RadioButtons*).

Task 4: Pizza Parlour Program ******* CHECKPOINT 2 *******

Write an application for use in a Pizza Parlour. The user enters his order, and the application displays the order and computes the price. Your solution could looks like this (decorations are optional):



Specifications:

1. Your program should allow at least two sizes of pizza, with different prices. Users must select a size. If a user tries to order without selecting a size, he should receive polite feedback asking him to please specify the size.
2. You should provide at least five different extra toppings, each with associated prices. Users can select any combination of extra toppings, or none at all.
3. You should correctly display the order in a *ListBox* and compute the total and display it in a *TextBox*.
4. When a new order is generated, the old order information should be cleared from the display.