

Inft2012 Application Programming

Lecture 5

Iteration

Sequence, selection , *iteration*

- We've seen a program as a *sequence* of statements; we've also seen *selection*, having the program choose which code to run based upon some test
- The real power of the computer lies in its ability to do the same thing over and over with minor changes; for example, to calculate the weekly pay of thousands of employees, using the same process but different data for each one
- *Iteration* (another word for repetition) is the continual execution of a piece of code until some stopping condition becomes true – or while some continuing condition remains true

While loop

- A 'loop' is the generic name for an iteration construct in programming
- Imagine serving free fruit salad at a cafeteria, where we want to keep serving while there are people to serve

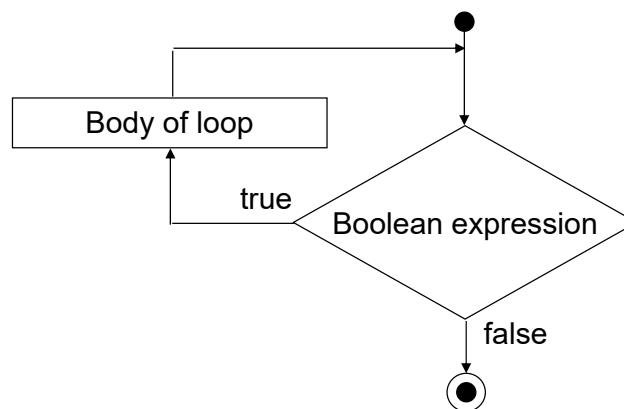
```
while (queueLength > 0)
{ <serve fruit salad>
}
```
- More generally,

```
while (<boolean expression>)
{ <statements>
}
```
- The <statements> are called the 'body' of the loop

3

Flowchart for while loop

- Note from the diagram that if the boolean expression is false at the start, the body won't be executed



4

Example of while loop

- Display the integers from 1 upward and their squares, so long as the squares are below a specified integer
- (Display nothing if the specified integer is below 1)

```
TbxOutput.Clear();
int i = 1;
int iSquare = 1;
while (iSquare <= iLimit)
{
    TbxOutput.AppendText(String.Format
        (" {0:d}   {1:d}\r\n", i, iSquare));
    i = i + 1;
    iSquare = i * i;
}
```

- (AppendText is a useful method of Textbox)

5

Pre-testing and post-testing

- The while loop *pre-tests* – it tests the boolean expression before getting to the body, and might not execute the body at all
- Sometimes we require a *post-test* – we want to do the body at least once, then test to see whether to keep looping
- Example: the user plays a game; when it's finished, ask if the user wants to play again, and repeat the game if the answer is yes
- It would be silly to ask if the user wants to play again before the first game has been played
- So we check the user's answer after the body, the game

6

Do-while loops

- The do-while loop offers a post-test

do

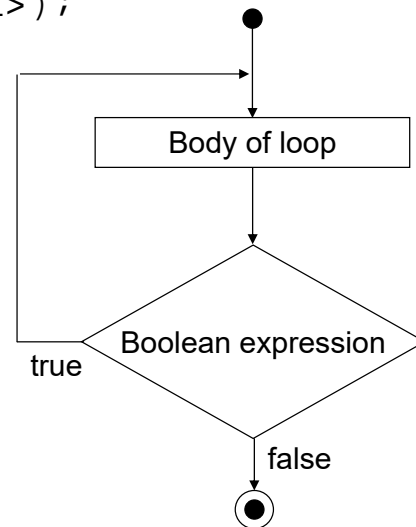
```
{ <statements> }
```

```
while (<boolean expression>);
```

- It ends with a semicolon (because it doesn't end with a brace)

- For example,

```
do {  
    <play game>;  
    <ask play again>;  
}  
while (<answer is yes>);
```



7

Example of do-while loop

- Display the integers from 1 upward and their squares, until computing a square above a specified positive integer; this is sure to loop at least once

```
TbxOutput.Clear();  
int i = 0;  
int iSquare;  
do  
{  
    i = i + 1;  
    iSquare = i * i;  
    TbxOutput.AppendText(String.Format  
        (" {0:d}    {1:d}\r\n", i, iSquare));  
}  
while (iSquare <= iLimit);
```

8

For loops

- The *for* statement is designed to repeat a piece of code a set number of times, while providing a counter that increments each time through the code
- Unfortunately, its syntax is not at all intuitive, and is open to confusing uses for which it wasn't intended

```
for (<initial statement>; <continuing
condition>; <loop-end statement>)
{ <statements (loop body)> }
```
- The *initial statement* is executed once, before looping begins;
- looping continues while the *continuing condition* is true;
- the *loop-end statement* is done at the end of each loop

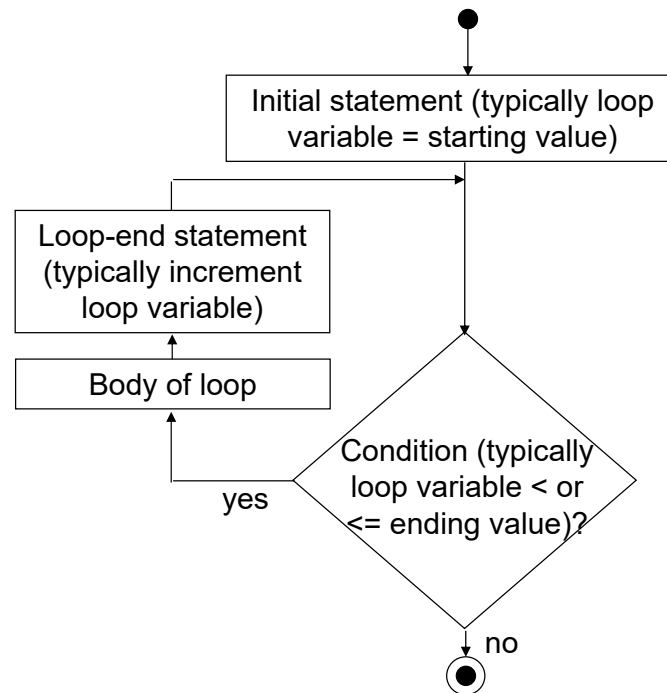
9

Typical for statement

- In a typical *for* statement . . .
 - the initial statement is used to (declare and) initialise a loop variable that will be incremented each time round the loop
 - the continuing condition is used to check the value of the loop variable
 - the loop-end statement is used to increment or decrement the value of the loop variable
- While other uses are possible, they are not recommended

Flowchart for *for*

- Notice that if the condition is false to start with (eg the starting value is greater than the ending value), the body won't be done at all



11

Example of for loop

- Display the integers from 1 to a specified integer and their squares; display nothing if the specified integer is below 1

```
TbxOutput.Clear();
int iSquare;
for (int i = 1; i <= iLimit; i++)
{
    square = i * i;
    TbxOutput.AppendText(String.Format
        (" {0:d}    {1:d}\r\n", i, iSquare));
}
```

- Think of this as “for every value of *i* from 1 to *iLimit*”
- Note that *i++* is shorthand for *i = i + 1*
- Lec5Demo1LoopTests demonstrates the three loops

12

Using the counter

- Just doing the same thing over and over again is generally pointless

```
for (int i = 1; i <= 10; i++)  
{ Flag(30, 30, 30); }
```

- What makes it useful is when we use the value of the loop counter to vary aspects of what's being done

```
for (int i = 1; i <= 10; i++)  
{ Flag(30 * i, 30 * i, 30); }
```

- Explore Lec5Demo2FlagLoops – check what each button does, and look at the code that makes it do that
- (Is 'i' an informative variable name?)

13

Classes, objects, and forms

- Lec5Demo2FlagLoops introduces some additional concepts that we haven't yet discussed
- We've been aware that the form and its code together make up a *class* – in this case, *class FrmFlagLoops*
- A class is a template for some objects
- When we create a new object of that class, the object is created according to that template, and is an *instance* of the class
- How does running a program create a new instance of the form? In Solution Explorer, look at Program under Program.cs:

```
Application.Run(new FrmFlagLoops());
```

14

Instances and instance variables

- Our form is an instance of the class `FrmFlagLoops`
- While we generally only have one instance of a form at any one time, in principle we could have many; and with other sorts of object, such as students, we would certainly have many
- Each instance needs its own copy of certain attributes; for example, each student needs its own name
- These attributes are declared as *instance variables*: within the class, but not within any method of the class
- Instance variables (in this case, the paper and three brushes) are available to all code in the instance, ie to all of the code that is written in this class

15

The form constructor

- A method that has the same name as its class is the *constructor*: it is called when a new instance of the class is created
- All of the form constructors we have seen so far consist of the single instruction `InitializeComponent();`, which creates the form
- Anything else that we want done to initialise the form can be done in the constructor, after this statement
- With this form we need to initialise `grPaper` to the `CreateGraphics()` method of the picture box
- If you try this with the brush initialisations, you will see that it doesn't work; it needs to be in a method

16

Don't mess with the variable

- It's possible to alter the loop variable within the loop body ...

```
for (int i = 1; i <= 10; i++)  
{  
    Flag(30 * i, 30 * i, 30);  
    i = i * 2;  
}
```

- ... but good programmers never do this
- The main point of a for loop is that we (programmers) can see at a glance how many times it will execute
- If we play with the loop variable, we lose that ability

17

Varying the increment

- The increment doesn't have to be 1, and the starting and ending values don't have to be literals (explicit values)

```
for (iCount = iLow; iCount <= iHigh;  
     iCount = iCount + 5)  
{  
    <body of loop>  
}
```

```
for (i = iStart; i > iFinish; i = i - 13)  
{  
    <body of loop>  
}
```

18

Loops within loops

- The body of a looping statement can include any statements, including ifs and further looping statements

```
for (iCount = 1; iCount <= 10; iCount++)  
{  
    iFactorial = 1;  
    for (iNum = 1; iNum <= iCount; iNum++)  
    {  
        iFactorial = iFactorial * iNum;  
    }  
    <display iFactorial (the factorial of  
        iCount)>  
}
```

- Loops within loops are often called *nested loops*; see the Nested For loop button in Lec5Demo1LoopTests

19

Care with nested loops

- If you want one variable to range from one value to another value . . . while a second variable also ranges from one value to another value . . .
- . . . you don't necessarily want nested loops
- You might just want one loop and then another, or one loop in which two variables change
- In a nested loop, the second variable goes through its full range *for every value* of the first variable
- If the outer loop has a range of n , and the inner loop a range of m , the inner body will be executed $n \times m$ times
- Be sure that this is what you intend!

20

Which form to use?

- How do you decide which form of loop to use?
- There's no single right answer; you should use whichever method seems best suited to the job you need done – so long as you understand its features
- For loops are common when the program knows in advance how many loops to make
- The choice between while and do-while will generally hinge on whether at least one loop is required (post-test), or there are circumstances in which we don't want to execute the body at all (pre-test)

21

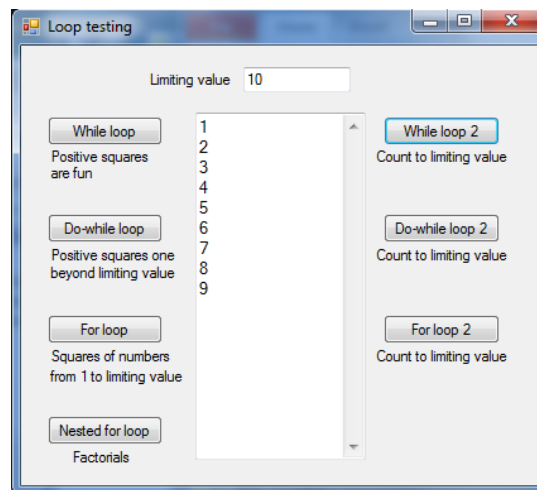
Care required with loops

- The different forms of loop are not entirely interchangeable
- When choosing a form of loop, you have to think about where its test is and whether its test is a stopping test or a continuing test
- These can have consequences on any initialisations you do and on any value you might use in the boolean expression
- Let's consider a simple example in which we use each form in turn to display the numbers from 1 to, say, 10
- (Continue with Lec5Demo1LoopTests)

22

While loop

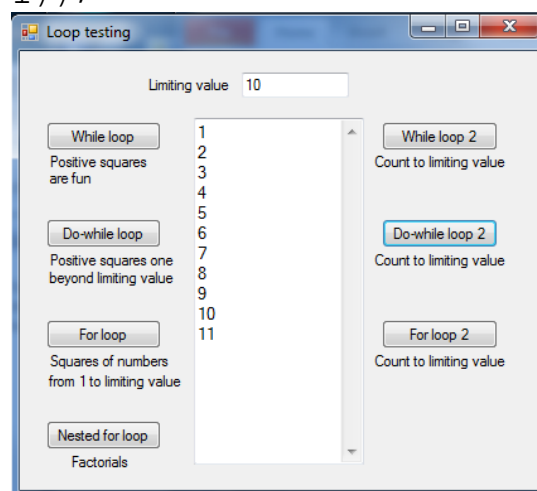
```
int i = 1;
while (i < iLimit)
{
    TbxOutput.AppendText(String.Format
        (" {0:d}\r\n", i));
    i = i + 1;
}
```



23

Do-while loop

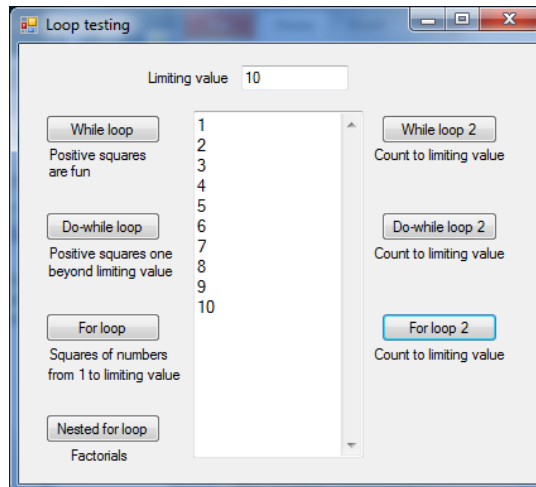
```
int i = 0;
do
{
    i = i + 1;
    TbxOutput.AppendText(String.Format
        (" {0:d}\r\n", i));
}
while (i <= iLimit);
```



24

For loop

```
for (int i = 1; i <= iLimit; i++)  
{  
    TbxOutput.AppendText(String.Format  
        (" {0:d}\r\n", i));  
}
```



25

Pre-testing and post-testing

- The distinction between pre-test and post-test wasn't really obvious in that run-through
- But what if the stopping value is less than the starting value?
- Set the stopping value to -3 and run the loops again
- This should show very clearly which loops are post-tested; that is, the body is executed once before the first boolean check is made

26

Debugging

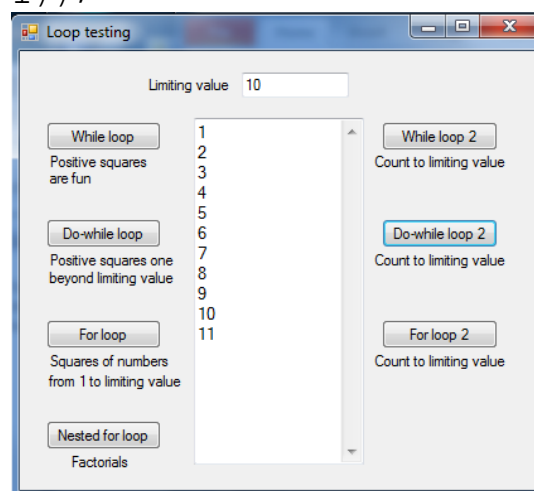
- It is very rare that your program will work the first time you run it
- The process of modifying your program to make it work as expected is called debugging
- Syntax errors, errors in the code that prevent it from making sense in C#, are picked up and pointed out as you write them
- Invalid data errors generally arise as a result of faulty data entry; we have seen a little about how to deal with these through exception handling
- That leaves logic errors, in which the program does what we told it rather than what we meant to tell it!
- To find these, we need to know what output we expect

27

Do-while loop

```
int i = 0;
do
{
    i = i + 1;
    TbxOutput.AppendText(String.Format
        (" {0:d}\r\n", i));
}
while (i <= iLimit);
```

- But we wanted
(and expected)
the numbers from
1 to 10!



28

Debugging

- We could deskcheck the code with pen and paper . . .
- . . . or we can step through the code on the computer and work out what happened
- First we'll reduce the endpoint from 10 to 3 so that we don't have so many iterations to check; this is a standard practice in debugging
- We could also set a breakpoint, as covered in last week's lecture, to temporarily suspend execution of the program to examine what is happening

29

Common errors with loops

- wrong start point
- wrong end point
- one loop too many
- one loop too few
- not changing the condition within the body (what consequence would this have?)
- looping once when no loop is required
- not looping when at least one loop is required
- Always desk check and/or debug the start and end of every loop you write

30

Sequence, selection, and iteration


- Programmers have long accepted that every piece of program code is made up of the three basic structures that we have now seen:
 - sequence
 - selection
 - iteration
- In fact this is a bit of a simplification
- For example, it doesn't accurately describe event-driven programming
- But as a generalisation, it works well within any defined piece of code such as a method

31

A little more about MessageBox

- The MessageBox Show() method has 21 different signatures – 21 different combinations of parameters
- How do we know? Look at the tool tip
- An option that might interest us at present is one that lets us specify what buttons should appear. As you've seen, if we don't specify, we get an OK button – but that isn't always enough.
- How do we find out about MessageBoxButtons?

{
 MessageBox.Show("Message", "Title", |
 5 of 21 DialogResult MessageBox.Show(string text, string caption, MessageBoxButtons buttons)
}
buttons: One of the System.Windows.Forms.MessageBoxButtons values that specifies which buttons to display in the message box.



Using Help

- We could try the online Help. Type MessageBox in the search box and press the search button
- Unfortunately, often either you can't find what you're looking for . . .
- . . . or you find countless blog questions asking variations on the same thing, and don't know which to choose . . .
- . . . or you get a really thorough reference, even for professionals, which means it can be completely overwhelming for beginners
- Getting actual help from online help is a true skill

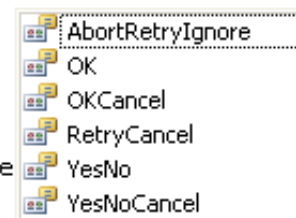
33

The value of autocomplete

- Fortunately, autocomplete gives us all the help we need, and we don't even have to look it up!
- Let's choose the YesNo option

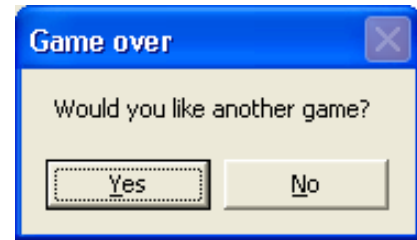
```
{  
    MessageBox.Show("Message", "Title", MessageBoxButtons.  
        return 0;  
}
```

```
ate void WhileButton_Click(object sender, EventArgs e
```



34

Using the answer



- Next question: how do we know what the user clicks?
- We see from that huge tool tip that `MessageBox.Show` is in fact a function – it returns a result; and the type of the result is `DialogResult`
- Using either Help or autocomplete, we discover that we can write something like

```
do
    { <play game>    }
while (MessageBox.Show("Again?", "Game over",
    MessageBoxButtons.YesNo) == DialogResult.Yes);
```

35

A suggestion for braces

- You'll have noticed that it's sometimes hard to keep track of which braces go with which, especially if the indentation goes awry
- We've already suggested putting a comment with each closing brace, to help you keep track
- Here's a another tip: whenever you create a structure that involves braces (a method, an if statement, a loop, etc) . . .
 - *immediately* type the opening brace; the closing brace will be provided;
 - press Enter, then add the closing comment at the end;
 - and *then* go back and write the code between the braces

36