

Day22 C#



Authorized & published by Summitworks Technologies Inc



Agenda

- Introduction to .Net Framework
 - .Net Framework Architecture
 - .Net components
 - .Net Design principles
- C# and .Net version history
 - .Net Framework version history
 - C# version history
- Setting up the Environment
 - Download and Install Visual Studio
- Datatypes
 - Integer
 - Float
 - Double
 - Boolean
 - String
 - Nullable
- C# Enum and Constant
- Namespace
- StringBuilder
- Boxing and Unboxing
- Pass by Value, by reference and out parameters
- Variables and Operators
 - Arithmetic operator
 - Relational operator
 - Logical operator
 - Assignment operator
 - Expressions
- Conditional Statements
 - IF
 - Loops (While and Do While)
 - Switch
 - Foreach

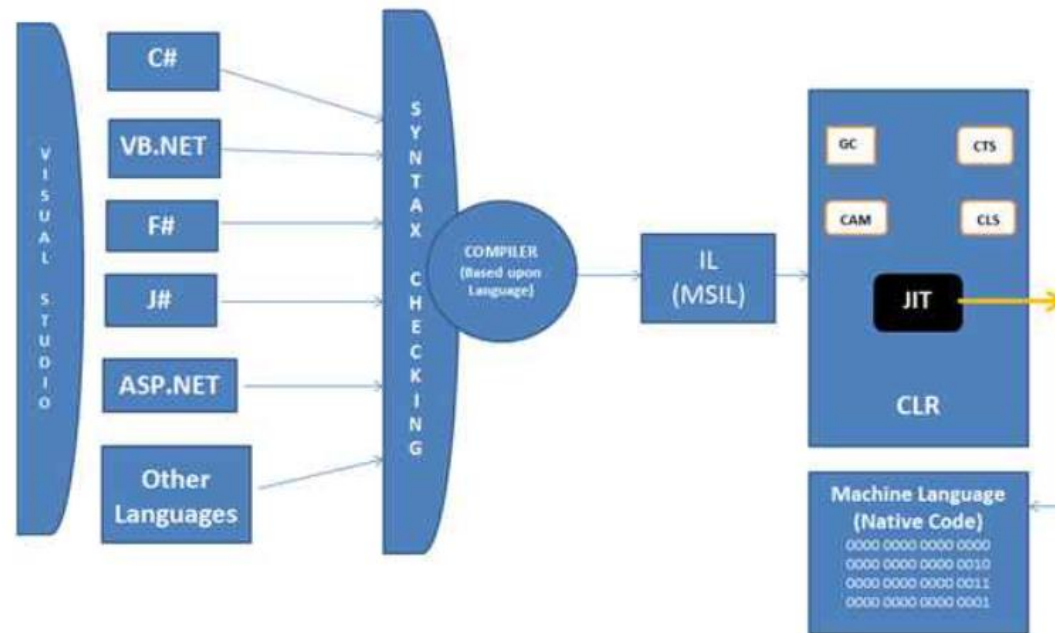
Agenda Cont...

- OOPS Concepts
 - Class and Objects
 - Inheritance
 - Type casting of reference type
 - Static and Dynamic binding
 - Abstract Class
 - Encapsulation
 - Polymorphism
 - Method Overloading and Method Overriding
- Access modifiers and Constructors
- Interface
- Collections
 - Generic Collections
 - Non-Generic Collections
- Delegates
- Anonymous methods and Lambda expressions
- Extension Methods
- More on Classes
 - Partial Class
 - Sealed Class
 - Static Class

Introduction to .Net Framework

.Net Framework Architecture

- The .NET Framework is a class of reusable libraries (collection of classes) given by Microsoft to be used in other .Net applications and to develop, build and deploy many types of applications on the Windows platforms
- Using .NET Framework we can develop Console Applications, Windows Forms, Window Presentation Foundation(WPF), Web Applications, Windows Communication Foundation (WCF)
- Primarily it runs on the Microsoft windows operating system



Introduction to .Net Framework Cont

Compiling a .NET program

What really happens when we compile a .NET program?

- The exe file that is created doesn't contain executable code, rather it's MicroSoft Intermediate Language (MSIL) code
- When you run the EXE, a special runtime environment (the Common Language Runtime or CLR) is launched and the IL instructions are executed by the CLR to the machine language
- The CLR comes up with a Just In Time Compiler that translates the IL to native language the first it is encountered

So the process of programming goes through like:

- We write a program in C#, VB.Net and other languages
- We compile our code to IL code based on the language compiler (csc.exe, vbc.exe and so on)
- Run your IL program that launches the CLR to execute your IL, using its JIT to translate your program into native code as it executes

Introduction to .Net Framework Cont

.NET Framework Components

It mainly contains two components,

Common Language Runtime (CLR)

.Net Framework Class Library

Common Language Runtime(CLR)

- Common Language Runtime (CLR) provides an environment to run all the .Net Programs
- Role of CLR is to locate, load and manage .NET objects
- The code which runs under the CLR is called as Managed Code
- CLR also helps us with memory management
- CLR allocates the memory for scope and de-allocates the memory if the scope is completed
- Language Compilers (e.g. C#, VB.Net, J#) will convert the Code/Program to Microsoft Intermediate Language(MSIL) intern this will be converted to Native Code by CLR

Introduction to .Net Framework Cont

.NET Framework Class Library (FCL)

- Framework Class Library also known as Base Class Library BCL
- BCL defines types that can be used to build any type of software application
- Developers can use BCL for database interactions, reading and writing to file

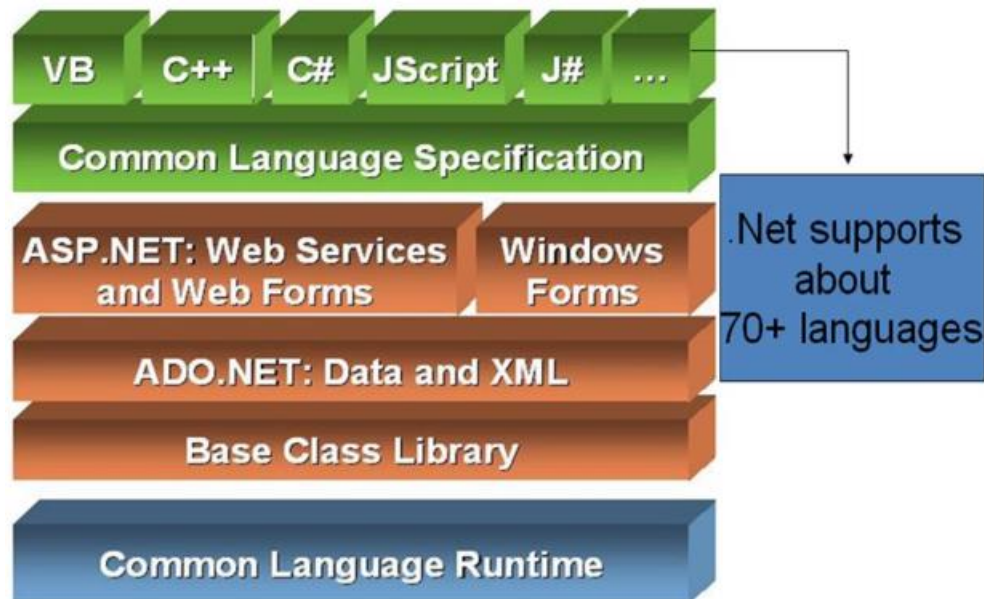
Common Type System (CTS)

- CTS describes all possible data types and all programming rules and conventions supported by the runtime
- The communication between programs written in any .NET compliant language, the types have to be compatible on the basic level

Introduction to .Net Framework Cont

Common Language Specification (CLS)

- It is a sub set of CTS and it specifies a set of rules that needs to be adhered or satisfied by all language compilers targeting CLR
- It helps in cross language inheritance and cross language debugging



C# Version History

| C# Version | .Net Version | Visual Studio | Features | Year |
|------------|------------------------|---------------|--|------|
| C# 1.0 | .Net Framework 1.0/1.1 | 2002 | <ul style="list-style-type: none">• Automatic Memory management using garbage collection• Attribute based programming | 2002 |
| C# 2.0 | .Net Framework 2.0 | 2005 | <ul style="list-style-type: none">• Generic Types and members• Anonymous methods• Partial types• Nullable types | 2005 |

C# Version History

| C# Version | .Net Version | Visual Studio | Features | Year |
|------------|------------------------|---------------|--|------|
| C# 3.0 | .Net Framework 3.0/3.5 | 2008 | <ul style="list-style-type: none">• Strongly-typed queries using LINQ• Anonymous types• Extension methods• Lambda expressions• Object initialization | 2008 |
| C# 4.0 | .Net Framework 4.0 | 2010 | <ul style="list-style-type: none">• Dynamic keyword• Optional parameter/named method argument | 2010 |
| C# 5.0 | .Net Framework 4.5 | 2012/2013 | <ul style="list-style-type: none">• Asynchronous programming | 2012 |

C# Version History

| C# Version | .Net Version | Visual Studio | Features | Year |
|------------|--------------------|---------------|---|------|
| C# 6.0 | .Net Framework 4.6 | 2013/2015 | <ul style="list-style-type: none">• Static imports• Exception filters• Null propagator• String Interpolation• Auto-property Initializer | |
| C# 7.0 | .Net Core | 2017 | <ul style="list-style-type: none">• out variables• Tuples• Discards• Pattern Matching• Local functions | |
| | | | | |

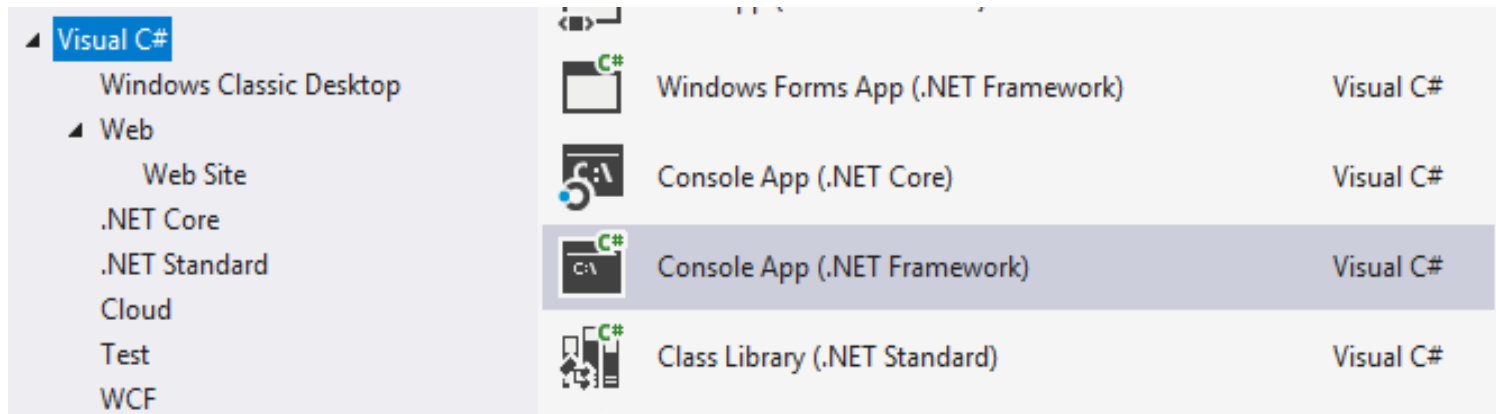
First Console Application

Download and Install the latest version of Visual Studio community from visualstudio.com

C# can be used in a window-based, web-based, or console application. To start with, we will create a console application to work with C#.

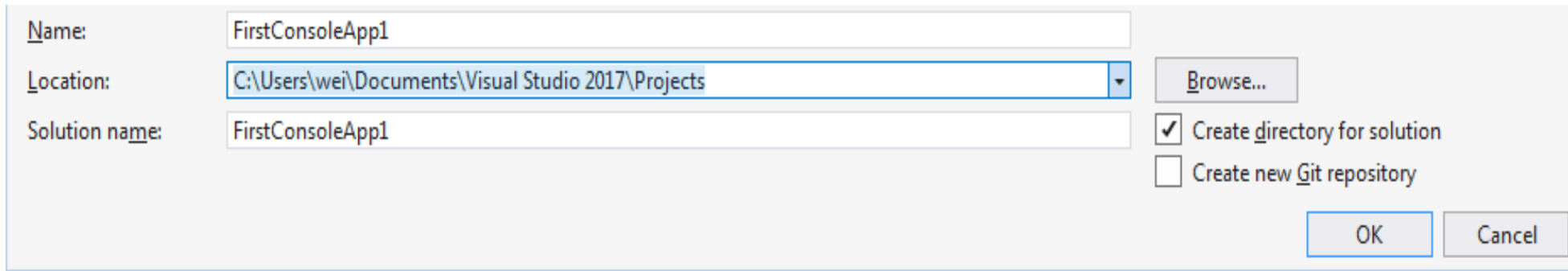
Step 1: Open Visual Studio 2017 installed on your local machine. Click on File -> New Project... from the top menu.

Step 2: From the **New Project** popup select Visual C# in the left side panel and select Console App in the right side panel.



First Console Application

Give an appropriate project name and location where you want to create all your project files and solution name. Click Ok to create the console project.



Name: FirstConsoleApp1

Location: C:\Users\wei\Documents\Visual Studio 2017\Projects

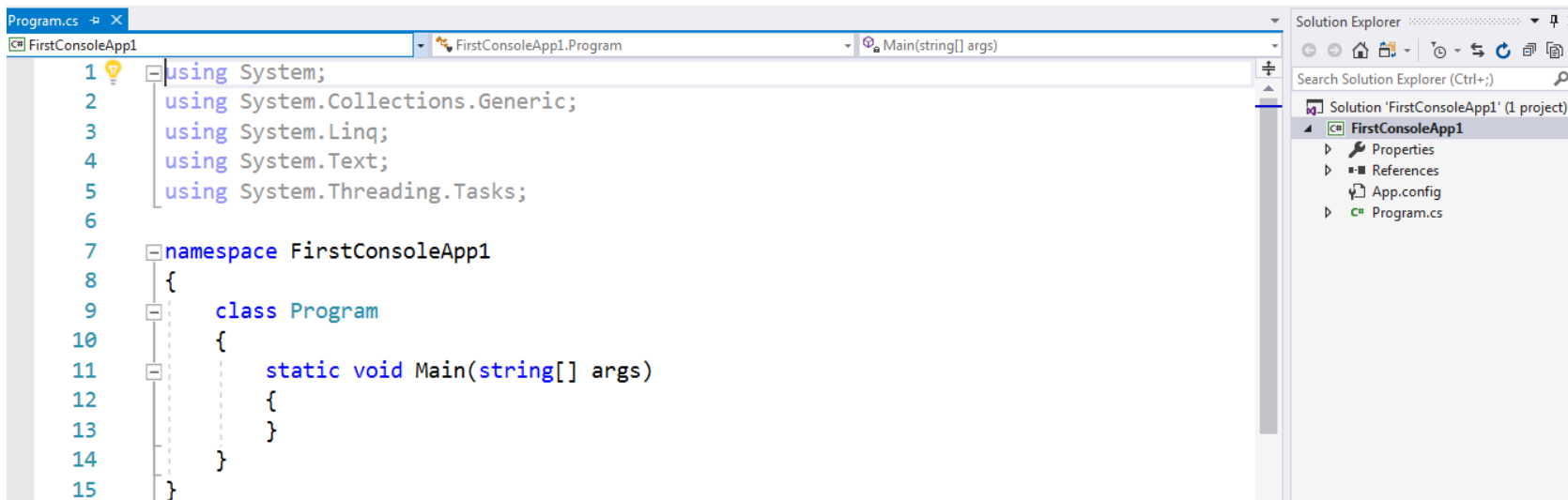
Solution name: FirstConsoleApp1

☒ Create directory for solution

☐ Create new Git repository

OK Cancel

program.cs will be created by default where you can write your c# code



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace FirstConsoleApp1
8 {
9     class Program
10     {
11         static void Main(string[] args)
12         {
13         }
14     }
15 }
```

Solution Explorer shows the project structure: FirstConsoleApp1 (1 project) containing Properties, References, App.config, and Program.cs.

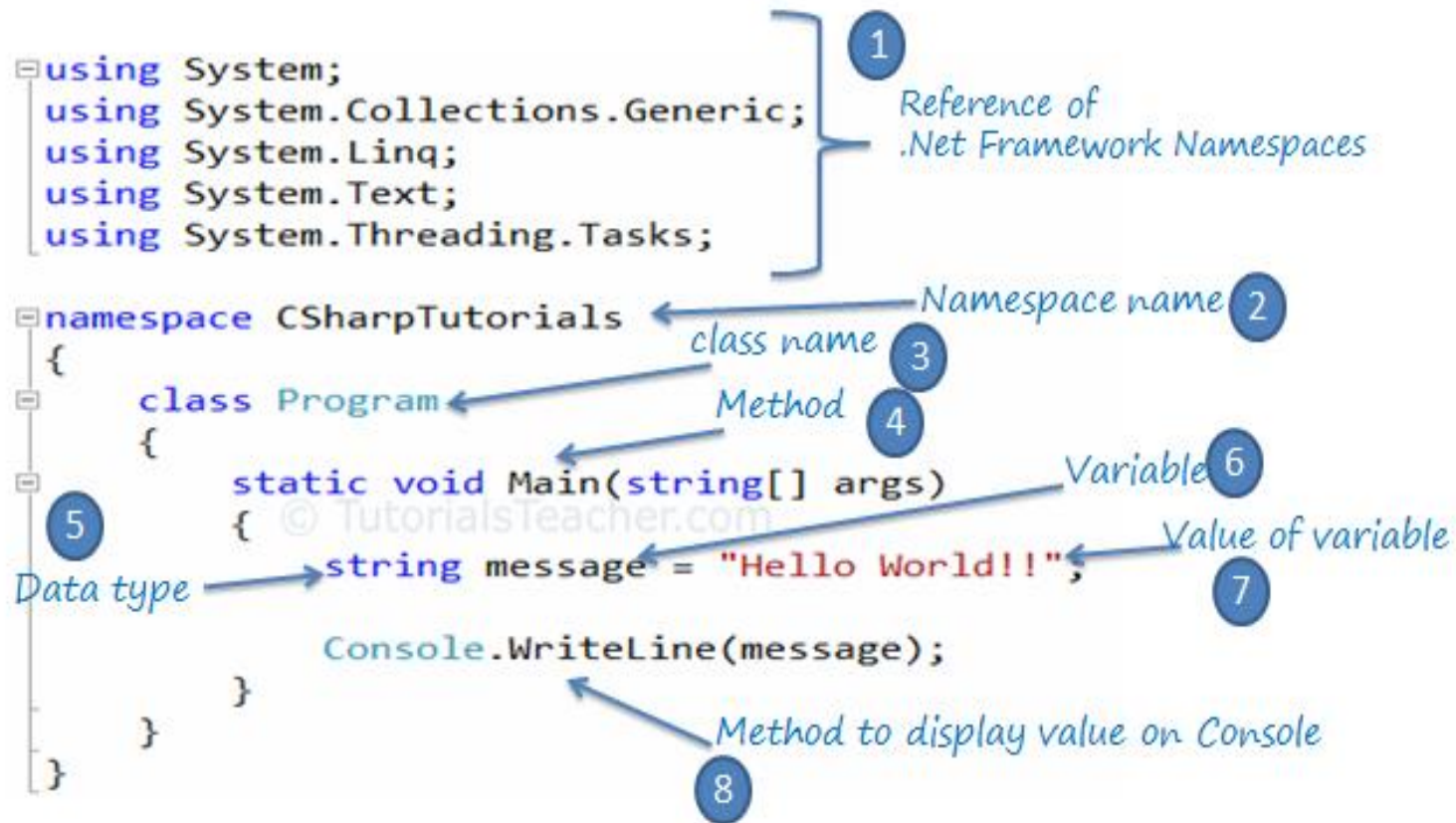
First Console Application

Lets us see a simple example that displays “Hello World!!” on the console.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FirstConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            string message = "Hello World!!";
            Console.WriteLine(message);
        }
    }
}
```

C# Code Structure



C# Code Structure

C# Code Structure

- using: brings in namespaces, like import in java
- namespace: Disambiguation of names, like C++ naming
- class: Single inheritance up to object, same as C++ or java
- Main(): Which is the entry point for the console application
- String : Is the datatype of the variable
- message: Is the variable
- “Hello World!!” is the Value
- WriteLine(): Display the message in the console

Strings and Characters

- C#'s Char type represents a Unicode character, occupies two bytes and specified inside single quotes
- `char a = 'A';`
- Escape sequences (`\`) express characters that cannot be interpreted literally
- String type represents an immutable sequence of Unicode characters
- String literal is specified inside double quotes `string a = "Hello";`
- String is a reference type but with equality operators it follows value type
- A verbatim string literal is prefixed with `@` and can span multiple lines.
- We can concatenate two strings using the `+` operator
- A string preceded with the `$` character is called an interpolated string

Arrays

- An array is a special type of data type which can store fixed number of values sequentially
- Array can be declared using a type name followed by square brackets[]

Example: Array declaration in C#

```
int[] intArray; // can store int values

bool[] boolArray; // can store boolean values

string[] stringArray; // can store string values

double[] doubleArray; // can store double values

byte[] byteArray; // can store byte values

Student[] customClassArray; // can store instances of Student class
```

Arrays

- Array can be declared and initialized at the same time using the new keyword

Example: Array Declaration & Initialization

```
// defining array with size 5. add values later on
int[] intArray1 = new int[5];

// defining array with size 5 and adding values at the same time
int[] intArray2 = new int[5]{1, 2, 3, 4, 5};

// defining array with 5 elements which indicates the size of an array
int[] intArray3 = {1, 2, 3, 4, 5};
```

- Arrays can be initialized after declaration
- String [] strArray1;
- strArray1 = new string[4]{"1st Element", "2nd Element, 3rd Element, 4th Element};

We can assign values to array index like int[] intArray = new int[3];

intArray[0] = 10;

Arrays

- An array is a special type of data type which can store fixed number of values in an sequential order starting with a zero-based index
- The elements in an array are stored in a contiguous block of memory, providing high efficient access
- Array can be declared using the type, array name followed by square brackets
- `int [] intArray;`
- Array can be declared and initialized at the same time using the new keyword
- `char [] vowels = new char[5];` or `char [] vowels = new char [5]{“a”, “e”, “i”, “o”, “u”};`
- Array elements can be retrieved by specifying the index of the array like `vowels[4] = u;`
- To iterate through each element in the array using for loop statement
- `for(int i=0; i< vowels.length; i++);`

Arrays

- All arrays inherits from the System.Array class which defines common methods and properties for all arrays
- It provides static methods for creating, manipulating, searching and sorting arrays
- Dynamically create an array using the CreateInstance method
- Example: Create an instance of an array that starts with index 1 using Array class
- `Array myArray = Array.CreateInstance(typeof(int), new int[1]{3}, new int[1]{1});`
- `myArray.SetValue(1,1);`
- `myArray.SetValue(2,2);`
- `myArray.SetValue(3,3);`
- Get and set elements regardless of the array type using the GetValue/SetValue methods
- Sort an array using the sort method
- Copy an array

Multidimensional Arrays

- A multi-dimensional array is a two dimensional series like rows and columns
- The two types of multi-dimensional array rectangular and jagged arrays
- To declare rectangular arrays use commas to separate each dimension
- Example: `int[,] intArray = new int[3,2]{ {1,2},{3,4},{5,6} };`
- The values of a multi-dimensional array can be accessed using two indexes the first index is for the row and the second index is for the column. Both the indexes start from zero
- Jagged arrays store arrays instead of other data type values directly
- Jagged arrays is initialized with two square brackets `[][]`, the first bracket specifies the size of an array and the second bracket specifies the dimension of the array which is going to be stored as values
- Example: `int [] [,] intJaggedArray = new int[3] [,];`
- All the arrays throw an `IndexOutOfRangeException` exception if the index does not exist

Variables

- A variable is a name given to the data value, the content of the variable can vary at anytime as long as it is accessible
- Variable is always defined with a data type
- A variable can be declared and initialized later or it can be declared and initialized at the same time
- Multiple variables can be defined separated by comma (,) in a single or multiple line till semicolon(;)
- A value must be assigned to a variable before using it otherwise it will give compile time error

Variables and Parameters

- A variable represents a storage location that has a modifiable value
- A variable can be local variable, parameter, field, or array element
- The Stack and the Heap are the places where variables and constants reside, each has very different lifetime semantics
- Stack is the block of memory for storing local variables and parameters
- Stack logically grows and shrinks as a function is entered and exited

- Static int Factorial (int x)
 {
 if(x==0) return 1;
 return x * Factorial(x-1);
 }

Operators

- Arithmetic Operators are used to perform mathematic operations on numbers
- The list of operators available in C#

| Operator | Description |
|----------|---|
| + | Adds two operands |
| - | Subtracts the second operand from the first |
| * | Multiplies both operands |
| / | Divides the numerator by de-numerator |
| % | Modulus Operator and a remainder of after an integer division |
| ++ | Increment operator increases integer value by one |
| -- | Decrement operator decreases integer value by one |

Operators

- Relational operators are used for performing relational operation on numbers
- The list of Relational operators available in C#

| Operator | Description |
|----------|---|
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. |

Operators

- Logical operators are used for performing logical operations on values
- List of Logical operators available in C#

| Operator | Description |
|----------|--|
| && | This is the Logical AND operator. If both the operands are true, then condition becomes true. |
| | This is the Logical OR operator. If any of the operands are true, then condition becomes true. |
| ! | This is the Logical NOT operator. |

Null Operator

- The ?? Operator is called the null coalescing operator that simplifies checking for null values
- It can be used with both nullable types and reference types
- It returns the left hand operand if the operand is not null, otherwise it returns the right hand operand
- Let's see an example

```
int? x = null;

// Set y to the value of x if x is NOT null; otherwise,
// if x == null, set y to -1.
int y = x ?? -1;

// Assign i to return value of the method if the method's result
// is NOT null; otherwise, if the result is null, set i to the
// default value of int.
int i = GetNullableInt() ?? default(int);

string s = GetStringValue();
// Display the value of s if s is NOT null; otherwise,
// display the string "Unspecified".
Console.WriteLine(s ?? "Unspecified");
```

Selection Statements

- A selection statement causes the program control to be transferred to a specific flow based upon whether a certain condition is true or false
- The following keywords are used in selection statements
 - If
 - Else
 - switch
 - case
 - default
- The if statement is used to evaluate a boolean expression before executing a set of statements. If an expression evaluates to true, then it will run one set of statements else it will run another set of statements.

Selection Statements

```
{  
    int m = 12;  
    int n = 18;  
  
    if (m > 10)  
        if (n > 20)  
        {  
            Console.WriteLine("Result1");  
        }  
        else  
        {  
            Console.WriteLine("Result2");  
        }  
}
```

Switch Statement

- The switch statement is used to evaluate an expression and run different statements based on the result of the expression. If one condition does not evaluate to true, the switch statement will then move to the next condition and so forth

Selection Statements

```
public static void Main()
{
    int caseSwitch = 1;

    switch (caseSwitch)
    {
        case 1:
            Console.WriteLine("Case 1");
            break;
        case 2:
            Console.WriteLine("Case 2");
            break;
        default:
            Console.WriteLine("Default case");
            break;
    }
}
```

Iteration Statements

- Iteration statements are used to create loops. Iteration statements cause embedded statements to be executed a number of times, subject to the loop-termination criteria
- The following keywords are used in the Iteration Statements:
 - while
 - do
 - for
 - foreach
- The while loop executes a statement or block of statements while a specified boolean expression evaluates to true. The expression is evaluated before each execution of the loop
- We can break out of the while loop at any point using the break statement.

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

Iteration Statements

- The do statement executes a statement or block of statements while a specified boolean expression evaluates to true. The expression is evaluated after each execution of the loop it is executed one or more times
- We can break out of the while loop at any point using the break statement

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

- The foreach statement allows the iteration of processing over the elements in arrays and collections.
- Within the foreach loop parentheses, the expression consists of two parts separated by the keyword in. To the right of in is the collection, and to the left is the variable with the type identifier matching whatever type the collection returns.

Iteration Statements

Each iteration queries the collection for a new value for *i*. As long as the collection `intNumbers` returns a value, the value is put into the variable *i* and the loop will continue. When the collection is fully traversed, the loop will terminate.

```
Int16[] intNumbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };  
foreach (Int16 i in intNumbers)  
{  
    System.Console.WriteLine(i);  
}
```

For loop is used when we want to repeat a certain set of statements for a particular number of times. The for statements defines initializer, condition, and Iterator sections

```
static void Main(string[] args)  
{  
    for (Int32 i = 0; i < 3; i++)  
    {  
        Console.WriteLine(i);  
  
        Console.ReadKey();  
    }  
}
```

Namespaces

- Namespaces are used to organize and provide a level of separation of codes. It can be considered as a container which contains of other namespaces, classes etc.
- The using directive obviates the requirement to specify the name of the namespace for every class
- The global namespace is the root namespace `global::System` will always refer to the .NET framework namespace `System`
- Namespaces play important role in writing cleaner codes and managing larger projects
- Namespaces solves the problem of naming conflict
- Namespaces can have the following types as its members
 - Namespaces
 - Classes
 - Interfaces
 - Structures
 - Delegates

Namespaces

Defining Namespace:

We can define using the namespace keyword

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
                "SampleMethod inside SampleNamespace");
        }
    }
}
```

Accessing Members of Namespace:

The members of namespace can be accessed using the dot(.) operator.

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


Access Modifiers

- Access modifiers are keywords used to specify the declared accessibility of a member or a type
- The four access modifiers are
 - public
 - protected
 - internal
 - private
- The following six accessibility levels can be specified using the access modifiers:
 - Public: Access is not restricted
 - protected: Access is limited to the containing class or derived from the containing class

Access Modifiers

- `internal`: Access is limited to the current assembly
- `protected internal`: Access is limited to the current assembly or derived from the containing class
- `private`: Access is limited to the containing type
- `private protected`: Access is limited to the containing class or derived from the containing class within the current assembly

Classes and Objects

- Class defines the kinds of data and the functionality their objects will have
- A type that is defined as a Class is a reference type
- Class is defined using the keyword class followed by the class name and the class body enclosed by a pair of curly braces.
- The Access modifier specify the access rules for the members as well as the class itself
- By Default the access modifier of class is internal and the data is private
- We have to create an instance of the class using the new operator to access the members of the class
- When we create an instance of the class on the managed heap the variable holds only the reference to the location of the object

Value and Reference Types

- The data type is a value type if it holds a data value within its own memory space.
- The value types are all stored in the stack memory
- The value types cannot be a null value
- Example: `int i = 100;`

This value 100 is stored in some memory location that is allocated for i

The following data types are all value types:

- Bool
- byte
- int
- double
- decimal
- float
- enum

Passing by value

- When you pass a value type variable from one method to another method, the system creates a separate copy of a variable in another method, so that if value got changed in the one method won't affect on the variable in another method.

Example: In this code the value of i in Main method remains unchanged even after passing it to the ChangeValue() methods and change it's value there

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

    Console.WriteLine(x);
}

static void Main(string[] args)
{
    int i = 100;

    Console.WriteLine(i);

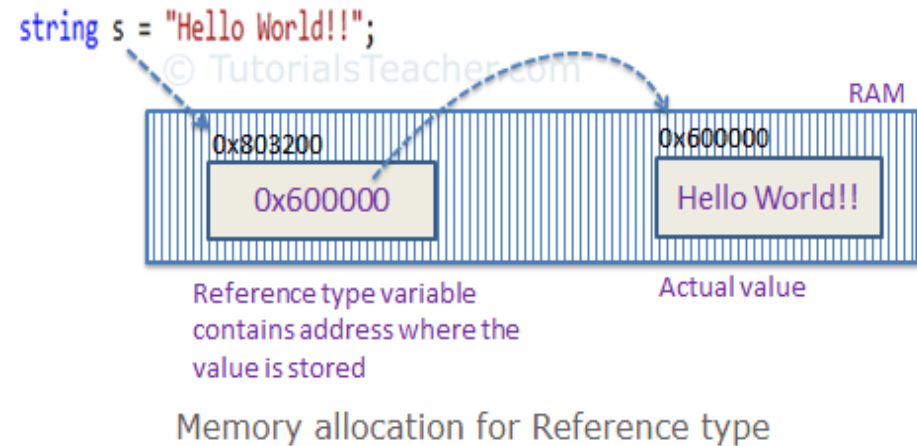
    ChangeValue(i);

    Console.WriteLine(i);
}
```

Reference Type

- Reference types contains a pointer to another memory location that holds the data
- Reference type are by default has null value

Example: String s = "Hello World";



The following data types are of reference types:

- String
- All arrays, even if their elements are value types
- Class
- Delegates

Passing by reference

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

In the code when we pass the Student object std1 to the ChangeReferenceType() method we are sending the memory address of the std1. When the method changes the StudentName it actually changes the StudentName of std1.

```
static void ChangeReferenceType(Student std2)
{
    std2.StudentName = "Steve";
}

static void Main(string[] args)
{
    Student std1 = new Student();
    std1.StudentName = "Bill";

    ChangeReferenceType(std1);

    Console.WriteLine(std1.StudentName);
}
```

Interfaces

- An interface only contains declarations of method, events and properties
- An interface cannot include private members
- Interface members are public by default
- Interface cannot contains fields, operator, constructors
- We can define interface using the keyword interface
- To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member
- If a class inherits from an interface, it must provide implementation for all interface members
- A class or struct can implement multiple interfaces.
- Interface can inherit from other interfaces.
- We cannot create an instance for an interface but an interface reference variable can point to a derived class object

Interfaces

We can define interface using the **interface** keyword, other classes can implement ILog by providing an implementation of the Ilog() method.

```
class ConsoleLog: ILog
{
    public void Log(string msgToPrint)
    {
        Console.WriteLine(msgToPrint);
    }
}

class FileLog : ILog
{
    public void Log(string msgToPrint)
    {
        File.AppendText(@"C:\Log.txt").Write(msgToPrint);
    }
}
```

```
interface ILog
{
    void Log(string msgToLog);
}
```

Interfaces

In the code the consoleLog class implements the Ilog interface to log the string on the console, whereas the FileLog class implements the Ilog to log the string into a text file.

Instantiate Object:

```
Ilog log = new ConsoleLog();
```

```
//Or
```

```
Ilog log = new FileLog();
```

Explicit Implementation:

We can implement interfaces explicitly by prefixing the interface name with the method name.

```
class ConsoleLog: Ilog
{
    public void Ilog.Log(string msgToPrint) // explicit implementation
    {
        Console.WriteLine(msgToPrint);
    }
}
```

Enum

- The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list
- Enum can be defined directly within the namespace so that all the classes in the namespace can access it equally
- By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1
- we will define an enumeration called days, which will be used to store the days of the week

```
public class EnumTest
{
    enum Day { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

    static void Main()
    {
        int x = (int)Day.Sun;
        int y = (int)Day.Fri;
        Console.WriteLine("Sun = {0}", x);
        Console.WriteLine("Fri = {0}", y);
    }
}
```

StringBuilder

- StringBuilder is a dynamic object that allows you to expand the number of characters in the string. It doesn't create a new object in the memory but dynamically expands memory to accommodate the modified string
- StringBuilder is found in the System.Text namespace we have to import this namespace in our code to use it
- We have to create a new instance of the stringBuilder class by initializing to a variable

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
```
- We can set the capacity and the length of stringBuilder by specifying the size

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!", 25);
```
- We can modify the contents of the stringBuilder using the build-in methods
- StringBuilder performs faster than string when concatenating multiple strings

StringBuilder

- List of methods we can use to modify the content of the stringbuilder

| Method name | Use |
|---|---|
| <code>StringBuilder.Append</code> | Appends information to the end of the current <code>StringBuilder</code> . |
| <code>StringBuilder.AppendFormat</code> | Replaces a format specifier passed in a string with formatted text. |
| <code>StringBuilder.Insert</code> | Inserts a string or object into the specified index of the current <code>StringBuilder</code> . |
| <code>StringBuilder.Remove</code> | Removes a specified number of characters from the current <code>StringBuilder</code> . |
| <code>StringBuilder.Replace</code> | Replaces a specified character at a specified index. |

StringBuilder

- Use Append method of StringBuilder to add or append a string to StringBuilder. AppendLine() method appends the string with a newline at the end

Example:

```
StringBuilder sb = new StringBuilder("Hello ",50);

sb.Append("World!!");
sb.AppendLine("Hello C#!");
sb.AppendLine("This is new line.");

Console.WriteLine(sb);
```

AppendFormat() is used to format input string into specified format and then append it

```
StringBuilder amountMsg = new StringBuilder("Your total amount is ");
amountMsg.AppendFormat("{0:C} ", 25);

Console.WriteLine(amountMsg);
```

StringBuilder

- Insert() method inserts the string at a specified index in stringbuilder

```
StringBuilder sb = new StringBuilder("Hello World!!",50);  
sb.Insert(5," C#");
```

```
Console.WriteLine(sb);
```

- Remove() method removes the string at a specified index with specified length

```
StringBuilder sb = new StringBuilder("Hello World!!",50);  
sb.Remove(6, 7);
```

```
Console.WriteLine(sb);
```

StringBuilder

- Replace() method replaces all occurrence of a specified string with a specified replacement string

```
StringBuilder sb = new StringBuilder("Hello World!!",50);  
sb.Replace("World", "C#");  
  
Console.WriteLine(sb);
```

- Indexer is used to set or get a character at specified index

```
StringBuilder sb = new StringBuilder("Hello World!!");  
  
for(int i=0; i< sb.Length; i++)  
    Console.Write(sb[i]);
```


StringBuilder

- ToString() method is used to read a string from the stringbuilder

```
StringBuilder sb = new StringBuilder("Hello World!!");
```

```
string str = sb.ToString(); // "Hello World!!"
```

Stack and Heap Memory

- The Heap is the block of memory in which objects (i.e reference type instances) resides
- Whenever a new object is created, it is allocated on the heap and a reference to that object is returned
- An object can't be deleted explicitly, an unreferenced object is eventually collected by the garbage collector
- C# enforces definite assignment, so it is impossible to access uninitialized memory
- Local variables must be assigned a value before they can be read
- Function arguments must be supplied when a method is called(unless marked as optional parameter)
- All other variables (such as fields and array elements) are automatically initialized by the runtime

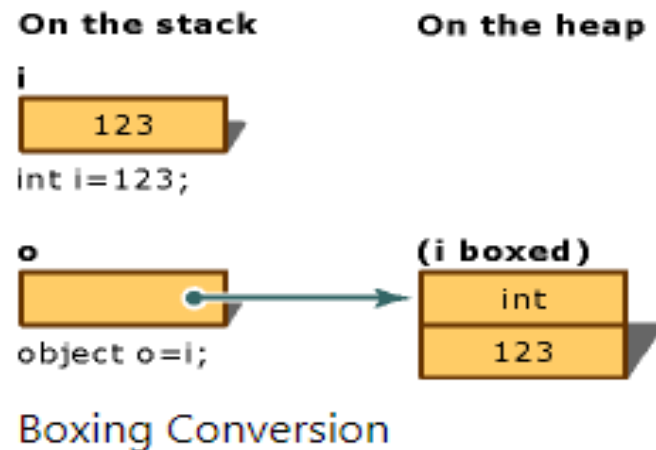
Boxing and Unboxing

- Boxing is an implicit conversion of a value type to the type object
- Boxing a value type allocates an object instance on the heap and copies the value into the new object
- Example `int i = 123;`

Implicitly applies the boxing operation on the variable i.

`object o = i;`

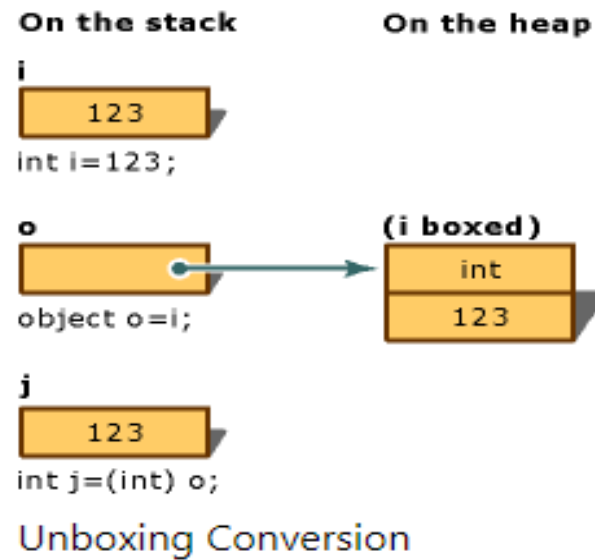
A reference of object o is created on the stack that references a value of the type int on the heap.



Unboxing

- Unboxing is an explicit conversion from the type object to a value type

```
int i = 123;    // a value type
object o = i;   // boxing
int j = (int)o; // unboxing
```



OOPS Concepts- Encapsulation

- Encapsulation is a way of hiding the data into a single unit called class, by doing this we can protect our data
- Using access modifiers we can achieve encapsulating our data from the outside world
 - Private If we create any function or variable inside the class as private then it will not be accessible to outside the class.
 - Protected members are visible only in derived classes
 - Internal members are visible only in derived classes that are located in the same assembly as the base class. They are not visible in derived classes located in a different assembly from the base class
 - Public members are visible in derived classes and are part of the derived class public interface

OOPS Concepts- Inheritance

Inheritance:

- Inheritance is a concept in which you define parent classes and child classes
- It allows you to define a child class that reuses (inherits), extends, or modifies the behavior of a parent class
- The class whose members are inherited is called the *base class*
- The class that inherits the members of the base class is called the *derived class*
- C# and .NET support *single inheritance* only
- inheritance is transitive, which allows you to define an inheritance hierarchy for a set of types
- Not all members of a base class are inherited by derived classes
- The following members are not inherited from the base class
 - [Static constructors](#), which initialize the static data of a class
 - [Instance constructors](#)
 - [Finalizers](#), which are called by the runtime's garbage collector to destroy instances of a class.

OOPS Concepts- Inheritance

- Let's see an example for Inheritance

```
Animal a1 = new Animal();  
a1.Talk();  
a1.Sing();  
a1.Greet();
```

```
//Output  
Animal constructor  
Animal talk  
Animal song  
Animal says Hello
```

```
class Animal  
{  
    public Animal()  
    {  
        Console.WriteLine("Animal constructor");  
    }  
    public void Greet()  
    {  
        Console.WriteLine("Animal says Hello");  
    }  
    public void Talk()  
    {  
        Console.WriteLine("Animal talk");  
    }  
    public virtual void Sing()  
    {  
        Console.WriteLine("Animal song");  
    }  
};
```

```
class Dog : Animal  
{  
    public Dog()  
    {  
        Console.WriteLine("Dog constructor");  
    }  
    public new void Talk()  
    {  
        Console.WriteLine("Dog talk");  
    }  
    public override void Sing()  
    {  
        Console.WriteLine("Dog song");  
    }  
};
```

```
Animal a2 = new Dog();  
a2.Talk();  
a2.Sing();  
a2.Greet();
```

```
//Output  
Animal constructor  
Dog constructor  
Animal talk  
Dog song  
Animal says Hello
```

OOPS Concepts- Inheritance

- A member's accessibility affects its visibility for derived classes as follows:
 - Private members are visible only in derived classes that are nested in their base class

Example:

```
public class A
{
    private int value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return this.value;
        }
    }
}

public class C : A
{
    // public int GetValue()
    // {
    //     return this.value;
    // }
}

public class Example
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//     10
```


OOPS Concepts- Inheritance

- Derived classes can also *override* inherited members by providing an alternate implementation by marking the member in the base class using the virtual keyword
- The base keyword is used to access an overridden function member from the subclass and calling a base class constructor

Abstract Class

- Abstract modifier in a class declaration indicates that the class is intended only to be a base class of other classes
- A class declared as abstract can never be instantiated only its concrete subclasses can be instantiated
- An abstract class may contain abstract methods and accessors
- A class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors

Abstract methods have the following features:

- An abstract method is implicitly a virtual method and provides only the method declaration
`public abstract void MyMethod();`
- The implementation is provided by a method [override](#), which is a member of a sub or derived class.

Polymorphism

- It allows you to invoke methods of derived class through base class reference during runtime
- It has the ability for classes to provide different implementations of methods that are called through the same name

Polymorphism is of two types:

- Compile time polymorphism/Overloading
- Runtime polymorphism/Overriding

Compile time polymorphism:

- Compile time polymorphism is method and operator overloading which is also called early binding
- The methods performs different task at the different input parameters

Runtime polymorphism:

- Runtime time polymorphism is done using inheritance and virtual functions
- Method overriding is called runtime polymorphism which is also called late binding

Collections

- Collections are similar to Arrays, it provides a more flexible way of working with a group of objects
- The group of objects can grow and shrink dynamically as the needs of the application
- Collections are Key value pairs so that you can retrieve the object quickly by using the key
- A collection is a class, so you must declare an instance of the class before you can add elements to that collection
- Collections are of two types
 - Non-Generic Collection:
 - The non-generic collections (ArrayList, Hashtable, SortedList, Queue etc.) store elements internally in 'object' arrays which, can of course, store any type of data.
 - Generic Collection:
 - Generic collections (List<T>, Dictionary<T, U>, SortedList<T, U>, Queue<T> etc) store elements internally in arrays of their actual types and so no boxing or casting is ever required

Collections- Non Generic

ArrayList:

- ArrayList can store elements of any datatype
- ArrayList resizes automatically as you add the elements
- ArrayList values must be cast to appropriate data types before using it
- ArrayList can contain multiple null and duplicate items
- ArrayList can be accessed using foreach or for loop or indexer
- Using the new keyword we create an object to access the elements of the array list

Adding element to the ArrayList

```
ArrayList arryList1 = new ArrayList();  
arryList1.Add(1);  
arryList1.Add("Two");  
arryList1.Add(3);  
arryList1.Add(4.5);
```

Collections- Non Generic

Pass ArrayList as an argument

```
//  
// Create an ArrayList and add two ints.  
//  
ArrayList list = new ArrayList();  
list.Add(5);  
list.Add(7);  
//  
// Use ArrayList with method.  
//  
Example(list);  
}  
  
static void Example(ArrayList list)  
{  
    foreach (int i in list)  
    {  
        Console.WriteLine(i);  
    }  
}
```

Combining two ArrayList

```
ArrayList arrayList1 = new ArrayList();  
arrayList1.Add(1);  
arrayList1.Add("Two");  
arrayList1.Add(3);  
arrayList1.Add(4.5);  
  
ArrayList arrayList2 = new ArrayList();  
arrayList2.Add(100);  
arrayList2.Add(200);  
  
//adding entire arrayList2 into arrayList1  
arrayList1.AddRange(arrayList2);
```

Collections- Non Generic

Sort and Reverse the elements in the ArrayList

```
ArrayList arryList1 = new ArrayList();  
arryList1.Add(300);  
arryList1.Add(200);  
arryList1.Add(100);  
arryList1.Add(500);  
arryList1.Add(400);  
  
Console.WriteLine("Original Order:");  
  
foreach(var item in arryList1)  
    Console.WriteLine(item);  
  
arryList1.Reverse();  
Console.WriteLine("Reverse Order:");  
  
foreach(var item in arryList1)  
    Console.WriteLine(item);  
  
arryList1.Sort();  
Console.WriteLine("Ascending Order:");  
  
foreach(var item in arryList1)  
    Console.WriteLine(item);
```

Collections- Non Generic

Remove an element from ArrayList

```
ArrayList arryList1 = new ArrayList();  
arryList1.Add(100);  
arryList1.Add(200);  
arryList1.Add(300);  
  
arryList1.Remove(100); //Removes 1 from ArrayList  
  
foreach (var item in arryList1)  
    Console.WriteLine(item);
```

Accessing individual elements in the ArrayList

```
ArrayList myArryList = new ArrayList();  
myArryList.Add(1);  
myArryList.Add("Two");  
myArryList.Add(3);  
myArryList.Add(4.5f);  
  
//Access individual item using indexer  
int firstElement = (int) myArryList[0]; //returns 1  
string secondElement = (string) myArryList[1]; //returns "Two"  
int thirdElement = (int) myArryList[2]; //returns 3  
float fourthElement = (float) myArryList[3]; //returns 4.5  
  
//use var keyword  
var firstElement = myArryList[0]; //returns 1
```


Collections- Non Generic

Stack:

- Stack stores the values in LIFO (Last in First out) style. The element which is added last will be the element to come out first
- Use the Push() method to add elements into Stack
- The Pop() method returns and removes elements from the top of the Stack. Calling the Pop() method on the empty Stack will throw an exception
- Stack allows null value and also duplicate values
- The Peek() method always returns top most element in the Stack

Collections- Non Generic

Important Properties and Methods of Stack

| Property | Usage |
|----------|---|
| Count | Returns the total count of elements in the Stack. |

| Method | Usage |
|----------|--|
| Push | Inserts an item at the top of the stack. |
| Peek | Returns the top item from the stack. |
| Pop | Removes and returns items from the top of the stack. |
| Contains | Checks whether an item exists in the stack or not. |
| Clear | Removes all items from the stack. |

Collections- Non Generic

Add values into Stack

```
Stack myStack = new Stack();  
myStack.Push("Hello!!");  
myStack.Push(null);  
myStack.Push(1);  
myStack.Push(2);  
myStack.Push(3);  
myStack.Push(4);  
myStack.Push(5);
```

Accessing Stack elements

```
public static void Main()  
{  
    Stack myStack = new Stack();  
    myStack.Push("Hello!!");  
    myStack.Push(null);  
    myStack.Push(1);  
    myStack.Push(2);  
    myStack.Push(3);  
    myStack.Push(4);  
    myStack.Push(5);  
  
    foreach (var item in myStack)  
        Console.WriteLine(item);  
}
```

Collections- Non Generic

To Peek the element in the Stack

```
public static void Main()
{
    Stack myStack = new Stack();
    myStack.Push(1);
    myStack.Push(2);
    myStack.Push(3);
    myStack.Push(4);
    myStack.Push(5);

    Console.WriteLine(myStack.Peek());
    Console.WriteLine(myStack.Peek());
    Console.WriteLine(myStack.Peek());
}
```

To Remove and return values in the Stack

```
public static void Main()
{
    Stack myStack = new Stack();
    myStack.Push(1);
    myStack.Push(2);
    myStack.Push(3);
    myStack.Push(4);
    myStack.Push(5);

    Console.WriteLine("Number of elements in Stack: {0}", myStack.Count);

    while (myStack.Count > 0)
        Console.WriteLine(myStack.Pop());

    Console.WriteLine("Number of elements in Stack: {0}", myStack.Count);
}
```

Collections- Non Generic

Queue

- The Queue stores the values in FIFO (First in First out) style. The element which is added first will come out First
- Use the Enqueue() method to add elements into Queue
- The Dequeue() method returns and removes elements from the beginning of the Queue. Calling the Dequeue() method on an empty queue will throw an exception
- The Peek() method always returns top most element

Collections- Non Generic

Important Properties and Methods of Queue

| Property | Usage |
|----------|---|
| Count | Returns the total count of elements in the Queue. |

| Method | Usage |
|------------|--|
| Enqueue | Adds an item into the queue. |
| Dequeue | Removes and returns an item from the beginning of the queue. |
| Peek | Returns an first item from the queue |
| Contains | Checks whether an item is in the queue or not |
| Clear | Removes all the items from the queue. |
| TrimToSize | Sets the capacity of the queue to the actual number of items in the queue. |

Collections- Non Generic

Add elements in Queue

```
Queue queue = new Queue();  
queue.Enqueue(3);  
queue.Enqueue(2);  
queue.Enqueue(1);  
queue.Enqueue("Four");
```

To remove elements in Queue

```
public static void Main()  
{  
    Queue queue = new Queue();  
    queue.Enqueue(3);  
    queue.Enqueue(2);  
    queue.Enqueue(1);  
    queue.Enqueue("Four");  
  
    Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);  
  
    while (queue.Count > 0)  
        Console.WriteLine(queue.Dequeue());  
  
    Console.WriteLine("Number of elements in the Queue: {0}", queue.Count);  
}
```

Collections- Non Generic

To Peek the element in Queue

```
public static void Main()
{
    Queue queue = new Queue();
    queue.Enqueue(3);
    queue.Enqueue(2);
    queue.Enqueue(1);
    queue.Enqueue("Four");

    Console.WriteLine("Number of elements in the Queue: {0}", queue.Count)

    Console.WriteLine(queue.Peek());
    Console.WriteLine(queue.Peek());
    Console.WriteLine(queue.Peek());

    Console.WriteLine("Number of elements in the Queue: {0}", queue.Count)
```

Checks whether an element exists in a Queue

```
public static void Main()
{
    Queue queue = new Queue();
    queue.Enqueue(3);
    queue.Enqueue(2);
    queue.Enqueue(1);
    queue.Enqueue("Four");

    Console.WriteLine(queue.Contains(2)); // true
    Console.WriteLine(queue.Contains(100)); //false
}
```


Collections- Non Generic

Hashtable:

- Hashtable stores key-value pairs of any datatype where the Key must be unique
- The Hashtable key cannot be null whereas the value can be null
- Hashtable retrieves an item by comparing the hashcode of keys. So it is slower in performance than Dictionary collection
- Hashtable uses the default hashcode provider which is `object.GetHashCode()`. You can also use a custom hashcode provider
- Use `DictionaryEntry` with `foreach` statement to iterate Hashtable

Collections- Non Generic

Important Methods of Hashtable

| Methods | Usage |
|---------------|---|
| Add | Adds an item with a key and value into the hashtable. |
| Remove | Removes the item with the specified key from the hashtable. |
| Clear | Removes all the items from the hashtable. |
| Contains | Checks whether the hashtable contains a specific key. |
| ContainsKey | Checks whether the hashtable contains a specific key. |
| ContainsValue | Checks whether the hashtable contains a specific value. |
| GetHash | Returns the hash code for the specified key. |

Collections- Non Generic

Add key-value into Hashtable

- The Add() method adds an item with a key and value into the Hashtable
- Key and value can be of any data type
- Key should be unique and cannot be null whereas value can be null

```
public static void Main()
{
    Hashtable ht = new Hashtable();

    ht.Add(1, "One");
    ht.Add(2, "Two");
    ht.Add(3, "Three");
    ht.Add(4, "Four");
    ht.Add(5, null);
    ht.Add("Fv", "Five");
    ht.Add(8.5F, 8.5);

    Console.WriteLine("Number of elements: {0}", ht.Count);
}
```

Collections- Non Generic

Hashtable can include all the elements of [Dictionary](#)

```
public static void Main()
{
    Dictionary<int, string> dict = new Dictionary<int, string>();

    dict.Add(1, "one");
    dict.Add(2, "two");
    dict.Add(3, "three");

    Hashtable ht = new Hashtable(dict);

    Console.WriteLine("Total elements: {0}", ht.Count);

}
```

Collections- Non Generic

Access elements from Hashtable

```
public static void Main()
{
    Hashtable ht = new Hashtable();

    ht.Add(1, "One");
    ht.Add(2, "Two");
    ht.Add(3, "Three");
    ht.Add(4, "Four");
    ht.Add("Fv", "Five");
    ht.Add(8.5F, 8.5F);

    string strValue1 = (string)ht[2];
    string strValue2 = (string)ht["Fv"];
    float fValue = (float) ht[8.5F];

    Console.WriteLine(strValue1);
    Console.WriteLine(strValue2);
    Console.WriteLine(fValue);
}
```

Remove elements in Hashtable

```
public static void Main()
{
    Hashtable ht = new Hashtable();
    ht.Add(1, "One");
    ht.Add(2, "Two");
    ht.Add(3, "Three");
    ht.Add(4, "Four");
    ht.Add("Fv", "Five");
    ht.Add(8.5F, 8.5);

    ht.Remove("Fv");

    foreach (var key in ht.Keys )
        Console.WriteLine("Key:{0}, Value:{1}",key , ht[key]);
}
```

Checking for Existing element in the Hashtable

```
public static void Main()
{
    Hashtable ht = new Hashtable();
    ht.Add(1, "One");
    ht.Add(2, "Two");
    ht.Add(3, "Three");
    ht.Add(4, "Four");

    Console.WriteLine(ht.Contains(2)); // returns true
    Console.WriteLine(ht.ContainsKey(2)); // returns true
    Console.WriteLine(ht.Contains(5)); // returns false
    Console.WriteLine(ht.ContainsValue("One")); // returns true
}
```

Generic Collection

- A generic collection gets all the benefit of generics. It doesn't need to do boxing and unboxing while storing or retrieving items and so performance is improved
- Generics denotes with angel bracket <>
- Compiler applies specified type for generics at compile time
- Generics can be applied to interface, abstract class, method, static method, property, event, delegate and operator
- Generic are type safe. You get compile time errors if you try to use a different type of data than the one specified in the definition
- Generics performs faster by not doing boxing & unboxing

Generic Collection

The following are widely used generic collections

| Generic Collections | Description |
|-------------------------|--|
| List<T> | Generic List<T> contains elements of specified type. It grows automatically as you add elements in it. |
| Dictionary<TKey,TValue> | Dictionary<TKey,TValue> contains key-value pairs. |
| SortedList<TKey,TValue> | SortedList stores key and value pairs. It automatically adds the elements in ascending order of key by default. |
| HashSet<T> | HashSet<T> contains non-duplicate elements. It eliminates duplicate elements. |
| Queue<T> | Queue<T> stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. |
| Stack<T> | Stack<T> stores the values as LIFO (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. |

Generic Collection

Generic List<T>

- List<T> stores elements of the specified type and it grows automatically
- List<T> can store multiple null and duplicate elements
- List<T> can be access using indexer, for loop or foreach statement
- List<T> is ideal for storing and retrieving large number of elements
- List<T> can be assigned to **IList<T>** or **List<T>** type of variable. It provides more helper method When assigned to List<T> variable

Generic Collection

Add Elements into List:

```
public static void Main()
{
    IList<int> intList = new List<int>();
    intList.Add(10);
    intList.Add(20);
    intList.Add(30);
    intList.Add(40);

    Console.WriteLine(intList.Count);

    IList<string> strList = new List<string>();
    strList.Add("one");
    strList.Add("two");
    strList.Add("three");
    strList.Add("four");
    strList.Add("four");
    strList.Add(null);
    strList.Add(null);
}
```

Add elements using object initializer syntax

```
IList<int> intList = new List<int>(){ 10, 20, 30, 40 };
```

//Or

```
IList<Student> studentList = new List<Student>() {
    new Student(){ StudentID=1, StudentName="Bill"},
    new Student(){ StudentID=2, StudentName="Steve"},
    new Student(){ StudentID=3, StudentName="Ram"},
    new Student(){ StudentID=1, StudentName="Moin"}
};
```

Generic Collection

- The AddRange() method adds all the elements from another collection

```
IList<int> intList1 = new List<int>();  
intList1.Add(10);  
intList1.Add(20);  
intList1.Add(30);  
intList1.Add(40);
```

```
List<int> intList2 = new List<int>();  
intList2.Add(50);  
intList2.Add(60);  
intList2.Add(70);  
intList2.Add(80);
```

```
intList2.AddRange(intList1);  
Console.WriteLine(intList2.Count);
```

Generic Collection

Accessing elements from the List using For loop

```
List<int> intList2 = new List<int>();
intList2.Add(50);
intList2.Add(60);
intList2.Add(70);
intList2.Add(80);
intList2.AddRange(intList1);

Console.WriteLine(intList2.Count);

for(int i = 0; i<= intList2.Count; i++)
{
    Console.WriteLine(i);
}
```

Accessing elements from List using foreach

```
//Accessing elements using foreach loop
List<int> primes = new List<int>(new int[] { 2, 3, 5 });

// See if List contains 3.
foreach (int number in primes)
{
    if (number == 3) // Will match once.
    {
        Console.WriteLine("Contains 3");
    }
}
```

Generic Collection

Copy Array to the List

//Copying Array to the List

```
int[] arr = new int[3]; // New array with 3 elements.  
arr[0] = 2;  
arr[1] = 3;  
arr[2] = 5;
```

```
List<int> list = new List<int>(arr); // Copy to List.  
Console.WriteLine(list.Count);
```

Search an element in the List

//Searching element in List

```
List<int> primes = new List<int>(new int[] { 19, 23, 29 });  
int index = primes.IndexOf(23); // Exists.  
Console.WriteLine(index);
```

```
index = primes.IndexOf(10); // Does not exist.  
Console.WriteLine(index);
```

Generic Collection

- **Contains:** This method scans a List. It searches for a specific element to see if that element occurs at least once in the collection.
- **Exists** returns whether a List element is present. This is an instance method that returns true or false depending on whether any element matches the Predicate parameter. We invoke this method with a lambda expression.

```
//Using Exist method
```

```
bool exists = primes.Exists(element => element > 10);  
Console.WriteLine(exists);
```

- **Find** is used to search an element in the List it is more efficient than using a for loop. We invoke this method with a lambda expression.

```
// Finds first element greater than 20
```

```
int result = primes.Find(item => item > 20);  
  
Console.WriteLine(result);
```

Generic Collection

- **Join string list.** We use **string.Join** on a List of strings. This is helpful when we need to turn several strings into one comma-delimited string. It requires the **ToArray** instance method on List. This ToArray is not an extension method.

```
//Join String using string.Join|
List<string> cities = new List<string>();
cities.Add("New York");
cities.Add("Mumbai");
cities.Add("Berlin");
cities.Add("Istanbul");

// Join strings into one CSV line.
string line = string.Join(",", cities.ToArray());
Console.WriteLine(line);
```

Generic Collection

Convert List to single string using StringBuilder

```
List<string> cats = new List<string>(); // Create new list of strings
cats.Add("Devon Rex"); // Add string 1
cats.Add("Manx"); // 2
cats.Add("Munchkin"); // 3
cats.Add("American Curl"); // 4
cats.Add("German Rex"); // 5

StringBuilder builder = new StringBuilder();
foreach (string cat in cats) // Loop through all strings
{
    builder.Append(cat).Append("|"); // Append string to StringBuilder

    string result = builder.ToString(); // Get string from StringBuilder
    Console.WriteLine(result);
}
```


Generic Collection

We can Remove element based on the value with Remove() or based on the index with RemoveAt().

```
List<string> dogs = new List<string>()
{ "maltese", "otterhound",
  "rottweiler", "bulldog", "whippet" };
dogs.Remove("bulldog"); // Remove bulldog
foreach (string dog in dogs)
{
    Console.WriteLine(dog);
}
// Contains: maltese, otterhound, rottweiler, whippet

dogs.RemoveAt(1); // Remove second dog

foreach (string dog in dogs)
{
    Console.WriteLine(dog);
}
// Contains: maltese, rottweiler, whippet
```

Generic Collection

CopyTo () copies the list elements to an array, before coping we must make sure the array is properly allocated.

```
var list = new List<int>() { 5, 6, 7 };

// Create an array with length of three.
int[] array = new int[list.Count];

// Copy the list to the array.
list.CopyTo(array);

// Display.
Console.WriteLine(array[0]);
Console.WriteLine(array[1]);
Console.WriteLine(array[2]);
```

Generic Collection

Converting a List to an array

```
List<string> list1 = new List<string>();  
list1.Add("one");  
list1.Add("two");  
list1.Add("three");  
list1.Add("four");  
list1.Add("five");  
  
// Convert to string array.  
string[] array1 = list1.ToArray();  
Test(array1);  
}  
  
static void Test(string[] array)  
{  
    Console.WriteLine("Array received: " + array.Length);  
}
```

Generic Collection

Dictionary:

- A Dictionary stores Key-Value pairs where the key must be unique
- When defining the Dictionary we have provide the type of the key and the type of the value
- Before adding a KeyValuePair into a dictionary, check that the key does not exist using the ContainsKey() method
- Dictionary cannot include duplicate or null keys, where as values can be duplicated or set as null. Keys must be unique otherwise it will throw a runtime exception.
- Use a foreach or for loop to iterate a dictionary
- Use dictionary indexer to access individual item

Generic Collection

Important Methods of Dictionary

| Method | Description |
|---------------|--|
| Add | Adds an item to the Dictionary collection. |
| Add | Add key-value pairs in Dictionary<TKey, TValue> collection. |
| Remove | Removes the first occurrence of specified item from the Dictionary<TKey, TValue>. |
| Remove | Removes the element with the specified key. |
| ContainsKey | Checks whether the specified key exists in Dictionary<TKey, TValue>. |
| ContainsValue | Checks whether the specified key exists in Dictionary<TKey, TValue>. |
| Clear | Removes all the elements from Dictionary<TKey, TValue>. |
| TryGetValue | Returns true and assigns the value with specified key, if key does not exists then return false. |

Generic Collection

Adding elements into Dictionary

```
static void Main(string[] args)
{
    //Defining and Adding elements into Dictionary
    Dictionary<string, long> phonebook = new Dictionary<string, long>();
    phonebook.Add("Alex", 4154346543);
    phonebook["Jessica"] = 4159484588;
}
```

```
IDictionary<string, long> phonebook1= new Dictionary<string, long>();
```

```
phonebook1.Add(new KeyValuePair<string, long>("John", 4017653929));
phonebook1.Add(new KeyValuePair<String, long>("Jamie", 6012347898));
```

```
//The following is also valid
phonebook1.Add("Alvin", 6053892345);
```

Generic Collection

Searching elements in the Dictionary

ContainsKey: It searches the given Key is present in the Dictionary if it's there it returns true else returns false.

```
Dictionary<string, int> dictionary = new Dictionary<string, int>();

dictionary.Add("apple", 1);
dictionary.Add("windows", 5);

// See whether Dictionary contains this string.
if (dictionary.ContainsKey("apple"))
{
    int value = dictionary["apple"];
    Console.WriteLine(value);
}

// See whether it contains this string.
if (!dictionary.ContainsKey("acorn"))
{
    Console.WriteLine(false);
}
```

Generic Collection-Dictionary

Accessing elements in Dictionary using foreach loop will evaluate each element individually, and an index is not needed.

- Foreach loop returns an Enumeration in the result set.
- A foreach-loop on KeyValuePair is faster than the looping over Keys and accessing values in the loop body.

```
//Accessing using foreach
Dictionary<string, int> d = new Dictionary<string, int>()
{
    {"cat", 2},
    {"dog", 1},
    {"llama", 0},
    {"iguana", -1}
};

// Loop over pairs with foreach.
foreach (KeyValuePair<string, int> pair in d)
{
    Console.WriteLine("{0}, {1}", pair.Key, pair.Value);
}

// Use var keyword to enumerate dictionary.
foreach (var pair in d)
{
    Console.WriteLine("{0}, {1}", pair.Key, pair.Value);
}
```


Generic Collection

We cannot perform sorting method directly on Dictionary rather we can sort the keys in a separate List collection and loop it.

```
//Sorting Dictionary
```

```
// Acquire keys and sort them.
```

```
var list = d.Keys.ToList();  
list.Sort();
```

```
// Loop through keys.
```

```
foreach (var key in list)
```

```
{
```

```
    Console.WriteLine("{0}: {1}", key, d[key]);
```

```
}
```

Delegates

- Delegate is a type which safely encapsulates a method
- A Delegate is a type that references a method
- Once a delegate is assigned to a method, it behaves exactly like that method
- The Delegate method can be used like any other method with parameter and return value
- The type of the delegate is defined by name of the delegate
- A delegate does not care about class of the object that it references, only matter is that the method signature should be the same as of delegate

Delegates have following properties,

- Delegates are similar to C++ function pointer but it is type safe in nature.
- Delegate allows method to pass as an argument.
- Delegate can be chained together.
- Multiple methods can be called on a single event.

Delegates

Syntax of a delegate

Step 1 - Declaration

Delegate is getting declared here.

```
Modifier delegate return_type delegate_name ( [Parameter....])
```

Step 2 - Instantiation

Object of delegate is getting created as passing method as argument

```
Delegate_name delg_object_name = new Delegate_name( method_name);
```

Step 3 - Invocation

Delegate is getting called here.

```
Delg_object_name([parameter....]);
```

Delegates

```
public delegate int AddDelegate(int num1, int num2);

static void Main(string[] args)
{
    // Creating method of delegate, and passing Function Add as argument.
    AddDelegate funct1 = new AddDelegate(Add);
    // Invoking Delegate.....
    int k = funct1(7, 2);
    Console.WriteLine(" Sumation = {0}", k);
    Console.Read();
}
```

Anonymous Method

- Anonymous method is a method without any name
- Anonymous method can be defined using the delegate keyword
- Anonymous method must be assigned to a delegate
- Anonymous method can be passed as a parameter
- Anonymous method can be used as event handlers
- Anonymous method can access outer variables or functions

Limitations of Anonymous Method:

- It cannot contain jump statement like goto, break or continue.
- It cannot access ref or out parameter of an outer method
- It cannot be used on the left side of the is operator

Anonymous Method

- Inline Anonymous Method

```
static void Main(string[] args)
{
    SqaureRoot doSquare;
    //Inline Anonymous method
    doSquare = x => Math.Sqrt(x);
    Console.WriteLine(doSquare(122));
}
```

Anonymous Method

- Let us see an example for Inline and multi-line anonymous methods

```
public delegate double SqaureRoot(int value1);
public delegate int Calculate(int value1, int value2, int value3);
class Demo1
{
    static void Main(string[] args)
    {
        SqaureRoot doSquare;
        doSquare = x => Math.Sqrt(x);
        Console.WriteLine(doSquare(122));
        Calculate cal;
        cal = (x, y, z) =>
        {
            Console.WriteLine("Multiple");
            var result = x * y * z;
            return result;
        };
    }
};
```

Lambda Expression

- A lambda expression is an unnamed methods written in the place of a delegate instance.
- The lambda expression is a shorter way of representing anonymous method using some special syntax

The following anonymous method checks if student is teenager or not.

```
delegate(Student s) { return s.Age > 12 && s.Age < 20; };
```

The above anonymous method can be represented using a Lambda Expression as below

```
s => s.Age > 12 && s.Age < 20
```


Extension Method

- Extension methods enable you to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.
- Extension methods can be added to your own custom class, .NET framework classes, or third party classes or interfaces
- An extension method is a special kind of static method, but they are called as if they were instance methods on the extended type.
- An extension method is a static method of a static class, where the "this" modifier is applied to the first parameter. The type of the first parameter will be the type that is extended.
- Extension methods are only in scope when you explicitly import the namespace into your source code with a using directive.

Extension Method

Important points for the use of Extension Methods:

- An extension method must be defined in a top-level static class.
- An extension method with the same name and signature as an instance method will not be called.
- Extension methods cannot be used to override existing methods.
- The concept of extension methods cannot be applied to fields, properties or events.
- Overuse of extension methods is not a good style of programming.

Extension Method

Let us see an example for extension method `IsGreaterThan()` is an extension method for `int` type, which returns `true` if the value of the `int` variable is greater than the supplied integer parameter.

Example: Extension Method

```
int i = 10;  
  
bool result = i.IsGreaterThan(100); //returns false
```

The `IsGreaterThan()` method is not a method of `int` data type (`Int32` struct). It is an extension method written by the programmer for the `int` data type.

The `IsGreaterThan()` extension method will be available throughout the application by including the namespace in which it has been defined.

Extension Method

- To define an extension method, first of all, define a static class

we have created an `IntExtensions` class under the `ExtensionMethods` namespace in the following example. The `IntExtensions` class will contain all the extension methods applicable to `int` data type.

Example: Create a Class for Extension Methods

```
namespace ExtensionMethods
{
    public static class IntExtensions
    {
    }
}
```

The first parameter of the extension method specifies the type on which the extension method is applicable. Since we are going to use this extension method on `int` type the first parameter must be `int` type preceded with `this` modifier.

Extension Method

Example: Define an Extension Method

```
namespace ExtensionMethods
{
    public static class IntExtensions
    {
        public static bool IsGreaterThan(this int i, int value)
        {
            return i > value;
        }
    }
}
```

Now, you can include the ExtensionMethods namespace wherever you want to use this extension method.

Extension Method

```
using ExtensionMethods;

class Program
{
    static void Main(string[] args)
    {
        int i = 10;

        bool result = i.IsGreaterThan(100);

        Console.WriteLine(result);
    }
}
```

Extension Method

- Extension methods are additional custom methods which were originally not included with the class.
- Extension methods can be added to custom, .NET Framework or third party classes, structs or interfaces.
- The first parameter of the extension method must be of the type for which the extension method is applicable, preceded by the **this** keyword.
- Extension methods can be used anywhere in the application by including the namespace of the extension method.

Partial Class and Partial Methods

Class in C# resides in a separate physical file with a .cs extension. C# provides the ability to have a single class implementation in multiple .cs files using the ***partial modifier*** [keyword](#).

The *partial* modifier can be applied to a class, method, interface or structure.

Example : MyPartialClass splits into two files, PartialClassFile1.cs and PartialClassFile2.cs

Example: PartialClassFile1.cs

```
public partial class MyPartialClass
{
    public MyPartialClass()
    {
    }

    public void Method1(int val)
    {
        Console.WriteLine(val);
    }
}
```

Example: PartialClassFile2.cs

```
public partial class MyPartialClass
{
    public void Method2(int val)
    {
        Console.WriteLine(val);
    }
}
```


Partial Class and Partial Methods

- MyPartialClass in PartialClassFile1.cs defines the constructor and one public method, Method1, whereas PartialClassFile2 has only one public method, Method2.
- The compiler combines these two partial classes into one class as below:

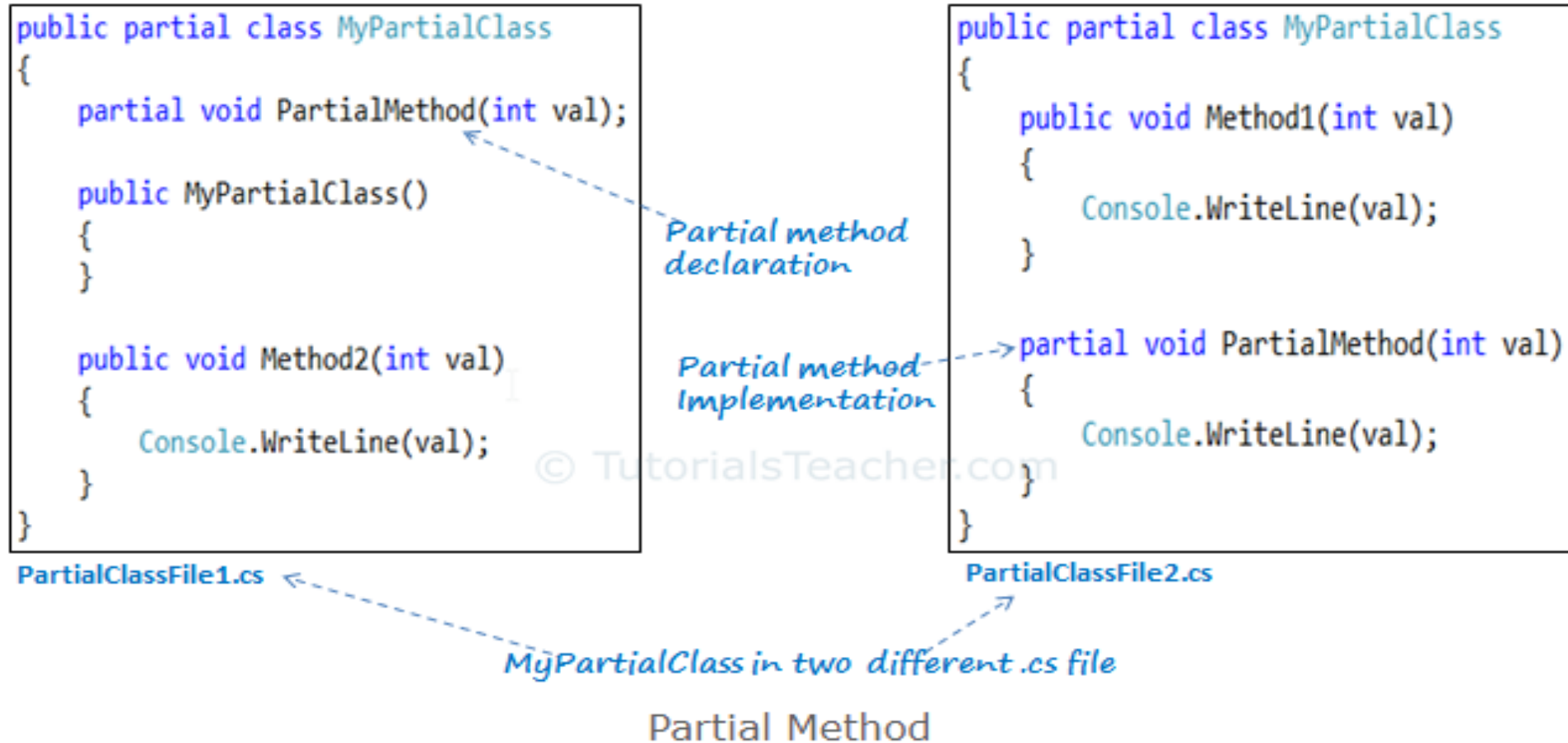
```
public class MyPartialClass
{
    public MyPartialClass()
    {
    }

    public void Method1(int val)
    {
        Console.WriteLine(val);
    }

    public void Method2(int val)
    {
        Console.WriteLine(val);
    }
}
```

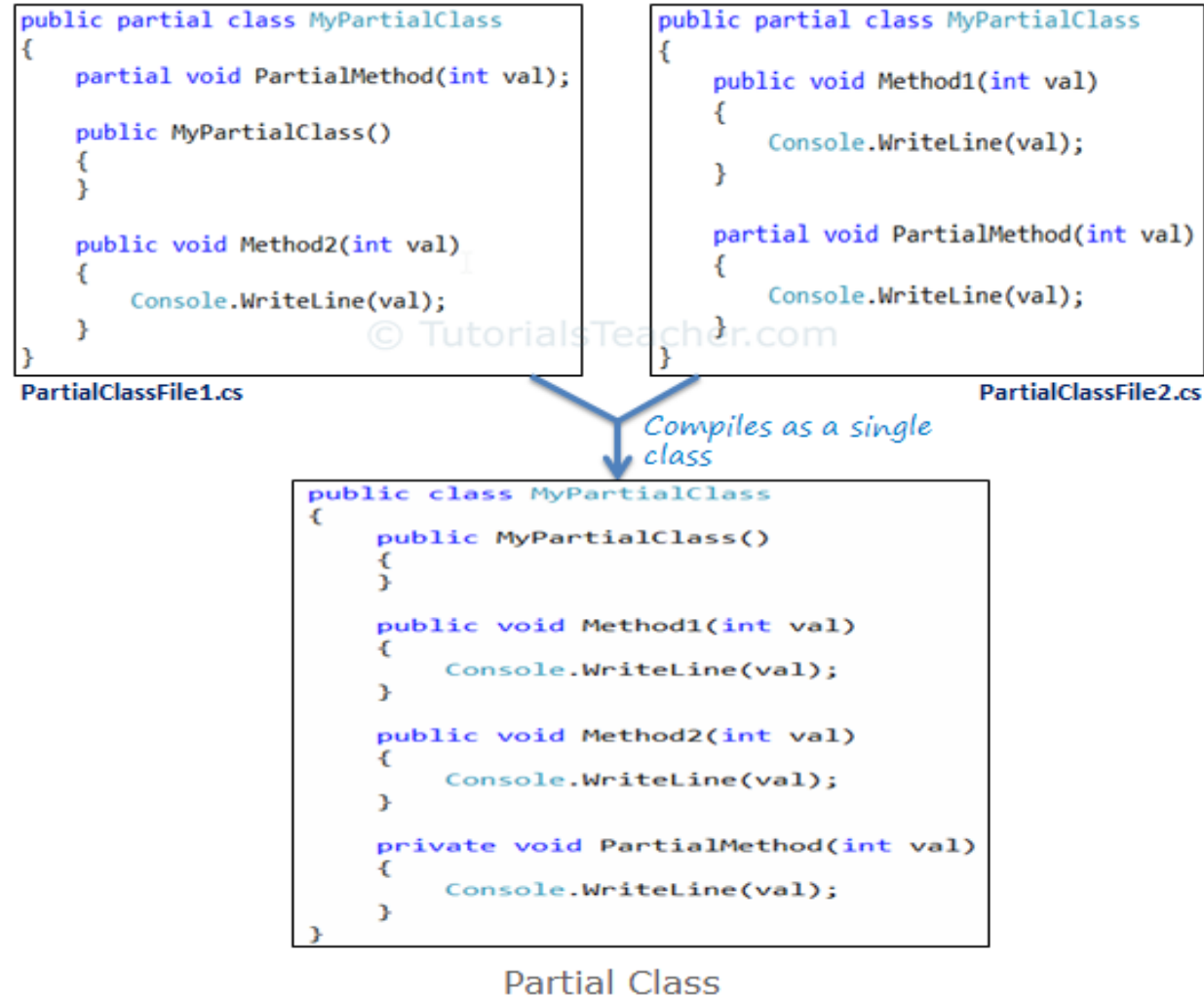
Partial Class and Partial Methods

The following image illustrates partial class and partial methods:



Partial Class and Partial Methods

The compiler combines the two partial classes into a single final class:



Partial Class and Partial Methods

Advantages of Partial Class:

- Multiple developers can work simultaneously with a single class in separate files.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. For example, Visual Studio separates HTML code for the UI and server side code into two separate files: .aspx and .cs files.

Partial Methods

- A partial method must be declared in one of the partial classes.
- A partial method may or may not have an implementation.
- If the partial method doesn't have an implementation in any part then the compiler will not generate that method in the final class.

Requirement of partial method:

- The partial method declaration must begin with the partial modifier.
- The partial method can have a ref but not an out parameter.
- Partial methods are implicitly private methods.
- Partial methods can be static methods.
- Partial methods can be generic.

Sealed Class

- Sealed class in C# with the sealed keyword cannot be inherited. In the same way, the sealed keyword can be added to the method.
- When you use sealed modifiers in C# on a method, then the method loses its capabilities of overriding. The sealed method should be part of a derived class and the method must be an overridden method.

Let us see an example of sealed class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Demo {
    class Program {
        static void Main(string[] args) {
            Result ob = new Result();
            string str = ob.Display();

            Console.WriteLine(str);
            Console.ReadLine();
        }
    }

    public sealed class Result {
        public string Display() {
            return "Passed";
        }
    }
}
```

Static Class

Static Class:

- The **static** modifier makes an item non-instantiable, it means the static item cannot be instantiated. If the static modifier is applied to a class then that class cannot be instantiated using the **new** keyword.
- If the **static** modifier is applied to a variable, method or property of class then they can be accessed without creating an object of the class, just use **className.propertyName**, **className.methodName**.

Static Constructor:

A static or non-static class can have a static constructor without any access modifiers like public, private, protected, etc.

- A static constructor in a non-static class runs only once when the class is instantiated for the first time.
- A static constructor in a static class runs only once when any of its static members accessed for the first time.