|  |
| --- |
| **[Summary]** |
| [OOP]  A programming model where programs are organized around objects and data rather than action and logic. |
| [OOP Features]  **Class -** Collection of objects. A class is a blueprint of an object that contains variables for storing data and functions to perform operations on the data. (Logical representation of Data)  **Object -** Object is an instance of a class. Objects are run-time entities of object oriented systems. It is a bundle of related variables and methods.  When object created using new operator, memory is allocated for the class in heap. It an instance and its starting address will be stored in stack memory.  **Access Modifiers & Access Levels –**   |  |  | | --- | --- | | **C# Modifier** | **Definition** | | [**public**](https://msdn.microsoft.com/en-us/library/yzh058ae.aspx) | The type or member can be accessed by any other code in the **same assembly or another assembly** that references it. | | [**private**](https://msdn.microsoft.com/en-us/library/st6sy9xe.aspx) | The type or member can only be accessed by code in the **same class**. | | [**protected**](https://msdn.microsoft.com/en-us/library/bcd5672a.aspx) | The type or member can only be accessed by code in **the same class or in a derived class**. | | [**internal**](https://msdn.microsoft.com/en-us/library/7c5ka91b.aspx) | The type or member can be accessed by any code in the **same assembly**, but not from another assembly. | | **protected internal** | The type or member can be accessed by any code in the **same assembly**, or by **any derived class** in another assembly. |   **Class Members -**  Properties & Fields -  Fields and properties represent information that an object contains. Fields are like variables because they can be read or set directly.  Properties have get and set procedures, which provide more control on how values are set or returned.  Methods -  A method is an action that an object can perform.  Constructors -  Constructors are class methods that are executed automatically when an object of a given type is created.  A constructor can run only once when a class is created.  Destructors -  Destructors are used to destruct instances of classes.  Destructors cannot be called. They are invoked automatically. A destructor does not take modifiers or have parameters.  A class can only have one destructor.  Events -  Events enable a class or object to notify other classes or objects when something of interest occurs.  The class that sends (or raises) the event is called the publisher and the classes that receive (or handle) the event are called subscribers.  To declare an event in a class, use the event keyword.  Nested Classes -  A class defined within another class is called nested. By default, the nested class is private.  **Anonymous Type -**  Anonymous types enable you to create objects without writing a class definition for the data type. Instead, the compiler generates a class for you. |
| [OOP Concepts]  **Abstraction -**  Abstraction is a process of **hiding the implementation** details and displaying the essential features.  Abstractions are usually implemented as **abstract classes** or **interfaces**.  Abstractions are at the core of many architectural patterns, such as plug-ins, inversion of control (IoC), pipelines, and so on.  **Encapsulation -**  **Wrapping up a data member and method** together into a single unit (in other words class) is called Encapsulation.  Encapsulation is enclosing the related operations and data related to an object into that object.  In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private and public getter and setter methods are provided to manipulate these attributes. Thus, encapsulation makes the **concept of data hiding** possible.  **Inheritance -**  Inheritance enables you to create a new class that reuses, extends, and modifies the behavior that is defined in another class.  The class whose members are inherited is called the **base class**, and the class that inherits those members is called the **derived class**.  Note: C# doesn't support multiple inheritance, i.e. you can specify only one base class for a derived class. We can **implement multiple inheritance** in C# using interface.  By default all classes can be inherited. However, you can specify whether a class must not be used as a base class. (By using **sealed keyword**)  Overriding Members -  To change the behavior of the inherited member, you need to override it. That is, you can define a new implementation of the method, property or event in the derived class.   |  |  | | --- | --- | | **C# Modifier** | **Definition** | | [virtual](https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) | Allows a class member to be overridden in a derived class. | | [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) | Overrides a virtual (overridable) member defined in the base class. | | [abstract](https://msdn.microsoft.com/en-us/library/sf985hc5.aspx) | Requires that a class member to be overridden in the derived class. | | [new Modifier](https://msdn.microsoft.com/en-us/library/435f1dw2.aspx) | Hides a member inherited from a base class |   Interfaces -  Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation.  They are implemented by classes, and defined as separate entities from classes. An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.  **Polymorphism -**  Polymorphism means one name, many forms. One function behaves in different forms.  It has the ability for classes to provide different implementations of methods that are called through the same name.  Compile time polymorphism -  Compile time polymorphism is method and operators overloading. It is also called early binding.  In method overloading method performs the different task at the different input parameters.  Runtime polymorphism -  Runtime polymorphism is done using inheritance and virtual functions. Method overriding is called runtime polymorphism. It is also called late binding.  When **overriding** a method, you change the behavior of the method for the derived class. **Overloading** a method simply involves having another method with the same prototype.  Method overloading has nothing to do with inheritance or virtual methods. |
| [Other]  **Generics -**  Classes, structures, interfaces and methods in the .NET Framework can include type parameters that define types of objects that they can store or use.  The most common example of generics is a collection, where you can specify the type of objects to be stored in a collection.  **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.  Note: Event handlers are nothing more than methods that are invoked through delegates. |

|  |
| --- |
| [Object Oriented Programming] |

# **Brief:**

A programming model where programs are organized around objects and data rather than action and logic.

It is about create objects and then building data and functions around these objects.

# OOP Features

Main features covered under Object Oriented Programming.

## Class

Collection of objects.

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

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

Class does not occupy any memory hence it is only a **logical representation of Data**.

To define class:

class SampleClass

{

}

## Object

Object is an **instance of a class**.

Objects are **run-time entities** of object oriented systems, they may represent a person, a place or any item that the program must handle.

It is a **bundle of related variables and methods**.

A class will not occupy any memory hence to work with data represented by class, you must have to create variable of class which is called an object.

When object created using new operator, **memory is allocated for the class in heap**. It an instance and its starting address will be stored in stack memory. (If not created with new operator, it will not allocate memory in heap and its value will be **null**.)

To define object:

class SampleClass

{

}

SampleClass sampleObject = new SampleClass();

## Access Modifiers and Access Levels

All classes and class members can specify what **access level** they provide to other classes by using ***access modifiers***.

|  |  |
| --- | --- |
| **C# Modifier** | **Definition** |
| [**public**](https://msdn.microsoft.com/en-us/library/yzh058ae.aspx) | The type or member can be accessed by any other code in the **same assembly or another assembly** that references it. |
| [**private**](https://msdn.microsoft.com/en-us/library/st6sy9xe.aspx) | The type or member can only be accessed by code in the **same class**. |
| [**protected**](https://msdn.microsoft.com/en-us/library/bcd5672a.aspx) | The type or member can only be accessed by code in **the same class or in a derived class**. |
| [**internal**](https://msdn.microsoft.com/en-us/library/7c5ka91b.aspx) | The type or member can be accessed by any code in the **same assembly**, but not from another assembly. |
| **protected internal** | The type or member can be accessed by any code in the **same assembly**, or by **any derived class** in another assembly. |

The following access modifiers are available:

## Class Members

## **Properties and Fields**

Fields and properties represent information that an object contains. Fields are like variables because they can be read or set directly.

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

To define a field:

class SampleClass

{

public string sampleField;

}

C# allows you either to create a private field for storing the property value or use so-called **auto-implemented properties** that create this field automatically behind the scenes and provide the basic logic for the property procedures.

To define an auto-implemented property:

class SampleClass

{

public int SampleProperty { get; set; }

}

## **Methods**

A method is an action that an object can perform.

To define a method of a class:

class SampleClass

{

public int sampleMethod(string sampleParam)

{

// Insert code here

}

}

A class can have several **implementations**, or **overloads**, of the same method that differ in the number of parameters or parameter types.

To overload a method:

public int sampleMethod(string sampleParam) {};

public int sampleMethod(int sampleParam) {}

## **Constructors**

Constructors are class methods that are **executed automatically** when an object of a given type is created.

Constructors usually initialize the data members of the new object. A constructor can **run only once** when a class is created.

To define a constructor for a class:

public class SampleClass

{

public SampleClass()

{

// Add code here

}

}

## **Destructors**

Destructors are used to **destruct instances** of classes.

Destructors cannot be called. They are **invoked automatically**. A destructor does not take modifiers or have parameters.

A class can only have **one destructor**.

(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.)

To define a destructor for a class:

class SampleClass

{

~ SampleClass() // destructor

{

// cleanup statements...

}

}

## **Events**

Events enable a class or object to **notify other classes or objects** when something of interest occurs.

The class that sends (or raises) the event is called the ***publisher*** and the classes that receive (or handle) the event are called ***subscribers***.

To declare an event in a class, use the **event** keyword.

To raise an event, invoke the **event delegate**. To subscribe to an event, use the **+=** operator; to unsubscribe from an event, use the **-=** operator.

The following **keywords** apply to **events**.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| [**static**](https://msdn.microsoft.com/en-us/library/98f28cdx.aspx) | Makes the event available to callers at any time, even if no instance of the class exists. |
| [**virtual**](https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) | Allows derived classes to override the event behavior by using the [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) keyword. |
| [**sealed**](https://msdn.microsoft.com/en-us/library/88c54tsw.aspx) | Specifies that for derived classes it is no longer virtual. |
| [**abstract**](https://msdn.microsoft.com/en-us/library/sf985hc5.aspx) | The compiler will not generate **add** and **remove** event accessor blocks and therefore derived classes must provide their own implementation. |

To declare and raise an event that uses EventHandler:

public class SampleEventArgs

{

public SampleEventArgs(string s) { Text = s; }

public String Text {get; private set;} // readonly

}

public class Publisher

{

// Declare the delegate (if using non-generic pattern).

public delegate void SampleEventHandler(object sender, SampleEventArgs e);

// Declare the event.

public event SampleEventHandler SampleEvent;

// Wrap the event in a protected virtual method

// to enable derived classes to raise the event.

protected virtual void RaiseSampleEvent()

{

// Raise the event by using the () operator.

if (SampleEvent != null)

SampleEvent(this, new SampleEventArgs("Hello"));

}

}

## **Nested Classes**

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

class SampleClass

{

class NestedClass

{

// Add code here.

}

}

// to create instance of nested class

SampleClass.NestedClass nestedInstance = new SampleClass.NestedClass();

## Anonymous Type

Anonymous types enable you to **create objects without writing a class definition** for the data type. Instead, the **compiler generates a class** for you.

The class has no usable name and contains the properties you specify in declaring the object.

To create an instance of an anonymous type:

// sampleObject is an instance of a simple anonymous type.

var sampleObject =

new { FirstProperty = "A", SecondProperty = "B" };

# OOP Concepts

All programming languages supporting Object Oriented Programming will be supporting these main concepts:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

## Abstraction

Abstraction is a process of **hiding the implementation** details and displaying the essential features.

An abstraction is a type that describes a contract but does not provide a full implementation of the contract.

Abstractions are usually implemented as **abstract classes** or **interfaces**, and they come with a well-defined set of reference documentation describing the **required semantics of the types** implementing the contract.

Abstractions are at the core of many architectural patterns, such as plug-ins, inversion of control (IoC), pipelines, and so on.

**Real example of Abstraction:**

Suppose you have 3 mobile phones as in the following:    
*Nokia 1400 (Features: Calling, SMS)   
Nokia 2700 (Features: Calling, SMS, FM Radio, MP3, Camera)   
Black Berry (Features: Calling, SMS, FM Radio, MP3, Camera, Video Recording, Reading E-mails)*

Abstract information (necessary and common information) for the object "Mobile Phone" is that it makes a call to any number and can send SMS.

So that, for a mobile phone object you will have the abstract class as in the following:

 abstract class MobilePhone

 {

     public void Calling();

     public void SendSMS();

 }

 public class Nokia1400 : MobilePhone

 {

 }

 public class Nokia2700 : MobilePhone

 {

     public void FMRadio();

     public void MP3();

     public void Camera();

 }

 public class BlackBerry : MobilePhone

 {

     public void FMRadio();

     public void MP3();

     public void Camera();

     public void Recording();

     public void ReadAndSendEmails();

}

## Encapsulation

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

Encapsulation is enclosing the related operations and data related to an object into that object.

In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private and public getter and setter methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible.

Example of encapsulation:

class Program {

public class Account {

private decimal accountBalance = 500.00;

public decimal CheckBalance() {

return accountBalance;

}

}

static void Main() {

Account myAccount = new Account();

decimal myBalance = myAccount.CheckBalance();

/\* This Main method can check the balance via the public

\* "CheckBalance" method provided by the "Account" class

\* but it cannot manipulate the value of "accountBalance" \*/

}

}

## Inheritance

Inheritance enables you to create a new class that reuses, extends, and modifies the behavior that is defined in another class.

The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. However, all classes in C# implicitly inherit from the [Object](https://msdn.microsoft.com/en-us/library/system.object.aspx) class that supports .NET class hierarchy and provides low-level services to all classes.

Note: C# doesn't support multiple inheritance, i.e. you can specify only one base class for a derived class. We can implement multiple inheritance in C# using interface. Classes can inherit from multiple interfaces at the same time.

By default all classes can be inherited. However, you can specify whether a class must not be used as a base class, or create a class that can be used as a base class only.

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

public sealed class A { }

You can also use the **sealed** modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties. More about [sealed keyword](https://msdn.microsoft.com/en-us/library/88c54tsw.aspx).

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. Members marked as abstract, or included in an abstract class, must be implemented by classes that derive from the abstract class. More about [abstract keyword](https://msdn.microsoft.com/en-us/library/sf985hc5.aspx).

A simple example to understand inheritance in C#.

Using System;

Public class BaseClass

{

    Public BaseClass ()

    {

        Console.WriteLine ("Base Class Constructor executed");

    }

    Public void Write ()

    {

        Console.WriteLine ("Write method in Base Class executed");

    }

}

Public class ChildClass: BaseClass

{

    Public ChildClass ()

    {

        Console.WriteLine("Child Class Constructor executed");

    }

    Public static void Main ()

    {

        ChildClass CC = new ChildClass ();

        CC.Write ();

    }

}

In the Main () method in ChildClass we create an instance of childclass. Then we call the write () method. If you observe the ChildClass does not have a write() method in it. This write () method has been inherited from the parent BaseClass.

The output of the above program is   
**Output:**

*Base Class Constructor executed  
Child Class Constructor executed  
Write method in Base Class executed*

## **Overriding Members**

By default, a derived class inherits all members from its base class. If you want to change the behavior of the inherited member, you need to override it. That is, you can define a new implementation of the method, property or event in the derived class.

The following modifiers are used to control how properties and methods are overridden:

|  |  |
| --- | --- |
| **C# Modifier** | **Definition** |
| [virtual](https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) | Allows a class member to be overridden in a derived class. |
| [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) | Overrides a virtual (overridable) member defined in the base class. |
| [abstract](https://msdn.microsoft.com/en-us/library/sf985hc5.aspx) | Requires that a class member to be overridden in the derived class. |
| [new Modifier](https://msdn.microsoft.com/en-us/library/435f1dw2.aspx) | Hides a member inherited from a base class |

## **Interface**

Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation.

They are implemented by classes, and defined as separate entities from classes. An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.

To define an interface & implement interface in class:

interface ISampleInterface

{

void SampleMethod();

}

class ImplementationClass : ISampleInterface

{

// Explicit interface member implementation:

void ISampleInterface.SampleMethod()

{

// Method implementation.

}

static void Main()

{

// Declare an interface instance.

ISampleInterface obj = new ImplementationClass();

// Call the member.

obj.SampleMethod();

}

}

## Polymorphism

Polymorphism means one name, many forms. One function behaves in different forms.

It has the ability for classes to provide different implementations of methods that are called through the same name.

**Polymorphism is of two types:**

1. Compile time polymorphism/Overloading
2. Runtime polymorphism/Overriding

**Compile Time Polymorphism**

Compile time polymorphism is method and operators overloading. It is also called early binding.

In method overloading method performs the different task at the different input parameters.

**Runtime Polymorphism**

Runtime polymorphism is done using inheritance and virtual functions. Method overriding is called runtime polymorphism. It is also called late binding.

When **overriding** a method, you change the behavior of the method for the derived class. **Overloading** a method simply involves having another method with the same prototype.

Method overloading has nothing to do with inheritance or virtual methods.

**Example of polymorphism with both scenario:**

***Description:*** *First, create a base class called Shape, and derived classes such as Rectangle, Circle, and Triangle. Give the Shape class a virtual method called Draw, and override it in each derived class to draw the particular shape that the class represents. Create a List<Shape> object and add a Circle, Triangle and Rectangle to it. To update the drawing surface, use a [foreach](https://msdn.microsoft.com/en-us/library/ttw7t8t6.aspx) loop to iterate through the list and call the Draw method on each Shape object in the list. Even though each object in the list has a declared type of Shape, it is the run-time type (the overridden version of the method in each derived class) that will be invoked.*

public class Shape

{

// A few example members

public int X { get; private set; }

public int Y { get; private set; }

public int Height { get; set; }

public int Width { get; set; }

// Virtual method

public virtual void Draw()

{

Console.WriteLine("Performing base class drawing tasks");

}

}

class Circle : Shape

{

public override void Draw()

{

// Code to draw a circle...

Console.WriteLine("Drawing a circle");

base.Draw();

}

}

class Rectangle : Shape

{

public override void Draw()

{

// Code to draw a rectangle...

Console.WriteLine("Drawing a rectangle");

base.Draw();

}

}

class Triangle : Shape

{

public override void Draw()

{

// Code to draw a triangle...

Console.WriteLine("Drawing a triangle");

base.Draw();

}

}

class Program

{

static void Main(string[] args)

{

// Polymorphism at work #1: a Rectangle, Triangle and Circle

// can all be used whereever a Shape is expected. No cast is

// required because an implicit conversion exists from a derived

// class to its base class.

System.Collections.Generic.List<Shape> shapes = new System.Collections.Generic.List<Shape>();

shapes.Add(new Rectangle());

shapes.Add(new Triangle());

shapes.Add(new Circle());

// Polymorphism at work #2: the virtual method Draw is

// invoked on each of the derived classes, not the base class.

foreach (Shape s in shapes)

{

s.Draw();

}

// Keep the console open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

/\* Output:

Drawing a rectangle

Performing base class drawing tasks

Drawing a triangle

Performing base class drawing tasks

Drawing a circle

Performing base class drawing tasks \*/

In C#, every type is polymorphic because all types, including user-defined types, inherit from [Object](https://msdn.microsoft.com/en-us/library/system.object.aspx).

# Other

## Generics

Classes, structures, interfaces and methods in the .NET Framework can include *type parameters* that define types of objects that they can store or use.

The most common example of generics is a collection, where you can specify the type of objects to be stored in a collection.

To define a generic class:

Public class SampleGeneric<T>

{

public T Field;

}

To create an instance of a generic class:

SampleGeneric<string> sampleObject = new SampleGeneric<string>();

sampleObject.Field = "Sample string";

For more information about [generics](https://msdn.microsoft.com/en-us/library/ms172192.aspx).

## 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.

**Note:** Event handlers are nothing more than methods that are invoked through delegates.

To create a delegate:

public delegate void SampleDelegate(string str);

To create a reference to a method that matches the signature specified by the delegate:

class SampleClass

{

// Method that matches the SampleDelegate signature.

public static void sampleMethod(string message)

{

// Add code here.

}

// Method that instantiates the delegate.

void SampleDelegate()

{

SampleDelegate sd = sampleMethod;

sd("Sample string");

}

}

For more information about [delegate](https://msdn.microsoft.com/en-us/library/900fyy8e.aspx).