**OOPS Interview Questions**

**What are the differences between procedure-oriented languages and object-oriented languages?**

*Procedure Oriented Language:*

PO Language is fully concentrates on Procedures/functions/methods. It normally works as a

sequence of actions as seen in flowchart or in any algorithm. It follows top-down approach. It totally focuses on methods and not the data which is utilized by methods. In PO languages if data is used by many methods then its declared as global data but there is a problem if we do that, what is that, if we forgot or by mistake if we consume that data in some other method than it comes with problem. Mostly these scenarios happen in large systems.

Example: COBOL, PASCAL, C, FORTRAN etc.

*Object Oriented Language:*

OO concepts says it thinks about data and binds that data and methods those are manipulating that data into one entity known as object and then utilize that object into system. It uses bottom up approach.

Example: C++, Java, C#, VB.Net etc.

There are some fundamental concepts of OO Language which a language has to follow to be a truly OO language.

• OBJECT

• CLASS

• ABSTRACTION

• ENCAPSULATION

• DATA HIDING / INFORMATION HIDING

• INHERITANCE

• POLYMORPHISM

**List features of object oriented programming.**

Object oriented programming features:   
Follows bottom up approach.  
Emphasis is on data.  
Programs are divided into objects.  
Functions and data are bound together.  
Communication is done through objects.  
Data is hidden.  
  
**List features of procedure oriented programming.**

Procedure oriented programming features:  
Follows top down approach.  
Emphasis is on procedure.  
Programs are divided into functions.  
Data moves around freely.

**What is Object Oriented programming?**

According to Wikipedia, **Object-oriented programming (OOP)** is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. Programming techniques may include features such as **abstraction, encapsulation, inheritance, and polymorphism**.

**Explain basic features of Object Oriented Programming.**

###### *Abstraction*

Abstraction is a process of identifying the relevant qualities and behaviors an object should possess.

Let’s take an example to understand abstraction. A Laptop consists of many things such as processor, motherboard, RAM, keyboard, LCD screen, wireless antenna, web camera, USB ports, battery, speakers etc. To use it, you don't need to know how internally LCD screens, keyboard, web camera, battery, wireless antenna, speaker’s works.  You just need to know how to operate the laptop by switching it on. The intrinsic details are invisible. Think about if you would have to call to the engineer who knows all internal details of the laptop before operating it. This would have highly expensive as well as not easy to use everywhere by everyone. So here the Laptop is an object that is designed to hide its complexity.

Think if you need to write a piece of software to track the student’s details of a school, you may probably need to create Students objects. People comes in all different backgrounds, educational qualifications, locations, hobbies, ages and have multiple religion, language but in terms of application, an student is just a name, age, class and roll number, while the other qualities are not relevant to the application. Determining what other qualities (background, qualifications, location, hobbies etc) are in terms of this application is abstraction.

*In object-oriented software, complexity is managed by using abstraction.* *Abstraction is a process that involves identifying the critical behavior of an object and eliminating irrelevant and complex details.* A well thought-out abstraction is usually simple, and easy to use in the perspective of the user, the person who is using your object.

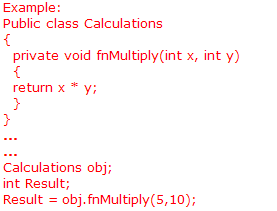
###### *Encapsulation*

Encapsulation is a method for protecting data from unwanted access or alteration by packaging it in an object where it is only accessible through the object's interface. Encapsulation is often referred to as information hiding. But both are different. In fact *information hiding* is actually the result of Encapsulation. Encapsulation makes it possible to separate an object's implementation from its original behavior - to restrict access of its internal data. This restriction facilitates certain details of an object’s behavior to be hidden. This allows to protect an object's internal state from corruption by its user.

It is the mechanism by which Abstraction is implemented. In other words you can say that it is the result of the Encapsulation. For example, the Laptop is an object that encapsulates many technologies/hardware that might not be understood clearly by most people who use it.

*Encapsulation is the ability of an object to hide its data and methods from the rest of world. It is one of the fundamental principles of OOPs*.

Say we create a class, named Calculations. This class may contain a few members in form of properties, events, fields or methods. Once the class is created, we may instantiate the class by creating an object out of it. The object acts as an instance of this class, members of the class are not exposed to the outer world directly, rather, they encapsulated by the class.



###### *Inheritance*

*Inheritance is the ability to define a new class or object that inherits the behavior and its functionality of an existing class. The new class or object is called a child or subclass or derived class while the original class is called parent or base class.*

For example, in a software company Software Engineers, Sr. Software Engineers, Module Lead, Technical Lead, Project Lead, Project Manager, Program Manager, Directors all are the employees of the company but their work, perks, roles, responsibilities differs. So in OOP, the Employee base class would provide the common behaviors of all types/level of employee and also some behaviors properties that all employee must have for that company. The particular sub class or child class of the employee would implement behaviors specific to that level of the employee. So by above example you can notice that the main concept behind inheritance are *extensibility and code reuse* (in this case you are extending the Employee class and using its code into sub class or derived class).

###### *Polymorphism*

As name suggests, Polymorphism means an ability to assume different forms at different places. In OOP, it is a language's ability to handle objects differently based on their runtime type and use. Polymorphism is briefly described as "one interface, many implementations". *Polymorphism is a characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form.*

There are two types of polymorphism.

1. Compile time polymorphism - It is achieved by overloading functions and operators
2. Run time polymorphism - It is achieved by overriding virtual functions

Let’s say you have a class that have many Load methods having different parameters, this is called Compile time polymorphism. Let’s take another example where you have a virtual method in the base class called Load with one parameter and you have redefined its functionality in your sub class by overriding base class Load method, this is called Run time polymorphism.

**State some of the advantages of object oriented programming?**

Some of the advantages of object oriented programming are as follows: -  
1) A clear modular structure can be obtained which can be used as a prototype and it will not reveal the mechanism behind the design. It does have a clear interface.  
2) Ease of maintenance and modification to the existing objects can be done with ease.  
3) A good framework is provided which facilitates in creating rich GUI applications.

Data hiding helps create secure programs.  
Redundant code can be avoided by using inheritance.  
Multiple instances of objects can be created.  
Work can be divided easily based on objects.  
Inheritance helps to save time and cost.  
Easy upgrading of systems is possible using object oriented systems.

**What are the problems faced by the developer using object oriented programming language?**

These are some of the problems faced by the developer using object oriented language they are: -  
1) Object oriented uses design patterns which can be referred to as anything in general.  
2) Repeatable solution to a problem can cause concern and disagreements and it is one of the major problems in software design.

**What are the basic concepts of OOPs?**

The following are the basic concepts of OOPs:  
Classes, Objects, Data abstraction and encapsulation, Polymorphism, Inheritance, Message Passing, and Dynamic Binding.

**What is data encapsulation?**  
Wrapping up of member data and member functions together in a class is called data encapsulation.  
  
**What is data abstraction?**  
Data abstraction refers to the act of providing only required features and hiding all the non-essential details for usage.

**Explain the abstraction and encapsulation principle.**

***Abstraction*** means hiding the internal details and just exposing the functionality. For Example:-When you change the gear of your car, you know the gears will be changed without knowing how they are functioning internally. Abstraction focuses on the outside view of an object (i.e. the interface)  
  
***Encapsulation*** means put the data and the function that operate on that data in a single unit(information hiding) .Encapsulation prevents clients from seeing its inside view, where the behavior of the abstraction is implemented.

Go to the source! Grady Booch says (in Object Oriented Analysis and Design, page 49, second edition):

"Abstraction and encapsulation are complementary concepts: abstraction focuses on the observable behavior of an object... encapsulation focuses upon the implementation that gives rise to this behavior... encapsulation is most often achieved through information hiding, which is the process of hiding all of the secrets of object that do not contribute to its essential characteristics."

In other words: abstraction = the object externally; encapsulation (achieved through information hiding) = the object internally,

*Example 1:*

In the .NET Framework, the System.Text.StringBuilder class provides an abstraction over a string buffer. This buffer abstraction lets you work with the buffer without regard for its implementation. Thus, you're able to append strings to the buffer without regard for how the StringBuilder internally keeps track of things such the pointer to the buffer and managing memory when the buffer gets full (which it does with encapsulation via information hiding).

*Example 2:*

Encapsulation is defined as the process of wrapping up the data members and member functions together into a single unit. It is dependent on the object data type.  
Abstraction is the process of eliminating unnecessary details thereby highlighting the important features. It is independent of the object data type.  
  
A real world example, consider you have setup a big building(say a company), the details regarding materials used to built (glass, bricks), type of work, manager of the company, number of floors, design of the building, cost of the building etc. can be classified as ABSTRACTION. Whereas, type of glass or bricks (grey one or red one) used, who all work for which all departments and how they work, cost of each and every element in the building, etc, comes under Data ENCAPSULATION.

Abstraction-outer layout, used in terms of design  
Encapsulation-inner layout, used in terms of implementation.

**What is the advantage of encapsulation in object oriented programming?**

Encapsulation is important because it maintains object integrity. By providing only the most necessary means of data access in the interface, we can prevent unwanted changes to the object’s state through hidden, or encapsulated, methods and properties. With well defined interfaces that rarely change, user interactions and unit testing become oblivious to changes in implementation.

One advantage is that with encapsulation, when using objects, the object need not reveal all its attributes and behaviors. In good Object-Oriented Design (at least what is generally accepted as good), an object should only reveal the interfaces needed to interact with it. Details not pertinent to the use of the object should be hidden from other objects. It is really the fundamental concept of Object-Oriented. Whenever the interface/implementation paradigm is covered, we are really talking encapsulation. This encapsulation pertains equally to data and behavior.

Another big advantage of encapsulation is that it makes using information for unintended purposes more difficult, and this reduces logic errors.

| What is the difference between encapsulation and data hiding? |
| --- |

There are a lot of pretty much different concepts. Three a little bit related are Info Hiding, Data Hiding and Encapsulation.

With no details, you may use all terms as a synonyms, but still if you are talking about Paradigm or Concept of OOP you should clearly understand the difference.

So, *Data Hiding* is a concept of making all the data (fields) private e.g. not accessible from the other objects, classes, APIs in the system. The only class/object/API have to know about the data and how to operate them. Private fields with accessors is a good example.

As for the *Info Hiding* - this more about the methods. Now, you are trying to hide e.g. make private the implementation of methods, to hide the design decisions. For example, you API should return Interface Object reference but not Class Object reference and then another client API wouldn't know how you API is implemented. For example, if you are getting cash in ATM you should just type your pin a get your money and you don't really know how they are transferred to you from you balance.

Finally, the *Encapsulation* is much more than just Data Hiding. Some folks do think this about data, some - about methods. I should say this about both. Also, I'm not in favour of talking about Hiding at all when you describe the incapsulation term. My call is that the Encapsulation is logical integration of the data and/or the function within one entity (class, object, package, library, API etc) with or without hiding them from another entities. Also, I believe that one more very important thing is that Function (operations, responsibility, scope of methods) is encapsulated the only in the case if there is now any other entity with the same (or very similar) Function in the system.

And few samples to sum up. If we clearly understand all the concept than we may say three following things about our application:

1. Library Blah encapsulate all operations (methods) needed to work with the Database.

2. Design decisions that were made during implementation of Library Blah are hidden within the library.

3. All the data that are transferred from/to the Database through Library Blah are hidden within appropriate Transfer Objects and accessible only through their accessors.

**What are ADTs?**  
ADTs stand for abstract data types. Classes which provide data abstraction are referred to as ADTs.  
  
**What is inheritance?**  
The process of inheriting the properties of one object by another object is called inheritance.

C# supports two types of Inheritance mechanisms:  
1) Implementation Inheritance   
2) Interface Inheritance  
  
**What is Implementation Inheritance?**When a class (type) is derived from another class (type) such that it inherits all the members of the base type it is Implementation Inheritance.  
  
**What is Interface Inheritance?**When a type (class or a struct) inherits only the signatures of the functions from another type it is Interface Inheritance.   
  
In general, Classes can be derived from another class, hence support Implementation inheritance. At the same time Classes can also be derived from one or more interfaces. Hence they support Interface inheritance. Structs can derive from one more interface, hence support Interface Inheritance. Structs cannot be derived from another class they are always derived from System.ValueType.

**Give example of Implicit and Explicit Interface Implementations.**

As mentioned before .Net support multiple implementations, the concept of implicit and explicit implementation provide safe way to implement methods of multiple interfaces by hiding, exposing or preserving identities of each of interface methods, even when the method signatures are the same.

Let's consider the interfaces defined below.

interface IDisposable

{

void Dispose();

}

Here you can see that the class *Student* has implicitly and explicitly implemented the method named *Dispose()* via *Dispose* and *IDisposable.Dispose*.

class Student : IDisposable

{

public void Dispose()

{

Console.WriteLine("Student.Dispose");

}

void IDisposable.Dispose()

{

Console.WriteLine("IDisposable.Dispose");

}

}

**What are the advantages of inheritance?**

Inheritance offers the following advantages --

1.Developement model closer to real life object model with hierarchical relationships

2.Reusability -- facility to use public methods of base class without rewriting the same

3.Extensibility -- extending the base class logic as per business logic of the derived class

4.Data hiding -- base class can decide to keep some data private so that it cannot be altered by the derived class

**Discuss how derived class access to base class members.**

A derived class has access to the public, protected, internal, and protected internal members of a base class. Even though a derived class inherits the private members of a base class, it cannot access those members. However, all those private members are still present in the derived class and can do the same work they would do in the base class itself. For example, suppose that a protected base class method accesses a private field. That field has to be present in the derived class in order for the inherited base class method to work properly.

**Why doesn't the Java language support multiple inheritance?**

Whenever you find yourself asking why Java has or does not have some feature, consider the design goals behind the Java language. With that in mind, I started my search by skimming through "The Java Language Environment" by James Gosling and Henry McGilton (Sun Microsystems), a white paper published in May 1996 that explains some of the reasoning behind Java's design.

As the white paper states, the Java design team strove to make Java:

* Simple, object oriented, and familiar
* Robust and secure
* Architecture neutral and portable
* High performance
* Interpreted, threaded, and dynamic

The reasons for omitting multiple inheritance from the Java language mostly stem from the "simple, object oriented, and familiar" goal. As a simple language, Java's creators wanted a language that most developers could grasp without extensive training. To that end, they worked to make the language as similar to C++ as possible (familiar) without carrying over C++'s unnecessary complexity (simple).

In the designers' opinion, multiple inheritance causes more problems and confusion than it solves. So they cut multiple inheritances from the language (just as they cut operator overloading). The designers' extensive C++ experience taught them that multiple inheritances just weren’t worth the headache.

**Note:** For a discussion of the diamond problem, a classic multiple inheritance challenge, read Bill Venners's "[Designing with Interfaces](http://www.javaworld.com/javaworld/jw-12-1998/jw-12-techniques.html)" (*JavaWorld,* December 1998) and Tony Sintes's "[Java Diamonds Are Forever](http://www.javaworld.com/javaworld/javaqa/2001-03/02-qa-0323-diamond.html)" (*JavaWorld,* March 2001).

Instead, Java's designers chose to allow multiple interface inheritance through the use of interfaces, an idea borrowed from Objective C's protocols. Multiple interface inheritance allows an object to inherit many different method signatures with the caveat that the inheriting object must implement those inherited methods.

Multiple interface inheritance still allows an object to inherit methods and to behave polymorphically on those methods. The inheriting object just doesn't get an implementation free ride. For an excellent discussion of interface inheritance, read Wm. Paul Rogers's "[Reveal the Magic Behind Subtype Polymorphism](http://www.javaworld.com/javaworld/jw-04-2001/jw-0413-polymorph.html)" (*JavaWorld,* April 2001).

**Why multiple inheritances are not supported by Java/C#?**

Due to the concept of overridding Java doesn't support multiple inheritance.

We have the concept of Deadly Diamond of Death .

i.e.

As we have an overridding concept in Java.

We know what overridding is where the method name including parameters return types should be same.

If mutliple inheritance is provided in java.

eg:

A<- B A-<C and now B C<-D

We have a method viz. display() in A we are overriding that in B and C. If D extends that two classes then if I call the display() then there will be ambiguity. To call which version of display().

As we don’t have the concept of virtual functions what we have in c++.

This is the reason why we don’t have multiple inheritances in java.

When the multiple inheritance is allowed, means when a language allows the class to extend multiple classes, that leads to the ambiguity as to which class method to consider, when two of its parents have the same method signature. This is called diamond ring problem.

**What are the types of Inheritance?**

There are five types of inheritance.  
1. Single inheritance  
2. Multilevel inheritance  
3. Multiple inheritance  
4. Hierarchical inheritance  
5. Hybrid inheritance

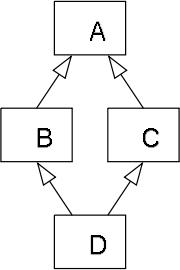
We can take support for multiple inheritances also but with the help of Interface in Java/C#.

**What is diamond problem/dreaded diamond in OOP languages?**

In [object-oriented](http://object-oriented) [programming languages](http://language) with [multiple inheritance](http://inheritance) and knowledge organization, 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?

For example, in the context of [GUI](http://gui) [software development](http://development), a class Button may inherit from both classes Rectangle (for appearance) and Clickable (for functionality/input handling), and classes Rectangle and Clickable both inherit from the Object class. Now if the equals method is called for a Button object and there is no such method in the Button class but there is an overridden equals method in both Rectangle and Clickable, which method should be eventually called?

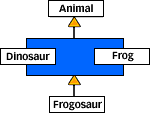
It is called the "diamond problem" because of the shape of the class inheritance diagram in this situation. In this article, class A is at the top, both B and C separately beneath it, and D joins the two together at the bottom to form a diamond shape.



*A diamond class inheritance diagram.*

**Give an example of diamond problem. How can this problem be solved?**

This Jurassic Park scenario potentially could be represented by the following inheritance hierarchy:



**Figure 1. Multiple inheritance in Jurassic Park**

The diamond problem can arise in inheritance hierarchies like the one shown in Figure 1. In fact, the diamond problem gets its name from the diamond shape of such an inheritance hierarchy. One way the diamond problem can arise in the *Jurassic Park* hierarchy is if both Dinosaur and Frog, but not Frogosaur, override a method declared in Animal. Here's what the code might look like if Java supported traditional multiple inheritance:

abstract class Animal {

abstract void talk();

}

class Frog extends Animal {

void talk() {

System.out.println("Ribit, ribit.");

}

class Dinosaur extends Animal {

void talk() {

System.out.println("Oh I'm a dinosaur and I'm OK...");

}

}

// (This won't compile, of course, because Java

// only supports single inheritance.)

class Frogosaur extends Frog, Dinosaur {

}

The diamond problem rears its ugly head when someone tries to invoke talk() on a Frogosaur object from an Animal reference, as in:

Animal animal = new Frogosaur();

animal.talk();

Because of the ambiguity caused by the diamond problem, it isn't clear whether the runtime system should invoke Frog's or Dinosaur's implementation of talk(). Will a Frogosaur croak "Ribbit, Ribbit." or sing "Oh, I'm a dinosaur and I'm okay..."?

The diamond problem would also arise if Animal had declared a public instance variable, which Frogosaur would then have inherited from both Dinosaur and Frog. When referring to this variable in a Frogosaur object, which copy of the variable -- Frog's or Dinosaur's -- would be selected? Or, perhaps, would there be only one copy of the variable in a Frogosaur object?

*Solution*

The best solution is to try to avoid such diamond problems.  That will keep the design simple.

In Java, interfaces solve all these ambiguities caused by the diamond problem. Through interfaces, Java allows multiple inheritance of interface but not of implementation. Implementation, which includes instance variables and method implementations, is always singly inherited. As a result, confusion will never arise in Java over which inherited instance variable or method implementation to use.

Languages that allow only single inheritance (such as Ada, [Objective-C](http://objective-c), [PHP](http://php), C#, [Delphi](http://delphi)/[Free Pascal](http://pascal) and Java) allow the multiple inheritance of interfaces (called protocols in Objective-C). Interfaces are essentially abstract base classes with all abstract methods and no data members. The problem is therefore avoided since there is always only one implementation of a specific method or property and no ambiguity arises.

**Explain polymorphism in C#.**

Polymorphism is one of the fundamental concepts of OOP.  
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.

*Types of Polymorphism:*

There are 2 types of Polymorphism namely  
1.Compile time polymorphism (or) Overloading  
2.Runtime polymorphism (or) Overriding

*Compile Time Polymorphism:*

*Compile time polymorphism is method and operator overloading. It is also called early binding.  
Method with same name but with different arguments is called method overloading.*public class A1  
{  
public void hello()  
{ Console.WriteLine(“Hello”); }  
  
public void hello(string s)  
{ Console.WriteLine(“Hello {0}”,s); }  
}

*Runtime Time Polymorphism:*

*Runtime time polymorphism is done using inheritance and virtual/abstract functions. Method overriding is called runtime polymorphism. It is also called late binding.*Method overriding occurs when child class declares a method that has the same type arguments as a method declared by one of its super class.  
*When overriding a method, you change the behavior of the method for the derived class.*

public Class parent  
{  
virtual void hello()  
{ Console.WriteLine(“Hello from Parent”); }  
}  
  
public Class child : parent  
{  
override void hello()  
{ Console.WriteLine(“Hello from Child”); }  
}  
  
public static void main()  
{  
parent objParent = new child();  
objParent.hello(); // Hello from Child  
}

**Explain polymorphism in C# with a simple example?**   
*Polymorphism allows you to invoke derived class methods through a base class reference during run-time.* An example is shown below.using System;public class DrawingObject{ public virtual void Draw() { Console.WriteLine("I am a drawing object."); }}public class Triangle : DrawingObject{ public override void Draw() { Console.WriteLine("I am a Triangle."); }}public class Circle : DrawingObject{ public override void Draw() { Console.WriteLine("I am a Circle."); }}public class Rectangle : DrawingObject { public override void Draw() { Console.WriteLine("I am a Rectangle."); }}public class DrawDemo{ public static void Main() { DrawingObject[] DrawObj = new DrawingObject[4]; DrawObj[0] = new Triangle(); DrawObj[1] = new Circle(); DrawObj[2] = new Rectangle(); DrawObj[3] = new DrawingObject(); foreach (DrawingObject drawObj in DrawObj) { drawObj.Draw(); } }}  
**When can a derived class override a base class member?**   
A derived class can override a base class member only if the base class member is declared as virtual or abstract.  
  
**What is the difference between a virtual method and an abstract method?**   
A virtual method must have a body where as an abstract method should not have a body.  
  
**Can fields inside a class be virtual?**No, Fields inside a class cannot be virtual. Only methods, properties, events and indexers can be virtual.  
  
**Give an example to show for hiding base class methods?**   
Use the new keyword to hide a base class method in the derived class as shown in the example below.  
using System;  
public class BaseClass  
{  
 public virtual void Method()  
 {  
 Console.WriteLine("I am a base class method.");  
 }  
}  
public class DerivedClass : BaseClass  
{  
 public new void Method()  
 {  
 Console.WriteLine("I am a child class method.");  
 }  
  
 public static void Main()  
 {  
 DerivedClass DC = new DerivedClass();  
 DC.Method();  
 }  
}  
  
**Can you access a hidden base class method in the derived class?**   
Yes, Hidden base class methods can be accessed from the derived class by casting the instance of the derived class to an instance of the base class as shown in the example below.

using System;  
public class BaseClass  
{  
 public virtual void Method()  
 {  
 Console.WriteLine("I am a base class method.");  
 }  
}  
public class DerivedClass : BaseClass  
{  
 public new void Method()  
 {  
 Console.WriteLine("I am a child class method.");  
 }  
  
 public static void Main()  
 {  
 DerivedClass DC = new DerivedClass();  
 ((BaseClass)DC).Method();  
 }  
}

**What is the difference between shadow and override? or**

**What is the difference between new and override?**

The differences between the two C# keywords is due to polymorphism. When a *virtual method* is called on a reference, the *actual type* of the object to which the reference refers is used to determine which method implementation should be used. When a method of a base class is *overridden* in a derived class (subclass), the version defined in the derived class is used. This is so even should the calling application be unaware that the object is an instance of the derived class.

The following example illustrates the override keyword:

public class BaseClass

{

public virtual void AMethod()

{

}

}

public class DerivedClass : BaseClass

{

public override void AMethod()

{

}

}

*// ...*

BaseClass baseClass = new DerivedClass();

baseClass.AMethod();

In the preceding example, DerivedClass.AMethod will be called; because, it overrides BaseClass.AMethod.

If the new keyword were used instead of the override keyword, then the method in the derived class would not override the base class method, but would *hide* it instead. The following is an example illustrating the new keyword:

public class BaseClass

{

public virtual void AMethod()

{

}

}

public class DerivedClass : BaseClass

{

public new void AMethod()

{

}

}

*// ...*

BaseClass baseClass = new DerivedClass();

baseClass.AMethod();

DerivedClass derivedClass = new DerivedClass();

derivedClass.AMethod();

The preceding example will call BaseClass.AMethod first, followed by DerivedClass.AMethod. Effectively, the two methods are unrelated except for the fact that they have the same name.

If neither the new nor overrides keyword is specified, the result is the same as if the new were specified. However, a compiler warning will be generated to ensure that the programmer is aware that a method in the base class is being hidden.

**Explain static/early binding with an example.**

When a reference type variable is initialized, the reference may hold either:

a) an object of the same type as that of the reference type

b) an object of a type related to the reference type

In the most general terms, ***static binding*** *means that references are resolved at compile time.* For example,

Animal a = new Animal();

a.Roar(); // The compiler can resolve this method call statically.

*Example 1:*

public class MyClass

{

public void DoSomething(){...}

}

public class MyOtherClass

{

public MyOtherClass()

{

MyClass mc = new MyClass();

mc.DoSomething();

}

}

The call to DoSomething was bound statically to the instance of MyClass that we had created on the previous line. This is known at compile time, and therefore is the only thing that can happen when this code runs.

*Example 2:*

1

2                  public class Base

3                  {

4                  public void Print()

5                  {

6                  Console.WriteLine("Base");

7                  }

8                  }

9

10              public class Derv : Base

11              {

12              public void Print()

13              {

14              Console.WriteLine("Derv");

15              }

16              }

17

The code below is perfectly legal in this context:

1

2                  Base obj;

3

4                  obj = new Base();

5

6                  // This is possible since Derv is related to Base

7                  obj = new Derv();

8

In such a scenario, *when a method is invoked against a reference variable, the actual method going to be called depends upon the type of the reference.* For example:

1

2                  Base obj;

3                  Derv der;

4

5                  obj = new Base();

6                  obj.Print(); // Output: 'Base'

7

8                  obj = new Derv();

9                  obj.Print(); // Output: 'Base'

10

11              der = new Derv();

12              der.Print(); // Output: 'Derv'

13

At line number 9, the compiler determines which method to call (the Base.Print method or Derv.Print method) by checking the type of the reference variable obj. Since the reference variable obj (not the actual object stored at the reference) happens to be of type Base, the compiler calls Base.Print. In a similar manner, at line number 12, the compiler invokes Derv.Print as the reference variable der is of type Derv.

This is known as *Static Binding* because the compiler does all this at compile time while building the final IL code, which is thus static (unchangeable at run-time).

Method Hiding

The type Derv derives from Base and thus inherits the Base.Print method. But note that Derv itself defines a Print method. When a derived class defines a method with the same name as that of a method in the base class, the base class’s method would never be invoked if the type of reference variable is that of the derived class. In the previous code snippet at line number 12, though Derv inherits the Base.Print method, it was Derv.Print that was invoked. Here, Derv.Print is said to be *hiding* Base.Print.

In order to make sure that this is indeed what the developer requires, the compiler would issue a warning saying that Derv.Print is hiding Base.Print. The compiler simply wants to bring to the attention of the programmer that there exists a method with the same name in the base class, and that the derived class’s method is hiding it. To get rid of the warning, we need to use the new keyword:

1

2                  public class Derv : Base

3                  {

4                  public new void Print()

5                  {

6                  Console.WriteLine("Derv");

7                  }

8                  }

9

This reassures the compiler that the programmer knows about this issue, and that this implementation is purposeful.

**Explain Dynamic/late binding with an example.**

**Dynamic binding** means that references are resolved *at run time*. For example,

public void MakeSomeNoise(object a) {

   // Things happen...

   ((Animal) a).Roar(); // You won't know if this works until runtime!

}

When you start dealing with class hierarchies and virtual methods, compiler will start using so called VTABLEs. At that time the compiler doesn't know exactly what method to call and it has to wait until runtime to figure out the right method to be invoked (this is done through VTABLE). This is called dynamic binding.

*Example 1:*

public abstract class WidgetBase

{

public abstract void DoSomething();

}

public class ShinyWidget : WidgetBase

{

public override void DoSomething()

{

// implementation

}

}

public class DullWidget : WidgetBase

{

public override void DoSomething()

{

// implementation

}

}

public class MyOtherClass

{

public void DoSomethingWithAWidget(WidgetBase widget)

{

widget.DoSomething();

}

}

When someone instantiates MyOtherClass, they will call DoSomethingWithAWidget and pass in an instance of ShinyWidget or DullWidget. We have no idea which one is going to be passed in, so we can't know at compile time which implementation of DoSomething is going to be called. This is dynamic binding.  
  
The concept of static/dynamic binding is certainly not only applicable to abstract classes. This is only one example. The same concept applies to virtual methods, in many cases.

MyNonAbstractBaseClass bc = new ImInheritedFromMNABC();

bc.VirtalMethodName();

This would use dynamic binding as well.

*Example 2:*

By using the **virtual** and **override** keywords, you could control which method is invoked in an inheritance scenario.

1

2                  public class Base

3                  {

4                  public virtual void Print()

5                  {

6                  Console.WriteLine("Base");

7                  }

8                  }

9

10              public class Derv : Base

11              {

12              public override void Print()

13              {

14              Console.WriteLine("Derv");

15              }

16              }

17

This definition would yield the following result.

1

2                  Base obj;

3

4                  obj = new Base();

5                  obj.Print(); // Output: 'Base'

6

7                  obj = new Derv();

8                  obj.Print(); // Output: 'Derv'

9

Note that at line number 8, even though the reference variable **obj** was of type **Base**, the method that was invoked was **Derv.Print** rather than **Base.Print**. This is because of the following reasons:

* **Base.Print** declares itself as **virtual**, which means derived classes may override this method
* **Derv.Print** explicitly states that this method overrides the **Base.Print** method, by using the **override** keyword

Thus, when the **Print** method is invoked, the derived type’s method gets invoked.

In addition to **Derv**, there could be more types which derive from **Base**, or even from **Derv** or any of its subclasses. Consequently, at any point of time, the reference variable **obj** could be holding any object instance whose type is in some way related to the type **Base**. It is thus not possible to find out what is the actual type of the instance held in **obj** until runtime. Hence, binding happens at runtime when the actual object instance exists, and the method is then invoked dynamically. This is known as *Dynamic Binding*.

*Example 3: Dynamic/late Binding with hiding.*

class A {

public virtual void WhoAreYou() { Console.WriteLine("I am an A"); }

}

class B : A {

public override void WhoAreYou() { Console.WriteLine("I am a B"); }

}

class C : B {

public new virtual void WhoAreYou() { Console.WriteLine("I am a C"); }

}

class D : C {

public override void WhoAreYou() { Console.WriteLine("I am a D"); }

}

C c = new D();

c.WhoAreYou();// "I am a D"

A a = new D();

a.WhoAreYou();// "I am a B" !!

**What is the concept of V-Table in C#?**

***vtable****, is a mechanism used in a programming language to support dynamic dispatch (or run-time method binding).*

Suppose a program contains several classes in an inheritance hierarchy: a superclass, Cat, and two subclasses, HouseCat and Lion. Class Cat defines a virtual function named speak, so its subclasses may provide an appropriate implementation (i.e., either meow or roar).

When the program calls the speak method on a Cat pointer (which can point to a Cat class, or any subclass of Cat), the run-time environment must be able to determine which implementation to call, depending on the actual type of object that is pointed to.

**When is a vtable created: compile time or run time? In particular, if a declare a class having one or more virtual functions, and in first case I create an object of it and in second case I don't create an object. What difference it makes in terms of number of vtable’s created?**  
A vtable is created in an instance of a class when the base class has one or more virtual functions.  The object of the class doesn't need a vtable because this object will call it's member functions.  An object subclassed off the object with virtual functions will have a vtable so that at runtime, an object of this class will use it's member function and not the base classes.

**Does that mean that a VTABLE is created per class, no matter their object is created or not; considering only classes having virtual functions and those derived from them.**

A vtable is created per \*object\*, because only at runtime will the object know if it the subclassed class or base class.  This could be a lot of memory if you have 10000 objects of a class with virtual functions, but for normal data demands, you'll never have to worry about the extra memory.

**Give example how virtual functions are called?**

To implement virtual functions, C++ uses a special form of late binding known as the virtual table. The virtual table is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as “vtable”, “virtual function table”, “virtual method table”, or “dispatch table”.

Because knowing how the virtual table works is not necessary to use virtual functions, this section can be considered optional reading.

The virtual table is actually quite simple, though it’s a little complex to describe in words. First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it’s own virtual table. This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.

Second, the compiler also adds a hidden pointer to the base class, which we will call \*\_\_vptr. \*\_\_vptr is set (automatically) when a class instance is created so that it points to the virtual table for that class. Unlike the \*this pointer, which is actually a function parameter used by the compiler to resolve self-references, \*\_\_vptr is a real pointer. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that \*\_\_vptr is inherited by derived classes, which is important.

By now, you’re probably confused as to how these things all fit together, so let’s take a look at a simple example:

class Base

{

public:

virtual void function1() {};

virtual void function2() {};

};

class D1: public Base

{

public:

virtual void function1() {};

};

class D2: public Base

{

public:

virtual void function2() {};

};

Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.

The compiler also adds a hidden pointer to the most base class that uses virtual functions. Although the compiler does this automatically, we’ll put it in the next example just to show where it’s added:

class Base

{

public:

FunctionPointer \*\_\_vptr;

virtual void function1() {};

virtual void function2() {};

};

class D1: public Base

{

public:

virtual void function1() {};

};

class D2: public Base

{

public:

virtual void function2() {};

};

When a class object is created, \*\_\_vptr is set to point to the virtual table for that class. For example, when a object of type Base is created, \*\_\_vptr is set to point to the virtual table for Base. When objects of type D1 or D2 are constructed, \*\_\_vptr is set to point to the virtual table for D1 or D2 respectively.

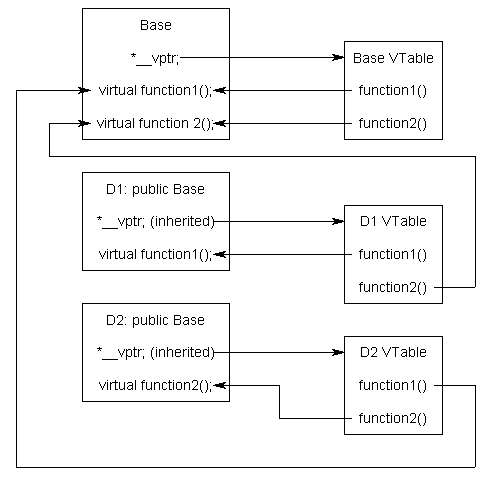
Now, let’s talk about how these virtual tables are filled out. Because there are only two virtual functions here, each virtual table will have two entries (one for function1(), and one for function2()). Remember that when these virtual tables are filled out, each entry is filled out with the most-derived function an object of that class type can call.

Base’s virtual table is simple. An object of type Base can only access the members of Base. Base has no access to D1 or D2 functions. Consequently, the entry for function1 points to Base::function1(), and the entry for function2 points to Base::function2().

D1’s virtual table is slightly more complex. An object of type D1 can access members of both D1 and Base. However, D1 has overridden function1(), making D1::function1() more derived than Base::function1(). Consequently, the entry for function1 points to D1::function1(). D1 hasn’t overridden function2(), so the entry for function2 will point to Base::function2().

D2’s virtual table is similar to D1, except the entry for function1 points to Base::function1(), and the entry for function2 points to D2::function2().

Here’s a picture of this graphically:



Although this diagram is kind of crazy looking, it’s really quite simple: the \*\_\_vptr in each class points to the virtual table for that class. The entries in the virtual table point to the most-derived version of the function objects of that class are allowed to call.

So consider what happens when we create an object of type D1:

int main()

{

D1 cClass;

}

Because cClass is a D1 object, cClass has it’s \*\_\_vptr set to the D1 virtual table.

Now, let’s set a base pointer to D1:

int main()

{

D1 cClass;

Base \*pClass = &cClass;

}

Note that because pClass is a base pointer, it only points to the Base portion of cClass. However, also note that \*\_\_vptr is in the Base portion of the class, so pClass has access to this pointer. Finally, note that pClass->\_\_vptr points to the D1 virtual table! Consequently, even though pClass is of type Base, it still has access to D1’s virtual table.

So what happens when we try to call pClass->function1()?

int main()

{

D1 cClass;

Base \*pClass = &cClass;

pClass->function1();

}

First, the program recognizes that function1() is a virtual function. Second, uses pClass->\_\_vptr to get to D1’s virtual table. Third, it looks up which version of function1() to call in D1’s virtual table. This has been set to D1::function1(). Therefore, pClass->function1() resolves to D1::function1()!

Now, you might be saying, “But what if Base really pointed to a Base object instead of a D1 object. Would it still call D1::function1()?”. The answer is no.

int main()

{

Base cClass;

Base \*pClass = &cClass;

pClass->function1();

}

In this case, when cClass is created, \_\_vptr points to Base’s virtual table, not D1’s virtual table. Consequently, pClass->\_\_vptr will also be pointing to Base’s virtual table. Base’s virtual table entry for function1() points to Base::function1(). Thus, pClass->function1() resolves to Base::function1(), which is the most-derived version of function1() that a Base object should be able to call.

By using these tables, the compiler and program are able to ensure function calls resolve to the appropriate virtual function, even if you’re only using a pointer or reference to a base class!

Calling a virtual function is slower than calling a non-virtual function for a couple of reasons: First, we have to use the \*\_\_vptr to get to the appropriate virtual table. Second, we have to index the virtual table to find the correct function to call. Only then can we call the function. As a result, we have to do 3 operations to find the function to call, as opposed to 2 operations for a normal indirect function call, or one operation for a direct function call. However, with modern computers, this added time is usually fairly insignificant.

**Give example how virtual functions are called.**

A complete understanding of everything is very important in the programming world. I am taking this post to just give some explanations I know and the understanding I have with the Virtual Functions. This is an explanation given by one of the trainer who taught me C#. I'm not sure whether he has a blog or a personal website, if yes, I will definitely link him in my post.

Virtual functions are normal member functions of a class, which can be over-ridden in the derived classes. The whole functionality can be replaced in the over-riding function. In C#, the virtual functions will be declared with a keyword 'virtual' and the over-riding functions will be declared with 'override' key word.

Example in C#:

using System;

namespace ConsoleApplication1

{

class BaseClass

{

public BaseClass()

{

// TODO

}

virtual public void MyFunction1()

{

Console.WriteLine("MyFunction in Base class");

}

virtual public void MyFunction2()

{

Console.WriteLine("MyFunction in Base class");

}

}

class DerivedClass : BaseClass

{

public DerivedClass()

How the call to the derived class virtual function is being made? Let me present a C++ equivalent of the above code:

class BaseClass

{

public:

BaseClass()

{

}

virtual public void MyFunction1()

{

cout<<"MyFunction1 in Base class"<<endl;

}

virtual public void MyFunction2()

{

cout<<"MyFunction2 in Base class"<<endl;

}

};

class DerivedClass

{

public:

DerivedClass ()

{

}

public void MyFunction()

{

cout<<"MyFunction in Base class"<<endl;

}

};

void main()

{

DerivedClass \*obj;

obj->MyFunction();

}

I am going to explain the virtual functions with the C++ example and will give some more additional code which will explain the call semantics of the virtual functions. Whenever, there is a virtual function in the class, a v-table is constructed in the memory. The v-table has a list of addresses to the virtual functions of the class and pointers to the functions from each of the objects of the derived class. When a virtual function call is made, the v-table is used to get the addresses of the function and the function is called.

Let me also explain what are the steps that executed in a constructor. Before it executes the 1st line of code in the constructor body, the following steps are performed:

1. Base class constructer is called  
2. Call the contained object constructors  
3. Set the VPtr

The step we are interested is the 3rd step - setting the VPtr. Every object has a VPtr and is stored in the 1st memory location of the object memory. VPtr will have the address of the Virtual Table which has the function addresses of all the virtual functions available. Using the Vptr we can easily access the VirtualTable of the class and make function calls as normally done with the help of function pointers. So, as you all guessed by now, the base class constructor will be called and the VPtr of the base class will be set. But, when the call returns to the derived class constructor, the derived class VPtr will be set to its own VTable address.

How to get the VPtr of the class?

long \*vptr = (long \*) &obj;

// gets the vptr of the class which is stored in the 1st memory location of each object of the class

We got the VPtr, how we need to get the address of the VTable which is also a pointer.

long \*vtable = (long \*)\*vptr;

// get the vtable address

VTable can be considered as an array of function pointers. The addresses of virtual functions declared in the class will be stored in the order they are declared. The 0th location will have the 1st virtual function's address, 1st location have the 2nd virtual function address, so on and so forth. Now we have the address of the VTable in the pointer vtable. Lets declare a function pointer.

typedef void (\_\_stdcall \*FunctionPtr)();

FunctionPtr fp = (FunctionPtr)vtable[0];

// pointing to MyFunction1 which is in the 0th location of the v-table

We have also declared fp which is a function pointer holding a function that returns void, accepting no arguments. Let's now call the function:

fp();

fp = (FunctionPtr)vtable[1]; // pointing to 1st location which has MyFunction2

fp();

Now, lets talk about setting the VPtr (3rd step in a constructor - before any user written code is executed). It does set the Virtual Table address. It is just another step which is performed when ever a constructor is called. Now, lets review the below lines of code:

DerivedClass \*ptr = new DerivedClass();

ptr->MyFunction1(); // calls DerivedClass MyFunction1

ptr->BaseClass::BaseClass();

// BaseClass constructor is called - so VPtr altered to point to BaseClass's VTable

ptr->MyFunction1(); // calls BaseClass MyFunction1

From the comments you would have understood what is really happening. Fortunately, C# doesn't provide a way to call a base class constructor. However, you can use new operator to create a new copy of the BaseClass in this scenario to achieve the same result.

Are you a person who wish to combine many steps into one (hates to waste time in typing few more lines of code or love to minimize memory utilization by reducing the no. of variables declared or confuse someone who may try to understand your code)? If you are, following is another method of calling the first virtual function of that class:

((void(\*)())\*(long \*)\*(long \*)&obj)();

**Difference between abstract class and interface**

*To know the difference between the two, a developer must think about* [*abstraction*](http://en.wikipedia.org/wiki/Abstraction_(computer_science)#Abstraction_in_object_oriented_programming) *and* [*encapsulation*](http://en.wikipedia.org/wiki/Information_hiding)*; the two paradigm that* [*Object Oriented Programming*](http://en.wikipedia.org/wiki/Object-oriented_programming) *heavily relies on in modelling reality. Without inheritance and interfaces, we are stuck with complex trees of conditions, iterations and recursions that is probably duplicated again and again to describe a similar characteristic between two entities.*

*This post will discuss the difference between abstract and interface, along with an (awesome!!!) example – better than you’ve seen elsewhere.*  
 **Abstract class:**

* cannot be instantiated.
* is a special type of class in which you can have members without implementation.
* as we know in C#/VB/Java, a class can only inherit from 1 class. This also applies for Abstract Class.
* normally used for framework-type library classes: providing default behavior for some of its class members, but forcing the developer to implement others.
* [is believed to be faster in Java](http://mindprod.com/jgloss/interfacevsabstract.html), **HOWEVER** I cannot find the same claim in .Net. The speed difference is \*probably\* negligible and only relevant to the most academic field.
* the aim: making sure something is \*eventually\* implemented.
* A class can inherit an abstract class without implementing all its abstract method.  
  However only a class that has all its method implemented can be instantiated to an object.
* **IS-A** relationship.
* e.g. Student **IS A** Person, Employee **IS A** Person.

// 'framework library' for a person

// a person can enrol and submit

// however, the class that consume this framework library

// need to provide 'where' the paperwork need to be sent

public abstract Person

{

public abstract SendPaperWork(string paperwork)

public void Enrol()

{

SendPaperWork("enrolment");

}

public void Submit()

{

SendPaperWork("report");

}

}

// by inheriting Person abstract class

// we are enabling student to enrol and submit

// however, SendPaperWork need to be implemented

// because we need to tell it explicitly 'where'

// to send the enrolment/ submission

public class Student : Person

{

public override SendPaperWork(string paperwork)

{

School.Send(paperwork);

}

}

// an employee send the paperwork to a different 'place' than student

public class Employee : Person

{

public override SendPaperWork(string paperwork)

{

Company.Send(paperwork);

}

}

**Interface:**

* cannot be instantiated.
* is a special type of abstract class in which all the members do not have any implementations.
* enables [polymorphism](http://en.wikipedia.org/wiki/Polymorphism_(computer_science)). A class can implement more than 1 Interfaces.
* normally used for application classes: providing contract for ensuring interactibility.
* the aim: making sure something is interchangeable.
* A class that implements an Interface need to contain all the implementation, otherwise the compiler will throw an error.
* **CAN-DO** relationship.
* e.g. Student **CAN** enrol, Student **CAN** submit assignment.

public interface ICanEnrol

{

void Enrol();

}

public interface ICanSubmit

{

void Submit();

}

public class Student : ICanEnrol, ICanSubmit

{

public void Enrol()

{

School.Send("enrolment");

}

public void Submit()

{

School.Send("report");

}

}

public class Employee : ICanEnrol, ICanSubmit

{

public void Enrol()

{

Company.Send("enrolment");

}

public void Submit()

{

Company.Send("report");

}

}

public class MailServer

{

public void SendAllSubmissions()

{

// AllSubmitters is a collection of students and employees

foreach (ICanSubmit submitter in AllSubmitters)

{

// The MailServer does not care if

// the submitter is a student

// or an employee, as long as it can submit

submitter.Submit()

}

}

}

**[What is Method overloading?](http://question)**

Method overloading occurs when a class contains two methods with the same name, but different signatures.

**[What is method overloading?](http://question)**

Method overloading allows us to write different version of the same method in a class or derived class. Compiler automatically selects the most appropriate method based on the parameter supplied.

public class MultiplyNumbers

{  
 public int Multiply(int a, int b)

{

return a \* b;

}

public int Multiply(int a, int b, int c)

{

return a\*b\*c;

}

}

To call the above method, you can use following code.

MultiplyNumbers mn = new MultiplyNumbers();

int number = mn.Multiply(2, 3) // result = 6

int number1 = mn.Multiply(2, 3, 4) // result = 24  
  
You can't have a overload method with same number parameters but different return type. In order to create overload method, the return type must be the same and parameter type must be different or different in numbers.

**What is the use/advantage of function/method overloading?**

One of the key features of object-oriented programming is *polymorphism*. Polymorphism permits objects to behave in different ways according to the manner in which they are used. One part of polymorphism is the ability for a [method](http://www.blackwasp.co.uk/CSharpMethods.aspx) to behave differently according to the types and number of [parameters](http://www.blackwasp.co.uk/CSharpMethodParameters.aspx) that are passed to it. This is achieved through *method overloading*.

Method overloading allows the programmer to define many methods with the same name but with a different set of parameters. Each combination of parameter types is known as a *signature* of the method. When a call is made to one of these overloaded methods, the compiler automatically determines which of the methods should be used according to the arguments used in the call and the available method signatures.

One of the greatest advantages of method overloading is the improvement that it provides to code *readability and maintainability*. In languages that do not support this technique, or that of optional operands, a new method must be created for every possible combination of parameters. For example, in the ANSI C programming language to truncate a value you would use *trunc*, *truncf* or *truncl* according to the data type being [rounded](http://www.blackwasp.co.uk/RoundingDecimals.aspx). In C#, method overloading allows you to always call *Math.Truncate*. This becomes even more useful when a change in the requirements of the program means that data types change. Unlike with the older languages, the C# truncate method would require no code modification.

When using method overloading, each version of a method should perform the same general function using different data types or numbers of parameters. Although it is possible to create two methods with the same name that perform completely different tasks, this just reduces the quality of your code.

Overloading a method allows you to keep your interface consistent, and allows you to logically call the same method regardless of the types of data you are passing in. You will find that using the same name will help you remember what a procedure does, as opposed to having to come up with new names or a naming convention, to help you keep things straight.

**What is operator overloading?**

Operator overloading (a feature that was initially introduced in C++) is a concept that enables us to redefine the existing operators so that they can be used on user defined types like classes and structs.

*Declaring Operator overloading*

| public static <return Type> operator +(<ParamList, ...., .... >)  {  //Implementation pf your operator  } |
| --- |

**Why use Operator Overloading?**

Operator overloading makes a program clearer than accomplishing the same operations with explicit method calls. Now if you want to add tow complex numbers of Type *ComplexNumber* you just need to instantiate tow objects from it then do this:

| z = x + y; //x, y & z are ComplexNumber |
| --- |

But, if you don't have the operator overload for + operator what you should do to add them?? You have to repeat the implementation of the + operator several times, or put it in a method and call it several times.

**What is operator overloading? What are the advantages of operator overloading?**

It is one of the features of Object oriented programming which gives an **extra ability** to an **operator** to act on a **User-defined operand(Objects)**.

*Advantages/Uses of Operator Overloading:*

* Extensability: An operator will act differently depending on the operands provided.
* Operator is not limited to work only with primitive Data Type.
* Makes code much more readable.

**What is Method Overriding? How to override a function in C#?**

Method overriding is a feature that allows you to invoke functions (that have the same signatures) that belong to different classes in the same hierarchy of inheritance using the base class reference. C# makes use of two keywords: virtual and overrides to accomplish Method overriding. Let's understand this through small examples.

P1.cs

class BC

{

public void Display()

{

System.Console.WriteLine("BC::Display");

}

}

class DC : BC

{

new public void Display()

{

System.Console.WriteLine("DC::Display");

}

}

class Demo

{

public static void

Main()

{

BC b;

b = new BC();

b.Display();

}

}

Output : BC::Display

Use the override modifier to modify a method, a property, an indexer, or an event. An override method provides a new implementation of a member inherited from a base class. The method overridden by an override declaration is known as the overridden base method. The overridden base method must have the same signature as the override method.  
You cannot override a non-virtual or static method. The overridden base method must be virtual, abstract, or override.

**[In which cases you use override and new base?](http://question)**

Use the new modifier to explicitly hide a member inherited from a base class. To hide an inherited member, declare it in the derived class using the same name, and modify it with the new modifier.

**[What is Method Overriding? How to override a function in C#?](http://question)**

Method overriding is a feature that allows to invoke functions (that have the same signatures) and that belong to different classes in the same hierarchy of inheritance using the base class reference. In C# it is done using keywords **virtual** and **overrides**.

Use the override modifier to modify a method, a property, an indexer, or an event. An override method provides a new implementation of a member inherited from a base class. The method overridden by an override declaration is known as the overridden base method. The overridden base method must have the same signature as the override method.   
  
You cannot override a non-virtual or static method. The overridden base method must be virtual, abstract, or override.

**What is a sealed class?**

It is a class, which cannot be sub-classed. It is a good practice to mark your classes as sealed, if you do not intend them to be sub-classed. An error occurs if a sealed class is specified as the base class of another class.

*Some points to remember:*

* A class, which restricts inheritance for security reason is declared, sealed class.
* Sealed class is the last class in the hierarchy.
* Sealed class can be a derived class but can't be a base class.
* A sealed class cannot also be an abstract class. Because abstract class has to provide functionality and here we are restricting it to inherit.

**How do you prevent a class from being inherited?**

Mark it as sealed.

**Can you inherit from multiple base classes in C#?**

No. C# does not support multiple inheritances, so you cannot inherit from more than one base class. You can however, implement multiple interfaces.

**Explain the implementation phase with respect to OOP?**

The design phase is followed by OOP, which is the implementation phase. OOP provides specifications for writing programs in a programming language. During the implementation phase, programming is done as per the requirements gathered during the analysis and design phases.

**Give an example for object based programming language.**  
Ada, VB, etc are examples for object based programming language.

**What are the steps involved in message passing?**

The following are the steps involved in message passing:  
Creating classes, creating objects, and creating communication between objects.

**Explain about message passing in object oriented programming?**

Message passing is a method by which an object sends data to another object or requests other object to invoke method. This is also known as interfacing. It acts like a messenger from one object to other object to convey specific instructions.

**What is a class?**  
Class is an entity which consists of member data and member functions which operate on the member data bound together.

*A class is a template definition of the methods and variables for a particular kind of object. In other words, class is the blue print from which individual objects are created.*

**What is an object?**  
Objects are instances of classes. Class is a collection of similar kind of objects. When a class is created it doesn’t occupy any memory, but when instances of class is created i.e., when objects are created they occupy memory space.

Basically, *an object is anything that is identifiable as a single material item.* You can see around and find many objects like Camera, Monitor, Laptop, etc. In OOP perspective, an object is nothing but an instance of a class that contains real values instead of variables.

**What is the difference between class, object and instance? Explain with real example. Is a telephone receiver set an object? If it is so, why?**

In OO Programming, we often hear of terms like “Class”, “Object” and “Instance”; but what actually is a Class / Object / Instance?

In short, an object is a software bundle of related state and behavior. A class is a blueprint or prototype from which objects are created. An instance is a single and unique unit of a class.

Example, we have a blueprint (class) represents student (object) with fields like name, age, course (class member). And we have 2 students here, Foo and Bob. So, Foo and Bob is 2 different instances of the class (Student class) that represent object (Student people).

Let me go into details…

*Object*Real world objects shares 2 main characteristics, *state and behavior*. Human have state (name, age) and behavior (running, sleeping). Car have state (current speed, current gear) and state (applying brake, changing gear). Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. *An object stores its state in fields and exposes its behavior through methods.*

Telephone receiver set is an object because it has behaviors like MakeCall(), ReceiveCall(), etc and attributes like color, size, etc.

*Class*Class is a “template” / “blueprint” that is used to create objects. Basically, a class will consist of field, static field, method, static method and constructor. Field is used to hold the state of the class (eg: name of Student object). Method is used to represent the behavior of the class (eg: how a Student object going to stand-up). Constructor is used to create a new Instance of the Class.

*Instance*An instance is a unique copy of a Class that representing an Object. When a new instance of a class is created, the CLR will allocate a room of memory for that class instance.

**How class and object are related to each other?**

An object is the instantiation of the class. Class is a collection of similar kind of objects. The class must have been created before one can create an object. Each object consists of:

* + identity (e.g. name)
  + state (variables, current conditions, information/data)
  + methods (operations, actions/processes that change state)

**What is an Abstract Class?**

A class that cannot be instantiated. An abstract class is a class that must be inherited and have the methods overridden. An abstract class is essentially a blueprint for a class without any implementation.

**When do you absolutely have to declare a class as abstract?**

1. When the class itself is inherited from an abstract class, but not all base abstract methods have been overridden.

2. When at least one of the methods in the class is abstract.

**What is an interface class?**

Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation. They are implemented by classes, and defined as separate entities from classes.

**Can you inherit multiple interfaces?**

Yes. .NET does support multiple interfaces.

**What happens if you inherit multiple interfaces and they have conflicting method names?**

It’s up to you to implement the method inside your own class, so implementation is left entirely up to you. This might cause a problem on a higher-level scale if similarly named methods from different interfaces expect different data, but as far as compiler cares you’re okay.

**What is the difference between a Class and an Interface?**

In .Net/ C# a *class* can be defined to implement an *interface* and also it supports multiple implementations. When a *class* implements an *interface*, an *object* of such *class* can be encapsulated inside an *interface*.

If *MyLogger* is a class, which implements *ILogger,* there we can write

ILogger log = new MyLogger();

A *class* and an *interface* are two different types (conceptually). *Theoretically a* class *emphasis the idea of encapsulation, while an* interface *emphasis the idea of abstraction (by suppressing the details of the implementation).* The two poses a clear separation from one to another. Therefore it is very difficult or rather impossible to have an effective meaningful comparison between the two, but it is very useful and also meaningful to have a comparison between an interface and an abstract class.

**What’s the difference between an interface and abstract class?**

In an interface class, all methods are abstract - there is no implementation. In an abstract class some methods can be concrete. In an interface class, no accessibility modifiers are allowed. An abstract class may have accessibility modifiers.

| Interfaces vs. Abstract Classes |  |  |
| --- | --- | --- |
| **Feature** | **Interface** | **Abstract class** |
| Multiple inheritance | A class may implement several interfaces. | A class may extend only one abstract class. |
| Default implementation | An interface cannot provide any code at all, much less default code. | An abstract class can provide complete code, default code, and/or just stubs that have to be overridden. |
| Constants | Static final constants only, can use them without qualification in classes that implement the interface. On the other paw, these unqualified names pollute the namespace. You can use them and it is not obvious where they are coming from since the qualification is optional. | Both instance and static constants are possible. Both static and instance intialiser code are also possible to compute the constants. |
| Third party convenience | An interface implementation may be added to any existing third party class. | A third party class must be rewritten to extend only from the abstract class. |
| is-a vs -able or can-do | Interfaces are often used to describe the peripheral abilities of a class, not its central identity, e.g. an Automobile class might implement the Recyclable interface, which could apply to many otherwise totally unrelated objects. | An abstract class defines the core identity of its descendants. If you defined a Dog abstract class then Damamation descendants are Dogs, they are not merely dogable. Implemented interfaces enumerate the general things a class can do, not the things a class is. |
| Plug-in | You can write a new replacement module for an interface that contains not one stick of code in common with the existing implementations. When you implement the interface, you start from scratch without any default implementation. You have to obtain your tools from other classes; nothing comes with the interface other than a few constants. This gives you freedom to implement a radically different internal design. | You must use the abstract class as-is for the code base, with all its attendant baggage, good or bad. The abstract class author has imposed structure on you. Depending on the cleverness of the author of the abstract class, this may be good or bad. Another issue that's important is what I call "heterogeneous vs. homogeneous." If implementors/subclasses are homogeneous, tend towards an abstract base class. If they are heterogeneous, use an interface. (Now all I have to do is come up with a good definition of hetero/homogeneous in this context.) If the various objects are all of-a-kind, and share a common state and behavior, then tend towards a common base class. If all they share is a set of method signatures, then tend towards an interface. |
| Homogeneity | If all the various implementations share is the method signatures, then an interface works best. | If the various implementations are all of a kind and share a common status and behavior, usually an abstract class works best. |
| Maintenance | If your client code talks only in terms of an interface, you can easily change the concrete implementation behind it, using a factory method. | Just like an interface, if your client code talks only in terms of an abstract class, you can easily change the concrete implementation behind it, using a factory method. |
| Speed | Slow, requires extra indirection to find the corresponding method in the actual class. Modern JVMs are discovering ways to reduce this speed penalty. | Fast |
| Terseness | The constant declarations in an interface are all presumed public static final, so you may leave that part out. You can't call any methods to compute the initial values of your constants. You need not declare individual methods of an interface abstract. They are all presumed so. | You can put shared code into an abstract class, where you cannot into an interface. If interfaces want to share code, you will have to write other bubblegum to arrange that. You may use methods to compute the initial values of your constants and variables, both instance and static. You must declare all the individual methods of an abstract class abstract. |
| Adding functionality | If you add a new method to an interface, you must track down all implementations of that interface in the universe and provide them with a concrete implementation of that method. | If you add a new method to an abstract class, you have the option of providing a default implementation of it. Then all existing code will continue to work without change. |

**In which Scenario you will go for Interface or Abstract Class?**

The choice of whether to design your functionality as an interface or an abstract class can sometimes be a difficult one. An *abstract class* is a class that cannot be instantiated, but must be inherited from. An abstract class may be fully implemented, but is more usually partially implemented or not implemented at all, thereby *encapsulating common functionality* for inherited classes.

An *interface*, by contrast, is a totally abstract set of members that can be thought of as defining a contract for conduct. The implementation of an interface is left completely to the developer.

The answer isn't always clear-cut. Use abstract classes and interfaces in combination to optimize your design trade-offs. Here are some guidelines that might help you:

• **IS-A vs. CAN-DO relationship** A type can inherit only one implementation. If the

derived type can't claim an IS-A relationship with the base type, don't use a base type;

use an interface. Interfaces imply a CAN-DO relationship. If the CAN-DO functionality

appears to belong with various object types, use an interface. For example, a type can

convert instances of itself to another type (**IConvertible**), a type can serialize an

instance of itself (**ISerializable**) , etc. Note that value types must be derived from

**System.ValueType**, and therefore, they cannot be derived from an arbitrary base class.

In this case, you must use a CAN-DO relationship and define an interface. .NET value types can implement interfaces. Thus, primitives—such as Int32— can implement the [IComparable](http://icomparable) interface, for example, making them comparable.

*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.*

*Examples from FCL:*

In the FCL, the classes related to streaming data use an implementation inheritance design.

The **System.IO.Stream** class is the abstract base class. It provides a bunch of methods, such

as **Read** and **Write.** Other classes—**System.IO.FileStream, System.IO.MemoryStream,**

and **System.Net.Sockets.NetworkStream**—are derived from **Stream**. Microsoft chose an IS-A

relationship between each of these three classes and the **Stream** class because it made implementing

the concrete classes easier. For example, the derived classes need to implement only synchronous I/O operations; they inherit the ability to perform asynchronous I/O operations from the **Stream** base class.

Admittedly, choosing to use inheritance for the stream classes isn't entirely clear-cut; the **Stream** base class actually provides very little implementation. However, if you consider the Microsoft Windows Forms control classes, in which **Button, CheckBox, ListBox,** and all of the other controls are derived from **System.Windows.Forms.Control**, it's easy to imagine all of the code that **Control** implements, which the various control classes simply inherit to function correctly.

By contrast, Microsoft designed the FCL collections to be interface based. The **System.Collections.Generic** namespace defines several collection-related interfaces:

**IEnumerable<T>, ICol lect ion<T>, IList<T>,** and **IDictionary<TKey, TValue>.**

Then Microsoft provided a number of classes, such as **List<T>, Dictionary<TKey, TValue>,**

**Queue<T>, Stack<T>,** and so on, that implement combinations of these interfaces. Here the

designers chose a CAN-DO relationship between the classes and the interfaces because the

implementations of these various collection classes are radically different from one another.

In other words, there isn't a lot of sharable code between a **List<T>,** a **Dictionary<TKey,**

**TValue>,** and a **Queue<T>.**

* **Design flexibility and Ease of use**

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. In case of Abstract class we can define COMMON functionalities in super class and those can be used in the derived class where as in Interface we can’t do that. ( this i would say as advantage of abstract class). This helps to reuse code. It's generally easier for you as a developer to define a new type derived from a base type than to implement all of the methods of an interface. The base type can provide a lot of functionality, so the derived type probably needs only relatively small modifications to its behavior. If you supply an interface, the new type must implement all of the members.

When creating a standalone project which can be changed at will, use an interface in preference to an abstract class; because, it offers more design flexibility. Interfaces offer more design flexibility; precisely because, they can be implemented by any class regardless of its type hierarchy. In case of Interface the derived class can implement any number of interface but restricted to extend only one abstract class (this i would say as advantage of Interface)

*Example*

The **System.IO.Stream** class is the abstract base class. It provides a bunch of methods, such

as **Read** and **Write.** Other classes—**System.IO.FileStream, System.IO.MemoryStream,**

and **System.Net.Sockets.NetworkStream**—are derived from **Stream**. The subclasses make use of the common methods available in superclass.

• **Versioning/Adding functionality** If you add a method to the base type, the derived type inherits the new method's default implementation for free. If additional functionality is needed in derived classes, it can be added to the base class without breaking code. This simplifies versioning. An abstract class can be extended by adding new non-abstract methods with default implementations. Also, a [convenience method](http://method) is easily added to an abstract class.

In fact, the user's source code doesn't even have to be recompiled. Adding a new member to an interface forces the inheritor of the interface to change its source code and recompile. This means that an interface cannot be modified without breaking its contract with the classes which implement it. Once an interface has been shipped, its member set is permanently fixed. An API based on interfaces can only be extended by adding new interfaces.

If interfaces are poorly designed and have to change or not easy to implement we will not receive the benefits of using them and all they achieve is to make our code more complex. We need to think out the interfaces very carefully at design time before getting too far into the implementation because once they are implemented and made available to other resources they should not change.

*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.*

*Example*

If we consider interface IDisposable from FCL, it has Dispose() method. The client of this interface implements this interface in .NET 2.0. Now, in case, it is thought to provide extra capabilities to this interface in future versions of .NET. If a new member is introduced, the existing client code implementing this interface would fail. The client code has to implement all members of this changed interface. This also requires code to be recompiled. The solution to this problem would be to create a new interface say IDisposable\_v2.

IDisposable\_v2: IDisposable

{

//new member

}

This would ensure that existing client code does not break. But this kind of solution is difficult to maintain with multiple versions of interface. This is a drawback with interface design. Extra care should be taken while designing interface to ensure interface is not changed in future. If there is a small chance that interface would be required to be changed, go with abstract class.

• **Consistent implementation** No matter how well an interface contract is documented,

it's very unlikely that everyone will implement the contract 100 percent correctly. In

fact, COM suffers from this very problem, which is why some COM objects work correctly

only with Microsoft Office Word or with Microsoft Internet Explorer. By providing

a base type with a good default implementation, you start off using a type that works

and is well tested; you can then modify parts that need modification.

**Can we create a Java/C# class inside an interface?**

Yes, classes can be declared inside interfaces. This technique is sometimes used where the class is a constant type, return value or method argument in the interface. When a class is closely associated with the use of an interface it is convenient to declare it in the same compilation unit. This proximity also helps ensure that implementation changes to either are mutually compatible.

A class defined inside an interface is implicitly public static and operates as a top level class. The static modifier does not have the same effect on a nested class as it does with class variables and methods. The example below shows the definition of a StoreProcessor interface with nested StorageUnit class which is used in the two interface methods.

**Can an interface extend an abstract class?**

*In Java/C# an interface cannot extend an abstract class. An interface may only extend a super-interface.* And an abstract class may implement an interface. It may help to think of interfaces and classes as separate lines of inheritance that only come together when a class implements an interface, the relationship cannot be reversed.

**What is Marker Interface in .Net? How can I implement in Dot Net? What is the exact use of this? If possible please explain with an example.**

Marker interfaces as the name suggests, doesn’t have anything in it, just used to mark a class.   
  
If you have worked with asp.net, you might have used IRequiredSessionState. This has no member, just acts as a Marker interface.   
  
In .NET we use Marker Interface using Attributes.

**Is it recommended to use Marker Interface?**

CONSIDER defining an interface if you need to support its functionality on types that already inherit from some other type.

AVOID using marker interfaces (interfaces with no members).

If you need to mark a class as having a specific characteristic (marker), in general, use a custom attribute rather than an interface.

*// Avoid*

public interface IImmutable {} *// empty interface*

public class Key: IImmutable {

...

}

*//Do*

[Immutable]

public class Key {

...

}

Methods can be implemented to reject parameters that are not marked with a specific attribute as follows:

public void Add(Key key, object value){

if(!key.GetType().IsDefined(typeof(ImmutableAttribute), false)){

throw new ArgumentException("The parameter must be immutable","key");

}

}

| **RICO MARIANI** |
| --- |
| Of course any kind of marking like this has a cost. Attribute testing is a lot more costly than type checking. You might find that it's necessary to use the marker interface approach for performance reasons—measure and see. My own experience is that true markers (with no members) don't come up very often. Most of the time, you need a no- kidding-around interface with actual functionality to do the job, in which case there is no choice to make. |

The problem with this approach is that the check for the custom attribute can occur only at runtime. Sometimes, it is very important that the check for the marker be done at compile-time. For example, a method that can serialize objects of any type might be more concerned with verifying the presence of the marker than with type verification at compile-time. Using marker interfaces might be acceptable in such situations. The following example illustrates this design approach:

public interface ITextSerializable {} *// empty interface*

public void Serialize(ITextSerializable item){

*// use reflection to serialize all public properties*

...

}

**DO** provide at least one type that is an implementation of an interface.

This helps to validate the design of the interface. For example, System.Collections.ArrayList is an implementation of the System.Collections.IList interface.

**DO** provide at least one API consuming each interface you define (a method taking the interface as a parameter or a property typed as the interface).

This helps to validate the interface design. For example, List<T>.Sort consumes IComparer<T> interface.

**DO NOT** add members to an interface that has previously shipped.

Doing so would break implementations of the interface. You should create a new interface to avoid versioning problems.

Except for the situations described in these guidelines, you should, in general, choose classes rather than interfaces in designing managed code reusable libraries.

**How we can use a class defined inside an interface?**   
**or Why we define a class inside an interface?**

A class is defined inside an interface to bind the interface to a TYPE.

A small but nonsense example:

interface employee{   
    class Role{   
          public String rollname;   
          public int Role id;   
          public Object person;   
     }   
    Role getRole();   
    // other methods   
}

In the above interface you are binding the Role type strongly to the employee interface (employee.Role).

Another usage than those linked by Eric P: defining a default/no-op implementation of the interface.

interface IEmployee

{

    void workHard ();

    void procrastinate ();

    class DefaultEmployee implements IEmployee

    {

        void workHard () { procrastinate(); };

        void procrastinate () {};

    }

}

Yet, another sample — implementation of Null Object Pattern:

interface IFoo

{

    void doFoo();

    IFoo NULL\_FOO = new NullFoo();

    final class NullFoo implements IFoo

    {

        public void doFoo () {};

        private NullFoo ()      {};

    }

}

...

IFoo foo = IFoo.NULL\_FOO;

...

bar.addFooListener (foo);

...

**Which of the following are true about the class defined inside an interface**1. it is not possible in the java Laungage.  
2. The class is always public.  
3. The class is always static.  
4. the class methods cannot call the methods declared in the interface.  
5. the class methods can call only the static methods declared in the interface.

You should know this first regarding nested classes in an interface:  
  
You can put the definition of a class inside the definition of an interface. The calss will be an inner class to the interface. An inner class to an interface will be static and public by default. The code structure would be like this :

**interface** Port {

//Methods & constants declared in the interface...

**class** Info {

//Definition of the class...

}

}

This declares the interface Port with an inner class Info. Objects of the inner class would be of type Port.Info. You might create one with a statement like this:  
  
Port.Info info = new Port.Info();  
  
A class that implements the inteface would have no direct connection with the inner class to the interface- it would just need to impleemnts the methods declared by the interface, but it is highly likely it would make use of objects of the inner class.  
  
On behalf of above description: Answer 2,3,4 are correct.  
5 is incorrect because interface methods can't be static.

**List the areas of applications of object oriented programming?**

The following are few areas of applications of object oriented programming:  
CAD/CAM systems  
Office automation and decision support systems  
Object oriented databases  
Real time systems  
Simulation and modeling.  
  
**How does a class provide data hiding?**  
Data hiding is provided by a class by the use of visibility label private which allows only the member functions to access the data declared as private.

**What does member functions represent?**  
Member functions of a class represent the behavior of a class.

**[Can we call a base class method without creating instance?](http://question)**

Yep. But ..   
  
\* Its possible If its a static method.   
  
\* Its possible by inheriting from that class also.   
  
\* Its possible from derived classes using base keyword.

**Default Access modifiers in C#?**

An **enum** has default modifier as ***public***   
  
A **class** has default modifiers as ***private***. It can declare members (methods etc) with following access modifiers:   
public   
protected   
internal   
private   
protected internal   
  
An **interface** has default modifier as ***public***   
  
A **struct** has default modifier as ***private*** and it can declare its members (methods etc) with following access modifiers:   
public   
internal   
private

**[What is Protected Internal access modifier in C#?](http://question)**

Protected Internal is a access modifiers for the members (methods or functions) ie. you can't declare a class as protected internal explicitly. The members access is limited to the current assembly or types derived from the containing class.   
  
Protected Internal means the method is accessible by anything that can access the protected method UNION with anything that can access the internal method.

**[What is Internal access modifier in C#?](http://question)**

The internal keyword is an access modifier for types and type members ie. we can declare a class as internal or its member as internal. Internal members are accessible only within files in the same assembly (.dll). In other words, access is limited exclusively to classes defined within the current project assembly.   
  
For more details see http://msdn.microsoft.com/en-us/library/7c5ka91b(VS.71).aspx

**[What is Private access modifier in C#?](http://question)**

The private keyword is a member access modifier ie. we can't explicitly declare a class as Private, however if do not specify any access modifier to the class, its scope will be assumed as Private. Private access is the least permissive access level of all access modifiers.   
  
Private members are accessible only within the body of the class or the struct in which they are declared. This is the default access modifier for the class declaration.

**[What is Public access modifier in C#?](http://question)**

The public keyword is an access modifier for types and type members ie. we can declare a class or its member (functions or methods) as Public. There are no restrictions on accessing public members.

**[What is Protected access modifier in C#?](http://question)**

The protected keyword is a member access modifier. It can only be used in a declaring a function or method not in the class ie. a class can't be declared as protected class.   
  
A protected member is accessible from within the class in which it is declared, and from within any class derived from the class that declare this member. In other words access is limited to within the class definition and any class that inherits from the class   
  
A protected member of a base class is accessible in a derived class only if the access takes place through the derived class type.

**[Can we specify the access modifier for explicitly implemented interface method?](http://question)**

No, we can't specify the access modifier for the explicitly implemented interface method. By default its scope will be internal.

**[What is pure virtual function?](http://question)**

When you define only function prototype in a base class without and do the complete implementation in derived class. This base class is called abstract class and client won’t able to instantiate an object using this base class.   
  
A pure virtual function is a function that must be overridden in a derived class and need not be defined. A virtual function is declared to be "pure" using the curious "=0"   
syntax:   
class Base {   
public:   
void f1(); // not virtual   
virtual void f2(); // virtual, not pure   
virtual void f3() = 0; // pure virtual   
};

**[What is an Interface?](http://question)**

An interface is a contract & defines the requisite behavior of generalization of types.   
  
An interface mandates a set of behavior, but not the implementation. Interface must be inherited. We can't create an instance of an interface.   
  
An interface is an array of related function that must be implemented in derived type. Members of an interface are implicitly public & abstract.   
  
An interface can inherit from another interface.

**[What is sealed modifiers?](http://question)**

Sealed types cannot be inherited & are concrete.   
Sealed modifiers can also be applied to instance methods, properties, events & indexes. It can't be applied to static members.   
  
Sealed members are allowed in sealed and non-sealed classes.

**[What is Abstract Class?](http://question)**

Abstract class exists extensively for inheritance. We can't create an instance of an abstract class. Abstract type must be inherited.   
  
Static, Value Types & interface doesn't support abstract modifiers.   
  
Static members cannot be abstract. Classes with abstract member must also be abstract.

**[Can Struct be inherited?](http://question)**

No, Struct can't be inherited as this is implicitly sealed.

**[What is Virtual method?](http://question)**

Virtual Method has implementation & provide the derived class with the option to override it.

**[What is Abstract method?](http://question)**

Abstract method doesn't

**?**The anlaysis or the object oriented analysis phase considers the system as a solution to a problem in its environment or domain. Developer concentrates on obtaining as much information as possible about the problem. Critical requirements needs to be identified.

**Explain about the Design Phase?**In the design phase, the developers of the system document their understanding of the system. Design generates the blue print of the system that is to be implemented. The first step in creating an object oriented design is the identification of classes and their relationships.

The Object-Oriented Paradigm provide the implementation & forces the derived class to override the method.

**Explain about the analysis phase**

**Observations**

**Describe the basic approach used in functional decomposition.**

Functional decomposition is the approach to analysis that breaks down (decomposes) a problem into its functional parts without too much concern for global requirements and future modifications.

**What are three reasons that cause requirements to change?**

The user's understanding of what they need and what is possible grows and changes as they discuss the problem with analysts. The developer's understanding of what is possible and what is needed evolves as they become familiar with the domain and with the software. The technical environment evolves, forcing changes in how to implement.

**I advocate thinking about responsibilities rather than functions. What is meant by this? Give an example.**

Rather than thinking first about how something is done (functions), the analyst should focus on what the routine is responsible for doing - how it does it does not matter. The control program is much simpler in this case.

**Define "coupling" and "cohesion". What is "tight" coupling?**

Cohesion is how strongly the internal operations of a routine are related to each other. Coupling is how strongly a routine is dependent upon other routines.

**7. A class is a complete definition of the behavior of an object. What three aspects of an object does it describe?**

The three elements of a class are: the data elements, the methods, the interfaces (ways that data and methods can be accessed). (p. 17)

**10. Define encapsulation. Give one example of encapsulation of behavior.**

Any kind of hiding. Both data and behavior may be encapsulated. (p. 21)

**12. What are the three perspectives for looking at objects?**

*Conceptual*: the high-level concepts in a system (concepts, not software). At the conceptual level, an object is a set of responsibilities.

*Specification*: the interfaces between things in the software (software, not code). At the specification level, an object is a set of methods.

*Implementation*: how an individual routine works (code). At the implementation level, an object is code and data. (p. 13, 15-16)

**Interpretations**

**13. Sometimes, programmers use "modules" to isolate portions of code. Is this an effective way to deal with changes in requirements? Why or why not?**

Changes to one function or routine can have impacts on other routines. Usually, routines are not independent (p. 10).

**14. It is too limited to define an abstract class as a class that does not get instantiated. Why is this definition too limited? What is a better (or at least alternative) way to think about abstract classes?**

It is too limited because it only talks in terms of its implementation: what the abstract class does and how it is treated as software. It does not describe why I would want to use an abstract class: the motivation for it and how to think about it. It ignores the "conceptual perspective" of objects that analysts need to keep in mind as they work with users to understand problems. At the conceptual level, an abstract class is a placeholder for a set of classes. It gives a way to assign a name or label to a set of classes so that I can interact with them as a whole without getting trapped by the details. (p. 19)

**15. How does encapsulation of behavior help to limit the impact of changes in requirements? How does it save programmers from unintended side effects?**

It makes the control program much less complicated since it does not have to be responsible for as much. It limits the impact that changes to the internals of an object can have on the rest of the application. (p. 24)

**16. How do interfaces help to protect objects from changes that are made to other objects?**

Interfaces define the only ways that those external objects can communicate with the object. It protects me from side effects because I know what is coming into the system.

**17. A classroom is used to describe objects in a system. Describe this classroom from the conceptual perspective.**

The classroom contains students who are responsible for their own behaviors: how to move from here to there, how to go from class to class. It contains a teacher who tells students where to go.

**Opinions and Applications**

**1. Changing requirements is one of the greatest challenges faced by systems developers. Give one example from your own experience where this has been true.**

**2. There is a fundamental weakness in functional decomposition when it comes to changes in requirements. Do you agree? Why or why not?**

**3. What do you think is the best way to deal with changing requirements?**

**UML questions**

**What is UML?**

UML is Unified Modeling Language. It is a graphical language for *visualizing specifying* *constructing* and *documenting* the artifacts of the system. It allows you to create a blue print of all the aspects of the system, before actually physically implementing the system.

**What is modeling? What are the advantages of creating a model?**

Modeling is a proven and well-accepted engineering technique which helps build a model. Model is a simplification of reality; it is a blueprint of the actual system that needs to be built. Model helps to visualize the system. Model helps to specify the structural and behavior of the system. Model helps make templates for constructing the system. Model helps document the system.

**What are the different views that are considered when building an object-oriented software system?**

Normally there are 5 views.

*Use Case view* - This view exposes the requirements of a system.

*Design View* - Capturing the vocabulary.

*Process View* - modeling the distribution of the systems processes and threads.

*Implementation view* - addressing the physical implementation of the system.   
*Deployment view* - focus on the modeling the components required for deploying the system.

**What are diagrams?**

Diagrams are graphical representation of a set of elements most often shown made of things and associations.

**What are the major three types of modeling used?**

Major three types of modeling are *structural, behavioral*, and *architectural*.

**Mention the different kinds of modeling diagrams used?**

Modeling diagrams that are commonly used are, there are 9 of them. *Use case diagram, Class Diagram, Object Diagram, Sequence Diagram, state chart Diagram, Collaboration Diagram, Activity Diagram, Component diagram, Deployment Diagram.*

**What is SDLC?** SDLC is Software Development Life Cycle. SDLC of a system included processes that are Use case driven, Architecture centric and Iterative and Incremental. This Life cycle is divided into phases. Phase is a time span between two milestones. The milestones are Inception, Elaboration, Construction, and Transition. Process Workflows that evolve through these phase are Business Modeling, Requirement gathering, Analysis and Design, Implementation, Testing, Deployment. Supporting Workflows are Configuration and change management, Project management.

**What are Relationships?** There are different kinds of relationships: Dependencies, Generalization, and Association. Dependencies are relations ships between two entities that that a change in specification of one thing may affect another thing. Most commonly it is used to show that one class uses another class as an argument in the signature of the operation. Generalization is relationships specified in the class subclass scenario, it is shown when one entity inherits from other. Associations are structural relationships that are: a room has walls, Person works for a company. Aggregation is a type of association where there is a has a relationship, That is a room has walls, Ã±o if there are two classes room and walls then the relationship is called a association and further defined as an aggregation.

**How are the diagrams divided?**

The nine diagrams are divided into static diagrams and dynamic diagrams.

*Static Diagrams* (Also called Structural Diagram): Class diagram, Object diagram, Component Diagram, Deployment diagram.

*Dynamic Diagrams* (Also called Behavioral Diagrams): Use Case Diagram, Sequence Diagram, Collaboration Diagram, Activity diagram, State chart diagram.

**What are Messages?**

A message is the specification of a communication, when a message is passed that results in action that is in turn an executable statement.

**State some of the criticisms present with UML.**

Some of the disadvantages presented by UML are as follows  
1) Language bloat: - This language is very long in length and complex to understand. Infrequent and redundancy elements are present in UML which retards the learning of the subject.  
2) Confusion is overwhelming when using UML due to lack of proper abstract notation.

**What are the major three types of modeling used?**

Major three types of modeling are structural, behavioral, and architectural.

**How are the diagrams divided?**

The nine diagrams are divided into static diagrams and dynamic diagrams.

**Explain about dependency?**

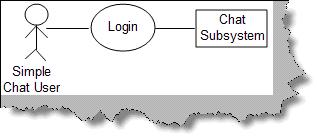
This form of relationship exists when a change to a certain element changes the definition and structure of the other element as well. This is indicated by a pointing arrow from the dependent side to the independent side.

**What is a Use Case?**

A use case specifies the behavior of a system or a part of a system, Ã³se cases are used to capture the behavior that need to be developed. It involves the interaction of actors and the system.

### Can you explain use case diagrams?

Use case diagram answers what system does from the user point of view. Use case answer ‘What will the system do?’. Use cases are mainly used in *requirement document* to depict clarity regarding a system. There are three important parts in a use case scenario, actor and use case.   
  
**Scenario: -** A scenario is a sequence of events which happen when a user interacts with the system.  
  
**Actor: -** Actor is who of the system, in other words the end user.   
  
**Use Case: -** Use case is task or the goal performed by the end user. Below figure ‘Use Case’ shows a simple scenario with ‘Actor’ and a ‘Use Case’. Scenario represents an accountant entering accounts data in the system. As use case’s represent action performed they are normally represented by strong verbs.  
  
Actor’s are represented by simple stick man and use case by oval shape as shown in figure ‘Use Case’ below.

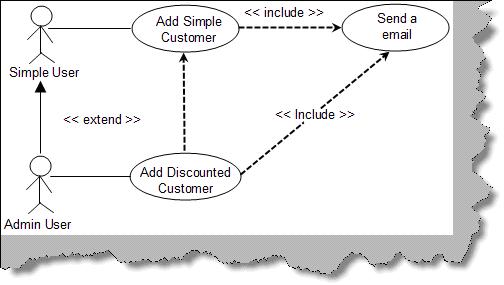


**Figure: - Use Case**

### Can you explain primary and secondary actors? Actors are further classified in to two types primary and secondary actors. Primary actors are the users who are the active participants and they initiate the user case, while secondary actors are those who only passively participate in the use case.

### How does a simple use case look like?

*Use case’s have two views of representation in any requirement document. One is the use case diagrams and the other is a detail step table about how the use case works.* So it’s like a pair first an over view is shown using a use case diagram and then a table explaining the same in detail. Below is a simple ‘login’ use case shown diagrammatically and then a detail table with steps about how the use case is executed.



**Figure: - Login Use Case**

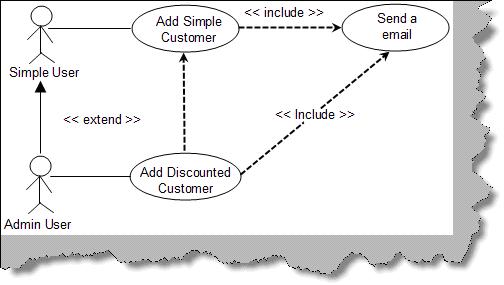
| **Use Case** | Rel001 |
| --- | --- |
| **Use Case Name** | Login |
| **Description** | This uses depicts the flow of how user will log-in into the chat application. |
| **Primary Actor** | Simple chat user. |
| **Trigger** | User types chat application on URL of the browser. |
| **Pre-condition** | NA |
| **Assumption** | No password is currently present for the system Rooms will remain constant as explained in the assumption section of this document |
| **Failed End conditions** | Duplicate user name is not allowed in the chat application. |
| **Action** | User clicks on the log-in button. |
| **Main Scenario** | • User types chat application on URL of the browser which in turn opens the main page. • In the main page of application user is popped up with ‘Enter user name’ option and various ‘rooms’ option drop down menu. • User then types the name and selects one of the room from drop down menu and then clicks on the ‘Log-in’ button. • Application then checks whether the user name is unique in the system if not then user is popped up with error message that “user already exist”. • After entering the unique name the user is finally logged in the application. |
| **Action** | NA |
| **Alternate Scenario** | NA |
| **Success Scenarios** | 1. Opens page of a selected room in that other user names and their messages can be seen. |
| **Note and Open Issues** | NA |

**Table: - Login use case table**

**Note**: - You must be wondering why we have this pair why not just a use case table only. Use case diagrams are good to show relationship between use case and they also provide high over view. The table explanation of a use case talks details about the use case. So when a developer or a user is reading a requirement document, he can get an overview by looking at the diagram if he is interested he can read the use case tables for more details.

### Can you explain ‘Extend’ and ‘Include’ in use cases?

‘Extend’ and ‘Include’ define relationships between use cases. Below figure ‘Extend and Include’ shows how these two fundamentals are implemented in a project. The below use case represents a system which is used to maintain customer. When a customer is added successfully it should send an email to the admin saying that a new customer is added. Only admin have rights to modify the customer. First lets define extend and include and then see how the same fits in this use case scenario.  
  
***Include:* -** Include relationship represents an invocation of one use case by the other. If you think from the coding perspective it’s like one function been called by the other function.  
  
***Extend:* -** This relationship signifies that the extending use case will work exactly like the base use case only that some new steps will inserted in the extended use case.  
  
Below figure ‘Extend and Include’ shows that ‘add customer’ is same as the ‘add discounted customer’. The ‘Add discounted customer’ has an extra process, to define discount for the discounted customer which is not available for the simple customer. One of the requirements of the project was that when we add a customer, the system should send an email. So after the customer is added either through ‘Add simple customer’ use case or ‘Add discounted customer’ use case it should invoke ‘send a email’ use case. So we have defined the same with a simple dotted line with <<include>> as the relationship.

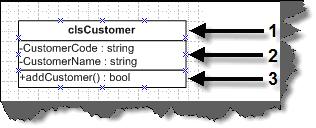


**Figure: - Extend and Include**

**Note**: - One of the points to be noted in the diagram ‘Extend and Include’ is we have defined inheritance relationship between simple and admin user. This also helps us defining a technical road map regarding relationships between simple and admin user.

### Can you explain class diagrams?

Class diagram  
  
Class is basically a prototype which helps us create objects. Class diagram defines the static structure of the project. A class represents family of an object. By using Class we can create uniform objects.  
  
In the below figure you can see how the class diagram looks. Basically there are three important sections which are numbered as shown in the below. Let’s try to understand according to the numbering:-  
• Class name:-This is the first section or top most section of the Class which represents the name of the Class (clsCustomer).  
• Attributes:-This is the second section or the middle section of the class which represents the properties of the system.  
• Methods: - This section carries operation or method to act on the attributes.

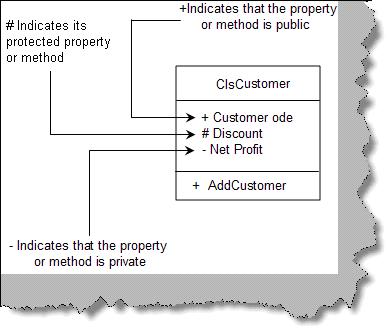


**Figure: - Three sections of the class**

Now, in the next section we will have a look on Association relationship between these classes.

### How do we represent private, public and protected in class diagrams?

In order to represent visibility for properties and methods in class diagram we need to place symbols next to each property and method as shown in figure ‘Private, Public and Protected’. ‘+’ indicates that it’s public properties/methods. ‘-‘indicates private properties which means it can not be accessed outside the class. ‘#’ indicate protected/friend properties. Protected properties can only be seen within the component and not outside the component.

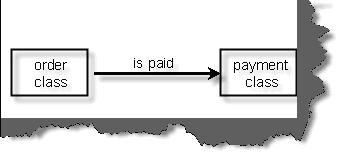


**Figure: - Private, public and protected**

### What does associations in a class diagram mean?

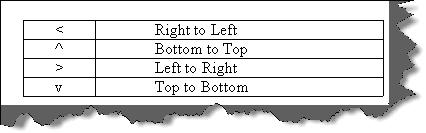
### Associations in Class diagrams

A single Class cannot represent the whole module in a project so we need one or more classes to represent a module. For instance, a module named ‘customer detail’ cannot be completed by the customer class alone, to complete the whole module we need customer class, address class, phone class in short there is relationship between the classes. So by grouping and relating between the classes we create module and these are termed as Association. In order to associate them we need to draw the arrowed lines between the classes as shown in the below figure.   
  
In the figure ‘Order is paid by payments class’, we can see Order class and the Payment class and arrowed line showing relationship that the order class is paid using payment class in other words order class is going to be used by payment class to pay the order. The left to right marked arrow basically shows the flow that order class uses the payment class.  
In case payment class using the order class then the marked arrow should be right to left showing the direction of the flow.



**Figure:- Order is paid by Payments class**

There are four signs showing the flow:-

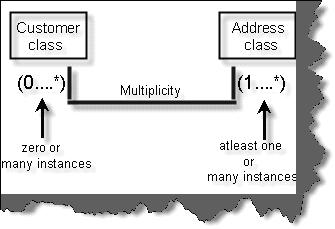


**Figure: - Direction signs in UML**

### Multiplicity

Multiplicity can be termed as classes having multiple associations or one class can be linked to instances of many other classes. If you look at the below figure the customer class is basically associated with the address class and also observes the notations (\*, 0 and 1).If you look at the right hand side the (1….\*) notation indicates that at least one or many instance of the address class can be present in the customer class. Now towards left hand side we have (0….\*) notation indicating that address class can exist without or many customer class can link him.  
In order to represent multiplicity of classes we have to show notations like (1….\*), (0….\*) as shown in below figure.

**Note**: ‘\*’ means “many” where as ‘(0, 1)’ means “(zero or at least one)” respectively.

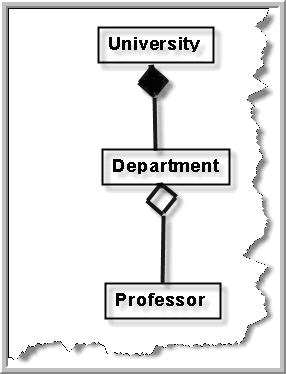


**Figure: - Multiplicity in Classes**

### Can you explain aggregation and composition in class diagrams?

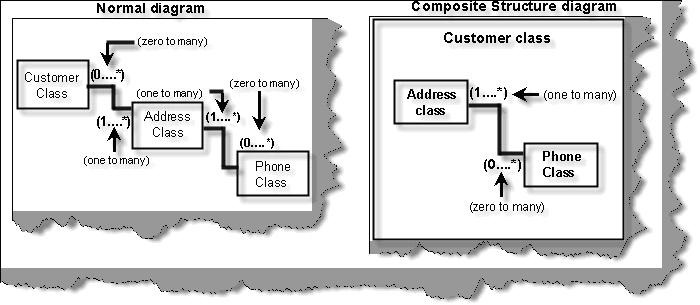
In this Association there are two types mainly Aggregation Association and Composition Association.  
  
**Aggregation** *Association signifies that the whole object can exist without the Aggregated   
Object.* For example in the below figure we have three classes university class, department class and the Professor Class. *The university cannot exist without department which means that university will be closed as the department is closed. In other words lifetime of the university depend on the lifetime of department.*  
In the same figure we have defined second Association between the department and the Professor. *In this case, if the professor leaves the department still the department continues in other words department is not dependent on the professor this is called as Composition Association.*

**Note**: - The filled diamond represents the aggregation and the empty diamond represents the composition. You can see the figure below for more details.

  
**Figure: - Aggregation and composition in action**

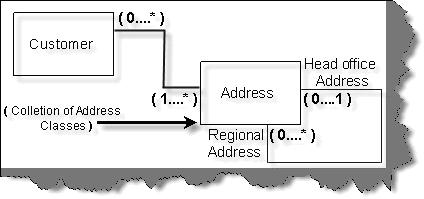
### What are composite structure diagram and reflexive association in class diagrams?

**Composite structure diagram** When we try to show Aggregation and Composition in a complete project the diagram becomes very complicated so in order to keep it simple we can use Composite structure diagram. In the below figure we have shown two diagrams one is normal diagram other is Composite structure diagram and the simplicity can easily be identified. In the composite diagram the aggregated classes are self contained in the main class which makes it simpler to read.



**Figure: - Composite Structure diagram**

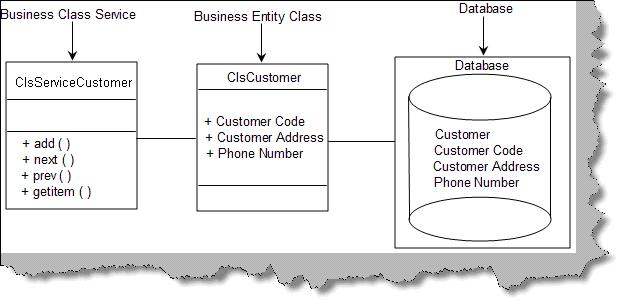
**Reflexive associations**  
In many scenarios you need to show that two instances of the same class are associated with each other and this scenario is termed as Reflexive Association. For instance in the below figure shows Reflexive Association in the real project. Here you can see customer class has multiple address class and addresses can be a Head office, corporate office or Regional office. One of the address objects is Head office and we have linked the address object to show Reflexive Association relationship. This is the way we can read the diagram Regional address object is blocked by zero or one instance of Head office object.



**Figure: - Reflexive association**

### Can you explain business entity and service class?

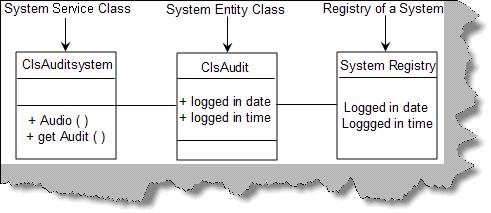
Business entity objects represent persistent information like tables of a database. Just making my point clearer they just represent data and do not have business validations as such. For instance below figure ‘Business entity and service’ shows a simple customer table which with three fields ‘Customer Code’,’ Customer Address’ and ‘Phone Number’. All these fields are properties in ‘ClsCustomer’ class. So ‘ClsCustomer’ class becomes the business entity class. The business entity class by itself can not do anything it’s just a place holder for data. In the same figure we have one more class ‘ClsServiceCustomer’. This class aggregates the business entity class and performs operations like ‘Add’,’ Next’ (Move to next record), ‘Prev’ (Move to previous record) and ‘GetItem’ (get a customer entity depending on condition).  
  
With this approach we have separated the data from the behavior. The service represents the behavior while the business entity represents the persistent data.



**Figure:-Business entity and service**

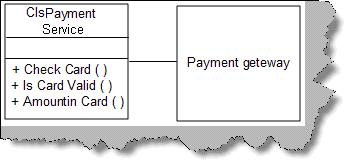
### Can you explain System entity and service class?

System entity class represents persistent information which is related to the system. For instance in the below figure ‘System entity and service class’ we have a system entity class which represents information about ‘loggedindate’ and ‘loggedintime’ of the system registry. System service class come in two flavors one is it acts like a wrapper in the system entity class to represent behavior for the persistent system entity data. In the figure you can see how the ‘ClsAudit’ system entity is wrapped by the ‘ClsAuditSytem’ class which is the system service class. ‘ClsAuditSystem’ adds ‘Audit’ and ‘GetAudit’ behavior to the ‘ClsAudit’ system entity class.



**Figure: - System entity and service class**

The other flavor of the system service class is to operate on non-persistent information. The first flavor operated on persistent information. For instance the below figure ‘Non-persistent information’ shows how the class ‘ClsPaymentService’ class operates on the payment gateway to Check is the card exists , Is the card valid and how much is the amount in the card ?. All these information are non-persistent. By separating the logic of non-persistent data in to a system service class we bring high reusability in the project.

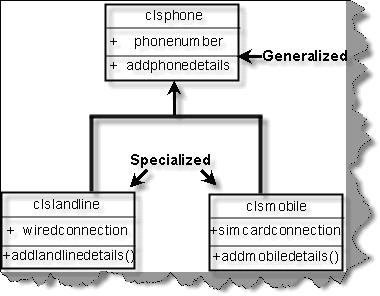


**Figure: - Non-persistent information**

Note: - The above question can be asked in interview from the perspective of how you have separated the behavior from the data. The question will normally come twisted like ‘How did you separate the behavior from the data?’.

### Can you explain generalization and specialization?

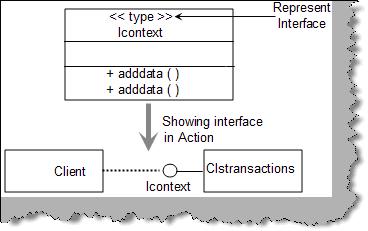
**Generalization and specialization:** In Generalization and Specialization we define the parent-child relationship between the classes. In many instance you will see some of the classes have same properties and operation these classes are called super class and later you can inherit from super class and make sub classes which have their own custom properties. In the below figure there are three classes to show Generalization and Specialization relationship. All phone types have phone number as a generalized property but depending upon landline or mobile you can have wired or simcard connectivity as specialized property. In this diagram the clsphone represent Generalization whereas clslandline and clsmobile represents specialization.



**Figure: - Generalization and Specialization**

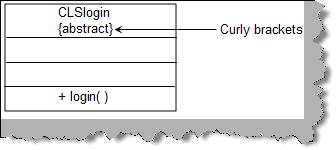
### How do we represent an abstract class and interface UML?

Interface is represented by <<type>> in the class diagram. Below figure ‘Interface in action’ shows we have defined an interface ‘IContext’. Note the ‘<<type>>’ represents an interface. If we want to show that the interface is used in a class we show the same with a line and a simple circle as shown in figure ‘Interface in Action’ below.



**Figure: - Interface in action**

Abstract classes are represented by ‘{abstract}’ as shown in figure ‘Abstract classes in action’.



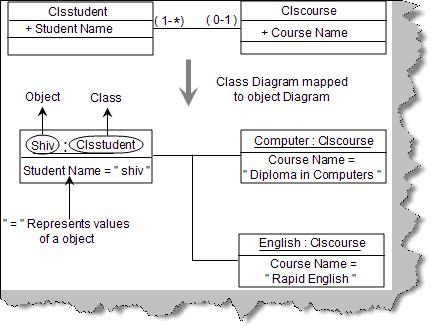
**Figure: - Abstract classes in action.**

### How do we achieve generalization and specialization?

By using inheritance.

### Can you explain object diagrams in UML?

Class represents shows the static nature of the system. From the previous question you can easily judge that class diagrams shows the types and how they are linked. Classes come to live only when objects are created from them. Object diagram gives a pictorial representation of class diagram at any point of time. Below figure ‘Object diagram’ shows how a class looks in when actual objects are created. We have shown a simple student and course relationship in the object diagram. So a student can take multiple courses. The class diagram shows the same with the multiplicity relationship. We have also shown how the class diagram then looks when the objects are created using the object diagram. We represent object with Object Name: Class Name. For instance in the below figure we have shown ‘Shiv : ClsStudent’ i.e ‘Shiv’ is the object and ‘ClsStudent’ the class. As the objects are created we also need to show data of the properties, the same is represented by ‘PropertyName=Value’ i.e. ‘StudentName=Shiv’.



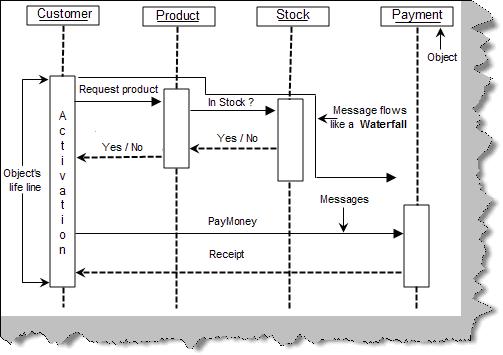
**Figure: - Object diagrams**

The diagram also states that ‘ClsStudent’ can apply for many courses. The same is represented in object diagram by showing two objects one of the ‘Computer’ and the other of ‘English’.

**Note**: - Object diagrams should only be drawn to represent complicated relationship between objects. It’s possible that it can also complicate your technical document as lot. So use it sparingly.

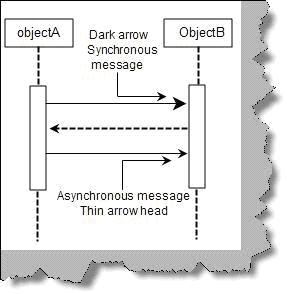
### Can you explain sequence diagrams?

**Sequence diagrams**  
*Sequence diagram shows interaction between objects over a specific period time.* Below figure 'Sequence diagram' shows how a sequence diagram looks like. In this sequence diagram we have four objects 'Customer','Product','Stock' and 'Payment'. The message flow is shown vertically in waterfall manner i.e. it starts from the top and flows to the bottom. Dashed lines represent the duration for which the object will be live. Horizontal rectangles on the dashed lines represent activation of the object. Messages sent from a object is represented by dark arrow and dark arrow head. Return message are represented by dotted arrow. So the figure shows the following sequence of interaction between the four objects:-  
  
• Customer object sends message to the product object to request if the product is available or not.  
• Product object sends message to the stock object to see if the product exists in the stock.  
• Stock object answers saying yes or No.  
• Product object sends the message to the customer object.  
• Customer object then sends a message to the payment object to pay money.  
• Payment object then answers with a receipt to the customer object.  
  
One of the points to be noted is product and stock object is not active when the payment activity occurs.



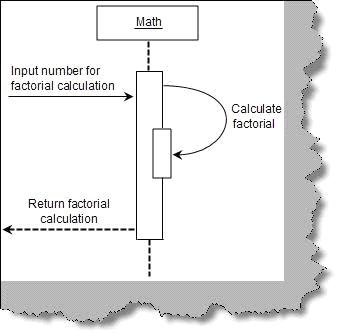
**Figure: - Sequence diagram**

**Messages in sequence diagrams**  
There are five different kinds of messages which can be represented by sequence   
**Synchronous and asynchronous messages:-**  
Synchronous messages are represented by a dark arrow head while asynchronous messages are shown by a thin arrow head as shown in figure ‘Synchronous and Asynchronous’.



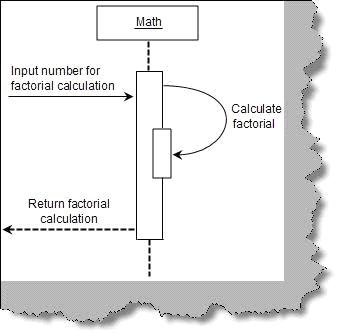
**Figure: - Synchronous and Asynchronous**

**Recursive message:-**  
We have scenarios where we need to represent function and subroutines which are called recursively. Recursive means the method calling himself. Recursive messages are represented by small rectangle inside a big rectangle with an arrow going from the big rectangle to the small rectangle as shown in figure ‘Recursive message’.



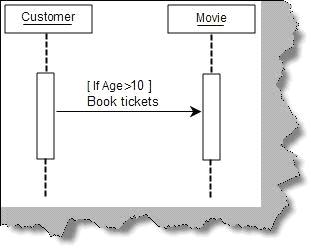
**Figure: - Recursive message**

**Message iteration:-**  
  
Message iteration represents loops during sequences of activity. Below figure ‘message iteration’ shows how ‘order’ calls the ‘orderitem’ objects in a loop to get cost. To represent loop we need to write ‘For each <<object name>>’. In the below figure the object is the ‘orderitem’. Also note the for each is put in a box to emphasize that it’s a loop.



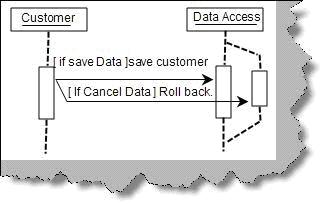
**Figure: - Message iteration**

**Message constraint:-**  
If we want to represent constraints it is put in a rectangle bracket as shown in figure ‘message constraint’. In the below figure ‘message constraint’ the ‘customer’ object can call ‘book tickets’ only if the age of the customer is greater than 10.



**Figure: - Message constraint**

**Message branching:-**  
Below figure ‘message branching’ shows how ‘customer’ object have two branches one is when the customer calls save data and one when he cancels the data.

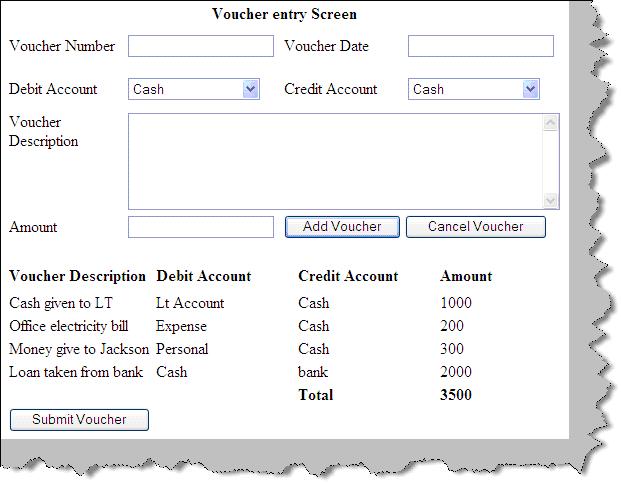


**Figure: - Message branching**

**Doing Sequence diagram practically**  
Let’s take a small example to understand sequence diagram practically. Below is a simple voucher entry screen for accounts data entry. Following are the steps how the accountant will do data entry for the voucher:-

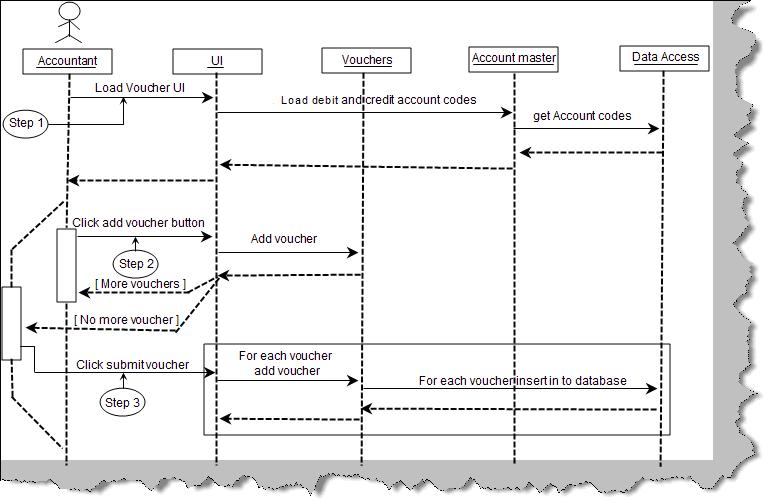
* Accountant loads the voucher data entry screen. Voucher screen loads with debit account codes and credit account codes in the respective combo boxes.
* Accountant will then fill in all details of the voucher like voucher description, date, debit account code, credit account code, description, and amount and then click ‘add voucher’ button.
* Once ‘add voucher’ is clicked it will appear in the voucher screen below in a grid and the voucher entry screen will be cleared and waiting for new voucher to be added. During this step voucher is not added to database it’s only in the collection.
* If there are more vouchers to be added the user again fills voucher and clicks ‘add voucher’.
* Once all the vouchers are added he clicks ‘submit voucher’ which finally adds the group of vouchers to the database.

Below figure ‘Voucher data entry screen’ shows pictorially how the screen looks like.



**Figure: - Voucher data entry screen**

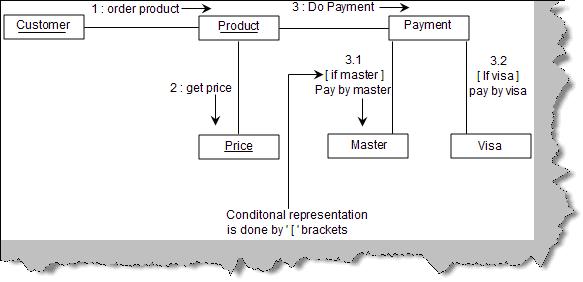
Figure ‘Voucher data entry sequence diagram’ shows how the sequence diagram looks like. Below diagram shows a full sequence diagram view of how the flow of the above screen will flow from the user interface to the data access layer. There are three main steps in the sequence diagram, let’s understand the same step by step.  
  
**Step 1:-** The accountant loads the voucher data entry screen. You can see from the voucher data entry screen image we have two combo boxes debit and credit account codes which are loaded by the UI. So the UI calls the ‘Account Master’ to load the account code which in turn calls the data access layer to load the accounting codes.  
  
**Step 2:-** In this step the accountant starts filling the voucher information. The important point to be noted in this step is that after a voucher is added there is a conditional statement which says do we want to add a new voucher. If the accountant wants to add new voucher he again repeats step 2 sequence in the sequence diagram. One point to be noted is the vouchers are not added to database they are added in to the voucher collection.  
  
**Step 3:-** If there are no more vouchers the accountant clicks submit and finally adds the entire voucher in the database. We have used the loop of the sequence diagram to show how the whole voucher collection is added to the database.



**Figure: - Voucher data entry sequence diagram**

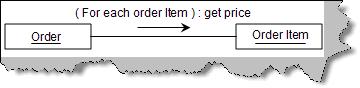
### Can you explain collaboration diagrams?

**Collaboration diagrams**  
  
Collaboration diagrams provide the same information as shown by sequence diagram but they show it in a different way. In sequence diagram we pay more attention to the time and sequence order, but in collaboration diagram we pay more emphasis on the interaction messages between the objects.   
  
Figure ‘Collaboration diagrams’ shows how a collaboration diagram looks like. Below are some points you can easily pick up looking at the figure:-  
• Objects are represented in rectangle.  
• Messages are represented by an arrow and sequence number. For instance in figure ‘collaboration diagrams’ we can see the ‘order product’ has a arrow which shows that the message is sent from the customer object to the product and ‘1’ represents that it’s the first messages.  
• Conditional statements are denoted by square brackets ‘[‘.  
• We can also group sequence numbers by grouping them using decimals. For instance ‘Pay by master’ and ‘Pay by Visa’ are all grouped in to message sequence number ‘3’ (‘Do payment’). So they are denoted by 3,3.1 and 3.2 respectively.



**Figure: - Collaboration diagrams**

You can represent the for each loop using the ‘for each’ keyword as shown in below figure ‘for each in collaboration’.

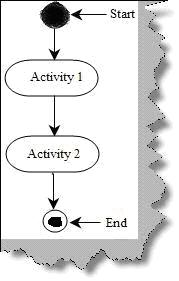


**Figure: - For each in collaboration**

***Note: - Try drawing a collaboration diagram for the same voucher data entry screen. Sequence and collaboration diagrams are also called as iteraction diagrams.***

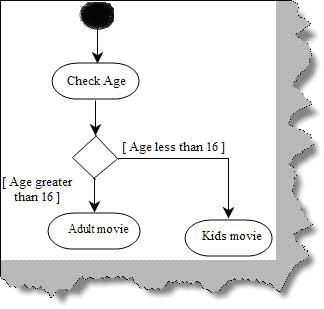
### Can you explain activity diagrams?

Activity diagrams  
  
*Activity diagram are used to capture complicated process flows in a system.* Below figure ‘Activity’ shows a simple activity diagram. Some of the points which you can easily note from the activity diagram figure are:-  
  
• Start of an activity is denoted by a dark circle.  
• End of an activity is denoted by a dark circle inside a white circle.  
• Activities are denoted by simple oval rectangles.



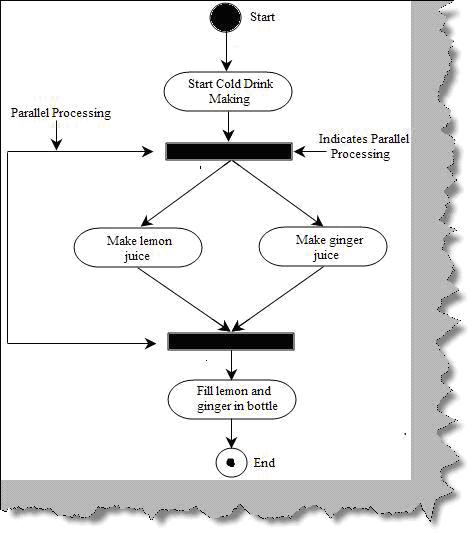
**Figure: - Activity**

A decision in activity diagram is as shown figure ‘Decision in Activity Diagram’. Figure shows the condition in a ‘[‘ (Square bracket). So the first activity is ‘Check Age’, if the age is greater than 16 then we can go for ‘Adult Movie’ activity or else we need to execute the ‘Kids Movie’ activity.



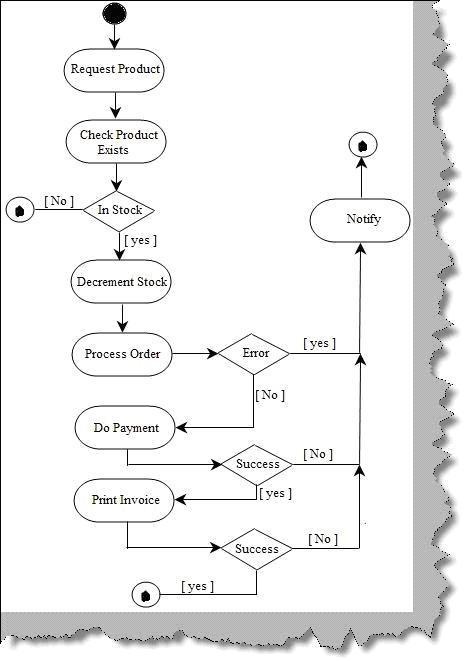
**Figure: - Decision in Activity Diagram**

There are situations in project where we need to execute two parallel activities in a project. A solid bold line represents from where the activities will split and execute in parallel and then again a solid line is where the parallel activities will meet. For instance in the below figure ‘Parallel Processing’ we can see how ‘Make lemon juice’ and ‘Make Ginger Juice’ activities are executed in parallel.



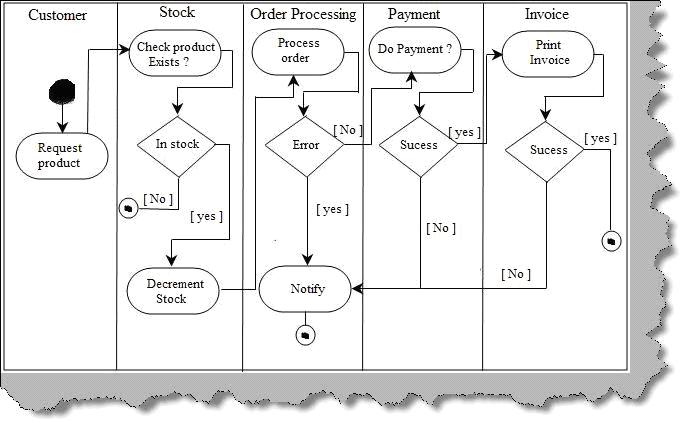
**Figure: - Parallel Processing**

In big and complex activity diagrams it’s very difficult to figure out which object is responsible for which activities. This problem is solved by ‘Swimlanes’. Consider the below figure ‘Without Swimlanes’. The whole activity diagram looks very complex and it’s very difficult to figure out which object is responsible for which activity.



**Figure: - Without Swimlanes**

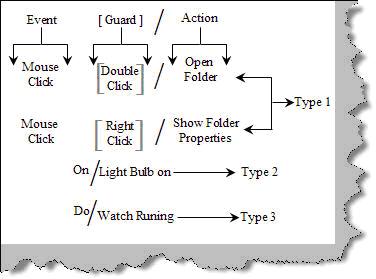
Now see the below figure ‘With Swimlanes’ we have partitioned the activity to the respective objects like Customer, Stock, Order processing, Payment and Invoice. These partitions are termed as ‘Swimlanes’ , so if you feel that the activity diagram is complex think about using ‘Swimlanes’ it can really make your activity diagram readable.



**Figure: - With Swimlanes**

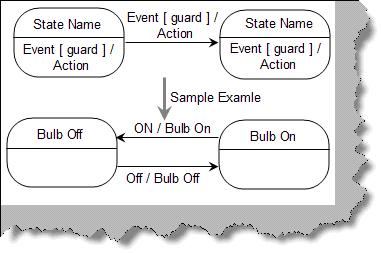
### What is state chart diagram?

**State Chart Diagram**  
  
*State diagram depicts different states that an object goes through during their life cycle. State diagram depicts how an object responds to events.* We think state diagrams as optional and should be used if your project has scenarios where the object goes through lot of complicated states and transitions. If your project does not have such kind of scenarios then sequence, collaboration or activity would be sufficient to meet your needs. So all objects have states and an object moves from one state to other state when there is some event or transition.   
  
There are three important things when we consider state of an object event, guard and action. Let’s first define these three things: - .  
  
**Action**: - Action triggers an object state from one state to another.  
  
**Event**: - Event triggers the action.  
  
**Guard**: - Guard is condition on which it evaluates which action to be triggered.  
  
These three things are principle component of state chart diagrams. Below figure ‘Types of event and action’ shows how they are represented in state chart diagrams.



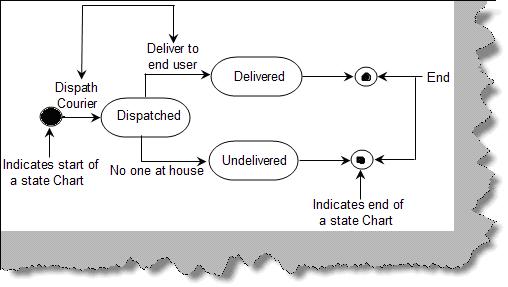
**Figure: - Types of Event and Action**

There are three ways by which we can represent the same.  
  
**Type 1:-** This methodology of writing is used when we need to map an event with action using a guard. For instance in the above figure ‘Types of Event and Action’ shows the event à mouse click, guard à double click and the resulting action à open folder.  
  
**Type 2:-** The guard is optional. For instance sometimes we only have events and actions, i.e. with out the guard. For instance when the even ‘On’ happens the action is that ‘Light Bulb is on’.  
  
**Type 3:-** This type of representation shows an infinite loop for an action. For instance the ‘Watch will be running’ infinitely during a state, as shown in figure ‘Type of Event and Action’.  
  
Now that we know how to write event, actions and guard, let’s see how state diagram looks like. Below figure ‘State example’ shows how a state looks like. It’s an oval rectangle as shown below. In order to show a transition we need to show an arrow from one state to other state as shown in figure ‘State example’.



**Figure: - State example**

Below figure ‘Sample state chart’ shows a simple state chart diagram. Some points which are immediately visible from the diagrams are as follows:-  
  
• A dark black arrow indicates start of a state chart diagram.  
• A dark circle with a white circle outside indicates end of a state chart diagram.  
• Circular rectangle indicates state while arrows indicate events / transitions.

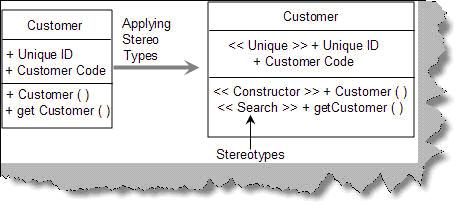


**Figure: - Sample state chart**

State is represented as shown in figure ‘Basic element of state diagram’. It’s a simple rectangle which is rounded. In the top section we give the state name. The below section is optional which has ‘do/action’. It represents a long running activity when the object goes through this state.

### Can you explain stereotypes in UML?

*Stereotypes are a way to define variations on existing UML model. This variation is brought in to place to extend UML in a consistent manner.* They are displayed in double less than and double greater than sign with a simple text as shown below. The below figure shows at the left hand side a class diagram with out stereo types while the right hand side shows with stereo types. You can easily make out how the class diagram is readable with stereo types. For instance the ‘Customer()’ can be mistaken for a method but we have clarified that it’s a constructor by using stereo types.

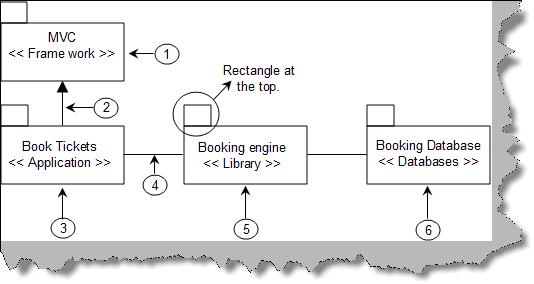


**Figure: - Stereotypes**

Below are some of the commonly used stereo types while writing UML.  
  
<<Application>>:- Used to represent a UI system in a application.  
<<Database>> :- represents a database in a application.  
<<Table>> :- A table with in database.  
<<Library>> :- A reusable library or function.  
<<File>> :- Physical file on a folder.  
<<Executable>> :- A software component which can be executed.  
<<Web services>> :- Represents a web service.  
<<JDBC>> :- Java database connectivity , a JAVA API to connect to database.  
<<ODBC>> :- Open database connectivity , a Microsoft API to connect to database.

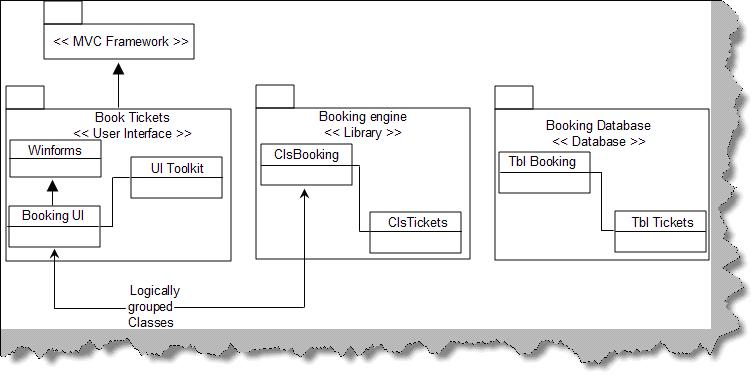
### Can you explain package diagrams?

Packages are like folders in a system which allows you to logically group UML diagrams. They make complex UML diagram readable. In actual projects they are used to logically group use cases and classes. So we can say there are two types of package diagrams one is class package diagram and other is use case package diagram. Package diagram depict a high level of overview for class and use cases.  
  
**Class package diagram: -** Class package diagram are used to logical group classes. You can think that package technically map to ‘Package’ in JAVA and ‘Namespaces’ in C# and VB.NET. Packages are denoted by small rectangle on a big rectangle as shown in figure ‘Package diagram’. One of the points to be noted is the stereotypes. We have numbered the figure so that we can understand it better.  
  
1 – We are using the MVC (Model View controller) framework. So we have denoted this package using the << Framework >> stereo type. Refer the commonly used stereo type table discussed in the previous sections.  
  
2 and 3 – ‘Book tickets’ is the second package which inherits from the MVC model. The stereo type is ‘<<application>>’ which means it’s a user interface.  
  
4 – A simple line shows a link between two classes stating that one class package uses the other class package.  
  
5 – This package is collection of the booking engine library.  
  
6 – The final package is the database.



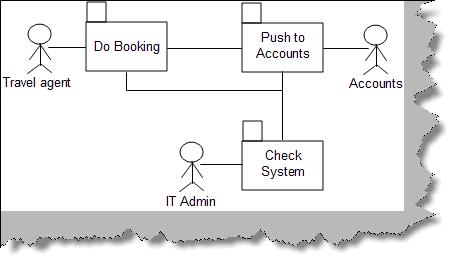
**Figure: - Package diagram**

As said before packages are nothing but collection of logical classes or any UML entity. We have shown the detail of the above package diagram. We should restrict from using package diagram for showing the in depth architecture as it can become very complicated. For instance the below diagram ‘Detail Package diagram’ shows how complicated it can look if use the package diagram to show in depth architecture. To avoid complication its good to only draw an over all diagram as shown in ‘Package diagram’.



**Figure: - Detail Package diagram**

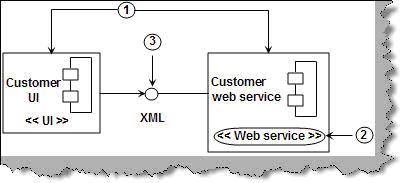
Use case package diagram: - The way we have logically grouped classes we can also use the package diagram to logically group use cases. Below figure shows how a use case package diagram looks like.



**Figure: - Use Case Package**

### Can you explain component diagrams?

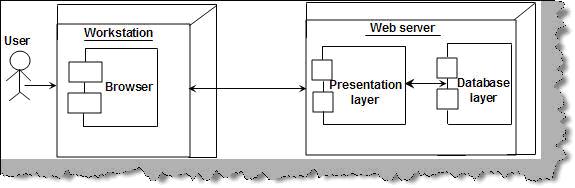
*Component diagrams achieve the same objective like package diagrams. They show the dependencies among software components.* Below figure ‘Component diagram’ shows a sample component diagram for simple data entry application which uses a web service to interact with the database. We have numbered the steps to understand the details of the component diagram.  
  
1 – Two rectangles are shown to represent a component of a system.  
  
2 – Stereo types are used to denote what kind of system it represents.  
3 – A line with a circle denotes an interface by which the external world can interact with the component. For instance in the figure we have represented a ‘Customer Web service’ which can is interacted by using XML.



**Figure: - Component Diagram**

### Can you explain deployment diagrams?

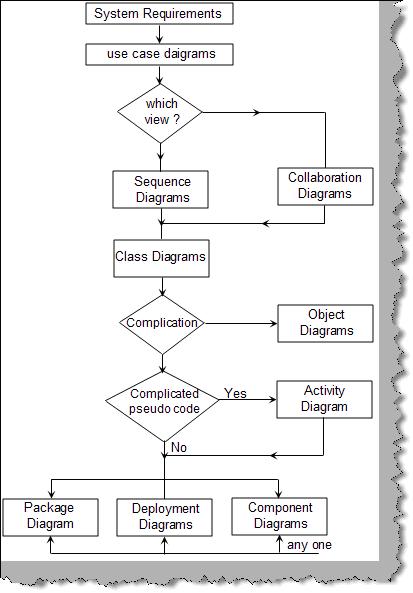
*Deployment diagrams represents an overall static view of how software and hardware nodes in the application. They show what the hardware is and which components are installed on which hardware.* In deployment diagram we represent the hardware with a solid box and simple underlined text description showing which hardware is it. You can have more than one component on a single hardware. So the browser is an application UI which resides on the workstation computer and the database and web server resides on the web server hardware. Deployment diagram can be more complex with firewalls, payment gateways, PDA devices, VPN etc.



**Figure: - Deployment diagram**

### Can you explain how UML flows in actual project?

In actual projects we do not draw all the diagrams. Every UML diagram is made for a purpose. It completely depends on what’s the nature of the project. In short you should ask yourself questions like, is this diagram important, what’s my need etc. So below is a flow which you can follow in your project, again as we said it depends on what kind of scenario you want to depict.



**Figure: - UML flow in actual projects**

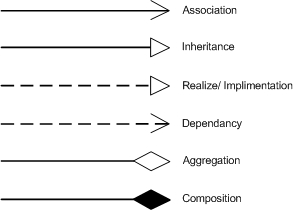
• The first step is to derive use cases from the requirement documents.  
• Once use cases are derived we need to decide the messages which will flow in the system. This can be done using interaction diagrams. If you need to know the object creation life times we use the sequence diagram and if we want to concentrate on the messages we use the collaboration diagrams. So depending on scenario we need to make a choice which diagram we need to draw.  
• Now that we are clear about messages we can draw class diagrams to depict the static part of the project i.e. classes.  
• If we find any complicated class relationships we draw object diagrams.  
• If we need to depict any complicated code we need to represent same with a activity diagram.  
• Finally to give an overview of the project we can use package diagram, component or deployment diagram. As said before we can use combination of component and deployment diagram to give a overview of the architecture.  
  
***Note: - Never say in an interview that we have used all UML diagrams in the technical document. It can give a very bad impression. As said every UML diagram is drawn according to the scenario of the project.***

**What is a Class Diagram?**

*A class diagrams are widely used to describe the types of objects in a system and their relationships.* Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design.

The Class diagrams, physical data models, along with the system overview diagram are in my opinion the most important diagrams that suite the current day rapid application development requirements.

UML Notations:



**What is a Package Diagram?**

Package diagrams are used to reflect the organization of packages and their elements. When used to represent class elements, package diagrams provide a visualization of the name-spaces. In my designs, I use the package diagrams to organize classes in to different modules of the system.

**What is a Sequence Diagram?**

A sequence diagrams model the flow of logic within a system in a visual manner, it enable both to document and validate your logic, and are used for both analysis and design purposes. Sequence diagrams are the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within your system.

**What is the difference between Association, Aggregation and Composition?**

Association is a (\***a**\*) relationship between two classes, where one class use another. But aggregation describes a special type of an association.

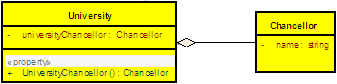
Aggregation is the (\***the**\*) relationship between two classes. When object of one class has an (\***has**\*) object of another, if second is a part of first (containment relationship) then we called that there is an aggregation between two classes. Unlike association, aggregation always insists a direction.

public class University

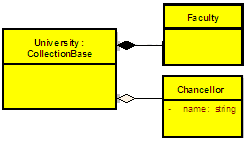
{

private Chancellor universityChancellor = new Chancellor();

}



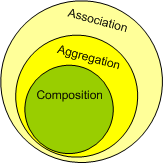
In this case I can say that *University* aggregate *Chancellor* or *University* has an (\***has-a**\*) *Chancellor*. But even without a *Chancellor* a *University* can exists. But a *University* cannot exist without *Faculties*, the life time of a *University* attached with the life time of its *Faculty* (or Faculties). If *Faculties* are disposed the *University* will not exist or wise versa. In that case we called that *University* is composed of *Faculties*. So that composition can be recognized as a special type of an aggregation.



Same way, as another example, you can say that, there is a composite relationship in-between a *KeyValuePairCollection* and a *KeyValuePair*. The two mutually depend on each other.

.Net and Java uses the Composite relation to define their Collections. I have seen Composition is being used in many other ways too. However the more important factor, that most people forget is the life time factor. The life time of the two classes that has bond with a composite relation mutually depend on each other. If you take the .net Collection to understand this, there you have the Collection Element define inside (it is an inner part, hence called it is composed of) the Collection, farcing the Element to get disposed with the Collection. If not, as an example, if you define the Collection and it’s Element to be independent, then the relationship would be more of a type Aggregation, than a Composition. So the point is, if you want to bind two classes with Composite relation, more accurate way is to have a one define inside the other class (making it a protected or private class). This way you are allowing the outer class to fulfill its purpose, while tying the lifetime of the inner class with the outer class.

So in summary, we can say that aggregation is a special kind of an association and composition is a special kind of an aggregation. (*Association->Aggregation->Composition*)



**Define a relationship.** **What are the different relationships that exist between classes in the object oriented paradigm?**

In Unified Modeling Language (UML) class diagrams, a relationship is the connection between classes or interfaces.

You can use several relationships to define the structure between classes or interfaces:

* Association relationships imply that instances of one class connect to instances of another class.
* Dependency relationships imply that a change to one class might affect another class.
* Extends relationships imply that one class is a specialization of another class.
* Implements relationships imply that one class provides a specification and the other class implements the specification.

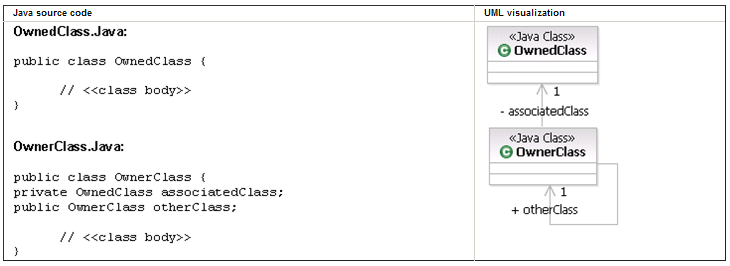
You can also use note attachment relationships to provide additional information about diagram elements.

* **Association relationships**  
  In Unified Modeling Language (UML) class diagrams, an association is a structural relationship that indicates that objects of one classifier (such as a class and interface) are connected and can navigate to objects of another classifier.
* **Dependency relationships**  
  In Unified Modeling Language (UML) class diagrams, a dependency relationship indicates that a change to one class (the supplier) might cause a change in the other class (the client). The supplier is independent because a change in the consumer does not affect the supplier.
* **Extends relationships**  
  In Unified Modeling Language (UML) class diagrams, an extends relationship (also called an inheritance or an is-a relationship) implies that a specialized (child) class is based on a general (parent) class.
* **Implements relationships**  
  In Unified Modeling Language (UML) class diagrams, an implements relationship exists between two classes when one of them must implement, or realize, the behavior specified by the other.
* **Owned element association relationships**  
  In Unified Modeling Language (UML) class diagrams, an owned element association relationship is a type of association that dictates ownership.
* **Adornments for unresolved references**  
  In Unified Modeling Language (UML) class diagrams, Java classes and interfaces or Enterprise JavaBeans (EJB) enterprise beans belonging to different applications or projects that are not currently loaded in the workspace can result in an unresolved reference.

**How do we represent association relationship in code?**

In UML class diagrams, association relationships in a Java™ application represent the following things:

* A semantic relationship between two or more classes that specifies connections between their instances
* A structural relationship that specifies that objects of one class are connected to objects of a second (possibly the same) class

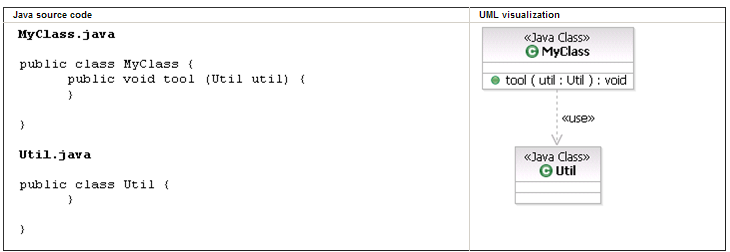


**How do we represent dependency relationship in code?**

In UML class diagrams, dependency relationships in a Java™ application connect two classes to indicate that there is a connection between the two classes, and that the connection is more temporary than an association relationship. A dependency relationship indicates that the consumer class does one of the following things:

* Temporarily uses a supplier class that has global scope
* Temporarily uses a supplier class as a parameter for one of its operations
* Temporarily uses a supplier class as a local variable for one of its operations
* Sends a message to a supplier class

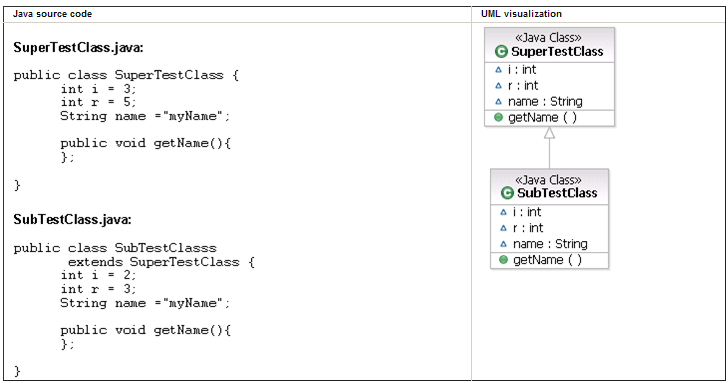
As the following figure illustrates, a dependency relationship connector is displayed as a dashed line with an open arrow that points from the client class to the supplier class.



**How do we represent extends relationship in code?**

In Unified Modeling Language (UML) class diagrams, an extends relationship (also called an inheritance or an is-a relationship) implies that a specialized (child) class is based on a general (parent) class.

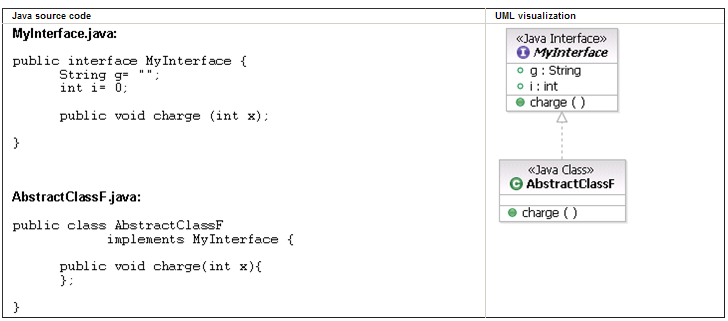
As the following figures illustrates, an extends relationship connector is displayed as a solid line with an unfilled arrowhead that points from the specialized (child) Java™ class or Enterprise JavaBeans™ (EJB) enterprise bean to the general (parent) Java class or EJB enterprise bean.



**How do we represent implements relationship in code?**

In UML class diagrams, an implements relationship represents a class that implements the operations in a Java™ interface.

As the following figure illustrates, an implements relationship connector is displayed as a dashed line with an unfilled arrowhead. The connector points from the client (that realizes the behavior) to the supplier (that specifies the behavior).

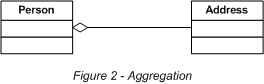


**What is the difference between composition and aggregation and how would you express it in your programs?**

*Aggregation:*

If inheritance gives us *'is-a'* and composition gives us *'part-of'*, we could argue that aggregation gives us a *'has-a'* relationship. Within aggregation, the lifetime of the part is not managed by the whole. To make this clearer, we need an example. For the past 12+ months I have been involved with the implementation of a CRM system, so I am going to use part of this as an example.

The CRM system has a database of customers and a separate database that holds all addresses within a geographic area. Aggregation would make sense in this situation, as a Customer 'has-a' Address. It wouldn't make sense to say that an Address is 'part-of' the Customer, because it isn't. Consider it this way, if the customer ceases to exist, does the address? I would argue that it does not cease to exist. Aggregation is shown on a UML diagram as an unfilled diamond (see Figure 2).



So how do we express the concept of aggregation in C#? Well, it's a little different to composition. Consider the following code:

public class Address  
{  
 . . .  
}

public class Person

{

private Address address;

public Person(Address address)

{

this.address = address;

}

. . .

}

Person would then be used as follows:

Address address = new Address();  
Person person = new Person(address);

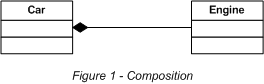
or

Person person = new Person( new Address() );

As you can see, Person does not manage the lifetime of Address. If Person is destroyed, the Address still exists. This scenario does map quite nicely to the real world.

*Composition*

As we know, inheritance gives us an *'is-a'* relationship. To make the understanding of composition easier, we can say that composition gives us a *'part-of'* relationship. Composition is shown on a UML diagram as a filled diamond (see Figure 1).



If we were going to model a car, it would make sense to say that an engine is part-of a car. Within composition, the lifetime of the part (Engine) is managed by the whole (Car), in other words, when Car is destroyed, Engine is destroyed along with it. So how do we express this in C#?

public class Engine  
{  
 . . .   
}

public class Car

{

Engine e = new Engine();

.......

}

As you can see in the example code above, Car manages the lifetime of Engine.

**What is the importance of multiplicity? How to use in class diagram?**

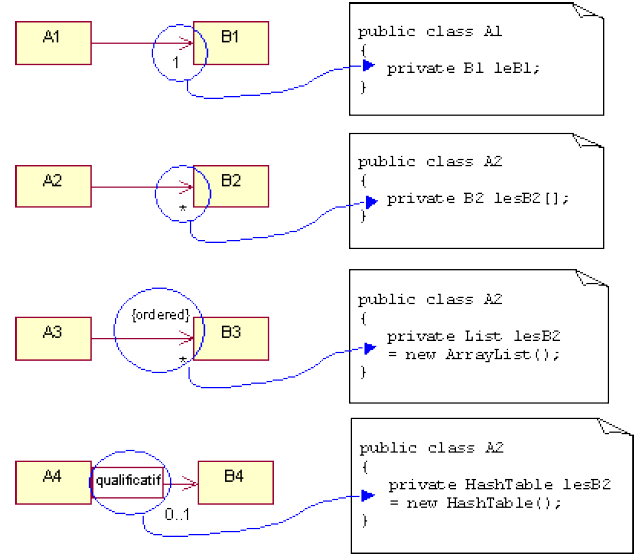
Multiplicity information can be linked to an association to show how many instances of class A are linked with instances of class B. Multiplicity information can be linked to both ends of association relationships.



The multiplicity of the association is labeled on either end of the line, one multiplicity indicator for each direction ([Table 1](http://www.agilemodeling.com/artifacts/classDiagram.htm#Table1MultiplicityIndicators) summarizes the potential multiplicity indicators you can use).

**Table 1. Multiplicity Indicators.**

| **Indicator** | **Meaning** |
| --- | --- |
| 0..1 | Zero or one |
| 1 | One only |
| 0..\* | Zero or more |
| 1..\* | One or more |
| n | Only *n* (where *n* > 1) |
| 0..n | Zero to *n* (where *n* > 1) |
| 1..n | One to *n* (where *n* > 1) |

**How do we represent multiplicity in association relationship in code?  
**

**Illustrate the concepts of aggregation and composition by giving example.**

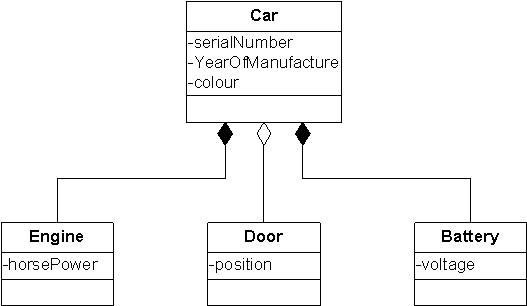
Suppose you want to model a car. Some objects only make sense within a whole car and not on their own. We can say that the car is composed of other objects. Polymorphism uses two terms to describe this, aggregation and composition. These relationships form a whole-part relationship that you can use to decompose objects into more manageable entities.

* Any object that can exist and be used independently of the relationship uses aggregation.
* Any object that has no meaning outside of the relationship uses composition.

An example

For example, a class named Car would have an engine. Objects that fall into this relationship use of the term "is a part of". "An engine is a part of a car". In UML, an object relationship that is formed by aggregation is drawn using an empty diamond. An object relationship that is formed using composition is drawn using a filled diamond.

The following UML diagram illustrates the concepts of aggregation and composition.



In the diagram above, the battery and the engine have no meaning outside of the car, as the car cannot work without either of them, so the relationship is formed using composition. However, a car can work without doors, so the relationship is formed using aggregation.

From the previous diagram, we can create a possible small code sample that shows the two forms of relationship. Objects that use composition are created as inner classes.

namespace MyCars

{

public class Car

{

*// Aggregation uses instances of objects created outside of*

*// this class*

protected Door FrontRight;

protected Door FrontLeft;

protected Door RearRight;

protected Door RearLeft;

*// inner classes used to create objects that are intrinsically*

*// linked to the class Car*

protected class Engine

{

public int horsePower;

}

protected class Battery

{

public int voltage;

}

*// Composition uses instances of objects that are created as*

*// part of this object*

protected Engine theEngine;

protected Battery theBattery;

public Car()

{

theEngine = new Engine();

theBattery = new Battery();

}

}

public class Door

{

public int position;

}

}

**What is the difference between dependency and association?**

| **Dependency** | **Association** |
| --- | --- |
| A dependency in the [Unified Modeling Language](http://language) exists between two defined elements if a change to the definition of one may result in a change to the other. In UML this is indicated by a dashed line pointing from the dependent (or client) to the independent (or supplier) element. | An association is a relationship between two classifiers, such as classes or use cases that describes the reasons for the relationship and the rules that govern the relationship. |
| It is represented by a dotted line with an open arrowhead that shows one entity depends on the behavior of another entity. As the following figure illustrates, a dependency is displayed in the diagram editor as a dashed line with an open arrow that points from the client model element to the supplier model element. | An association appears as a solid line between two classifiers. |
| You can use dependency relationships in class diagrams, component diagrams, deployment diagrams, and use case diagrams. | An association represents a structural relationship that connects two classifiers such as classes, use-cases. |
| Dependency relationships usually do not have names. | The name of an association describes the nature of the relationship between two classifiers and should be a verb or phrase. |
|  | An association end specifies the role that the object at one end of a relationship performs. Each end of a relationship has properties that specify the role of the association end, its multiplicity, visibility, navigability, and constraints. |

**How do you use dependency relationship in your model?**

You can add dependency relationships to your model to accomplish the following goals:

* Connect two packages to indicate that at least one model element in the consumer package is dependent on one model element in the supplier package. The dependency relationship does not indicate that all model elements in the consumer package are dependent.
* Connect two classes to indicate that the connection between them is at a higher level of abstraction than an association relationship. The dependency relationship indicates that the consumer class performs one of the following functions:
  + Temporarily uses a supplier class that has global scope
  + Temporarily uses a supplier class as a parameter for one of its operations
  + Temporarily uses a supplier class as a local variable for one of its operations
  + Sends a message to a supplier class
* Connect components to interfaces or other components to indicate that they use one or more of the operations that the interface specifies or that they depend on the other component during compilation.

**What are the types of dependency relationships?**

Because a dependency relationship can represent several different types of relationships, keywords or stereotypes are used to show the precise nature of the dependency.

| **Type of dependency** | **Keyword or Stereotype** | **Description** |
| --- | --- | --- |
| Abstraction | «abstraction», «derive», «refine», or «trace» | Relates two model elements, or sets of model elements, that represent the same concept at different levels of abstraction, or from different viewpoints. |
| Binding | «bind» | Connects template arguments to template parameters to create model elements from templates. |
| Realization | «realize» | Indicates that the client model element is an implementation of the supplier model element, and the supplier model element is the specification. |
| Substitution | «substitute» | Indicates that the client model element takes the place of the supplier. The client model element must conform to the contract or interface that the supplier model element establishes. |
| Usage | «use», «call», «create», «instantiate», or «send» | Indicates that one model element requires another model element for its full implementation or operation. |

**Note:** Several relationships use a connector that is similar to that of a dependency relationship, but are not considered types of dependency relationships; for example, extend relationships and include relationships.

**1. What is the difference between an "is-a" relationship and a "has-a" relationship? What are the two types of "association" relationships?**

"is-a" indicates that one object is a "kind of" a class; for example, a "sail boat" is a kind of "boat" which is a kind of "type of transportation".

"has-a" indicates that one class "contains" another class; for example, a car has wheels.

There are two types of "associations": containment (has-a) and "part-of" (p. 29)

**2. In the Class diagram, a class is shown as a box, which can have up to three parts. Describe these three parts.**

The top box is the name (label) of the class. This is required.

The middle box, if it is shown, shows the data members of the class.

The bottom box, if it is shown, shows the methods (functions) of the class. (p. 32)

**3. Define cardinality.**

Cardinality indicates the number of things that another object can have (p. 36)

**4. What is the purpose of a Sequence diagram?**

The Sequence diagram is one type of Interaction Diagram in the UML. It shows how objects interact with other objects. (p. 38)

**Interpretations**

**1. Give an example of an "is-a" relationship and the two "association" relationships. Using these examples,**

Draw them in a Class diagram

Show cardinality on this Class diagram

Is-a example: "Sailboat" is-a "boat"

Has-a example: Sailboat has-a sail (one to many)

Uses example: A marina contains one or more Sailboats (p. 32)

**2. Figure 2-8 shows a Sequence diagram. How many steps are shown in the figure? How many objects are shown and what are they?**

There are 13 steps in the diagram

There are 6 objects shown: Main, ShapeDB, Collection, shape1:Square, shape2:Circle, and Display. (p. 39)

**3. When objects communicate with each other, why is it more appropriate to talk about "sending a message" than "invoking an operation"?**

When objects "talk" to each other, it is called "sending a message." You are sending a request to another object to do something rather than telling the other object what to do. You allow the other object to be responsible enough to figure out what to do. Transferring responsibility is a fundamental principle of object-oriented programming. It is quite different from procedural programming where you retain control of what to do next, and thus might "calling a method" or "invoke an operation" in another object.

**Opinions and Applications**

**1. How many steps should be shown on a Sequence diagram?**

As many as it takes to communicate clearly, and no more