Classes consists of :

class members that include properties that describe class data, methods that define class behavior, and events that provide communication between different classes and objects

class SampleClass

{

private string \_sampleField;

public int SampleProperty { get; set; }

}

// Properties have get and set procedures, which provide more control on how values are set or returned.

// Fields are like variables because they can be read or set directly.

A struct type is a value type that is typically used to encapsulate small groups of related variables,

public struct Book

{

public decimal price;

public string title;

public string author;

}

Structs can also contain [constructors](https://msdn.microsoft.com/en-us/library/ace5hbzh.aspx), [constants](https://msdn.microsoft.com/en-us/library/ms173119.aspx), [fields](https://msdn.microsoft.com/en-us/library/ms173118.aspx), [methods](https://msdn.microsoft.com/en-us/library/ms173114.aspx), [properties](https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx), [indexers](https://msdn.microsoft.com/en-us/library/6x16t2tx.aspx), [operators](https://msdn.microsoft.com/en-us/library/ms173145.aspx), [events](https://msdn.microsoft.com/en-us/library/awbftdfh.aspx), and [nested types](https://msdn.microsoft.com/en-us/library/ms173120.aspx), although if several such members are required, you should consider making your type a class instead.

* Within a struct declaration, fields cannot be initialized unless they are declared as const or static.
* A struct cannot declare a default constructor (a constructor without parameters) or a destructor.
* Structs are copied on assignment. When a struct is assigned to a new variable, all the data is copied, and any modification to the new copy does not change the data for the original copy. This is important to remember when working with collections of value types such as Dictionary<string, myStruct>.
* Structs are value types and classes are reference types.
* Unlike classes, structs can be instantiated without using a new operator.
* Structs can declare constructors that have parameters.
* A struct cannot inherit from another struct or class, and it cannot be the base of a class. All structs inherit directly from System.ValueType, which inherits from System.Object.
* A struct can implement interfaces.
* A struct can be used as a nullable type and can be assigned a null value.

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 are a special kind of static method, but they are called as if they were instance methods on the extended type.

Class has internal access by default and if you specify it’s default constructor it is private by default otherwise in absence of default constructor it is public access.

A static constructor is used to initialize any [static](https://msdn.microsoft.com/en-us/library/98f28cdx.aspx) data, or to perform a particular action that needs to be performed once only. It is called automatically before the first instance is created or any static members are referenced. If you have both instance and static constructor then static constructor will only fire when ever the static field or static class is referenced regardless of constructor call order which means either static constructor is called before instance or vice versa.

Constructor can be public,private,protected,internal,protected internal and static

Static constructors have the following properties:

* A static constructor does not take access modifiers or have parameters.
* A static constructor is called automatically to initialize the [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx) before the first instance is created or any static members are referenced.
* A static constructor cannot be called directly.
* The user has no control on when the static constructor is executed in the program.
* A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
* Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the LoadLibrarymethod.
* If a static constructor throws an exception, the runtime will not invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain in which your program is running.

Destructors

Destructors are used to destruct instances of classes. In the .NET Framework, the garbage collector automatically manages the allocation and release of memory for the managed objects in your application. However, you may still need destructors to clean up any unmanaged resources that your application creates. There can be only one destructor for a class.

Nested Classes

A class defined within another class is called *nested*. By default, the nested class is private.

C#

class Container

{

class Nested

{

// Add code here.

}

}

To create an instance of the nested class, use the name of the container class followed by the dot and then followed by the name of the nested class:

C#

Container.Nested nestedInstance = new Container.Nested()

protected internalThe type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

To specify that a class cannot be used as a base class:

public sealed class A { }

To specify that a class can be used as a base class only and cannot be instantiated:

public abstract class B { }

The abstract modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes. Members marked as abstract, or included in an abstract class, must be implemented by classes that derive from the abstract class.

Abstract classes have the following features:

* An abstract class cannot be instantiated.
* An abstract class may contain abstract methods and accessors.
* It is not possible to modify an abstract class with the [sealed](https://msdn.microsoft.com/en-us/library/88c54tsw.aspx) modifier because the two modifers have opposite meanings. The sealed modifier prevents a class from being inherited and the abstract modifier requires a class to be inherited.
* A non-abstract 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.
* Abstract method declarations are only permitted in abstract classes.
* Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. For example:
* public abstract void MyMethod();

The implementation is provided by an overriding method [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx), which is a member of a non-abstract class.

* It is an error to use the [static](https://msdn.microsoft.com/en-us/library/98f28cdx.aspx) or [virtual](https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

* It is an error to use the abstract modifier on a static property.
* An abstract inherited property can be overridden in a derived class by including a property declaration that uses the [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) modifier.

Static(Shared in VB) class or Members are shared by all instance of a class

The virtual keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

public virtual double Area()

{

return x \* y;

}

The implementation of a virtual member can be changed by an [overriding member](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) in a derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

By default, methods are non-virtual. You cannot override a non-virtual method.

You cannot use the virtual modifier with the static, abstract, private, or override modifiers.

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 overridedeclaration 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.

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](https://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx).

You cannot use the new, static, or virtual modifiers to modify an override method.

An overriding property declaration must specify exactly the same access modifier, type, and name as the inherited property, and the overridden property must be virtual, abstract, or override.

When used as a declaration modifier, the new keyword explicitly hides a member that is inherited from a base class. When you hide an inherited member, the derived version of the member replaces the base class version. Although you can hide members without using the new modifier, you get a compiler warning. If you use new to explicitly hide a member, it suppresses this warning.

To hide an inherited member, declare it in the derived class by using the same member name, and modify it with the new keyword.

public class BaseC

{

public int x;

public void Invoke() { }

}

public class DerivedC : BaseC

{

new public void Invoke() { }

}

An interface contains definitions for a group of related functionalities that a [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx) or a [struct](https://msdn.microsoft.com/en-us/library/ah19swz4.aspx) can implement.

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.

An interface has the following properties:

* An interface is like an abstract base class. Any class or struct that implements the interface must implement all its members.
* An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
* Interfaces can contain events, indexers, methods, and properties.
* Interfaces contain no implementation of methods.
* A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

## **Delegates**

A delegate is a type that defines a method signature, and can provide a reference to any method with a compatible signature. You can invoke (or call) the method through the delegate. Delegates are used to pass methods as arguments to other methods.

Indexers are a syntactic convenience that enable you to create a [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx), [struct](https://msdn.microsoft.com/en-us/library/ah19swz4.aspx), or [interface](https://msdn.microsoft.com/en-us/library/87d83y5b.aspx) that client applications can access just as an array. Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array.

Generics

Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code.

## **Generics Overview**

* Use generic types to maximize code reuse, type safety, and performance.
* The most common use of generics is to create collection classes.
* The .NET Framework class library contains several new generic collection classes in the [System.Collections.Generic](https://msdn.microsoft.com/en-us/library/system.collections.generic.aspx) namespace. These should be used whenever possible instead of classes such as [ArrayList](https://msdn.microsoft.com/en-us/library/system.collections.arraylist.aspx) in the [System.Collections](https://msdn.microsoft.com/en-us/library/system.collections.aspx) namespace.
* You can create your own generic interfaces, classes, methods, events and delegates.
* Generic classes may be constrained to enable access to methods on particular data types.
* Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

The event keyword is used to declare an event in a publisher class.

Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

### **Uses for Serialization**

Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

### **Binary Serialization**

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams.

### **XML Serialization**

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. [System.Xml.Serialization](https://msdn.microsoft.com/en-us/library/system.xml.serialization.aspx) contains the classes necessary for serializing and deserializing XML.

### **Basic Serialization**

The only requirement in basic serialization is that the object has the [SerializableAttribute](https://msdn.microsoft.com/en-us/library/system.serializableattribute.aspx) attribute applied. The [NonSerializedAttribute](https://msdn.microsoft.com/en-us/library/system.nonserializedattribute.aspx) can be used to keep specific fields from being serialized.

### **Custom Serialization**

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked [SerializableAttribute](https://msdn.microsoft.com/en-us/library/system.serializableattribute.aspx) and implement the [ISerializable](https://msdn.microsoft.com/en-us/library/system.runtime.serialization.iserializable.aspx) interface.

**Reflection** provides objects (of type [Type](https://msdn.microsoft.com/en-us/library/system.type.aspx)) that describe assemblies, modules and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.

Reflection is useful in the following situations:

* When you have to access attributes in your program's metadata. For more information, see [Retrieving Information Stored in Attributes](https://msdn.microsoft.com/en-us/library/71s1zwct.aspx).
* For examining and instantiating types in an assembly.
* For building new types at runtime. Use classes in [System.Reflection.Emit](https://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx).
* For performing late binding, accessing methods on types created at run time.

Typeof(string).IsClass – true

Typeof(int).IsValueType- true

Typeof(abc).IsGenricType – true

Topics pending

Tuples,LINQ, Multithreading,Expression Tree,Generics

C# Latest Features

## Throw expressions

It is easy to throw an exception in the middle of an expression: just call a method that does it for you! But in C# 7.0 we are directly allowing throw as an expression in certain places:

class Person

{

public string Name { get; }

public Person(string name) => Name = name ?? throw new ArgumentNullException(name);

public string GetFirstName()

{

var parts = Name.Split(" ");

return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No name!");

}

public string GetLastName() => throw new NotImplementedException();

}

## Literal improvements

C# 7.0 allows \_ to occur as a **digit separator** inside number literals:

var d = 123\_456;

var x = 0xAB\_CD\_EF;

* 1. Out Variables

 the ability to declare a variable right at the point where it is passed as an out argument:

public void PrintCoordinates(Point p)

{

p.GetCoordinates(out int x, out int y);

WriteLine($"({x}, {y})");

}

### 4. Switch statements with patterns

We’re generalizing the switch statement so that:

* You can switch on any type (not just primitive types)
* Patterns can be used in case clauses
* Case clauses can have additional conditions on them

switch(shape)

{

case Circle c:

WriteLine($"circle with radius {c.Radius}");

break;

case Rectangle s when (s.Length == s.Height):

WriteLine($"{s.Length} x {s.Height} square");

break;

case Rectangle r:

WriteLine($"{r.Length} x {r.Height} rectangle");

break;

default:

WriteLine("<unknown shape>");

break;

case null:

throw new ArgumentNullException(nameof(shape));

}