**C# Preparation**

**SOLID PRINCIPLES**

**1. Separation of Concerns**

Logger Implementation or Connection String Implementation

**2. Open Closed Principle**

public class Employee

{

Id, name, salary

Employee(id, name)

CalculateBonus()

{

Return salary \* 0.1M;

}

}

Employee class having a CalculateBonus() – by default 10%.

Later on there’s a new requirement that Permanent employees should have 10% bonus, but Temporary should have 5%.

What we can do is introduce a new Property called Type which stores whether it is a “Permanent” or “Temporary” employee. And put if/else block to calculate respective bonuses.

This is violating OCP.

Solution is to either implement Employee as abstract class/or interface/or Virtual Override

**3. Liskov Substitution Principle**

If S is a subtype of T, then objects of type T may be replaced with objects of type S

It is considered as the extension of OCP.

Example

In the above example of employees, let’s say we solve the OCP by making Employee class as abstract and calculateBonus() also as abstract. But there’s a new type of employee called ContractEmployee. This employee doesn’t have Bonus. Then how will you handle its overridden CalculateBonus()?

In brute force approach, this is how :-

public class ContractEmployee()

{

public override decimal CalculateBonus(decimal salary)

{

throw new exception();

}

}

The problem with this case is that now you cannot replace the object of Employee with the object of ContractEmployee. This is because if you do this :-

Employee emp = new ContractEmployee(1, “abc”);

You’ll get an exception if you call emp.ContractEmployee()

One way to solve this is to use interfaces.

public interface IEmployee

{

GetMinSalary();

}

Public interface Bonus

{

CalculateBonus();

}

Let’s say there’s a method GetMinSalary() which every class wants to use. So we’ll implement this interface for all the 4 classes (1 Parent + 3 subclasses) and the other interface only for the classes which require it.

This way, we can do this…

IEmployee e = // any of the 4 classes

**4. Interface Segregation Principle**

“No client should be forced to depend on methods it doesn’t use”

Or

“One fat interface need to split into many smaller ones so that the client can know their relevant interfaces”

Interface segregation principle is required to solve the design problem of the application. When all the tasks are done by a single class or in other words, one class is used in almost all the application classes then it has become a fat class with overburden. Inheriting such class will results in having sharing methods which are not relevant to derived classes but its there in the base class so that will inherit in the derived class.

Example:-

public interface IRepository

{

Void writeData(object Data);

Object ReadData();

}

Public class DatabaseRepository : IRepository

{

//implements both the interfaces methods

}

Public class XMLRepository: IRepository

{

//implements only write

//So ReadData() with throw exception

}

So here, instead of having a single fat IRepository interface, we should have 2 different interfaces for each Read and write

**5. Dependency Inversion Principle**

“High level modules should not depend on low level modules. Both should depend on abstractions.”

“Abstraction should not depend on details, details should depend on abstraction”

Example :-

1. Repository Pattern
2. Business Level layer directly depending on Data Access Layer

Causes unit testing issues…

**DESIGN PATTERNS**

**1. Factory Method**

Factory Method is a Design Pattern which defines an interface for creating an object but lets the classes that implement the interface decide which class to instantiate. Factory Pattern lets a class postpone instantiation to sub-classes.

The factory pattern is used to replace class constructors, abstracting the process of object generation so that the type of the object instantiated can be determined at run-time.

The classes and objects participating in the above UML class diagram are as follows:

1. *Product*

This defines the interface of objects the factory method creates

1. *ConcreteProduct*

This is a class that implements the Product interface.

1. *Creator*

This is an abstract class and declares the factory method, which returns an object of type Product.

This may also define a default implementation of the factory method that returns a default ConcreteProduct object.

This may call the factory method to create a Product object.

1. *ConcreteCreator*

This is a class that implements the Creator class and overrides the factory method to return an instance of a ConcreteProduct.

Assume you have three different cards which are considered here as classes MoneyBack, Titanium and Platinum, all of them implement abstract class CreditCard. You need to instantiate one of these classes, but you don't know which of them, it depends on the user. This is a perfect scenario for the Factory Method design pattern.

Who is what?

The classes and objects participating in the above class diagram can be identified as follows:

1. Product - CreditCard
2. ConcreteProduct- MoneyBackCreditCard, TitaniumCreditCard, PlatinumCreditCard
3. Creator- CardFactory
4. ConcreteCreator- MoneyBackCardFactory, TitaniumCardFactory, PlatinumCardFactory

2. Abstract Factory

AF provides an interface for creating families of related or dependent objects without specifying their concrete classes. We can say it is just an object maker which can create more than one type of object.

The object it produces is known to the client only by that object's interface, not by the object's actual concrete implementation.

## **How does it differ from Factory Method?**

First of all, both of them fall under Creational category and it means both will solve the problem relating to object creation. Factory Method and Abstract Factory design pattern are about creating objects.

## **Factory Method Design Pattern**

Here, we define an interface which will expose a method which will create objects for us. Return type of that method is never a concrete type; rather, it will be some interface (or may be an abstract class).

* Creates object through inheritance
* Produce only one product
* Implements code in the abstract creator that makes use of the concrete type that sub class produces

## **Abstract Factory Design Pattern**

Here, we define an interface which will create families of related or dependent objects. In simple words, interface will expose multiple methods each of which will create some object. Again, here method return types will be generic interfaces. All these objects will together become part of some important functionality.

* Creates object through composition
* Produce families of products
* Concrete factories implements factory method to create product

The classes and objects participating in the above UML class diagram are as follow.

1. *AbstractFactory*  
   This is an interface for operations which is used to create abstract product.
2. *ConcreteFactory*  
   This is a class which implements the AbstractFactory interface operations to create concrete products.
3. *AbstractProduct*  
   This declares an interface for a type of product object
4. *Product*  
   This defines a product object to be created by the corresponding concrete factory also implements the AbstractProduct interface
5. *Client*  
   This is a class which uses AbstractFactory and AbstractProduct interfaces to create a family of related objects.

Now, let’s understand this with a real world example,

The example here has an implementation of an Abstract Factory as an Interface IMobilePhone that has methods that can create a Smart Phone object and a Normal Phone object. The client codes against IMobilePhone and gets ISmartPhone and INormalPhone interfaces.

In case of "Nokia", it creates a family of Nokia objects (SmartPhone and NormalPhone) and in case of "Samsung", creates a family of Samsung objects (SmartPhone and NormalPhone).

The client doesn't care which object (Nokia SmartPhone and NormalPhone or Samsung SmartPhone and NormalPhone), IMobilePhone interface returns as it codes against ISmartPhone and INormalPhone interface.

**Who is what?**

The classes and objects participating in the above class diagram can be identified as shown below.

* AbstractFactory- IMobilePhone
* ConcreteFactory - Nokia, Samsung
* AbstractProduct- ISmartPhone, INormalPhone
* Product- NokiaPixel, Nokia1600, SamsungGalaxy, SamsungGuru
* Client- MobileClient

**GARBAGE COLLECTION**

<https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals>

**EXTENSION METHODS**

Extension methods are those methods which allow you to inject additional methods to a class without modifying, deriving or recompiling the original class, struct or interface. Extension methods can be added to your own custom class, .NET framework classes, or third party classes or interfaces.

Now let's see how to write an extension method.

An extension method is actually a special kind of static method defined in a static class. To define an extension method, first of all, define a static class.

For example, we have created an IntExtensions class under the ExtensionMethods namespace in the following example. The IntExtensions class will contain all the extension methods applicable to int data type. (You may use any name for namespace and class.)

namespace ExtensionMethods

{

public static class IntExtensions

{

}

}

Now, define a static method as an extension method where the first parameter of the extension method specifies the type on which the extension method is applicable. We are going to use this extension method on int type. So the first parameter must be int preceded with the ***this*** modifier.

For example, the IsGreaterThan() method operates on int, so the first parameter would be, this int i.

namespace ExtensionMethods

{

public static class IntExtensions

{

public static bool IsGreaterThan(this int i, int value)

{

return i > value;

}

}

}

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

using ExtensionMethods;

class Program

{

static void Main(string[] args)

{

int i = 10;

bool result = i.IsGreaterThan(100);

Console.WriteLine(result);

}

}

Linq implements all of its functionality using Extension Methods.

**TASKS vs THREADS**

**Task is more abstract then threads**. It is always advised to use tasks instead of thread as it is created on the thread pool which has already system created threads to improve the performance.

**Thread Does Not Return Results**– Thread cannot be used to directly return result from the method. Though we can use shared field to get the result whereas with task we can always get the return values

**No Continuation in Thread-**We cant tell a thread to start some other operation when it is done with it work. Though we can wait for the thread to complete by using Join() but that will block out main thread.On the other hand we can tell the task to continue with the other task by using the ContinueWith() method of the task

**Cancellation** – We cannot cancel a thread while it is in middle of the operation. But cancellation in tasks are supported with the help of CancellationTokeSource class

**Exception –**One of the major difference between task and thread is the propagation of exception. While using thread if we get the exception in the long running method it is not possible to catch the exception in the parent function but the same can be easily caught if we are using tasks.

# **IS VS AS OPERATOR KEYWORD IN C#**

The difference between [***is***](https://www.geeksforgeeks.org/c-is-operator-keyword/) and [***as***](https://www.geeksforgeeks.org/c-as-operator-keyword/) operators are as follows:

* The **is** operator is used to check if the run-time type of an object is compatible with the given type or not whereas **as** operator is used to perform conversion between compatible reference types or Nullable types.
* The **is** operator is of boolean type whereas **as** operator is not of boolean type.
* The **is** operator returns true if the given object is of the same type whereas **as** operator returns the object when they are compatible with the given type.
* The **is** operator returns false if the given object is not of the same type whereas **as** operator return null if the conversion is not possible.
* The **is** operator is used for only reference, boxing, and unboxing conversions whereas **as** operator is used only for nullable, reference and boxing conversions.

**YIELD**

Yield is a  [contextual keyword](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/#contextual-keywords) which is used to do custom stateful iteration over a collection. When we use it in a statement, we indicate that the method, operator, or get accessor in which it appears is an iterator. Using yield to define an iterator removes the need for an explicit extra class (the class that holds the state for an enumeration, see [IEnumerator<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerator-1) for an example) when you implement the [IEnumerable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable) and [IEnumerator](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerator) pattern for a custom collection type.

**Advantages**

# The benefit of using the yield keyword is that it lazily generate a sequence of objects. For e.g, the Enumeration.Range does not have any kind of collection internally. It generates the next number on demand.

# **ANONYMOUS TYPE**

In C#, an anonymous type is a type (class) without any name that can contain public read-only properties only. It cannot contain other members, such as fields, methods, events, etc.

You create an anonymous type using the *new* operator with an [object initializer](https://www.tutorialsteacher.com/csharp/csharp-object-initializer) syntax. The [implicitly typed variable- var](https://www.tutorialsteacher.com/csharp/csharp-var-implicit-typed-local-variable) is used to hold the reference of anonymous types.

The following example demonstrates creating an anonymous type student with three properties named Id, FirstName, and LastName.

var student = new { Id = 1, FirstName = "James", LastName = "Bond" };

The properties of anonymous types are read-only and cannot be initialized with a null, anonymous function, or a pointer type. The properties can be accessed using dot (.) notation, same as object properties.

In Linq, we return anonymous type a lot of times in Select method.

**ANONYMOUS METHOD**

As the name suggests, an anonymous method is a method without a name. Anonymous methods in C# can be defined using the delegate keyword and can be assigned to a variable of delegate type.

Anonymous methods can access variables defined in an outer function.

Example:

public delegate void Print(int value);

static void Main(string[] args)

{

Print print = delegate(int val) {

Console.WriteLine("Inside Anonymous method. Value: {0}", val);

};

print(100);

}

**LAMBDA EXPRESSION**

The lambda expression is a shorter way of representing [anonymous method](https://www.tutorialsteacher.com/csharp/csharp-anonymous-method) using some special syntax.

For example, following anonymous method checks if student is teenager or not:

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

The above anonymous method can be represented using a Lambda Expression in C# and VB.Net as below:

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

## **Assign Lambda Expression to Delegate**

The lambda expression can be assigned to Func<in T, out TResult> type delegate. The last parameter type in a Func delegate is the return type and rest are input parameters. Visit [Func delegate](https://www.tutorialsteacher.com/csharp/csharp-func-delegate) section of C# tutorials to know more about it.

Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;

Student std = new Student() { age = 21 };

bool isTeen = isStudentTeenAger(std);// returns false

**EXPRESSION**

Expression are basically the types introduced by Linq which are used to generate runtime executable query.

Lambda Expression can be assigned to the Func or Action type delegates to process over in-memory collections. The .NET compiler converts the lambda expression assigned to Func or Action type delegate into executable code at compile time.

LINQ introduced the new type called [Expression](https://msdn.microsoft.com/en-us/library/bb335710(v=vs.110).aspx) that represents strongly typed lambda expression. It means lambda expression can also be assigned to Expression<TDelegate> type. The .NET compiler converts the lambda expression which is assigned to Expression<TDelegate> into an [Expression tree](https://www.tutorialsteacher.com/linq/expression-tree) instead of executable code. This expression tree is used by remote LINQ query providers as a data structure to build a runtime query out of it (such as LINQ-to-SQL, EntityFramework or any other LINQ query provider that implements IQueryable<T> interface).

Example

Expression<Action<Student>> printStudentName = s => Console.WriteLine(s.StudentName);

- **Both int.Parse and Convert.ToInt32** are used to convert string into the integer but Only difference between them is to Convert.ToInt32 handle null and returns '0' as output and int.parse is not going to handle NULL and will give a Argument Null Exception.

Therefore, Convert.ToInt32 will take a little longer. So, if you are using them in large iterations, and you’re sure that string contains numerical value, then use Parse. Otherwise, and specially in the case of user input, take Convert.ToInt32.

Int.TryParse() can be used in combination with int.Parse(). It returns a bool value depending on whether the string value can be converted or not.

**== vs Equals**

1. **Equality Comparison**

Value Types – Both will always do value comparison

Reference Types – Always reference comparison unless we override it in the class.

String – Always value comparison.

But when we take the reference of a string in object, we do value comparison in case of Equals and reference comparison in case of ==.

2. **Null Checks**

== will not throw any exception if one of the value is null, Equals will throw.

3. **Compile Time vs Runtime**

== will compare during compile time, while Equals will compare during runtime.

4. **Parameter specification**

We can specify additional parameters to Equals while doing the string comparison like IgnoreCase etc.

== and Equals by default always give same result, unless in following condition :-

object str = newstring(newchar[] { 't', 'e', 's', 't' });

object str1 = newstring(newchar[] { 't', 'e', 's', 't' });

Console.WriteLine(str==str1); *// false*

Console.WriteLine(str.Equals(str1)); *// true*

**AGGREGATION, COMPOSITION and ASSOCIATION**

These refers to the type of relationships which exist between different objects in our application.

Aggregation and composition basically describes the container relationship.

**AGGREGATION**

Aggregation is a directional relationship where one object can own the another object, but these two can also survive independently.

For e.g, Teacher – Student. Employee - Address

Realistically, a list holding objects which does not demand that all objects gets disposed when the list itself is disposed is called aggregation.

**COMPOSITION**

Composition is a special or stronger type of Aggregation when the two objects are dependent on one another and when we delete the Parent, we delete the child as well.

For e.g the relationship between Person and DateOfBirth. It doesn’t make sense to keep the date of birth if the person doesn’t exist.

**ASSOCIATION**

Association is the super set of these relationships and can be used to represent any of the one-to-one etc relationships.

For e.g Teacher student relationships..

**SHALLOW AND DEEP COPY**

**Shallow**

Shallow copying is creating a new object and then copying the non static fields of the current object to the new object. If the field is a value type, a bit by bit copy of the field is performed. If the field is a reference type, the reference is copied but the referred object is not, therefore the original object and its clone refer to the same object. A shallow copy of an object is a new object whose instance variables are identical to the old object. In .Net shallow copy is done by the object method MemberwiseClone().

Any changes done in one object will get reflected in another.

**Deep**

Deep copy is creating a new object and then copying the non-static fields of the current object to the new object. If a field is a value type, a bit by bit copy of the field is performed. If a field is a reference type, a new copy of the referred object is performed. A deep copy of an object is a new object with entirely new instance variables, it does not share objects with the old.

**DISPOSE VS FINALIZE**

**Finalize**

The [Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netcore-3.1) 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 class or through a derived class.

If a type does override the [Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netcore-3.1) method, the garbage collector adds an entry for each instance of the type to an internal structure called 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](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netcore-3.1) method automatically under the following conditions:

* 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](https://docs.microsoft.com/en-us/dotnet/api/system.gc.suppressfinalize?view=netcore-3.1) method.
* **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.

[Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netcore-3.1) operations have the following limitations:

* 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](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose?view=netcore-3.1) implementation.
* The finalizers of two objects are not guaranteed to run in any specific order, even if one object refers to the other. That is, if Object A has a reference to Object B and both have finalizers, Object B might have already been finalized when the finalizer of Object A starts.

The [Finalize](https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize?view=netcore-3.1) method might not run to completion or might not run at all under the following exceptional circumstances:

* If another finalizer blocks indefinitely (goes into an infinite loop, tries to obtain a lock it can never obtain, and so on). Because the runtime tries to run finalizers to completion, other finalizers might not be called if a finalizer blocks indefinitely.
* If the process terminates without giving the runtime a chance to clean up. In this case, the runtime's first notification of process termination is a DLL\_PROCESS\_DETACH notification.

Differences

- Major difference between Dispose and Finalize is that the dispose is a deterministic call while you cannot not sure when Finalize will be called.

- Dispose method is called by the user code but finalize is called by the Garbage Collector.

- Dispose method can be called explicitly but Finalize is called immediately before the object destruction.

- There’s a performance impact in case of Finalize because we cannot be sure when will it be called. Dispose is called immediately.

- The Finalizers of two objects are not guaranteed to run in any specific order. But dispose will be called in specific order.

- Though it is not a good practice, but we can call Dispose method multiple times. But the object will ignore all the calls after the first one. Finalize is called only once.

**To create an immutable class is a three step process:**

Step 1: Remove the setters of the **class**, only have getters. The first step towards **creating** an **immutable class** is to remove the setters. ...

Step 2: Provide parameters via constructor.

Step 3: **Make** the variables of the property READONLY.

**Why do we need Run Time Polymorphism**

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

**ABSTRACTION VS ENCAPSULATION VS DATA HIDING**

**ENCAPSULATION**

The process of bundling data and corresponding methods (behavior) together into a single unit is called **encapsulation.**   
  
In other words, encapsulation is a programming technique that binds the class members (variables and methods) together and prevents them from being accessed by other classes, thereby we can keep variables and methods safes from outside interference and misuse.

Every C# class is an example of encapsulation because we write everything within the class only that binds variables and methods together and hides their complexity from other classes.

**ABSTRACTION**

Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user. For example, when you login to your Amazon account online, you enter your user\_id and password and press login, what happens when you press login, how the input data sent to amazon server, how it gets verified is all abstracted away from the you.

Technical Example:

Let’s say we are working on a Retail application. We are in the business logic and we have to calculate the final price of a product. Now, before calculating the final price, we have to consider some situations. Like there might be some discount applied on the product. There could be several complexities while applying discount, like whether it is a coupon applied or there is some promotional offer by the store. All my business logic should get is the final price. The task of applying discount should not be done here, instead it should be abstracted away in Discount class.

**DATA HIDING**

Encapsulation leads to Data Hiding.

Data hiding is the process is hiding the private data of a class and exposing only what is required by the outer world.

For example, while working on a banking application, the a/c balance should always be a private field. We must expose some methods which checks whether the user who requires access is authentic or not and only then show him the balance.

**INHERITANCE VS COMPOSITION**

Both inheritance and composition are used to define relationships between classes in Object oriented programming.

- Major difference between these two is that Inheritance is an “is-a” relationship. For e.g a car is a Vehicle, a Customer is a Person.

But composition is used to define a “has-a” relationship. For example, a Person has an address, a Car has an engine.

- Inheritance results in a tightly coupled code as any change made in the parent or superclass is reflected back in the child class and there might be a code break here. But composition results in a loosely coupled code and therefore we can easily change the component parts without breaking the code.

**GLOBAL ASSEMBLY CACHE**

Each computer where the Common Language Runtime is installed has a machine-wide code cache called the Global Assembly Cache. The Global Assembly Cache stores assemblies specifically designated to be shared by several applications on the computer.

A private assembly is an assembly that is available to particular application where they are kept, and a Shared Assembly is a public assembly that is shared by multiple applications. That means, a Private Assembly cannot be references outside the scope of the folder where they are kept and a Shared Assembly is one that can be referenced by more than one application.

In order to share an assembly, the assembly must be explicitly built for this purpose by giving it a cryptographically strong name . By contrast, a private assembly name need only be unique within the application that uses it.

The classes that ship with the .NET Framework are all built as shared assemblies.

Shared Assemblies are stored in GAC.

**TYPES OF JIT**

Pre-JIT: - Pre-JIT compiles complete source code into native code in a single  
compilation cycle. This is done at the time of deployment of the application.

Econo-JIT: - Econo-JIT compiles only those methods that are called at runtime.  
However, these compiled methods are removed when they are not required.

Normal-JIT: - Normal-JIT compiles only those methods that are called at runtime.  
These methods are compiled the first time they are called, and then they are stored in  
cache. When the same methods are called again, the compiled code from cache is used  
for execution.

**VAR vs DYNAMIC**

- var is a statically typed variable. It results in a strongly typed variable, in other words the data type of these variables are inferred at compile time. dynamic are dynamically typed variables. This means, their type is inferred at run-time and not the compile time in contrast to var type.

- var type of variables are required to be initialized at the time of declaration or else they encounter the compile time error. dynamic type variables need not be initialized when declared.

- var does not allow the type of value assigned to be changed after it is assigned to. This means that if we assign an integer value to a var then we cannot assign a string value to it. dynamic allows the type of value to change after it is assigned to initially.

- var variables cannot be used for property or return values from a function. They can only be used as local variable in a function. dynamic variables can be used to create properties and return values from a function.

- var supports intellisense while dynamic doesn’t.

**OBJECT VS DYNAMIC**

**- Purpose**: Object is used as a base type for all the reference types in c#, while dynamic was introduce to ease the interoperability feature in .net framework.

- **Type Checking –** Dynamic type variables handle type checking at compile time, while object type variable does it during compile time.

- **Functions**: All of the base functions like Equals, ToString etc are available for the type object but not in the case of Dynamic.

- **Intellisense**: object type supports intellisense while dynamic doesn’t.

- **Operator:** Cannot perform any mathematic operator on types object (first we need to do unboxing). Can do on dynamic.

object a = 10;

object b = 20;

object c = a + b; //Illegal

dynamic d = 12;

dynamic e = 33;

dynamic f = d + e; // 45

**REFLECTION**

Reflection is a process by which a program can examine and manipulate program objects at run time using the assemblies.  
Common use of Reflection:

– Load assemblies at runtime  
– it allows you to learn what assembly defines a particular item such as a class or enumeration  
– List a class’s field,properties, constructors, event and methods  
– Get information about a property such as type and if it is read only  
– Get and Set property’s value  
– Get information about item’s attribute etc..