Execution control structures and repetition

Introduction to control structures and repetition

Execution control structures are used to conditionally execute portions of code. The C# control structures are

if, *else*, and *else if*
switch-case
goto, *break*, *continue*, *return*

Repetition (iteration) of an operation or sequence of operations is achieved through the use of loops. The C# loop structures are

for
while
do-while
foreach

The following describes the syntax for the conditional control structures and loop structures in C#.

Conditional Control: if, else, and else if

The syntax of the conditional control if is:

if (*Boolean expression*) *expression to execute*;

or

if (*Boolean expression*)
{
    expression to execute;
}

For this expression, if the Boolean expression is true then the expression following it will be executed, otherwise the expression is skipped.

Create windows form in Visual Studio and add a textbox and button to it. Make sure the textbox is named textBox1 and the button is button1. Double click on the button to get the method that executes when the button is clicked. You should see the following

```
private void button1_Click(object sender, System.EventArgs e)
{
```

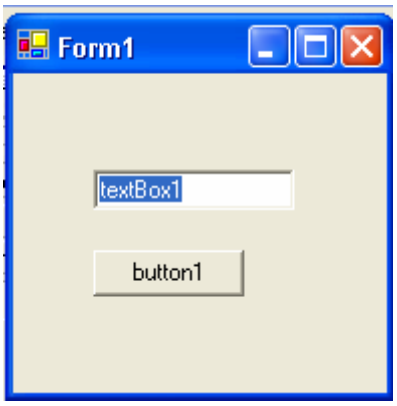


```
}
```

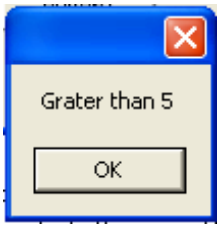
Add the following code all on one line to

```
if(Convert.ToInt32(this.textBox1.Text)>5) MessageBox.Show("Grater than
5");
```

Build the windows form and then run it. You should get the following window:



If you type a number less than five and press the button, nothing should happen since the expression is false and therefore the code following the Boolean control expression will not execute. If you type a number larger than 5 in the text box and press the button, a message box will open as follows



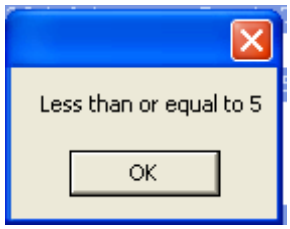
It would be nice if we could have the program provide a message to tell us is the number is not grater than five. This can be done with the *if-else* structure that follows:

```
if (Boolean expression)
{
        expression to execute if Boolean expression is true;
}
else
{
        expression to execute if Boolean expression is false;
}
```

In this structure if the Boolean expression is true, the code in the "{}" following the if will execute, otherwise the expression in the "{}" after the else will execute. As an example, in the previous example replace your code by the following code:

```
if(Convert.ToInt32(this.textBox1.Text)>5)
{
      MessageBox.Show("Grater than 5");
}
else
{
      MessageBox.Show("Less than or equal to 5");
}
```

Now if you run your application and type in an integer less than six you will get the following message box:



Now this is fine, but sometimes you want to do one of a number of different things based on which one of a set of different conditions comes true. The *if-else if-else* structure is one method to do this. The structure is as follows:

```
if (Boolean expression 1)
{
      expression to execute if Boolean expression 1 is true;
}
else if (Boolean expression 2)
{
      expression to execute if Boolean expression 1 is false, but Boolean expression 2 is
      true;
}
else if (Boolean expression3)
{
      expression to execute if Boolean expression 1 and 2 are false, but Boolean
      expression 3 is true;
}
else
{
      expression to execute if all Boolean expression are false;
}
```

In this structure if the Boolean expression in the first line is true, the code in the "{}" following it will execute and then jump to after the end of the structure, if it is false, it will jump to the next *else if* expression. It then tests the *else if*'s Boolean expression, if it

is true it will execute the code in the following "{}" and then jumps to after the end of the structure, or if it is false it goes to the next *else if* Boolean expression and continues the same process. If none of the Boolean expressions are true, the code in the "{}" after the last *else* will execute. As many *else if* clauses as desired can be put after the initial if clause.

Conditional Control: switch-case

When there is a set of cases which we want to select between, it is sometimes easier to use the *switch-case* structure that follows:

```
switch (variable)
{
        case value1:
                code to execute if variable=value1;
                break;
        case value2:
                code to execute if variable=value2;
                break;
        case value3:
                code to execute if variable=value3;
                break;
        default:
                code to execute if variable is not equal to any of the variables;
                break;
}
```

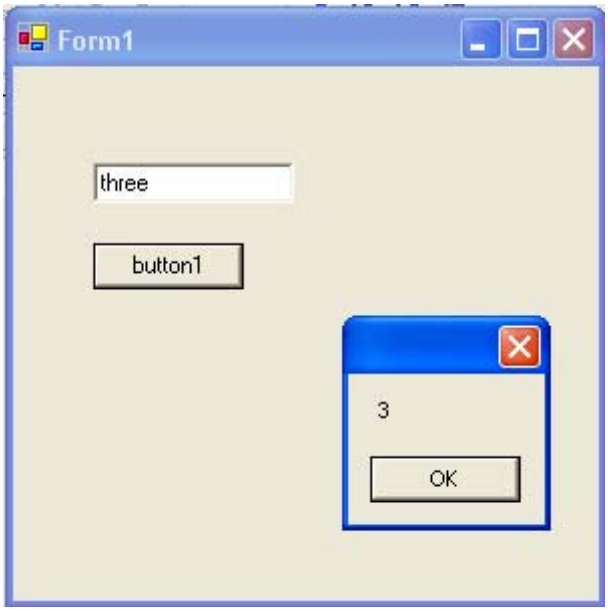
The structure works as follows. If the variable is equal to any of the values, the code after that value executes and exits to after the end of the switch structure, and if the variable does not equal any of the case value, then it executes the default and exits. You may use any number of cases as desired and may exclude the *default*.

The variable may be of any type, and the values must be of the same type. For example, in the example of the last section replace the code by the following:

```
switch (this.textBox1.Text)
{
       case "one":
              MessageBox.Show("1");
              break;
       case "two":
              MessageBox.Show("2");
              break;
       case "three":
              MessageBox.Show("3");
              break;
```

```
        case "four":
              MessageBox.Show("4");
              break;
        case "five":
              MessageBox.Show("5");
              break;
        default:
              MessageBox.Show("Out of range");
              break;
}
```

Now if you build and run the program, you can type one, two, three, four, or five in the text box and press the button then it will show the integer number that it represents as follows:



If you type any other word, the application will show a message box stating `"Out of range."`

Loop structure: for

The *for-loop* allows us to execute an operation repetitively for a number of times. The syntax of the *for-loop* is as follows

for (*initializer*; *condition*; *iterator*) *code to execute*;

or

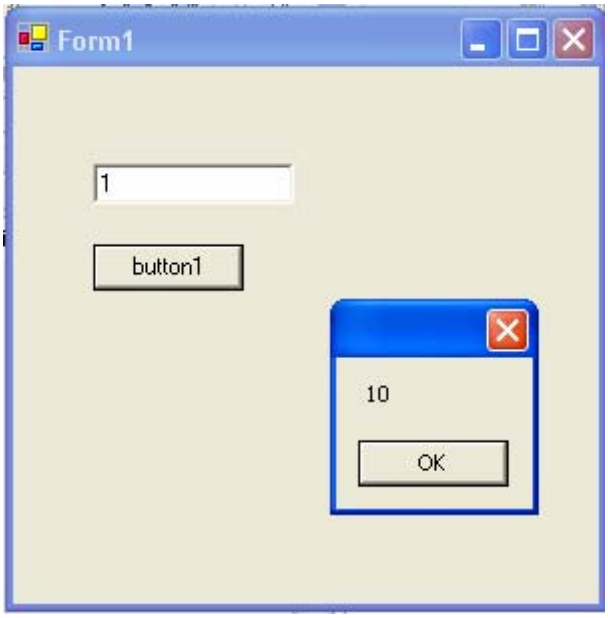
for *(initializer*; *condition*; *iterator*)
{
    lines of code to execute;
}

The *initializer* is the expression that is executed before the loop is started and normally is normally used to initialize the counter of the loop. The *condition* is a Boolean expression that is checked before each loop and if it is true the loop will execute. The *iterator* is an expression that will be evaluated after each loop and is normally used to increment the counter.

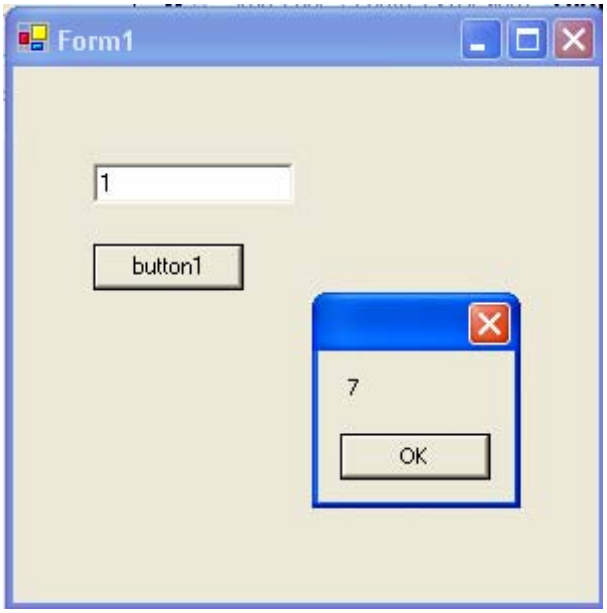
As an example of a for-loop replace the following code into your application:

```
int sum = Convert.ToInt32(this.textBox1.Text);
for (int i=2; i<5; i++)
{
      sum += i;
}
MessageBox.Show(sum.ToString());
```

If you type a number in the text box and press the button, the program will take this number and add 2, 3, and 4 to it and then stop and display the message box.



As another example, change the i++ in the *iterator* to i += 2 and build and run the application. Now you will get



In this case, the loop started with i=2 the first time, and added 2 to get i=4 for the second time and stopped before the next iteration since i=6.

Loop structure: while

The *while* structure works similar to the *for-loop*, but does not have the *initializer* and *iterator*. The syntax for the *while* loop is:

while (*condition*) *code to execute each time*;

or

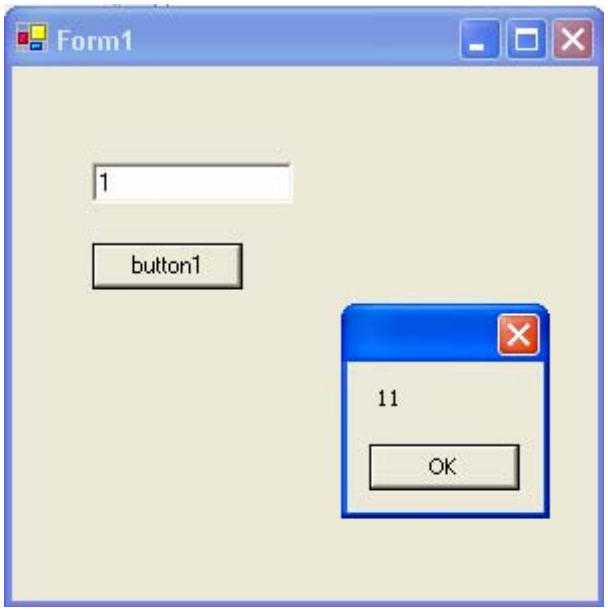
while (*condition*)
{
	code to execute each time;
}

For the *while* loop, the code inside "{}" will execute over and over until the *condition* becomes false. The condition is checked before the execution of the code each time.

As an example of the *while-loop*, replace the following code into your application:

```
int sum = Convert.ToInt32(this.textBox1.Text);
while (sum<10)
	{
		sum += 2;
	}
MessageBox.Show(sum.ToString());
```


If you build and run it, you will get a message box looking as follows. What happened is that 2 is added to sum until it becomes larger than 9, so that if you start with sum = 1, it will stop when sum becomes 11.



Loop structure: do while

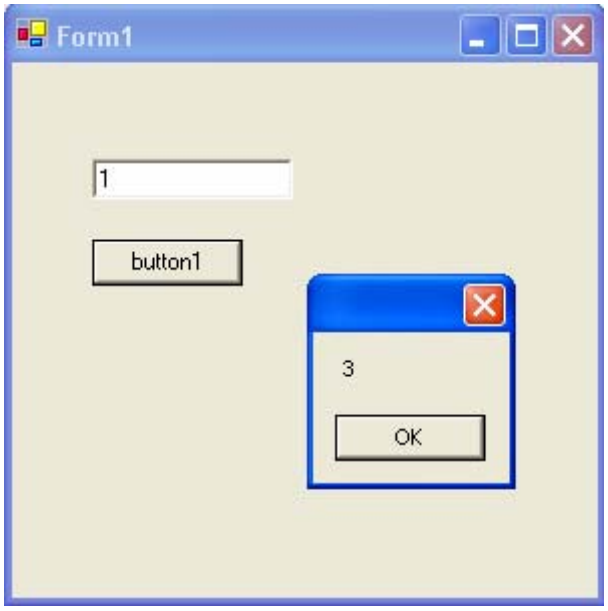
The *do-while* structure works similar to the *while-loop*, but the condition is tested after the code is executed so the code executes at least once:

```
do
{
        code to execute each time;
} while (condition)
```

For the *do-while* loop, the code inside "{}" will execute over and over until the *condition* becomes false. The condition is checked after the execution of the code each time.

If you run the previous example, but with a do-while loop as follows, you will get the output shown.

```
int sum = Convert.ToInt32(this.textBox1.Text);
do
{
      sum += 2;
}while (sum<0);
MessageBox.Show(sum.ToString());
```



Note that even though 1<0, the loop executed once before the condition was tested and, therefore, resulted in a sum of 3.

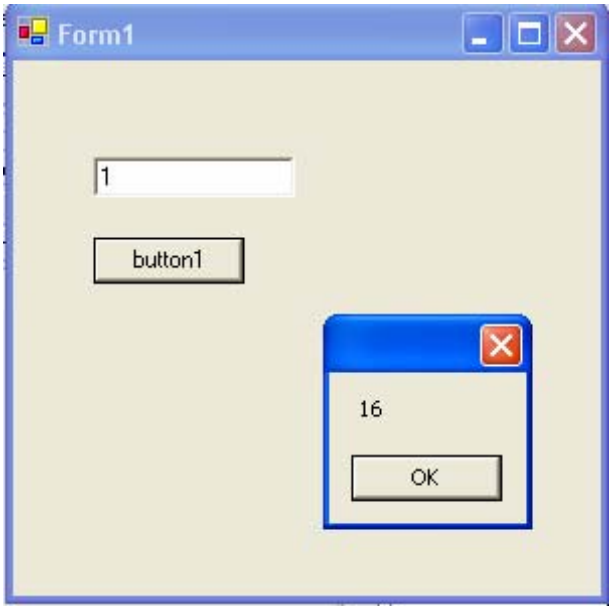
Loop structure: foreach

The *foreach-loop* is like a *for-loop*, but will execute the given code once for each item of a collection of items. The *foreach-loop* syntax is as follows:

foreach (*item in item-collection*)
{
	conde lines to execute for each item;
}

The following example shows the *foreach-loop* used with an array of integers. In this case it will add the three integers 3, 5, and 7 contained in the integerCollection to the sum. Type the following code into you application, build and run it to get the following output.

```
int sum = Convert.ToInt32(this.textBox1.Text);
int[] integerCollection = {3,5,7};
foreach (int item in integerCollection)
{
      sum += item;
}
MessageBox.Show(sum.ToString());
```

Redirection and jumps: goto

The *goto* command instructs the program to move from one location in the code to another. The syntax of the *goto* is as follows:

goto *label*;
label:

The label can be any word that is unique. The *goto* command provides an unstructured change in the sequence of commands executed and if used excessively can make following the execution flow difficult. It is recommended that it be used sparingly and only to redirect flow control to a following line.

Redirection and jumps: break

The *break* command as used above, directs the program to skip to the end of the control structure or loop. The break command will force the program to exit to the line following the control structure or loop. The syntax of the *break* is as follows:

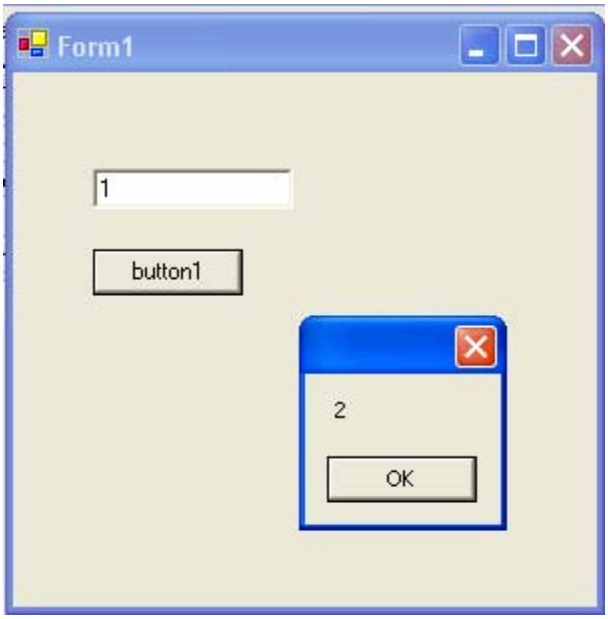
break;

As an example input the following code into your application.

```
int sum = Convert.ToInt32(this.textBox1.Text);
for (int i=0; i<4; i++)
```

```
{
	sum += i;
	if(sum==2) break;
	sum += i;
}
MessageBox.Show(sum.ToString());
```

The *for-loop* should stop as soon as the sum = 2 after the fist addition at the start of the loop. The output will be as follows:



Redirection and jumps: continue

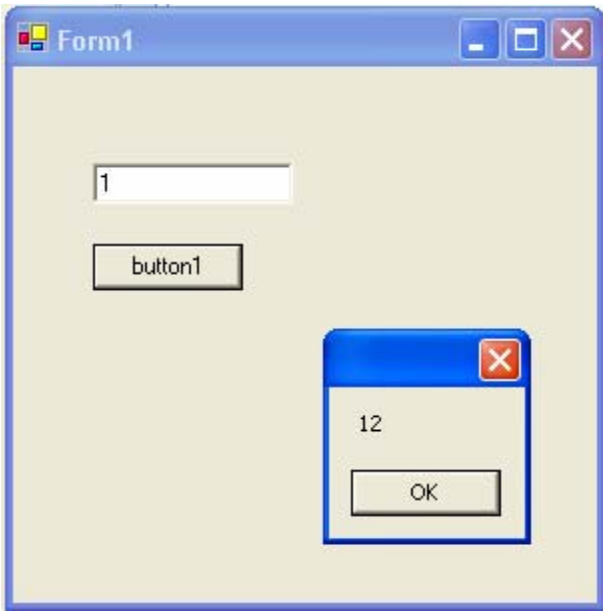
The *continue* command is used in the loops to end the processing of one iteration, sending the program to the start of the next iteration in the loop. Unlike the *break* that completely ends the execution of the loop, the *continue* command only ends the current iteration of the loop. The syntax of the *continue* command is as follows:

```
continue;
```

To see how the continue command works, type the following in your application:

```
int sum = Convert.ToInt32(this.textBox1.Text);
for (int i=0; i<4; i++)
{
	sum += i;
	if(sum==2) continue;
	sum += i;
}
MessageBox.Show(sum.ToString());
```

```
As can be seen, the second summation will be skipped only in the loop
that the first sum = 2. In this case the output will be:
```



Redirection and jumps: return

The *return* command as used to exit methods. If the method is return is void, the syntax of the *return* is as follows:

return;

If the method returns a value, the *return* must be followed by the *value* to be returned. The syntax of the *return* in this case must be:

return *value*;