**C# Preparation**

**SOLID PRINCIPLES**

**1. Separation of Concerns**

Logger Implementation or Connection String Implementation

**2. Open Closed Principle**

A class must be open for extension and closed for modification.

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.

Here we are modifying the existing class which is violating OCP. If we go this way, we will have to perform a lot of testing to whether the Employee class is working fine or not.

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.

In .Net library, we have IEnumerable and ICollection. This is also an example of Interface segregation principle.

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

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.

**TASK**

Tasks are constructs used to implement what is known as the [Promise Model of Concurrency](https://en.wikipedia.org/wiki/Futures_and_promises). In short, they offer you a "promise" that work will be completed at a later point, letting you coordinate with the promise with a clean API.

* Task represents a single operation which does not return a value.
* Task<T> represents a single operation which returns a value of type T.

It’s important to reason about tasks as abstractions of work happening asynchronously, and not an abstraction over threading. By default, tasks execute on the current thread and delegate work to the Operating System, as appropriate. Optionally, tasks can be explicitly requested to run on a separate thread via the Task.Run API.

Using await allows your application or service to perform useful work while a task is running by yielding control to its caller until the task is done. Your code does not need to rely on callbacks or events to continue execution after the task has been completed. The language and task API integration does that for you. If you’re using Task<T>, the await keyword will additionally "unwrap" the value returned when the Task is complete.

**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/Chaining 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.

**Parent/Child Relationship-** We can apply Parent/Child relationship in Tasks. A Task at one time becomes parent of multiple tasks. Parent Task does not complete until it's child tasks are completed. We do not have any such mechanim in Thread class.

# **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" \t "_blank) 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 an expression tree 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);

Here’s the link to further understand this - <https://www.tutorialsteacher.com/linq/linq-expression>

- **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**

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

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

- 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.

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.

**ASSEMBLY**

Assemblies form the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. Assemblies take the form of executable (.exe) or dynamic link library (.dll) files, and are the building blocks of .NET applications. They provide the common language runtime with the information it needs to be aware of type implementations.

In .NET and .NET Framework, you can build an assembly from one or more source code files. In .NET Framework, assemblies can contain one or more modules. This allows larger projects to be planned so that several developers can work on separate source code files or modules, which are combined to create a single assembly. For more information about modules, see How to: Build a multifile assembly.

Assemblies have the following properties:

* Assemblies are implemented as .exe or .dll files.
* For libraries that target .NET Framework, you can share assemblies between applications by putting them in the global assembly cache (GAC). You must strong-name assemblies before you can include them in the GAC. For more information, see Strong-named assemblies.
* Assemblies are only loaded into memory if they are required. If they aren't used, they aren't loaded. This means that assemblies can be an efficient way to manage resources in larger projects.
* You can programmatically obtain information about an assembly by using reflection. For more information, see Reflection (C#) or Reflection (Visual Basic).

**ASSEMBLY MANIFEST**

Every assembly has an assembly manifest file. Similar to a table of contents, the assembly manifest contains:

* The assembly's identity (its name and version).
* A file table describing all the other files that make up the assembly, such as other assemblies you created that your .exe or .dll file relies on, bitmap files, or Readme files.
* An assembly reference list, which is a list of all external dependencies, such as .dlls or other files. Assembly references contain references to both global and private objects. Global objects are available to all other applications. In .NET Core, global objects are coupled with a particular .NET Core runtime. In .NET Framework, global objects reside in the global assembly cache (GAC). System.IO.dll is an example of an assembly in the GAC. Private objects must be in a directory level at or below the directory in which your app is installed.

Because assemblies contain information about content, versioning, and dependencies, the applications that use them needn't rely on external sources, such as the registry on Windows systems, to function properly. Assemblies reduce .dll conflicts and make your applications more reliable and easier to deploy. In many cases, you can install a .NET-based application simply by copying its files to the target computer.

**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**

- The data type of var variables are inferred at compile time. dynamic are dynamically typed variables. This means, their type is inferred at run-time.

- 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 run 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..

**INTERFACES VS ABSTRACT CLASS**

|  |  |
| --- | --- |
| **ABSTRACT CLASS** | **INTERFACE** |
| Can have method implementation | Cannot have implementation |
| Can contain fields | Cannot contain fields |
| Multiple Inheritance cannot be achieved | MI can be achieved |
| We can specify different access modifiers | Cannot specify access modifiers |
| It can be fully, partially or not implemented | It should be fully implemented |
| Can contain static members | Cannot contain statics |

**When to Use Abstract Class vs Interface**

* + Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing a common functionality to unrelated classes.
  + Interfaces are also good when we want to have something similar to multiple inheritances since we can implement multiple interfaces.
  + Whenever we want to maintain state, we use abstract classes as interfaces cannot have fields.
  + If we are designing small, concise bits of functionality, use interfaces. If we are designing large functional units, use an abstract class.

**CONST VS READONLY**

|  |  |
| --- | --- |
| **CONST** | **READONLY** |
| It is a compile time constant | It is assigned at runtime |
| Necessary to assign value during declaration | Can be assigned in constructors also |
| It cannot be static | Can be static |
| Accessed using class name | Object reference |
| Can be declared inside a method | Cannot |

**DELEGATE**

A delegate is an abstraction over one or more function pointers in a type safe manner. With delegate, we can use a function as data. That is it allows us to pass function to be passed a parameter, return from a function like a value etc. They have a signature in form of return type and parameters and all the functions which are pointed to by this delegate must have the same signature.

**Why do we need to have delegates?** Delegates are used for implementing callbacks. They also help to implement anonymous methods.

**Multicast Delegate**

A multicast delegate is one where a delegates is pointing to multiple functions. When we invoke this delegate, all of these functions are called in FIFO order. When we want to add multiple function, we make use of overload operator +=, rather than the direct assignment.

Invoking multiple methods by one delegate may lead into a problematic situation. If one of the methods invoked by a delegate throws an exception, then the complete iteration would be aborted. You can avoid such a scenario by iterating the method invocation list on your own. The Delegate class defines a method GetInvocationList that returns an array of Delegate objects.

**EVENTS**

Events are basically wrappers around delegates which helps us to implement Publisher-Subscriber model in .Net. The class which raises the event is called Publisher and the class which received the notification is called as Subscriber.

Defining an event is a two-step process. First, you need to define a delegate type that will hold the list of methods to be called when the event is fired. Next, you declare an event using the event keyword.

**DELEGATE VS EVENT**

|  |  |
| --- | --- |
| **DELEGATE** | **EVENT** |
| It is basically a type safe function pointer. | It is an event notification mechanism that depends on delegates. |
| It is an independent concept | Wrapper around delegate |
| Accessed using Class Name | Accessed using class object |

**Singleton Vs Static Class**

|  |  |
| --- | --- |
| **Singleton** | **Static** |
| Fundamentally, it is a design pattern. | It is a keyword used in front of fields and class. |
| Singleton can be passed as parameter. | It cannot be passed as parameter. |
| It can implement interfaces. | Static class cannot implement interfaces. |
| We can create a single instance of Singleton. | Cannot create instance of static class. |
| They can inherit or be inherited from. | They cannot be. |
| They are stored on normal heap. | They are stored in High Frequency heap. |

**Ref vs Out**

|  |  |
| --- | --- |
| **Ref** | **Out** |
| The parameter or argument must be initialized first before it is passed to ref. | It is not compulsory to initialize a parameter or argument before it is passed to an out. |
| It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method. | A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method. |
| When we use REF, data can be passed bi-directionally. | When we use OUT data is passed only in a unidirectional way (from the called method to the caller method). |

**Indexer**

An indexer is a special type of property that allows a class or a structure to be accessed like an array for its internal collection. C# allows us to define custom indexers, generic indexers, and also overload indexers.

An indexer can be defined the same way as property with this keyword and square brackets [].

class StringDataStore

{

private string[] strArr = new string[10]; // internal data storage

public string this[int index]

{

get

{

if (index < 0 || index >= strArr.Length)

throw new IndexOutOfRangeException("Index out of range");

return strArr[index];

}

set

{

if (index < 0 || index >= strArr.Length)

throw new IndexOutOfRangeException("Index out of range");

strArr[index] = value;

}

}

}

StringDataStore strStore = new StringDataStore();

strStore[0] = "One";

strStore[1] = "Two";

strStore[2] = "Three";

strStore[3] = "Four";

for(int i = 0; i < 10 ; i++)

Console.WriteLine(strStore[i]);

**HASHTABLE VS DICTIONARY**

|  |  |
| --- | --- |
| **Hashtable** | **Dictionary** |
| Hashtable is a loosely typed (non-generic) collection, this means it stores key-value pairs of any data types. | Dictionary is a statically collection. So it can store key-value pairs of specific data types. |
| Data retrieval is slower than dictionary because of boxing-unboxing. | Data retrieval is faster than Hashtable. |
| Hashtable returns null if we try to find a key which does not exist. | Dictionary throws an exception if we try to find a key which does not exist. |
| Hashtable is thread safe. | Only public static members are thread safe in Dictionary. |
| Order of insertion is not maintained in hashtable | It is maintained |

**Sealed Modifier**

When applied to a class, it means that this particular class cannot be inherited. A method, indexer, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the sealed keyword before the [override](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override) keyword in the class member declaration.

* The main purpose of a sealed class is to take away the inheritance feature from the class users so they cannot derive a class from it. One of the best usage of sealed classes is when you have a class with static members. For example, the **Pens** and **Brushes** classes of the **System.Drawing** namespace.
* Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster.

**ICollection**

It is an interface defined in System.Collection.Generic namespace which implements IEnumerable interface and provides other functionality to a collection. For e.g, Lists and Dictionary have Add(), Remove(), clear(), Count methods and properties because they implement ICollection.

**IQueryable**

It is an interface defined in System.Linq namespace which implements IEnumerable interface and is used to form query expressions in c#. It helps us to save a lot of resources because it first forms the complete query and then executes it along the data provider. For e.g, if we have a Linq query in which we have specified a Where clause, IEnumerable will first load all the objects from Select clause in the memory and then apply the filter. But IQueryable will load only those resources which match the where clause condition.

**MULTI THREADING**

**Thread Synchronization**

Synchronization in C# is a mechanism that makes sure only one process or thread accesses the critical section of the program. All the other threads have to wait until the critical section is free before they can enter it. It is mainly used in case of transactions like deposit, withdraw etc.

Synchronization can be handled using various methods. These methods are divided into 4 categories in general. These are as follows:

* Blocking Methods (Join)
* Locking Constructs (lock)
* No blocking synchronization
* Signaling

**Locks**

Locks are one of the most basic way of achieving thread synchronization in c#.

**Monitor**

The Monitor class in C# provides a mechanism that synchronizes access to objects. Let us simplify the above definition. In simple words, we can say that, like the lock, we can also use this class to protect shared resources in a multi-threaded environment. This can be done by acquiring an exclusive lock on the object so that only one thread can enter into the critical section at any given point of time.

The Monitor is a static class and belongs to the System.Threading namespace.

**Difference between Monitor and lock in C#**

The lock is the shortcut for Monitor.Enter with try and finally. So, the lock provides the basic functionality to acquire an exclusive lock on a synchronized object. But, If you want more control to implement advanced multithreading solutions using TryEnter() Wait(), Pulse(), and PulseAll() methods, then the Monitor class is your option.

**Mutex**

A Mutex is like a C# lock, but it can work across multiple processes. In other words, Mutex can be computer-wide as well as application-wide.

A Mutex is a synchronization primitive that can also be used for interprocess synchronization. When two or more threads need to access a shared resource at the same time, the system needs a synchronization mechanism to ensure that only one thread at a time uses the resource. Mutex is a synchronization primitive that grants exclusive access to the shared resource to only one thread. If a thread acquires a Mutex, the second thread that wants to acquire that Mutex is suspended until the first thread releases the Mutex.

**Semaphore**

The Semaphore in C# is used to limit the number of threads that can have access to a shared resource concurrently. In other words, we can say that Semaphore allows one or more threads to enter into the critical section and execute the task concurrently with thread safety. So, in real-time, we need to use Semaphore when we have a limited number of resources and we want to limit the number of threads that can use it.

How does Semaphore work in C#?

The Semaphores are Int32 variables that are stored in operating system resources. When we initialize the semaphore object we initialize it with a number. This number basically used to limits the threads that can enter into the critical section.

So, when a thread enters into the critical section, it decreases the value of the Int32 variable with 1 and when a thread exits from the critical section, it then increases the value of the Int32 variable with 1. The most important point that you need to remember is when the value of the Int32 variable is 0, then no thread can enter into the critical section.

**DEADLOCK**

In simple words, we can define a deadlock in C# is a situation where two or more threads are unmoving or frozen in their execution because they are waiting for each other to finish.

For example, let’s say we have two threads Thread1 and Thread2 and at the same time let say we have two resources Resource1 and Resource2. The Thread1 locked the Resource1 and trying to acquire a lock on Respurce2. At the same time, Thread2 acquired a lock on Resource2 and trying to acquire a lock on Resource1.

**Ways to Avoid Deadlock**

* + by using Monitor.TryEnter method and setting a timeout value
  + by acquiring locks in a specific order

**Static Class**

The following list provides the main features of a static class:

* Contains only static members.
* Cannot be instantiated as it is abstract implicitly.
* Is sealed.
* Cannot contain Instance Constructors, but can contain static constructor.
* Cannot inherit except from Object class.
* All the static classes and members are stored in a special memory area called High Frequency Heap.

Creating a static class is therefore basically the same as creating a class that contains only static members and a private constructor. A private constructor prevents the class from being instantiated. The advantage of using a static class is that the compiler can check to make sure that no instance members are accidentally added. The compiler will guarantee that instances of this class cannot be created.

**Static Constructors**

Static constructors have the following properties:

* A static constructor doesn't take access modifiers or have parameters.
* A class or struct can only have one static constructor.
* Static constructors cannot be inherited or overloaded.
* A static constructor cannot be called directly and is only meant to be called by the common language runtime (CLR). It is invoked automatically.
* The user has no control on when the static constructor is executed in the program.
* A static constructor is called automatically. It initializes the class before the first instance is created or any static members are referenced. A static constructor runs before an instance constructor. A type's static constructor is called when a static method assigned to an event or a delegate is invoked and not when it is assigned. If static field variable initializers are present in the class of the static constructor, they're executed in the textual order in which they appear in the class declaration. The initializers run immediately prior to the execution of the static constructor.
* If you don't provide a static constructor to initialize static fields, all static fields are initialized to their default value as listed in Default values of C# types.
* The static constructor is called only when a static method assigned to an event or a delegate is invoked and not when it is assigned.

Q. What will happen if a static constructor throws an exception?

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.

Most commonly, a TypeInitializationException exception is thrown when a static constructor is unable to instantiate a type or for an unhandled exception occurring within a static constructor.

**Private Constructor**

A private constructor is a special instance constructor. It is generally used in classes that contain static members only. If a class has one or more private constructors and no public constructors, other classes (except nested classes) cannot create instances of this class.

The declaration of the private constructor prevents the automatic generation of a parameterless constructor. Note that if you do not use an access modifier with the constructor it will still be private by default. However, the private modifier is usually used explicitly to make it clear that the class cannot be instantiated.

Private constructors are used to prevent creating instances of a class when there are no instance fields or methods, such as the Math class, or when a method is called to obtain an instance of a class. If all the methods in the class are static, consider making the complete class static. For more information see Static Classes and Static Class Members.

**Few ways to avoid Instantiation of a Class**

Those ways are:

* Abstract
* Static Class
* Private and protected constructor

**Func, Action and Predicate**

* The Func delegate takes zero, one or more input parameters, and returns a value (with its out parameter).
* Action takes zero, one or more input parameters, but does not return anything.
* Predicate is a special kind of Func. It represents a method that contains a set of criteria mostly defined inside an if condition and checks whether the passed parameter meets those criteria or not.

**Advantages of Generics**

* Generics increase the reusability of the code. You don't need to write code to handle different data types.
* Generics are type-safe. You get compile-time errors if you try to use a different data type than the one specified in the definition.
* Generic has a performance advantage because it removes the possibilities of boxing and unboxing.

**Boxing and UnBoxing Examples**

Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type. When the common language runtime (CLR) boxes a value type, it wraps the value inside a System.Object instance and stores it on the managed heap. Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system in which a value of any type can be treated as an object.

Consider the following declaration of a value-type variable:

int i = 123;

The following statement implicitly applies the boxing operation on the variable i:

// Boxing copies the value of i into object o.

object o = i;

The result of this statement is creating an object reference o, on the stack, that references a value of the type int, on the heap. This value is a copy of the value-type value assigned to the variable i.

* A boxing conversion makes a copy of the value. So, changing the value of one variable will not impact others.

int i = 10;

object o = i; // boxing

o = 20;

Console.WriteLine(i); // output: 10

* The casting of a boxed value is not permitted. The following will throw an exception.

Example: Invalid Conversion Copy

int i = 10;

object o = i; // boxing

double d = (double)o; // runtime exception

First do unboxing and then do casting, as shown below.

Example: Valid Conversion Copy

int i = 10;

object o = i; // boxing

double d = (double)(int)o; // valid

Note:

Boxing and unboxing degrade the performance. So, avoid using it. Use generics to avoid boxing and unboxing. For example, use List instead of ArrayList.

**Managed Execution Process**

* **Choose a Compiler**
* **Compiling to MSIL** – A Portable Executable (PE) file is created in this step which contains the metadata and MSIL code. MSIL code is a set of CPU independent instructions which can be efficiently converted into native code. It includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations. Metadata describes the types in your code, including the definition of each type, the signatures of each type's members, the members that your code references, and other data that the runtime uses at execution time.
* **Compiling MSIL to Native Code** - Before you can run Microsoft intermediate language (MSIL), it must be compiled against the common language runtime to native code for the target machine architecture. .NET provides two ways to perform this conversion:
* A .NET just-in-time (JIT) compiler – on Demand Compiling
* Ngen.exe (Native Image Generator) – Example of Ahead of Time Compilation.
* **Running Code**

**Link -** <https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process>

**Linq**

* Linq Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. IEnumerable<T> queries are compiled to delegates. IQueryable and IQueryable<T> queries are compiled to expression trees. For more information, see Expression trees.
* Types that support IEnumerable<T> or a derived interface such as the generic IQueryable<T> are called queryable types. A queryable type requires no modification or special treatment to serve as a LINQ data source. If the source data is not already in memory as a queryable type, the LINQ provider must represent it as such. For example, LINQ to XML loads an XML document into a queryable XElement type.
* LINQ queries are based on generic types, which were introduced in version 2.0 of .NET Framework. You do not need an in-depth knowledge of generics before you can start writing queries. However, you may want to understand two basic concepts:
* When you create an instance of a generic collection class such as List<T>, you replace the "T" with the type of objects that the list will hold. For example, a list of strings is expressed as List<string>, and a list of Customer objects is expressed as List<Customer>. A generic list is strongly typed and provides many benefits over collections that store their elements as Object. If you try to add a Customer to a List<string>, you will get an error at compile time. It is easy to use generic collections because you do not have to perform run-time type-casting.
* IEnumerable<T> is the interface that enables generic collection classes to be enumerated by using the foreach statement. Generic collection classes support IEnumerable<T> just as non-generic collection classes such as ArrayList support IEnumerable.
* LINQ query variables are typed as IEnumerable<T> or a derived type such as IQueryable<T>. When you see a query variable that is typed as IEnumerable<Customer>, it just means that the query, when it is executed, will produce a sequence of zero or more Customer objects. If you prefer, you can avoid generic syntax by using the var keyword. The var keyword instructs the compiler to infer the type of a query variable by looking at the data source specified in the from clause.
* Query variable is the variable to which the result of a linq query is returned. Range variable the variable through which we iterate over the collection inside linq query.

**Type Conversion**

Sometimes need to copy a value into a variable or method parameter of another type. For example, you might have an integer variable that you need to pass to a method whose parameter is typed as double. Or you might need to assign a class variable to a variable of an interface type. These kinds of operations are called type conversions. In C#, you can perform the following kinds of conversions:

* **Implicit conversions**: No special syntax is required because the conversion always succeeds and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
* **Explicit conversions (casts)**: Explicit conversions require a cast expression. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.
* **User-defined conversions**: User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship. For more information, see User-defined conversion operators.
* **Conversions with helper classes**: To convert between non-compatible types, such as integers and System.DateTime objects, or hexadecimal strings and byte arrays, you can use the System.BitConverter class, the System.Convert class, and the Parse methods of the built-in numeric types, such as Int32.Parse.

For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes or interfaces. No special syntax is necessary because a derived class always contains all the members of a base class.

For reference types, an explicit cast is required if you need to convert from a base type to a derived type:

* // Create a new derived type.

Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.

Animal a = g;

// Explicit conversion is required to cast back

// to derived type. Note: This will compile but will

// throw an exception at run time if the right-side

// object is not in fact a Giraffe.

Giraffe g2 = (Giraffe)a;

For more info - <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions>

**Value and Reference Type**

* Value types and reference types are the two main categories of C# types. A variable of a value type contains an instance of the type. This differs from a variable of a reference type, which contains a reference to an instance of the type. By default, on assignment, passing an argument to a method, and returning a method result, variable values are copied. In the case of value-type variables, the corresponding type instances are copied.
* If a value type contains a data member of a reference type, only the reference to the instance of the reference type is copied when a value-type instance is copied. Both the copy and original value-type instance have access to the same reference-type instance. For e.g, struct is a value type. And it can contain a value type int and a reference type List. Here, same reference of List is hold by the copy and original.
* A nullable value type T? represents all values of its underlying value type T and an additional null value. You cannot assign null to a variable of a value type, unless it's a nullable value type.

**INHERITANCE**

Facts:

* Not all members of a base class are inherited by derived classes. The following members are not inherited:
* Static constructors, which initialize the static data of a class.
* Instance constructors, which you call to create a new instance of the class. Each class must define its own constructors.
* Finalizers, which are called by the runtime's garbage collector to destroy instances of a class.

Reference - <https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/inheritance>

**CONSTRUCTOR**Facts:

* If a derived class does not invoke a base-class constructor explicitly, the default constructor is called implicitly. Therefore, in this case it is necessary for base class to have a default constructor.

**ABSTRACT CLASSES**

When a base class declares a method as virtual, a derived class can override the method with its own implementation. If a base class declares a member as abstract, that method must be overridden in any non-abstract class that directly inherits from that class. If a derived class is itself abstract, it inherits abstract members without implementing them. Abstract and virtual members are the basis for polymorphism, which is the second primary characteristic of object-oriented programming.

You can declare a class as abstract if you want to prevent direct instantiation by using the new operator. An abstract class can be used only if a new class is derived from it. An abstract class can contain one or more method signatures that themselves are declared as abstract. These signatures specify the parameters and return value but have no implementation (method body). An abstract class doesn't have to contain abstract members; however, if a class does contain an abstract member, the class itself must be declared as abstract. Derived classes that aren't abstract themselves must provide the implementation for any abstract methods from an abstract base class.

An abstract class can have a constructor. In general, a class constructor is used to initialize fields. Along the same lines, an abstract class constructor is used to initialize fields of the abstract class. We would provide a constructor for an abstract class if we want to initialize certain fields of the abstract class before the instantiation of a child-class takes place.

We cannot create an object of Abstract Class but we can create a reference of it.

**Words of Wisdom**

* Always use Parse() or TryParse() when checking for int values in a string obtained through user input.
* When doing string manipulation inside a for loop, if there are too many manipulations, use string builder instead of string for performance.
* To determine whether two class instances refer to the same location in memory (which means that they have the same identity), use the static Object.Equals method. (System.Object is the implicit base class for all value types and reference types, including user-defined structs and classes).
* The accessibility of a member can never be greater than the accessibility of its containing type. For example, a public method declared in an internal type has only internal accessibility.
* Classes can only have public or internal modifiers. Exception – Nested classes.
* Static methods can be overloaded but not overridden, because they belong to the class, and not to any instance of the class.
* Although a field cannot be declared as static const, a const field is essentially static in its behavior. It belongs to the type, not to instances of the type. Therefore, const fields can be accessed by using the same ClassName.MemberName notation that's used for static fields. No object instance is required.

**MARSHALING**

Marshaling is the process of creating a bridge between managed code and unmanaged code; it is the homer that carries messages from the managed to the unmanaged environment and reverse. It is one of the core services offered by the CLR (Common Language Runtime.)

Because much of the types in unmanaged environment do not have counterparts in managed environment, you need to create conversion routines that convert the managed types into unmanaged and vice versa; and that is the marshaling process.

**TYPES OF HEAP MEMORY**

Loader Heap: contains CLR structures and the type system

High Frequency Heap: statics, MethodTables, FieldDescs, interface map

Low Frequency Heap: EEClass, ClassLoader and lookup tables

Stub Heap: stubs for CAS, COM wrappers, PInvoke

Large Object Heap: memory allocations that require more than 85k bytes

GC Heap: user allocated heap memory private to the app

JIT Code Heap: memory allocated by mscoreee (Execution Engine) and the JIT compiler for managed code

Process/Base Heap: interop/unmanaged allocations, native memory, etc.

**PRIVATE SET VS NO SET**

class Contact

{

public string Name { get; }

public string Address { get; private set; }

public Contact(string contactName, string contactAddress)

{

// Both properties are accessible in the constructor.

Name = contactName;

Address = contactAddress;

}

// Name isn't assignable here. This will generate a compile error.

//public void ChangeName(string newName) => Name = newName;

// Address is assignable here.

public void ChangeAddress(string newAddress) => Address = newAddress;

}