

C# Programming Languages

3nd Class

Introducer. Waleed Kareem Awad

Computer Science Department

College of CSIT

University of Anbar

2021-2022

C# Introduction

What is C#?

C# is pronounced "C-Sharp".

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.

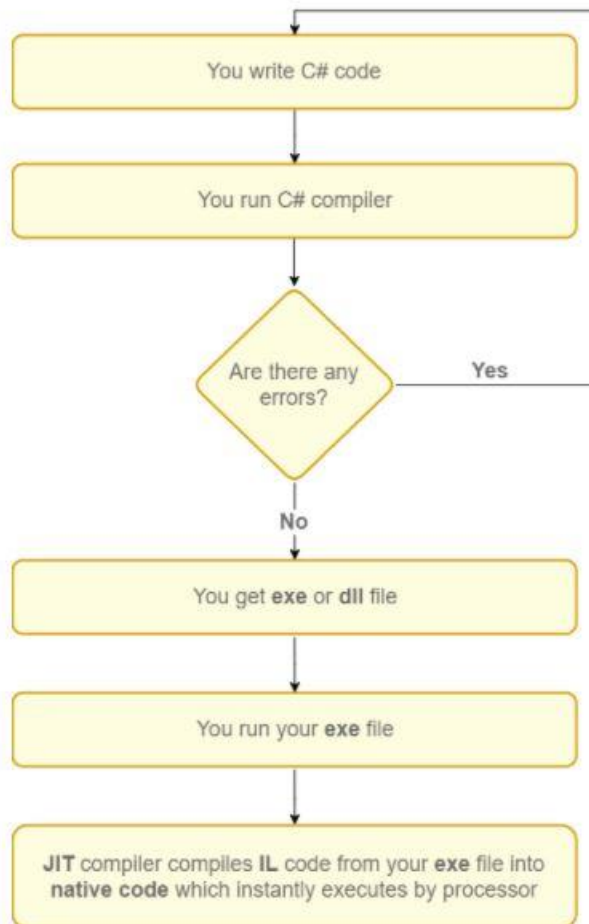
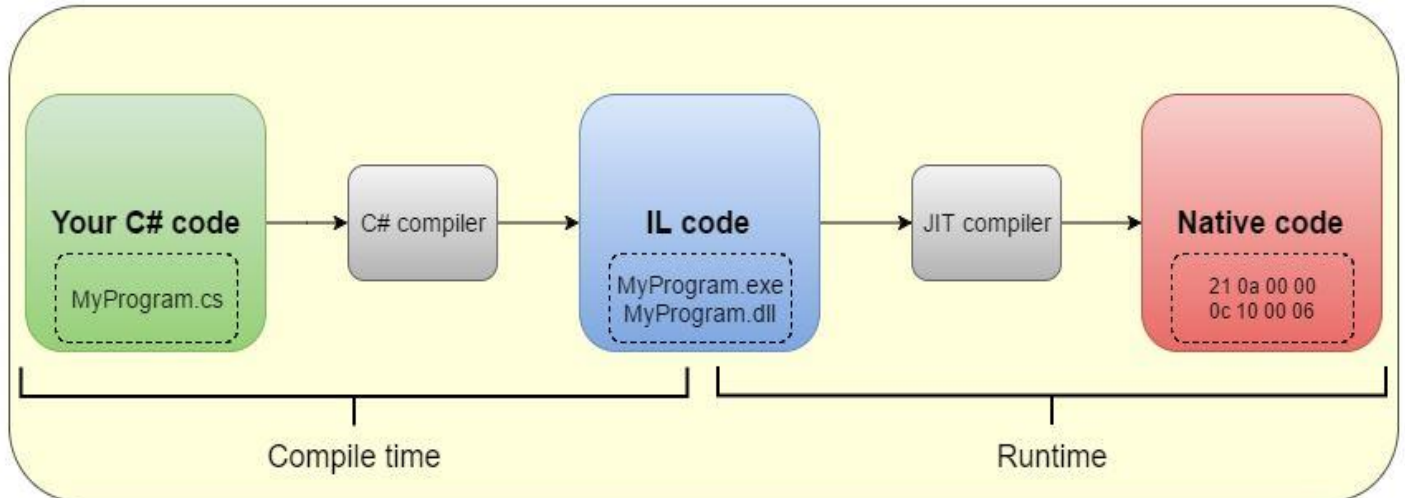
C# is used for:

- Mobile applications
- Desktop applications
- Web applications
- Web services
- Web sites
- Games
- Database applications
- And much, much more!

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

How C# Program Work



Introducer: waleed K. awad

Email: waleed.kareem@uoanbar.edu.iq

Introduction To C#

C# Structures

```
using System;

namespace MyApplication
{
    class First Program
    {
        static void Main(tring[] args)
        {
            يكتب الكود هنا//
        }
    }
}
```

C# Variables

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
```

The const keyword is useful when you want a variable to always store the same value, so that others (or yourself) won't mess up your code. An example that is often referred to as a constant, is PI (3.14159...).

Note: You cannot declare a constant variable without assigning the value. If you do, an error will occur: A const field requires a value to be provided.

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";
```

You will learn more about [data types](#) in the next chapter.

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

C# Data Types

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:

1. **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.
2. **Floating point types** represents numbers with a fractional part, containing one or more decimals. Valid types are **float** and **double**.

- **Booleans**

A boolean data type is declared with the bool keyword and can only take the values true or false

```
bool isCSharpFun = true;  
bool isFishTasty = false;  
Console.WriteLine(isCSharpFun); // Outputs True  
Console.WriteLine(isFishTasty); // Outputs False
```

- **Characters**

The char data type is used to store a single character. The character must be surrounded by single quotes, like 'A' or 'c'.

- **Strings**

The string data type is used to store a sequence of characters (text). String values must be surrounded by double quotes

Example of all Types

```
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
```

C# Type Casting

C# Type Casting

Type casting is when you assign a value of one data type to another type.

In C#, there are two types of casting:

- **Implicit Casting** (automatically) - converting a smaller type to a larger type size
`char -> int -> long -> float -> double`
- **Explicit Casting** (manually) - converting a larger type to a smaller size type
`double -> float -> long -> int -> char`

Implicit Casting

Implicit casting is done automatically when passing a smaller size type to a larger size type:

Example

```
int myInt = 9;  
  
double myDouble = myInt;    // Automatic casting: int to double  
  
Console.WriteLine(myInt);  
  
Console.WriteLine(myDouble);
```

Explicit Casting

Explicit casting must be done manually by placing the type in parentheses in front of the value:

Example

```
double myDouble = 9.78;  
  
int myInt = (int) myDouble;    // Manual casting: double to int  
  
Console.WriteLine(myDouble);    // Outputs 9.78  
  
Console.WriteLine(myInt);       // Outputs 9
```

Type Conversion Methods

It is possible to convert data types explicitly by using built-in methods, such as `Convert.ToBoolean`, `Convert.ToDouble`, `Convert.ToString`, `Convert.ToInt32` (int) and `Convert.ToInt64` (long):

Example

```
int myInt = 10;

double myDouble = 5.25;

bool myBool = true;

Console.WriteLine(Convert.ToString(myInt)); // convert int to
string

Console.WriteLine(Convert.ToDouble(myInt)); // convert int to
double

Console.WriteLine(Convert.ToInt32(myDouble)); // convert double
to int

Console.WriteLine(Convert.ToString(myBool)); // convert bool to
string
```

C# User Input

Get User Input

You have already learned that `Console.WriteLine()` is used to output (print) values. Now we will use `Console.ReadLine()` to get user input.

In the following example, the user can input his or hers username, which is stored in the variable `userName`. Then we print the value of `userName`:

Example

```
// Type your username and press enter
Console.WriteLine("Enter username:");

// Create a string variable and get user input from the keyboard and
// store it in the variable
string userName = Console.ReadLine();

// Print the value of the variable (userName), which will display the
// input value
Console.WriteLine("Username is: " + userName);
```

User Input and Numbers

The `Console.ReadLine()` method returns a `string`. Therefore, you cannot get information from another data type, such as `int`. The following program will cause an error:

Example


```
Console.WriteLine("Enter your age:");  
  
int age = Console.ReadLine();  
  
Console.WriteLine("Your age is: " + age);
```

ERROR MESSAGE

The error message will be something like this: Cannot implicitly convert type 'string' to 'int'

Like the error message says, you cannot implicitly convert type 'string' to 'int'.

Example

```
Console.WriteLine("Enter your age:");  
  
int age = Convert.ToInt32(Console.ReadLine());  
  
Console.WriteLine("Your age is: " + age);
```

Note: If you enter wrong input (e.g. text in a numerical input), you will get an exception/error message (like System.FormatException: 'Input string was not in a correct format.').

C# Operators

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 = 10 + 20;
```

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	x % y

remainder			
++	Increment	Increases the value of a variable by 1	x++
--	Decrement	Decreases the value of a variable by 1	x--

C# 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

C# 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

C# 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	<code>x < 5 && x < 10</code>
	Logical or	Returns true if one of the statements is true	<code>x < 5 x < 4</code>
!	Logical not	Reverse the result, returns false if the result is true	<code>!(x < 5 && x < 10)</code>

C# Operators

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 = 10 + 20;
```

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	$x--$

C# 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

C# 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

C# 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)

C# Conditions and If Statements

C# supports the usual logical conditions from mathematics:

- Less than: $a < b$
- Less than or equal to: $a \leq b$
- Greater than: $a > b$
- Greater than or equal to: $a \geq b$
- 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**.

```
int x = 20;  
int y = 18;  
if (x > y)  
{ Console.WriteLine("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**.

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

The else if Statement

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

```
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."); }
```

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;

Instead of writing:

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);
```

C# Switch

C# 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 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;
}
```

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.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

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;
}
```

C# Iterations Statements

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.

- **C# 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);
}
```


• 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);
}
```

• C# 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++; }
```

• 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);
```

C# Break and Continue

C# Break

The break statement to "jump out" of a switch statement. It 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);  
}
```

C# 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

Introducer: waleed K. awad

Email: waleed.kareem@uoanbar.edu.iq

```
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)
```

Introducer: waleed K. awad

Email: waleed.kareem@uoanbar.edu.iq

```
{  
    i++;  
    continue;  
}  
Console.WriteLine(i);  
i++;  
}
```

C# Methods

C# Math

The C# Math class has many methods that allows you to perform mathematical tasks on numbers You call any static method by specifying the name of the class in which the method is declared, followed by the member access (.) operator and the method name.

ClassName.MethodName(argument);

Method	Description	Example
Abs(<i>x</i>)	absolute value of <i>x</i>	Abs(23.7) is 23.7 Abs(0.0) is 0.0 Abs(-23.7) is 23.7
Ceiling(<i>x</i>)	rounds <i>x</i> to the smallest integer not less than <i>x</i>	Ceiling(9.2) is 10.0 Ceiling(-9.8) is -9.0
Cos(<i>x</i>)	trigonometric cosine of <i>x</i> (<i>x</i> in radians)	Cos(0.0) is 1.0
Exp(<i>x</i>)	exponential method e^x	Exp(1.0) is 2.71828 Exp(2.0) is 7.38906
Floor(<i>x</i>)	rounds <i>x</i> to the largest integer not greater than <i>x</i>	Floor(9.2) is 9.0 Floor(-9.8) is -10.0
Log(<i>x</i>)	natural logarithm of <i>x</i> (base e)	Log(Math.E) is 1.0 Log(Math.E * Math.E) is 2.0
Max(<i>x</i> , <i>y</i>)	larger value of <i>x</i> and <i>y</i>	Max(2.3, 12.7) is 12.7 Max(-2.3, -12.7) is -2.3
Min(<i>x</i> , <i>y</i>)	smaller value of <i>x</i> and <i>y</i>	Min(2.3, 12.7) is 2.3 Min(-2.3, -12.7) is -12.7
Pow(<i>x</i> , <i>y</i>)	<i>x</i> raised to the power <i>y</i> (i.e., x^y)	Pow(2.0, 7.0) is 128.0 Pow(9.0, 0.5) is 3.0
Sin(<i>x</i>)	trigonometric sine of <i>x</i> (<i>x</i> in radians)	Sin(0.0) is 0.0
Sqrt(<i>x</i>)	square root of <i>x</i>	Sqrt(900.0) is 30.0
Tan(<i>x</i>)	trigonometric tangent of <i>x</i> (<i>x</i> in radians)	Tan(0.0) is 0.0

Math.Max(x,y)

The **Math.Max(x,y)** method can be used to find the highest value of x and y :

Example

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(Math.Max(5, 10));
    }
}
```

Math.Min(x,y)

The **Math.Min(x,y)** method can be used to find the lowest value of x and y :

Example

```
Math.Min(5, 10)
```

Math.Sqrt(x)

The **Math.Sqrt(x)** method returns the square root of x :

Example

```
Math.Sqrt(64);
```

C# 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
    }
}
```

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!"
```

Note: 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();
}

// I just got executed!
// I just got executed!
// I just got executed!
```

C# 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");
}

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

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);
}
// Liam is 5
// Jenny is 8
// Anja is 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));
}
```

```
// Outputs 8 (5 + 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));
}

// Outputs 8 (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);
}

// Outputs 8 (5 + 3)
```

C# 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]);
```

Array Length

To find out how many elements an array has, use the **Length** property:

Example

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

Console.WriteLine(cars.Length);

// Outputs 4
```

Loop Through an Array

You can loop through the array elements with the **for** loop, and use the **Length** property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.Length; i++)
{
    Console.WriteLine(cars[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 1

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

The example above can be read like this: **for each** **string** element (called **i** - as in **index**) in **cars**, print out the value of **i**.

If you compare the **for** loop and **foreach** loop, you will see that the **foreach** method is easier to write, it does not require a counter (using the **Length** property), and it is more readable.

Example 2

```
public static void Main( string[] args )
{
    int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
    int total = 0;

    // add each element's value to total
    foreach ( int number in array )
        total += number;

    Console.WriteLine( "Total of array elements: {0}", total );
} // end Main
} // end class ForEachTest
```

C# Arrays

Sort Arrays

There are many array methods available, for example **Sort()**, which sorts an array alphabetically or in an ascending order:

Example

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

// Sort an int
int[] myNumbers = {5, 1, 8, 9};
Array.Sort(myNumbers);
foreach (int i in myNumbers)
{
    Console.WriteLine(i);
}
```

System.Linq Namespace

Other useful array methods, such as **Min**, **Max**, and **Sum**, can be found in the **System.Linq** namespace:

Example

```
using System;
using System.Linq;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
```

```
{
    int[] myNumbers = {5, 1, 8, 9};
    Console.WriteLine(myNumbers.Max()); // returns the
largest value
    Console.WriteLine(myNumbers.Min()); // returns the
smallest value
    Console.WriteLine(myNumbers.Sum()); // returns the sum
of elements
} } }
```

Other Ways to Create an Array

If you are familiar with C#, you might have seen arrays created with the **new** keyword, and perhaps you have seen arrays with a specified size as well. In C#, there are different ways to create an array:

```
// Create an array of four elements, and add values later
string[] cars = new string[4];

// Create an array of four elements and add values right
away
string[] cars = new string[4] {"Volvo", "BMW", "Ford",
"Mazda"};

// Create an array of four elements without specifying the
size
string[] cars = new string[] {"Volvo", "BMW", "Ford",
"Mazda"};

// Create an array of four elements, omitting the new
keyword, and without specifying the size
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

An Example of Array

```
using System;

public class InitArray
{
    public static void Main( string[] args )
    {
        // initializer list specifies the value for each element
        int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

        Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // headings

        // output each array element's value
        for ( int counter = 0; counter < array.Length; counter++ )
            Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
    }
}
```

Another Example

```
using System;

public class SumArray
{
    public static void Main( string[] args )
    {
        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
        int total = 0;

        // add each element's value to total
        for ( int counter = 0; counter < array.Length; counter++ )
            total += array[ counter ];

        Console.WriteLine( "Total of array elements: {0}", total );
    }
}
```

Output///// Total of array elements: 849

C# Strings

C# Strings

A **string** is an object of type String whose value is text. Internally, the text is stored as a sequential read-only collection of Char objects.

A **string** variable contains a collection of characters surrounded by double quotes:

Example

Create a variable of type string and assign it a value:

```
string greeting = "Hello";
```

String Length

A string in C# is actually an object, which contain properties and methods that can perform certain operations on strings. For example, the length of a string can be found with the **Length** property:

```
string txt = "ABCDEFGHJKLMNOPQRSTUVWXYZ";  
  
Console.WriteLine("The length of the txt string is: " +  
txt.Length);
```

Other Methods

There are many string methods available, for example **ToUpper()** and **ToLower()**, which returns a copy of the string converted to uppercase or lowercase:

```
string txt = "Hello World";  
Console.WriteLine(txt.ToUpper()); // Outputs "HELLO WORLD"  
Console.WriteLine(txt.ToLower()); // Outputs "hello world"
```

String Concatenation

The `+` operator can be used between strings to combine them. This is called **concatenation**:

Example

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

Note that we have added a space after "John" to create a space between `firstName` and `lastName` on print.

You can also use the `string.Concat()` method to concatenate two strings:

Example

```
string firstName = "John ";  
string lastName = "Doe";  
string name = string.Concat(firstName, lastName);  
Console.WriteLine(name);
```

Access Strings

You can access the characters in a string by referring to its index number inside square brackets `[]`.

This example prints the **first character** in **myString**:

Example

```
string myString = "Hello";  
Console.WriteLine(myString[0]); // Outputs "H"
```

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

This example prints the **second character** (1) in **myString**:

Example

```
string myString = "Hello";  
Console.WriteLine(myString[1]); // Outputs "e"
```

You can also find the index position of a specific character in a string, by using the **IndexOf()** method:

Example

```
string myString = "Hello";  
Console.WriteLine(myString.IndexOf("e")); // Outputs "1"
```

Another useful method is **Substring()**, which extracts the characters from a string, starting from the specified character position/index, and returns a new string. This method is often used together with **IndexOf()** to get the specific character position:

Example

```
// Full name
string name = "Waleed Kareem";

// Location of the letter D
int charPos = name.IndexOf("K");

// Get last name
string lastName = name.Substring(charPos);

// Print the result
Console.WriteLine(lastName);
```

Special Characters

Because strings must be written within quotes, C# will misunderstand this string, and generate an error:

Example

```
string txt = "We are the so-called "Vikings" from the north.";
```


The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

The sequence \" inserts a double quote in a string:

Example

```
string txt = "We are the so-called \"Vikings\" from the north.";
```

The sequence \' inserts a single quote in a string:

Example

```
string txt = "It\'s alright."; //output: It's alright
```

The sequence `\\` inserts a single backslash in a string:

```
string txt = "The character \\ is called backslash.";
```

Other useful escape characters in C# are:

Code	Result
<code>\n</code>	New Line
<code>\t</code>	Tab
<code>\b</code>	Backspace

Adding Numbers and Strings

If you add two numbers, the result will be a number:

Example

```
int x = 10;  
int y = 20;  
int z = x + y; // z will be 30 (an integer/number)
```

WARNING!

C# uses the + operator for both addition and concatenation.

Remember: Numbers are added. Strings are concatenated.

If you add two strings, the result will be a string concatenation:

Example

```
string x = "10";  
string y = "20";  
string z = x + y; // z will be 1020 (a string)
```

C# Classes

C# OOP

What is OOP?

OOP stands for Object-Oriented Programming. Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

❖ Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

What are Classes and Objects?

- Classes and objects are the two main aspects of object-oriented programming.
- Look at the following illustration to see the difference between class and objects:



Another example:



- When the individual objects are created, they inherit all the variables and methods from the class.

C# Classes and Objects

You learned from the previous chapter that C# is an object-oriented programming language.

Everything in C# is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the **class** keyword:

Create a class named "Car" with a variable **color**:

```
class Car
{
    string color = "red";
}
```

Create an Object

An object is created from a class. We have already created the class named **Car**, so now we can use this to create objects.

To create an object of **Car**, specify the class name, followed by the object name, and use the keyword **new**:

Example

Create an object called "myObj" and use it to print the value of **color**:

```
class Car
{
    string color = "red";
}
```

```
static void Main(string[] args)
{
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
}
```

Note that we use the dot syntax (.) to access variables/fields inside a class (**myObj.color**).

Note that Data in class (represented by fields), and Behavior (represented by methods/functions).

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of **Car**:

```
class Car
{
    string color = "red";
    static void Main(string[] args)
    {
        Car myObj1 = new Car();
        Car myObj2 = new Car();
        Console.WriteLine(myObj1.color);
        Console.WriteLine(myObj2.color);
    }
}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the fields and methods, while the other class holds the **Main()** method (code to be executed)).

- prog2.cs
- prog.cs

prog2.cs

```
class Car
{
    public string color = "red";
}
```

prog.cs

```
class Program
{
    static void Main(string[] args)
    {
        Car myObj = new Car();
        Console.WriteLine(myObj.color);
    }
}
```

Did you notice the **public** keyword? It is called an access modifier, which specifies that the **color** variable/field of **Car** is accessible for other classes as well, such as **Program**.

C# Class Members

Fields and methods inside classes are often referred to as "Class Members". Class Members consist of two parts:

- Instance: accessible from an object.

```
person person = new person();
person.Intorduce();
```

- Static: accessible from class.

```
Console.WriteLine(" ");
```

Example

Create a **Car** class with three class members: two fields **and** one method.

```
class MyClass
{
    // Class members
    string color = "red";           // field
    int maxSpeed = 200;             // field
    public void fullThrottle()     // method
    {
        Console.WriteLine("The car is going as fast as it can!");
    }
}
```

Fields

You learned that variables inside a class are called fields, and that you can access them by creating an object of the class, and by using the dot syntax (.).

The following example will create an object of the **Car** class, with the name **myObj**. Then we print the value of the fields **color** and **maxSpeed**:

Example

```
class Car
{
    string color = "red";
    int maxSpeed = 200;

    static void Main(string[] args)
    {
        Car myObj = new Car();
        Console.WriteLine(myObj.color);
        Console.WriteLine(myObj.maxSpeed);
    }
}
```

➤ You can also leave the fields blank, and modify them when creating the object:

Example

```
class Car
{
    string color;

    int maxSpeed;

    static void Main(string[] args)
    {
        Car myObj = new Car();
        myObj.color = "red";
        myObj.maxSpeed = 200;
        Console.WriteLine(myObj.color);
        Console.WriteLine(myObj.maxSpeed);
    }
}
```

➤ This is especially useful when creating multiple objects of one class:

Example

```
class Car
{
    string model;
    string color;
    int year;

    static void Main(string[] args)
    {
        Car Ford = new Car();
        Ford.model = "Mustang";
        Ford.color = "red";
        Ford.year = 1969;

        Car Opel = new Car();
        Opel.model = "Astra";
        Opel.color = "white";
        Opel.year = 2005;

        Console.WriteLine(Ford.model);
        Console.WriteLine(Opel.model);
    }
}
```

C# Constructors

A **constructor** is a **special method** that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

Example

Create a constructor:

```
// Create a Car class
class Car
{
    public string model; // Create a field

    // Create a class constructor for the Car class
    public Car() //without parameters
    {
        model = "Mustang"; // Set the initial value for model
    }

    static void Main(string[] args)
    {
        Car Ford = new Car(); // Create an object of the Car
        Class (this will call the constructor)
        Console.WriteLine(Ford.model); // Print the value of
        model
    }
}

// Outputs "Mustang"
```

Note that the constructor name must match the class name, and it cannot have a return type (like void or int).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, C# creates one for you. However, then you are not able to set initial values for fields.

Constructor Parameters

- Constructors can also take parameters, which is used to initialize fields.
- The following example adds a **string modelName** parameter to the constructor. Inside the constructor we set **model** to **modelName** (**model=modelName**). When we call the constructor, we pass a parameter to the constructor ("**Mustang**"), which will set the value of **model** to "**Mustang**":

Example

```
class Car
{
    public string model;

    // Create a class constructor with a parameter
    public Car(string modelName)
    {
        model = modelName;
    }

    static void Main(string[] args)
    {
        Car Ford = new Car("Mustang");
        Console.WriteLine(Ford.model);
    }
}
// Outputs "Mustang"
```

You can have as many parameters as you want:

Example

```
class Car
{
    public string model;
    public string color;
    public int year;

    // Create a class constructor with multiple parameters
    public Car(string modelName, string modelColor, int
modelYear)
```

```
{
    model = modelName;
    color = modelColor;
    year = modelYear;
}

static void Main(string[] args)
{
    Car Ford = new Car("Mustang", "Red", 1969);
    Console.WriteLine(Ford.color + " " + Ford.year + " " +
Ford.model);
}
}
// Outputs Red 1969 Mustang
```

C# Access Modifiers

Access Modifiers By now, you are quite familiar with the **public** keyword that appears in many of our examples:

```
public string color;
```

The **public** keyword is an **access modifier**, which is used to set the access level/visibility for classes, fields, methods and properties.

➤ **C# has the following access modifiers:**

Modifier	Description
public	The code is accessible for all classes
private	The code is only accessible within the same class
protected	The code is accessible within the same class, or in a class that is inherited from that class.
internal	The code is only accessible within its own assembly, but not from another assembly.

➤ **For now, lets focus on **public** and **private** modifiers.**

Public Modifier

If you declare a field with a **public** access modifier, it is accessible for all classes.

Private Modifier

If you declare a field with a **private** access modifier, it can only be accessed within the same class:

Example

```
class Car
{
    private string model = "Mustang";

    static void Main(string[] args)
    {
        Car myObj = new Car();
        Console.WriteLine(myObj.model);
    }
}
```

Note: By default, all members of a class are private if you don't specify an access modifier:

C# Properties (Get and Set)

Properties and Encapsulation

Before we start to explain properties, you should have a basic understanding of "**Encapsulation**".

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare fields/variables as **private**
- provide **public get** and **set** methods, through **properties**, to access and update the value of a **private** field

Properties

You learned from the previous chapter that **private** variables can only be accessed within the same class (an outside class has no access to it). However, sometimes we need to access them - and it can be done with properties.

A property is like a combination of a variable and a method, and it has two methods: a **get** and a **set** method:

Example

```
class Person
{
    private string name; // field
    public string Name    // property
    {
        get { return name; }
        set { name = value; }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person myObj = new Person();
        myObj.Name = "Liam";
        Console.WriteLine(myObj.Name);
    }
}
```

Why Encapsulation?

- Better control of class members (reduce the possibility of yourself (or others) to mess up the code)
- Fields can be made **read-only** (if you only use the **get** method), or **write-only** (if you only use the **set** method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data.