Chapter 2 : C# as a Language

C# is pronounced "C-Sharp".

C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg. It is an object-oriented programming language created by Microsoft that runs on the .NET Framework.

C# has roots from the C family, and the language is close to other popular languages like C++ and Java.

The first version was released in year 2002. The latest version, **C# 8**, was released in September 2019.

C# is used for:

- Mobile applications
- Desktop applications
- Web applications
- Web services
- Web sites
- Games
- Database applications

Why Use C#?

- It is one of the most popular programming language in the world
- It is easy to learn and simple to use
- It has a huge community support
- C# is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs.
- As C# is close to C, C++ and Java, it makes it easy for programmers to switch to C# or vice versa

Advantages of C# language.

The advantages of C# language must be known by every programmer or those who are interested in a programming language. There are many advantages and features of C# language that make it a more useful programming language than other programming languages like Java, C, C++, etc. In this article, I am going to tell about the advantages of C# language.

However, there are many advantages of C# language but some are the important advantages of C# language which are described here.

Object-oriented

C# programming language is a pure object-oriented language so that it allows you to create modular maintainable applications and reusable codes. This is one of the biggest advantages of C# over C++ languages.

Cross-Platform

The most important requirement for C# programming is the NET framework. Your machine has to install the NET Framework to run your application well.

Automatic Garbage Collection

In C# programming, a very efficient system installed that collects and erases garbage automatically present on the system. However, we called that C# language is very efficient in managing the system because it doesn't create a mess in the system, and the system doesn't get hanged during execution.

Avoid the problem of memory leak.

The major benefit of C# language is its strong memory backup. C# programming language contains high memory backup so that memory leakage problem and other such types of problem is not occurring as it happens in the case of C++ language.

Easy-to-Development

C# language has a rich class of libraries that make many functions easy to be implemented. The C# programming language influences most of the programmers of the world and has a history in the programming world.

Better Integration

An application written in .NET will have better integration and interpret-ability as compared to other NET Technologies. C# programming runs on C.L.R that making it easy to integrate with components written in other languages.

Cost-benefit

The maintenance cost is less and is safer to run as compared to other languages. C# language can develop iOS, Android and Windows Phone native apps, with the help of the Xamarin framework.

Familiar syntax

It is pretty easy to pick up and work productively with a working knowledge of languages like C, C++, Java because its core syntax is similar to C-style languages.

Programming support

You can buy support from Microsoft in C# programming. If any issue occurs you can solve it with the support of Microsoft.

Properties and Indexers

C# programming has features like Properties and Indexers which are not available in Java language.

Most useful

It can develop iOS, Android and Windows Phone native apps, with the help of the Xamarin framework. However, it is also greatly used for developing a Windows app (Mobile, Desktop).

Most Powerful

C# language is the most powerful programming language for the .NET Framework.

Motivate towards work

We already discussed that .NET applications work on Windows platforms only and Microsoft keeps retiring support for old Windows platforms. So always you would need to upgrade your .Net framework.

But after the upgrade, this could be an advantage or a disadvantage as well. Hence, it always motivates you to work hard and excel in your field and this is a good thing in my point of view.

Disadvantages of C# Language.

1. C# is completely based on Microsoft .Net framework this is the reason why this is not a flexible language.
2. As we know if we change anything in C# written code than we have to compile first. One more thing that when we start WPF software we get a slow loading problem and that's the reason why C# runs slowly.

C# - Program Structure

Creating Hello World Program

A C# program consists of the following parts −

- Namespace declaration
- A class
- Class methods
- Class attributes
- A Main method
- Statements and Expressions
- Comments

```
using System;

namespace HelloWorldApplication
{
   class HelloWorld
   {
      static void Main(string[] args)
      {
         /* my first program in C# */
         Console.WriteLine("Hello World");
         Console.ReadKey();
      }
   }
}
```

When this code is compiled and executed, it produces the following result −

```
Hello World
```

Let us look at the various parts of the given program −

- The first line of the program **using System;** - the **using** keyword is used to include the **System** namespace in the program. A program generally has multiple **using** statements.
- The next line has the **namespace** declaration. A **namespace** is a collection of classes. The *HelloWorldApplication* namespace contains the class *HelloWorld*.
- The next line has a **class** declaration, the class *HelloWorld* contains the data and method definitions that your program uses. Classes generally contain multiple methods. Methods define the behavior of the class. However, the *HelloWorld* class has only one method **Main**.
- The next line defines the **Main** method, which is the **entry point** for all C# programs. The **Main** method states what the class does when executed.
- The next line /*...*/ is ignored by the compiler and it is put to add **comments** in the program.
- The Main method specifies its behavior with the statement **Console.WriteLine("Hello World");**

  WriteLine is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.

- The last line **Console.ReadKey();** is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

It is worth to note the following points −

- C# is case sensitive.
- All statements and expression must end with a semicolon (;).
- The program execution starts at the Main method.
- Unlike Java, program file name could be different from the class name.

Compiling and Executing the Program

If you are using Visual Studio.Net for compiling and executing C# programs, take the following steps −

- Start Visual Studio.
- On the menu bar, choose File -> New -> Project.
- Choose Visual C# from templates, and then choose Windows.
- Choose Console Application.
- Specify a name for your project and click OK button.
- This creates a new project in Solution Explorer.
- Write code in the Code Editor.
- Click the Run button or press F5 key to execute the project. A Command Prompt window appears that contains the line Hello World.

You can compile a C# program by using the command-line instead of the Visual Studio IDE −

- Open a text editor and add the above-mentioned code.
- Save the file as **helloworld.cs**
- Open the command prompt tool and go to the directory where you saved the file.
- Type **csc helloworld.cs** and press enter to compile your code.

- If there are no errors in your code, the command prompt takes you to the next line and generates **helloworld.exe** executable file.
- Type **helloworld** to execute your program.
- You can see the output Hello World printed on the screen.

C# Syntax

We created a C# file called Program.cs, and we used the following code to print "Hello World" to the screen:

```
using System;

namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

WriteLine or Write

The most common method to output something in C# is `WriteLine()`, but you can also use `Write()`.

The difference is that `WriteLine()` prints the output on a new line each time, while `Write()` prints on the same line (note that you should remember to add spaces when needed, for better readability):

Example:

```
Console.WriteLine("Hello World!");
Console.WriteLine("I will print on a new line.");

Console.Write("Hello World! ");
Console.Write("I will print on the same line.");
```

Result:

```
Hello World!
I will print on a new line.
Hello World! I will print on the same line.
```

C# Variables

Variables are containers for storing data values.

In C#, there are different **types** of variables (defined with different keywords), for example:

- **`int`** - stores integers (whole numbers), without decimals, such as 123 or -123
- **`double`** - stores floating point numbers, with decimals, such as 19.99 or -19.99
- **`char`** - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- **`string`** - stores text, such as "Hello World". String values are surrounded by double quotes

- **`bool`** - stores values with two states: true or false

Declaring (Creating) Variables

To create a variable, you must specify the type and assign it a value:

Syntax

```
type variableName = value;
```

Where *type* is a C# type (such as `int` or `string`), and *variableName* is the name of the variable (such as **x** or **name**). The **equal sign** is used to assign values to the variable.

To create a variable that should store text, look at the following example:

Example

Create a variable called **name** of type `string` and assign it the value "**John**":

```
string name = "John";
Console.WriteLine(name);
```

To create a variable that should store a number, look at the following example:

Example

Create a variable called **myNum** of type `int` and assign it the value **15**:

```
int myNum = 15;
Console.WriteLine(myNum);
```

You can also declare a variable without assigning the value, and assign the value later:

Example

```
int myNum;
myNum = 15;
Console.WriteLine(myNum);
```

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

Example

Change the value of `myNum` to 20:

```
int myNum = 15;
myNum = 20; // myNum is now 20
Console.WriteLine(myNum);
```

Constants

However, you can add the `const` keyword if you don't want others (or yourself) to overwrite existing values (this will declare the variable as "constant", which means unchangeable and read-only):

Example

```
const int myNum = 15;
myNum = 20; // error
```

Other Types

A demonstration of how to declare variables of other types:

Example

```
int myNum = 5;
double myDoubleNum = 5.99D;
char myLetter = 'D';
bool myBool = true;
string myText = "Hello";
```

Display Variables

The `WriteLine()` method is often used to display variable values to the console window.

To combine both text and a variable, use the `+` character:

Example

```
string name = "John";
Console.WriteLine("Hello " + name);
```

You can also use the `+` character to add a variable to another variable:

Example

```
string firstName = "John ";
string lastName = "Doe";
string fullName = firstName + lastName;
Console.WriteLine(fullName);
```

For numeric values, the `+` character works as a mathematical operator (notice that we use `int` (integer) variables here):

Example

```
int x = 5;
int y = 6;
Console.WriteLine(x + y); // Print the value of x + y
```

From the example above, you can expect:

- x stores the value 5
- y stores the value 6
- Then we use the `WriteLine()` method to display the value of x + y, which is **11**

Declare Many Variables

To declare more than one variable of the **same type**, use a comma-separated list:

Example

```
int x = 5, y = 6, z = 50;
Console.WriteLine(x + y + z);
```

C# Identifiers

All C# **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code:

Example

```
// Good
int minutesPerHour = 60;

// OK, but not so easy to understand what m actually is
int m = 60;
```

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits and the underscore character (_)
- Names must begin with a letter
- Names should start with a lowercase letter and it cannot contain whitespace
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like C# keywords, such as `int` or `double`) cannot be used as names

C# Data Types

A variable in C# must be a specified data type:

Example

```
int myNum = 5;               // Integer (whole number)
double myDoubleNum = 5.99D;  // Floating point number
char myLetter = 'D';         // Character
bool myBool = true;          // Boolean
string myText = "Hello";     // String
```

A data type specifies the size and type of variable values. It is important to use the correct data type for the corresponding variable; to avoid errors, to save time and memory, but it will also make your code more maintainable and readable. The most common data types are:

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
bool	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter, surrounded by single quotes
string	2 bytes per character	Stores a sequence of characters, surrounded by double quotes

Numbers

Number types are divided into two groups:

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are `int` and `long`. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. Valid types are `float` and `double`.

Integer Types

Int

The `int` data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the `int` data type is the preferred data type when we create variables with a numeric value.

Example

```
int myNum = 100000;
Console.WriteLine(myNum);
```

Long

The `long` data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when `int` is not large enough to store the value. Note that you should end the value with an "L":

Example

```
long myNum = 15000000000L;
```

```
Console.WriteLine(myNum);
```

Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

Float

The `float` data type can store fractional numbers from 3.4e−038 to 3.4e+038. Note that you should end the value with an "F":

Example

```
float myNum = 5.75F;
Console.WriteLine(myNum);
```

Double

The `double` data type can store fractional numbers from 1.7e−308 to 1.7e+308. Note that you can end the value with a "D" (although not required):

Example

```
double myNum = 19.99D;
Console.WriteLine(myNum);
```

C# Operators :

Operators are used to perform operations on variables and values.

In the example below, we use the + **operator** to add together two values:

Example

```
int x = 100 + 50;
```

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example

```
int sum1 = 100 + 50;        // 150 (100 + 50)
int sum2 = sum1 + 250;      // 400 (150 + 250)
int sum3 = sum2 + sum2;     // 800 (400 + 400)
```

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations:

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	x++
--	Decrement	Decreases the value of a variable by 1	x--

Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example

```
int x = 10;
```

The **addition assignment** operator (+=) adds a value to a variable:

Example

```
int x = 10;
x += 5;
```

A list of all assignment operators:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Arrays :

Create an Array

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

```
string[] cars;
```

We have now declared a variable that holds an array of strings.

To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in **cars**:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
Console.WriteLine(cars[0]);
// Outputs Volvo
```

Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
Console.WriteLine(cars[0]);
// Now outputs Opel instead of Volvo
```

FOR EXAMPLE:

```
using System;
namespace MyApplication
{
 class Program
 {
  static void Main(string[] args)
  {
   string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
   cars[0] = "Opel";
   Console.WriteLine(cars[0]);
  } } }
```

Result : Opel

Function:

Function is a block of code that has a signature. Function is used to execute statements specified in the code block. A function consists of the following components:

Function name: It is a unique name that is used to make Function call.

Return type: It is used to specify the data type of function return value.

Body: It is a block that contains executable statements.

Access specifier: It is used to specify function accessibility in the application.

Parameters: It is a list of arguments that we can pass to the function during call.

Function Syntax

```
<access-specifier><return-type>FunctionName(<parameters>)
{
    // function body
    // return statement
}
```

Access-specifier, parameters and return statement are optional.

Methods :

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method is defined with the name of the method, followed by parentheses **()**. C# provides some pre-defined methods, which you already are familiar with, such as `Main()`, but you can also create your own methods to perform certain actions:

Example

Create a method inside the Program class:

```
class Program
{
  static void MyMethod()
  {
    // code to be executed
```

```
  }
}
```

Example Explained

- `MyMethod()` is the name of the method
- `static` means that the method belongs to the Program class and not an object of the Program class. You will learn more about objects and how to access methods through objects later in this tutorial.
- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

Note: In C#, it is good practice to start with an uppercase letter when naming methods, as it makes the code easier to read.

Call a Method:

To call (execute) a method, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, `MyMethod()` is used to print a text (the action), when it is called:

Example

Inside `Main()`, call the `myMethod()` method:

```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
  MyMethod();
}

// Outputs "I just got executed!"
```

A method can be called multiple times:

Example

```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
  MyMethod();
  MyMethod();
  MyMethod();
}
```

Method Parameters:

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `string` called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example

```
static void MyMethod(string fname)
{
  Console.WriteLine(fname + " Refsnes");
}

static void Main(string[] args)
{
  MyMethod("Liam");
  MyMethod("Jenny");
  MyMethod("Anja");
}
```

Default Parameter Value

You can also use a default parameter value, by using the equals sign (=). If we call the method without an argument, it uses the default value ("Norway"):

Example

```
static void MyMethod(string country = "Norway")
{
  Console.WriteLine(country);
}

static void Main(string[] args)
{
  MyMethod("Sweden");
  MyMethod("India");
  MyMethod();
  MyMethod("USA");
}
```

Multiple Parameters

You can have as many parameters as you like:

Example

```
static void MyMethod(string fname, int age)
```

```
{
  Console.WriteLine(fname + " is " + age);
}

static void Main(string[] args)
{
  MyMethod("Liam", 5);
  MyMethod("Jenny", 8);
  MyMethod("Anja", 31);
}
```

Return Values

The `void` keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as `int` or `double`) instead of `void`, and use the `return` keyword inside the method:

Example

```
static int MyMethod(int x)
{
  return 5 + x;
}

static void Main(string[] args)
{
  Console.WriteLine(MyMethod(3));
}
```

This example returns the sum of a method's **two parameters**:

Example

```
static int MyMethod(int x, int y)
{
  return x + y;
}

static void Main(string[] args)
{
  Console.WriteLine(MyMethod(5, 3));
}
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

Example

```
static int MyMethod(int x, int y)
{
  return x + y;
}

static void Main(string[] args)
{
  int z = MyMethod(5, 3);
  Console.WriteLine(z);
}
```

Named Arguments

It is also possible to send arguments with the *key: value* syntax.

That way, the order of the arguments does not matter:

Example

```
static void MyMethod(string child1, string child2, string child3)
{
  Console.WriteLine("The youngest child is: " + child3);
}

static void Main(string[] args)
{
  MyMethod(child3: "John", child1: "Liam", child2: "Liam");
}
```

Named arguments are especially useful when you have multiple parameters with default values, and you only want to specify one of them when you call it:

Example

```
static void MyMethod(string child1 = "Liam", string child2 = "Jenny", string child3
= "John")
{
  Console.WriteLine(child3);
}

static void Main(string[] args)
{
  MyMethod("child3");
}
```

Control Statement :

If ... Else:

Conditions and If Statements

C# supports the usual logical conditions from mathematics:

- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- Equal to a == b
- Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

C# has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

The if Statement

Use the `if` statement to specify a block of C# code to be executed if a condition is `True`.

Syntax

```
if (condition)
{
  // block of code to be executed if the condition is True
}
```

Note that `if` is in lowercase letters. Uppercase letters (If or IF) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is `True`, print some text:

Example

```
if (20 > 18)
{
  Console.WriteLine("20 is greater than 18");
}
```

We can also test variables:

Example

```
int x = 20;
int y = 18;
if (x > y)
{
  Console.WriteLine("x is greater than y");
}
```

Example explained

In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the `>` operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is `False`.

Syntax

```
if (condition)
```

```
{
  // block of code to be executed if the condition is True
}
else
{
  // block of code to be executed if the condition is False
}
```

Example

```
int time = 20;
if (time < 18)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

The else if Statement

Use the `else if` statement to specify a new condition if the first condition is `False`.

Syntax

```
if (condition1)
{
  // block of code to be executed if condition1 is True
}
else if (condition2)
{
  // block of code to be executed if the condition1 is false and condition2 is True
}
else
{
  // block of code to be executed if the condition1 is false and condition2 is
False
}
```

Example

```
int time = 22;
if (time < 10)
{
  Console.WriteLine("Good morning.");
}
else if (time < 20)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

Short Hand If...Else (Ternary Operator)

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

Syntax

```
variable = (condition) ? expressionTrue :  expressionFalse;
```

Example

```
int time = 20;
if (time < 18)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
```

You can simply write:

Example

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
Console.WriteLine(result);
```

Switch Statements

Use the `switch` statement to select one of many code blocks to be executed.

Syntax

```
switch(expression)
{
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
    break;
}
```

This is how it works:

- The `switch` expression is evaluated once
- The value of the expression is compared with the values of each `case`
- If there is a match, the associated block of code is executed

- The `break` and `default` keywords will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;
switch (day)
{
  case 1:
    Console.WriteLine("Monday");
    break;
  case 2:
    Console.WriteLine("Tuesday");
    break;
  case 3:
    Console.WriteLine("Wednesday");
    break;
  case 4:
    Console.WriteLine("Thursday");
    break;
  case 5:
    Console.WriteLine("Friday");
    break;
  case 6:
    Console.WriteLine("Saturday");
    break;
  case 7:
    Console.WriteLine("Sunday");
    break;
}
// Outputs "Thursday" (day 4)
```

The break Keyword

When C# reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

The default Keyword

The `default` keyword is optional and specifies some code to run if there is no case match:

Example

```
int day = 4;
switch (day)
{
  case 6:
    Console.WriteLine("Today is Saturday.");
    break;
  case 7:
    Console.WriteLine("Today is Sunday.");
    break;
  default:
```

```
    Console.WriteLine("Looking forward to the Weekend.");
    break;
}
// Outputs "Looking forward to the Weekend."
```

Looping Statement :

While Loop

Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

While Loop

The `while` loop loops through a block of code as long as a specified condition is `True`:

Syntax

```
while (condition)
{
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
int i = 0;
while (i < 5)
{
  Console.WriteLine(i);
  i++;
}
```

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

The Do/While Loop

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do
{
  // code block to be executed
}
while (condition);
```

The example below uses a `do/while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
int i = 0;
do
{
  Console.WriteLine(i);
  i++;
}
while (i < 5);
```

For Loop

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:

Syntax

```
for (statement 1; statement 2; statement 3)
{
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example

```
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

Example explained

Statement 1 sets a variable before the loop starts (`int i = 0`).

Statement 2 defines the condition for the loop to run (`i` must be less than `5`). If the condition is `true`, the loop will start over again, if it is `false`, the loop will end.

Statement 3 increases a value (`i++`) each time the code block in the loop has been executed.

Another Example

This example will only print even values between 0 and 10:

Example

```
for (int i = 0; i <= 10; i = i + 2)
{
  Console.WriteLine(i);
}
```

The foreach Loop

There is also a `foreach` loop, which is used exclusively to loop through elements in an **array**:

Syntax

```
foreach (type variableName in arrayName)
{
  // code block to be executed
}
```

The following example outputs all elements in the **cars** array, using a `foreach` loop:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
foreach (string i in cars)
{
  Console.WriteLine(i);
}
```

Break

You have already seen the `break` statement used in an earlier chapter of this tutorial. It was used to "jump out" of a `switch` statement.

The `break` statement can also be used to jump out of a **loop**.

This example jumps out of the loop when `i` is equal to `4`:

Example

```
for (int i = 0; i < 10; i++)
{
  if (i == 4)
  {
    break;
  }
  Console.WriteLine(i);
}
```

Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of `4`:

Example

```
for (int i = 0; i < 10; i++)
{
  if (i == 4)
  {
    continue;
  }
  Console.WriteLine(i);
}
```

Break and Continue in While Loop

You can also use `break` and `continue` in while loops:

Break Example

```
int i = 0;
while (i < 10)
{
  Console.WriteLine(i);
  i++;
  if (i == 4)
  {
    break;
  }
}
```

Continue Example

```
int i = 0;
while (i < 10)
{
  if (i == 4)
  {
    i++;
    continue;
  }
  Console.WriteLine(i);
  i++;
}
```