THE NEW COLLEGE (AUTONOMOUS)

Sponsored by: The Muslim Educational Association of Southern India

(Affiliated to the Madras University & Re-Accredited by NAAC with 'A' Grade)

Chennai-600014

PROGRAMMING IN C#

Subject Code: 17BRM513

Semester-V

Date: 15-07-2018

Batch: 2018-2019

Prepared By

Prof. F. MOHAMED ILYAS

Department of Information Systems Management, Shift-II

The New College (Autonomous)

Chennai-600014

PROGRAMMING IN C#

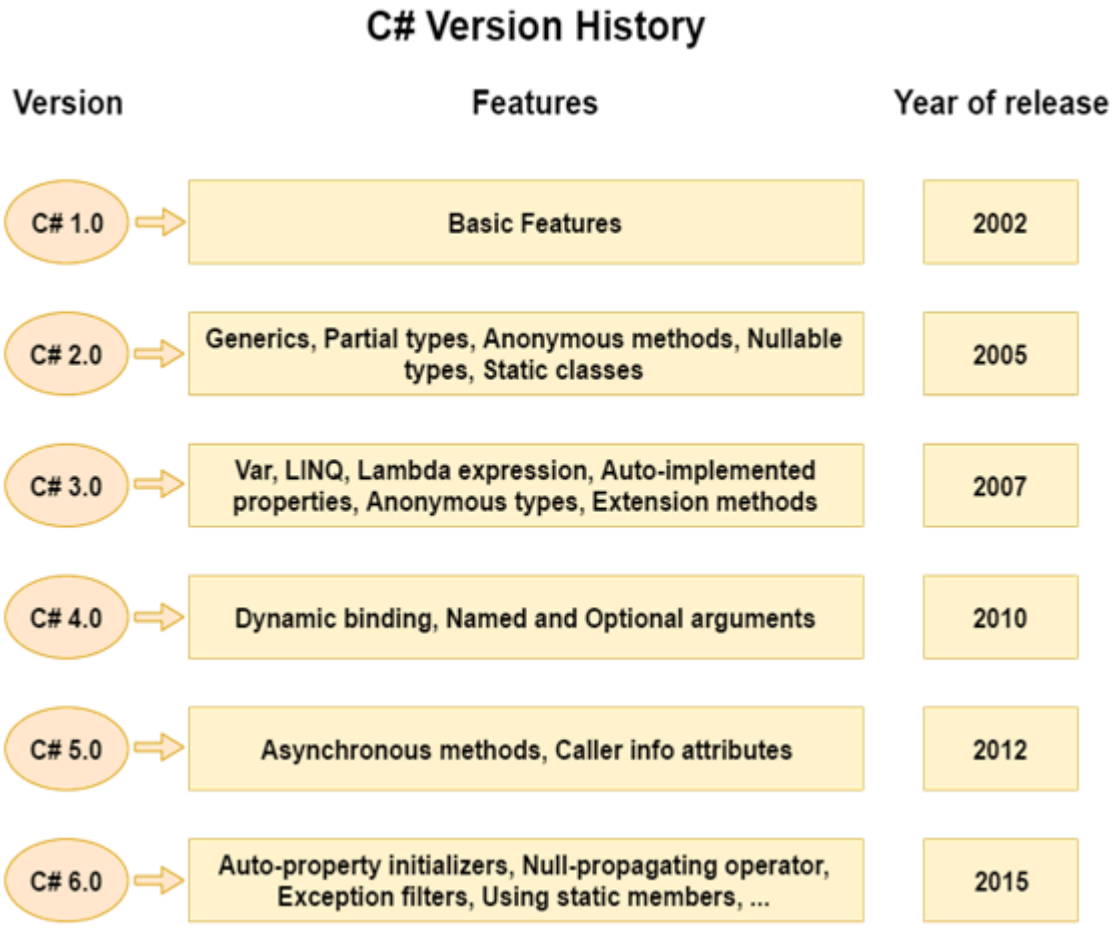
UNIT-I

Introduction of C# Programming Language

C# History

C# is pronounced as **"C-Sharp"**. It is an object-oriented programming language provided by **Microsoft** that runs on **.Net Framework.**

Anders Hejlsberg is known as the **founder of C# language.**



What is C#

C# is pronounced as "C-Sharp". It is an object-oriented programming language provided by Microsoft that runs on .Net Framework.

By the help of C# programming language, we can develop different types of secured and robust applications:

- Window applications
- Web applications
- Distributed applications
- Web service applications
- Database applications etc.

C# is approved as a standard by European Computer Manufacturers Association (**ECMA**) and ISO. C# is designed for CLI (Common Language Infrastructure). CLI is a specification that describes executable code and runtime environment.

C# programming language is influenced by C++, Java, Eiffel, Modula-3, Pascal etc. languages.

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

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

C# IDE

The easiest way to get started with C#, is to use an IDE.

An IDE (Integrated Development Environment) is used to edit and compile code.

In our tutorial, we will use Visual Studio Community, which is free to download from https://visualstudio.microsoft.com/vs/community/.

Applications written in C# use the .NET Framework, so it makes sense to use Visual Studio, as the program, the framework, and the language, are all created by Microsoft.

C# Features

C# is object oriented programming language. It provides a lot of **features** that are given below.

1. Simple
2. Modern programming language
3. Object oriented
4. Type safe
5. Interoperability
6. Scalable and Updateable
7. Component oriented
8. Structured programming language
9. Rich Library
10. Fast speed

1) Simple

C# is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Modern Programming Language

C# programming is based upon the current trend and it is very powerful and simple for building scalable, interoperable and robust applications.

3) Object Oriented

C# is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grow.

4) Type Safe

C# type safe code can only access the memory location that it has permission to execute. Therefore it improves a security of the program.

5) Interoperability

Interoperability process enables the C# programs to do almost anything that a native C++ application can do.

6) Scalable and Updateable

C# is automatic scalable and updateable programming language. For updating our application we delete the old files and update them with new ones.

7) Component Oriented

C# is component oriented programming language. It is the predominant software development methodology used to develop more robust and highly scalable applications.

8) Structured Programming Language

C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

9) Rich Library

C# provides a lot of inbuilt functions that makes the development fast.

10) Fast Speed

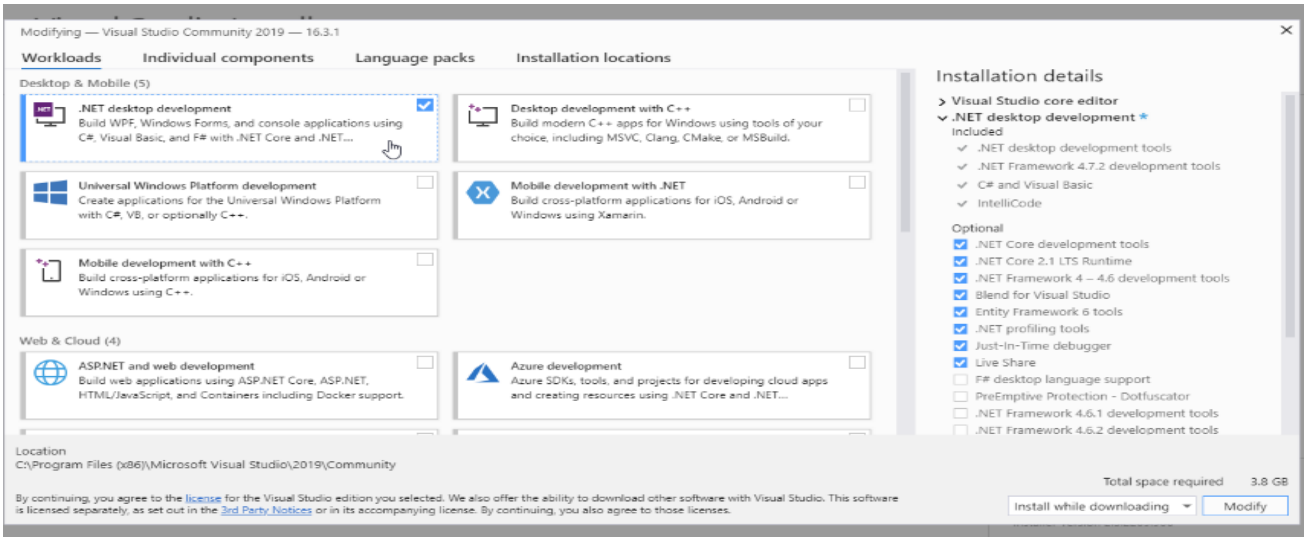
The compilation and execution time of C# language is fast.

Applications of C#: (or) C# used for:

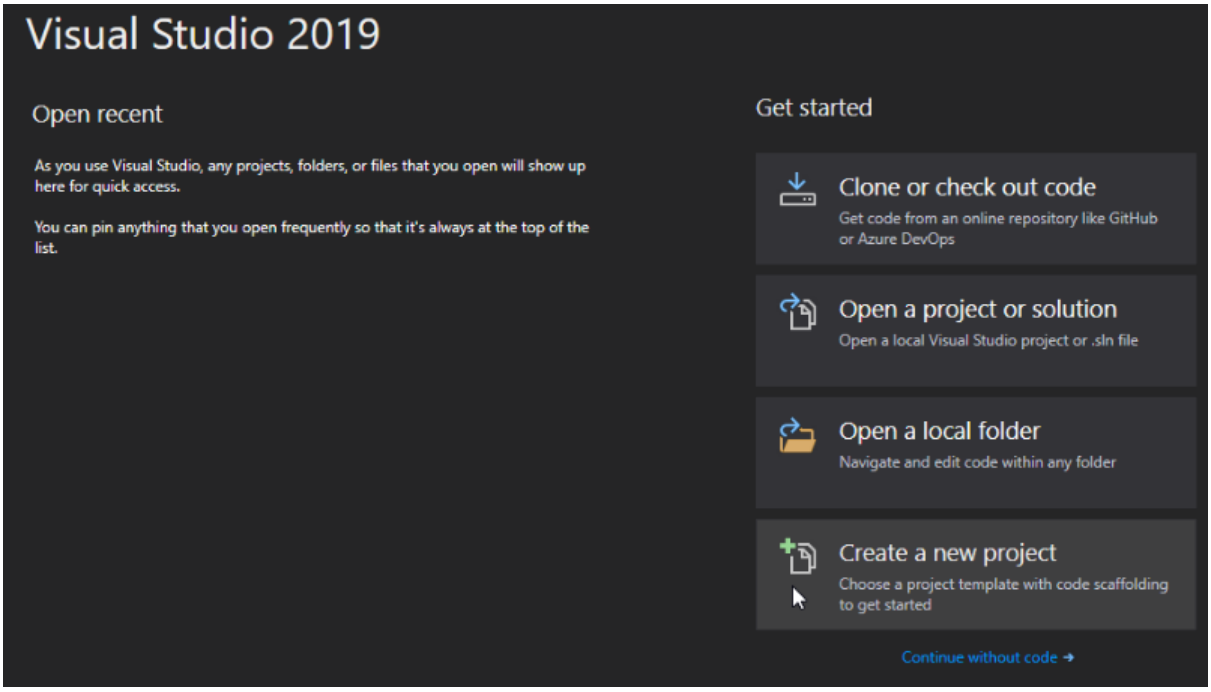
- Mobile applications
- Desktop applications
- Web applications
- Web services
- Web sites
- Games
- VR
- Database applications
- And much, much more!

C# Installation

Once the Visual Studio Installer is downloaded and installed, choose the .NET workload and click on the **Modify/Install** button:

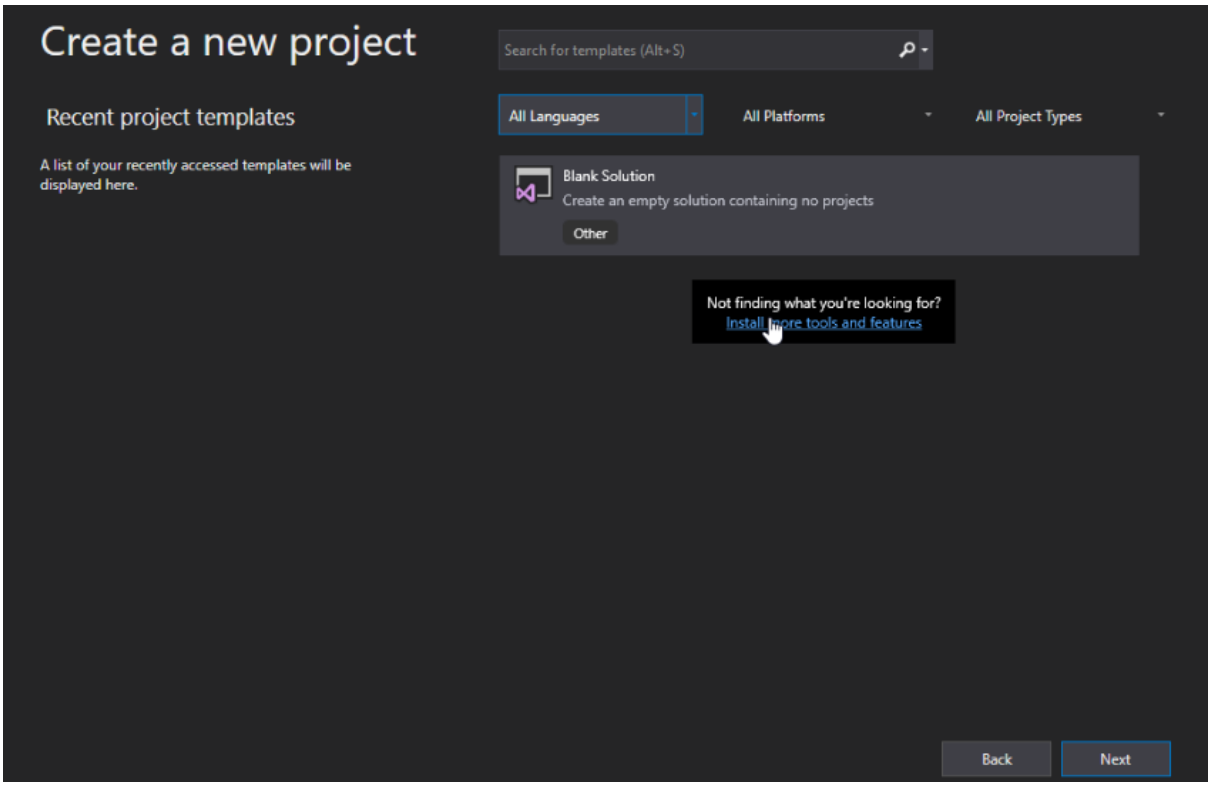


After the installation is complete, click on the **Launch** button to get started with Visual Studio.

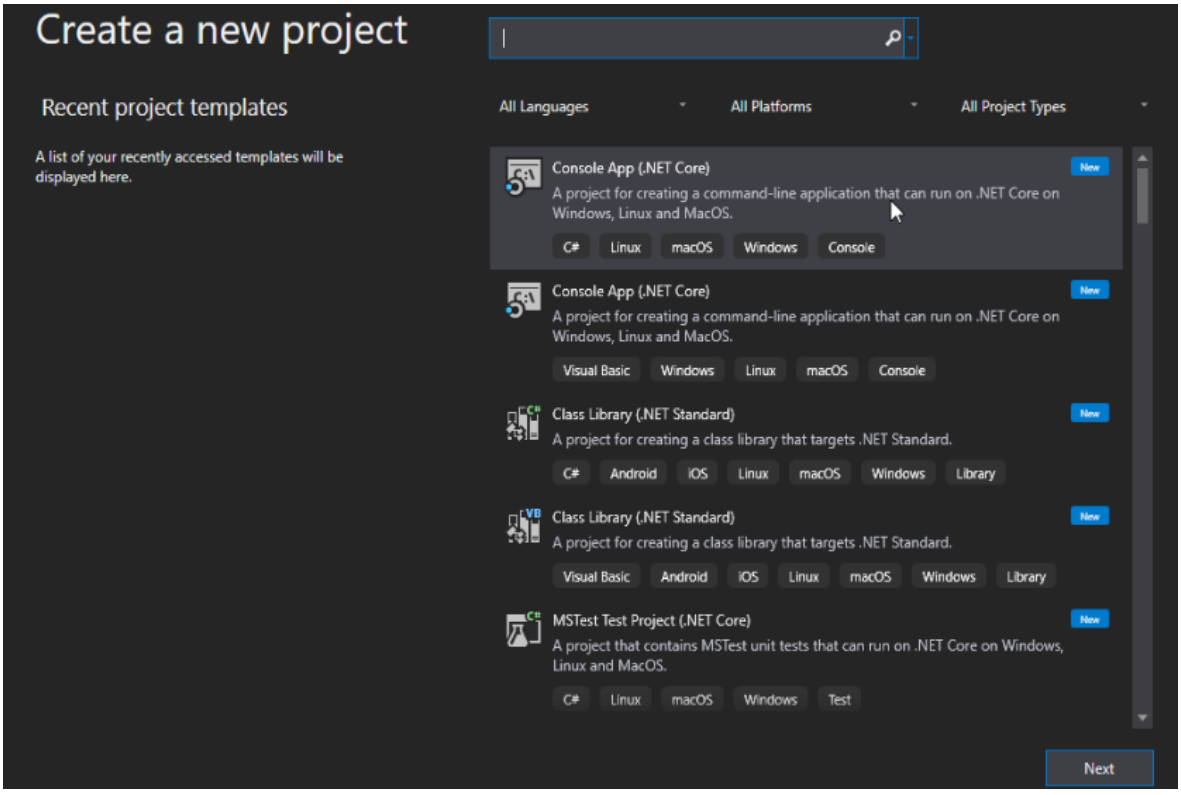


On the start window, choose **Create a new project**:

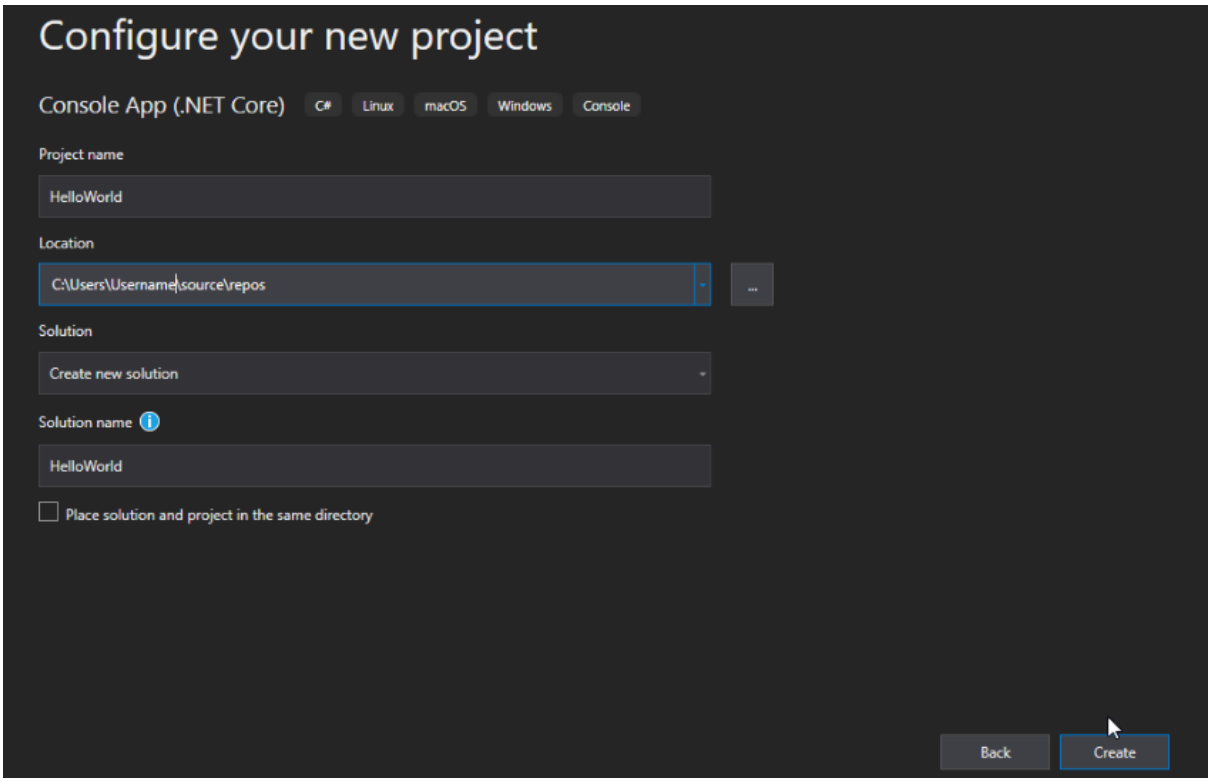
Then click on the "Install more tools and features" button:



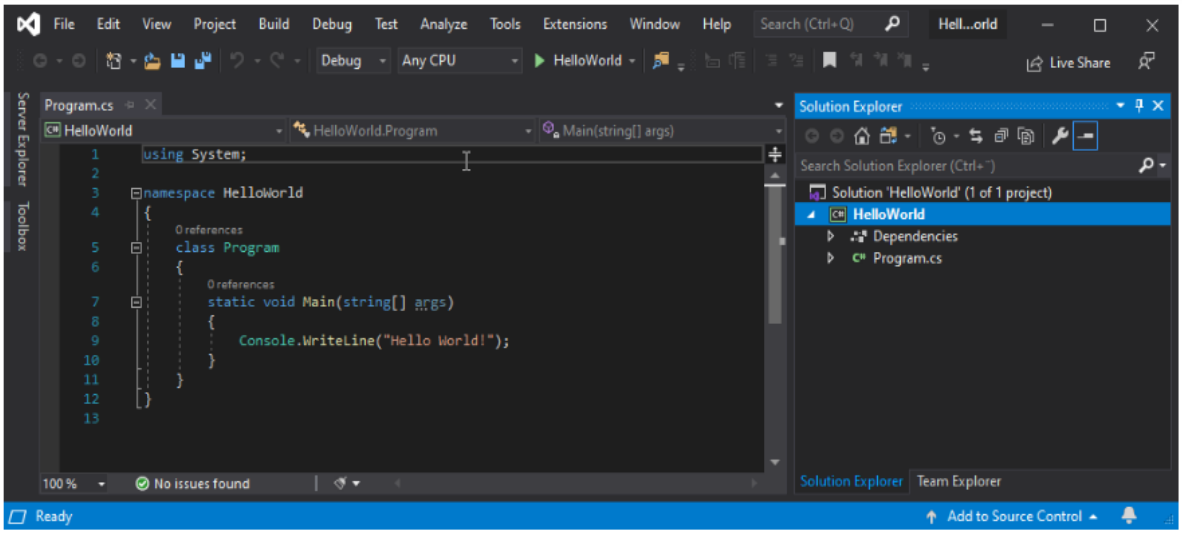
Choose "Console App (.NET Core)" from the list and click on the Next button:



Enter a name for your project, and click on the Create button:



Visual Studio will automatically generate some code for your project:



Structure of C# Program (or) Simple C# Program

```
using System;

namespace HelloWorld

{

 class Program

 {

  static void Main(string[] args)

  {

   Console.WriteLine("Hello World!");

   Console.ReadKey();

  }

 }

}
```

Run the program by pressing the **F5** button on your keyboard (or click on "**Debug**" -> "**Start Debugging**"). This will compile and execute your code.

```
Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Result:

```
Hello World!
```

Example explained

Line 1: `using System` means that we can use classes from the `System` namespace.

Line 2: A blank line. C# ignores white space. However, multiple lines makes the code more readable.

Line 3: `namespace` is a used to organize your code, and it is a container for classes and other namespaces.

Line 4: The curly braces `{}` marks the beginning and the end of a block of code.

Line 5: `class` is a container for data and methods, which brings functionality to your program. Every line of code that runs in C# must be inside a class. In our example, we named the class Program.

Don't worry if you don't understand how `using System`, `namespace` and `class` works. Just think of it as something that (almost) always appears in your program, and that you will learn more about them in a later chapter.

Line 7: Another thing that always appear in a C# program, is the `Main` method. Any code inside its curly brackets `{}` will be executed. You don't have to understand the keywords before and after Main. You will get to know them bit by bit while reading this tutorial.

Line 9: `Console` is a class of the `System` namespace, which has a `WriteLine()` method that is used to output/print text. In our example it will output "Hello World!".

If you omit the `using System` line, you would have to write `System.Console.WriteLine()` to print/output text.

Note: Every C# statement ends with a semicolon `;`

Note: C# is case-sensitive: "MyClass" and "myclass" has different meaning.

C# Namespaces

Namespaces in C# are used to organize too many classes so that it can be easy to handle the application.

In a simple C# program, we use System.Console where System is the namespace and Console is the class. To access the class of a namespace, we need to use namespacename.classname. We can use **using** keyword so that we don't have to use complete name all the time.

In C#, global namespace is the root namespace. The global::System will always refer to the namespace "System" of .Net Framework.

C# namespace example

Let's see a simple example of namespace which contains one class "Program".

```
using System;
namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Hello Namespace!");
      }
   }
}
```

Output:

```
Hello Namespace!
```

Assemblies

An Assembly is a basic building block of .Net Framework applications. It is basically compiled code that can be executed by the CLR. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An Assembly can be a DLL or exe depending upon the project that we choose.

Assemblies are basically the following two types:

1. Private Assembly
2. Shared Assembly

1. Private Assembly

It is an assembly that is being used by a single application only. Suppose we have a project in which we refer to a DLL so when we build that project that DLL will be copied to the bin folder of our project. That DLL becomes a private assembly within our project. Generally the DLLs that are meant for a specific project are private assemblies.

2. Shared Assembly

Assemblies that can be used in more than one project are known to be a shared assembly. Shared assemblies are generally installed in the GAC. Assemblies that are installed in the GAC are made available to all the .Net applications on that machine.

However there are two more types of assemblies in .Net, Satellite Assembly and Shared Assembly.

GAC

GAC stands for Global Assembly Cache. It is a memory that is used to store the assemblies that are meant to be used by various applications.

Every computer that has CLR installed must have a GAC. GAC is a location that can be seen at "C:\Windows\assembly" for .Net applications with frameworks up to 3.5. For higher frameworks like 4 and 4.5 the GAC can be seen at:
"C:\Windows\Microsoft.NET\assembly\GAC_MSIL".

C# Comments

Comments can be used to explain C# code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

C# Single Line Comment

The single line comment starts with // (double slash). Any text between // and the end of the line is ignored by C# (will not be executed).

This example uses a single-line comment before a line of code:

Example

```
// This is a comment

Console.WriteLine("Hello World!");
```

This example uses a single-line comment at the end of a line of code:

Example

```
Console.WriteLine("Hello World!");  // This is a comment
```

C# Multi-line Comments

Multi-line comments start with /* and ends with */.

Any text between /* and */ will be ignored by C#.

This example uses a multi-line comment (a comment block) to explain the code:

Example

```
/* The code below will print the words Hello World

to the screen, and it is amazing */

Console.WriteLine("Hello World!");
```

Single or multi-line comments?

It is up to you which you want to use. Normally, we use // for short comments, and /* */ for longer.

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

```
Data_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.

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

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 values are 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 of Variables

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;
```

C# Value Data Types

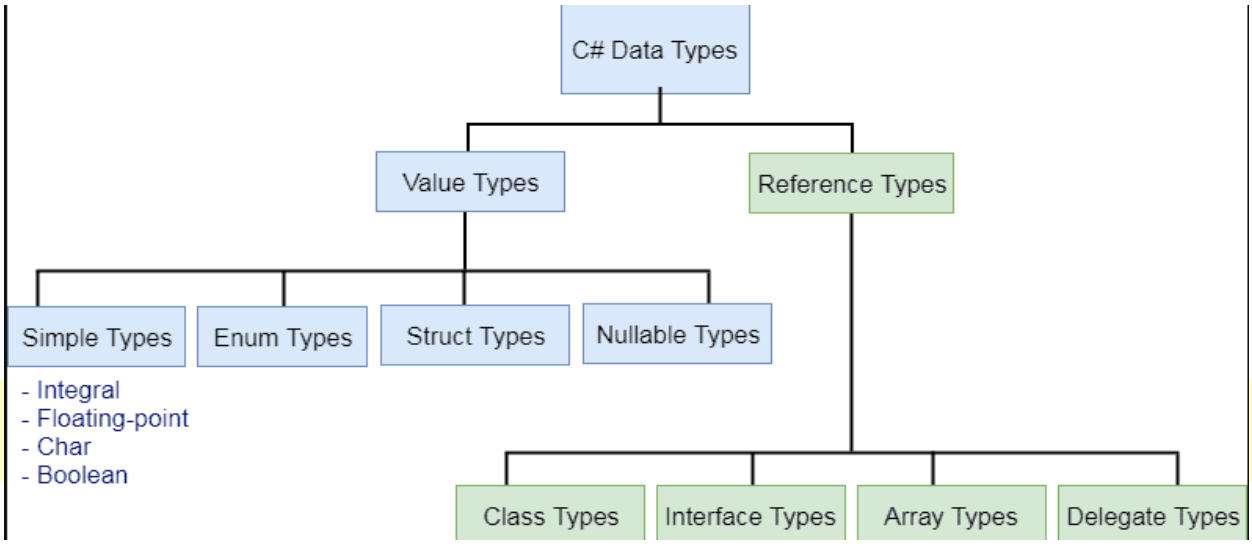
As explained in the variables chapter, a variable in C# must be a specified data type:

Example

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

```
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:



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

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

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);
```

Use float or double?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of `float` is only six or seven decimal digits, while `double` variables have a precision of about 15 digits. Therefore it is safer to use `double` for most calculations.

Scientific Numbers

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

Example

```
float f1 = 35e3F;

double d1 = 12E4D;

Console.WriteLine(f1);

Console.WriteLine(d1);
```

C# Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, C# has a `bool` data type, which can take the values `true` or `false`.

Boolean Values

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

Example

```
bool isCSharpFun = true;

bool isFishTasty = false;

Console.WriteLine(isCSharpFun);   // Outputs True

Console.WriteLine(isFishTasty);   // Outputs False
```

However, it is more common to return boolean values from boolean expressions, for conditional testing (see below).

Boolean Expression

A **Boolean expression** is a C# expression that returns a Boolean value: `True` or `False`.

You can use a comparison operator, such as the **greater than** (`>`) operator to find out if an expression (or a variable) is true:

Example

```
int x = 10;

int y = 9;

Console.WriteLine(x > y); // returns True, because 10 is higher than 9
```

Example

```
Console.WriteLine(10 > 9); // returns True, because 10 is higher than 9
```

In the examples below, we use the **equal to** (`==`) operator to evaluate an expression:

Characters

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

Example

```
char myGrade = 'B';

Console.WriteLine(myGrade);
```

Strings

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

Example

```
string greeting = "Hello World";

Console.WriteLine(greeting);
```

String Length method

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:

Example

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

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

Other Methods using strings

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

Example

```
string txt = "Hello World";

Console.WriteLine(txt.ToUpper());   // Outputs "HELLO WORLD"

Console.WriteLine(txt.ToLower());   // Outputs "hello world"
```

Special Characters or Escape Sequences

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

```
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.";
```

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

Example

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

Other useful escape characters in C# are:

Code	Result
\n	New Line
\t	Tab
\b	Backspace

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);      // Outputs 9

Console.WriteLine(myDouble);   // Outputs 9
```

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 also 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 using ReadLine Method

How to give Input from keyboard at runtime

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 ReadLine Method

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);
```

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'.

Luckily, for you, you just learned from the previous chapter (Type Casting), that you can convert any type explicitly, by using one of the `Convert.To` methods:

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.').

WriteLine or Write Output Method

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# 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)
```

Aritmetic Operators

Arithmetic operators are used to perform common mathematical operations and operations like add, sub, multiply and divide

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. 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

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;
```

C# Comparison Operators

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

Comparison operators are used to compare two values:

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# Built in Mathematical Methods

The C# Math class has many methods that allows you to perform mathematical tasks on numbers.

Math.Max(*x*,*y*)

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

Example

```
Math.Max(5, 10);
```

Math.Min(*x*,*y*)

The `Math.Min(x,y)` method can be used to find the lowest value of 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);
```

Math.Abs(*x*)

The Math.Abs(*x*) method returns the absolute (positive) value of *x*:

Example

```
Math.Abs(-4.7);
```

Math.Round()

Math.Round() rounds a number to the nearest whole number:

Example

```
Math.Round(9.99);
```

Conditional Statements in C#

C# 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."
```

Example explained

In the example above, time (20) is greater than 18, so the condition is False. Because of this, we move on to the else condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

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."
```

Example explained

In the example above, time (22) is greater than 10, so the **first condition** is False. The next condition, in the else if statement, is also False, so we move on to the else condition since **condition1** and **condition2** is both False - and print to the screen "Good evening".

However, if the time was 14, our program would print "Good day."

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 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)
```

C# Break

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);
}
```

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."
```

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

```
for (int i = 0; i < 10; i++)
{
  if (i == 4)
  {
```

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

goto statement

This statement is used to transfer control to the labeled statement in the program. The label is the valid identifier and placed just before the statement from where the control is transferred.

Example:

```
// C# program to illustrate the
// use of goto statement
using System;
 class ISM {
   // Main Method
   static public void Main()
   {
      int number = 20;
      switch (number) {
        case 5:
          Console.WriteLine("case 5");
          break;
      case 10:
          Console.WriteLine("case 10");
          break;
      case 20:
          Console.WriteLine("case 20");
            // goto statement transfer
            // the control to case 5
          goto case 5;
        default:
          Console.WriteLine("No match found");
          break;
```

```
        }
    }
}
```

Output:

```
case 20
case 5
```

Return statement

This statement terminates the execution of the method and returns the control to the calling method. It returns an optional value. If the type of method is void, then the return statement can be excluded.

Example:

```
// C# program to illustrate the
// use of return statement
using System;
  class ISM
 {
      // creating simple addition function
    static int Addition(int a)
    {
        // add two value and
        // return the result of addition
        int add = a + a;
       // using return statement
        return add;
    }

    // Main Method
    static void Main()
    {
        int number = 2;
       // calling addition function
        int result = Addition(number);
        Console.WriteLine("The addition is {0}", result);
    }
}
```

Output:

```
The addition is 4
```

Looping Statements in C#

Loops Definition:

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# 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);
```

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

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);

}
```

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);

}
```

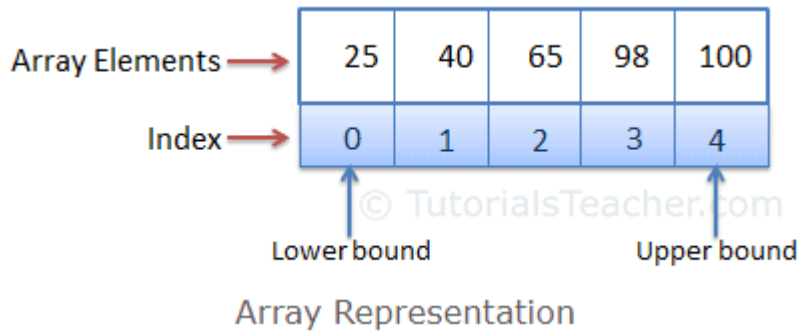
UNIT-II

C# Arrays

An array is the data structure that stores a fixed number of literal values (elements) of the same data type. Array elements are stored contiguously in the memory.

In C#, an array can be of three types: single-dimensional, multidimensional, and jagged array. Here you will learn about the single-dimensional array.

The following figure illustrates an array representation.



Array Representation

Array Declaration and Initialization

An array can be declared using by specifying the type of its elements with square brackets.

Example: Array Declaration

```
int[] evenNums;  // integer array
```

```
string[] cities; // string array
```

The following declares and adds values into an array in a single statement.

Example: Array Declaration & Initialization

```
int[] evenNums = new int[5]{ 2, 4, 6, 8, 10 };

string[] c = new string[3]{ "Mumbai", "London", "New York" };
```

Above, evenNums array can store up to five integers. The number 5 in the square brackets new int[5] specifies the size of an array. In the same way, the size of cities array is three. Array elements are added in a comma-separated list inside curly braces { }.

Arrays type variables can be declared using var without square brackets.

Example: Array Declaration using var

```
var evenNums = new int[]{ 2, 4, 6, 8, 10};

var cities = new string[]{ "Mumbai", "London", "New York" };
```

If you are adding array elements at the time of declaration, then size is optional.

Input and output operations on One-dimensional Array

```
using System;

public class array

{

   public static void Main()

{

   int[] arr = new int[5];

   int i;

      Console.Write("\n\nRead and Print elements of an array:\n");

      Console.Write("----------------------------------------\n");


   Console.Write("Input 5 elements in the array :\n");

   for(i=0; i<5; i++)

   {
```

```
            Console.Write("element - :{0}" ,i);
            arr[i] = Convert.ToInt32(Console.ReadLine());
    }


    Console.Write("\nElements in array are: ");
    for(i=0; i<5; i++)
    {
        Console.Write("{0}  ", arr[i]);
    }
    Console.Write("\n");
  }
}
```

```
Read and Print elements of an array:
----------------------------------------
Input 10 elements in the array :
element - 0 : 2
element - 1 : 4
element - 2 : 6
element - 3 : 8
element - 4 : 10

Elements in array are: 2  4  6  8  10
```

Accessing Array using foreach Loop

Use foreach loop to read values of an array elements without using index.

Example: Accessing Array using foreach Loop

```
int[] evenNums = { 2, 4, 6, 8, 10};
string[] cities = { "Mumbai", "London", "New York" };

foreach(var item in evenNums)
    Console.WriteLine(item);

foreach(var city in cities)
    Console.WriteLine(city);
```

C# - Multidimensional Arrays

C# supports multidimensional arrays up to 32 dimensions. The multidimensional array can be declared by adding commas in the square brackets. For example, [,] declares two-dimensional array, [, ,] declares three-dimensional array, [, , ,] declares four-dimensional array, and so on. So, in a multidimensional array, no of commas = No of Dimensions - 1.

The following declares multidimensional arrays.

Example: Multidimensional Arrays

```
int[,] arr2d; // two-dimensional array
int[, ,] arr3d; // three-dimensional array
int[, , ,] arr4d ; // four-dimensional array
int[, , , ,] arr5d; // five-dimensional array
```

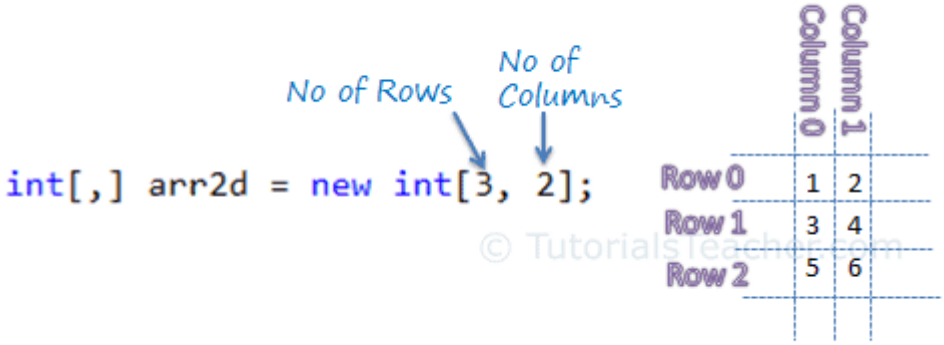
Let's understand the two-dimensional array. The following initializes the two-dimensional array.

Example: two-dimensional Array

```
int[,] arr2d = new int[3,2]{
                            {1, 2},
                            {3, 4},
                            {5, 6}
                        };

// or
int[,] arr2d = {
                    {1, 2},
                    {3, 4},
                    {5, 6}
                };
```

In the above example of a two-dimensional array, `[3, 2]` defines the no of rows and columns. The first rank denotes the no of rows, and the second rank defines no of columns. The following figure illustrates the two-dimensional array divided into rows and columns.



Two-dimensional Array

Input and Output operations in Two-dimensional Array

```
using System;

public class Exercise14

{

  public static void Main()

{

  int i,j;

            int[,] arr1 = new int[3,3];


    Console.Write("\n\nRead a 2D array of size 3x3 and print the matrix :\n");

    Console.Write("---------------------------------------------------\n");


  /* Stored values into the array*/

    Console.Write("Input elements in the matrix :\n");

for(i=0;i<3;i++)

{

   for(j=0;j<3;j++)

   {

              Console.Write("element - [{0},{1}] : ",i,j);

                    arr1[i,j] = Convert.ToInt32(Console.ReadLine());

   }

}
```

```
 Console.Write("\nThe matrix is : \n");
  for(i=0;i<3;i++)
  {
     Console.Write("\n");
     for(j=0;j<3;j++)
          Console.Write("{0}\t",arr1[i,j]);
    }
   Console.Write("\n\n");
   }
}
```

Sample Output:

```
Read a 2D array of size 3x3 and print the matrix :
--------------------------------------------------------
Input elements in the matrix :
element - [0,0] : 1
element - [0,1] : 2
element - [0,2] : 3
element - [1,0] : 4
element - [1,1] : 5
element - [1,2] : 6
element - [2,0] : 7
element - [2,1] : 8
element - [2,2] : 9

The matrix is :

1       2       3
4       5       6
7       8       9
```

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. Static keyword is used for the methods to be used in static void main method.

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.
- void means that this method does not have a return value. **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();  // calling a method

}

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

C# 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 + "Welcome");

}

static void Main(string[] args)

{

  MyMethod(“Arun");

  MyMethod("Thoufiq");

  MyMethod("Samsu");

}

// Arun Refsnes

// Thoufiq Refsnes

// Samsu Refsnes
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: `fname` is a **parameter**, while `Liam`, `Jenny` and `Anja` are **arguments**.

Value Type and Reference Type

In C#, these data types are categorized based on how they store their value in the memory. C# includes the following categories of data types:

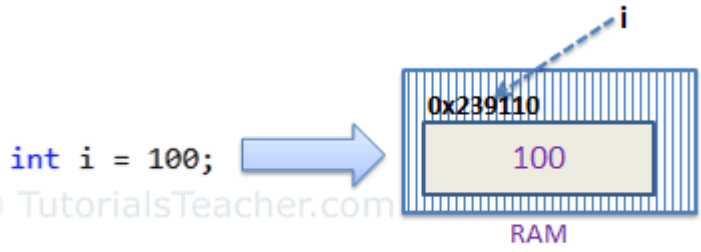
1. Value type — Pass by Value
2. Reference type — Pass by Reference
3. Pointer type

Value Type

A data type is a value type if it holds a data value within its own memory space. It means the variables of these data types directly contain values.

For example, consider integer variable `int i = 100;`

The system stores 100 in the memory space allocated for the variable i. The following image illustrates how 100 is stored at some hypothetical location in the memory (0x239110) for 'i':



Memory Allocation of Value Type Variable

Passing Value Type Variables

When you pass a value-type variable from one method to another, the system creates a separate copy of a variable in another method. If value got changed in the one method, it wouldn't affect the variable in another method.

Example: Passing Value Type Variables

```
class Pass
{
static void ChangeValue(int x)
{
   x =  200;

   Console.WriteLine(x);
}

static void Main(string[] args)
{
```

```
    int i = 100;

    Console.WriteLine(i);

    ChangeValue(i);

    Console.WriteLine(i);
    Console.ReadKey();
}
}
```

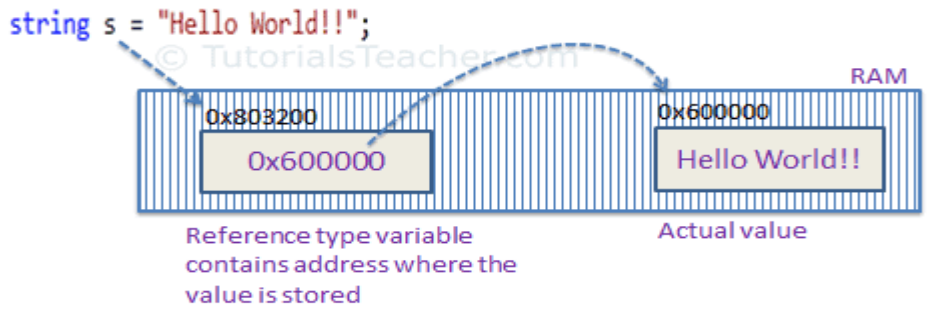
In the above example, variable i in the Main() method remains unchanged even after we pass it to the ChangeValue() method and change it's value there.

Reference Type

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

For example, consider the following string variable:

```
string s = "Hello World!!";
```



Memory Allocation of Reference Type Variable

The followings are reference type data types:

- String
- Arrays (even if their elements are value types)
- Class
- Delegate

Passing Reference Type Variables

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the variable's address. So, If we change the value of a variable in a method, it will also be reflected in the calling method.

Example: Passing Reference Type Variable

```
class Reference
{
static void Refer(ref int x)
{
   x = 200;
}

static void Main(string[] args)
{
        int i = 100;
        Console.WriteLine(i);
        Refer(ref i);
        Console.WriteLine(i);
        Console.ReadKey();
}
}
```

C# Member Overloading

If we create two or more members having same name but different in number or type of parameter, it is known as member overloading. In C#, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

C# Method Overloading

Having two or more methods with same name but different in parameters, is known as method overloading in C#.

The **advantage** of method overloading is that it increases the readability of the program because you don't need to use different names for same action.

You can perform method overloading in C# by two ways:

1. By changing number of arguments
2. By changing data type of the arguments

C# Method Overloading Example: By changing no. of arguments

Let's see the simple example of method overloading where we are changing number of arguments of add() method.

```
using System;
  static int add(int a,int b)
  {
      return a + b;
  }
  static int add(int a, int b, int c)
  {
```

```
        return a + b + c;
    }
public class MethodOverloading
{
    public static void Main()
    {
        Console.WriteLine(add(12, 23));
        Console.WriteLine(add(12, 23, 25));
    }
}
```

Output:

```
35
60
```

C# Member Overloading Example: By changing data type of arguments

Let's see the another example of method overloading where we are changing data type of arguments.

```
using System;
 static int add(int a, int b)
{
        return a + b;
    }
    static float add(float a, float b)
    {
        return a + b;
    }
}
public class TestMemberOverloading
{
    public static void Main()
    {
        Console.WriteLine(add(12, 23));
        Console.WriteLine(add(12.4f,21.3f));
    }
}
```

Output:

```
35
33.7
```

UNIT-III

C# Strings

In C#, string is an object of **System.String** class that represent sequence of characters. We can perform many operations on strings such as concatenation, comparision, getting substring, search, trim, replacement etc.

string vs String

In C#, *string* is keyword which is an alias for *System.String* class. That is why string and String are equivalent. We are free to use any naming convention.

```
string s1 = "hello";//creating string using string keyword
String s2 = "welcome";//creating string using String class
```

Reading String from User-Input

A string can be read out from the user input. ReadLine() method of console class is used to read a string from user input.

- **Example:**

```
// C# program to demonstrate Reading
// String from User-Input
using System;

class ISM
 {

   // Main Method
   static void Main(string[] args)
   {
        Console.WriteLine("Enter the String");

      // Declaring a string object read_user
      // and taking the user iput using
      // ReadLine() method
      String read_user = Console.ReadLine();

      // Displaying the user input
      Console.WriteLine("User Entered: " + read_user);
     }
}
```

Input:

Hello ISM !

Output:

```
Enter the String

User Entered: Hello ISM!
```

C# String ToString()

The C# ToString() method is used to get instance of String.

Return

It returns a string object.

C# String ToString() Method Example

```
using System;
      public class StringExample
  {
     public static void Main(string[] args)
     {
       String s1 = "Hello C#";
       int a = 123;
```

```
        String s2 = s1.ToString();
        String s3 = a.ToString();
        Console.WriteLine(s2);
        Console.WriteLine(s3);
    }
}
```

Output:

```
Hello C#
123
```

C# String Class Methods

String Copy()

The C# Copy() method is used to create a new instance of String with the same value as a specified String. It is a static method of String class. Its return type is string.

it takes a string argument which is used to create a copy of specified string.

Return

It returns string object.

C# String Copy() Method Example

```
using System;
    public class StringExample
    {
        public static void Main(string[] args)
        {
            String s1 = "Hello ";
            String s2 = string.Copy(s1);
            Console.WriteLine(s1);
            Console.WriteLine(s2);
        }
    }
```

Output:

```
Hello
Hello
```

C# String Concat()

The C# Concat() method is used to concatenate multiple string objects. It returns concatenated string. There are many overloaded methods of Concat().

It takes two String object arguments.

Return

It returns a string object.

C# String Concat() Method Example

```
using System;
   public class StringExample
   {
      public static void Main(string[] args)
      {
          String s1 = "Hello ";
          String s2 = "C#";
          Console.WriteLine(String.Concat(s1,s2));
          Console.ReadKey();
      }
   }
```

Output:

Hello C#

C# String Compare()

The C# Compare() method is used to compare first string with second string lexicographically. It returns an integer value.

If both strings are equal, it returns 0. If first string is greater than second string, it returns 1 else it returns -1.

Rule

s1==s2 returns 0

s1>s2 returns 1

s1<s2 returns -1

C# String Compare() Method Example

```
using System;
   public class StringExample
```

```
    {
        public static void Main(string[] args)
        {
            string s1 = "hello";
    string s2 = "hello";
    string s3 = "csharp";
    string s4 = "mello";
Console.WriteLine(string.Compare(s1,s2));
Console.WriteLine(string.Compare(s2,s3));
Console.WriteLine(string.Compare(s3,s4));
        }
    }
```

Output:

```
0
1
-1
```

C# String Insert()

The C# Insert() method is used to insert the specified string at specified index number. The index number starts from 0. After inserting the specified string, it returns a new modified string.

Parameters

first: It is used to pass as an index.

second: It is used to insert the given string at specified index.

Return

It returns a new modified string.

C# String Insert() Method Example

```
using System;
    public class StringExample
    {
        public static void Main(string[] args)
        {
          string s1 = "Hello C#";
          string s2 = s1.Insert(5,"-");
          Console.WriteLine(s2);
        }
```

```
}
```

Output:

Hello- C#

C# String Equals()

The C# Equals() method is used to check whether two specified String objects have the same value or not. If both strings have same value, it return true otherwise false.

Parameter

str: it is a string object.

Return

It returns boolean value either true or false.

C# String Equals() Method Example

```
using System;
public class StringExample
  {
    public static void Main(string[] args)
    {
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = "Bye";
        Console.WriteLine(s1.Equals(s2));
        Console.WriteLine(s1.Equals(s3));
    }
  }
```

Output:

```
True
False
```

C# Structures

Structure is a value type and a collection of variables of different data types under a single unit. It is almost similar to a class because both are user-defined data types and both hold a bunch of different data types. C# provide the ability to use pre-defined **data types**. However, sometimes the user might be in need to define its own data types which are also known as **User-Defined Data**

Types. Although it comes under the value type, the user can modify it according to requirements and that's why it is also termed as the user-defined data type.

Defining Structure: In C#, structure is defined using ***struct*** keyword. Using struct keyword one can define the structure consisting of different data types in it. A structure can also contain constructors, constants, fields, methods, properties, indexers and events etc.

Syntax:

```
Access_Modifier struct structure_name
{
// Fields
// Methods etc.
}
```

Example:

```
// C# program to illustrate the
// Declaration of structure
using System;
namespace ConsoleApplication
 {
// Defining structure
public struct Person
{
   // Declaring different data types
   public string Name;
   public int Age;
   public int Weight;
 }
  class ISM {
     // Main Method
   static void Main(string[] args)
   {
        // Declare P1 of type Person
      Person P1;
        // P1's data
      P1.Name = "Keshav Gupta";
      P1.Age = 21;
      P1.Weight = 80;

      // Displaying the values
```

```
        Console.WriteLine("Data Stored in P1 is " +
                          P1.Name + ", age is " +
                          P1.Age + " and weight is " +
                          P1.Weight);
    }
  }
}
```

Output:

```
Data Stored in P1 is Keshav Gupta, age is 21 and weight is 80
```

Explanation: In the above code, a structure with name **"Person"** is created with data members **Name**, **Age** and **Weight**.In the main method, **P1** of structure type Person is created. Now, P1 can access its data members with the help of **.(dot) Operator**.

C# Enumeration (or enum)

Enumeration (or enum) is a value data type in C#. It is mainly used to assign the names or string values to integral constants, that make a program easy to read and maintain. For example, the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.). The main objective of enum is to define our own data types(Enumerated Data Types). Enumeration is declared using ***enum*** keyword directly inside a namespace, class, or structure.

Syntax:

```
enum Enum_variable
{
    string_1...;
    string_2...;
    .
    .
}
```

In above syntax, Enum_variable is the name of the enumerator, and string_1 is attached with value 0, string_2 is attached value 1 and so on. Because by default, the first member of an enum has the value 0 and the value of each successive enum member is increased by 1. We can change this default value.

Example 1: Consider the below code for the enum. Here enum with name **month** is created and its data members are the name of months like jan, feb, mar, apr, may. Now let's try to print the default integer values of these enums. **An explicit cast is required to convert from enum type to an integral type.**

```
// C# program to illustrate the enums
// with their default values
using System;
```

```
namespace ConsoleApplication1 {
 // making an enumerator 'month'
enum month
{
     // following are the data members
   jan,
   feb,
   mar,
   apr,
   may
  }
  class Program {
   // Main Method
   static void Main(string[] args)
   {
      // getting the integer values of data members
      Console.WriteLine("The value of jan in month " +
                        "enum is " + (int)month.jan);
      Console.WriteLine("The value of feb in month " +
                        "enum is " + (int)month.feb);
      Console.WriteLine("The value of mar in month " +
                        "enum is " + (int)month.mar);
      Console.WriteLine("The value of apr in month " +
                        "enum is " + (int)month.apr);
      Console.WriteLine("The value of may in month " +
                        "enum is " + (int)month.may);
   }
}
}
```

Output:

```
The value of jan in month enum is 0
The value of feb in month enum is 1
The value of mar in month enum is 2
The value of apr in month enum is 3
The value of may in month enum is 4
```

Classes and Objects

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:

```
class classname
{
  variable declaration;
  methods declaration;
}
```

Example

```
class Rectangle
{
int length;
int width;

public void getdata(int x, int y)
{
length=x;
width=y;
}
}
```

length and width is a variable of int data type and it is private by default and its values are given only in method of the class only. we can use keyword public for the class methods, protected and internal for inheritance for its scope.

Create an Object

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

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

Example

Create an object called "rect" and use it to print the value of length and width.

```
class Rectangle

{

int length;
int width;

public void getdata(int x, int y)
 {
length=x;
width=y;
}

public int  area()
{
int a=length * width;
return(a);
}
}  // End of the Class Structure


class Program
{
static void Main(string[] args)
  {
   Rectangle r=new Rectangle();
   r.getdata(10,20);
   Console.WriteLine("Area of a Rectangle=" +r.area());
   Console.ReadKey();
  }
} // End of the Class Sample
```

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of Car:

```
class Car
  {
    public string color = "Green";

    public void fullspeed()    // method
    {
      Console.WriteLine("The car is going as fast as it can!");
    }
  }  // End of the Car Class

 class Program
  {

    static void Main(string[] args)
    {
      Car c1 = new Car();
      Car c2=new Car();
      Console.WriteLine("The color is " + c1.color);
      Console.WriteLine("The color is " + c2.color);
      c1.fullspeed();
      c2.fullspeed();
      Console.ReadKey();
    }

  }
```

Unit-IV

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()  //  void int

  {
```

```
    model = "Mustang"; // Set the initial value for model
  }


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

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.

C# | Default Constructor

If you don't provide a constructor for your class, C# creates one by default that instantiates the object and sets member variables to the default values as listed in the **Default Values Table**. Constructor without any parameters is called a default constructor. In other words, this type of constructor does not take parameters. The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class to different values.

The default constructor initializes:

- All numeric fields in the class to zero.
- All string and object fields to null.

Example 1:

```
// C# Program to illustrate the use
// of Default Constructor
using System;
class  multiplication
{
   int  a, b;
   // default Constructor
   public  multiplication()
   {
```

```
        a = 10;
        b = 5;
    }
// Main Method
 static  void  Main() {

    // an object is created,
    // constructor is called
    multiplication obj = new  multiplication();

    Console.WriteLine(obj.a);
    Console.WriteLine(obj.b);
    Console.WriteLine("The result of multiplication is: "
                                +(obj.a * obj.b));
}
}
```

Output:

```
10
5
The result of multiplication is: 50
```

Parameterized Constructors

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

Destructors in C#

Destructors in C# are methods inside the class used to destroy instances of that class when they are no longer needed. The Destructor is called implicitly by the .NET Framework's Garbage collector and therefore programmer has no control as when to invoke the destructor. An instance variable or an object is eligible for destruction when it is no longer reachable.

Important Points:

- A Destructor is unique to its class i.e. there cannot be more than one destructor in a class.
- A Destructor has no return type and has exactly the same name as the class name (Including the same case).
- It is distinguished apart from a constructor because of the *Tilde symbol (~)* prefixed to its name.
- A Destructor does not accept any parameters and modifiers.
- It cannot be defined in Structures. It is only used with classes.
- It cannot be overloaded or inherited.
- It is called when the program exits.
- Internally, Destructor called the Finalize method on the base class of object.

Example:

```
class Example
{
   // Rest of the class
   // members and methods.
```

```
  // Destructor
  ~Example()
   {
      // Your code
   }

}
```

Example for Destructor

```
 using System;
 class Employee
   {
      public Employee()
      {
        Console.WriteLine("Constructor Invoked");
      }
      ~Employee()
      {
         Console.WriteLine("Destructor Invoked");
      }
   }
  class TestEmployee
      {
     public static void Main(string[] args)
      {
         Employee e1 = new Employee();
         Employee e2 = new Employee();
      }
      }
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

Basic Principles of OOPS

Class

A class is the core of any modern Object Oriented Programming language such as C#.

In OOP languages it is mandatory to create a class for representing data.

A class is a blueprint of an object that contains variables for storing data and functions to perform operations on the data.

A class will not occupy any memory space and hence it is only a logical representation of data.

To create a class, you simply use the keyword "class" followed by the class name:

```
class Employee

{

}
```

Object

Objects are the basic run-time entities of an object oriented system. They may represent a person, a place or any item that the program must handle.

"An object is a software bundle of related variable and methods."

"An object is an instance of a class"

A class will not occupy any memory space. Hence to work with the data represented by the class you must create a variable for the class, that is called an object.

When an object is created using the new operator, memory is allocated for the class in the heap, the object is called an instance and its starting address will be stored in the object in stack memory.

When an object is created without the new operator, memory will not be allocated in the heap, in other words an instance will not be created and the object in the stack contains the value **null**.

```
class Employee

{

}
```

Syntax to create an object of class Employee:

```
Employee objEmp = new Employee();
```

All the programming languages supporting Object Oriented Programming will be supporting these three main concepts,

1. Encapsulation
2. Inheritance
3. Polymorphism

Abstraction

Abstraction is "To represent the essential feature without representing the background details."

Abstraction lets you focus on what the object does instead of how it does it.

Abstraction provides you a generalized view of your classes or objects by providing relevant information.

Abstraction is the process of hiding the working style of an object, and showing the information of an object in an understandable manner.
Abstract information (necessary and common information) for the object "Mobile Phone" is that it makes a call to any number and can send SMS.

Encapsulation

Wrapping up a data member and a method together into a single unit (in other words class) is called Encapsulation.

Encapsulation is like enclosing in a capsule. That is enclosing the related operations and data related to an object into that object.

Encapsulation is like your bag in which you can keep your pen, book etcetera. It means this is the property of encapsulating members and functions.

```
class Bag

 {

    book;

    pen;

    ReadBook();

}
```

Encapsulation means hiding the internal details of an object, in other words how an object does something.

Encapsulation prevents clients from seeing its inside view, where the behaviour of the abstraction is implemented.

Encapsulation is a technique used to protect the information in an object from another object.

Hide the data for security such as making the variables private, and expose the property to access the private data that will be public.

Inheritance

When a class includes a property of another class it is known as inheritance.

Inheritance is a process of object reusability.

I'm a Parent Class.

Polymorphism

Polymorphism means one name, many forms.

One function behaves in different forms.

In other words, "Many forms of a single object is called Polymorphism."

C# Inheritance

Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in C# by which one class is allowed to inherit the features(fields and methods) of another class.

Important terminology:

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in program

To inherit from a class, use the : symbol.

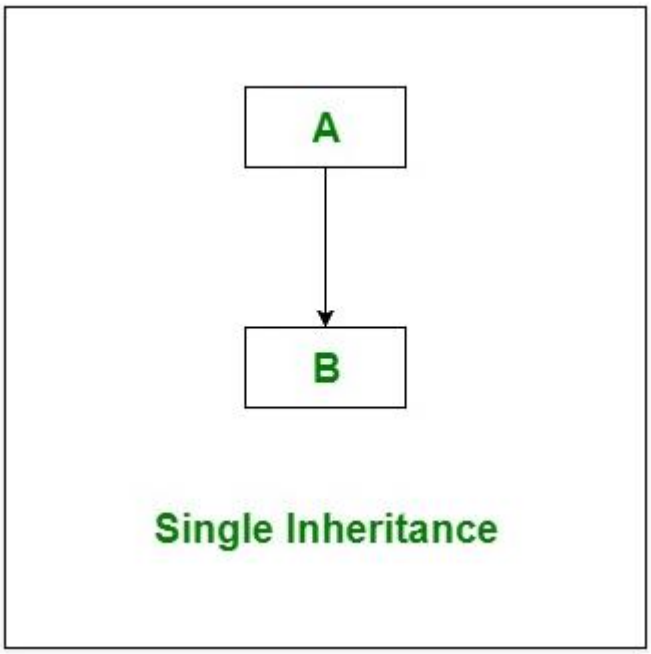
Syntax:

```
class derived-class : base-class
{
  // methods and fields
}
```

Types of Inheritance in C#

Below are the different types of inheritance which is supported by C# in different combinations.

Single Inheritance: In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.



Single Inheritance

In the example below, the Car class (child) inherits the fields and methods from the Vehicle class (parent):

Example

```
class Vehicle  // base class (parent)
{
  public string brand = "Ford";  // Vehicle field
  public void honk()            // Vehicle method
  {
  Console.WriteLine("Tuut, tuut!");
  }
} // End of the Class Vehicel
class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
  public void sound()
    {
```

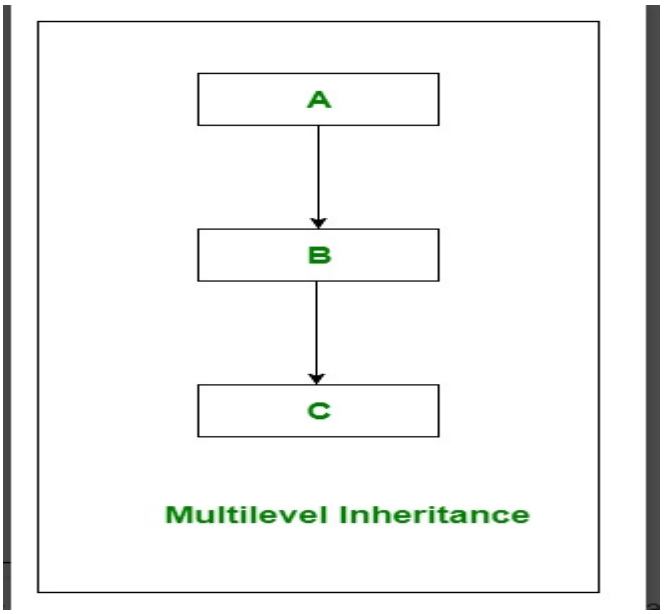
```
    Console.WriteLine(“ Boom, Boom”);
  }
} // End of the Class Car

class Program
  {
 static void Main(string[] args)
 {
  // Create a myCar object
  Car myCar = new Car();

  // Call the honk() method (From the Vehicle class) on the myCar object
  myCar.honk();

  // Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class
  Console.WriteLine(myCar.brand + " " + myCar.modelName);
 }
}
```

Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.




```
using System;
namespace Demo {
    class Son : Father
    {
    public void DisplayTwo()
   {
      Console.WriteLine("Son.. ");
    }
      static void Main(string[] args)
     {
      Son s = new Son();
     Console.writlen(s.a);
      Console.writlen(s.b);
      s.Display();
      s.DisplayOne();
      s.DisplayTwo();
      Console.ReadKey();
    }
  } // End of the Class Son
```

```
class Grandfather
{
    int a=20;
     public void Display() {
     Console.WriteLine("Grandfather...");
   }
} // End of the GranFather Class
class Father : Grandfather
{
     int b=100;
    public void DisplayOne()
   {
     Console.WriteLine("Father...");
   }
} // End of the Father Class
}
```

Output

```
Grandfather...
Father...
Son..
```

Unit-V

C# Interface

Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.

It is used *to achieve multiple inheritance* which can't be achieved by class. It is used *to achieve fully abstraction* because it cannot have method body.

Its implementation must be provided by class or struct. The class or struct which implements the interface, must provide the implementation of all the methods declared inside the interface.

C# interface example

Let's see the example of interface in C# which has draw() method. Its implementation is provided by two classes: Rectangle and Circle.

```
using System;
public interface Drawable
{
 void draw();
}
public class Rectangle : Drawable
{
    public void draw()
    {
        Console.WriteLine("drawing rectangle...");
    }
}
public class Circle : Drawable
{
    public void draw()
    {
        Console.WriteLine("drawing circle...");
    }
}
public class TestInterface
{
    public static void Main()
    {
        Drawable d;
        d = new Rectangle();
        d.draw();
        d = new Circle();
        d.draw();
    }
}
```

Output:

```
drawing ractangle...
drawing circle...
```

C# Exception Handling

Exception Handling in C# is *a process to handle runtime errors*. We perform exception handling so that normal flow of the application can be maintained even after runtime errors.

In C#, exception is an event or object which is thrown at runtime. All exceptions the derived from *System.Exception* class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It *maintains the normal flow* of the application. In such case, rest of the code is executed event after exception.

C# Exception Classes

All the exception classes in C# are derived from **System.Exception** class. Let's see the list of C# common exception classes.

Exception	Description
System.DivideByZeroException	handles the error generated by dividing a number with zero.
System.NullReferenceException	handles the error generated by referencing the null object.
System.InvalidCastException	handles the error generated by invalid typecasting.
System.IO.IOException	handles the Input Output errors.
System.FieldAccessException	handles the error generated by invalid private or protected field access.

C# Exception Handling Keywords

In C#, we use 4 keywords to perform exception handling:

- try
- catch
- finally, and
- throw

C# try/catch

In C# programming, exception handling is performed by try/catch statement. The **try block** in C# is used to place the code that may throw exception. The **catch block** is used to handled the exception. The catch block must be preceded by try block.

C# example without try/catch

```
using System;
public class ExExample
{
  public static void Main(string[] args)
    {
       int a = 10;
```

```
        int b = 0;
        int x = a/b;
        Console.WriteLine("Rest of the code");
    }
}
```

Output:

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
```

C# try/catch example

```
using System;
public class ExExample
{
    public static void Main(string[] args)
    {
        try
        {
            int a = 10;
            int b = 0;
            int x = a / b;
        }
        catch (Exception e)
         {
            Console.WriteLine(e);
          }
        Console.WriteLine("Rest of the code");
    }
}
```

Output:

```
System.DivideByZeroException: Attempted to divide by zero.

Rest of the code
```

The throw keyword

The `throw` statement allows you to create a custom error.

The `throw` statement is used together with an **exception class**. There are many exception classes available in C#: `ArithmeticException`, `FileNotFoundException`, `IndexOutOfRangeException`, `TimeOutException`, etc:

Example

```
static void checkAge(int age)
```

```
{
  if (age < 18)
  {
    throw new ArithmeticException("Access denied - You must be at least 18 years old.");
  }
  else
  {
    Console.WriteLine("Access granted - You are old enough!");
  }
}
static void Main(string[] args)
{
  checkAge(15);
}
```

The error message displayed in the program will be:

System.ArithmeticException: 'Access denied - You must be at least 18 years old.'

If `age` was 20, you would **not** get an exception:

C# finally

C# finally block is used to execute important code which is to be executed whether exception is handled or not. It must be preceded by catch or try block.

C# finally example if exception is handled

```
using System;
public class ExExample
{
   public static void Main(string[] args)
   {
      try
      {
         int a = 10;
         int b = 0;
         int x = a / b;
```

```
        }
        catch (Exception e) { Console.WriteLine(e); }
        finally { Console.WriteLine("Finally block is executed"); }
        Console.WriteLine("Rest of the code");
    }
}
```

Output:

```
System.DivideByZeroException: Attempted to divide by zero.
Finally block is executed
Rest of the code
```