[AspnetO](http://www.aspneto.com/)

We code, that works!

* [Home](http://www.aspneto.com/)

* [Asp.net](http://www.aspneto.com/category/aspnet/)

* [MVC](http://www.aspneto.com/category/mvc/)

* [Interview Questions](http://www.aspneto.com/category/interview-questions/)

You are here: [Home](http://www.aspneto.com/) / [Asp.net](http://www.aspneto.com/category/aspnet/) / Top 10 OOPS Concepts In C# .NET With Examples

Top 10 OOPS Concepts In C# .NET With Examples

[](http://www.aspneto.com/author/mayank/)By: [Mayank Modi](http://www.aspneto.com/author/mayank/) | Folls In: [Asp.net](http://www.aspneto.com/category/aspnet/), [Interview Questions](http://www.aspneto.com/category/interview-questions/), [OOP](http://www.aspneto.com/category/oop/) | Last Updated: Aug 26, 2016

Introduction to OOPS Concepts

OOP stands for Object Oriented Programming. It is a programming methodology that uses Objects to build a system or web applications using programming languages like C#, Vb.net etc.

Here, Objects plays a very important role because it hides the implementation details and exposed only the needed functionalities and related stuff that is required to adopt it. We can access class properties and methods by creating class object that I’ll explain below in this tutorial.

OOPS Concepts, Features & Fundamentals

OOPS contains list of elements that are very helpful to make object oriented programming stronger. Here is the list of top 10 OOPS concepts that we can implement in all major programming languages like c#, vb.net etc.

1] Class:

A class is a collection of objects and represents description of objects that share same attributes and actions. It contains characteristics of the objects such as objects attributes, actions or behaviors.

Here is the syntax and declaration example of Class:

public class Bike  
{  
       //your code goes here..  
}

**Tip:** You may also like [100+ Frequently Asked Interview Questions on .Net, SQL and OOPS Concepts](http://www.aspneto.com/100-frequently-asked-interview-questions-on-asp-net-sql-server-oop-concepts.html" \o "100+ Frequently Asked Interview Questions on Asp.net, SQL Server and OOP Concepts" \t "_blank) and many more tutorials on [Interview Questions](http://www.aspneto.com/category/interview-questions" \o "Tutorials on Interview Questions" \t "_blank) here.

2] Method:

Method is an object’s behavior.

For example, if you consider “Bike” as a class then its behavior is to get bike color, engine, mileage etc. Here is the example of Method:

public class Bike  
{  
       //here is some properties of class Bike  
       public string color;  
       public string engine;  
       public int mileage;

       //here is some behavior/methods of class Bike  
       public string GetColor()  
       {  
               return "red";  
       }  
       public int GetMileage()  
       {  
               return 65;  
       }  
}

In above example *GetColor()* and *GetMileage()* are the methods considered as a object’s behavior.

3] Object:

An object is a real-world entity that keeps together property states and behaviors.

For example, A “Bike” usually has common elements such as bike color, engine, mileage etc. In OOP terminology these would be called as a *Class Properties or Attributes* of a Bike object. Here is the example of Object:

public class Bike  
{  
       //This is the class that contains all properties and behavior of an object

       //here is some properties of class Bike  
       public string color;  
       public string engine;  
       public int mileage;

       //here is some behavior of class Bike  
       public string GetColor()  
       {  
               return "red";  
       }  
       public int GetMileage()  
       {  
               return 65;  
       }  
}

Now we have a complete set of class with its properties and methods. So the question raise in mind is how we can access the class with its object? Right?

It is simple to create its object and access *Bike* class. You can create object of class via single line of code as shown below.

//It also considered as an "Instance of a Bike Class"  
Bike objBike = new Bike();

//Accessing Bike class methods  
objBike.GetColor();  
objBike.GetMileage();

**Tip:** You may also like [Difference Between ExecuteReader, ExecuteNonQuery and ExecuteScalar](http://www.aspneto.com/ado-net-main-difference-between-executereader-executenonquery-executescalar.html" \o "Main Difference Between ExecuteReader, ExecuteNonQuery, ExecuteScalar" \t "_blank) and more tutorials on [OOPS Concepts](http://www.aspneto.com/category/oop" \o "Tutorials on OOPS Concepts" \t "_blank), [Asp.net](http://www.aspneto.com/category/aspnet" \o "Tutorials on Asp.net" \t "_blank) here.

4] Encapsulation:

Encapsulation is the process of keeping or enclosing one or more items within a single physical or logical package. In object oriented programming methodology it prevents access to implementation details.

Encapsulation is implemented by using access specifiers. An access specifier defines the scope and visibility of a class member. Available access specifiers are public, private, protected, internal etc.

**How we can achieve Encapsulation?**  
We can achieve Encapsulation by using *private* access modifier as shown in below example method.

private string GetEngineMakeFormula()  
{  
       private string formula = "a\*b";  
       return formula;  
}

**Example – [Encapsulation]**

public class Bike  
{  
       public int mileage = 65;  
       public string color = "Black";  
       private string formula = "a\*b";

       //Its public – so accessible outside class  
       public int GetMileage()  
       {  
               return mileage;  
       }

       //Its public – so accessible outside class  
       public string GetColor()  
       {  
               return color;  
       }

       //Its private – so not accessible outside class  
       private string GetEngineMakeFormula()  
       {  
               return formula;  
       }  
}

public class Program  
{  
       public static void Main(string[] args)  
       {  
               Bike objBike = new Bike();  
               Console.WriteLine("Bike mileage is : " + objBike.GetMileage()); //accessible outside "Bike"  
               Console.WriteLine("Bike color is : " + objBike.GetColor()); //accessible outside "Bike"  
               //we can't call this method as it is inaccessible outside "Bike"  
               //objBike.GetEngineMakeFormula(); //commented because we can't access it  
               Console.Read();  
       }  
}

So as you can see from above code that we hide *GetEngineMakeFormula()* method by using *private* access modifier because there is no need to give the make formula to users. So exposed only necessary methods for users to use it using *public* access modifier.

5] Abstraction:

Abstraction is the process of providing only essential information to the outside real world and hiding overall background details to present an object. It relies on the separation of interface and implementation.

For example, we continue with “Bike” as an example, we have no access to the piston directly, we can use *start button* to run the piston. Just imagine if a bike manufacturer allows direct access to piston, it would be very difficult to control actions on the piston. That’s the reason why a bike provider separates its internal implementation from its external interface.

**Example – [Abstraction]**

public class Bike  
{  
       public int mileage = 65;  
       public string color = "Black";  
       private string formula = "a\*b";

       //Its public – so accessible outside class  
       public int GetMileage()  
       {  
               return mileage;  
       }

       //Its public – so accessible outside class  
       public string GetColor()  
       {  
               return color;  
       }

       //Its private – so not accessible outside class  
       private string GetEngineMakeFormula()  
       {  
               return formula;  
       }

       //Its public – so accessible outside class  
       public string DisplayMakeFormula()  
       {  
               //"GetEngineMakeFormula()" is private but accessible and limited to this class only  
               return GetEngineMakeFormula();  
       }  
}

public class Program  
{  
       public static void Main(string[] args)  
       {  
               Bike objBike = new Bike();  
               Console.WriteLine("Bike mileage is : " + objBike.GetMileage()); //accessible outside "Bike"  
               Console.WriteLine("Bike color is : " + objBike.GetColor()); //accessible outside "Bike"  
               //we can't call this method as it is inaccessible outside "Bike"  
               //objBike.GetEngineMakeFormula(); //commented because we can't access it  
               Console.WriteLine("Bike color is : " + objBike.DisplayMakeFormula()); //accessible outside  
               Console.Read();  
       }  
}

As you can see from the above code that necessary methods and properties exposed using *public* access modifier and unnecessary methods and properties are hidden using *private* access modifier. This way we can implement abstraction or we can achieve abstraction in our code or web application.

**Note:** Abstraction and Encapsulation are related features in object oriented programming. Abstraction allows making relevant information visible and Encapsulation enables a programmer to implement the desired level of abstraction. That means hidden part of class acts like Encapsulation and exposed part of class acts like Abstraction.

6] Information Hiding:

Information Hiding concept restricts direct exposure of the data. Data is accessed indirectly using safe mechanism, methods in case of programming object. Follow the example given in Abstraction.

7] Inheritance:

Inheritance in OOP allows us to create a new class using an existing one meaning extending one class to another.

This concept can also be related with the real world example. Let’s take a Bike example again. A Bike manufacturer uses same mechanism of existing version of the bike while launching a new version with some additional added functionality. This allows manufacturer to save their time and efforts both.

The main advantage of extending classes is that it provides a convenient way to reuse existing fully tested code in different context thereby saving lots of time with existing coding and its model style.

**Example – [Inheritance]**

public class Base  
{  
       public Base()  
       {  
               Console.WriteLine("Constructor of Base Class");  
       }

       public void DisplayMessage()  
       {  
               Console.WriteLine("Hello, how are you?");  
       }  
}

public class Child : Base  
{  
       public Child()  
       {  
               Console.WriteLine("Constructor of Child class");  
       }  
}

public class Program  
{  
       public static void Main(string[] args)  
       {  
               Child objChild = new Child();  
               //Child class don't have DisplayMessage() method but we inherited from "Base" class  
               objChild.DisplayMessage();  
               Console.Read();  
       }  
}

As you can see in the previous example code, We created an object of a Child class in *Main()* method and then called *DisplayMessage()* method of Base class. If you notice that the Child class doesn’t have *DisplayMessage()* method in it. So obviously it is inherited from the Base class. When you execute following code, result would be as show below:  
**Example Result**

Constructor of Base Class  
Constructor of Child class  
Hello, how are you?

As per sample result, we can say that the “Base” class constructor will automatically be called before the “Child” class constructor.

Thus, here conclusion is that “Base/Parent” classes are automatically instantiated before “Child/Derived” classes.

8] Polymorphism:

The word *Polymorphism* means having many forms. Generally, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

Let’s take Bike example, A Bike can be into two forms like *cell start* or *kick start*. We can later on decide which form or method we will use to start bike to go for drive (meaning at *runtime*).

There are two types of Polymorphism:

* **Compile time polymorphism:** In this type of polymorphism, compiler identifies which polymorphism form it has to take and execute at compile time is called as *compile time polymorphism* or *early binding*. Examples of early binding are *Method Overloading* and *Operator Overloading*. The *Method Overloading* means more than one method having same name but different signatures (or parameters) in the same or different class.

– Advantage: Execution will be fast because everything about the method is known to compiler during compilation.  
– Disadvantage: It has lack of flexibility.

**Example – [Method Overloading]**

public class Base  
{  
       //1st: same method name, return type (object) but different parameter type (object)  
       public object Display(object a)  
       {  
               return (a);  
       }  
                 
       //2nd: same method name, return type (int) but different parameter type (int)  
       public int Display(int a)  
       {  
               return (a);  
       }  
}

public class Program  
{  
       public static void Main(string[] args)  
       {  
               Base objBase = new Base();  
               int val = 7;

               //here 2nd method will be called with type "int"  
               Console.WriteLine(objBase.Display(val));  
               Console.Read();  
       }  
}

In above example, when you run the program, *Display(int a)* method will be called first because *val* is of type *int* at compile time. The assigned *val* is only refer to as a *int* at execution time.

**Example Result**

7

**Note:** While overloading methods, a rule to follow is the overloaded methods must differ either in number of arguments they take or the data type of at least one argument. We can also consider *Method Overriding* as a compile time polymorphism that is called directly by using derived objects.

* **Runtime polymorphism:** In this type of polymorphism, compiler identifies which polymorphism form it has to take and execute at runtime but not at compile time is called as *runtime polymorphism* or *late binding*. Example of early binding is *Method Overriding*. The *Method Overriding* means having two methods with same name and same signature, one method in base class and other method in derived class. It must require changing the behavior of the base class methods in derived class to use its functionality differently.

– Advantage: It has flexibility to adjust object types at runtime.  
– Disadvantage: Execution will be slow as compiler has to get the information about the method to execute at runtime.

We need to use either virtual methods or abstract method to allow the derived class to override a method of the base class.

**Example – [Method Overriding by using virtual method]**

public class Base  
{  
       public virtual string BlogName()  
       {  
               return "AspnetO";  
       }  
}

public class Child : Base  
{  
       //same method name with same signature/parameters  
       public override string BlogName()  
       {  
               return "AspnetO – Quick Way To Learn Asp.net";  
       }  
}

public class Program  
{  
       public static void Main(string[] args)  
       {  
               Base objBase = new Child();  
               Console.WriteLine(objBase.BlogName());  
               Console.Read();  
       }  
}

In above example, when you run the program, at compile time the type of *objBase* is *Base* but it will still call the child class’s override method because at the *runtime*, the type of the *objBase* object refers to is *Child*.

**Example Result**

AspnetO – Quick Way To Learn Asp.net

**Example – [Method Overriding by using abstract method]**

public abstract class Base  
{  
       public abstract string BlogName();  
}

public class Child : Base  
{  
       //same method name with same signature/parameters  
       public override string BlogName()  
       {  
               //It's mandatory to implement abstract method in derived/child class  
               return "AspnetO – Quick Way To Learn Asp.net";  
       }  
}

public class Program  
{  
       public static void Main(string[] args)  
       {  
               Base objBase = new Child();  
               Console.WriteLine(objBase.BlogName());  
               Console.Read();  
       }  
}

In above example, when you run the program, at compile time the type of *objBase* is *Base* but it will still call the child class’s override method because at the *runtime*, the type of the *objBase* object refers to is *Child*.

**Example Result**

AspnetO – Quick Way To Learn Asp.net

**Note:** Method Overloading and Method Overriding both are different OOP concepts and important also. Don’t be panic with their names it looks similar.

9] Constructors:

Constructors are special methods, used when instantiating a class. A constructor can never return anything, which is why you don’t have to define a return type for it.

A normal method is defined like this:

public string bike()  
{  
}

A simple constructor(without parameters) can be defined like this:

public bike()  
{  
}

And here is the example of parameterized constructor:

public class bike  
{  
       private int mileage;  
       private string color;

       public bike()  
       {  
               //constructor without parameter  
       }

       public bike(int mil, string col)  
       {  
               //constructor with two parameters "mil" and "col"  
               mileage = mil;  
               color = col;  
       }

       public void DisplayBikeData()  
       {  
               Console.WriteLine("Bike's Mileage is " + mileage + " and color is " + color);  
       }  
}

Key points to note about constructor are:

* If no constructor defined then the CLR(Common Language Runtime) will provide an implicit constructor which is known as a *Default Constructor*.
* Constructor doesn’t return a value.
* Constructors can be overloaded.
* A class can have any number of constructors and they vary with the number of arguments that are passed, which is they should have different parameters or signatures.
* We don’t use references or pointers on constructors because their addresses cannot be taken.
* Constructor doesn’t be declared with the *virtual* keyword.

10] Destructors:

Since garbage cleanup is automatic system, framework will free the objects that are no longer in use BUT there may be times where we need to do some manual cleanup. In this case we can use Destructor, which is used to destroy the objects that we no longer want to use.

A destructor method called once an object is disposed, can be used to cleanup resources used by the object. Destructors don’t look very much like other methods.

Here is an example of a destructor for our Bike class:

public class Bike  
{  
       public Bike()  
       {  
               //Constructor  
       }  
       ~Bike()  
       {  
               //Destructor  
       }  
}

Once the class object is instantiated, *Constructor* will be called and when object is collected by the garbage collector, *Destructor* method will be called.