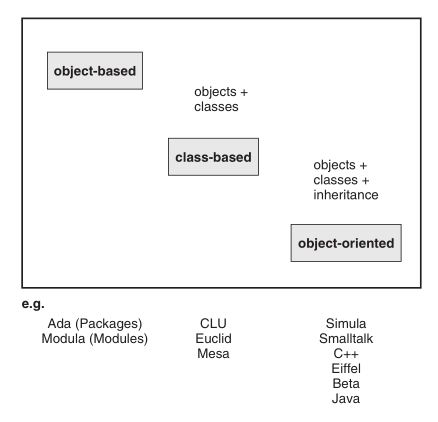
**Programming Interview Questions**

1. **What is the difference between an object based and an object oriented language?**

Ans:-

|  |  |
| --- | --- |
| **Object oriented language** | **Object based language** |
| Object oriented language support all feature of OOPS i.e. Encapsulation, Inheritance and polymorphism | Object based language doesn’t supports OOPS concept |
| Here there is no built in type of objects  are available | Build in type object are available |
| Example Object oriented languages are Eiffel, Python, Ruby, Java, C# etc. | Object based language is Ada, JavaScript, and Visual Basic (VB) |
| It’s statically type-checked | It’s statically type-checked |
| It’s object oriented where one can acquire or hide property of objects | It’s also called prototype based language |
| It does not supports build in object | It does supports build in types |



According to Wegner’s classification, a language is object-based if it creates and lets  
you manage objects as a primary language construct. In contrast, class-based languages have the class concept in addition to the object concept. This means that similar objects can be defined in appropriate classes, and that these classes are then responsible for creating objects as instances. Finally, an object-oriented language adds the inheritance concept to the features of a class-based language. This inheritance concept allows you to create a hierarchy of subclasses and superclasses. Such class hierarchies have to be supported by a polymorphic-type system to be able to call an object of a subclass indirectly in the context of a superclass, that is, through a typed identifier.

## *what is object based programming?*

***In a technical sense, the term "object-based language" may be used to describe any programming language that is based on the idea of encapsulating state and operations inside objects.***

1. **kbhit() function in C/C++**

Prototype: int kbhit(void);  
Header File: conio.h  
Explanation: This function is not defined as part of the ANSI C/C++ standard. It is generally used by Borland's family of compilers. It returns a non-zero integer if a key is in the keyboard buffer. It will not wait for a key to be pressed.

Example:

//Example will loop until a key is pressed

//Example will not work with all compilers

#include <conio.h>

#include <iostream>

using namespace std;

int main()

{

while(1)

{

if(kbhit())

{

break;

}

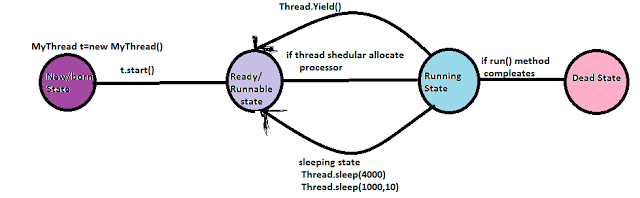
}

}

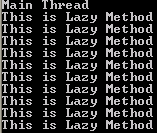
1. Life Cycle Of Thread in java With Example

This is one of the important concept in Core Java. A Thread can be any one of the five states. The Life Cycle of Thread in java is controlled by JVM. The Thread States are given below:  
  
a) **New or born State**

b) **Ready or Runnable State**  
c) **Running State**  
d) **Blocked State**  
e)**Dead State**

 **a) New State:**  
 When we write new MyThread() of  the Thread() class  the Thread has born  
  
**b) Ready State:**When we call start() method then the thread enter into Runnable State  
                                   Ex: t.start();  
  
Note:  After Starting a Thread we can not restart same thread once again, it violation leads to Runtime Error  saying"IlleagalThreadStateException"  
  
Ex:  MyThread t=new MyThread();  
           t.start();  
           t.start();//IllegalThreadStateException;

**Yield():**  
  
The thread which is called yield() method temporarily pause the execution to give the chance for remaining threads of same priority. If there is no waiting thread or all waiting threads having low priority. Then the same thread will get the chance immediately for the execution.  
  
public static native void yield();  
  
**Ex:**  
**class MyThread extends Thread**  
**{**  
**public void run()**  
**{**  
**for (int i=0; i< 10 ; i++ )**  
**{**  
**System.out.println("Child Thread");**  
**Thread.yield();**  
**}**  
**}**  
**}**  
**class YieldDemo**  
**{**  
**public static void main(String arg[])**  
**{**  
**MyThread t = new MyThread();**  
**t.start();**  
**for(int i =0;i<10;i++)**  
**{**  
**System.out.println("Main Thread");**  
**}**  
**}**  
**}**  
  
In this case main thread will get chance more no of times for execution. Because child thread  
intentionally calling “yield()” method. As the yield method is native method some Operating system may not provide the support for this.  
  
**Sleep():**  
  
If a method has to wait some predefined amount of time with out execution then we should go for sleep() method.  
  
public static void sleep(long ms)throws InterruptedException  
  
public static void sleep(long m, int m)throws InterruptedException  
  
**Ex:**  
**class MyThread extends Thread**  
**{**  
**public void run()**  
**{**  
**try**  
**{**  
**for (int i = 0;i<10;i++)**  
**{**  
**System.out.println("This is Lazy Method");**  
**Thread.sleep(1000);**  
**}**  
**}**  
**catch (InterruptedException e)**  
**{**  
**System.out.println(e);**  
**}**  
**}**  
**}**  
**class SleepDemo**  
**{**  
**public static void main(String arg[])throws InterruptedException**  
**{**  
**MyThread t = new MyThread();**  
**t.start();**  
**System.out.println("Main Thread");**  
**}**  
**}**  
 **Output:**

****

**Running State:**The Thread is Running state if Thread Scheduler selected it.

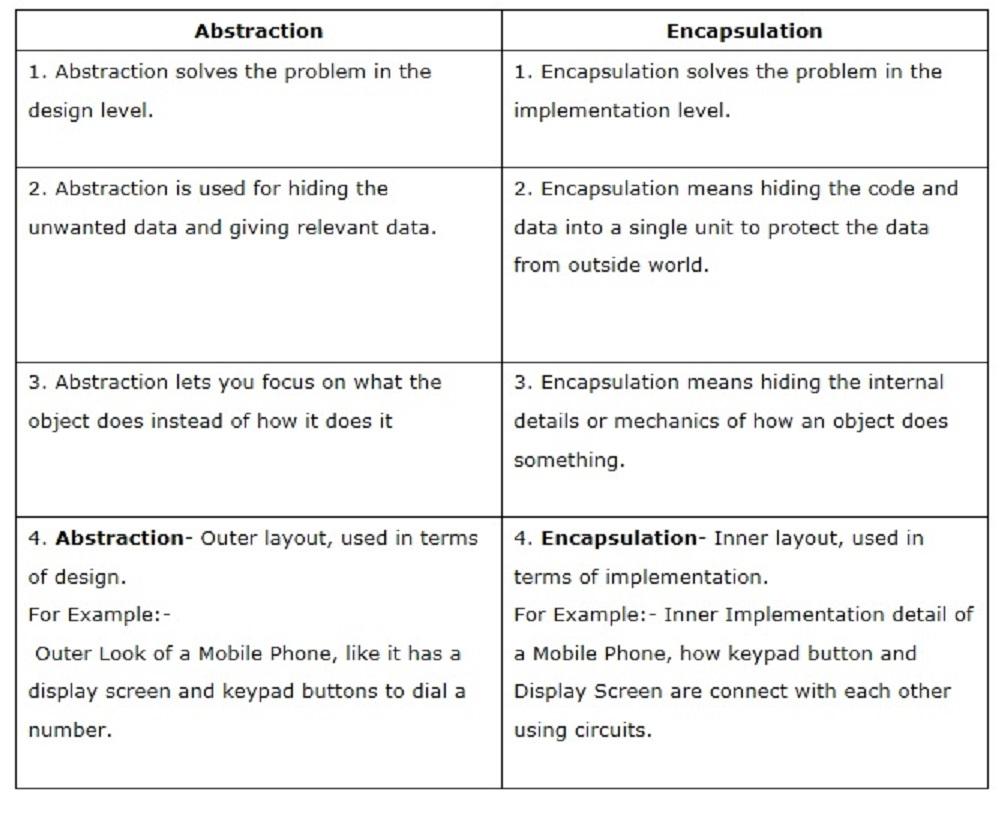
**Blocked State:**In this state The Thread might be in sleeping state,waiting state,yield state that means Thread is still alive but not eligible to run.  
 **Dead State:**  
A Thread is in Dead state or terminate when the Thread is successfully completed run() method.

## What is the Difference between Abstraction and Encapsulation?

**Ans:**

1) The most important difference between [Abstraction](http://www.java67.com/2012/08/difference-between-abstraction-and-encapsulation-java-oops.html) and [Encapsulation](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) is that Abstraction solves the problem at design level while Encapsulation solves it implementation level.

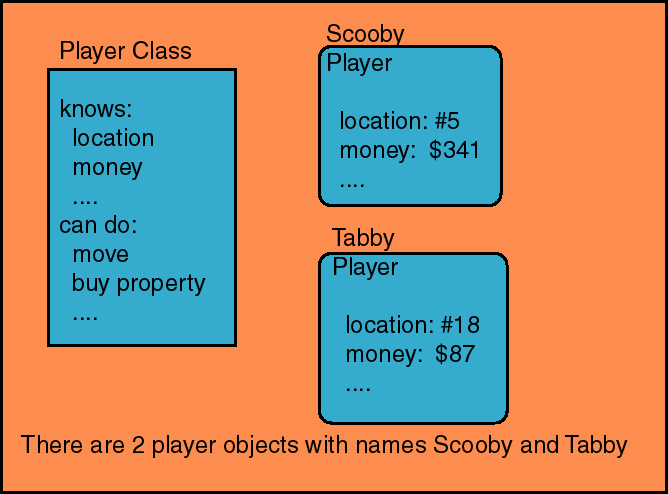
2) Abstraction is about hiding unwanted details while giving out most essential details, while Encapsulation means hiding the code and data into a single unit e.g. class or method to protect inner working of an object from outside world. In other words, Abstraction means extracting common details or generalizing things.  
  
3) Abstraction lets you focus on what the object does instead of how it does, while Encapsulation means hiding the internal details of how an object works. When you keep internal working details private, you can change it later with a better method. The [Head First Object Oriented Analysis and Design](https://www.amazon.com/Head-First-Object-Oriented-Analysis-Design/dp/0596008678/?tag=javamysqlanta-20) has some excellent examples of these OOP concepts, I suggest you read that book at least once to revisit OOP fundamentals.  
  
4) Abstraction focus on outer lookout e.g. moving of vehicle while Encapsulation focuses on internal working or inner lookout e.g. how exactly the vehicle moves.  
  
5) In Java, Abstraction is supported using [interface](http://www.java67.com/2012/09/what-is-difference-between-interface-abstract-class-java.html) and [abstract class](http://javarevisited.blogspot.sg/2013/05/difference-between-abstract-class-vs-interface-java-when-prefer-over-design-oops.html) while Encapsulation is supported using access modifiers e.g. public, private and protected.

Here is a nice table highlighting key **differences between Abstraction and Encapsulation** in Object Oriented Programming:  
  
Read more: <http://javarevisited.blogspot.com/2017/04/difference-between-abstraction-and-encapsulation-in-java-oop.html#ixzz4jClPjKuX>

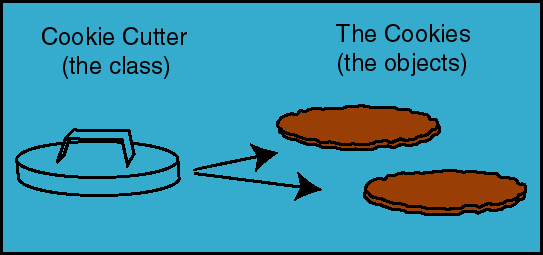
1. **What is the difference between a Class and an Object in Java?**

**Ans:**

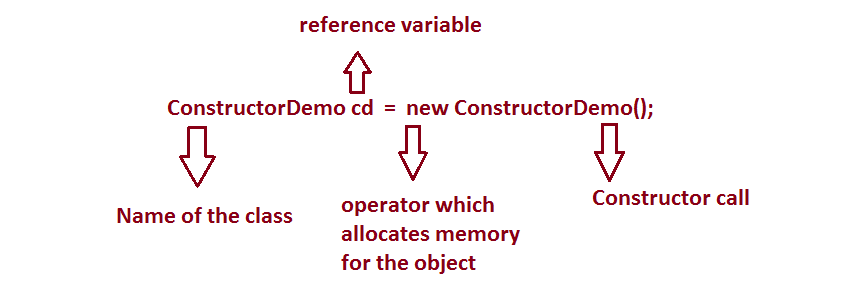
1) A class is what you create while coding, but object is created at runtime by your execution environment e.g. JVM. Though you write code, which is required to create object during coding e.g. new Student(), object is not created at that time. They are only created when you run your program, and when runtime executes that line. Usually [constructor](http://java67.blogspot.sg/2012/11/java-enum-example-with-constructor.html) of a class is called when an object is created in Java, but yes there are some anomalies as well e.g. Serialization.  
  
2) Most important difference between class and object is that an Object usually has state (though stateless object is also possible). This is used to differentiate with another object. For example, If you have a class to represent Student, then John and Mohan are two object of that class, which has different name, an attribute which differentiate them. A picture is worth more than 1000 words, and *difference between class and object* can be best explained by this image :



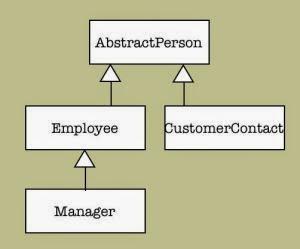
 Here Player is a class which is actually the blueprint of creating players and two players Scooby and Tabby are objects which is created by runtime environment, Java Virtual Machine in this case. Each Player has different value for their attribute, also known as state of Object. For example Scooy's position is 5th and has $341, while Tabby's position is 18th and has $87.  If you still doesn't quite get difference between Class and Object then here is one more example, which you might find more interesting then this one.



Here CookieCutter is a class which is a blueprint to create objects, the cookies. You can see we have two cookies here, one for you and one for me :)  
  
3) Though [Java is not pure Object oriented language](http://java67.blogspot.sg/2014/03/is-java-pure-object-oriented-programming-language.html), most of things are object in Java, for example primitive variables and operator are not object. In Java, objects are created on a special memory area, known as heap memory. No matter on which scope you create them e.g. locally or globally they are always created in heap space. On the other hand, classes are loaded into another special area of JVM memory, known as permgen space. From Java 8 onward, this space is also known as metaspace. You can create as many object from a class as you want, subject to your heap memory limit, because each object takes some memory. Once all the memory is exhausted, you can not create any more object and JVM will throw [java.lang.OutOfMemoryError: Java Heap Space](http://java67.blogspot.sg/2013/08/guide-of-javalangoutofmemoryerror-java-heap-space-tomcat-eclipse-minecraft-jboss.html) if you further try to create object.  
  
4) Object's are also known as instances in Java programming language, and in JVM class is also represented by an instance of java.lang.Class. On the other hand class is also know as type. A reference variable which holds reference of an object have a type, which denotes what kind of object it can reference. For example in following code ConstructorDemo is name of the class, known as type here, cd is a reference variable of type ConstructorDemo, which means it can either point a ConstructorDemo object or it's child classes. When you create Object using new() operator it automatically class the constructor of that class as well.



5) Class is an abstraction to contain code, mostly attributes and methods which operate on them. On the other hand object are the real thing which operate on them, but there comes [static methods](http://java67.blogspot.sg/2012/08/can-we-overload-static-method-in-java.html), which belongs to class.  
  
Read more: <http://www.java67.com/2014/08/what-is-difference-between-class-and-object-java-programming-oops.html#ixzz4jDMzvh3b>  
  
**6. Difference between Polymorphism and Inheritance in Java and OOP  
Ans:**



1. Inheritance defines father-son relationship between two classes, While Polymorphism take advantage of that relationship to add dynamic behaviour in your code.
2. Inheritance is meant for *code reuse*, initial idea is to reuse what is written inside Parent class and only write code for new function or behaviour in Child class. On the other hand Polymorphism allows Child to redefine already defined behaviour inside parent class. Without Polymorphism it's not possible for a Child to execute its own behaviour while represented by a Parent reference variable, but with Polymorphism he can do that.
3. Polymorphism helps tremendously during Maintenance. In fact many object oriented design principles are based on Polymorphism e.g. [programming for interface then implementation](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html), which advocates to use interface everywhere in your code, to represent variable, in method parameters, in return type of method etc; so that code can take advantage of polymorphism and do more than what was expected it to do during writing.
4. Java doesn't allow [multiple inheritance of classes](http://javarevisited.blogspot.sg/2011/07/why-multiple-inheritances-are-not.html), but allows multiple inheritance of Interface, which is actually require to implement Polymorphism. For example a Class can be Runnable, Comparator and Serializable at same time, because all three are interfaces. This makes them to pass around in code e.g. you can pass instance of this class to a method which accepts Serializable, or to Collections.sort() which accepts a [Comparator](http://java67.blogspot.sg/2013/08/difference-between-comparator-and-comparable-in-java-interface-sorting.html).
5. Both Polymorphism and Inheritance allow Object oriented programs to evolve. For example, by using Inheritance you can define new user types in an Authentication System and by using Polymorphism you can take advantage of already written authentication code. Since, [Inheritance](http://java67.blogspot.sg/2012/08/what-is-inheritance-in-java-oops-programming-example.html) guarantees minimum base class behaviour, a method depending upon super class or super interface can still accept object of base class and can authenticate it.
6. In UML diagram, Inheritance is represented using arrows, pointing towards Parent class. For example in this diagram, AbstractPerson is Parent class for Employee, Manager and CustomerContact class.  
     
   Read more: <http://www.java67.com/2014/04/difference-between-polymorphism-and-Inheritance-java-oops.html#ixzz4jDSfD3zF>
7. **Difference between Association, Composition and Aggregation in Java, UML and Object Oriented Programming**

**Ans:**

In Object-oriented programming, one object is related to other to use functionality and service provided by that object. This relationship between two objects is known as the *association*in  object oriented general software design and depicted by an arrow in Unified Modelling language or UML. Both Composition and Aggregation are the form of association between two objects, but there is a **subtle difference between composition and aggregation**, which is also reflected by their UML notation. We refer association between two objects as [Composition](http://javarevisited.blogspot.com/2015/06/difference-between-inheritance-and-Composition-in-Java-OOP.html), when one class ***owns***other class and other class can not meaningfully exist, when it's owner destroyed, for example, Human class is a composition of several body parts including Hand, Leg and Heart. When human object dies, all it's body part ceased to exist meaningfully, this is one example of Composition.

### An Example of Association, Composition and Aggregation in Java

Here is an example of composition and aggregation, in terms of Java Code. By looking at this code, you can gauge differences between these two. By the way, Composition is also very much preferred in object oriented design over inheritance, even Joshua Bloch has stated its importance in the classic book, [Effective Java](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20).  
  
**Composition :** Since Engine is-part-of Car, the relationship between them is Composition. Here is how they are implemented between Java classes.

**public** **class** **Car** {

//final will make sure engine is initialized

**private** **final** Engine engine;

**public** **Car**(){

engine = **new** Engine();

}

}

**class** **Engine** {

**private** String type;

}  
  
**Aggregation :** Since Organization has Person as employees, the relationship between them is Aggregation. Here is how they look like in terms of Java classes

**public** **class** **Organization** {

**private** List employees;

}

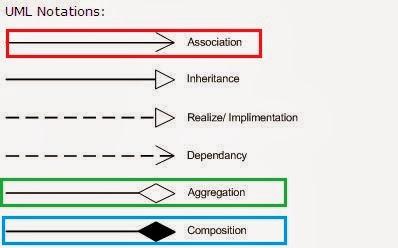
**public** **class** **Person** {

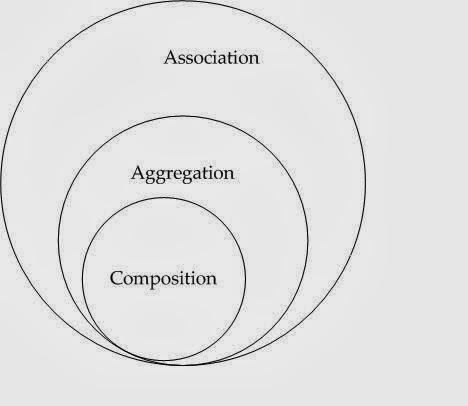
**private** String name;

}

## UML Diagram of Association, Composition and Aggregation

UML has different notations to denote aggregation, composition and association.  Association is denoted by the simple arrow while aggregation is denoted by  empty diamond-head arrow and composition is denoted by filled diamond-head arrow. When you draw UML diagram for two related class A and B, where A is associated with B then its denoted by A -> B. Similar way is used to show aggregation and composition between two classes. Here are UML notations for different kind of dependency between two classes.

  
As I said all three denotes relationship between object and only differ in their strength, you can also view them as below, where composition represents strongest form of relationship and association being the most general form.



### Association vs Composition vs Aggregation

Here is the list of differences between Composition and Aggregation in point format, for quick review. As I said the key difference between them comes from the point that in the case of [Composition](http://javarevisited.blogspot.sg/2013/06/why-favor-composition-over-inheritance-java-oops-design.html), One object is OWNER of another object, while in the case of aggregation, one object is just a USER or another object.  
  
1) If A and B two classes are related to each other such that, B ceased to exist, when A is destroyed, then the association between two objects is known as **Composition**. An example is Car and Engine. While if A and B are associated with each other, such that B can exist without being associated with A, then this association in known as **Aggregation**.

2) In the case of Composition A owns B e.g. Person is the owner of his Hand, Mind and Heart, while  in the case of Aggregation, A uses B e.g. Organization uses People as an employee.  
  
3) In UML diagram Association is denoted by a normal arrow head, while Composition is represented by filled diamond arrow head, and Aggregation is represented by an empty diamond arrow head, As shown in below and attached diagram in the third paragraph.  
  
Association  A---->B  
Composition  A-----<filled>B  
Aggregation  A-----<>B  
  
4) Aggregation is a lighter form of Composition, where a sub-part object can meaningfully exist without main objects.  
  
5) In Java, you can use [final keyword](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) to represent Composition. Since in Composition, Owner object expects a part object to be available and functions, by making it final, your provide guarantee that, when Owner will be created, this part object will exist. This is actually a *Java idiom to represent a strong form of association* i.e. composition between two objects.  
  
6) Another interesting word, which comes handy to understand difference between Composition and Aggregation in software design is "part-of" and "has". If one object is-part-of another object e.g. Engine is part of Car, then association or relationship between them is Composition. On the other hand, if one object just has another object e.g. Car has the driver then it's Aggregation.  
  
Read more: <http://javarevisited.blogspot.com/2014/02/ifference-between-association-vs-composition-vs-aggregation.html#ixzz4jDUmZta9>

**8. Difference between Abstract class vs Interface in Java**

**Ans:**

While deciding *when to use interface and abstract class*, it’s important to know difference between abstract class and interface in Java. In my opinion, following two differences between them drives decision about when to use abstract class or interface in Java.

1) Interface in Java can only contains declaration. You can not declare any concrete methods inside interface. On the other hand abstract class may contain both abstract and concrete methods, which makes abstract class an ideal place to provide common or default functionality. I suggest reading my post [10 things to know about interface in Java](http://javarevisited.blogspot.com/2012/04/10-points-on-interface-in-java-with.html) to know more about interfaces, particularly in Java programming language.

2) Java interface can extend multiple interface also Java class can implement multiple interfaces, Which means interface can provide more Polymorphism support than abstract class . By extending abstract class, a class can only participate in one Type hierarchy but by using interface it can be part of multiple type hierarchies. E.g. a class can be Runnable and Displayable at same time. One example I can remember of this is writing GUI application in J2ME, where  class extends Canvas and implements CommandListener to provide both graphic and event-handling functionality..

3) In order to implement interface in Java, until your class is abstract, you need to provide implementation of all methods, which is very painful. On the other hand abstract class may help you in this case by providing default implementation. Because of this reason, I prefer to have minimum methods in interface, starting from just one, I don't like idea of [marker interface](http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html), once annotation is introduced in Java 5. If you look JDK or any framework like Spring, which I does to understand OOPS and design patter better, you will find that most of interface contains only one or two methods e.g. Runnable, Callable, ActionListener etc.

I haven't included all syntactical difference between abstract class and interface in Java here, because focus here to learn when to use abstract class and interface and choosing one over other. Nevertheless you can see [difference between interface and abstract class](http://java67.blogspot.com/2012/09/what-is-difference-between-interface-abstract-class-java.html) to find  all those syntactical differences.

**When to use interface and abstract class in Java**

1) In Java particularly, decision between choosing Abstract class and interface may influence by the fact that multiple inheritance is not supported in Java. One class can only extend another class in Java. If you choose abstract class over interface than you lost your chance to extend another class, while at the same time you can implement multiple interfaces to show that you have multiple capability. One of the common example, in favor of interface over abstract class is [Thread vs Runnable](http://javarevisited.blogspot.com/2012/01/difference-thread-vs-runnable-interface.html) case. If you want to execute a task and need run() method it's better to implement Runnable interface than extending Thread class.

2) Let's see another case where an abstract class suits better than interface. Since abstract class can include concrete methods, it’s great for maintenance point of view, particularly when your base class is evolving and keep changing. If you need a functionality across all your implementation e.g. a common method, than, you need to change every single implementation to include that change if  you have chosen interface to describe your base class. Abstract class comes handy in this case because you can just define new functionality in abstract super class and every sub class will automatically gets it. In short, abstract class are great in terms of evolving functionality. If you are using interface, you need to exercise extra care while defining contracts because its not easy to change them once published.

3) Interface in Java is great for defining Types. Programming for interfaces than implementation is also one of the useful [Object oriented design principle](http://javarevisited.blogspot.de/2012/03/10-object-oriented-design-principles.html) which suggests benefit of using interface as argument to function, return type etc.

4) One more general rule of when to use abstract class and interface is to find out whether a certain class will form a IS-A hierarchy or CAN-DO-THIS hierarchy. If you know that you will be creating classes e.g. Circle, Square than it's better to create an abstract class Shape which can have area() and perimeter() as abstract method, rather than defining Shape as interface in Java. On the other hand if you are going to create classes which can do thinks like, can fly, you can use interface Flyable instead of abstract class.

5) Interface generally define capability e.g. Runnable can run(), Callable can call(), Displayable can display(). So if you need to define capability, consider using interface. Since a class can have multiple capabilities i.e. a class can be Runnable as well as Displayable at same time. As discussed in first point, Since [java does not allow multiple inheritance at class level](http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html), only way to provide multiple capability is via interfaces.

6) Let's see another example of where to use Abstract class and Interface in Java, which is related to earlier point. Suppose you have lot of classes to model which are birds, which can fly, than creating a base abstract class as Bird would be appropriate  but if you have to model other things along with Birds, which can fly e.g. Airplanes, Balloons or Kites than it's better to create interface Flyable to represent flying functionality. In conclusion, if you need to provide a functionality which is used by same type of class than use Abstract class and if functionality can be used by completely unrelated classes than use interface.

7) Another interesting use of Abstract class and interface is defining contract using interface and providing skeletal using abstract class. java.util.List from Java collection framework is a good example of this pattern. List is declared as interface and extends Collection and Iterable interface and AbstractList is an abstract class which implements List. AbstractList provides skeletal implementation of List interface. Benefit of using this approach is that it minimize the effort to implement this interface by concrete class e.g. [ArrayList or LinkedList](http://java67.blogspot.com/2012/12/difference-between-arraylist-vs-LinkedList-java.html). If you don't use skeletal implementation e.g. abstract class and instead decide to implement List interface than not only you need to implement all List methods but also you might be duplicating common code. Abstract class in this case reduce effort to implement interface.

8) Interface also provide more decoupling than abstract class because interface doesn't contain any implementation detail, while abstract class may contain default implementation which may couple them with other class or resource.

9) Using interface also help while implementing [Dependency Injection design pattern](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html) and makes testing easy. Many mock testing framework utilize this behavior.

Read more: <http://javarevisited.blogspot.com/2013/05/difference-between-abstract-class-vs-interface-java-when-prefer-over-design-oops.html#ixzz4jDWvStLo>

**9. What is Polymorphism in Java? Overriding or Overloading?**

**Ans:**

Both [overloading](http://java67.blogspot.sg/2012/08/what-is-method-overloading-in-java-example.html) and the [overriding](http://java67.blogspot.sg/2012/08/what-is-method-overriding-in-java-example-tutorial.html) concept are applied on methods in Java. Since polymorphism literally means taking multiple forms, So even though you have the name of the method same in the case of overloading and overriding, an actual method called can be any of those multiple methods with the same name. Let's see some more details on [method overloading and overriding](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html) to understand how polymorphism relates to overloading and overriding and How they are different.

Polymorphism vs Overriding

Overriding is a form of polymorphism which is used in Java to dynamically bind method from the subclass in response to a method call from sub class object referenced by superclass type. Method overriding is bonded using [dynamic binding in Java](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html).

### Suppose you have two methods size() in both base class and derived class and Base class variable is pointing to an object which happens to be subclass object at runtime then method from subclass will be called, i.e. overridden method will be called. This allows to program for interface than implementation, a popular [OOPS design principle](http://www.blogger.com/javarevisited.blogspot.de/2012/03/10-object-oriented-design-principles.html) because Polymorphism guarantees to invoke correct method based upon the object. Method overriding is key for much flexible design pattern in Java. Polymorphism vs Overloading

Method overloading is another form of Polymorphism though some people argue against that. In the case of overloading, you also got multiple methods with the same name but different method signature but a call to correct method is resolved at compile time using [static binding in Java](http://javarevisited.blogspot.de/2012/03/what-is-static-and-dynamic-binding-in.html). Overloading is a compile time activity oppose to Overriding which is runtime activity. Because of this reason overloading is faster than method overriding in Java. Though beware with an overloaded method which creates conflict e.g. methods with only one parameter e.g. int and long etc. See [What are method overloading in Java](http://java67.blogspot.sg/2012/08/what-is-method-overloading-in-java-example.html) for example and complete details.

## An example of Polymorphism in Java

Let's see a short example of Polymorphism in Java. In this example, Pet variable behaves polymorphic because it can be either Cat or Dog. this is also an example of method overriding because makeSound() method is overridden in [subclass](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) Dog and Cat.

**import** java.util.ArrayList;  
**import** java.util.List;  
  
**abstract** **class** Pet{  
    **public** **abstract** **void** makeSound();  
}  
**class** Cat **extends** Pet{  
    @**Override**  
    **public** **void** makeSound() {  
        **System**.out.println("Meow");  
    }     
}  
**class** Dog **extends** Pet{  
  
    @**Override**  
    **public** **void** makeSound() {  
        **System**.out.println("Woof");  
    }  
}

Let's test How Polymorphism concept work in Java:

/\*\*  
 \* Java program to demonstrate What is Polymorphism  
 \* @author Javin Paul  
 \*/  
**public** **class** PolymorphismDemo{  
  
    **public** **static** **void** main(**String** args[]) {  
        *//Now Pet will show How Polymorphism work in Java*  
        **List**<Pet> pets = **new** **ArrayList**<Pet>();  
        pets.add(**new** Cat());  
        pets.add(**new** Dog());  
        
        *//pet variable which is type of Pet behave different based*  
        *//upon whether pet is Cat or Dog*  
        for(Pet pet : pets){  
            pet.makeSound();  
        }  
    }  
}  
  
Output:  
Meow  
Woof  
  
Read more: <http://www.java67.com/2012/10/difference-between-polymorphism-overloading-overriding-java.html#ixzz4jDYVHKCz>

**10. What is difference between Overloading and Overriding in Java  
Ans:**

1) First and major difference between Overloading and Overriding is that former occur during [compile time](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html) while later occur during runtime.  
  
2) Second [difference between Overloading and Overriding](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html) is that, you can overload method in same class but you can only override method in sub class.  
  
3) Third difference is that you [can overload static method in Java](http://java67.blogspot.sg/2012/08/can-we-overload-static-method-in-java.html) but you [can not override static method in Java](http://java67.blogspot.sg/2012/08/can-we-override-static-method-in-java.html). In fact when you declare same method in Sub Class it's known as method hiding because it hide super class method instead of overriding it.  
  
4) Overloaded methods are bonded using [static binding](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html) and Type of reference variable is used, while Overridden method are bonded using [dynamic bondin](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html)g based upon actual Object.  
  
5) [Rules of Overloading and Overriding](http://java67.blogspot.sg/2012/09/what-is-rules-of-overloading-and-overriding-in-java.html) is different in Java. In order to overload a method you need to change its method signature but that is not required for overriding any method in Java.  
  
6) Another difference between method overloading and overriding is that private and final method can not be overridden but can be overloaded in Java.  
  
7) Overloaded method are fast as compare to Overridden method in Java.  
  
Read more: <http://www.java67.com/2012/09/difference-between-overloading-vs-overriding-in-java.html#ixzz4jDZXXsxF>

# 11. Difference between instance and Object in Java

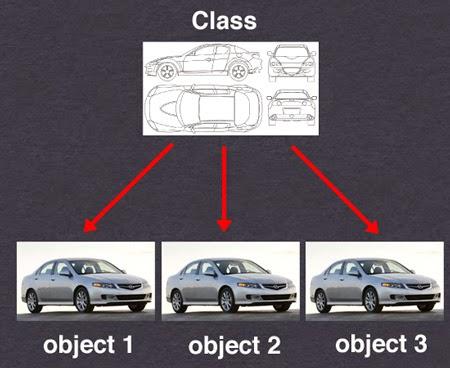
**Ans:**

## In Java functions are known as methods, similarly, *objects are known as instances in Java*. You have a class, which represent a blueprint of a real world thing e.g. Car, and object represents a real world car e.g. your car, my car, a red car or a blue car. They are also known as instances of the car.

## One reason of calling instance may be because they are a representation of that class at a particular instant. In practice, use the instance to say about one particular object, and use an object to talk about many objects. By the way, it's also worth remembering that Java has a class named Object, or java.lang.Object, which is the master class and every other class extend it.

## This is another reason why using instance is better because it will minimize confusion. So use Object when you want to talk about java.lang.Object and use instance when you want to talk about the object of OOPS. Object vs Instances

The basic concept of Object Oriented Programming (OOP) revolves around two things, Class, and Object. The class is the blueprint. The Object is an actual thing that is made up using that 'blueprint' (like the car example given above).  You cannot see the instances, all you see is code, which is your class. Object or instance are created at run-time and they are created in a specific memory area called [heap memory](http://java67.blogspot.sg/2013/08/guide-of-javalangoutofmemoryerror-java-heap-space-tomcat-eclipse-minecraft-jboss.html).  
  
Each instance consumes some memory depending upon how much and what value you store. For example "Java" is an instance of String class and holds memory required to represent those characters and to store some metadata. You might have also heard about class method vs instance methods, right? look we hardly call object method.  
  
There is no harm calling instance as an object but if you are following rest of Java convention then why not this one. Do you call Java method function? No right, then there is no point calling instance as an object, it will just create confusion nothing more.  
  
If you are senior Java developer or a trainer then it's your responsibility to pass right terminology to junior developers. Whatever they will hear from you, it will go a long way, so make sure you feed clear and concise information.



That's all about the **difference between Object and Instance in Java**. In general, it’s better to treat instance and object as the same thing to avoid confusion, but if you follow Java convention, better call them instance. By the way no matter, what you do;  people will use it as per their convenience and you can't argue with everyone that, no you are talking about the instance, please use instance word, or, No, you are talking about an actual object, please use object word etc. I would derive explanation based upon context. In short, use Object to talk about java.lang.Object class and use instance to talk about the object of OOP.  
  
Read more: <http://www.java67.com/2014/11/difference-between-instance-and-object-in-java.html#ixzz4jDaTE9Z4>

**12. Difference between Static and Dynamic Binding in Java  
Ans:**

1) Static binding is resolved at compile time, while Dynamic binding is resolved at runtime.  
  
2) Static binding only uses Type information, and method resolution is based upon type of reference variable, while dynamic or late binding resolves method based upon actual object.  
  
3) In Java programming language, private, static and final method are resolved using static binding, while only virtual methods are resolved using dynamic binding.  
  
4) True Polymorphism is achieved using dynamic binding, and its key of many design principles e.g. Strategy pattern, Open closed design pattern etc.

That's all about *difference between static and dynamic binding in Java*. You can play around this code to try different combination of method calling e.g. calling a sub class method using Super class object or vice-versa. Let me know if you have question related to understanding static or dynamic binding in Java.  
  
Read more: <http://www.java67.com/2014/02/difference-between-static-and-dynamic.html#ixzz4jDb9btRS>