Course Description:
Basic Principles of Object Oriented Programming in C Sharp Programming Language over .NET Framework.

Course Duration: 10 hours

Course Goals and Objectives:
After completing this course, trainee will be able to understand the basic concepts of C# programming language like Data Types, Inheritance, Polymorphism, Multithreading, Type Conversion, Garbage Collection Features and String and Array Data Structures.

## Data Types

There are two kinds of types in C#: value types and reference types.
public abstract class ValueType.
public class Object.

Variables of value types directly contain their data and stored on stack. With value types, the variables each have their own copy of the data, and it isn't possible for operations on one to affect the other (except for ref and out parameter variables). A value type derives from System.ValueType and contains the data inside its own memory allocation. C#'s value types are further divided into simple types, enum types, struct types, and nullable value types.

1- Simple types
    Signed integral: sbyte, short, int, long
    Unsigned integral: byte, ushort, uint, ulong
    Unicode characters: char
    IEEE binary floating-point: float, double
    High-precision decimal floating-point: decimal
    Boolean: bool
2- Enum types
    User-defined types of the form enum E {...}
3- Struct types
    User-defined types of the form struct S {...}
4- Nullable value types
    Extensions of all other value types with a null value

## Reference Types

Variables of reference types store references to their data known as objects. With reference types, it's possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. Reference types are stored in managed heaps. A reference type extends System.Object and points to a location in the memory that contains the actual data.

C#'s reference types are further divided into class types, interface types, array types, and delegate types.

1- Class types
   Ultimate base class of all other types: object
   Unicode strings: string
   User-defined types of the form class C {...}
2- Interface types
   User-defined types of the form interface {...}
3- Array types
   Single- and multi-dimensional, for example, int[ ] and int[ , ]
4- Delegate types
   User-defined types of the form delegate int D(int x)

## Member Overloading

Member overloading means creating two or more members on the same type that differ only in the number or type of parameters but have the same name. For example, in the following, the WriteLine Library method of System.Console class is overloaded

```
public static class Console
{
        public void WriteLine();
        public void WriteLine(string value);
        public void WriteLine(bool value);
}
```

Because only methods, constructors, and indexed properties can have parameters, only those members can be overloaded. Overloading is one of the most important techniques for improving usability, productivity, and readability of reusable libraries. Overloading on the number of parameters makes it possible to provide simpler versions of constructors and methods. Overloading on the parameter type makes it possible to use the same member name for members performing identical operations on a selected set of different types.

## Method Overloading

Method overloading allows programmers to use multiple methods with the same name. The methods are differentiated with their number of arguments and type of method arguments. Method overloading is an example of the polymorphism feature of an object oriented programming language. Method overloading can be achieved by the following:

1- By changing number of parameters in a method.
2- By changing the order of parameters in a method.
3- By using different data types for parameters.

Example

```
public class MethodOveloading
{
        public int add(int a, int b)
        {
                return a + b;
        }
        public int add(float a, float b)
        {
                return a + b;
        }
        public int add(int a, int b, int c)
        {
                return a+b+c;
        }
}
```

The in, ref, and out keywords are not considered part of the method signature for the purpose of overload resolution. Therefore, methods cannot be overloaded if the only difference is that one method takes a ref or in argument and the other takes an out argument. The following code, for example, will not compile:

```
class OverloadingInRefOut
{
        /*
                Compiler error : "Cannot define overloaded methods that differ
                only on ref and out"
        */
        public void CallByRef(out int i) {
            // Code Not Shown Here
        }
        public void CallByRef(ref int i) {
            // Code Not Shown Here
        }
        public void CallByRef(in int i) {
            // Code Not Shown Here
        }
}
```

Overloading is legal, however, if one method takes a ref, in, or out argument and the other has none of those modifiers, like this:

```
class OverloadingInRefOut
{
public void CallByRef(int num) {
     // Code Not Shown Here
}
public void CallByRef(ref int num) {
     // Code Not Shown Here
}
public void CallByRef(float num) {
    // Code Not Shown Here
}
public void CallByRef(out float num) {
    // Code Not Shown Here
}
}
```

## Type Conversion or Casting

Type Conversion or casting is a process of Copying a value of one type into a variable or method parameter of another type.
For example,
1- Assign the value of an integer variable to a variable of double type.
2- Pass an integer variable to a method parameter of type double.
2- Assign a class variable to a variable of an interface type.

These kinds of operations are called type conversions. In C#, you can perform the following kinds of conversions:

**1-Implicit conversions:** No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
For example-
1- Float to Double Conversion.

2- Integer to Double Conversion.
3- Byte to Integer Conversion.

```
int num = 200;
float fNum = num;
```

In above assignment Integer 200 is converted to Float 200.0f and asdigned to fNum. After execution of above code, values of num and fNum will be 200 and 200.0000 respectively.

**2-Explicit Conversion (Cast):** Explicit conversions require a cast expression. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.

```
        int numInt;
        float  numFloat= 10.10;
        numInt = (float) numFloat
```

Explicit Cast may throw Overflow exception when value of right side variable is more than the maximum value of left side variable.

3- User-defined conversions: User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship.

4- Conversions with helper classes: To convert between non-compatible types, such as string form of numeric data is converted to 32 bit integer data type using the Parse() method of Int32 class.

```
string str = "123";
int num = Int32.Parse (str);
double dNum = 23.15;
int num;

try
{
     Num =System.Convert.ToInt32(dNum);
}
catch (System.OverflowException) {
        System.Console.WriteLine( "Overflow in double to int conversion.");
}
```

## Conversion Functions

The static methods of the Convert class are primarily used to support conversion from one base data type to another base data types in the .NET Framework. The supported base type conversion functions are

| | | | |
|---|---|---|---|
| ToBoolean(), | ToChar(), | ToSByte(), | ToByte(), |
| ToInt16(), | ToInt32(), | ToInt64(), | ToUInt16(), |
| ToUInt32(), | ToUInt64(), | ToSingle(), | ToDouble(), |
| ToDecimal(), | ToDateTime() | ToString(). | |

A conversion method exists to convert every base type to every other base type. However, the actual call to a particular conversion method can produce one of five outcomes, depending on the value of the source base type and the target base type at run time. These five outcomes are:

**1-No Conversion-** This occurs when an attempt is made to convert from a type to itself (Example- by calling Convert.ToInt32() with an argument of type Int32). In this case, the method simply returns an instance of the original type.

**2-OverflowException**- This occurs when a narrowing conversion results in a loss of data. For example, trying to convert a Int32 instance whose value is 10000 to a Byte type throws an OverflowException because 10000 is outside the range of the Byte data type.

**3-Successful conversion**- For conversions between two different base types not listed in the previous outcomes, all widening conversions as well as all narrowing conversions that do not result in a loss of data will succeed and the method will return a value of the targeted base type.

**4-InvalidCastException**- This occurs when a particular conversion is not supported. An InvalidCastException is thrown for the following conversions:

- Conversions from Char to Boolean, Single, Double, Decimal or DateTime.
- Conversions from Boolean, Single, Double, Decimal or DateTime to Char.
- Conversions from DateTime to any other type except String.

**5-FormatException-** This occurs when the attempt to convert a string value to any other base type fails because the string is not in the proper format. The exception is thrown for the following conversions:

- A string to be converted to a Boolean value does not equal Boolean True String or Boolean False String.
- A string to be converted to a Char value consists of multiple characters.
- A string to be converted to any numeric type is not recognized as a valid number.
- A string to be converted to a DateTime is not recognized as a valid date and time value.

## Boxing

Boxing is used to store value types in the garbage-collected heap. Boxing is an implicit conversion of a value type to the type object or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

## Unboxing

Unboxing is an explicit conversion from the type object to a value type or from an interface type to a value type that implements the interface. An unboxing operation consists of:

1- Checking the object instance to make sure that it is a boxed value of the given value type.
2- Copying the value from the instance into the value-type variable.

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

1- Attempting to unbox null causes a NullReferenceException.
2- Attempting to unbox a reference to an incompatible value type causes an InvalidCastException.

```
class TestUnboxing
{
  static void Main()
  {
        int i = 123;
        object o = i; // implicit boxing
        try
        {
            int j = (short)o;  // attempt to unbox
            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Incorrect unboxing.", e.Message);
        }
    }
}
```

**Features of Boxing and Unboxing:**

1- Unboxing extracts the value type from the object.
2- Boxing is implicit; unboxing is explicit.
3- When the common language runtime (CLR) boxes a value type, it wraps the value inside a System.Object instance and stores it on the managed heap.

In relation to simple assignments, boxing and unboxing are computationally expensive processes. When a value type is boxed, a new object must be allocated and constructed.

**Final Member** (No Such Member Exists in C#)
There is no final keyword in C# but sealed keyword is used same as final in java.

**Sealed Class and Members**
In C# sealed modifier means not allow to change. It can be applied to classes as well as methods with different meanings. Modifier sealed is not applied to Fields, Properties, Indexers and constructors.

**1-Sealed Class-** When the sealed modifier is applied to a class, it prevents other classes to be inherited from it. It is an error to use the abstract modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.
In the following example, class B inherits from class A, but no class can inherit from class B.

```
class A
{
}
sealed class B : A {
        // This Class Never be a Base Class
}
class C : B
{
        // Compile Time Error, as attempting to inherit sealed class.
}
```

**2-Sealed Method**- When sealed modifier is applied to overridden virtual method or property in any class, this enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.

**Example:-** Class Z inherits from Y but Z cannot override the virtual function F1() that is declared in X and sealed in Y.

```
class X
{
        protected virtual void F1()
        {
                Console.WriteLine("X.F");
        }
        protected virtual void F2()
        {
                Console.WriteLine("X.F2");
        }
}
class Y : X
{
        sealed protected override void F1()
        {
                Console.WriteLine("Y.F");
        }
        protected override void F2()
        {
                Console.WriteLine("Y.F2");
        }
}

class Z : Y
{
        // Attempting to override F1() causes compiler error.
        protected override void F1()
        {
                Console.WriteLine("Z.F");
        }
        // Overriding F2 is allowed.
        protected override void F2()
        {
                Console.WriteLine("Z.F2");
        }
}
```

**Passing an argument by reference**

There are three different ways and three different purposes to pass method argument by reference which are illustrated as follows:

**1-The ref Parameter**

The ref keyword indicates a value that is passed by reference. When used in a method's parameter list, the ref keyword indicates that an argument is passed by reference, not by value. The ref keyword makes the formal parameter an alias for the argument, which must be a variable. In other words, any operation performed on the parameter is made on the argument. To use a ref parameter, both the method definition and the calling method must explicitly use the ref keyword, as shown in the following example.

```
void Method(ref int refArg)
{
   refArg = refArg + 45;
}

int num = 10;   // initialisation mandatory
Method(ref num);
Console.WriteLine(num);      // Output: 55
```

An argument that is passed to a ref parameter or in parameter must be initialized before it is passed. This differs from out parameters, whose arguments do not have to be explicitly initialized before they are passed. Members of a class can't have signatures that differ only by ref, in, or out. A compiler error occurs if the only difference between two members of a type is that one of them has a ref parameter and the other has an out, or in parameter. The following code, for example, doesn't compile.

```
class RefOutParameter
{
        /*
                Compiler error: "Cannot define overloaded methods that differ
                only on ref and out".
```

```
    */
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

## 2-The out Parameter (Output Parameter)

The out keyword causes arguments to be passed by reference. It makes the formal parameter an alias for the argument, which must be a variable. In other words, any operation performed on the parameter is made on the argument.

The out Parameter is differs from ref parameter and in parameter, its arguments do not have to be explicitly initialized before it is passed. To use an out parameter, both the method definition and the calling method must explicitly use the out keyword same as both ref parameter and in parameter.

**Example:**

```
void MethodOutArg (out int number)
{
    number = 555; // Must be modified here
}

// Variable num Need not to be initialized.
int num;
this.MethodOutArg (out num);
Console.WriteLine(num);

//Output:  value of num is now 555
```

## 3-The in Parameter (Input Parameter)

The in keyword causes arguments to be passed by reference. It makes the formal parameter an const alias for the argument, which must be a variable. It is like the ref or out keywords, except that in arguments cannot be modified by the called method. Whereas ref arguments may be modified, out arguments must be modified by the called method, and those modifications are observable in the calling context.

## Difference among ref, in and out

- Out Parameter must be modified by called method but need not to be initialised before it is passed.
- In Parameter need not to be modified by called method but must be initialised before it is passed.

- Ref Parameter may be modified by called method and must be initialised before it is passed.

**Abstract Class**

A class can be declared abstract. An abstract class contains abstract methods that have a signature definition but no implementation. Abstract classes cannot be instantiated. They can only be used through derived classes that implement the abstract methods.

```
class Shape
{
	public abstract int GetArea();
}
class Square : Shape
{
	int side;
	public Square(int n)
	{
		side = n;
	}

	// GetArea method is required to avoid a compile-time error.
	public override int GetArea()
	{
		return  side * side;
	}
	static void Main()
	{
		var sq = new Square(12);
		Console.WriteLine($"Area of the square = {sq.GetArea()}");
	}
}
```

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods. When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method.

```
public class D
{
        public virtual void DoWork(int i)
        {
                // Original implementation.
        }
}

public abstract class E : D
{
        /*
                Abstract class overrides the virtual method of its base class D
                and declares it abstract.
        */
        public abstract override void DoWork(int i);
}

public class F : E
{
        public override void DoWork(int i)
        {
                // New implementation.
        }
}
```

**Abstract classes have the following features:**

1 An abstract class cannot be instantiated.
2 An abstract class may contain abstract methods and accessors.
3 It is not possible to modify an abstract class with the sealed modifier because the two modifiers have opposite meanings.
4 The sealed modifier prevents a class from being inherited and the abstract modifier requires a class to be inherited.
5 A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

## Interface

An interface contains definitions for a group of related functionalities that a non-abstract class or a struct must implement. An Interface can be declared as either public or internal inside a namespace and can't have any other access modifiers. An Interface can have any access modifier when declared inside a class or structure.

Interfaces can contain instance methods, properties, events, indexers, or any combination of those four member types. Interfaces may contain static constructors, fields, constants, or operators. An interface may define static methods, which must have an implementation.

Interfaces cannot be initialized directly as they do not have a definition for members, they should be implemented by either a class or structure. By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

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.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface declares but doesn't provide a default implementation for. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

```
public interface ILogger
{
        void WriteLog(String Message);
        void DisplayLog(List Messages);
}
public class  LogMessageUpdate: IEquatable
{
        // Implementation of methods of interface
        public void WriteLog(string Message)
        {
                Console.WriteLine(Message);
        }
        void DisplayLog (List logMessage)
        {
                foreach(logMessage in logMessages) {
                        Console.WriteLine(logMessage)
                }
        }
}
```

**An interface has the following properties:**

1 An interface is typically like an abstract base class with only abstract members. Any class or struct that implements the interface must implement all its members. Optionally, an interface may define default implementations for some or all of its members.
2 An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
3 A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

**Example:** Implement Properties of Interface.

```
interface IEmployee
{
        string Name { get; set; }
        int Counter { get; }
}

public class Employee : IEmployee
{
        public static int totalEmp;
        private string _name;
        private int _counter;

        public Employee()
        {
                _counter = ++totalEmp;
        }

        public string Name
        {
                get { return  _name; }
                set { _name = value; }
        }

        public int Counter
        {
                get { _counter; }
        }
}
```

Example: Implement Multiple Interfaces

```
interface IEnglishDimensions
```

```
{
	float Length();
	float Width();
}
interface IMetricDimensions
{
	float Length();
	float Width();
}

/*
	Declare the Box class that implements the two interfaces:
	IEnglishDimensions and IMetricDimensions
*/

class Box : IEnglishDimensions, IMetricDimensions
{
	float lengthInches;
	float widthInches;
	public Box(float lengthInch, float widthInch)
	{
		this.lengthInches = lengthInches;
		this.widthInches = widthInches;
	}

	// Implementation of IEnglishDimensions:
	float IEnglishDimensions.Length()
	{
		return lengthInches;
	}

	float IEnglishDimensions.Width()
	{
		return widthInches;
	}
	// implementation of IMetricDimensions
	float IMetricDimensions.Length()
	{
		return (lengthInches * 2.54f);
	}
```

```
    float IMetricDimensions.Width()
    {
        return widthInches * 2.54f;
    }
}
```

## Garbage Collection

When you create any object in C#, CLR (Common Language Runtime) allocates memory for the object from heap. This process is repeated for each newly created object, but there is a limitation to everything, Memory is not un-limited and we need to clean some used space in order to make room for new objects, Here, the concept of garbage collection is introduced, Garbage collector manages allocation and reclaiming of memory. GC (Garbage Collector) makes a trip to the heap and collects all objects that are no longer used by the application and then makes them free from memory.
In the common language runtime (CLR), the garbage collector (GC) serves as an automatic memory manager. It provides the following benefits:

1 Allocates objects on the managed heap efficiently.
2 Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations.
3 Provides memory safety by making sure that an object cannot use the content of another object.

## The Finalize method

The Finalize method is used to perform cleanup operations on unmanaged resources held by the current object before the object is destroyed. The method is protected and therefore is accessible only through this own class or through a derived class.

## How finalization works

The Object class provides no implementation for the Finalize method, and the garbage collector does not mark types derived from Object for finalization unless they override the Finalize method. If a type does override the Finalize method, the garbage collector adds an entry for each instance of the type to the finalization queue. The finalization queue contains entries for all the objects in the managed heap whose finalization code must run before the garbage collector can reclaim their memory. The garbage collector then calls the Finalize method automatically under the following conditions:

1 After the garbage collector has discovered that an object is inaccessible, unless the object has been exempted from finalization by a call to the GC.SuppressFinalize() method.
2 On .NET Framework only, during shutdown of an application domain, unless the object is exempt from finalization. During shutdown, even objects that are still accessible are finalized.
3 Finalize is automatically called only once on a given instance, unless the object is re-registered by using a mechanism such as GC.ReRegisterForFinalize() and the GC.SuppressFinalize () method has not been subsequently called.

**Finalize operations have the following limitations:**
1 The exact time when the finalizer executes is undefined. To ensure deterministic release of resources for instances of your class, implement a Close method or provide a IDisposable.Dispose implementation.
2 The finalizers of two objects are not guaranteed to run in any specific order, even if one object refers to the other.
3 The thread on which the finalizer runs is unspecified.

## Overriding the Finalize method

The C# compiler does not allow you to override the Finalize method. Instead, you provide a finalizer by implementing a destructor for your class. A C# destructor automatically calls the destructor of its base class.

You should override Finalize for a class that uses unmanaged resources, such as file handles or database connections that must be released when the managed object that uses them is discarded during garbage collection. You should not implement a Finalize method for managed objects because the garbage collector releases managed resources automatically.

In addition, one should override the Finalize method for reference types only. The common language runtime only finalizes reference types. It ignores finalizers on value types.

## Implementing Dispose() method of IDisposable Interface

You implement a Dispose method to release unmanaged resources used by your application. The .NET garbage collector does not allocate or release unmanaged memory.

Because garbage collection is non-deterministic, you do not know precisely when the garbage collector performs finalization. To release resources immediately, you can also choose to implement the dispose pattern and the IDisposable interface. The IDisposable.Dispose implementation can be called by consumers of your class to free unmanaged resources

## Dispose() and Dispose(Boolean) Methods.

The IDisposable interface requires the implementation of a single parameterless method, Dispose. However, the dispose pattern requires two Dispose methods to be implemented:

1 A public non-virtual IDisposable.Dispose() implementation that has no parameters.
2 A protected virtual Dispose method whose signature is: protected virtual void Dispose(bool disposing)

### The Dispose() overload

Because the public, non-virtual parameterless Dispose () method is called by a consumer of the type, its purpose is to free unmanaged resources and to indicate that the finalizer, if one is present, doesn't have to run. Because of this, it has a standard implementation:

```
public void Dispose()
{
        // Dispose of unmanaged resources. Dispose(true);
        // Suppress finalization. GC.SuppressFinalize(this);
}
```

The Dispose method performs all object cleanup, so the garbage collector no longer needs to call the object's Object.Finalize() override.
Therefore, the call to the SuppressFinalize () method prevents the garbage collector from running the finalizer. If the type has no finalizer, the call to GC.SuppressFinalize() has no effect.

### The Dispose(Boolean) overload

In the second overload, the disposing parameter is a Boolean that indicates whether the method call comes from a Dispose method (its value is true) or from a finalizer (its value is false).

The body of the method consists of two blocks of code:
1- A block that frees unmanaged resources. This block executes regardless of the value of the disposing parameter.

2- A conditional block that frees managed resources. This block executes if the value of disposing is true. The managed resources that it frees can include:
   - Managed objects that implement IDisposable interface. The conditional block can be used to call their Dispose implementation.
   - Managed objects that consume large amounts of memory or consume scarce resources. Freeing these objects explicitly in the Dispose method releases them faster than if they were reclaimed non-deterministically by the garbage collector.

## Static Members

Static class members can be declared by using the static keyword before the type of field or return type of the member like field, event and property. A non-static class can contain static methods, fields, properties, or events. The static member is callable on a class even when no instance of the class has been created. The static member is always accessed by the class name, not the instance name.

Example

```
public class Car
{
        public static int NumberOfWheels = 4;
        private static int _gasTankSize = 15;

        public static int SizeOfGasTank
        {
                get  { return _gasTankSize; }
                set { _gasTankSize = value; }
        }

        public static void Drive()
        {
                // Code not shown here.
        }
                // Other non-static fields and properties...
}
```

Static Members can be accesses as

```
Car.Drive();  // Method Call
int i = Car.NumberOfWheels;  // Access Field
```

```
Car.SizeOfGasTank = 12;     // Set Property
```

**Features of a static members.**

1- Static methods and properties cannot access non-static fields and events in their containing type, and they cannot access an instance variable of any object unless it is explicitly passed in a method parameter.
2- Only one copy of a static member exists, regardless of how many instances of the class are created.
3- Static methods can be overloaded but not overridden, because they belong to the class, and not to any instance of the class.
4- Although a field cannot be declared as static const, a const field is essentially static in its behavior. It belongs to the type, not to instances of the type. Therefore, const fields can be accessed by using the same ClassName.MemberName notation that is used for static fields.
5- C# does not support static local variables (variables that are declared in method scope).
6- Static member belongs to the type, not to instances of the type.
7- Two common uses of static fields are to keep a count of the number of objects that have been instantiated, or to store a value that must be shared among all instances.

## Static Class

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, you cannot use the new operator to create a variable of the class type. Because there is no instance variable, members of a static class can only be accesses by using the class name itself.
For example, if you have a static class that is named ALOperations that has a public static method named FindGreater().

```
static class ALOperations
{
        public static void FindGreater()
        {
                // Code not shown here.
        }
}
```

you call the method as : `ALOperation.FindGreater()`

A static class can be used as a convenient container for sets of methods that just operate on input parameters and do not have to get or set any internal instance fields.
For example, in the .NET Framework Class Library, the static System.Math class contains methods that perform mathematical operations, without any requirement to store or retrieve data that is unique to a particular instance of the Math class.

```
double dub = - 3.14;
Math.Abs(dub);
Math.Floor(dub);
Math.Ceil(dub)
Math.Round(Math.Abs(dub));
```

A static constructor is only called one time, and a static class remains in memory for the lifetime of the application domain in which your program resides.

**Features of a static class:**

1- Static class contains only static members and a private constructor.
2- They cannot be instantiated. A private constructor prevents the class from being instantiated.
3- They cannot inherit from any class except Object.
4- Static classes are sealed and therefore cannot be inherited.
5- Static classes cannot contain an instance constructor; however, they can contain a static constructor.
6- Non-static classes should also define a static constructor if the class contains static members.

## Polymorphism

Polymorphism means "one name many forms". In other words, one object has many forms or has one name with multiple functionalities. Polymorphism provides the ability to a class to have multiple implementations with the same name. It is one of the core principles of Object Oriented Programming after encapsulation and inheritance.

## Types of Polymorphism.

There are two types of polymorphism in C#:

1- Static / Compile Time Polymorphism.
2- Dynamic / Runtime Polymorphism.

## Compile Time Polymorphism

It is also known as Static Binging or Early Binding. Method overloading is an example of Compile Time Polymorphism. In overloading, the method has a same name but different signatures. It is called Compile Time Polymorphism because the decision of which method is to be called is made at compile time.

Overloading is the concept in which method names are the same with a different set of parameters. In C# compiler checks the number of parameters passed and the type of parameter and make the decision of which method to call and it throw an error if no matching method is found.

**Member Overloading**

Member overloading means creating two or more members on the same type that differ only in the number or type of parameters but have the same name. Because only methods, constructors, and indexed properties can have parameters, only those members can be overloaded.

For example, in the following, the WriteLine Library method of System.Console class is overloaded.

```
public static class Console
{
        public void WriteLine();
        public void WriteLine(string value);
        public void WriteLine(bool value);
        .
        .
}
```

Overloading is one of the most important techniques for improving usability, productivity, and readability of reusable libraries. Overloading on the number of parameters makes it possible to provide simpler versions of constructors and methods. Overloading on the parameter type makes it possible to use the same member name for members performing identical operations on a selected set of different types.

**Method Overloading**

Method overloading allows programmers to use multiple methods with the same name. The methods are differentiated with their number of arguments and type of method arguments. Method overloading is an example of the polymorphism feature of an object oriented programming language.

Method overloading can be achieved by the following:

1- By changing number of parameters in a method.

2- By changing the order of parameters in a method.

3- By using different data types for parameters.

**Example**

```
public class MethodOveloading
{
        public int add(int a, int b)
        {
                return a + b;
        }
        public int add(float a, float b)
        {
                return a + b;
        }
        public int add(int a, int b, int c)
        {
                return a + b+c;
        }
}
```

The in, ref, and out keywords are not considered part of the method signature for the purpose of overload resolution. Therefore, methods cannot be overloaded if the only difference is that one method takes a ref or in argument and the other takes an out argument. The following code, for example, will not compile:

```
class OverloadingInRefOut
{
        /*
                Compiler error: "Cannot define overloaded methods that differ
                only on ref and out"
        */
        public void CallByRef (out int i)
        {
              // Code Not Shown Here
        }
        public void CallByRef(ref int i)
        {
              // Code Not Shown Here
        }
        public void CallByRef(in int i)
        {
             // Code Not Shown Here
        }
}
```

Overloading is legal, however, if one method takes a ref, in, or out argument and the other has none of those modifiers, like this:

```
class OverloadingInRefOut
{
        public void CallByRef(int num)
        {
                        // Code Not Shown Here
        }
        public void CallByRef(ref int num)
        {
                // Code Not Shown Here
        }
        public void CallByRef(float num)
        {
                // Code Not Shown Here
        }
        public void CallByRef(out float num)
        {
                // Code Not Shown Here
        }
}
```

**Runtime Polymorphism:**

Runtime Polymorphism has two distinct aspects:

1- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this polymorphism occurs, the object's declared type is no longer identical to its run-time type.

2- Base classes may define and implement virtual methods, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method.

In C#, a method in a derived class can have the same name as a method in the base class. You can specify how the methods interact by using the new and override keywords. The override modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event. An override method provides a new implementation of a member that is inherited from a base class. The method that is overridden by an override declaration is known as the overridden base method. The overridden base method must have the same signature as the override method.

You cannot override a non-virtual or static method. The overridden base method must be virtual, abstract, or override. You cannot use the new, static, or virtual modifiers to modify an override method.
An override declaration cannot change the accessibility of the virtual method. Both the override method and the virtual method must have the same access level modifier (private, protected, public and internal or any valid combination).

Two important rules:

1- By default, methods are non-virtual, and they cannot be overridden.
2- Virtual modifiers cannot be used with static, abstract, private, and override modifiers

## Override vs New

The override modifier extends the base class virtual method, and the new modifier hides an accessible base class method.

```
class BaseClass
{
        public virtual void Method1()
        {
                Console.WriteLine("Base - Method1");
        }
        public virtual void Method2()
        {
                Console.WriteLine("Base - Method2");
        }
}
class DerivedClass : BaseClass
{
        public override void Method1()
        {
                Console.WriteLine("Derived - Method1");
        }
        public new void Method2()
        {
                Console.WriteLine("Derived - Method2");
```

```
    }
}
```

Method1() is overriden in DerivedClass and addition of the Method2() in BaseClass causes a warning. The warning says that the Method2 in DerivedClass hides the Method2 in BaseClass. The warning is suppressed by using new keyword and ensure the programmer that this act is intensionally done.

## What is Thread?

Generally, a Thread is a lightweight process. In simple words, we can say that a Thread is a unit of a process that is responsible for executing the application code. By default, every process has at least one thread which is responsible for executing the application code and that thread is called as Main Thread. So, every application by default is a single-threaded application.

## The drawbacks of Single-Threaded Applications?

In a single thread application, all the logic or code present in the program will be executed by a single thread only i.e. the Main thread. For example, if we have three methods in our application and if all these three methods are going to be called from the Main method. Then the main thread is responsible to execute all these three methods sequentially i.e. one by one. It will execute the first method and once it completes the execution of the first method then only it executes the second method and so on.

## Multithreading

Multithreading or free-threading is the ability of an operating system to concurrently run programs that have been divided into subcomponents, or threads. Technically, multithreaded programming requires a multitasking/multithreading operating system, such as GNU/Linux, Windows NT/2000 or OS/2; capable of running many programs concurrently
With .NET, you can write applications that perform multiple operations at the same time. Operations with the potential of holding up other operations can execute on separate threads, a process known as multithreading or free threading.

Applications that use multithreading are more responsive to user input because the user interface stays active as processor-intensive tasks execute on separate threads. Multithreading is also useful when you create scalable applications, because you can add threads as the workload increases.

```
using System;
using System.Threading;

public class ThreadWork
{
	public static void DoWork()
	{
		for(int i = 0; i<3;i++)
		{
			Console.WriteLine("Working Thread...");
			Thread.Sleep(100);
		}
	}
}

class ThreadTest
{
	public static void Main()
	{
		Thread	thread1	=	new	Thread(ThreadWork.DoWork);
	thread1.Start();
		for (int i = 0; i<3; i++)
		{
			Console.WriteLine("In main."); Thread.Sleep(100);
		}
	}
}
```

The example displays output like the following:

```
/*
In main.
Working thread...
In main.
Working thread...
In main.
Working thread...
*/
```

Once a thread is in the ThreadState.Running state, the operating system can schedule it for execution. The thread begins executing at the first line of the method represented by the ThreadStart or ParameterizedThreadStart delegate supplied to the thread constructor. Note that the call to Start does not block the calling thread.

## Pausing and Resuming threads

After you have started a thread, you often want to pause that thread for a fixed period of time. Calling Thread.Sleep causes the current thread to immediately block for the number of milliseconds you pass to Sleep, yielding the remainder of its time slice to another thread. One thread cannot call Sleep on another thread. Calling Thread.Sleep(Timeout.Infinite) causes a thread to sleep until it is interrupted by another thread that calls Thread.Interrupt or is aborted by Thread.Abort.

## Thread Safety

When we are working in a multi threaded environment, we need to maintain that no thread leaves the object in an invalid state when it gets suspended. Thread safety basically means the members of an object always maintain a valid state when used concurrently by multiple threads. There are multiple ways of achieving this – The Mutex class or the Monitor classes of the Framework enable this.

## Applying Lock

You can put a lock on a block of code – which means that that block has to be executed at one go and that at any given time, only one thread could be executing that block. The syntax for the lock would be as follows:

```
lock(this) {        Console.WriteLine("Inside the lock now");        }
```

In the above code sample, the code block following the lock statement will be executed as one unit of execution, and only one thread would be able to execute it at any given time. So, once a thread enters that block, no other thread can enter the block till the first thread has exited it.

## Scheduling Threads

Every thread has a thread priority assigned to it. Threads created within the common language runtime are initially assigned the priority of ThreadPriority.Normal. Threads created outside the runtime retain the priority they had before they entered the managed environment. You can get or set the priority of any thread with the Thread.Priority property.

Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system. The details of the scheduling algorithm used to determine the order in which threads are executed varies with each operating system.

## ThreadState (public enum ThreadState)

The ThreadState enumeration defines a set of all possible execution states for threads. Once a thread is created, it's in at least one of the states until it terminates. Threads created within the CLR (common language runtime) are initially in the Unstarted state, while external, or unmanaged, threads that come into the runtime are already in the Running state.

A thread is transitioned from the Unstarted state into the Running state by calling Thread.Start(). Once a thread leaves the Unstarted state as the result of a call to Start(), it can never return to the Unstarted state. A thread can never leave the Stopped state. A Thread can be in following states with corresponding actions written before the state that cause a change of state.

Unstarted-
A thread is created within the common language runtime.

Running-
1- Another thread calls Interrupt.
2- Another thread calls Resume.
3- Another thread calls the Thread.Start method on the new thread, and the call returns.
4- The Start method does not return until the new thread has started running. There is no way to know at what point the new thread will start running, during the call to Start.
WaitSleepJoin-
1- The thread calls Sleep.
2- The thread calls
Monitor. Wait on another object.
3-The thread calls Join on another thread.

AbortRequested-
1- Another thread calls Suspend.
2- Another thread calls Abort.

Suspended-

The thread responds to a  Suspend request.

Stopped-
The thread responds to an Abort request.

## The String and StringBuilder types

Both String and StringBuffer are a sequential collection of characters that is used to represent text and Defined by following classes respectively.

- public sealed class String
- public sealed class StringBuilder

## String

A string is a sequential collection of characters that is used to represent text. A String object is a sequential collection of System.Char objects that represent a string; a System.Char object corresponds to a UTF-16 code unit. The value of the String object is the content of the sequential collection of System.Char objects, and that value is immutable (that is, it is read-only). String is an immutable type. That is, each operation that appears to modify a String object actually creates a new string.

## Instantiate a String object

You can instantiate a String object in the following ways:

1- By assigning a string literal to a String variable. This is the most commonly used method for creating a string.

```
string string1 = "This is a string";
```

2- By calling a String class constructor. The following example instantiates strings by calling several class constructors.

```
char[ ] chars = { 'w', 'o', 'r', 'd' };
string string1 = new string(chars);
```

3- By using the string concatenation operator (+ in C# and & or + in Visual Basic) to create a single string from any combination of String instances and string literals.

```
string string1 = "Today is " + DateTime.Now.ToString("D") + ".";
```

4- By retrieving a property or calling a method that returns a string.

```
string sentence = "Uttarakhand Board.";
string word2 = sentence.Substring (0, 11);
```

## String Builder

Although StringBuilder and String both represent sequences of characters, they are implemented differently. String is an immutable type. That is, each operation that appears to modify a String object actually creates a new string. For routines that perform extensive string manipulation (such as apps that modify a string numerous times in a loop), modifying a string repeatedly can exact a significant performance penalty. The alternative is to use StringBuilder, which is a mutable string class. Mutability means that once an instance of the class has been created, it can be modified by appending, removing, replacing, or inserting characters.

A StringBuilder object maintains a buffer to accommodate expansions to the string. New data is appended to the buffer if room is available; otherwise, a new, larger buffer is allocated, data from the original buffer is copied to the new buffer, and the new data is then appended to the new buffer.

1- Create a StringBuilder that expects to hold 99 characters. Initialize the StringBuilder with "UBTE Roorkee".
```
StringBuilder sb = new StringBuilder("UBTE Roorkee", 100);
```
2- Append four characters "( U K )" to the end of the StringBuilder.
```
sb.Append(new char[ ] { '(', 'U', 'K' , ')' });
```
3- Insert a string at the beginning of the StringBuilder.
```
sb.Insert(0, "Secretary");
```
4- Replace all lowercase e's with uppercase E's.
```
sb.Replace('e', 'E');
```

**Consider using the String class under these conditions:**

Following are the conditions where String class is good option to deal string type data or string literals.

1- When the number of changes that your application will make to a string is small. In these cases, StringBuilder might offer negligible or no performance improvement over String.
2- When you are performing a fixed number of concatenation operations, particularly with string literals. In this case, the compiler might combine the concatenation operations into a single operation.
3- When you have to perform extensive search operations while you are building your string. The StringBuilder class lacks search methods such as IndexOf or StartsWith.
4- You'll have to convert the StringBuilder object to a String for these operations, and this can negate the performance benefit from using StringBuilder.

## Conditions using the StringBuilder

These two conditions where StringBuilder class is good option to deal string type data or string literals.

1- When you expect your app to make an unknown number of changes to a string at design time (for example, when you are using a loop to concatenate a random number of strings that contain user input).
2- When you expect your app to make a significant number of changes to a string.

## How StringBuilder works

The StringBuilder.Length property indicates the number of characters the StringBuilder object currently contains. If you add characters to the StringBuilder object, its length increases until it equals the size of the StringBuilder.Capacity property, which defines the number of characters that the object can contain.

If the number of added characters causes the length of the StringBuilder object to exceed its current capacity, new memory is allocated, the value of the Capacity property is doubled, new characters are added to the StringBuilder object, and its Length property is adjusted. Additional memory for the StringBuilder object is allocated dynamically until it reaches the value defined by the StringBuilder.MaxCapacity property.

When the maximum capacity is reached, no further memory can be allocated for the StringBuilder object, and trying to add characters or expand it beyond its maximum capacity throws either an ArgumentOutOfRangeException or an OutOfMemoryException exception.

## Array

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements. If you want the array to store elements of any type, you can specify object as its type. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from Object.

```
class TestArrays
{
  static void Main()
  {
      // Declare a single-dimensional array.
```

```
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[ ] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[ ] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array.
        int[ , ] multiDimArray1 = new int[2, 3];

        // Declare and set array element values.
        int[ , ] mDimArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array.
        int[ ][ ] jaggedArray = new int[6][ ];

        //Set the 0th and 1st Jagged Array Elements Values.
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
        jaggedArray[1] = new int[3] { 1, 2, 3};

        // Declare and Set Jagged Elements Values.
        int[ ][ ] jaggedArray2 = new int[ ][ ] {
                                   new int[ ] { 1, 3, 5, 7, 9 },
                                   new int[ ] { 0, 2, 4, 6 },
                                   new int[ ] { 11, 22 }
                               };
  }
}
```

**An array has the following properties:**

1- An array can be Single-Dimensional, Multidimensional or Jagged.
2- The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
3- The default values of numeric array elements are set to zero, and reference elements are set to null.
4- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to null.
5- Arrays are zero indexed: array with n elements is indexed from 0 to n-1.
6- Array elements can be of any type, including an array type.

7- Array types are reference types derived from the abstract base type Array.

In C# you can use foreach iteration on all arrays.

```
int [ ] a_array = new int[ ] { 1, 2, 3, 4, 5, 6, 7 };
// foreach loop begin it will run till the last element of the array */
foreach(int items in a_array)
{
  Console.WriteLine(items);
}
```