

Introduction to C# and .NET Framework

What is C#?

- C# (C-Sharp) is an object oriented programming language developed by Microsoft that runs on the .NET Framework.
- C# is used to develop web apps, desktop apps, mobile apps, games and much more.
- C# has roots from the C programming, and the language is close to other popular languages like C++ and Java.

.NET Framework

- The .NET Framework is a software framework developed by Microsoft, which is free, cross platform and open source developer platform.
- Central to the .NET Framework is its runtime execution environment, known as the Common Language Runtime (CLR) or the .NET runtime.
- C# language is intended for use with .NET.

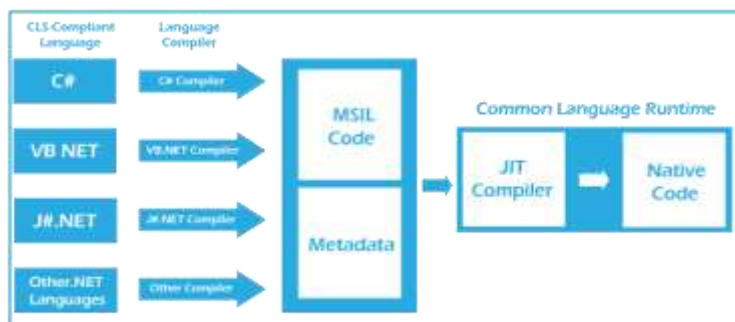
C# and .NET

- C# program can be executed in .NET environment by CLR.
- Before it can be executed by the CLR, it needs to be compiled as given below.
 - Compilation of source code to Microsoft Intermediate Language (IL).
 - Compilation of IL to platform-specific code by the CLR.

Common Language Runtime (CLR)

- .NET provides a run-time environment called the common language runtime(CLR) that runs the code and provides services that make the development process easier.
- .NET CLR is a runtime environment that manages and executes the code written in any .NET programming language.
- CLR is the virtual machine component of the .NET framework.
 - First, the C# code is compiled into MSIL (Microsoft intermediate language code) which is a platform independent code.
 - Second, the MSIL code is interpreted using CLR with respect to the native machine.

Converting Source Code into Native Code



Execution of a .NET Application

Important Features of MSIL

- **Support for Object Orientation and Interfaces.**
 - It support classic object-oriented programming, with single implementation of **inheritance** of classes.
 - In addition to classic object-oriented programming, MSIL also brings in the idea of **interfaces**,
- **Strong Data Typing as a Key to Language Interoperability.**
 - One important aspect of IL is that it is based on exceptionally strong data typing.
 - That means that all variables are clearly marked as being of a particular, specific data type.
 - Primitive data types such as int, double, char, bool, long, etc
 - Reference types
- **Garbage Collection**
 - The garbage collector is used for reallocation of unused memory of an application.
 - The .NET runtime relies on the garbage collector instead.
 - The purpose of this program is to clean up memory.
- **Security**
 - .NET can excel in terms of complementing the security mechanisms provided by Windows because it can offer **code-based security**, whereas Windows offers only role-based security.
- **Error Handling with Exceptions**
 - The .NET Framework is designed to facilitate handling of error conditions using the same mechanism based on exceptions that is employed by Java and C++.
- **Use of attributes**
 - Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, methods and classes to a program.
 - The .Net Framework provides two types of attributes: the **pre-defined attributes** and **custom built attributes**.

Assemblies in .NET Framework

- An assembly is the logical unit that contains compiled code targeted at the .NET Framework.
- Assemblies take the form of executable (.exe) or dynamic link library (. DLL) files, and are the building blocks of .NET applications.
- They provide the common language runtime with the information it needs to be aware of type implementations.
- Types of Assemblies
 - Private Assembly
 - Private assemblies are the simplest type. They normally ship with software and are intended to be used only with that software.
 - Shared Assemblies
 - Shared assemblies are intended to be common libraries that any other application can use.

Reflection

- In .NET assemblies developer stores metadata, including details of all the types and members of these types defined in the assembly, we can access this metadata programmatically using **Reflection**.

Parallel Programming

- The .NET Framework enables you to take advantage of all the multicore processors available today.
- The parallel computing capabilities provide the means to separate work actions and run these across multiple processors.
- The parallel programming APIs available now writing safe **multithreaded** code simple.

Asynchronous Programming

- You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming.
- C# supports simplified approach, **async** programming, that leverages asynchronous support in the .NET runtime.
- The **async** and **await** keywords in C# are the heart of **async** programming.
 - By using those two keywords, you can use resources in .NET Framework, .NET Core, or the Windows Runtime to create an asynchronous methods.

Creating .NET Applications Using C#

You can develop the following .NET applications using C#

- Console Application
- Windows Desktop Application
 - Windows Forms
 - Windows Presentation Foundation
- ASP.Net application
 - ASP.NET Web Forms
 - ASP.NET MVC

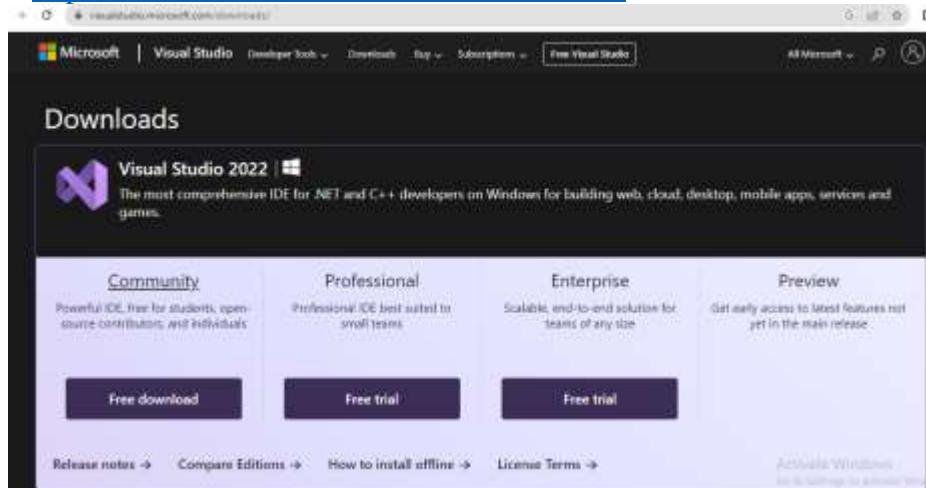
Creating Console Application

You can also use C# to create console applications: text-only applications that run in a DOS window. You can probably use console applications when unit testing class libraries and for creating UNIX or Linux daemon processes.

Installation of Software:

- Visual Studio 2022: The most comprehensive IDE for .NET and C++ developers on Windows for building web, cloud, desktop, mobile apps, services and games.

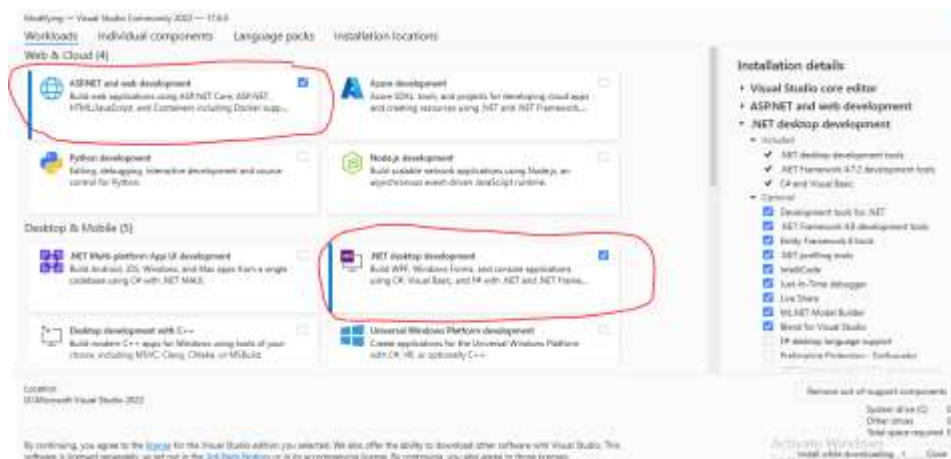
<https://visualstudio.microsoft.com/downloads/>



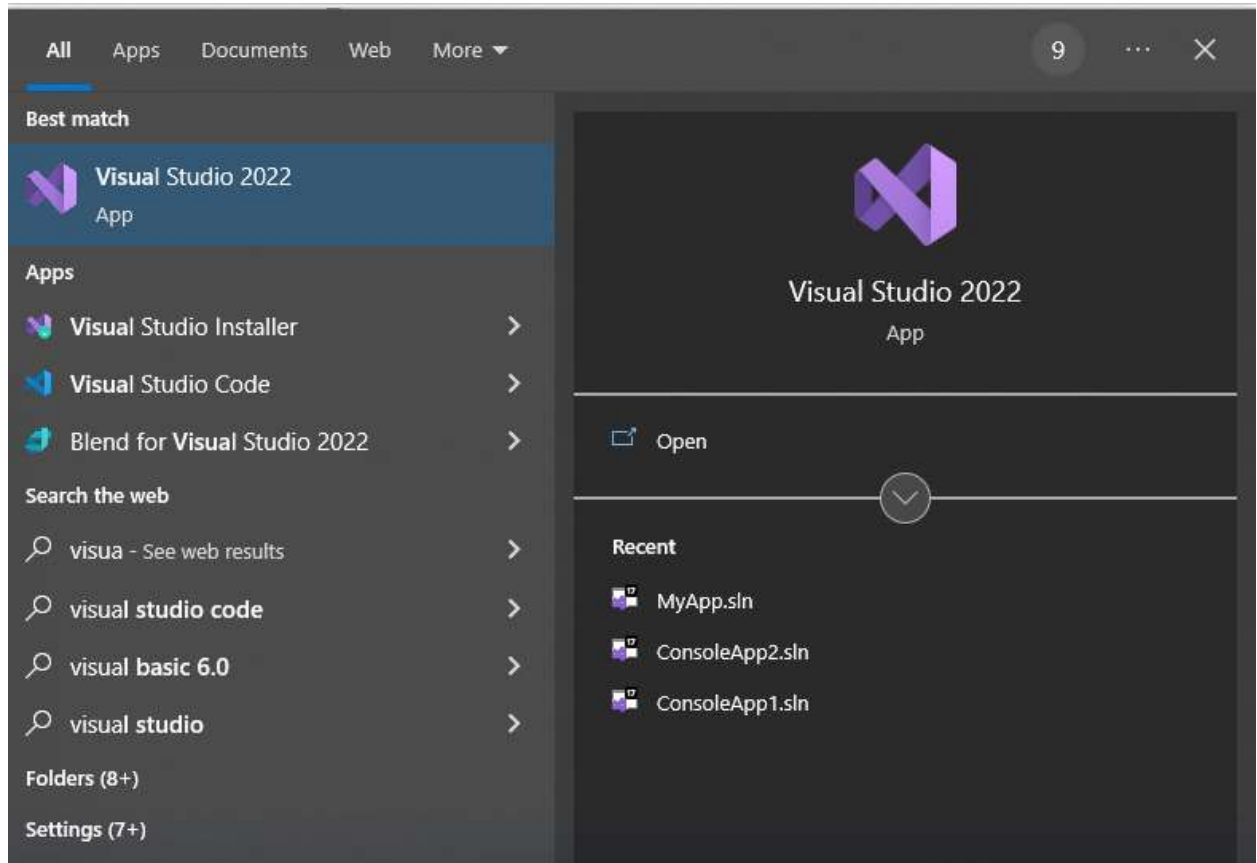
- Select community version of Visual Studio 2022 and download.
- After download is completed launch visual studio installer.



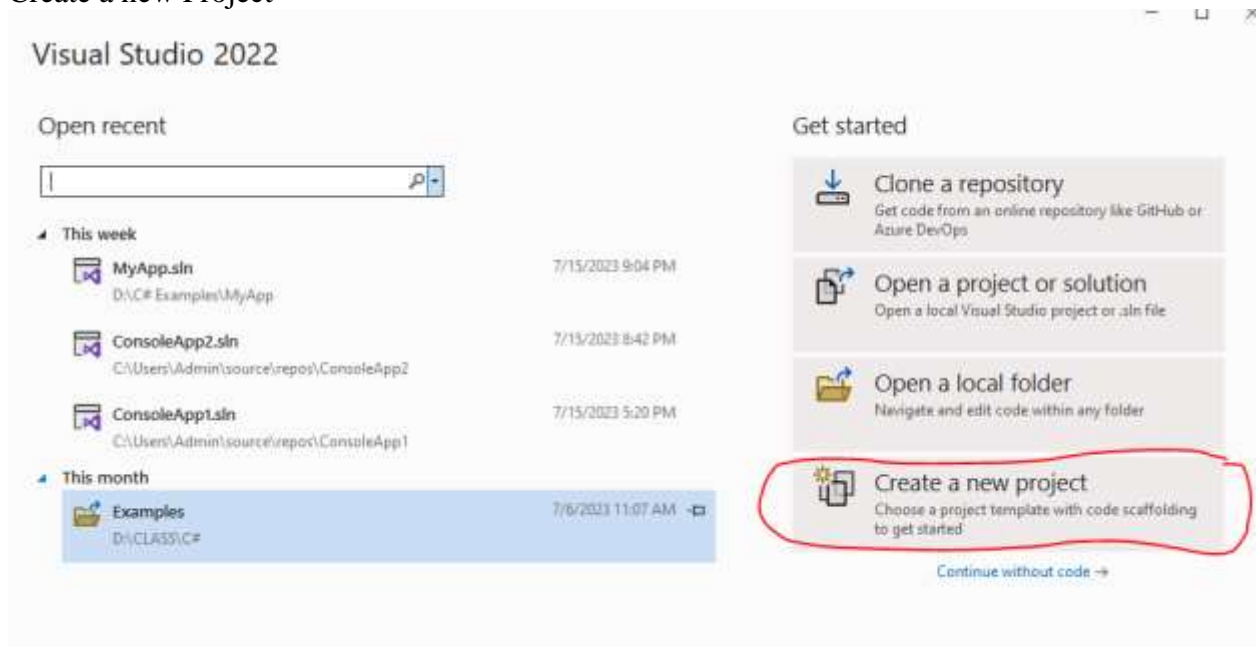
- Select components to install in current version
 - Select ASP.Net and Web Development, .Net Desktop Environment and then click Install while downloading.



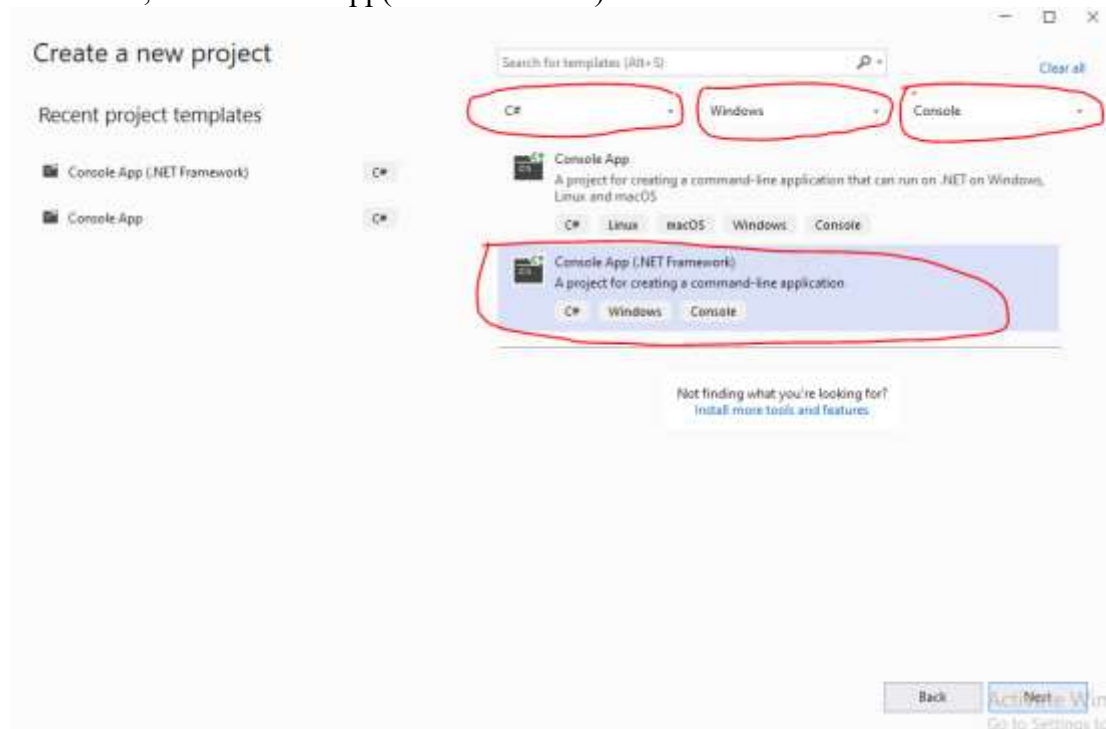
- After installation, open Visual Studio 2022



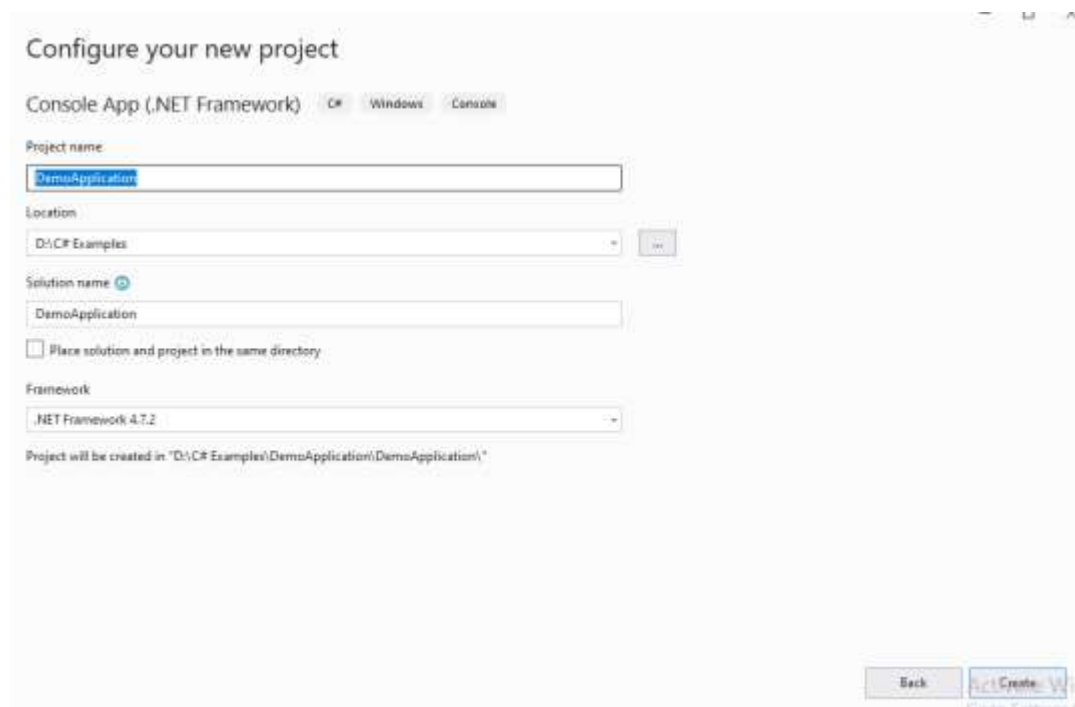
- Create a new Project



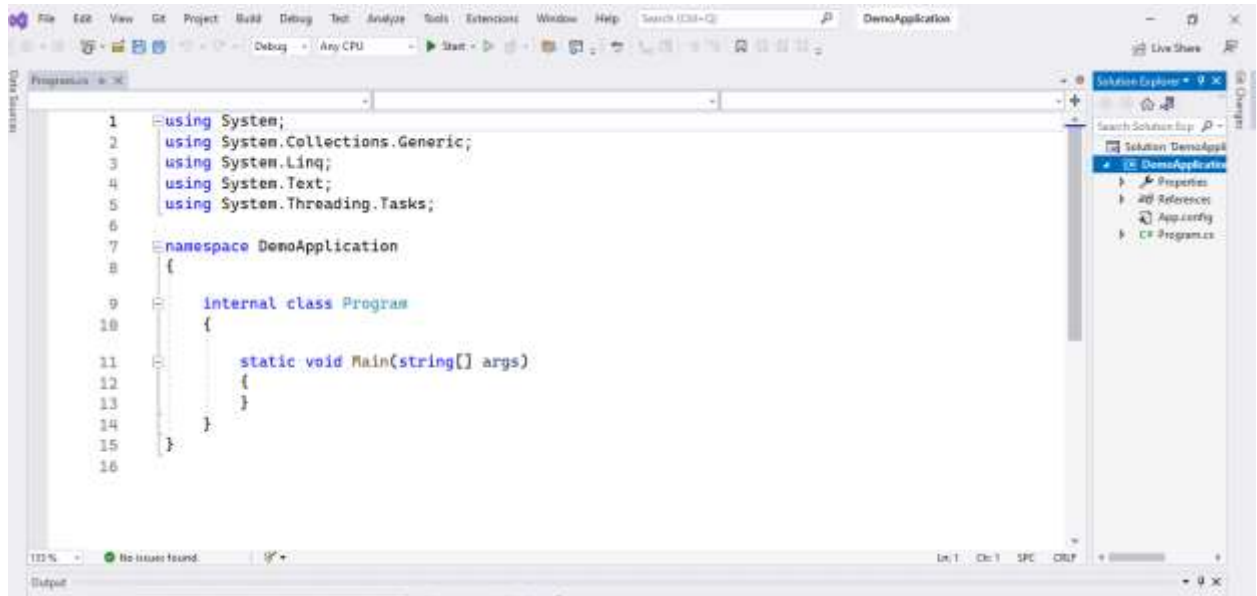
- Select the programming language as C#, platform as “Windows” , the project type as “Console”, and Console App(.Net framework) then click “Next”.



- Configure the project name, location and click “Create”

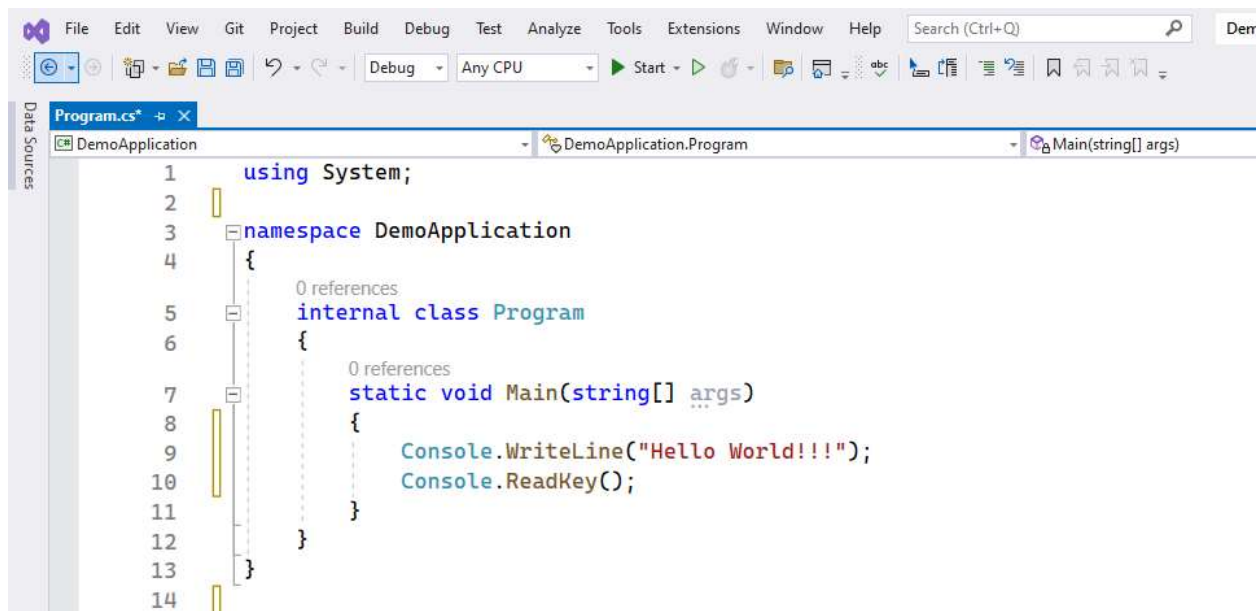


- After creating project, the default C# program will be generated as given below.



- Execute the program with simple print statement. Include the following code to test program.

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



Click on Start



Sample Output



Compiling and Running the Program using Command Prompt

- You can compile this program by simply running the C# command-line compiler (csc.exe) against the source file.
- Compilation
`csc Program.cs`
- Execution
`Program.exe`

Closer look at the program

- The first few lines in the previous code example are related to **namespace** which is a way to group together associated classes.
- The **namespace** keyword declares the namespace with which your class should be associated (optional).

Example:

```
namespace DemoApplication
{
}
```

- The **using** statement specifies a **namespace** that the compiler should look at to find any predefined classes.

Example:

```
using System;
```

```
// include System class to use I/O functions such as Console.WriteLine();
```


- Next, you declare a class called **Program**. All C# code must be contained within a class. The class declaration consists of the class **keyword**, followed by the class name and a pair of curly braces.

Example:

```
class Program
{
}
```

- Next, you declare a method called **Main()**. Every C# executable (such as console applications, Windows applications, and Windows services) must have an entry point – **Main()** method.

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

- Display data in output console window
 - **Console.WriteLine()** - to write a line of text to the console window with new line.
 - **Console.Write()** - to write a line of text to the console window without new line.
 - **Console.ReadKey()** - Adding this line forces the application to wait for the carriage-return key to be pressed before the application exits
-

Variables in C#

- We can declare variables in C# using the following syntax:

datatype identifier;

For example:

int number;

- **Initialization of Variables:**

- The variables must be initialized before it is used

- **Example:-**

```
using System;
class Program {
    static void Main(string args[]) {
        int a = 10;
        float pi = 3.14f;
        bool flag = false;

        Console.WriteLine("a = " + a);
        Console.WriteLine("pi = " + pi);
        Console.WriteLine("flag = " + flag);
    }
}
```

- **Type Inference:**

- Type inference makes use of the **var** keyword.
- The compiler “**infers**” what the type of the variable is by what the variable is initialized to.

- **Example:**

```
using System;
class Program {
    static void Main(string args[]) {
        var a = 10;
        var pi = 3.14f;
        var flag = false;

        Console.WriteLine("a = " + a);
        Console.WriteLine("pi = " + pi);
        Console.WriteLine("flag = " + flag);
    }
}
```

- There are a few rules that you need to follow:
 - The variable must be initialized. Otherwise, the compiler doesn't have anything from which to infer the type.
 - The initializer cannot be null.
 - The initializer must be an expression.
 - You can't set the initializer to an object unless you create a new object in the initializer.

➤ **Variable Scope:**

- The **scope** of a variable is the region of code from which the variable can be accessed.
- Rules:
 - A local variable is in scope until a closing brace indicates the end of the block statement or method in which it was declared.
 - A local variable that is declared in a **for**, **while**, or similar statement is in scope in the body of that loop.
 - A field (also known as a member variable) of a class is in scope for as long as its containing class is in scope.

- Example

```
using System;
class Program {
    static void Main(string args[]) {
        for(int i = 0; i <= 5; i++)
        {
            Console.WriteLine(i);
        }
        Console.WriteLine(i);    // i cannot access
    }
}
```

➤ **Constants:**

- As the name implies, a **constant** is a variable whose **value cannot be changed** throughout its lifetime.
- **const** keyword used for creating constant in C#.
- Example:

```
using System;
class Program
{
    static void Main(string args[])
    {
        const float pi = 3.14f;
        pi = 2.14;    // Error!!!
    }
}
```

Predefined Data Types

There are two types of predefined data types available in C# as given below. These types are stored in different places in memory.

- Value Types
 - stores its value directly
 - **stack** memory

- Reference Types
 - stores a reference to the value
 - **heap** memory

Value Types

➤ Integer types

NAME	CTS TYPE	DESCRIPTION	RANGE (MIN:MAX)
sbyte	System.SByte	8-bit signed integer	-128:127 ($-2^7:2^7-1$)
short	System.Int16	16-bit signed integer	-32,768:32,767 ($-2^{15}:2^{15}-1$)
int	System.Int32	32-bit signed integer	-2,147,483,648:2,147,483,647 ($-2^{31}:2^{31}-1$)
long	System.Int64	64-bit signed integer	-9,223,372,036,854,775,808: 9,223,372,036,854,775,807 ($-2^{63}:2^{63}-1$)
byte	System.Byte	8-bit unsigned integer	0:255 ($0:2^8-1$)
ushort	System.UInt16	16-bit unsigned integer	0:65,535 ($0:2^{16}-1$)
uint	System.UInt32	32-bit unsigned integer	0:4,294,967,295 ($0:2^{32}-1$)
ulong	System.UInt64	64-bit unsigned integer	0:18,446,744,073,709,551,615 ($0:2^{64}-1$)

➤ Floating-Point Types

NAME	CTS TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (APPROXIMATE)
float	System.Single	32-bit, single-precision floating point	7	$\pm 1.5 \times 10^{245}$ to $\pm 3.4 \times 10^{38}$
double	System.Double	64-bit, double-precision floating point	15/16	$\pm 5.0 \times 10^{2324}$ to $\pm 1.7 \times 10^{308}$

➤ Decimal Type

- The decimal type represents higher-precision floating-point numbers

NAME	CTS TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (APPROXIMATE)
decimal	System.Decimal	128-bit, high-precision decimal notation	28	$\pm 1.0 \times 10^{228}$ to $\pm 7.9 \times 10^{28}$

➤ Boolean Type

NAME	CTS TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (APPROXIMATE)
bool	System.Boolean	Represents true or false	NA	true or false

➤ Character Type

NAME	CTS TYPE	VALUES
char	System.Char	Represents a single 16-bit (Unicode) character

Predefined Reference Types

C# supports two predefined reference types, object and string,

NAME	CTS TYPE	DESCRIPTION
object	System.Object	The root type. All other types (including value types) in the CTS are derived from object.
string	System.String	Unicode character string

- object
 - It is a super class of all reference types which provides basic functions such as Equals(), ToString().
- string
 - Used to represents sequence of characters called “String”.
 - Example:

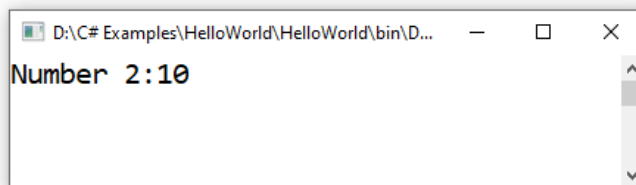
```
using System;
class Program {
    static void Main(string []args)
    {
        string firstname = "John";
        string lastname = "Smith";
        string name = firstname + lastname;
        Console.WriteLine(name);
    }
}
```

Type Casting

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

Example:

```
using System;
class Program {
    static void Main(string[] args){
        int number1= 10;
        double number2 = number1;
        Console.WriteLine("Number 2:" + number2);
        Console.ReadKey();
    }
}
```



Here, number1 **int** data type is automatically converted to **double** since it is smaller type than number2.

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

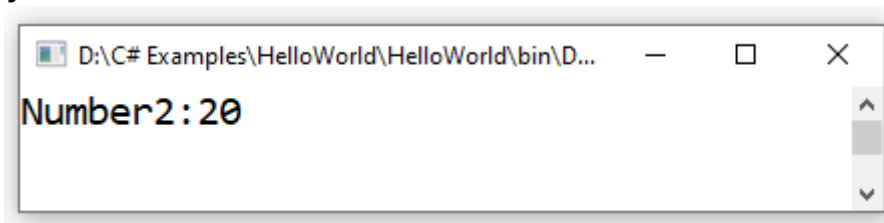
Example:

```
using System;
class Program {
    static void Main(string[] args) {
        double data1= 20.5;
        int data2 = data1; //Error
        Console.WriteLine("Number2:" + data2);
        Console.ReadKey();
    }
}
```

Here, the code `int data2 = data1;` generates compiler error since the bigger data type is assigned to smaller data type.

Apply Explicit Type Casting..

```
class Program {
    static void Main(string[] args) {
        double data1= 20.5;
        int data2 = (int)data1;
        Console.WriteLine("Number2:" + data2);
        Console.ReadKey();
    }
}
```



Type Conversion Methods:

It is also possible to convert data types explicitly by using built-in methods

- Convert.ToBoolean
- Convert.ToDouble
- Convert.ToString
- Convert.ToInt32(int)
- Convert.ToInt64 (long)

Example

```
using System;
class Program {
    static void Main(string[] args)
```

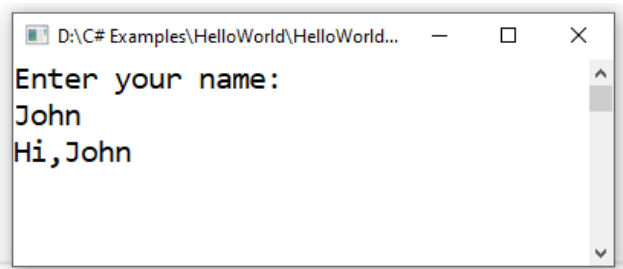
```
{  
    int data=20;  
    double number = Convert.ToDouble(data);  
    Console.WriteLine("Number:" + number);  
    Console.ReadKey();  
}  
}
```

Get User Input

The `Console.ReadLine()` method used to get user input from keyboard.

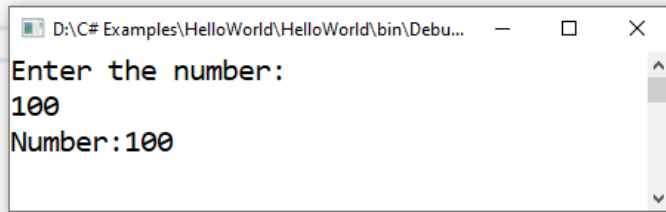
Example 1: How to store your name(string)?

```
using System;  
class Program  
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine("Enter your name: ");  
        string name = Console.ReadLine();  
        Console.WriteLine("Hi," + name);  
        Console.ReadKey();  
    }  
}
```



Example 2: How to get **Number** as input?

```
using System;  
class Program  
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine("Enter the number: ");  
        int num = Convert.ToInt32(Console.ReadLine());  
        Console.WriteLine("Number:"+num);  
        Console.ReadKey();  
    }  
}
```

Operators

- Operators are used to perform mathematical operations on variables and values.
- Types:
 - Arithmetic Operators: +, -, *, /, %, ++, --
 - Assignment Operators: =, +=, -=, *=, /=, %=
 - Comparison Operators: ==, !=, >, >=, <, <=
 - Logical Operators: &&, ||, !

String Concatenation

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

Example:

```
using System;
class Program {
    static void Main(string[] args) {
        string firstname = "John";
        string lastname = "Smith";
        Console.WriteLine("Full Name:" + firstname + " " + lastname);
        Console.ReadKey();
    }
}
```



Control Statements

Conditions and If Statements

➤ **if** and **else** Statement

- **if** statement to specify a block of C# code to be executed if a condition is **true**.
- **else** statement to specify a block of code to be executed if the condition is **false**.

Example: C# Program to check the given number is odd or even.

```
using System;
class Program {
    static void Main(string[] args) {
        Console.WriteLine("Enter the number: ");
        int num = Convert.ToInt32(Console.ReadLine());
        if(num%2 == 0) {
            Console.WriteLine("Even");
        }
        else{
            Console.WriteLine("Odd");
        }
        Console.ReadKey();
    }
}
```



➤ **if** and **else if** Statement

- **else if** statement to specify a new condition if the first condition is **false**.

Example: C# program to check the given number is positive or negative or zero.

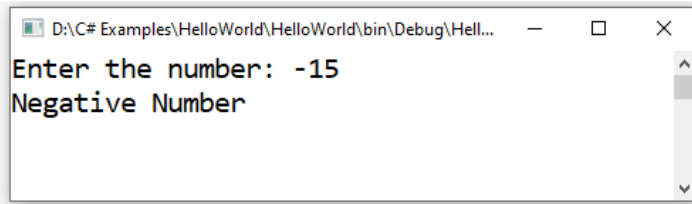
```
using System;
class Program {
    static void Main(string[] args) {
        Console.Write("Enter the number: ");
        int num = Convert.ToInt32(Console.ReadLine());
        if(num >= 0) {
            Console.WriteLine("Positive Number");
        }
        else if(num<0){
            Console.WriteLine("Negative Number");
        }
        else
        {

```

```

        Console.WriteLine("Zero");
    }
    Console.ReadKey();
}

```



Switch Statements

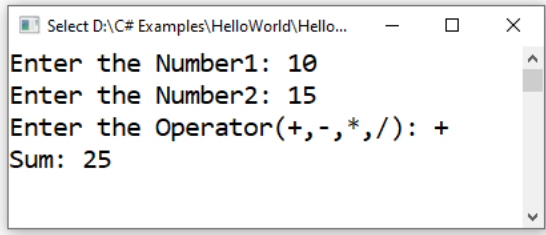
- The **switch** statement to select one of many code blocks to be executed.
- The **break** statement is used in every case to stop the fall through execution and it is **mandatory** in C#.

- **Example:** Simple Calculator Program

```

using System;
class Program {
    static void Main(string[] args) {
        Console.Write("Enter the Number1: ");
        int num1 = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter the Number2: ");
        int num2 = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter the Operator(+,-,*,/): ");
        int op = Console.Read();
        switch (op)
        {
            case '+':
                Console.WriteLine("Sum: " + (num1+num2));
                break;
            case '-':
                Console.WriteLine("Difference: " + (num1 - num2));
                break;
            case '*':
                Console.WriteLine("Product: " + (num1 * num2));
                break;
            case '/':
                Console.WriteLine("Division: " + (num1 / num2));
                break;
            default:
                Console.WriteLine("Invalid Operator");
                break;
        }
        Console.ReadKey();
    }
}

```

```
}  
}  
  

```

Looping Statements

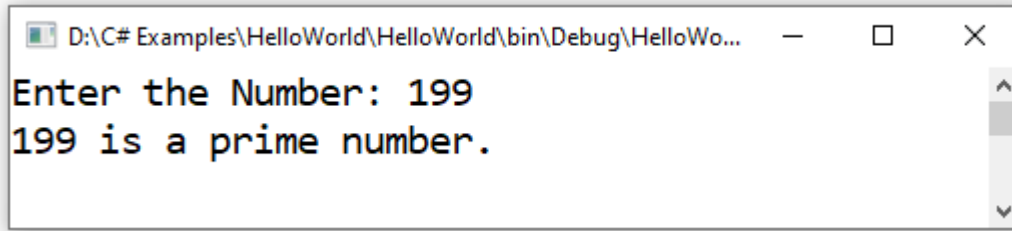
Loops can execute a block of code as long as a specified condition is reached.

➤ **for Loop**

- **for loop** is used to repeat the execution of a logic or set of statements.
- When you know exactly how many times you want to loop through a block of code, use the **for loop**.

Example: Program to Check the Prime Number using a for loop

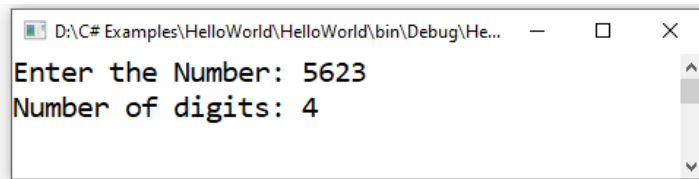
```
using System;  
class Program {  
    static void Main(string[] args) {  
        Console.Write("Enter the Number: ");  
        int num = Convert.ToInt32(Console.ReadLine());  
        bool flag = false;  
        for (int i = 2; i <= num / 2; ++i)  
        {  
            if (num % i == 0)  
            {  
                flag = true;  
                break;  
            }  
        }  
        if (!flag)  
            Console.WriteLine(num + " is a prime number.");  
        else  
            Console.WriteLine(num + " is not a prime number.");  
        Console.ReadKey();  
    }  
}
```



➤ **while Loop**

- The **while** loop loops through a block of code as long as a specified condition is true.
- **Example:** Count Number of Digits in an Integer using while loop

```
using System;
class Program {
    static void Main(string[] args) {
        Console.Write("Enter the Number: ");
        int num = Convert.ToInt32(Console.ReadLine());
        int count = 0;
        while (num != 0)
        {
            num /= 10;
            ++count;
        }
        Console.WriteLine("Number of digits: " + count);
        Console.ReadKey();
    }
}
```



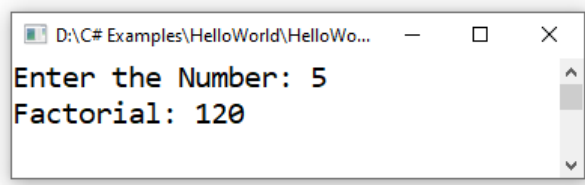
➤ **do while Loop**

- The **do while** 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:** Program to find the factorial of a given number using **do while** loop.

```
class Program {
    static void Main(string[] args) {
        Console.Write("Enter the Number: ");
        int num = Convert.ToInt32(Console.ReadLine());
        int fact = 1, i = 1;
        do
        {
```

```
        fact *= i;
        i++;
    } while (i <= num);

    Console.WriteLine("Factorial: " + fact);
    Console.ReadKey();
}
}
```



Arrays in C#

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

Create an Array

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

Example: To store set of numbers

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

Example: To store alphabetical letters

```
char[] letters = { 'a', 'b', 'c', 'd', 'e', 'f' };
```

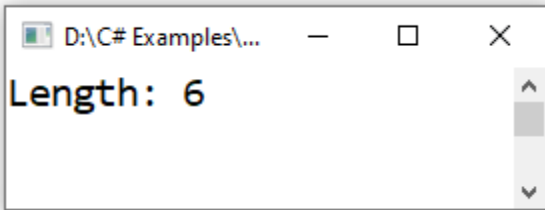
Example: To store collection of names

```
string[] names = { "John", "David", "Anish", "Victor", "Alban" };
```

Length of the Array:

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

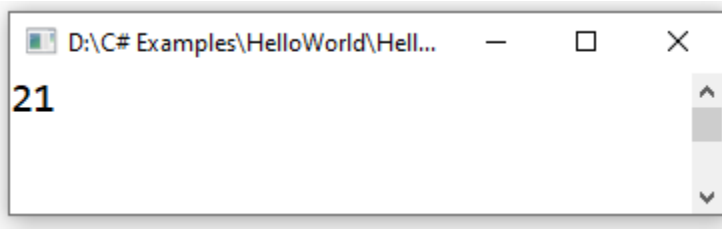
```
using System;
class Program {
    static void Main(string[] args) {
        int[] nums = { 11, 21, 13, 40, 15, 60 };
        Console.WriteLine("Length: " + nums.Length);
        Console.ReadKey();
    }
}
```



Accessing Array Elements:

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

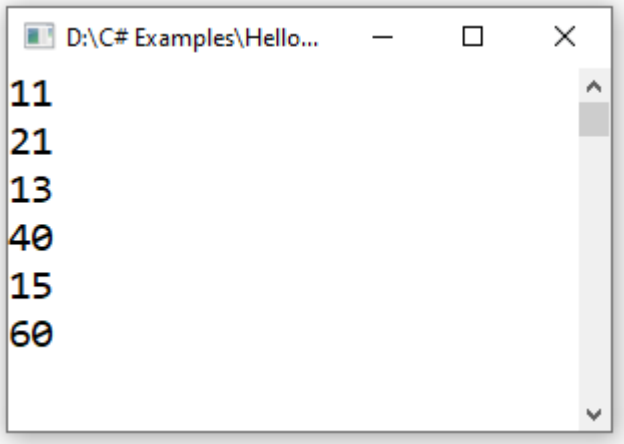
```
using System;
class Program {
    static void Main(string[] args) {
        int[] nums = { 11, 21, 13, 40, 15, 60 };
        Console.WriteLine(nums[1]);           //To print 21
        Console.ReadKey();
    }
}
```



To Access all the elements of an array

We can access all the elements of an array using for loop

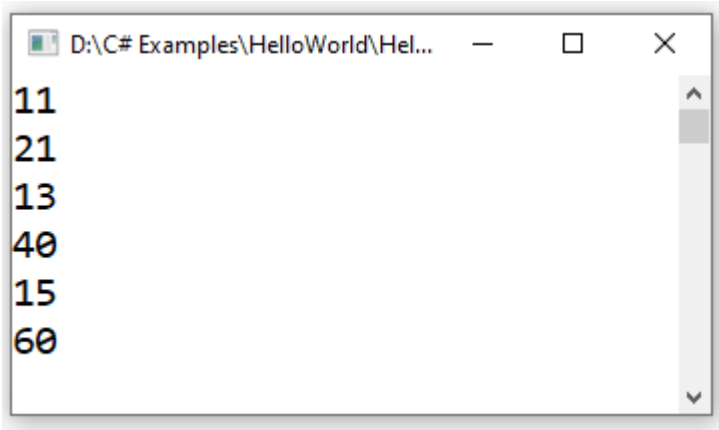
```
using System;
class Program {
    static void Main(string[] args) {
        int[] nums = { 11, 21, 13, 40, 15, 60 };
        for(int i = 0; i < nums.Length; i++)
        {
            Console.WriteLine(nums[i]);
        }
        Console.ReadKey();
    }
}
```

The **foreach** Loop

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

```
using System;
class Program {
    static void Main(string[] args) {
        int[] nums = { 11, 21, 13, 40, 15, 60 };
        foreach(int i in nums){
            Console.WriteLine(i);
        }
        Console.ReadKey();
    }
}
```

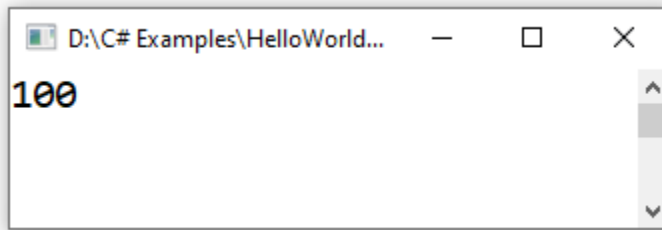


Change the value in an array

To change the value of a specific element, refer to the index number and assign new value.

```
using System;
class Program {
```

```
static void Main(string[] args) {  
    int[] nums = { 11, 21, 13, 40, 15, 60 };  
    nums[1] = 100; // replace 21 by 100  
    Console.WriteLine(nums[1]);  
    Console.ReadKey();  
}
```



Create an Empty Array:-

You can create an empty array using **new** operator and the values can be stored dynamically.

Example: To create an empty to store 5 numbers as given below

```
int[] nums = new int[5];
```

Here, the **new** operator creates an array object and allocate memory for five integer numbers.

Store data in an Empty Array

```
using System;  
class Program {  
    static void Main(string[] args) {  
        Console.Write("Enter the array size: ");  
        int size = Convert.ToInt32(Console.ReadLine());  
        int[] nums = new int[size];  
        Console.WriteLine("Enter the Numbers:-");  
        for(int i = 0; i < nums.Length; i++){  
            nums[i] = Convert.ToInt32(Console.ReadLine());  
        }  
        //to display  
        foreach(int i in nums) {  
            Console.WriteLine(i);  
        }  
        Console.ReadKey();  
    }  
}
```

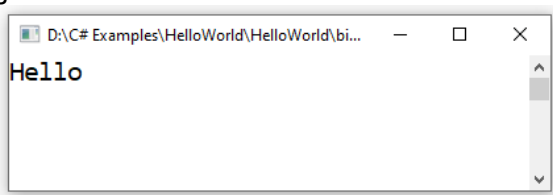
Methods in C#

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**.

Create Method in C#

A method is defined with the name of the method, followed by parenthesis().

```
using System;
class Program {
    static void sayHello()                //method definition
    {
        Console.WriteLine("Hello");
    }
    static void Main(string[] args) {
        sayHello();                      //calling a method
        Console.ReadKey();
    }
}
```



Notes:

- **sayHello()** 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.

Method Parameters and Arguments

The **data** can be passed to methods as **parameters**. The parameter act as a **variable** 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.

Example:

```
using System;
class Program {
    static void sayHello(string name)    //'name' is parameter
    {
        Console.WriteLine("Hello, " + name);
    }
    static void Main(string[] args) {
        string name = "John";
    }
}
```

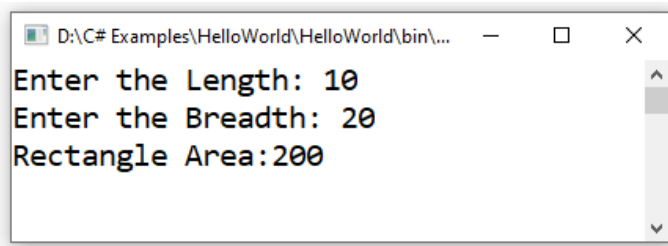
```
        sayHello(name);                                // 'name' is argument
        Console.ReadKey();
    }
}
```

Multiple Parameters and Arguments

You can have as many parameters as you like, just separate them with commas:

Example:-

```
using System;
class Program {
    static void findArea(int length,int breadth)
    {
        Console.WriteLine("Rectangle Area:" + (length*breadth));
    }
    static void Main(string[] args) {
        Console.Write("Enter the Length: ");
        int l = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter the Breadth: ");
        int b = Convert.ToInt32(Console.ReadLine());
        findArea(l,b);
        Console.ReadKey();
    }
}
```



Return Values

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: Program to reverse a given number using method

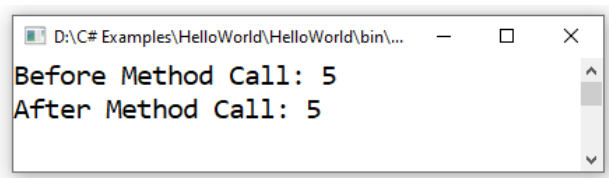
```
using System;
class Program {
    static int reverse(int number)
    {
        int rev = 0;
        while (number != 0)
        {
```

```
        int remainder = number % 10;
        rev = rev * 10 + remainder;
        number = number / 10;
    }
    return rev;
}
static void Main(string[] args) {
    Console.Write("Enter the Length: ");
    int no = Convert.ToInt32(Console.ReadLine());
    int result=reverse(no);
    Console.WriteLine("Reversed Number: " + result);
    Console.ReadKey();
}
}
```

Call By Value

Value type parameters are that pass a copy of original value to the function rather than reference. It does not modify the original value. A change made in passed value does not alter the actual value.

```
using System;
class Program {
    static void change(int a){
        a = a + 10;
    }
    static void Main(string[] args) {
        int a = 5;
        Console.WriteLine("Before Method Call: " + a);
        change(a);
        Console.WriteLine("After Method Call: " + a);
        Console.ReadKey();
    }
}
```

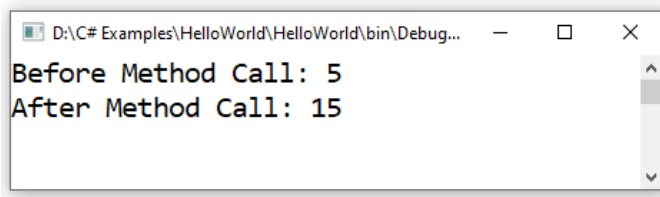


Here, the data 'a' is passed as value to the method **change(int a)** where 'a' is updated to 15. However, the original value of a does not change due to call by value.

Call By Reference

C# provides a **ref** keyword to pass argument as reference-type. It passes reference of arguments to the function rather than copy of original value. The changes in passed values are permanent and **modify** the original variable value.

```
using System;
class Program {
    static void change(ref int a){
        a = a + 10;
    }
    static void Main(string[] args) {
        int a = 5;
        Console.WriteLine("Before Method Call: " + a);
        change(ref a);
        Console.WriteLine("After Method Call: " + a);
        Console.ReadKey();
    }
}
```



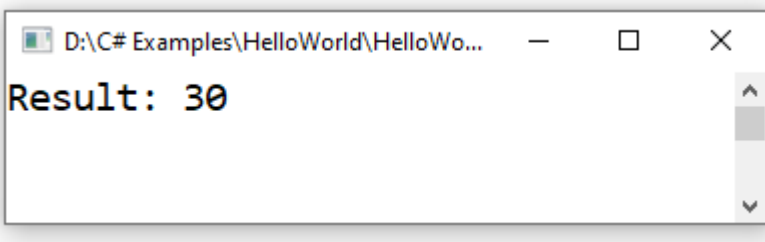
Here, due to **ref** keyword the address of a variable 'a' is passed to the method **change(ref int a)** through the method call **change(ref a)** which modifies the original value of 'a'.

The out Parameter in C#

C# provides **out** keyword to pass arguments as out-type. It is like reference-type, except that it does not require variable to initialize before passing. It is useful when we want a function to return multiple values.

```
using System;
class Program {
    static void sum(int a, int b, out int result){
        result = a + b;
    }
    static void Main(string[] args) {
        int result;
        int a = 10;
        int b = 20;
        sum(a, b, out result);
        Console.WriteLine("Result: " + result);
        Console.ReadKey();
    }
}
```

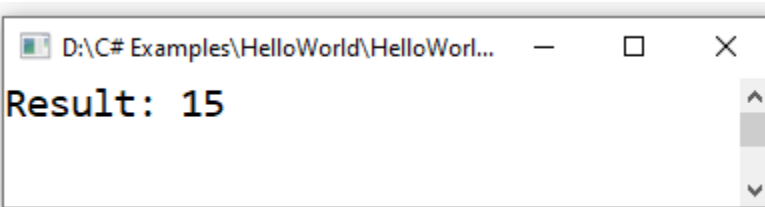
```
}  
}
```



Passing Array to Function

We can also pass array collection to function as arguments

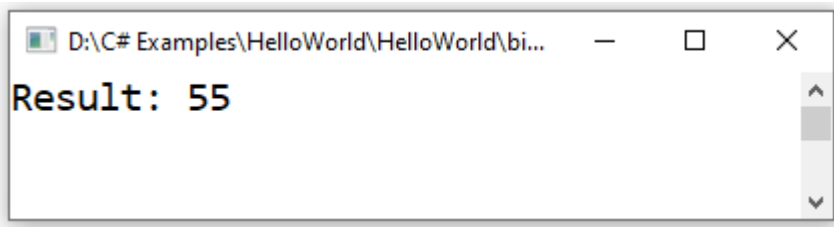
```
using System;  
class Program {  
    static void sum(int[] numbers,out int result){  
        int temp=0;  
        foreach(int i in numbers)  
        {  
            temp += i;  
        }  
        result = temp;  
    }  
    static void Main(string[] args) {  
        int result;  
        int[] data = { 1, 2, 3, 4, 5 };  
        sum(data,out result);  
        Console.WriteLine("Result: " + result);  
        Console.ReadKey();  
    }  
}
```



The **params** Keyword in C#

In C#, **params** is a keyword which is used to specify a parameter that takes variable number of arguments. It is useful when we don't know the number of arguments prior. Only one params keyword is allowed and no additional parameter is permitted after params keyword in a function declaration.


```
using System;
class Program {
    static int add(params int[] numbers){
        int sum=0;
        foreach(int i in numbers)
        {
            sum += i;
        }
        return sum;
    }
    static void Main(string[] args) {
        int result = add(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        Console.WriteLine("Result: " + result);
        Console.ReadKey();
    }
}
```



Method Overloading

With method overloading, multiple methods can have the same name with different parameters:

Example:

```
int MyMethod(int x)
float MyMethod(float x)
double MyMethod(double x, double y)
```

Example: To find sum of two numbers with different types using method overloading.

```
using System;
public class Program {
    static int sum(int x, int y) {
        return x + y;
    }
    static double sum(double x, double y){
        return x + y;
    }
    static double sum(int x, short y) {
        return x + y;
    }
    static void Main(string[] args) {
        int result1 = sum(8, 5);
    }
}
```

```
        double result2 = sum(4.3, 6.26);  
        double result3 = sum(400, 2);  
        Console.WriteLine("Result 1: " + result1);  
        Console.WriteLine("Result 2: " + result2);  
        Console.WriteLine("Result 3: " + result3);  
        Console.ReadKey();  
    }  
}
```

