# Language

1. **What C#**

This an object oriented programming language that runs on .NET framework.

1. **What is .NET?**

.NET is a [software framework](http://en.wikipedia.org/wiki/Software_framework) developed by [Microsoft](http://en.wikipedia.org/wiki/Microsoft) that runs on [Microsoft Windows](http://en.wikipedia.org/wiki/Microsoft_Windows). It includes class libraries and CLR. CLR provides features like security, [memory management](http://en.wikipedia.org/wiki/Memory_management), and [exception handling](http://en.wikipedia.org/wiki/Exception_handling).

1. **What are component of .NET CLR, CTS, CLS?**

**CLR**: The Common Language Runtime (CLR) is the [virtual machine](http://en.wikipedia.org/wiki/Virtual_machine) component of [Microsoft's](http://en.wikipedia.org/wiki/Microsoft) [.NET framework](http://en.wikipedia.org/wiki/.NET_framework)and is responsible for managing the execution of .NET programs. In a process known as [just-in-time compilation](http://en.wikipedia.org/wiki/Just-in-time_compilation), the compiled code is converted into machine instructions that, in turn, are executed by the computer's [CPU](http://en.wikipedia.org/wiki/CPU).[[1]](http://en.wikipedia.org/wiki/Common_Language_Runtime#cite_note-1) The CLR provides additional services including [memory management](http://en.wikipedia.org/wiki/Memory_management), [type safety](http://en.wikipedia.org/wiki/Type_safety) and [exception handling](http://en.wikipedia.org/wiki/Exception_handling). All programs written for the .NET framework, regardless of [programming language](http://en.wikipedia.org/wiki/Programming_language), are executed by the CLR. It provides exception handling, [garbage collection](http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) and [thread management](http://en.wikipedia.org/wiki/Thread_management).

**CTS**: CTS stands for Common Types System. In .NET we have various languages like C#, VB.NET, etc. There may be many situations where we want code written in one language to be used in another. In order to ensure that we have a smooth communication between different languages, we have CTS. CTS ensures that data types defined in two different languages get compiled to a common data type so that code written in one language can be used by another.

**CLS**: CLS stands for Common Language Specifications. It is a subset of CTS. CLS is a set of rules or guidelines which if followed ensures that code written in one .NET language can be used by another .NET language. For example one rule is that we cannot have member functions with same name with case difference only i.e we should not have add() and Add(). This may work in C# because it is case-sensitive but if try to use that C# code in VB.NET, it is not possible because VB.NET is not case-sensitive.

1. **How a program is executed in .NET?**
2. **What is polymorphism, Abstraction, Encapsulation?**

**Polymorphism:** Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

* At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.
* Base classes may define and implement [virtual](http://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) methods, and derived classes can [override](http://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

**Abstraction:** Abstraction is the process of refining away all the unneeded, unimportant attributes of an object and keeps only the characteristics best suitable for your domain. If you have an actual problem and provide a specific solution, you can use abstraction to formalize a more generic solution that can also solve more problems that have the same common pattern. That way you can increase the re-usability for your components or use components made by other programmers that are made for the same domain, or even for different domains. For example list or collection, these are very abstract classes that you can use almost everywhere and in a lot of domains. Imagine if .net only implemented an EmployeeList class and a CompanyList that could only hold a list of employees and companies with specific properties. Such classes would be useless in a lot of cases and what a pain would it be if you had to re-implement the whole functionality for a CarList for example. So the "List" is ABSTRACTED away from Employee, Company and Car. The List by itself is an abstract concept that can be implemented by its own class.

Interfaces, abstract classes or inheritance and polymorphism are tools to provide abstraction in c#.

**Encapsulation:** Encapsulation means that a group of related properties, methods, and other members are treated as a single unit or object (Like class). Think Encapsulation like a capsule (medicine tablet) which hides medicine inside it. Encapsulation is wrapping, just hiding properties and methods. Encapsulation is used for hide the code and data in a single unit to protect the data from the outside the world.

Class is the best example of encapsulation. Properties and access modifiers are also examples of this. Class keeps all data in group and access modifier decides, what can be accessed by whom. If we don’t have classes, variables will be just flow here and there in all code. If we don’t have access modifiers any one can write any data, which is also bad. Similarly we use properties to control how a property can be written.

1. **What is type safety in C#?**

In general sense It is the characteristic of code that allows the developer to be certain that a value or object will exhibit certain properties (i.e., be of a certain type) so that he/she can use it in a specific way without fear of unexpected or undefined behavior. In terms of memory Type-safe code (in context of .NET) accesses only the memory locations it is authorized to access. For example, type-safe code cannot read values from another object's private fields. It accesses types only in well-defined, allowable ways.

# Type System:

1. **What is “this” keyword?**

The “this” keyword refers to the current instance of the class. Other usage are

1. as a modifier of the first parameter of an extension method
2. To create indexers (ability to loop like collection <http://msdn.microsoft.com/en-us/library/6x16t2tx.aspx> ).
3. [To pass parameters between constructors](http://www.codeproject.com/Articles/7011/An-Intro-to-Constructors-in-C%29)
4. To cast itself to another type
5. [To internally reassign value type (struct) value](http://stackoverflow.com/questions/194484/whats-the-strangest-corner-case-youve-seen-in-c-or-net/1800162#1800162).

*You just need to remember till point 2, others you can forget.*

1. **What is value type, reference type?**

A value type stores its contents in memory allocated on the stack. When the variable x goes out of scope because the method in which it was defined has finished executing, the value is discarded from the stack.  A reference type, such as an instance of a class or an array, is allocated in a different area of memory called the heap. This memory isn't returned to the heap when a method finishes; it's only reclaimed when C#'s garbage collection system determines it is no longer needed.

Inbuilt primitive types (int, char, bool, struct) are value type, string and classes are reference type.

1. **What are primitive types?**

The primitive types are identified through keywords, which are aliases for predefined types in the System namespace. In short Int, long, double … kind of basic types are primitive types.

1. **How to define custom type (value type and reference type)?**

Classes are custom reference types, struct are value types.

1. **What is default value of value and reference types?**

All bits are zeroed for value type. So for int it will be zero, bool (0 => false). Reference types are null.

1. **What is boxing and unboxing?**

Boxing is the act of converting a value-type instance to a reference-type instance. Unboxing reverses the operation, by casting the reference type object back to the original value type:

int x = 9;

object obj = x; // Box the int

int y = (int)obj; // Unbox the int

Unboxing requires an explicit cast. The runtime checks that the stated value type matches the actual object type, and throws an InvalidCastException if the check fails.

1. **How to get class name from an Object?**

GetType(),Gets the [Type](http://msdn.microsoft.com/en-IN/library/system.type.aspx) of the current instance. There is another function typeOf to get type information, but

Typeof(Foo) is a static type lookup; essentially it occurs at compile time, so you only get the explicitly named type.

GetType() is a dynamic type lookup; it's a virtual method that gets called at runtime and will give you the exact type even if you are using polymorphism.

1. **What is difference between getType() and TypeOf()?**

typeof is used when you want to get the Type instance representing a specific type. GetType gives the runtime type of the object on which it is called, which may be different from the declared type ( in case of inheritance).

1. **What is difference between tryparseXXX and parseXXX?**

Like TryParseInt, it takes and out parameter of type (int in this case) and try to convert into desired type, if it fails, out parameter will have default value of type, if successful in conversion then that out parameter will have value. No exception is thrown in failure. However ParseInt will throw exceptions if conversion to integer fails.

1. **What do you mean by “strings are immutable”?**

By immutable we mean, value can never be changed, when we change value of string, a new memory is allocated and updated string is stored, old string is still in memory waiting for garbage collector.

1. **Why do we use stringbuilder?**

As we know string is immutable, so if we require an operation where lot of string operations (concatenation, replace,…) are required then string will consume lot of memory as every time a new instance will be created. Using Stringbuilder resolves this issue, it let you change (append, replace, remove) strings and at the end you can call ToString method to get the final string.

# Class

1. **What is difference between class and structure?**

Class is a reference type and struct is value type. (<http://geekswithblogs.net/BlackRabbitCoder/archive/2010/07/29/c-fundamentals-the-differences-between-struct-and-class.aspx>)

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Struct** | **Class** | **Notes** |
| **Is a reference type?** | No | Yes\* |  |
| **Is a value type?** | Yes | No |  |
| **Can have nested Types (enum, class, struct)?** | Yes | Yes |  |
| **Can have constants?** | Yes | Yes |  |
| **Can have fields?** | Yes\* | Yes | Struct instance fields cannot be initialized, will automatically initialize to default value. |
| **Can have properties?** | Yes | Yes |  |
| **Can have indexers** | Yes | Yes |  |
| **Can have methods?** | Yes | Yes |  |
| **Can have events?** | Yes\* | Yes | Structs, like classes, can have events, but care must be taken that you don’t subscribe to a copy of a struct instead of the struct you intended. |
| **Can have static members (constructors, fields, methods, properties, etc.)?** | Yes | Yes |  |
| **Can inherit?** | No\* | Yes\* | Classes can inherit from other classes (or object by default). Structs always inherit from System.ValueType and are sealed implicitly |
| **Can implement interfaces?** | Yes | Yes |  |
| **Can overload constructor?** | Yes\* | Yes | Struct overload of constructor does not hide default constructor. |
| **Can define default constructor?** | No | Yes | The struct default constructor initializes all instance fields to default values and cannot be changed. |
| **Can overload operators?** | Yes | Yes |  |
| **Can be generic?** | Yes | Yes |  |
| **Can be partial?** | Yes | Yes |  |
| **Can be sealed?** | Always\* | Yes | Structs are always sealed and can never be inherited from. |
| **Can be referenced in instance members using this keyword?** | Yes\* | Yes | In structs, this is a value variable, in classes, it is a readonly reference. |
| **Needs new operator to create instance?** | No\* | Yes | C# classes must be instantiated using new. However, structs do not require this. While new can be used on a struct to call a constructor, you can elect not to use new and init the fields yourself, but you must init all fields and the fields must be public! |

1. **What is constructor?**

A constructor is a method in the class which gets executed when you use the [new](http://msdn.microsoft.com/en-us/library/51y09td4.aspx) expression to create an object of a [class](http://msdn.microsoft.com/en-us/library/0b0thckt.aspx). A class may have multiple constructors that take different arguments. Constructors enable the programmer to set default values.

1. **Can we have multiple constructors, How it is decided which one is to be called?**

Yes we can have multiple constructors, We also may not have any constructor at all, in that case a default parameter less constructor is auto created by framework. When we have multiple constructor then, as in case of method overloading, the best matched method based on parameter is selected for execution. One Constructor can call another constructor.

1. **What is overloading?**

Overloading is what happens when you have two methods with the same name but different signatures. At compile time, the compiler works out which one it's going to call, based on the compile time types of the arguments and the target of the method call.

The signature of a method consists of the name of the method and the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. The signature of a method specifically **does not include the return type**, nor does it include the params modifier that may be specified for the right-most parameter.

1. **What is static?**

A static class is a class which can only contain static members, and therefore cannot be instantiated. A static member, unlike a regular (instance) member, is not associated with an instance of the class.

Instead, it is referenced through the type name. While an instance of a class contains a separate copy of all instance fields of the class, there is only one copy of each static field.

1. **What is static constructor? When it is called?**

A static constructor is used to initialize any [static](http://msdn.microsoft.com/en-us/library/98f28cdx.aspx) data, or to perform a particular action that needs to be performed once only. It is called automatically before the first instance is created or any static members are referenced.

class SimpleClass

{

// Static variable that must be initialized at run time.

static readonly long baseline;

// Static constructor is called at most one time, before any

// instance constructor is invoked or member is accessed.

static SimpleClass()

{

baseline = DateTime.Now.Ticks;

}

}

1. **Difference between Instance vs Static members?**

Instance members are specific to an object of a class; however static members are at class level hence shared by all objects of that class. To use instance members we write objectname. Membername, but for static we write classname.membername.

1. **What are constants?**

Constants are immutable values which are known at compile time and do not change for the life of the program. Constants are declared with the [const](http://msdn.microsoft.com/en-us/library/e6w8fe1b.aspx) modifier. Only the C# built-in types may be declared as const.

class Calendar1

{

public const int month = 12;

}

The constant month is always 12, and it cannot be changed even by the class itself. In fact, when the compiler encounters a constant identifier in C# source code (for example, months), it substitutes the literal value directly into the intermediate language (IL) code that it produces. Because there is no variable address associated with a constant at run time, const fields cannot be passed by reference

1. **What are readonly and how they are different from constants?**

As constants are resolved at compile time and does not support reference types, read only is used for that purpose. Use the [readonly](http://msdn.microsoft.com/en-us/library/acdd6hb7.aspx) modifier to create a class, struct, or array that is initialized one time at runtime (for example in a constructor) and thereafter cannot be changed.

const:

* Can't be static.
* Value is evaluated at compile time.
* Initiailized at declaration only.

readonly:

* Can be either instance-level or static.
* Value is evaluated at run time.
* Can be initialized in declaration or by code in the constructor.

1. **What is implicit conversion of types?**

**Conversion**: C# is statically-typed at compile time, after a variable is declared, it cannot be declared again or used to store values of another type unless that type is convertible to the variable's type. However, you might 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.

In implicit conversion no special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.

int num = 2147483647;

long bigNum = num;

Derived d = new Derived();

Base b = d; // Always OK.

1. **What is explicit conversion of types?**

 Explicit conversions require a cast operator. 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.

double x = 1234.7;

int a;

// Cast double to int.

a = (int)x;

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;

1. **What is difference between float, double and decimal?**

float and double are [floating *binary* point types](http://csharpindepth.com/Articles/General/FloatingPoint.aspx). In other words, they represent a number like this:

10001.10010110011

The binary number and the location of the binary point are both encoded within the value.

decimal is a [floating *decimal* point type](http://csharpindepth.com/Articles/General/Decimal.aspx). In other words, they represent a number like this:

12345.65789

Again, the number and the location of the *decimal* point are both encoded within the value – that's what makes decimal still a floating point type instead of a fixed point type.

*(probably no one will ever ask you this question)*

1. **Difference between passing a parameter by value and reference?**

4 cases:

* Passing value type by value: Value is copied in stack and changes made inside called function are not reflected in calling function.
* Passing value type by reference: Reference of value (memory location in stack is passed) to calling function, any changes made in function are actually being done on the original value hence they are reflected in calling function.
* Passing Reference type by value Value is copied in stack, as value of reference type is actually a reference to heap, so effectively a copy of reference in recreated in stack, pointing to original value in heap. Hence both are pointing to same object. Therefore any change made will be reflected everywhere.
* Passing Reference type by reference. Reference of value (which is again a reference to heap) is passed , again same object is pointed, but this time it is using actual original reference. So any changes will be made on same object. BUT because you are using original reference and if you perform any operation that result in change of reference, you changes will be limited to called function. Like if you have array (array is immutable) and in called function even if you pass array by reference, but if you re allocate it memory, any changes will be limited to called function.

1. **What does ref keyword does?**

Used to pass by reference

1. **Why is out parameter used and how it is different from ref parameter?**

A ref parameter can be assumed to be initially assigned; an out parameter cannot.

An out parameter must be assigned before control leaves the method normally. That is, this is illegal:

void foo(out int x) { return; }

x must be assigned before the return.

Behind the scenes, ref and out have the same implementation; the only difference is how the compiler tracks whether variables are assigned or not.

1. **What does params keyword do in passing parameters?**

By using the params keyword, you can specify a [method parameter](http://msdn.microsoft.com/en-us/library/8f1hz171.aspx) that takes a variable number of arguments. You can send a comma-separated list of arguments of the type specified in the parameter declaration or an array of arguments of the specified type. You also can send no arguments. If you send no arguments, the length of the params list is zero.

No additional parameters are permitted after the params keyword in a method declaration, and only one params keyword is permitted in a method declaration.

public int addTwoEach(params int[] args)

{

return args.Sum() + 2 \* args.Length;

}

1. **What is var keyword in C#?**

Beginning in Visual C# 3.0, variables that are declared at method scope can have an implicit type var. An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type.  Following are equivalent

var i = 10; // implicitly typed

int i = 10; //explicitly typed

This means you need not remember actual type of thing, based on value compiler with infer it, this is very usefull in Linq and let say you want to store one item of listbox, you need not remember the type of item of listbox just use var.

1. **What are namespaces and why they are used?**

The namespace keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.

* To structure the project into meaningful pieces
* To distinguish classes with the same name

That's basically it. Namespaces provide structure larger than just that of the project, since namespaces may span projects and assemblies. P*rimary* purpose of namespaces is to add structure to libraries **so that it becomes easier to find stuff you need and avoid stuff you do not need**.

1. **What are partial classes?**

It is possible to split the definition of a [class](http://msdn.microsoft.com/en-us/library/0b0thckt.aspx) or a [struct](http://msdn.microsoft.com/en-us/library/ah19swz4.aspx), an [interface](http://msdn.microsoft.com/en-us/library/87d83y5b.aspx) or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled. This is done if project is big and you need multiple people to work on same class. Even Visual studio uses this to generate classes like form1.cs for us to write code and designer.cs to contain auto generated code.

1. **How to manage equally of objects?**

It is sometimes necessary to compare two values for equality. In some cases, you are testing for value equality, also known as equivalence, which means that the values that are contained by the two variables are equal. In other cases, you have to determine whether two variables refer to the same underlying object in memory. This type of equality is called reference equality, or identity.

**Reference equality**

System.Object a = new System.Object();

System.Object b = a;

System.Object.ReferenceEquals(a, b); //returns true

**Value equality** is for value type and equal symbol, and is pretty simple. Value equality for reference type is bit complex. You have two student object how to know if they are equal? In fact what does equal means in this context? We override equal method (from system.object) in our class. For more detail please look at (<http://msdn.microsoft.com/en-us/library/ms173147(v=vs.80).aspx)> and if you don’t understand it let me know I will create a notes for it.

# Struct

1. **What is struct, how it is different from Class?**

Think of them as value type classes, these are value type and encapsulate a small set of variable (bigger struct are not advised) Structs can also contain  [constructors](http://msdn.microsoft.com/en-us/library/ace5hbzh.aspx),  [constants](http://msdn.microsoft.com/en-us/library/ms173119.aspx),  [fields](http://msdn.microsoft.com/en-us/library/ms173118.aspx),  [methods](http://msdn.microsoft.com/en-us/library/ms173114.aspx),  [properties](http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx),  [indexers](http://msdn.microsoft.com/en-us/library/6x16t2tx.aspx),  [operators](http://msdn.microsoft.com/en-us/library/ms173145.aspx),  [events](http://msdn.microsoft.com/en-us/library/awbftdfh.aspx), and [nested types](http://msdn.microsoft.com/en-us/library/ms173120.aspx), although if several such members are required, you should consider making your type a class instead.

public struct Book

{

public decimal price;

public string title;

public string author;

}

1. **Can we have properties, functions in Struct?**

We can have contain  [constructors](http://msdn.microsoft.com/en-us/library/ace5hbzh.aspx),  [constants](http://msdn.microsoft.com/en-us/library/ms173119.aspx),  [fields](http://msdn.microsoft.com/en-us/library/ms173118.aspx),  [methods](http://msdn.microsoft.com/en-us/library/ms173114.aspx),  [properties](http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx),  [indexers](http://msdn.microsoft.com/en-us/library/6x16t2tx.aspx),  [operators](http://msdn.microsoft.com/en-us/library/ms173145.aspx),  [events](http://msdn.microsoft.com/en-us/library/awbftdfh.aspx), and [nested types](http://msdn.microsoft.com/en-us/library/ms173120.aspx) in struct, if you are going to use most if it, please prefer class, as struc is stored in stack and we have limited space there.

1. **Can struct be inherited?**

First it is true you cannot from struct and struct cannot inherit from anything. However if we talk of internal all strut inherit from System.Valuetype, but still you cannot inherit from struct. Why?

Being a "value type" means that the entire struct - all of it's contents - are (usually) stored wherever you have a variable or member of that type. As a local variable or function parameter, that means on the stack. For member variables, that means stored entirely as part of the object.

As a (primary) example of why inheritance is a problem, consider how storage is affected at a low level if you allowed structs to have subtypes with more members. Anything storing that struct type would take up a variable amount of memory based on which subtype it ended up containing, which would be an allocation nightmare. An object of a given class would no longer have a constant, known size at compile time and the same would be true for stack frames of any method call. This does not happen for objects, which have storage allocated on the heap and instead have constant-sized references to that storage on the stack or inside other objects.

# Enum

1. **What is enumeration and what is use?**

Enums are used if want values to be from predefined set,

internal enum Color

{

White, // Assigned a value of 0

Red, // Assigned a value of 1

Green, // Assigned a value of 2

Blue, // Assigned a value of 3

Orange // Assigned a value of 4

}

* Enumerated types make the program much easier to write, read, and maintain. With enumerated types, the symbolic name is used throughout the code, and the programmer doesn’t have to mentally map the meaning of each hard-coded value (for example, white is 0 or vice versa). Also, should a symbol’s numeric value change, the code can simply be recompiled without requiring any changes to the source code. In addition, documentation tools and other utilities, such as a debugger, can show meaningful symbolic names to the programmer.
* Enumerated types are strongly typed. For example, the compiler will report an error if I attempt to pass Color.Orange as a value to a method requiring a Fruit enumerated type as a parameter. Or when I use Color. Pink, because Color does not have Pink value.

1. **How enumeration is different from constants?**

Use enums when you want to define a range of values that something can be. Constants should be for a single value, like PI.

1. **How to parse enum from integer and viceversa?**

int i = (int)Color.White; // I is 0

Color d = (Color)i; // d is white

1. **What are flags in enum?**

The flags attribute should be used whenever the enumerable represents a combination of values, rather than a single value. Such collections are usually manipulated using bitwise operators, for example:

[Flags]

public enum Direction

{

North = 1,

South = 2,

East = 4,

West = 8

}

To work with combined enum values, you use bitwise operators, such as | and &. These operate on the underlying integral values

Direction d = Direction.North | Direction.East;

int i = (int)d; // I will be 5

# Access modifiers

1. **What are access modifiers?**

Access modifiers are keywords used to specify the declared accessibility of a member or a type

**public** : Access is not restricted.

**protected** : Access is limited to the containing class or types derived from the containing class.

**Internal** : Access is limited to the current assembly.

[**protected** internal](http://msdn.microsoft.com/en-us/library/ms173121.aspx): Access is limited to the current assembly or types derived from the containing class.

**private** : Access is limited to the containing type.

Access modifiers are not allowed on namespaces. Namespaces have no access restrictions.

1. **What are default access modifier for class, function, property and variables**

Depending on the context in which a member declaration takes place, only certain declared accessibilities are permitted. If no access modifier is specified in a member declaration, a default accessibility is used.

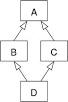
Top-level types, which are not nested into other types, can only have **internal** or **public** accessibility. The default accessibility for these types is **internal**.

|  |  |  |
| --- | --- | --- |
| Members of | Default member accessibility | Allowed declared accessibility of the member |
| enum | public | None |
| class | private | public  protected  internal  private  protected internal |
| interface | public | None |
| struct | private | public  internal  private |

# Inheritance

1. **What is inheritance?**

Inheritance describes the ability to create new classes based on an existing class. The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class. A derived class can have only one direct base class.

****

1. **Why C #does not support multiple inheritances?**

The diamond problem is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If a method in D calls a method defined in A (and does not override the method), and B and C have overridden that method differently, then from which class does it inherit: B, or C? It is called the "diamond problem" because of the shape of the class inheritance diagram in this situation.

1. **What is order of constructor calling in inheritance?**

Base class objects are always constructed before any deriving class. Thus the constructor for the base class is executed before the constructor of the derived class.

1. **What does “base” refer to?**

You can access base class members in a subclass even when those base members are overridden in the superclass using the [base](http://msdn.microsoft.com/en-us/library/hfw7t1ce(v=vs.90).aspx) keyword. For instance, you can create a derived class which contains a method with the same signature as in the base class. If you prefaced that method with the [new](http://msdn.microsoft.com/en-us/library/435f1dw2(v=vs.90).aspx) keyword, you indicate that this is an all-new method belonging to the derived class. You could still provide a method for accessing the original method in the base class with the base keyword.

1. **What does virtual keyword do?**

The virtual keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. The implementation of a virtual member can be changed by an [overriding member](http://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) in a derived class. When a virtual method is invoked, the run-time type of the object is checked for an overriding member (this u will understand after example, but keep this in mind). By default, methods are non-virtual. You cannot override a non-virtual method, but you can hide them.

public class Employee

{

public virtual int CalculatePay()

{

return salary;

}

}

1. **What does override do?**

The override modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

public class Employee

{

public virtual int CalculatePay()

{

return salary;

}

}

public class SalesEmployee : Employee

{

public override int CalculatePay()

{

return salary + salesbonus;

}

}

* The overridden base method must have the same signature as the override method.
* The overridden base method must be virtual, abstract, or override.
* You cannot override a non-virtual or static method.
* An override declaration cannot change the accessibility of the virtual method. Both the override method and the virtual method must have the same [access level modifier](http://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx).

1. **What is use of abstract class?**

The [abstract](http://msdn.microsoft.com/en-us/library/sf985hc5.aspx) keyword enables you to create classes that must be implemented in a derived class. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.

1. **How to intentionality hide inherited member?**

Using new keyword

1. **What does new keyword do in inheritance?**

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

public class BaseC

{

public int x;

public void Invoke() { }

}

public class DerivedC : BaseC

{

new public void Invoke() { }

}

1. **What does Sealed do?**

A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes prevent derivation.

public sealed class D

{

// Class members here.

}

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](http://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) keyword in the class member

public class D : C

{

public sealed override void DoWork() { }

}

1. **Conversion in inheritance?**

Child class to Base class is implicit, Base to child requires explicit syntax, and however explicit syntax does not guarantee conversion success. (for why? Check out Inheritance\_P2.docx note)

1. **What is difference between “as” and “is” operator?**

IS: Checks if an object is compatible with a given type. For example:

if (E is Manager)

{

Manager M5 = (Manager)E;

}

An is expression evaluates to true if the provided expression is non-null, and the provided object can be cast to the provided type without causing an exception to be thrown.

AS: You can use the as operator to perform certain types of conversions between compatible reference types or [nullable types](http://msdn.microsoft.com/en-us/library/1t3y8s4s.aspx). The as operator is like a cast operation. However, if the conversion isn't possible,  as  returns  null  instead of raising an exception.

Manager M5 = E as Manager; //exception can be raised

if (M5 != null)

{

// USE M5 here

}

1. **How function is selected for execution in inheritance?**

Functions are first looked in type and its base class.

# Interface

1. **What is Interface?**

Interfaces are contracts between components, if a class has implemented some interface, we are sure that functions, properties of interface are available in that class. Interfaces in C # provide a way to achieve runtime polymorphism. Using interfaces we can invoke functions from different classes through the same Interface reference, whereas using virtual functions we can invoke functions from different classes in the same inheritance hierarchy through the same reference.  An interface contains only the signatures of  [methods](http://msdn.microsoft.com/en-us/library/ms173114.aspx),  [properties](http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx),  [events](http://msdn.microsoft.com/en-us/library/awbftdfh.aspx) or [indexers](http://msdn.microsoft.com/en-us/library/6x16t2tx.aspx). A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition

public interface IEnumerator //Interface

{

bool MoveNext();

object Current { get; }

void Reset();

}

1. **What is difference between abstract class and interface?**

|  |  |  |
| --- | --- | --- |
| Feature | Interface | Abstract class |
| Multiple inheritance | A class may inherit several interfaces. | A class may inherit only one abstract class. |
| Default implementation | An interface cannot provide any code, just the signature. | An abstract class can provide complete, default code and/or just the details that have to be overridden. |
| Access Modfiers | An interface cannot have access modifiers for the subs, functions, properties etc everything is assumed as public | An abstract class can contain access modifiers for the subs, functions, properties |
| Core VS Peripheral | Interfaces are used to define the peripheral abilities of a class. In other words both Human and Vehicle can inherit from a IMovable interface. | An abstract class defines the core identity of a class and there it is used for objects of the same type. |
| Homogeneity | If various implementations only share method signatures then it is better to use Interfaces. | If various implementations are of the same kind and use common behavior or status then abstract class is better to use. |
| Speed | Requires more time to find the actual method in the corresponding classes. | Fast |
| Adding functionality (Versioning) | If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method. | If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly. |
| Fields and Constants | No fields can be defined in interfaces | An abstract class can have fields and constrants defined |

**CAN-DO and IS-A** relationship is also define the difference between Interface and abstract class

* If you anticipate creating multiple versions of your component, create an abstract class. Abstract classes provide a simple and easy way to version your components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, you must create a whole new interface.
* If the functionality you are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes.
* If you are designing small, concise bits of functionality, use interfaces. If you are designing large functional units, use an abstract class.
* If you want to provide common, implemented functionality among all implementations of your component, use an abstract class. Abstract classes allow you to partially implement your class, whereas interfaces contain no implementation for any members.

1. **How interface is implemented?**

internal class Countdown : IEnumerator

{

int count = 11;

public bool MoveNext() { return count-- > 0; }

public object Current { get { return count; } }

public void Reset() { throw new NotSupportedException(); }

public void Print()

{

IEnumerator e = new Countdown();

while (e.MoveNext())

{

Console.Write(e.Current); // 109876543210

}

}

}

1. **What is explicit implementation of interface and how it is different from implicit implementation?**

Implicit is when you define your interface via a member on your class. Explicit is when you define methods within your class on the interface. When a member is explicitly implemented, it cannot be accessed through a class instance, but only through an instance of the interface. Explicit interface implementation also allows the programmer to inherit two interfaces that share the same member names and give each interface member a separate implementation.

IList.CopyTo would be implicitly implemented as:

public void CopyTo(Array array, int index)

{

throw new NotImplementedException();

}

Explicit

void ICollection.CopyTo(Array array, int index)

{

throw new NotImplementedException();

}

As said above The difference being that implicitly is accessible throuh your class you created when it is cast as that class as well as when its cast as the interface. Explicit implentation allows it to only be accessible when cast as the interface itself.

myclass.CopyTo //invalid with explicit

((IList)myClass).CopyTo //valid with explicit.

1. **How to implement interface elements virtually?**

An implicitly implemented interface member is, by default, sealed. It must be marked virtual or abstract in the base class in order to be overridden

public interface IUndoable { void Undo(); }

public class TextBox : IUndoable

{

public virtual void Undo()

{

Console.WriteLine("TextBox.Undo");

}

}

public class RichTextBox : TextBox

{

public override void Undo()

{

Console.WriteLine("RichTextBox.Undo");

}

}

RichTextBox r = new RichTextBox();

r.Undo(); // RichTextBox.Undo

((IUndoable)r).Undo(); // RichTextBox.Undo

((TextBox)r).Undo(); // RichTextBox.Undo

1. **How to re-implement Interface in subclass?**

A subclass can reimplement any interface member already implemented by a base class. It works whether or not the member is virtual in the base class. It also works whether a member is implemented implicitly or explicitly

In above example if I change

public class RichTextBox : TextBox, IUndoable

{

public new void Undo() { Console.WriteLine("RichTextBox.Undo"); }

}

RichTextBox r = new RichTextBox();

r.Undo(); // RichTextBox.Undo Case 1

((IUndoable)r).Undo(); // RichTextBox.Undo Case 2

((TextBox)r).Undo(); // TextBox.Undo Case 3

Case 23 result changes

# Exception Handling

1. **What is exception handling? Why do we need it?**

**Exception** represents errors that occur during application execution. Exceptions can be generated by the common language runtime (CLR), by the .NET Framework or any third-party libraries, or by application code. You can also explicitly create exception Exceptions are created by using the throw keyword.

**Exception handling**: The C# language's exception handling features help you deal with any unexpected or exceptional situations that occur when a program is running. Exception handling uses the try, catch, and finally keywords to try actions that may not succeed, to handle failures when you decide that it is reasonable to do so, and to clean up resources afterward.

1. **What are constructing that help in exception handling? Or what does try, catch and finally do?**

**The try Block:** A try block contains code that might potentially throw an exception. The exception-recovery code should be placed in one or more catch blocks. You create one catch block for each kind of exception that your application can safely recover from. A try block must be associated with at least one catch or finally block; it makes no sense to have a try block that stands by itself, and C# will prevent you from doing this.

**The catch Block**: A catch block contains code to execute in response to an exception. A try block can have zero or more catch blocks associated with it. If the code in a try block doesn’t cause an exception to be thrown, the CLR will never execute the code contained within any of its catch blocks. The thread will simply skip over all of the catch blocks and execute the code in the finally block (if one exists). After the code in the finally block executes, execution continues with the statement following the finally block.

**A finally block** contains code that’s guaranteed to execute, the code in a finally block performs the cleanup operations required by actions taken in the try block. For example, if you open a file in a try block, you’d put the code to close the file in a finally block

1. **Can we have finally without catch?**

Yes try and finally can exists without catch

1. **Can we have multiple catch in a try? What will order of execution? Can we execute multiple catch?**

Multiple catch blocks with different exception filters can be chained together. The catch blocks are evaluated from top to bottom in your code, but only one catch block is executed for each exception that is thrown. The first catch block that specifies the exact type or a base class of the thrown exception is executed. If no catch block specifies a matching exception filter, a catch block that does not have a filter is selected, if one is present in the statement. It is important to position catch blocks with the most specific (that is, the most derived) exception types first.

1. **What will happen if only try and finally are used?**

Code in try will be executed, if exception occurs in try then no other line will be executed in try block, finally will start executing. Even if no exception is raised, after executing all try, finally will be executed.

1. **What will happen if there is exception in catch block?**

An exception thrown in a catch block will behave the same as an exception thrown without it - it will go up the stack until it is caught in a higher level catch block, if one exists.

1. **What does using keyword do? What statements using is resolved to?**

Provides a convenient syntax that ensures the correct use of [IDisposable](http://msdn.microsoft.com/en-us/library/system.idisposable.aspx) objects. The using statement ensures that [Dispose](http://msdn.microsoft.com/en-us/library/system.idisposable.dispose.aspx) is called even if an exception occurs while you are calling methods on the object. You can achieve the same result by putting the object inside a try block and then calling [Dispose](http://msdn.microsoft.com/en-us/library/system.idisposable.dispose.aspx) in a finally block; in fact, this is how the using statement is translated by the compiler.

using (Font font1 = new Font("Arial", 10.0f))

{

byte charset = font1.GdiCharSet;

}

Resolves to

Font font1 = new Font("Arial", 10.0f);

try

{

byte charset = font1.GdiCharSet;

}

finally

{

if (font1 != null)

((IDisposable)font1).Dispose();

}

1. **How to explicitly throw exception?**

Programs can throw a predefined exception class in the [System](http://msdn.microsoft.com/en-us/library/system.aspx) namespace (except where previously noted), or create their own exception classes by deriving from[Exception](http://msdn.microsoft.com/en-us/library/system.exception.aspx). The throw statement is used to throw exception

class MyException : System.Exception {}

// ...

Throw new MyException();

The following list identifies practices to avoid when throwing exceptions:

* Exceptions should not be used to change the flow of a program as part of ordinary execution. Exceptions should only be used to report and handle error conditions.
* Exceptions should not be returned as a return value or parameter instead of being thrown.
* Do not throw [System.Exception](http://msdn.microsoft.com/en-us/library/system.exception.aspx), [System.SystemException](http://msdn.microsoft.com/en-us/library/system.systemexception.aspx), [System.NullReferenceException](http://msdn.microsoft.com/en-us/library/system.nullreferenceexception.aspx), or [System.IndexOutOfRangeException](http://msdn.microsoft.com/en-us/library/system.indexoutofrangeexception.aspx) intentionally from your own source code.
* Do not create exceptions that can be thrown in debug mode but not release mode. To identify run-time errors during the development phase, use Debug Assert instead.

1. **What is difference between throw and throw ex?**

throw ex resets the stack trace (so your errors would appear to originate from HandleException)

throw doesn't - the original offender would be preserved.

1. **What is stack trace?**

The execution stack keeps track of all the methods that are in execution at a given instant. A trace of the method calls is called a stack trace. The stack trace listing provides a way to follow the call stack to the line number in the method where the exception occurs.

# Generics

1. **What is Generics? Why it is used?**

 Generics allow you to define type-safe data structures, without committing to actual data types. you get to reuse data processing algorithms without duplicating type-specific code

public class GenericList<T>

{

void Add(T input) { }

}

class TestGenericList

{

private class ExampleClass { }

static void Main()

{

// Declare a list of type int.

GenericList<int> list1 = new GenericList<int>();

// Declare a list of type string.

GenericList<string> list2 = new GenericList<string>();

// Declare a list of type ExampleClass.

GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();

}

}

Please remember that generics are always resolved at compile time.

1. **What is generic Type and generic method?**

In a generic type or method definition, a type parameters is a placeholder for a specific type that a client specifies when they instantiate a variable of the generic type. A generic class, such as GenericList<T>

A generic method is a method that is declared with type parameters, as follows:

static void Swap<T>(ref T lhs, ref T rhs)

{

T temp;

temp = lhs;

lhs = rhs;

rhs = temp;

}

1. **How to limit types for generics?**

When you define a generic class, you can apply restrictions to the kinds of types that client code can use for type arguments when it instantiates your class. If client code tries to instantiate your class by using a type that is not allowed by a constraint, the result is a compile-time error.

|  |  |
| --- | --- |
| Constraint | Description |
| where T: struct | The type argument must be a value type. Any value type except [Nullable](http://msdn.microsoft.com/en-us/library/system.nullable.aspx) can be specified. See [Using Nullable Types (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/2cf62fcy.aspx)for more information. |
| where T : class | The type argument must be a reference type; this applies also to any class, interface, delegate, or array type. |
| where T : new() | The type argument must have a public parameter less constructor. When used together with other constraints, the new () constraint must be specified last. |
| where T : <base class name> | The type argument must be or derive from the specified base class. |
| where T : <interface name> | The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic. |
| where T : U | The type argument supplied for T must be or derive from the argument supplied for U. |

1. **What is nullable type?**

Nullable types are instances of the [System.Nullable<T>](http://msdn.microsoft.com/en-us/library/b3h38hb0.aspx) struct. A nullable type can represent the correct range of values for its underlying value type, plus an additional nullvalue. Remember that value type do not support null.

int? num = null;

* Nullable types represent value-type variables that can be assigned the value of null. You cannot create a nullable type based on a reference type. (Reference types already support the null value.)
* The syntax **T?** is shorthand for [Nullable<T>](http://msdn.microsoft.com/en-us/library/b3h38hb0.aspx), where **T** is a value type. The two forms are interchangeable.
* Assign a value to a nullable type just as you would for an ordinary value type, for example int? x = 10; or double? d = 4.108. A nullable type can also be assigned the value null: int? x = null.
* Use the [HasValue](http://msdn.microsoft.com/en-us/library/sksw8094.aspx) and [Value](http://msdn.microsoft.com/en-us/library/ydkbatt6.aspx) read-only properties to test for null and retrieve the value

1. **How does generic resolves need of boxing and unboxing?**

Earlier if we have to use code for different type, we have to convert it to object type. Let’s say you need a collection to work with many type, we created collection like ArrayList that takes object type as parameter, now if we use it with int or bool, they will be converted to object, ie boxing, and we will take objects from Arraylist then we will unbox them to actual types. Using generic we reuse same code but with specific type, and generic is resolved at compile time, hence no need of boxing and unboxing.

# Collections

1. **What is Collection? How it is different from array?**

For many applications, we want to create and manage groups of related objects. There are two ways to group objects: by creating arrays of objects, and by creating collections of objects. Arrays are most useful for creating and working with a fixed number of strongly-typed objects. Collections provide a more flexible way, unlike arrays, the group of objects we work with can grow and shrink dynamically. For some collections, we can assign a key to any object that we put into the collection so that we can quickly retrieve the object by using the key.

1. **What IEnumrabale and IEnumarator interfaces?**

IEnumarator supports iteration over collections i.e. getting items one by one. We commonly use foreach for this which hides the complexity of these interfaces. IEnumarator has a property current which points to current item (in array or collection) and has two methods MoveNext and Reset to move to next item and set back current to before first item respectively. When we do foreach, there methods and Current property is used to get us items one by one.

IEnumrabale is interface that exposes a method GetEnumarator to return an enumerator (which in turn iterates over collection).

1. **What is difference between Array, ArrayList?**

Arrays are of fixed length, and all element of array are of same type unless array of object is created. ArrayList can grow and shrink dynamically. Arrays can be multidimensional but ArrayLists are single dimensional. ArrayLists are not strongly typed; every Insertion or Retrial will need a cast. Insertion needs cast to object type and retrieval required cast to original type. Behind the scene ArrayList also uses array to store data, they just increase automatically when get full, so there is also a performance hit to expand the size of the internal array when it hits its capacity.

1. **What is difference between ArrayList and List<T>?**

List<T> is a generic class. It supports storing values of a specific type without casting to or from object (which would have incurred boxing/unboxing overhead when T is a value type in the ArrayList case). ArrayList simply stores object references. ArrayList belongs to the days that C# didn't have generics (before 2.0)

1. **What is dictionary?**

It is a generic Key value pair collection. The Dictionary<TKey, TValue> class provides a mapping from a set of keys to a set of values. Retrieving a value by using its key is very fast, Dictionary internally implements HashTable. A key cannot be null, but a value can be, if the value type *TValue* is a reference type.

1. **What is hash table? How does hash table work?**

Let's assume you want to fill up a library of books and you want to be able to easily find them again when you need them. So, you decide that if the person that wants to read a book knows the title of the book, with the aid of the librarian, should be able to go find the book easily and quickly.

So, how can you do that? Well, obviously you can keep some kind of list of where you put each book, but then you have the same problem as searching the library, you need to search the list. Granted, the list would be smaller, and easier to search, but still, you don't want to search sequentially from one end of the library (or list) to the other.

You want something that, with the title of the book, can give you the right spot at once .Instead of just starting to fill up the library from one end to the other. You take the title of the book, run it through a small computer program, which spits out a shelf number and a slot number on that shelf. This is where you place the book.

The beauty of this program is that later on, when a person comes back in to read the book, you feed the title through the program once more, and get back the same shelf number and slot number that you were originally given, and this is where the book is located. The program, is called a hash algorithm or hash computation, and usually works by taking the data fed into it (the title of the book in this case) and calculates a number from it .For simplicity, let's say that it just converts each letter and symbol into a number, and sums them all up. In reality it's a lot more complicated than that.

The beauty of such an algorithm is that if you feed the same input into it again and again, it will keep spitting out the same number each time. So that's basically how a hash table works. As we read above dictionary is also based on hash table, only difference is dictionary is generic. Hashing is way to generate The objects used as keys by a Hashtable are required to override the [Object.GetHashCode](http://msdn.microsoft.com/en-us/library/system.object.gethashcode(v=vs.110).aspx) method

Read <http://msdn.microsoft.com/en-us/library/system.collections.hashtable(v=vs.110).aspx>

1. **What is difference between collection and generic collection?**

Prior to 2.0 collection (ArrayList, Hashtables) store object, hence boxing and unboxing is invoked. Generic collections (List<T>, Dictionary <Tkey, TValue>) are based on principal of generics.

# Misc

1. **What is regular expression and what is its use?**

A regular expression (abbreviated regex or regexp) is a sequence of [characters](http://en.wikipedia.org/wiki/Character_(computing)) that forms a search pattern, mainly for use in [pattern matching](http://en.wikipedia.org/wiki/Pattern_matching) with [strings](http://en.wikipedia.org/wiki/String_(computer_science)), or [string matching](http://en.wikipedia.org/wiki/String_matching), i.e. "find and replace"-like operations. [System.Text.RegularExpressions.Regex](http://msdn.microsoft.com/en-IN/library/system.text.regularexpressions.regex.aspx) class can be used to search strings. We create a regular expression then use IsMatch method of Regex class to validate if a string matches the expression pattern.

"^\\d{3}-\\d{3}-\\d{4}$" is a regular expression that will look 3 digits then – then 3 digits then – then 4 digits(US phone format).

System.Text.RegularExpressions.Regex.IsMatch(s, sPattern);

Where sPattern is regular expression and s is string to check. Regular expression help up to validate complex and wide range of possible strings for a format. Like email ids.. Lot of lines of code is saved using regular expression, but creating one requires learning them.

# Delegates and Events

1. **What is delegate? What is it used for?**

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance. Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates.

public delegate int PerformCalculation(int x, int y);

* Delegates allow methods to be passed as parameters.
* Delegates can be used to define callback methods.
* Delegates can be chained together; for example, multiple methods can be called on a single event.

1. **Is delegate execution sync of async?**

As we know delegate can reference multiple functions (multicast) by default these functions are synchronously invoked.

public delegate void MyDelegate(string s);

void Example()

{

MyDelegate objMyDelegate = null;

objMyDelegate = new MyDelegate(MyTask);

IAsyncResult result = null;

result = objMyDelegate.BeginInvoke("TestString", new AsyncCallback(TaskCompleted), null);

}

public void MyTask(string s)

{

}

public void TaskCompleted(IAsyncResult R)

{

}

objMyDelegate.BeginInvoke("TestString", new AsyncCallback(TaskCompleted), null); Invoke delegate and registers a callback TaskCompleted, when MyTask is finished TaskComleted is called.

1. **What are multicast delegate?**

Ability to register multiple functions in delegate. The multicast delegate contains a list of the assigned delegates. When the multicast delegate is called, it invokes the delegates in the list, in order. Only delegates of the same type can be combined.

1. **What are func, Action and predicate delegates?**

The difference between Func and Action is simply whether you want the delegate to return a value (Func) or not (Action). Predicate delegate are special Func delegate that return Boolean. Please note these are just names for delegate, like VB.NET has SUB and function two names for method (sub does not return value, function does). There is no difference in execution of behind scene logic.

As delegates are commonly used hence .NET provides you many overrides (16 overrides!) of Func and Action delegate so you have to rarely create your own delegate.

<http://msdn.microsoft.com/en-us/library/018hxwa8(v=vs.110).aspx>

<http://msdn.microsoft.com/en-us/library/bb549151(v=vs.110).aspx>

1. **What event? What is use of event?**
2. **Create a sample event? What is standard event pattern?**
3. **How are event and delegates related?**

# Garbage Collector

1. **What is Garbage collector, how does it work?**
2. **What is Idisposiable, when it is used?**
3. **What is finalizer?**
4. **How is dispose different from finalizer?**

# New Features of C#

1. **What is anonymous type?**

Anonymous types are the new concept in C#3.0 which allows creating new types without defining them. Yani bina class declare kiye class ki tarah use kerna. Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

The following example shows an anonymous type that is initialized with two properties named Amount and Message.

var v = new { Amount = 108, Message = "Hello" };

This is used too much in LINQ.

1. **What is anonymous function?**

An anonymous function is an "inline" statement or expression that can be used wherever a delegate type is expected.

static void SomeFunction(string s)

{

Console.WriteLine(s);

}

delegate void TestDelegate(string s); *You know how to use delegates*

TestDelegate testDelA = new TestDelegate(SomeFunction);

*By Anonymous function you can do it like (no need of explicit function)*

TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

1. **What is extension function?**

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on, and the parameter is preceded by the [this](http://msdn.microsoft.com/en-us/library/dk1507sz.aspx) modifier.

public static class MyExtensions

{

public static int WordCount(this String str)

{

return str.Split(new char[] { ' ', '.', '?' },

StringSplitOptions.RemoveEmptyEntries).Length;

}

}

1. **What is lambda expression?**

Lambda expression is an inline delegate introduced with C # 3.0 languages. It’s a concise way to represent an anonymous method. It provides a syntax to create and invoke functions. Although Lambda expressions are simpler to use than anonymous methods, they do slightly differ on how they are implemented. Both anonymous methods and Lambda expressions allow you define the method implementation inline; however, an anonymous method explicitly requires you to define the parameter types and the return type for a method. Lambda expression uses the type inference feature of C# 3.0 which allows the compiler to infer the type of the variable based on the context.

**n => n % 2 == 1**

* **n** is the input parameter
* **n % 2 == 1** is the expression

You can read **n => n % 2 == 1** like: "input parameter named n goes to anonymous function which returns true if the input is odd".

# Serialization

1. **What is Serialization?**

When you create an object in a .Net framework application, you don't need to think about how the data is stored in memory. Because .Net framework takes care of that for you. However, if you want to store the contents of an object to a file, send an object to another process or transmit it across the network, you do have to think about how the object is represented because you will need to convert it to a different format. This conversion is called serialization. The reverse process is called deserialization.

1. **What are different types of serializations?**

Two main kind of serialization exists Binary and XML.

Binary

# Threading

1. **What is threading, why it is used?**

Threading is a way to create multiple threads for a process. It is done to achieve parallel execution, leverage multiple cores in machine. A thread is defined as the execution path of a program. Good use of threading optimizes CPU utilization and archives greater throughput.

1. **What is difference between thread and process?**

Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces. A *process*, in the simplest terms, is an executing program. One or more threads run in the context of the process. A *thread* is the basic unit to which the operating system allocates processor time.

1. **Create a thread?**

static void Main()

{

Thread t = new Thread(WriteY);

t.Start();

for (int i = 0; i < 1000; i++) Console.Write("x");

}

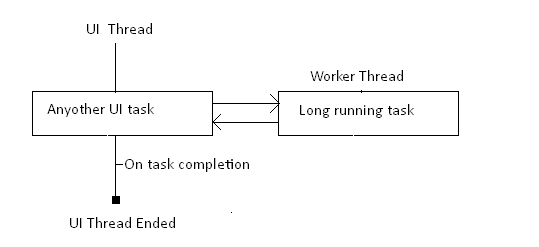
static void WriteY()

{

for (int i = 0; i < 1000; i++) Console.Write("y");

}

1. **What is background worker?**

Background Worker creates threads, mainly used with Windows application to do long running processing, so that UI does not freeze. It is creates thread from thread pool and reports its execution progress

1. Create a BackgroundWorker object.
2. Tell the BackgroundWorker object what task to run on the background thread (the DoWork function).
3. Tell it what function to run on the UI thread when the work is complete the RunWorkerCompleted function
4. **How to wait for a thread to complete?**

ThreadName.Join()

1. **What is thread synchronization?**

Multiple threads can execute asynchronously. The asynchronous nature of threads means that access to resources such as file handles, network connections, and memory must be coordinated. Otherwise, two or more threads could access the same resource at the same time, each unaware of the other's actions. The result is unpredictable data corruption. Steps taken to ensure that we don’t let such condition arise in our code is called thread synchronization.

1. **What is lock (monitor)?**

The lock (C#) and can be used to ensure that a block of code runs to completion without interruption by other threads. In more technical terms the lock keyword marks a statement block as a critical section by obtaining the mutual-exclusion lock for a given object, executing a statement, and then releasing the lock.

In simpler terms, you want some statements to be executed by only one thread at a time, you use lock.

Any statement inside lock will be executed by a thread until then no other thread will start executing those statements

class Account

{

decimal balance;

private Object thisLock = new Object();

public void Withdraw(decimal amount)

{

lock (thisLock)

{

if (amount > balance)

{

throw new Exception("Insufficient funds");

}

balance -= amount;

}

}

}

Check this code, Let’s assume your balance is 1000 and you withdraw 700. This will be fine if you have single threaded application, as every time it checks if amount is less than balance. But what if we have 2 (or more) thread executing Withdraw method. There me be situation when

**balance -= amount;** is executed by one thread, other thread is checking if (amount > balance)

Second thread may also execute **balance -= amount;** and thus balance will be -400 which is wrong.

To stop this we put a lock around these statement, now only thread can enter in this area at a time, once it will get out of this section, only then other thread will execute, hence our result will never be negative.

**Please note object used in Lock must be reference type only.**

1. **What is threadpool?**

Lot of application need to create thread, lot of their time goes in creating and disposing them.  The thread pool enables you to use threads more efficiently by providing your application with a pool of worker threads that are managed by the system. The threads in the managed thread pool are background threads.

1. **What is mutex?**

Lock is good for single process, i.e. you can synchronize multiple thread but within a process. Mutex expands this concept across process boundary. 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. WaitOne is method used to wait for mutex. ReleaseMutex is used to exit from mutex. //Try running two instances of the executable for this program. Only one of them will be allowedto execute at a time.)

class Program

{

static void Main(string[] args)

{

using (var mutex = new Mutex(false, "WordCountProgram"))

{

// Wait a few seconds if contended, in case another instance

// of the program is still in the process of shutting down.

if (!mutex.WaitOne(TimeSpan.FromSeconds(3), false))

{

Console.WriteLine("Another app instance is running. Bye!");

return;

}

RunProgram();

}

}

private static void RunProgram()

{

Console.WriteLine("Running. Press Enter to exit");

Console.ReadLine();

}

}

1. **What is semaphore?**

Semaphore allows multiple but fixed number of threads in critical section. Think of semaphores as bouncers at a nightclub. There are a dedicated number of people that are allowed in the club at once. If the club is full no one is allowed to enter, but as soon as one person leaves another person might enter.

using System;

using System.Collections.Generic;

using System.Text;

using System.Threading;

namespace TheNightclub

{

public class Program

{

public static Semaphore Bouncer { get; set; }

public static void Main(string[] args)

{

// Create the semaphore with 3 slots, where 3 are available.

Bouncer = new Semaphore(3, 3);

// Open the nightclub.

OpenNightclub();

}

public static void OpenNightclub()

{

for (int i = 1; i <= 50; i++)

{

// Let each guest enter on an own thread.

Thread thread = new Thread(new ParameterizedThreadStart(Guest));

thread.Start(i);

}

}

public static void Guest(object args)

{

// Wait to enter the nightclub (a semaphore to be released).

Console.WriteLine("Guest {0} is waiting to entering nightclub.", args);

Bouncer.WaitOne();

// Do some dancing.

Console.WriteLine("Guest {0} is doing some dancing.", args);

Thread.Sleep(500);

// Let one guest out (release one semaphore).

Console.WriteLine("Guest {0} is leaving the nightclub.", args);

Bouncer.Release(1);

}

}

}

# 

# Advanced

1. **What is dll, exe?**

A DLL (.dll) file contains a library of functions and other information that can be accessed by a Windows program. When a program is launched, links to the necessary .dll files are created. If a static link is created, the .dll files will be in use as long as the program is active. If a dynamic link is created, the .dll files will only be used when needed. DLL files can also be used by more than one program. In fact, they can even be used by multiple programs at the same time.

An executable is a [file](http://searchexchange.techtarget.com/definition/file) that is capable of being executed or run as a program in the Windows computer.

EXE:

1. It's a executable file
2. When loading an executable, no export is called, but only the module entry point.
3. When a system launches new executable, a new process is created
4. The entry thread is called in context of main thread of that process.

DLL:

1. It's a Dynamic Link Library
2. There are multiple exported symbols.
3. The system loads a DLL into the context of an existing process.
4. **What is assembly?**

An assembly is a "unit of deployment" for .NET, almost always a .exe or .dll.  Assembly is the smallest unit of deployment of a .net application. When you compile an application, the MSIL code created is stored in an assembly. Assemblies include both executable application files that you can run directly from Windows without the need for any other programs (these have a .exe file extension), and libraries (which have a .dll extension) for use by other applications.

In addition to containing MSIL, assemblies also include meta information (that is, information about the information contained in the assembly, also known as metadata) and optional resources (additional data used by the MSIL, such as sound files and pictures).

**Private Assembly:** The dll or exe which is sole property of one application only. It is generally stored in application root folder

**Public/Shared assembly:** It is a dll which can be used by multiple applications at a time. A shared assembly is stored in **GAC** i.e [Global Assembly Cache](http://msdn.microsoft.com/en-us/library/yf1d93sz).

**Satellite Assembly:** A satellite assembly is a compiled library (DLL) that contains (“localizable”) resources such as strings, bitmaps, etc. You are likely to use them when creating a multilingual (UI) application. Satellite assemblies provide you with the capability of designing and deploying your solution to multiple cultures, rather than hard coding strings, bitmaps, etc., into your main application. Satellite assemblies are used to deploy applications in multiple cultures (not languages), with 1 satellite assembly per culture

1. **What is GAC? How does it resolve dll hell?**

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.

Starting with the .NET Framework 4, the default location for the global assembly cache is **%windir%\Microsoft.NET\assembly**. In earlier versions of the .NET Framework, the default location is **%windir%\assembly**.

**DLL Hell:**

DLL Hell refers to the set of problems caused when multiple applications attempt to share a common component like a dynamic-link library (DLL) or a Component Object Model (COM) class. In the most typical case, one application will install a new version of the shared component that is not backward compatible with the version already on the machine. Although the application that has just been installed works fine, existing applications that depended on a previous version of the shared component might no longer work. The reason for these issues is that version information about the different components of an application aren't recorded or enforced by the system. Also, changes made to the system on behalf of one application will typically affect all applications on the machine—building an application today that is completely isolated from changes is not easy.

In GAC Unlike in COM, there is no need for the assembly in GAC to be registered before its use. Each assembly is accessed globally without any conflict by identifying its name, version, architecture, culture and public key.

<http://msdn.microsoft.com/en-us/library/ms973843.aspx>

1. **How an assembly reference is resolved?**

By default, the runtime attempts to bind with the exact version of an assembly that the application was built with. This default behavior can be overridden by configuration file settings.

The runtime uses the following steps to resolve an assembly reference:

* 1. [Determines the correct assembly version](http://msdn.microsoft.com/en-us/library/8f6988ab(v=vs.110).aspx) by examining applicable configuration files, including the application configuration file, publisher policy file, and machine configuration file.
  2. [Checks whether the assembly name has been bound to before](http://msdn.microsoft.com/en-us/library/aa98tba8(v=vs.110).aspx) and, if so, uses the previously loaded assembly. If a previous request to load the assembly failed, the request is failed immediately without attempting to load the assembly.
  3. [Checks the global assembly cache](http://msdn.microsoft.com/en-us/library/4a9t8a9a(v=vs.110).aspx). If the assembly is found there, the runtime uses this assembly.
  4. If the assembly is not found in the GAC then the CLR will search for it in the "probing path". Which by default is only the directory that contains the EXE. It only looks for a match on the assembly name and will stop searching on the first match.It then checks the [AssemblyVersion] number. If it doesn't match you'll get an exception, it won't keep looking for another assembly with the same name.

1. **What is application domain?**

Application domains provide a flexible and secure method of isolating running applications. They share many of the same characteristics of a process, e.g. they have their own copies of statics, assemblies and so forth, but they are contained within a single process. From the operating system's point of view a process is just a process no matter how many AppDomains it may contain.Unlike a process however, an AppDomain does not have any threads unless you explicitly create them. A thread can run code in any AppDomain.AppDomains are part of the same process and thus actually share the same managed heap.

Application domains are usually created and manipulated by run-time hosts. Everything you usually think of as "per program" (static variables etc) is actually per-AppDomain. Application domains aid security, separating applications from each other and each other's data. A single process can run several application domains, with the same level of isolation that would exist in separate processes. Running multiple applications within single process increases server scalability.

1. **What is remoting?**

.NET remoting enables you to build widely distributed applications easily, whether the application components are all on one computer or spread out across the entire world. You can build client applications that use objects in other processes on the same computer or on any other computer that is reachable over its network. You can also use .NET remoting to communicate with other application domains in the same process.

There are 2 common types of protocols used in .NET Remoting: tcp for binary stream and http for SOAP stream. (Second one will remind you of WCF)

1. **What is COM?**

COM is a platform-independent, distributed, object-oriented system for creating binary software components that can interact. COM is the foundation technology for Microsoft's OLE (compound documents) and ActiveX (Internet-enabled components) technologies. You will use COM if you are using Excel or Word in your .NET application.

1. **What is globalization and localization?**

Globalization is the process of designing and developing applications that function for multiple cultures. Localization is the process of customizing your application for a given culture and locale. Resource files play a major role in developing multicultural application.

1. **What is singleton pattern/class/implementation?**

A class whose only one instance can be created is called singleton class. Following is multithreaded version of singleton class.

public sealed class Singleton

{

private static volatile Singleton instance;

private static object syncRoot = new Object();

private Singleton() { }

public static Singleton Instance

{

get

{

if (instance == null)

{

lock (syncRoot)

{

if (instance == null)

instance = new Singleton();

}

}

return instance;

}

}

}

1. **What is difference between static and singleton?**

A singleton allows access to a single created instance - that instance (or rather, a reference to that instance) can be passed as a parameter to other methods, and treated as a normal object. A static class allows only static methods.

1. **What are attributes in C#?**

Attributes provide a way of associating metadata, or declarative information, with code (assemblies, types, methods, properties. Common attributes are [WebMethod], [Serializable]. For more on attributes <http://msdn.microsoft.com/en-us/library/z0w1kczw.aspx>

1. **What are indexers?**

Indexers allow your class to be used just like an array. On the inside of a class, you manage a collection of values any way you want. These objects could be a finite set of class members, another array, or some complex data structure. Regardless of the internal implementation of the class, its data can be obtained consistently through the use of indexers.

class IntIndexer

{

private string[] myData;

public IntIndexer(int size)

{

myData = new string[size];

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

{

myData[i] = "empty";

}

}

public string this[int pos]

{

get

{

return myData[pos];

}

set

{

myData[pos] = value;

}

}

static void Main(string[] args)

{

int size = 10;

IntIndexer myInd = new IntIndexer(size);

myInd[9] = "Some Value";

myInd[3] = "Another Value";

myInd[5] = "Any Value";

Console.WriteLine("\nIndexer Output\n");

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

{

Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);

}

}

}

1. **What are iterators?**

An iterator method or get accessor performs a custom iteration over a collection. An iterator method uses the [Yield](http://msdn.microsoft.com/en-us/library/hh156729.aspx) (Visual Basic) or [yield return](http://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx) (C#) statement to return each element one at a time. When a Yield or yield return statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

static void Main()

{

foreach (int number in EvenSequence(5, 18))

{

Console.Write(number.ToString() + " ");

}

// Output: 6 8 10 12 14 16 18

Console.ReadKey();

}

public static System.Collections.Generic.IEnumerable<int>

EvenSequence(int firstNumber, int lastNumber)

{

// Yield even numbers in the range.

for (int number = firstNumber; number <= lastNumber; number++)

{

if (number % 2 == 0)

{

yield return number;

}

}

}