Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. C# is a type-safe language, and the C# compiler guarantees that values stored in variables are always of the appropriate type e.g. you cannot store an int in a string variable.

Variable Data types include:

string – Represents text as a series of Unicode characters e.g. string name = “Matthew Dorrian”;

int – Represents whole numbers e.g. int age = 21;

decimal – Represents numbers that can decimal points (higher accuracy than double/ can accept more decimal places) e.g. decimal cost = 14.55;

double – Represents numbers that can decimal points e.g. double cost = 14.55;

bool – Represents true or false e.g. bool flag = true;

char – Represent a character of text e.g. char YesOrNo = ‘N’;

byte – Represent 8 bit number (0 – 255) e.g. byte a = 240;

Keywords

Keywords are predefined, reserved identifiers that have special meanings to the compiler. They cannot be used as identifiers in your program unless they include @ as a prefix. For example, @if is a valid identifier but if is not because if is a keyword.

Example Keywords include:

void - When used as the return type for a method, **void** specifies that the method doesn't return a value.

static - Use the **static** modifier to declare a static member, which belongs to the type itself rather than to a specific object. The **static** modifier can be used with classes, fields, methods, properties, operators, events, and constructors, but it cannot be used with indexers, destructors, or types other than classes.

override - The **override** modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

Access Modifiers

All types and type members have an accessibility level, which controls whether they can be used from other code in your assembly or other assemblies. You can use the following access modifiers to specify the accessibility of a type or member when you declare it:

public - The type or member can be accessed by any other code in the same assembly or another assembly that references it.

private - The type or member can be accessed only by code in the same class or struct.

protected - The type or member can be accessed only by code in the same class or struct, or in a class that is derived from that class.

internal - The type or member can be accessed by any code in the same assembly, but not from another assembly e.g. same namespace

Convert

At times you will need to convert a variable from one base type to another base type e.g. string to int. To do this, C# has a built in `Convert` class and has several members to convert types.

Example type conversions:

`Convert.ToDouble()` – converts to a double

`Convert.ToInt32()` – converts to a int

`Convert.ToDecimal()` – converts to a decimal

Writing to and Reading from the Console

The Console is used to represent the standard input, output, and error streams for console applications. We usually use the console to write data to the console and to allow users to enter data into the console and use this data.

To write data to the console, the console has two members we usually use which are:

`Console.WriteLine()` – Write data to the console followed by a new line e.g Console.WriteLine("Hello");

`Console.Write()` - Write data to the console followed e.g. Console.Write("Age: " + 21);

To read data from the console, the console has members to read the data:

`Console.ReadLine()` - Reads the next line of characters from the standard input stream

Methods

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments.

Methods are declared in a class or struct by specifying the access level such as **public** or **private**, optional modifiers such as **abstract** or **sealed**, the return value, the name of the method, and any method parameters. These parts together are the signature of the method.

Methods can return a value to the caller. If the return type, the type listed before the method name, is not **void**, the method can return the value by using the **return** keyword. A statement with the **return** keyword followed by a value that matches the return type will return that value to the method caller. The **return** keyword also stops the execution of the method.

If the return type is **void**, a **return** statement without a value is still useful to stop the execution of the method. Without the **return** keyword, the method will stop executing when it reaches the end of the code block. Methods with a non-void return type are required to use the **return** keyword to return a value.

Example methods:

```
public void PrintName(string name)
{
    Console.WriteLine(“Name: ” + name);
}

public int GetAge()
{
    int age = 21;
    return age;
}
```

Comments

You can easily add comments to any part of your program by using // for single line comments or /* */ for multiline comments. Example:

```
// This is a single Comment

/*
This is a multiline comment
Line Two
*/
```

String Escape Sequences

Strings allow you to use string escape sequences to perform certain string actions.

Example string escape sequences include:

`\n` - Takes a new line in a string

`\t` – Takes a horizontal tab E.g: Console.WriteLine(“Hello\nThis will be a new line\tTabbed”);

String Operations

The String class provides a number of methods that can be used to create new strings from existing ones as the String class is Immutable

Immutable means that the contents of the string object cannot be changed after the object is created.

Key Methods include:

- **ToLower()** – Convert a string to all lowercase

```
string name = "Matthew Dorrian";
Console.WriteLine(name);
string lowerCaseName = name.ToLower();
Console.WriteLine(lowerCaseName);
```

```
Matthew Dorrian
matthew dorrian
```

- **ToUpper()** – Convert a string to all uppercase

```
//ToUpperCase
string name = "Matthew Dorrian";
Console.WriteLine(name);
string upperCaseName = name.ToUpper();
Console.WriteLine(upperCaseName);
```

```
Matthew Dorrian
MATTHEW DORRIAN
```

- **Trim()** – Remove leading and trailing whitespace from a string

```
string spaces = "       Matthew Dorrian       ";
Console.WriteLine(spaces);
string trimmedName = spaces.Trim();
Console.WriteLine(trimmedName);
```

```
       Matthew Dorrian
Matthew Dorrian
```

- **Substring(int start)** – Gets the substring from the specified start position and the end of the string

```
//Substring
string name = "Matthew Dorrian";
Console.WriteLine(name);
string surname = name.Substring(7);
Console.WriteLine(surname);
```

```
Matthew Dorrian
Dorrian
```

- **Spilt(char delimiter)** – Separate strings into a string array using the delimiter

```
//Spilt
string name = "Matthew Dorrian";
Console.WriteLine(name);
string[] spiltName = name.Split(' ');
foreach (var item in spiltName)
{
    Console.WriteLine(item);
}
```

```
Matthew Dorrian
Matthew
Dorrian
```

Loops

One of the essential techniques when writing code is looping - the ability to repeat a block of number of times. In C#, they come in 4 different variants, and we will have a look at each one of them.

The while Loop –

The while loop is probably the most simple one, so we will start with that. The while loop simply executes a block of code as long as the condition you give it is true. It is a pre check loop meaning it checks the condition before the code executes e.g.:

```
//variable declaration
int number = 0;

//while number is less than 5
while (number < 5)
{
    //Print number and increment by 1
    Console.WriteLine(number);
    number = number + 1;
}
```

The do while Loop –

The opposite is true for the do while loop, which works like the while loop in other aspects through. The do loop evaluates the condition after the loop has executed, which makes sure that the code block is always executed at least once. It is a post check loop meaning it checks the condition after the code has been executed at least once e.g.:

```
//variable declaration
int number = 0;

//do while loop
do
{
    //print number and increment by 1
    Console.WriteLine(number);
    number = number + 1;
}
//while number is less than 5
while (number < 5);
```

The for Loop –

The for loop is a bit different. It's preferred when you know how many iterations you want, either because you know the exact amount of iterations, or because you have a variable containing the amount e.g.:

```
//variable declaration
int number = 5;

//for loop to loop 5 times
for (int i = 0; i < number; i++)
{
    //print the number represented by i
    Console.WriteLine(i);
}
```

The for loop consists of 3 parts - we initialize a variable for counting, set up a conditional statement to test it, and increment the counter (++ means the same as "variable = variable + 1").

The foreach Loop –

The last loop we will look at, is the foreach loop. It operates on collections of items, for instance arrays or other built-in list types.

```
//create a list that can accept string variables
List<string> list = new List<string>();

//add three strings to the list
list.Add("Matthew Dorrian");
list.Add("Narelle Allen");
list.Add("Neal Anderson");

//use a foreach toloop through the list
foreach (string name in list)
{
    //print the item
    Console.WriteLine(name);
}
```

We use the foreach loop to run through each item, setting the name variable to the item we have reached each time. That way, we have a named variable to output.

As you can see, we declare the name variable to be of the string type – you always need to tell the foreach loop which datatype you are expecting to pull out of the collection. In case you have a list of various types, you may use the object class instead of a specific class, to pull out each item as an object.

When working with collections, you are very likely to be using the foreach loop most of the time, mainly because it's simpler than any of the other loops for these kind of operations.

The if else statement

Often a value is not known and by using an if-statement, we make a logical decision based on it. In an if-statement, we test expressions. These evaluate to true or false.

In order to test multiple conditions, we can use if else if statements to check each condition and perform the applicable action. If no condition is met, you can ensure a statement is always ran by placing an end else statement e.g.:

```
//declare temperature as int
int temperature = 40;

//check temperature and print applicable statement
if (temperature > 45)
{
    Console.WriteLine("Porridge is too hot");
}
else if (temperature < 35)
{
    Console.WriteLine("Porridge is too cold");
}
else
{
    Console.WriteLine("Porridge is just right ... nom");
}
```

Boolean operators

C# provides a large set of operators, which are symbols that specify which operations to perform in an expression.

Operations on integral types to use in conditional statements e.g. if else, while include:

== equal to

!= not equal to

> greater than

< less than

<= less than or equal to

>= greater than or equal to

Conditional operators include:

&& Conditional AND

|| Conditional OR

! Conditional NOT

The switch statement

The switch statement is like a set of if statements. It's a list of possibilities, with an action for each possibility, and an optional default action, in case nothing else evaluates to true. A simple switch statement looks like this:

```
//declare variable to 1
int number = 1;

//use the switch to check the variable against two conditions
switch (number)
{
    //if number is 0, print 0 statement
    case 0:
        Console.WriteLine("The number is zero!");
        break;
    //if number is 1, print 1 statement
    case 1:
        Console.WriteLine("The number is one!");
        break;
    //use break keyword to break out of the switch statement
}
```

The identifier to check is put after the switch keyword, and then there's the list of case statements, where we check the identifier against a given value.

You will notice that we have a break statement at the end of each case. C# simply requires that we leave the block before it ends.

Classes

A *class* is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events. A class is like a blueprint. It defines the data and behaviour of a type.

Every public class has a default constructor. This is inserted by the C# compiler and is not shown in the C# code. It receives no parameters. It has no internal logic. It is removed if you add an explicit constructor. If you have an explicit constructor, you can still declare a default blank constructor.

Explicit Constructors allow the user to create instances of classes and allows the user to pass parameters to the constructor to populate the attributes of the instance.

Example class shown below:

```
public class Animal
{
    //Class varibles
    public string Name;
    public double Weight;

    //Custom Constructor
    3 references
    public Animal(string n, double w)
    {
        Name = n;
        Weight = w;
    }

    //Blank Constructor
    0 references
    public Animal()
    {

    }

    //Class Methods
    1 reference
    public void GetFat(double wi)
    {
        Weight += wi;
    }
}
```

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself.

An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the new keyword followed by the name of the class that the object will be based on, like this:

```
//Animal instance created using blank constructor
Animal blankDog = new Animal();

//Animal instance created using explicit constructor
Animal dog = new Animal("Matthew the dog", 50.5);
```