# Flow Control

In all the code worked on in previous chapters, the program has executed from top to bottom. This chapter describes two methods for controlling program flow:

1. Branching – Executes code conditionally i.e. *“execute this code if var x < 10”*
2. Looping – Repeatedly executes the same statements, either a certain amount of times or until a test condition has been reached.

Both techniques involve the use of Boolean logic.

## Boolean Logic

In the previous chapter the bool type introduced can hold on of only two values: **true** or **false**. Bool types are used to store the result of a comparison.

**Boolean Comparison Operators** (also known as relational operators)

|  |  |  |  |
| --- | --- | --- | --- |
| **OPERATOR** | **CATEGORY** | **True if** | **False if** |
| = | Equal to | expression1 = expression2 | expression1 <> expression2 |
| != | Not equal to | expression1 != expression2 | expression1 = expression2 |
| < | Less than | expression1 < expression2 | expression1 >= expression2 |
| > | Greater than | expression1 > expression2 | expression1 <= expression2 |
| <= | Less than or equal to | expression1 <= expression2 | expression1 > expression2 |
| >= | Greater than or equal to | expression1 >= expression2 | expression1 < expression2 |

Take the following example code:

bool isLessThan10;  
isLessThan10 = myVal < 10;

isLessThan10 will be assigned the value true if myVal stores a value of less than 10, or false otherwise.

Can also use these comparison operators on other data types such as strings. For example:

bool isShea;  
isShea = myString == “Shea”;

Here isShea is true if myString stores the string “Shea”.

Can also compare variables with Boolean values.

bool isTrue;  
isTrue = myBool == true;

However, here you are limited to == and != operators.

|  |  |  |  |
| --- | --- | --- | --- |
| **OPERATOR** | **CATEGORY** | **EXAMPLE EXPRESSION** | **RESULT** |
| && | Binary | var1 = var2 && var3; | var1 is assigned the value **true** if var2 and var3 are both **true**, or **false** otherwise. (logical AND) |
| || | Binary | var1 = var2 || var3; | var1 is assigned the value **true** if either var2 or var3 is (or both) **true**, or **false** otherwise. (logical OR) |

The result of these operators is exactly the same as & and |, but there is an important distance in the way it is obtained, which can result in far better performance.

Both look at the values of the first operand (var2 above), and based on the value of this operand may not need to process the second operands (var3 above).

If the value of the first operand of the && operator is false, then there is no need to consider the value of the second operand, because the result will be **false** regardless. Similarly, the || operator returns **true** if its first operand is **true**, regardless of the second operand.

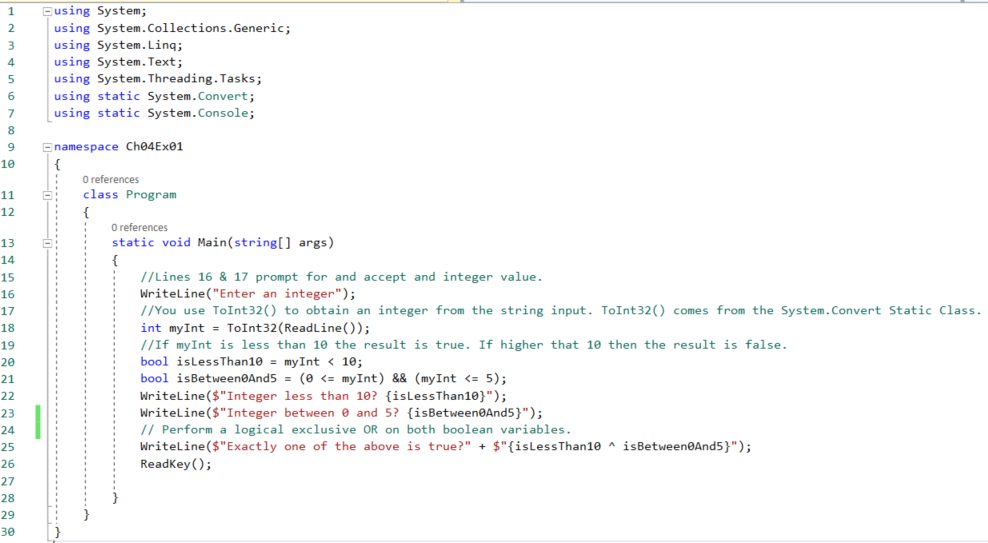
## Boolean Bitwise and Assignment Operators

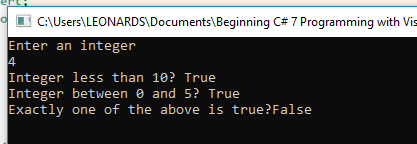
Boolean comparisons can be combined with assignments by combining Boolean bitwise and assignment operations. These work in the same way as mathematical assignment operators.

|  |  |  |  |
| --- | --- | --- | --- |
| **OPERATOR** | **CATEGORY** | **EXAMPLE EXPRESSION** | **RESULT** |
| &= | Binary | var1 &= var2; | var1 is assigned the value that is the result of var2 & var2. |
| |= | Binary | var1 |= var2; | var1 is assigned the value that is the result of var2 | var2. |
| ^= | Binary | var1 ^= var2; | var1 is assigned the value that is the result of var2 ^ var2. |

The equation var1 ^= var2 is similar to var1 = var2 where var1 = true and var2 = false.

### Exercise 1





However, this code is limited because if we are to enter in a string, an exception will occur. To handle this we can use a Try {} … Catch {} block or we could also check to see if the entered value is an integer before performing the conversion using the GetType() method.

### Operator Precedence

|  |  |
| --- | --- |
| **PRECEDENCE** | **OPERATORS** |
| Highest | ++, -- (used as prefixes); (), +, - (unary), !, ~ |
|  | \*, /, % |
|  | +, - |
|  | <<, >> |
|  | <, >, <=, >= |
|  | ==, != |
|  | & |
|  | ^ |
|  | | |
|  | && |
|  | || |
|  | =, \*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |= |
| Lowest | ++, -- (used as suffixes) |

The && is processed after the <= and >= operators below,.

var1 **=** var2 **<=** 4 **&&** var2 **>=** 2**;**

Doesn’t hurt to add parenthesis to make expressions clearer. For example:

var1 **=** **(**var2 **<=** 4**)** **&&** **(**var2 **>=** 2**);**

## Branching

Branching is the act of controlling which line of code should be executed next. The line to jump to is controlled by some kind of conditional statement. This conditional statement is based on a comparison between a test value and one or more possible values using Boolean logic. Three branching techniques available in C#:

1. The ternary operator
2. The if statement
3. The switch statement

### The Ternary Operator

The simplest way to perform a comparison is to use the *ternary (or conditional) operator.* For example, you might use the following to test the value of an **int** variable called **myInteger**.

string resultString **=** **(**myInteger **<** 10**)** **?** "Less than 10" **:** "Greater than or equal to 10"**;**

The result of the ternary operator is one of two string, both of which may be assigned to resultString. The choice of which string to assign is made by comparing the value of myInteger to 10. Therefore, a value of less that 10 will result in the first string being assigned, and a value of greater than 10, the second string will be assigned.

### The if Statement

The if statement is a far more versatile and useful way to make decisions. Unlike ?: statements, the if statements don’t have a result, instead you use the statement to conditionally execute other statements.

An if statement in its simplest form:

**if** **(<**test**>)**

**<**code executed **if** **<**test**>** **is** **true>;**

If <test> is evaluated to true then the code that follows will compile. If <test> returns false, program execution resumes at the next line of code. You can also use the else statement in conjunction with the if. For example, this statement is executed if <test> evaluates to false.

**if** **(<**test**>)**

**{**

**<**code executed **if** **<**test**>** **is** **true>;**

**}**

**else**

**{**

**<**code executed **if** **<**test**>** **is** **false>;**

**}**

Code from Ternary example above.

string resultString **=** **(**myInteger **<** 10**)** **?** "Less than 10" **:** "Greater than or equal to 10"**;**

Rewritten using the if statement:

string resultString**;**

**if** **(**myInteger **<** 10**)**

**{**

resultString **=** "Less than 10"**;**

**}**

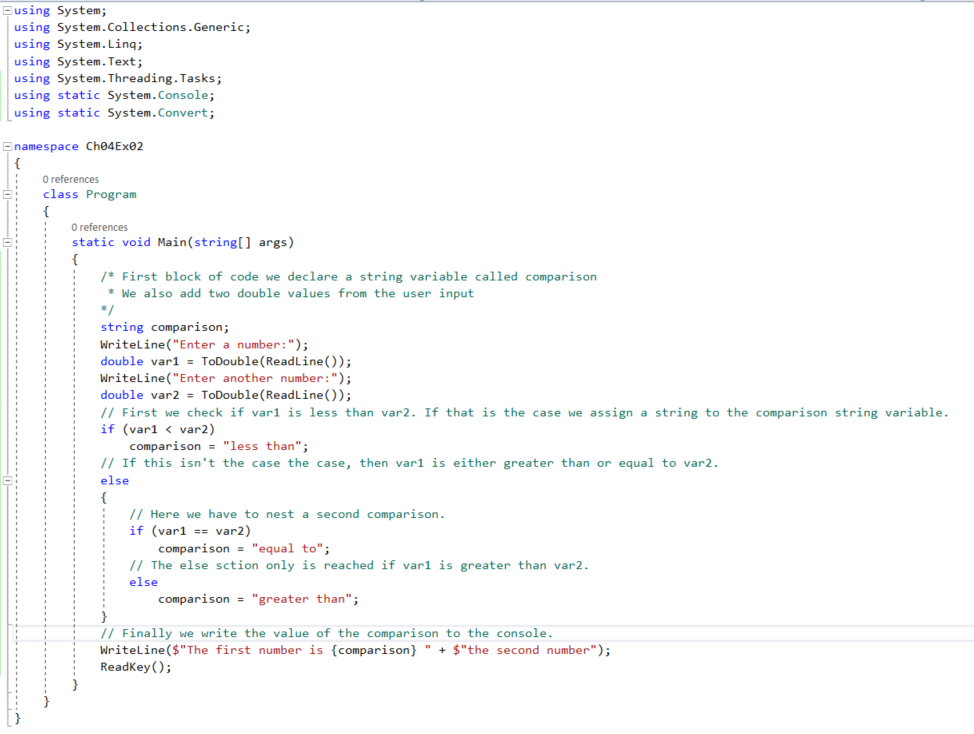
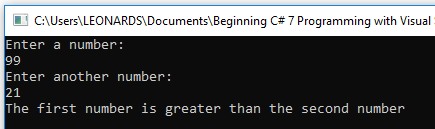
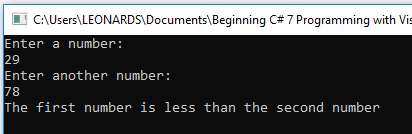
**else**

**{**

resultString **=** "Greater than 10"**;**

**}**

### Exercise 2



### Checking More Conditions Using if Statements

Consider the following else if statements.

**if** **(**var1 **==** 1**)**

**{**

// Do something

**}**

**else** **if** **(**var1 **==** 2**)**

**{**

// Do something

**}**

**else** **if** **(**var1 **==** 3 **||** var1 **==** 4**)**

**{**

// Do something

**}**

**else**

**{**

// Do something

**}**

These else if statements are really just two separate statements, and the code if functionally identical to the previous code, but much easier to read. If making multiple comparisons like this, consider using the switch statement as an alternative branching structure.

### The switch statement

The switch statement is similar to the if statement in that it will execute code conditionally based on the value of a test. However, switch enables you to test for multiple values of a test variable in one go, rather than just a single condition.

This test is limited to discrete values, rather than clauses such as, “greater than X,” so its use is slightly different; however it can be a powerful technique. Take the following example:

**switch** **(<**testVar**>)**

**{**

**case** **<**comparisonVal1**>:**

**<**code to execute **if** **<**testVar**>** **==** **<**comparisonVal1**>** **>**

**break;**

**case** **<**comparisonVal2**>:**

**<**code to execute **if** **<**testVar**>** **==** **<**comparisonVal2**>** **>**

**break;**

**case** **<**comparisonVal3**>:**

**<**code to execute **if** **<**testVar**>** **==** **<**comparisonVal3**>** **>**

**break;**

**default:**

**case** **<**comparisonVal4**>:**

**<**code to execute **if** **<**testVar**>** **==** **<**comparisonVal4**>** **>**

**break;**

**}**

What happens here is that the value in **(<**testVar**>)** is compared to each of the **<**comparisonVal1**>** values (specified with case statements). If there is a match then the code supplied is executed. If no match then the default section is executed.

The **break** statement here simply terminates the switch statement, and processing continues on the statement following the structure.

Another alternative method for preventing flow from one case statement to the next is to use the **return** statement. The return statement results in the termination of the current function. You could also use a **goto** statement. The **goto** statement transfers the program control directly to a labelled statement. For example:

**switch** **(<**testVar**>)**

**{**

**case** **<**comparisonVal1**>:**

**<**code to execute **if** **<**testVar**>** **==** **<**comparisonVal1**>** **>**

**goto** **case** **<**comparisonVal2**>;**

**case** **<**comparisonVal2**>:**

**<**code to execute **if** **<**testVar**>** **==** **<**comparisonVal2**>** **>**

**break;**

**...**

Here’s one exception to the rule that the processing of one case statement can’t run freely into the next. If you place multiple case statements together (stack them) before a single block of code, then you are in effect checking for multiple conditions at once. If any of these conditions are met then the code is executed. For example:

**switch** **(<**testVar**>)**

**{**

**case** **<**comparisonVal1**>:**

**case** **<**comparisonVal2**>:**

**<**code to execute **if** **<**testVar**>** **==** **<**comparisonVal1**>** or **<**testVar**>** **==** **<**comparisonVal2**>**

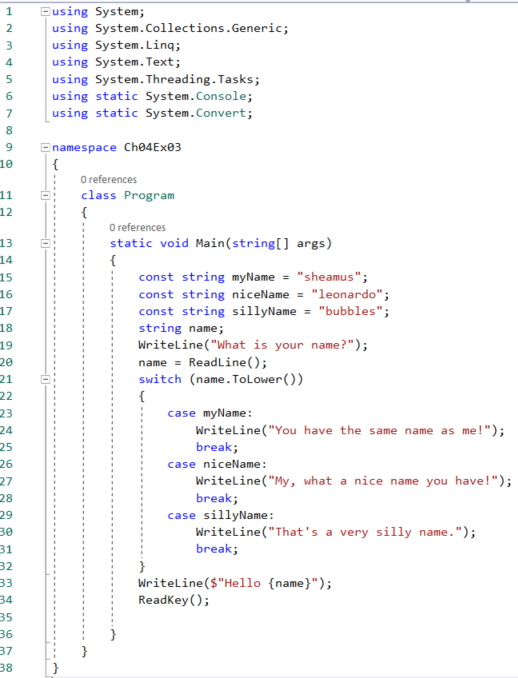
**break;**

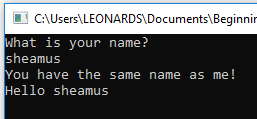
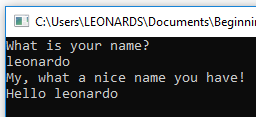
**...**

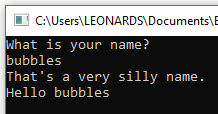
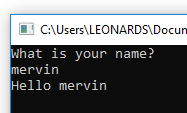
These conditions also apply to the default statement. There is no rule stipulating that this statement must be the last in the list of comparisons, and you cans tack it with case statements if you want.

Adding a breakpoint with break, or return, ensures that a valid execution path exists through the structure in all cases.

### Exercise 3







#### How it works

* The code sets up three constant strings, accepts a string from the user and then writes it out to the console.
* When comparing what the user has entered to your variable values, we force it into lowercase using the ToLower command which is from the System.Convert class. This is a command that works with all string variables and comes in handy when you’re not sure what the user has entered.
* The switch statement itself attempts to match the string entered with the constant values defined, and if successful writes to the console a personalised messaged. If no match made you get a generic greeting.

## Looping

Looping refers to the repeated execution of statements. This technique means you can repeat operations as many times as you want, without having to write the same code over and over.

Take for instance the following code. If we wanted to calculate the amount of money in a bank account after 10 years, assuming that interest is paid each year and no other money flows into or out of the account.

double balance **=** 1000**;**

double interestRate **=** 1.05**;** // 5% interest/ year

balance **\*=** interestRate**;**

balance **\*=** interestRate**;**

balance **\*=** interestRate**;**

balance **\*=** interestRate**;**

balance **\*=** interestRate**;**

balance **\*=** interestRate**;**

balance **\*=** interestRate**;**

balance **\*=** interestRate**;**

balance **\*=** interestRate**;**

balance **\*=** interestRate**;**

Writing the same code 10 times is quite wasteful and has limitations if we wanted to change the duration from 10 years to another value. Instead we can have a loop that executes the instruction a number of times.

Another important type of loop is one in which you loop until a certain condition is fulfilled.

### do Loops

do Loops operate as follows.

1. The code you have marked out for looping is executed.
2. A Boolean test is performed
3. The code executes again to evaluate if this test is true, and so on.
4. When the test evaluates to false, the loop exits.

For example you could write the following code to write numbers from 1 to 10 in a column:

int i **=**1**;**

**do**

**{**

WriteLine**(**"{0}"**,** i**++);**

**}** **while** **(**i **<=** 10**);**