**Q) How many types of memory areas are allocated by JVM?**

**Ans**. JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

An implementation Its implementation is known as JRE (Java Runtime Environment).

Runtime Instance Whenever We write java command on the command prompt to run the java class**, an instance of JVM is created.**

**The JVM performs following operation:**

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

**JVM provides definitions for the:**

* Memory area
* Class file format
* Register set
* Garbage-collected heap
* Fatal error reporting etc.

**JVM Architecture**

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



**1) Classloader**

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. **There are three built-in classloaders in Java.**

**Bootstrap ClassLoader:** This is the first classloader which is the super class of Extension classloader. It **loads the rt.jar** file which contains all class files of Java Standard Edition like **java.lang package** classes**, java.net package** classes, **java.util package** classes, java.io package classes, **java.sql package** classes etc.

**Extension ClassLoader:** This is the child classloader of Bootstrap and parent classloader of System classloader. It loades the jar files located inside $JAVA\_HOME/jre/lib/ext directory.

**System/Application ClassLoader:** This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. We can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

//Let's see an example to print the classloader name

public class ClassLoaderExample

{

    public static void main(String[] args)

    {

        // Let's print the classloader name of current class.

        //Application/System classloader will load this class

        Class c=ClassLoaderExample.class;

        System.out.println(c.getClassLoader());  // sun.misc.Launcher$AppClassLoader@4e0e2f2a

        //If we print the classloader name of String, it will print null because it is an

        //in-built class which is found in rt.jar, so it is loaded by Bootstrap classloader

        System.out.println(String.class.getClassLoader());  //null

    }

}

Test it Now

Output:

sun.misc.Launcher$AppClassLoader@4e0e2f2a

null

These are the internal classloaders provided by Java. If we want to create your own classloader,

1.we need to extend the ClassLoader class.

2) Class(Method) Area

**Class(Method)** Area stores per-class structures such as the runtime **constant pool, field and method data**, the code for methods.

**3) Heap**

It is the runtime data area in which objects are allocated.

**4) Stack**

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

**5) Program Counter Register**

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

**6) Native Method Stack**

It contains all the native methods used in the application.

**7) Execution Engine**

It contains:

A virtual processor

Interpreter: Read bytecode stream then execute the instructions.

**Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

8) Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

**How ClassLoader Works in Java:**

Java class loaders are used to load classes at runtime. **ClassLoader in Java works on three principles: delegation, visibility, and uniqueness.** **Delegation principle** forward request of class loading to parent class loader and only loads the class if the parent is not able to find or load the class. **Visibility principle** allows child class loader to see all the classes loaded by parent ClassLoader, but parent class loader can not see classes loaded by a child. **Uniqueness principle** allows one to load a class exactly once, which is basically achieved by delegation and ensures that child ClassLoader doesn't reload the class already loaded by a parent. Correct understanding of class loader is a must to resolve issues like NoClassDefFoundError in Java and java.lang.ClassNotFoundException, which are related to class loading.  
  
What is ClassLoader in Java

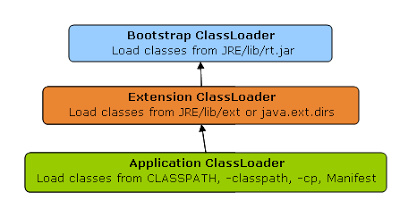
**ClassLoader in Java is a class that is used to load class files in Java**. Java code is compiled into a class file by javac compiler and JVM executes Java program, by executing byte codes written in the class file. ClassLoader is responsible for loading class files from file systems, networks, or any other source.   
  
There is three default class loader used in Java**, Bootstrap, Extension, and System or Application class loader.**   
Every class loader has a predefined location, from where they load class files. **The bootstrap class loader is responsible for loading standard JDK class files from rt.jar** and it is the parent of all class loaders in Java.   
  
The bootstrap class loader doesn't have any parents if we call String.class.getClassLoader() it will return null and any code based on that may throw NullPointerException in Java. The **bootstrap class loader** is also known as **Primordial ClassLoader in Java.**    
  
  
**Extension ClassLoader** delegates class loading request to its parent, Bootstrap, and if unsuccessful, loads class form jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by  sun.misc.Launcher$ExtClassLoader.   
  
The third default class loader used by JVM to load Java classes is called **System or Application class loader** and it is responsible for loading application specific classes from CLASSPATH environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR.   
  
**Application class loader** is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$AppClassLoader class. Also, except for the Bootstrap class loader, which is implemented in the native language mostly in C,  all  Java class loaders are implemented using java.lang.ClassLoader.

In short here is the location from which Bootstrap, Extension, and Application ClassLoader load Class files.

1) **Bootstrap ClassLoader** - **JRE/lib/rt.jar**

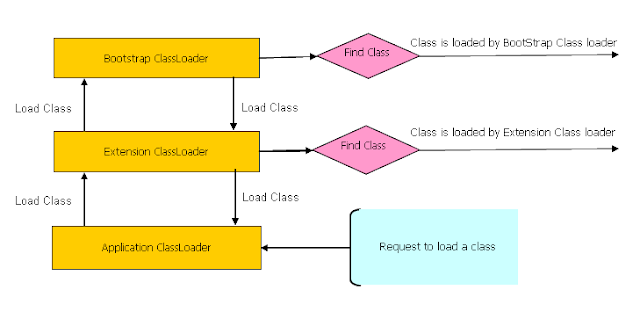
2) **Extension ClassLoader** - **JRE/lib/ext or any directory denoted by java.ext.dirs**

3) **Application ClassLoader** - CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside JAR file.



**How ClassLoader works in Java?**

**Java ClassLoader works in three principles:** **delegation**, **visibility**, and **uniqueness**. In this section, we will see those rules in detail and understand the working of Java ClassLoader with example. By the way here is a diagram that explains How ClassLoader to load class in Java using delegation.



**1. Delegation principles**

As discussed on when a class is loaded and initialized in Java, a class is loaded in Java, when it's needed. Suppose we have an application-specific class called Abc.class, the first request of loading this class will come to Application ClassLoader which will delegate to its parent Extension ClassLoader which further delegates to Primordial or Bootstrap class loader.   
  
Primordial will look for that class in rt.jar and since that class is not there, a request comes to Extension class loader which looks on jre/lib/ext directory and tries to locate this class there, if the class is found there than Extension class loader will load that class and Application class loader will never load that class but if it's not loaded by extension class-loader than Application class loader loads it from Classpath in Java. Remember Classpath is used to load class files while PATH is used to locate executables like javac or java command.

**2. Visibility Principle**

**According to the visibility principle**, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. This means if class Abc is loaded by Application class loader than trying to load class ABC explicitly using extension ClassLoader will throw either java.lang.ClassNotFoundException. as shown in the below Example

package test;  
  
import java.util.logging.Level;  
import java.util.logging.Logger;  
  
/\*\*  
 \* Java program to demonstrate How ClassLoader works in Java,

 \* in particular about the visibility principle of ClassLoader.

 \*  
 \* @author Javin Paul  
 \*/  
  
public class ClassLoaderTest {  
   
    public static void main(String args[]) {  
        try {           
            //printing ClassLoader of this class  
            System.out.println("ClassLoaderTest.getClass().getClassLoader() : "  
                                 + ClassLoaderTest.class.**getClassLoader**());  
  
           
            //trying to explicitly load this class again using Extension class loader  
            Class.forName("test.ClassLoaderTest", true   
                            ,  ClassLoaderTest.class.getClassLoader().getParent());  
        } catch (ClassNotFoundException ex) {  
            Logger.getLogger(ClassLoaderTest.class.getName()).log(Level.SEVERE, null, ex);  
        }  
    }  
  
}  
  
Output:  
ClassLoaderTest.getClass().getClassLoader() : sun.misc.Launcher$AppClassLoader@601bb1  
16/08/2012 2:43:48 AM test.ClassLoaderTest main  
SEVERE: null  
java.lang.ClassNotFoundException: test.ClassLoaderTest  
        at java.net.URLClassLoader$1.run(URLClassLoader.java:202)  
        at java.security.AccessController.doPrivileged(Native Method)  
        at java.net.URLClassLoader.findClass(URLClassLoader.java:190)  
        at sun.misc.Launcher$ExtClassLoader.findClass(Launcher.java:229)  
        at java.lang.ClassLoader.loadClass(ClassLoader.java:306)  
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)  
        at java.lang.Class.forName0(Native Method)  
        at java.lang.Class.forName(Class.java:247)  
        at test.ClassLoaderTest.main(ClassLoaderTest.java:29)

**3. Uniqueness Principle**

According to this principle, a class loaded by Parent should not be loaded by Child ClassLoader again. Though it is completely possible to write a class loader which violates Delegation and Uniqueness principles and loads class by itself, it's not something which is beneficial. We should follow all class loader principles while writing your own ClassLoader.

How to load class explicitly in Java

Java provides API to explicitly load a class by Class.forName(classname) and Class.forName(classname, initialized, classloader), remember JDBC code which is used to load JDBC drives we have seen in Java program to Connect Oracle database.   
  
As shown in the above example we can pass the name of ClassLoader which should be used to load that particular class along with the binary name of the class. Class is loaded by calling loadClass() method of java.lang.ClassLoader class which calls findClass() method to locate bytecodes for the corresponding class.   
  
In this example, Extension ClassLoader uses java.net.URLClassLoader which searches for class files and resources in JAR and directories. any search path which is ended using "/" is considered a directory.   
  
If findClass() does not found the class than it throws java.lang.ClassNotFoundException and if it finds it calls defineClass() to convert bytecodes into a .class instance which is returned to the caller.

**Where to use ClassLoader in Java?**

ClassLoader in Java is a powerful concept and used in many places. One of the popular examples of ClassLoader is AppletClassLoader which is used to load a class by Applet since Applets are mostly loaded from the internet rather than a local file system.  
  
By using separate ClassLoader we can also load the same class from multiple sources and they will be treated as different classes in JVM. J2EE uses multiple class loaders to load a class from a different location like classes from the WAR file will be loaded by Web-app ClassLoader while classes bundled in EJB-JAR is loaded by another classloader.   
  
Some web server also supports hot deploy functionality which is implemented using ClassLoader. We can also use ClassLoader to load classes from a database or any other persistent store.

**Q. What is method overloading in Java?  
Ans.** Method overloading in Java is a programming concept **when programmer declares two methods of the same name but with different method signature,** e.g. change in the argument list or change in the type of argument. method overloading is a powerful Java programming technique to declare a method which does a similar performance but with a different kind of input. One of the most popular **examples of method overloading is System.out.println() method which is overloaded to accept all kinds of data types in Java.** we have println() method which takes String, int, float,double or even char in output.

**properties of method overloading in Java**

Overloaded methods are bonded using **static binding** in Java. Which **occurs during compile time** i.e. when we compile Java program. During the compilation process, compiler bind method calls to the actual method.  
2) **Overloaded methods are fast** because they are bonded during compile time and no check or binding is required during runtime.  
3) Most important rule of method overloading in Java is that two overloaded methods must have a different signature.  
1) A number of argument to a method is part of method signature.  
2) Type of argument to a method is also part of method signature  
3) Order of argument also forms part of method signature provided they are of different type.  
**4) The return type of method is not part of the method signature in Java.**  
  
Method Overloading Example in Java

Here is a list of method and there corresponding overloaded method with reason that How they are overloaded :  
Original method :

 public void  show(String message){

      System.out.println(message);

}

Overloaded method : number of argument is different

 public void  show(String message, boolean show){

      System.out.println(message);

}

Overloaded method : type of argument is different

 public void  show(Integer message){

      System.out.println(message);

}  
Not a Overloaded method : only return type is different

 public boolean show(String message){

      System.out.println(message);

      return false;

}

In the summary method, overloading means multiple methods with the same name but with a different signature. **remember return type is not part of the method signature.** method overloading is also completely different from method overriding which is a similar concept and we will see in the next article

**Method overriding in Java**

Method overriding in Java is a concept based on polymorphism OOPS concept which allows the programmer to create two methods with the same name and method signature on the interface and its various implementation and the actual method is called at runtime depending upon the type of an object at runtime. Method overriding allows us to write flexible and extensible code in Java because we can introduce new functionality with the minimal code change.    
In method overloading, *only name of two overloaded methods is the same but method signature must be different while in the method overriding, the method signature must be the same*. method overriding represent true polymorphic behaviour, where the only name needs to be the same underlying method logic can be different.

Rules of method overriding in Java:-

There are few rules which needs to be followed while overriding any method in Java, failure to follow these rules results in a compile-time error in Java.

1) The first and most important rule regarding method overriding in Java is that **we can only override a method in a subclass. we cannot override the method in the same class.**

2) A second important rule of method overriding in Java that name and signature of **method must be the same in Super class and Sub class** or **in the interface and its implementation.**

3) The third rule to override a method in Java is that the overriding **method can not reduce the accessibility of the overridden method in Java**. For example, if the overridden method is public than the overriding method can not be protected, private or package-private;   
But the opposite is true overriding method can increase the accessibility of method in Java, i.e. if the overridden method is protected than The overriding method can be protected or public.

4) Another worth noting rule of method overriding in Java is that **the overriding  method can not throw checked Exception which is higher in the hierarchy than the overridden method**. This means if the **overridden method throws IOException than the overriding method can not throw java.lang.Exception** in its throws clause because of java.lang.Exception comes higher than IOException in Exception hierarchy.   
  
This rule doesn't apply to RuntimeException in Java, which is not even needed to be declared in a throws clause in Java.

5) **we cannot override private, static and final the method in Java**. *private and static method are bonded during compile time using static binding in Java* and doesn't resolve during runtime. **the overriding the final method in Java is a compile-time error**. Though private and static methods can be hidden if we declare another method with the same and signature in the subclass.

6) The **overridden method is called using dynamic binding in Java at runtime** based upon the type of Object.

7) If we are extending the abstract class or implementing interface than we need to override all abstract methods unless our class is not abstract. an abstract method can only be used by using method overriding.

8) Always use **@Override** annotation while the overriding method in Java. Though this is not a rule but its one of the best Java coding practices to follow. From Java 6 we can use @Override annotation on a method inherited from the interface as well.

**Method Overriding Example in Java**

we have used the Runnable interface which has an abstract run() method. We have two class Task and PeriodicTask which implements the Runnable interface and override run method.   
  
For the purpose of demonstrating how method overriding works in Java we are calling run() method in same thread, which we should not, see difference between run and start method to know why. Because run() is overridden in two separate class, call to run() method will be resolved during runtime depending upon type of Object.

/\*\*  
 \*  
 \* Java program to demonstrate how to override a method in Java.  
 \* Overridden method are resolved during runtime based upon the type of object  
 \*  
 \* @author Javin  
 \*/  
**public class** CollectionTest {  
    public static void main(String args[]) {  
      Runnable task = new Task();  
     task.run(); //call overridden method in Task  
      task = new PeriodicTask();  
      task.run(); //calls overridden method in PeriodicTas  
    }}  
class Task implements Runnable{  
    @Override  
    public void run() {  
        System.out.println("Run method overridden in Task class");  
    }   
}  
class PeriodicTask extends Task{  
     @Override  
    public void run() {  
        System.err.println("overridden method run() in PeriodicTask class");  
    }  
}  
  
Output:  
Run method is overridden in Task class  
overridden method run() in PeriodicTask class

The overridden method is also slower as compared to static and final methods because of dynamic binding but it provides we flexibility, many popular Object-oriented design principles are based upon method overriding in Java.

**Q. What is the method of hiding in Java?**When we declare two static methods with same name and signature in both superclass and subclass then they hide each other i.e. a call to the method in the subclass will call the static method declared in that class and a call to the same method is superclass is resolved to the static method declared in the super-class.  
**Q. Is Java a Pure Object Oriented Programming Language?**

The short answer is no. My answer is based on the fact that in a pure object-oriented language everything is an object and there are many things in Java that are not objects like primitive **data types like boolean, char, short, int, long, float, double, different kinds of arithmetic, logical and bitwise operator like +, -. \*, /, &&, || etc.** There are only a few pure OO programming languages are Smalltalk and Eiffel, If there is more, I may not know but Smalltalk is often touted as the purest form of an object-oriented language.  
Though Java is probably the most successful Object-oriented programming language, which also got some **functional programming touch in Java 8**, it has been never considered 100% or pure object-oriented programming language. If it were, all its primitives would be objects.  
It actually moves half-way in this direction with String (and perhaps Array), but it doesn't quite go far enough. Actually, one could argue that as String and Array aren't inheritable, that makes those parts of Java at best object-based.

**Q. Is Java is Pure Object-Oriented language?**

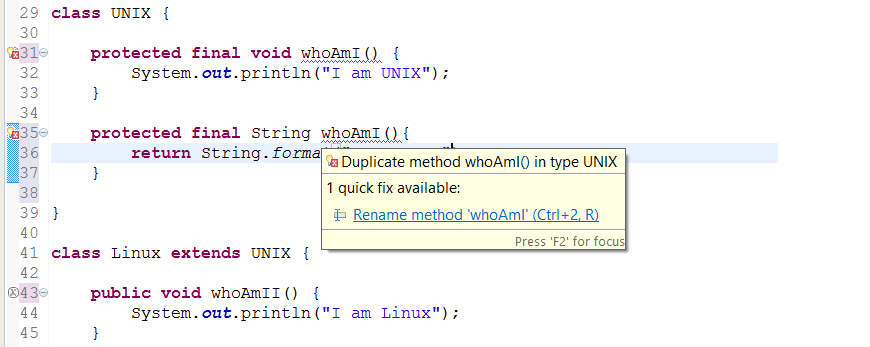
There are seven qualities to be satisfied for a programming language to be pure Object Oriented. They are:  
1. Encapsulation/Data Hiding  
2. Inheritance  
3. Polymorphism  
4. Abstraction  
5*. All predefined types are objects  
6. All operations are performed by sending messages to objects*7. All user-defined types are objects.  
If we look at these seven qualities, Java does satisfy most of them. Java supports Encapsulation at class and package level, It supports Abstraction, Inheritance, and Polymorphism, and all user-defined types are also objects.  
What it doesn't support is #5, all predefined types are not objects in Java, **because we can define primitive types.** **This means it also violates #6. That's why Java is not a pure object-oriented language.**

**Q. Why Java is not Pure Object-Oriented language?**

Smalltalk is often considered one of the purest Object-oriented languages and comparing Java with Smalltalk will give we sufficient reasons, why Java is not 100% object-oriented language. The following point makes sense to me.

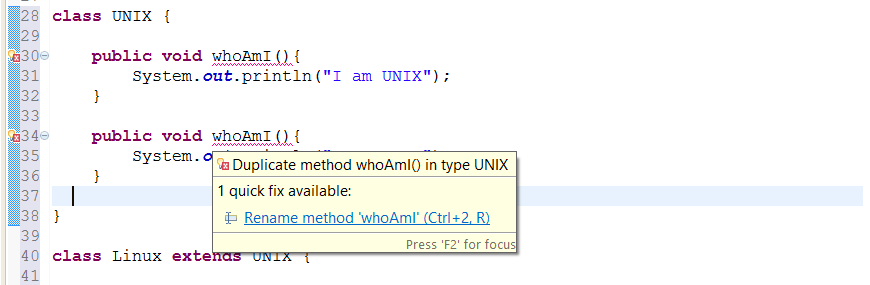
1) Primitive data types are either stored directly in fields or on the stack rather than on the heap. This is the reason Java is not considered a pure OO programming language  
2) If we have to classify in "pure OO" or "non-pure OO" we have a problem. It is better to talk about "purity levels". Smalltalk has a higher purity level than Java. "Primitive types" in Smalltalk are actually "Primitive Classes" and in Smalltalk, all "procedures" or "functions" are really messages  
  
On closing notes, We can make our program pure object-oriented by using Autoboxing, but *Java compiler supports primitive data types, so Java cannot be Pure object-oriented unless it makes everything in terms of objects.*  
  
 **Rules of Method Overloading and Overriding in Java**

2) **Return type of method is not part of method signature**, so just changing the return type will not overload a method in Java.  In fact, just changing the return type will result in compile time error as "duplicate method X in type Y. Here is a screenshot of that error in Eclipse :

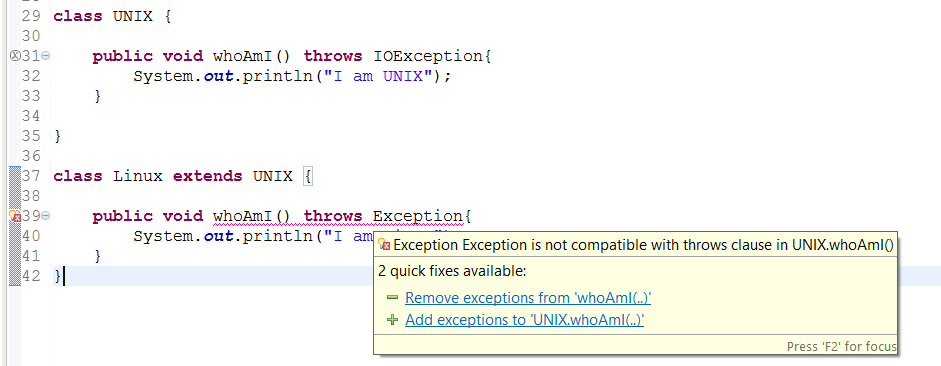


Method Overriding Rules in Java

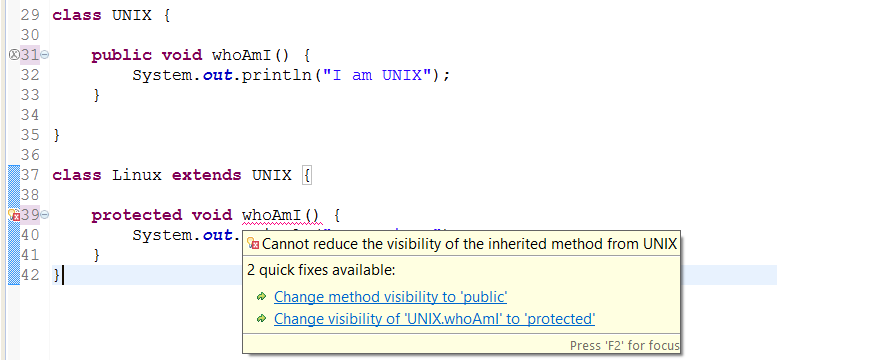
Overriding is completely different than overloading and so it's rules are also different. For terminology, original method is known as **overridden** method and new method is known as **overriding** method. Following rules must be followed to correctly override a method in Java :  
1) A method can only be overridden in sub class, not in same class. If we try to create two methods with same signature in one class compiler will complain about it saying "duplicate method in type Class", as shown in following screenshot :



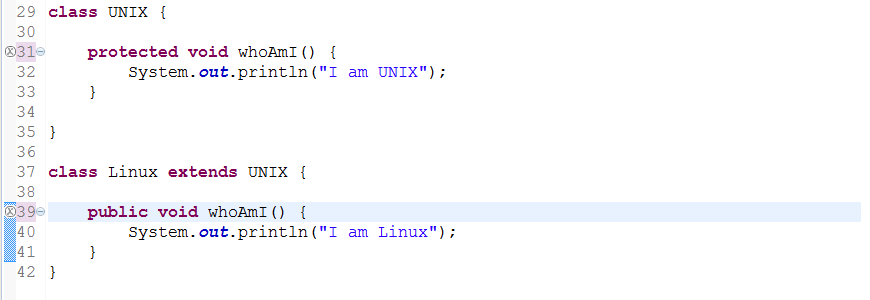
2) Overriding method cannot throw checked Exception which is higher in hierarchy, than checked Exception thrown by overridden method. For example if overridden method throws IOException or ClassNotfoundException, which are checked Exception, than overriding method can not throw java.lang.Exception because it comes higher in type hierarchy (it's super class of IOException and ClassNotFoundExcepiton). If we do so, compiler will catch we as seen in following image :

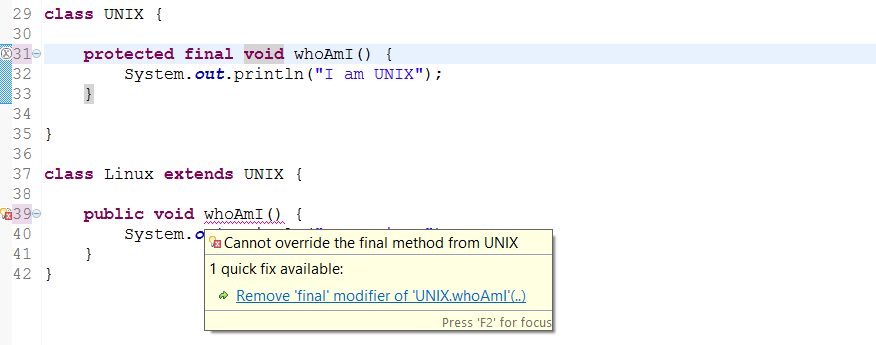


3) Overriding method can not reduce access of overridden method. It means if overridden method is defined as public than overriding method can not be protected or package private. Similarly if original method is protected then overriding method cannot be package-private. We can see what happens if we violate this rule in Java, as seen in this screenshot it will throw compile time error saying "We cannot reduce visibility of inherited method of a class".

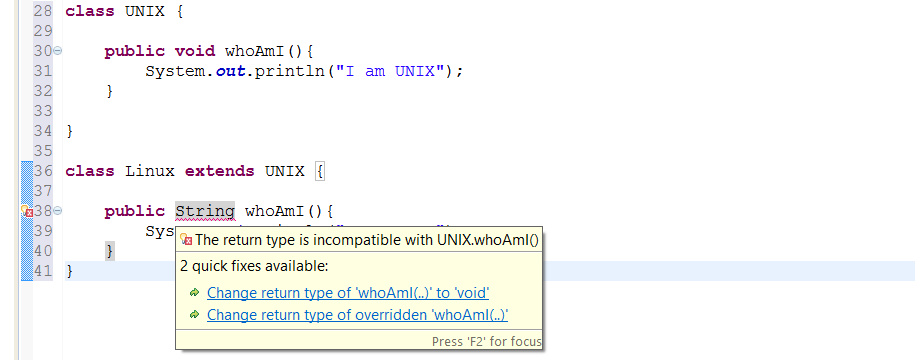


4) Overriding method can increase access of overridden method. This is opposite of earlier rule, according to this if overridden method is declared as protected than overriding method can be protected or public. Here is an example to see that it's allowed in Java :



5**) private, static and final method can not be overridden in Java**. we can hide private and static method but trying to override final method will result in compile time error "Cannot override the final method from a class" as shown in below screenshot :

6) Return type of overriding method must be same as overridden method. Trying to change return type of method in child class will throw compile time error "return type is incompatible with parent class method" as shown in following screenshot.

[](https://4.bp.blogspot.com/-lzbzFVMIxHk/VDafe9HmJQI/AAAAAAAACAU/81dKojfi_X0/s1600/Method%2BOverriding%2BCannot%2Bbe%2Bdone%2Bby%2Bjust%2Bchanging%2Breturn%2Btype.png)

**Difference between Overloading and Overriding in Java**

1) First and major difference between Overloading and Overriding is that former occur during **compile time while later occur during runtime**.  
2) Second difference between Overloading and Overriding is that, we can overload method in same class but we can only override method in sub class.  
3) Third difference is that we can overload static method in Java but we cannot override static method in Java. In fact when we 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 and Type of reference variable is used, while the Overridden method is bonded using dynamic bonding based upon actual Object.  
5) Rules of Overloading and Overriding is different in Java. In order to overload a method we 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 methods can not be overridden but can be overloaded in Java.**7) The overloaded method are fast as compare to the Overridden method in Java.

**Q. Can we overload static method in Java Program -?**

**Yes**, we can overload static method in Java. In terms of method overloading static method is just like normal methods and in order to overload the static method we need to provide another static method with the same name but different method signature. Static overloaded method is resolved using Static Binding during compile time. Overloading method in Java is completely different than the overriding method and as discussed in the last article we can not override static method in Java but we can certainly overload a static method in Java. Here is an example which confirms that we can overload static method in Java:

In this example, we have a static method called **greet**(**String** name) which takes a String argument as name and print a default greeting message as "Hello John". Now to show that we can overload static method in Java I have provided another static method with the same name but a different method signature which not only takes the name of a person to greet but also greeting message likeGood Morning, Good Evening etc.

/\*\*     
 \* Java program to show that we can overload static method in Java.  
 \*/  
public class StaticOverloadingTest {  
    public static void main(String args[]) {  
        greet("John"); //will call a static method with one String argument  
        greet("John", "Good Morning"); //overloaded static method will be call      
    }   
    /\*  
     \* static method which will be overloaded  
     \*/  
    public static void greet(String name){  
        System.out.println("Hello " + name);  
    }   
    /\*  
     \* Another static method which overloads above Hello method  
     \* This shows that we can overload static method in Java  
     \*/  
    public static void greet(String name, String greeting){  
        System.out.println(greeting + " " + name);  
    }  
}  
Output  
Hello John  
Good Morning John

**Q. Can we override static method in Java(Method hiding)**

**No**, we cannot override static method in Java because method overriding is based upon dynamic binding at runtime and static methods are bonded using static binding at compile time. Though we can declare a method with same name and method signature in sub class which does look like we can override static method in Java but in **reality that is method hiding.** Java won't resolve method call at runtime and depending upon type of Object which is used to call static method, corresponding method will be called. It means if we use Parent class's type to call static method, original static will be called from patent class, on the other hand if we use Child class's type to call static method, method from child class will be called.

/\*\*  
 \*  
 \* Java program which demonstrate that we can not override static method in Java.  
 \* Had Static method can be overridden, with Super class type and sub class object  
 \* static method from sub class would be called in our example, which is not the case.  
 \* @author  
 \*/  
public class CanWeOverrideStaticMethod {  
     public static void main(String args[]) {  
         Screen scrn = new ColorScreen();  
        //if we can  override static , this should call method from Child class  
        scrn.show(); //IDE will show warning, static method should be called from classname  
    }   
 }  
class Screen{  
   /\*  
     \* public static method which can not be overridden in Java  
     \*/  
    public static void show(){  
        System.out.printf("Static method from parent class");  
    }  
}  
class ColorScreen extends Screen{  
    /\*  
     \* static method of same name and method signature as existed in super  
     \* class, this is not method overriding instead this is called  
     \* method hiding in Java  
     \*/  
    public static void show(){  
        System.err.println("Overridden static method in Child Class in Java");  
    }  
}  
  
Output:  
Static method from parent class

This output confirms that **we can not override static method in Java** and static method are bonded based upon type information and not based upon Object. had Static mehtod be overridden, method from Child class or ColorScreen would have been called.

**Q. Can we Override Private Method in Java? Inner Class?**

**No**, we cannot override private methods in Java, private methods are non-virtual in Java and access differently than non-private one. Since method overriding can only be done on derived class and private methods are not accessible in a subclass, we just can not override them. one more possibility of overriding private methods in an inner class, since private methods are accessible in an inner class.This will also not work because private methods are bonded during compile time and only Type (or Class) is used to locate a private method.  
Had the private method overridden that it would have called method from child class. By the way, compiler will not complain, it will treat method with exact same signature in child class as separate method, and this is known as **method hiding in Java**.

**Q. Should We Make Private method Final in Java**

This is another common doubt among Java programmers, using final and private method in Java is a good choice, but I don't think it offer any benefit in terms of performance. Rather, this decision should be taken on the basis of design, functionality and ensuring readability of code.   
Since making a method final,  just limit it’s ability to be overridden, it doesn't make much sense to mark a private method as final in Java, because private method can not overridden in Java by any means. So compiler will definitely perform sort of optimization it can e.g. inlining or caching it. 

**Important Points about private keyword in Java**

1. We can apply private access modifier fields, methods and any inner class in Java. It’s most restricted access modifier and only accessible in the class they are declared. They are not visible outside the class and accessing them outside will result in compile time error.

2.Top level Classes can not be private in Java. They can only be either public or without any access modifier i.e. only accessible in the package they are declared. In case of public class, name of Java source file must match with the name of public class.

3.Though private methods or variables are not accessible outside of  the class, they are declare. They can be accessed via reflection by using setAccessible(true) and changing there private visibility.

4.As we have seen in this article, private methods can not be overridden in Java, not even inside nested or inner classes.

5.**Private members runs faster than non-private one, because of static binding**. They are also better candidate for optimization from compiler because they can not be overridden.

Private Method Overriding Example

In order to prove theory, we must see an example. In this Java program, we have declared a private method and trying to override them inside an Inner class. In main method, we are calling private method by using reference variable of Outer class, which is also parent but pointing to object of sub class.  Though it does show that private method is accessible in Inner class, we just can not override them. Let’s see what does output shows.

/\*\*

\* Java Program to demonstrate, private method can not be overridden in Java,

\* not even on Inner classes. Main reason of that behavior is because they are bonded

\* using static binding in Java.

\*/

public class PrivateMemberExample {

private String i\_m\_private = "I am private member, not accessible outside this Class";

private void privateMethod() {

System.out.println("Private method of Outer Class");

}

public static void main(String args[]) {

PrivateMemberExample outerClass = new PrivateMemberExample();

NestedClass nc = outerClass.new NestedClass();

nc.showPrivate(); //shows that private method are accessible in inner class.

outerClass = nc;

outerClass.privateMethod(); // This will not call private method from inner class,

// which shows we can not override

// private method inside inner class.

}

class NestedClass extends PrivateMemberExample {

public void showPrivate() {

System.out.println("Accessing Private members of Outer class: " + i\_m\_private);

privateMethod();

}

/\*

\* private method trying to be overridden,

\* instead it’s just hiding parent class method.

\*/

private void privateMethod() {

System.out.println("Private method of Nested Class");

}

}

}

Output

Accessing Private members of Outer class: I am private member, not accessible outside this Class

Private method of Outer Class

From output it’s clear that both the call to private method, which is made by using reference variable with type of parent class result in invoking private method from parent class, which is also outer class in our case. Had it was overridden, in case of second call, which is made using object of Inner sub class, would result in execution of private method of nested class. This proves that private method can not be overridden in Java, not in sub class and not even in Inner class.

That's all about question, Can we override private method in Java?. We have learned that this is not possible in Java. Though this is a real tough Java question, especially followup question which involves, Can we override private method on Inner class in Java. It can confuse we, if we are not sure about how private methods are bonded in Java. Let me know, if we think otherwise.

***Covariant Method Overriding of Java 5 - Coding Best Practices***

Sometimes knowledge of a specific Java feature can improve code quality, Covariant method overriding is one of such feature. Covariant method overriding was introduced in Java 5, but it seems it lost between other more powerful features of that release. Covariant method overriding helps to remove type casting on client-side**, by allowing we to return a subtype of actually return type of overridden method.**Covariant method overriding can be really useful while overriding methods that returns object e.g. clone() method. Since clone() return object every client needs to cast on to appropriate subclass, not anymore.  
By using Java 5 covariant overriding, we can directly return subtype instead of an object, as we will see in examples of this article. This feature is not a star feature like Generics or Enum, but it's definitely something worth knowing, given overriding methods are an integral part of Java programming.  
  
Covariant Method Overriding Example

As I said, one of the best examples of this is the overriding clone() method, which is declared in java.lang.Object class and has a return type of Object. If we have used java.util.Date in our project and has called its clone method to make a copy, we might know that we need to cast it back to Date as shown below :

Date d = new Date();

Date clone = (Date) d.clone();  
wouldn't it be great if the clone() method can return Date instead of Object? Well it can from Java 5 onwards, but unfortunately, code in java.util.Date is not yet updated to take advantage of this, at least till Java 6. Now let's create our own class and override clone() method to demonstrate the use of covariant method overriding in Java.

import java.util.Date;

/\*\*

\* Java program to demonstrate how to use covariant method overriding to avoid

\* casting at client side.

\* @author http://javarevisited.blogspot.com

\*/

public class CovariantMethodOverridingTest {

public static void main(String args[]) {

//without covariant overriding, cast at client side needed

Date d = new Date();

Date clone = (Date) d.clone(); //casting required

//with covariant method overriding, no client side cast

Duck duck = new Duck(0xFFFFFF);

Duck copy = duck.clone(); //no casting

}

}

class Duck implements Cloneable{

private int color = 0xFFFFFF;

public Duck(int color){

this.color = color;

}

public int getColor(){

return color;

}

@Override

protected Duck clone() {

Duck clone = null;

try{

clone = (Duck) super.clone();

}catch(CloneNotSupportedException ex){

throw new RuntimeException(ex);

}

return clone;

}

}

We can see in our test program, we don't need to typecast object returned from clone() method to Duck, because instead of java.lang.Object, its returning java.util.Date. This is also obvious from looking at the overriding clone() method in Duck class, we can see its return type is Duck, and not Object.  
In fact, if we have implemented our own Comparator or Comparable in Java, we can also see that their compare() and compareTo() directly return the type specified in Type parameter instead of object. Though this looks a very small and subtle change, it impacts is huge.  
I am sure they will implement clone() method of java.util.Date sooner or later to take advantage of this feature, unless they decide to deprecate it in favour of new Date and Time API coming in Java 8.  
  
  
That's all on covariant method overriding of Java 5 guys. Always take advantage of this feature, while the overriding method in Java 5, 6,7, and very soon on Java 8. We can return any subtype, based upon our need. All other rules of method overriding remain the same. I know Java is vast, and there are many more features like this, which can be really handy on a day to day job but many times remain hidden from a large number of Java developers.  
  
**Q. What is polymorphism in Java?**

A code written using polymorphism concept is much flexible to change and quite easy to maintain than the one which is written without polymorphism.   
In Java programming whole concept of interface is based on Polymorphism and it is a famous design principle that to code for interface than implementation to take advantage of Polymorphism and introducing new implementation in future.  
Polymorphism is an Oops concept which advice use of common interface instead of concrete implementation while writing code. When we program for interface our code is capable of handling any new requirement or enhancement arise in near future due to new implementation of our common interface.   
If we don't use common interface and rely on concrete implementation, we always need to change and duplicate most of our code to support new implementation.

How Polymorphism supported in Java

Java has excellent support of polymorphism in terms of Inheritance, method overloading and method overriding. Method overriding allows Java to invoke method based on a particular object at run-time instead of declared type while coding. To get hold of concept let's see an example of polymorphism in Java:

public class TradingSystem{

   public String getDescription(){

      return "electronic trading system";

   }

}

public class DirectMarketAccessSystem extends TradingSystem{

   public String getDescription(){

     return "direct market access system";

   }

}

public class CommodityTradingSystem extends TradingSystem{

   public String getDescription(){

     return "Futures trading system";

  }

}

Here we have a super class called TradingSystem and there two implementation DirectMarketAccessSystem and CommodityTradingSystem and here we will write code which is flexible enough to work with any future implementation of TradingSystem we can achieve this by using Polymorphism in Java which we will see in the further example.

Where to use Polymorphism in code

Probably this is the most important part of this Java Polymorphism tutorial and It’s good to know where we can use Polymorphism in Java while writing code. Its common practice to always replace concrete implementation with interface it’s not that easy and  comes with practice but here are some common places where I check for polymorphism:

1) Method argument:

Always use super type in method argument that will give we leverage to pass any implementation while invoking a method. For example:

public void showDescription(TradingSystem tradingSystem){

   tradingSystem.description();

}

If we have used concrete implementation e.g. CommodityTradingSystem or DMATradingSystem then that code will require frequent changes whenever we add new Trading system.

2) Variable names:

Always use Super type while we are storing reference returned from any Factory method in Java, This gives we the flexibility to accommodate any new implementation from Factory. Here is an example of polymorphism while writing Java code which we can use retrieving reference from Factory:

String systemName = Configuration.getSystemName();

TradingSystem system = TradingSystemFactory.getSystem(systemName);  
3) The return type of method

The return type of any method is another place where we should be using interface to take advantage of Polymorphism in Java. In fact, this is a requirement of Factory design pattern in Java to use interface as a return type for factory method.

public TradingSystem getSystem(String name){

   //code to return appropriate implementation

}  
Method overloading and method overriding use concept of Polymorphism in Java where method name remains the same in two classes but actual method called by JVM depends upon an object at run time and done by dynamic binding in Java.   
Java supports both overloading and overriding of methods. In case of overloading method signature changes while in case of overriding method signature remains same and binding and invocation of the method is decided on runtime based on actual object.   
This facility allows Java programmers to write very flexibly and maintainable code using interfaces without worrying about concrete implementation. One disadvantage of using Polymorphism in code is that while reading code we don't know the actual type which annoys while we are looking to find bugs or trying to debug the program.  
  
Parametric Polymorphism in Java

Java started to support parametric polymorphism with the introduction of Generic in JDK1.5. Collection classes in JDK 1.5 are written using Generic Type which allows Collections to hold any type of an object at run time without any change in code and this has been achieved by passing actual Type as a parameter.   
  
For example, see the below code of a parametric cache written using Generic which shows the use of parametric polymorphism in Java. Read how to create Generic class and methods in Java for more details.

interface cache{

  public void put(K key, V value);

  public V get(K key);

}

That’s all on polymorphism in java, for now, please suggest and share some of the other coding practices that involve the use of polymorphic behavior of java for the benefit of all. Thank we.

**Q. Difference between Method Overloading and Overriding in Java?**

1) First and most important difference between method overloading and overriding is that, In case of method overloading in Java, signature of method changes while in case of method overriding it remain same.  
2) Second major difference between method overloading vs overriding in Java is that We can overload method in one class but overriding can only be done on subclass.  
3) We can not override static, final and private method in Java but we can overload static, final or private method in Java.  
4) **Overloaded** method in Java is bonded by static binding and **overridden** methods are subject to dynamic binding.  
5) Private and final method can also be not overridden in Java.  
Things to Remember

1) In case of method overloading method signature gets changed while in case of overriding signature remains same.  
2) Return type is not part of method signature in Java.  
3) Overloaded method can be subject to compile time binding but overridden method can only be bind at run-time.  
4) Both overloaded and overridden method has same name in Java.  
5) Static method can not be overridden in Java.  
6) Since private method is also not visible outside of class, it can not be overridden and method binding happens during compile time.  
7) From Java 5 onwards we can use annotation in Java to declare overridden method just like we did with @override. @override annotation allows compiler, IDE like NetBeans and Eclipse to cross verify or check if this method is really overrides super class method or not.  
**Q. Can we override a method that throws runtime exception without throws clause? (answer)  
Ans. Yes**, there is no restriction on unchecked exceptions while overriding. On the other hand, in the case of checked exception, an overriding exception cannot throw a checked exception which comes higher in type hierarchy e.g. if the original method is throwing IOException than the overriding method cannot throw java.lang.Exception or java.lang.Throwable.  
  
**Q. How do we call a superclass version of an overriding method in a subclass? (answer)  
Ans**. We can call a superclass version of an overriding method in the subclass by using super keyword. For example to call the toString() method from java.lang.Object class, we can call super.toString().  
  
**Q. Can we override a non-static method as static in Java? (answer)  
Ans. Yes**, we can override the non-static method in Java, no problem on them but it should not be private or final :)  
**Q. Can we have a non-abstract method inside an interface? (answer)  
Ans.** From Java 8 onward we can have a non-abstract method inside interface, prior to that it was not allowed as all method was implicitly public abstract. From JDK 8, we can add static and default methods inside an interface.

**Q. When to make a method final in Java**

**Ans**. This gives we the first hint when to make a method final in Java, obviously to prevent subclass for changing its definition, technical **preventing it from overriding.** There are lot of clues on How to use final keyword on methods in Java API itself, one of the prime example of this java.lang.Object class, which declares a lot of method final including wait method with timeout.  
Another example of making a method final is template method, which outlines an algorithm in the Template method design pattern. Since, we don't want a subclass to change the outline of the algorithm, making it final will prevent any accidental or malicious overriding. In this tutorial, we will learn a few things, which will help we to effectively use final keywords with Java methods.

Final method in Java

1) Methods which are called from constructor should be final, otherwise a subclass may override and alter the behavior of method, which may result in surprise and undesirable behavior.  
2) Consider making performance critical method final, this given compiler more opportunity to optimize it. Compiler may inline or cache such methods because they can not be changed during application life time. Though modern JIT are even capable of in-lining non final method, if they think it can result in performance gain and it's not overridden, making a method final can do this in-lining even before JIT comes into picture.  
3) Sensitive methods, whose behaviour’s are deemed complete and which are not supposed to be overridden, should be final.  
4) Consider making template methods final, while using template method design pattern.  
In short, making a method final is mainly driven by design and safety requirement. Though performance can also be consider one of the reason, it is not the main reason in today's Java world of advanced JIT. It's also a good idea to document, why we made a method final, that will help developer, who will maintain our code. There is also another doubt about making a private method final or not. Since private method can not be overridden, neither in subclass, where it's not accessible, nor in inner class where it's accessible, because they are bonded using static binding using class name. Making a private method final doesn't provide any additional value.

**Q. Difference between final, finally and finalize method in Java**

**Final** keyword can be used along with variable, method and Class in Java. If we make a variable final, we can not change its value, it will act like a constant. final variables are initialized at the time of creation except in the case of a **blank final variable which is initialized in Constructor**. If we make a method final in Java, we can not override it in subclass .   
If we make a class final means it can not be sub classed. Making a class final automatically makes all its method final and this is sometimes required due to security reason, This is one of the reasons Why String is final in Java.   
In short final is not related at all with either finally or finalize keyword. final keyword also helps to write Immutable classes which are critical for designing thread-safe multi-threading system and reducing the amount of synchronization.

**finally** is used for exception handling along with try and catch. As per Java programming language’s rule, for exception handling, we at least need either catch or finally block. finally, the block has a special advantage over catch that it's guaranteed to be executed despite whether Exception is thrown or not.  
This makes it, an ideal place to close system resources like InputStream or OutputStream, which is required to release scarce file descriptor. Closing streams, network connection, database connection in a finally block is good coding practice in Java.   
  
From Java 7, we can use try with resource block to close resource automatically. Since finally is guaranteed to be executed in most cases, it also gives birth to some tricky Java questions where finally doesn't execute like returning value from a finally block, calling System.exit from try block etc. finally block always execute, except in the case of JVM dies i.e. calling System.exit() . Again finally is not related to final or finalize in any way.

**Finalize**() method, finalize() is called by Garbage collection thread just before collecting eligible Objects. This is the last chance for an object to perform any cleanup but since it's not guaranteed that whether finalize() will be called, its bad practice to keep resources till finalize call. Though we can build a safety net on finalize by double-checking scarce resources.

So, final, finally and finalize all are a different keyword, they are used for different purpose. the only similarity between them is that they are a Java programming language keyword, other than that final, finalize and finally are completely different than each other.

**Q. ) What is a compile time constant in Java? What is the risk of using it?**public static final variables are also known as a compile time constant, the public is optional there. They are replaced with actual values at compile time because compiler know their value up-front and also knows that it cannot be changed during run-time. One of the problem with this is that if we happened to use a public static final variable from some in-house or third party library and their value changed later than our client will still be using old value even after we deploy a new version of JARs. To avoid that, make sure we compile our program when we upgrade dependency JAR files.  
  
**Q. Difference between equals method and "==" operator in Java**

**Ans**: - Both equals() and "==" operator in Java is used to compare objects to check equality but the main difference between equals method and  the == operator is that equals method is a method and later is an operator. Since Java doesn’t support operator overloading, == behaves identical for every object but equals() is method, which can be overridden in Java and logic to compare objects can be changed based upon business rules. Another notable difference between == and equals method is that former is used to compare both primitive and objects while later is only used for objects comparison.  
At the same time, beginners struggle to find when to use equality operator (==) and when to use equals method for comparing Java objects. 

* **"==" equality operator in Java**

"==" or equality operator in Java is a binary operator provided by Java programming language and used **to compare primitives and objects.** In terms of comparing primitives like boolean, int, float "==" works fine but when it comes to comparing objects it creates confusion with equals method in Java.  
The equality operator or "==" **compare two objects based on memory reference**. so **"==" operator will return true only if two object reference it is comparing represent exactly same object otherwise "==" will return false.**   
  
 **equals method in Java?**

**Equals() method** is defined in Object class in Java and used for checking equality of two objects defined by business logic e.g. two Employees are considered equal if they have same empId etc.   
equal has contracted with hashcode method in Java and whenever we override equals method we also need to override hashcode() in Java.   
**Default implementation of equals provided in Object class is similar to "==" equality operator** and return true if we are comparing two references to the same object.  
It’s one of the Java best practice to override equals in Java to define equality based on business requirement. It’s also worth noting that equals should be consistent with compareTo in Java, So that when we store objects in TreeMap or TreeSet Collection, which uses compareTo for checking equality, behavior remains consistent.

**Difference between == and equals in Java**

Main difference between == and equals in Java is that "==" is used to compare **primitives** while equals() method is recommended to check equality of **objects**.   
Another difference between them is that, If both "==" and equals() is used to compare objects than == returns true only if both references points to same object while **equals() can return true or false based on its overridden implementation**.One of the popular cases is comparing two String in Java in which case == and equals() method return different results.

1. Comparing String with == and equals

String comparison is a common scenario of using both == and equals method. Since java.lang.String class override equals method, It return true if two String object contains same content but == will only return true if two references are pointing to the same object.   
Here is an example of comparing two Strings in Java for equality using == and equals() method which will clear some doubts:

String personalLoan = new String("cheap personal loans");  
String homeLoan = new String("cheap personal loans");  
       
//since two strings are different object result should be false  
boolean result = personalLoan == homeLoan;  
System.out.println("Comparing two strings with == operator: " + result);  
       
//since strings contains same content , equals() should return true  
result = personalLoan.equals(homeLoan);  
System.out.println("Comparing two Strings with same content using equals method: " + result);  
       
homeLoan = personalLoan;  
//since both homeLoan and personalLoand reference variable are pointing to same object  
//"==" should return true  
result = (personalLoan == homeLoan);  
System.out.println("Comparing two reference pointing to same String with == operator: " + result);  
  
Output:  
Comparing two strings with == operator: false  
Comparing two Strings with same content using equals method: true  
Comparing two references pointing to same String with == operator: true

2. Comparing two objects with "==" and equals.

Another scenario which creates confusion between == and equals method is when we compare two Objects. When we compare two references pointing to an object of type Object we should see the same result from both == operator and equals method because default implementation of equals method just compare memory address of two objects and return true if two reference variable are pointing towards an exactly same object.   
  
Here is example of == vs equals method for comparing two objects:

Object obj1 = new Object();  
Object obj2 = new Object();     
// == should return false  
result = (obj1==obj2);  
System.out.println("Comparing two different Objects with == operator: " + result);  
       
//equals should return false because obj1 and obj2 are different  
result = obj1.equals(obj2);  
System.out.println("Comparing two different Objects with equals() method: " + result);  
       
// "==" should return true because both obj1 and obj2 points same object  
obj1=obj2;  
result = (obj1==obj2);  
System.out.println("Comparing two reference pointing to same Object with == operator: " + result);  
  
Output:  
Comparing two different Objects with == operator: false  
Comparing two different Objects with equals() method: false  
Comparing two references pointing to the same Object with == operator: true

Summary

1) use == to compare primitive e.g. boolean, int, char etc, while use equals() to compare objects in Java.

2) == return true if two reference are of same object. Result of equals() method depends on overridden implementation.

3) For comparing String use equals() instead of  == equality operator.

That’s all on the difference between equals method and  == operator in Java.  As I said the main difference between them is that one of them is an operator and other is a method and == is used to compare both primitive and objects while equals() method is used to check equality of objects only.

**Q. How to override hashcode in Java example ?**

Equals and hashcode methods are two primaries but yet one of the most important methods for java developers to be aware of. Java intends to provide equals and hashcode for every class to test equality and to provide a hash or digest based on the content of the class. The importance of hashcode increases when we use the object in different collection classes which works on **hashing principle** e.g. hashtable and hashmap. A well-written hashcode method can **improve performance drastically** by distributing objects uniformly and avoiding a collision.  
In this article, we will see how to correctly override the hashcode() method in java with a simple example.  
We will also examine the important aspects of hashcode contracts in java. This is in continuation of my earlier post on overriding equals method in Java, if we haven’t read already I would suggest going through it.

**General Contracts for hashCode() in Java**

1) If two objects are equal by equals() method then there hashcode returned by hashCode() method must be the same.

2) Whenever the hashCode() method is invoked on the same object more than once within a single execution of the application, hashCode() must return the same integer provided no information or fields used in equals and hashcode is modified. This integer is not required to be the same during multiple executions of application though.

3) If two objects are not equaled by equals() method **it is not require** that there hashcode must be different. Though it’s always good practice to return different hashCode for unequal object. Different hashCode for distinct object can improve performance of hashmap or hashtable by reducing collision.

To better understand concept of equals and hashcode and what happens if we don’t override them properly I would recommend understanding of How HashMap works in Java

Overriding hashCode method in Java

We will follow step by step approach for overriding hashCode method. This will enable us to understand the concept and process better.

1) Take a prime hash e.g. 5, 7, 17 or 31 (prime number as hash, results in distinct hashcode for distinct object)

2) Take another prime as multiplier different than hash is good.

3) Compute hashcode for each member and add them into final hash. Repeat this for all members which participated in equals.

4) Return hash

  Here is an example of hashCode() method

   @Override

    public int hashCode() {

        int hash = 5;

        hash = 89  hash + (this.name != null ? this.name.hashCode() : 0);

        hash = 89  hash + (int) (this.id ^ (this.id >>> 32));

        hash = 89  hash + this.age;

        return hash;

    }

It’s always good to check null before calling hashCode() method on members or fields to avoid NullPointerException, if member is null than return zero. Different data types has different way to compute hashCode.Integer members are simplest we just add there value into hash, for other numeric data-type are converted into int and then added into hash. Joshua bloach has full tables on this. I mostly relied on IDE for this.

**Better way to override equals and hashCode**

In my opinion, **null check**, instanceof check, etc and also frees we to remember how to compute hashcode for different data-types.

Let’s see how we can override the hashcode method in Netbeans and Eclipse.

Things to remember while overriding hashcode in Java

1. Whenever we override equals method, hashcode should be overridden to be in compliance with equals hashcode contract.

2. hashCode() is declared in Object class and return type of **hashcode method is int and not long.**

3. For an immutable object, we can cache the hashcode once generated for improved performance.

4. Test our hashcode method for equals hashcode compliance.

5. **If we don't override hashCode() method properly our Object may not function correctly on hash-based collection e.g. HashMap, Hashtable or HashSet.**

A complete example of equals and hashCode

public class Stock {

       private String symbol;

       private String exchange;

       private long lotSize;

       private int tickSize;

       private boolean isRestricted;

       private Date settlementDate;

       private BigDecimal price;

          @Override

       public int hashCode() {

              final int prime = 31;

              int result = 1;

              result = prime \* result

                           + ((exchange == null) ? 0 : exchange.hashCode());

              result = prime \* result + (isRestricted ? 1231 : 1237);

              result = prime \* result + (int) (lotSize ^ (lotSize >>> 32));

              result = prime \* result + ((price == null) ? 0 : price.hashCode());

              result = prime \* result

                           + ((settlementDate == null) ? 0 : settlementDate.hashCode());

              result = prime \* result + ((symbol == null) ? 0 : symbol.hashCode());

              result = prime \* result + tickSize;

              return result;

       }

       @Override

       public boolean equals(Object obj) {

              if (this == obj) return true;

              if (obj == null || this.getClass() != obj.getClass()){

                     return false;

              }

              Stock other = (Stock) obj;

return

this.tickSize == other.tickSize && this.lotSize == other.lotSize &&

this.isRestricted == other.isRestricted &&

(this.symbol == other.symbol|| (this.symbol != null && this.symbol.equals(other.symbol)))&&

(this.exchange == other.exchange|| (this.exchange != null && this.exchange.equals(other.exchange))) &&

(this.settlementDate == other.settlementDate|| (this.settlementDate != null && this.settlementDate.equals(other.settlementDate))) &&

(this.price == other.price|| (this.price != null && this.price.equals(other.price)));

      }}

Writing equals and hashcode using Apache Commons EqualsBuilder and HashCodeBuilder

EqualsBuilder and HashCodeBuilder from Apache commons are a much better way to override equals and hashcode method, at least much better than ugly equals, hashcode generated by Eclipse. I have written the same example by using HashCodebuilder and EqualsBuilder and now we can see how clear and concise they are.

    @Override

    public boolean equals(Object obj){

        if (obj instanceof Stock) {

            Stock other = (Stock) obj;

            EqualsBuilder builder = new EqualsBuilder();

            builder.append(this.symbol, other.symbol);

            builder.append(this.exchange, other.exchange);

            builder.append(this.lotSize, other.lotSize);

            builder.append(this.tickSize, other.tickSize);

            builder.append(this.isRestricted, other.isRestricted);

            builder.append(this.settlementDate, other.settlementDate);

            builder.append(this.price, other.price);

            return builder.isEquals();

        }

        return false;

    }

    @Override

    public int hashCode(){

        HashCodeBuilder builder = new HashCodeBuilder();

        builder.append(symbol);

        builder.append(exchange);

        builder.append(lotSize);

        builder.append(tickSize);

        builder.append(isRestricted);

        builder.append(settlementDate);

        builder.append(price);

        return builder.toHashCode();

    }

    public static void main(String args[]){

        Stock sony = new Stock("6758.T", "Tkyo Stock Exchange", 1000, 10, false, new Date(), BigDecimal.valueOf(2200));

        Stock sony2 = new Stock("6758.T", "Tokyo Stock Exchange", 1000, 10, false, new Date(), BigDecimal.valueOf(2200));

        System.out.println("Equals result: " + sony.equals(sony2));

        System.out.println("HashCode result: " + (sony.hashCode()== sony2.hashCode()));

    }

The only thing to concern is that it adds a dependency on apache-commons jar, most people use it but if we are not using than we need to include it for writing equals and hashcode method.

must override all abstract method. For example if a class ABC has an abstract method abc() then a class EFG, which extends ABC must override abc() to be a concrete class i.e. whose instance can be created. On related note, always use @Override annotation while overriding a method in Java. It not only help our fellow programmers to see our intent but also helps our friend compiler to find subtle mistakes which can easily ship due to blind faith on our other friend IDE's content assist. Believe me its too easy to import a class with same name but from different package with code auto-completion feature of modern IDEs.  
  
3) A class can be abstract even without any abstract method. Though, its not mandatory (not ensured by any compiler rule) to have abstract method inside abstract class, but this is not advisable, because there is no point doing it. If our class don't have any abstract behaviour (behaviour which changes between different types) then its good candidate of being a concrete class rather than abstract. There is only one case where I think abstract class without abstract method can be used is when we need a marker class instead of marker interface, but to be frank, I have yet to find a practical use case. Though thinking a class as abstract at start can be good for discovering some level of abstraction.

4) Both top level and nested class can be make abstract in Java, no restriction on this from Java programming language.  
That's all about Abstract class in Java. Always remember that anything abstract (e.g. abstract class or abstract method) is not complete in Java, and we must extend abstract class and override abstract method to make use of that class. Remember, Abstraction is very important to design flexible systems. Flexibility comes from using interfaces and abstract class at top level and leverage inheritance and Polymorphism to supply different implementation without changing the code which uses them.

**Q. What is the Actual Use of interface in Java?**

**Ans.** **Java 8 to allow lambda expressions**, now **interface can have one non-abstract method,** **also known as a default method**). So, if we can't define anything, why we need an interface?

**Interface also allows multiple inheritance in Java,** which makes it possible for a class to become Canvas, as well as EventListener, which is used to draw graphics as well as to to process events.

**public final class String**

**implements** **java.io.Serializable, Comparable<String>, CharSequence**  
**Why we need Interface in Java**

There are several reasons, an application developer needs an interface, one of them is Java's feature to provide **multiple inheritance at interface level**. It allows we to write flexible code, which can adapt to handle future requirements. Some of the concrete reasons, why we need interface is :  
1) If we only implement methods in subclasses, the callers will not be able to call them via the interface (**not common point where they are defined**).  
2) Java 8 will introduce default implementation of methods inside the interface, but that should be used as exception rather than rule. Even Java designer used in that way, it was introduced to **maintain backward compatibility along with supporting lambda expression**. All evolution of Stream API was possible due to this change.  
3) Interfaces are a way to declare a contract for implementing classes to fulfil; it's the primary tool to create abstraction and decoupled designs between consumers and producers.  
4) **Because of multiple inheritance, interface** allows we to treat one thing differently. For example a class can be treated as Canvas during drawing and EventListener during event processing. **Without interface, it's not possible for a class to behave like two different entity at two different situation**s. Here is an example of how interface supports multiple inheritance in Java

**interface** **Canvas**{

public void paint(Graphics g);

}

**interface EventListener**{

public boolean process(Event e);

}

pubic class Game **implements Canvas, EventListener**{

@Override

public void paint(Graphics g){

g.drawLine(Color.RED);

}

@Override

public boolean process(Event e){

KeyCode code = e.getKeyPressed().getCode();

}

}

5) **Interface are key of API design. In fact smaller interface like Comparable, Runnable, Callable makes core of Java API.** Though great care is required while designing and publishing interface, because once published, we can not change the interface without breaking up all our clients, i.e. classes which have implemented our interface. In an extreme case, from Java 8 onwards, we can use default method to rescue, but as I said, it should be the exception than the rule.  
6)  "Programming to interface than implementation" is one of the popular Object-oriented design principle, and the use of interface promotes this. A code written on interface is much more flexible than the one which is written on implementation.  
7) **The use of interface allows we to supply a new implementation,** **which could be more robust, more performance in the later stages of our development**.  
In short main use of the interface is to facilitate polymorphism. the interface allows a class to behave like multiple types, which is not possible without multiple inheritances of class. It also ensures that we follow programming to the interface than the implementation pattern, which eventually adds a lot of flexibility in our system.

**Ques. Can an interface extend more than one interface in Java?  
Ans. Yes**, an interface can extend more than one interface in Java, it's perfectly valid.  
  
**Ques. Can a class extend more than one class in Java?  
Ans. No**, a class can only extend another class because Java doesn't support multiple inheritances but yes, it can implement multiple interfaces.  
  
**What is difference between interface and abstract class in Java**

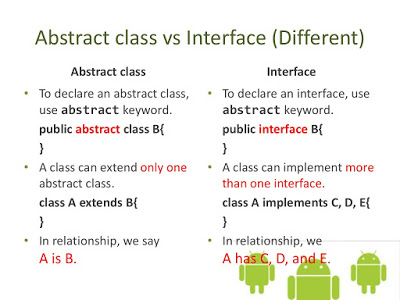
**Ans**. After Java 8 introduced default methods and allowed **interfaces to have both default and static methods.**

e.g. when to use interface in Java and when to use abstract class is Java.

**Abstract class vs Interface in Java**

In the last section, we saw what is abstract class and interface and now let's see the difference between interface and abstract class in Java.  
1) First and the major difference between abstract class and an interface is that an **abstract class is a class while the interface is an interface**, means by extending the abstract class we can not extend another class because Java does not support multiple inheritances but we can implement multiple inheritance in Java.  
2) The second difference between interface and abstract class in **Java is that we can not create a non-abstract method in an interface,** every method in an interface is by default abstract, but we **can create a non-abstract method in abstract class**. Even a class which doesn't contain any abstract method can be made abstract by using the abstract keyword.  
3) The third difference between abstract class vs interface in Java is that interface is better suited for Type declaration and abstract class is more suited for code reuse and evolution perspective.

4) The fourth difference between abstract class and interface in Java is that **abstract class are slightly faster than interface** because interface involves a search before calling any overridden method in Java. This is not a significant difference in most of the cases but if we are writing a time critical application then we may not want to leave any stone unturned.  
5) Another notable difference between interface and abstract class is that when we add a new method in existing interface it breaks all its implementation and we need to provide an implementation in all clients which is not good. By using an abstract class we can provide a default implementation for a new method in the superclass without breaking existing clients.  
  
Here is a nice summary slide of highlighting key differences between an abstract class and interface in Java:



**1) Can abstract class have constructors in Java?**

**Ans. Yes**, an abstract class can declare and define a constructor in Java. Since we cannot create an instance of an abstract class, a constructor can only be called **during constructor** **chaining**, i.e. when we create an instance of the concrete implementation class.   
  
**Q. If we can not instantiate abstract class?**

**Ans.** Well, **it can still be used to initialize common variables, which are declared inside an abstract class, and used by the various implementation.**   
  
Also even if we don’t provide any constructor, the compiler will add default no-argument constructor in an abstract class, without that our subclass will not compile, **since the first statement in any constructor implicitly calls super(),** default superclass constructor in Java.

**2) Can abstract class implements interface in Java? do they require to implement all methods?**

**Ans**. **Yes**, **an abstract class can implement an interface by using the implements keyword.** Since they are abstract, they don’t need to implement all methods. It’s good practice to provide an abstract base class, along with an interface to declare Type. One example of this is **java.util.List** interface and corresponding **java.util.AbstractList** abstract class.   
Since AbstractList implements all common methods,  concrete implementations like LinkedList and ArrayList are free from the burden of implementing all methods, had they implemented List interface directly.   
It’s best of both worlds, we can get the advantage of interface for declaring type, and flexibility of abstract class to implement common behavior in one place. Effective Java has a nice chapter on how to use interface and abstract class in Java, which is worth reading.

**Q.) Can an abstract class be final in Java?**

**Ans. No**, an abstract class can not be final in Java. Making them **final will stop the abstract class from being extended,** which is the only way to use an abstract class. They are also opposite of each other, abstract keyword enforces to extend a class, for using it, on the other hand, **final keyword prevents a class from being extended.** In real world also, abstract signifies incompleteness, while final is used to demonstrate completeness. Bottom line is, we can not make our class abstract and final in Java, at same time, it’s a compile time error.

**Q.) Can abstract class have static methods in Java?**

**Ans. Yes**, an abstract class can declare and define static methods, nothing prevents from doing that. But, we must follow guidelines for making a method static in Java, as it’s not welcomed in a object oriented design, because static methods can not be overridden in Java. It’s very rare, we see static methods inside abstract class, but as I said, if we have very good reason of doing it, then nothing stops we.

**Q) Can we create an instance of abstract class?**

**Ans**. **No**, we can not create instance of abstract class in Java, they are incomplete. Even though, if our abstract class don’t contain any abstract method, we can not create instance of it. By making a class abstract,  we told compiler that, it’s incomplete and should not be instantiated. Java compiler will throw error, when a code tries to instantiate abstract class.

**Q) Is it necessary for an abstract class to have an abstract method?**

**Ans. No**, It’s not mandatory for an abstract class to have any abstract method. We can make a class abstract in Java, by just using abstract keyword in class declaration. Compiler will enforce all structural restriction, applied to abstract class, e.g. now allowing to create any instance. By the way, it’s debatable whether we should have abstract method inside abstract class or interface. In my opinion, abstract class should have abstract methods, because that’s the first thing programmer assumes, when he see that class. That would also go nicely along principle of least surprise.

**Q) When do we favour abstract class over interface?**

**Ans**. **Since it’s almost impossible to add a new method on a published interface,** it’s **better to use abstract class,** when evolution is concern. **Abstract class in Java evolves better than interface**. Similarly, **if we have too many methods inside interface, we are creating pain for all it’s implementation,** consider providing an abstract class for default implementation. This is the pattern followed in Java collection package, **we can see AbstractList provides default implementation for List interface.**

**Q) Can abstract class contains main method in Java ?**

**Ans. Yes**, abstract class can contain main method, it just another static method and we can execute Abstract class with main method, until we don’t create any instance.

**Q. Can we Overload or Override main method in Java?**

**Ans**. The short answer to, can we overload the main method in Java is **Yes**, we can overloading, nothing stops from overloading, but JVM will always call the original main method, it will never call our overloaded main method.

Code Example of Overloading Main Method in Java

/\*\*

\* Java Program to show that we can overload main method in Java

\* but we cannot override main method.

\*

\* @author Javin Paul

\*/

public class Helloworld {

/\*\*

\* Standard main method, JVM will only call this method

\* even if we provided multiple overloaded version.

\*

\*/

public static void main(String[] args) {

System.out.println("Inside main(String[] args) method ....");

}

/\*\*

\* An overloaded main method which accepts Integer[] instead of

\* String[] as argument.

\* @param args

\*/

public static void main(Integer[] args){

System.out.println("Inside main(Integer[] args) method ....");

}

/\*\*

\* Another overloaded main method which accepts Double[] instead of

\* String[] as argument.

\* @param args

\*/

public static void main(Double[] args){

System.out.println("Inside main(Double[] args) method ....");

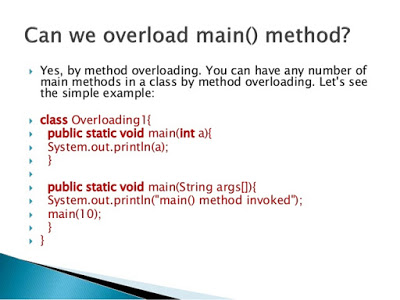
}

}

Output

Inside main(String[] args) method ....

In this example, we can see that we have two main methods, one accepts String array as an argument and the other accept Integer array as an argument, but we can see that when we run our program from the command line, only the main method with string array as an argument is called.  
  
There was no error, no ambiguity, JVM will always call this main method, no matter how many overloaded main method we will put on this class. Then questions come how do we call our overloaded main? Well, we can call it just like any other method.  
  
Simply calling main(new int[]{1, 2, 3}) from the original main method will invoke our overloaded main with integer array as an argument. Since there is no compilation error, it proves that we can overload the main method in Java.  
  
Regarding Overriding we have already proven that the static method cannot be overridden, they can only be hidden. See that post to learn this concept by following an example. In short, the main method can be overloaded but cannot be overridden in Java.

[](https://www.java67.com/2014/02/can-you-run-java-program-without-main-method.html)

That's all about overloading and overriding the main method in Java. Now we know that it's possible to overload main in Java but it's not possible to override it, simply because it's a static method. Execution of Java program has no impact on overloading main because JVM always calls the original main method and if it doesn't found in class then it throws java.lang.NoSuchMethodError: main Exception in thread "main" error at runtime.

# **Ques. Double Checked Locking in Java and Why it was broken before JDK 5?**

**Ans.** **One of the simplest ways to write thread-safe Singleton was to make the getInstance() method synchronized** **but prior to JDK 1.6, a simple uncontented synchronization block was expensive**, and that lead many developers to write the getInstance() method of Singleton class using double-checked locking idiom. This was one of the clever idioms of that time which only uses synchronization when the Singleton object is created as seen in the following code and thus improves the performance of the getInstance() method, which is used to retrieve the Singleton object.  
  
Here is one example of a thread-safe Singleton class in Java using double-checked locking pattern:

class DCLSingleton {

private static **volatile DCLSingleton \_instance = null;**

**private DCLSingleton() {**

**}**

public static DCLSingleton instance() {

if (\_instance == null) { // 1st check

synchronized (DCLSingleton.class) {

if (\_instance == null) // 2nd check

{

\_instance = new DCLSingleton();

}

}

}

return \_instance;

}

}

There are a couple of important points to note about this code:  
  
1) A static **volatile field is used to hold the instance of the Singelton class**. The volatile variable is key here, without volatile, it won't be thread-safe.  
2) The **constructor is made private** to disable instance creation outside of this class, but we can create an instance inside the class and that's what your getInstance() method does.  
  
3) **There are two checks** to see if the \_instance member variable is initialized or not and that's why this code **idiom it's called double-checked locking idiom**.  
  
4) **The first check is non-synchronized,** which may see a partially constructed object because of instruction re-ordering by the compiler or JVM.  
  
5) The second check is inside the synchronized block and only **executes one time during lifespan of Singleton.** That's why we get the performance boost becuase locking only happens one time during the lifespan of Singleton instance.

**Q. Why Double-checked locking is broken Prior to 5**

**Ans**. **The sole purpose of Double-checked locking was to avoid excessive synchronization** and **hence it relies on non-synchronized access of \_instance field at the 1st check point.** This appears harmless, but it is not.  
  
Suppose one thread, Thread-1 is inside the synchronized block and it's creating Singleton instance and assigning a reference to \_instance variable. In the meantime, the Thread scheduler stops the Thread-1. Now, a second thread, Thread-2 enters and come to 1st check point which is not synchronized, now there is a possibility that it can see half-initialized \_instnace field and return that to the client, leading to subtle bugs in your program.  
  
This issue was fixed by introducing the happens-before guarantee provided by **the volatile variable in Java 1.5**. According to this rule, write to a volatile field will happen before any read, **which negates the possibility of seeing half initialized instance of Singleton class**.

Safe alternatives of Double-checked Locking Pattern

If getInstance() method is not bottleneck than the simplest way to avoid Double-checked locking is make the whole method synchronized. Java has come a long way in performance and penalty to synchronizing is a lot lesser than what it used to be.  
  
**Other alternative includes using Enum as Singleton in Java**. Enum guarantees lot of features required by Singleton pattern out-of-the-box e.g. initialization to Enum constants are thread-safe, Enum also provides Serialization guarantee that only one instance of Singleton will exits and it's very easy to code as seen below:

public enum ThreadSafeSingleton{

INSTANCE;

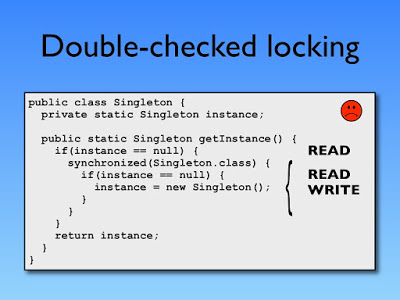
}

Another safe way to create thread-safe Singleton is **using eager initialization which initializes Singleton instance at the time Singleton class is loaded into memory** as opposed to when client class the getInstance() method. If your Singleton is not very heavy than this idiom works quite well.

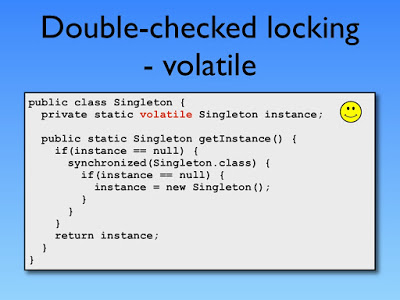
class EagerSingleton {

public static EagerSingleton singleton = new EagerSingleton ();

}  
  
**One more alternative of double checked locking is Initialization on Demand Holder idiom**, which **uses an inner class to encapsulate Singleton instance**. This idiom takes advantage of the fact that Inner class is not loaded until they are referenced.  
  
We can still use double checked locking pattern to create thread-safe Singleton classes in Java, but when we use double checked locking, don't forget to include the volatile modifier on Singleton instance.  
  
Here is a bad and good example of double checked locking with Singleton in Java, following code misses the volatile modifier, hence Singleton is not thread-safe:



W**e need to use the volatile variable to make the above code thread-safe in Java 1.5 as shown below:**



**Q. Difference between Serializable vs Externalizable in Java - Interview Question**

**Ans.**

**Java provides default mechanism to preserver object state** and our job many times would be as simple as adding implements Serializable on any class.

**Difference between Serializable and Externalizable in Java**

1) One of the obvious difference between Serializable and Externalizable is that **Serializable is a marker interface** i.e. does not contain any method but **Externalizable interface contains two methods writeExternal() and readExternal().**

2) The second difference between Serializable vs Externalizable is responsibility of Serialization. when a class implements Serializable interface, default Serialization process gets kicked of and that **takes responsibility of serializing super class state.** When any class in Java implement java.io.**Externalizable than its our responsibility to implement Serialization process** i.e. preserving all important information.

3) This difference between Serializable and Externalizable is performance. We **cannot do much to improve performance of default serialization process except reducing number of fields to be serialized by using transient and static keyword** but with **Externalizable interface we have full control over Serialization process.**

4) Another important difference between Serializable and Externalizable interface is maintenance. When your Java class implements **Serializable** **interface** its tied with default representation which is fragile and **easily breakable if structure of class changes** e.g. adding or removing field. By using java.io.**Externalizable interface we can create your own custom binary format for your object.**

# **Q. What is transient variable in Java - Serialization Example**

**Ans**. **transient variable in Java is a variable whose value is not serialized during Serialization** **and which is initialized by its default value during de-serialization**, for example for object transient variable it would be null. this behaviour can be customized by using Custom Serialized form or by using Externalizable interface. transient variable is used to prevent any object from being serialized and we can make any variable transient by using transient keyword.

## How to use transient variable in Java - Serialization Example

Here is a complete code example of  Serialization in Java which demonstrate How to use transient variable in Java program; transient variables are not serialized during Serialization process and initialize with default value during deserialization.

**package** test;  
  
**import** java.io.FileInputStream;  
**import** java.io.FileOutputStream;  
**import** java.io.ObjectInputStream;  
**import** java.io.ObjectOutputStream;  
**import** java.io.Serializable;  
  
/\*\*  
 \*  
 \* Java program to demonstrate **What is transient variable in Java** and fact that value of  
 \* transient variable is not serialized and during serialization it initialized with  
 \* default value of that data type. e.g. If transient variable is Object than after  
 \* deserialization its value would be null.  
 \*  
 \* @author Javin  
 \*/  
**public** **class** TransientTest {  
  
   
    **public** **static** **void** main(**String** args[]) {  
   
       **Book** narnia = **new** **Book**(1024, "Narnia", "unknown", 2);  
       **System**.out.println("Before Serialization: " + narnia);  
       
        **try** {  
            **FileOutputStream** fos = **new** **FileOutputStream**("narnia.ser");  
            **ObjectOutputStream** oos = **new** **ObjectOutputStream**(fos);  
            oos.writeObject(narnia);  
  
            **System**.out.println("Book is successfully Serialized ");  
  
            **FileInputStream** fis = **new** **FileInputStream**("narnia.ser");  
            **ObjectInputStream** ois = **new** **ObjectInputStream**(fis);  
            **Book** oldNarnia = (**Book**) ois.readObject();  
           
            **System**.out.println("Book successfully created from Serialized data");  
            **System**.out.println("Book after seriazliation : " + oldNarnia);  
           
        } **catch** (**Exception** e) {  
            e.printStackTrace();  
        }  
  
    }  
   
   
}  
  
*/\*  
 \* A class which implements Serializable interface and has a transient variable.  
 \*/*  
**class** **Book** **implements** **Serializable**{  
    **private** **int** ISBN;  
    **private** **String** title;  
    **private** **String** author;  
    **private** **transient** **int** edition = 1; *//transient variable not serialized*  
  
    **public** **Book**(**int** ISBN, **String** title, **String** author, **int** edition) {  
        **this**.ISBN = ISBN;  
        **this**.title = title;  
        **this**.author = author;  
        **this**.edition = edition;  
    }  
  
    @**Override**  
    **public** **String** toString() {  
        **return** "Book{" + "ISBN=" + ISBN + ", title=" + title + ", author=" + author + ", edition=" + edition + '}';  
    }  
   
}  
  
Output:  
Before Serialization: **Book**{ISBN=1024, title=Narnia, author=unknown, edition=2}  
**Book** is successfully Serialized  
**Book** successfully created from Serialized data  
**Book** after seriazliation : **Book**{ISBN=1024, title=Narnia, author=unknown, edition=0}

If we look at this example of Serializing Object in Java we will realize that value **of transient variables are not serialized** and persisted and during deserialization, those values are initialized with there default value which is zero in case of int variable. Since constructor also didn't run during de-serialization it won't get the value provided during constructor. In Summary use transient variable carefully in Java.

**Ques. Difference between transient vs volatile variable or modifier in Java**

**Ans.** Both transient and volatile modifiers are completely different to each other. Main difference between transient vs volatile variable is that **transient variables are not serialized during Serialization process in Java** while **volatile variables are used to provide alternative synchronization in Java**. By using volatile keyword or modifier on fields, signals compiler that this variable is accessed by multiple thread and any reordering or optimization by compiler should be avoided.

**Q. Difference between transient vs volatile variables Java**

**Ans**.   
**1)** By making a variable or field transient in a Class **prevents it from being Serialized in Java**. Along with **static variables, transient variables are not serialized** during Serialization, and they are **initialized with their default value during deserialization** process e.g. an int transient variable is initialized with zero during deserialization in Java.

**2)** On the other hand **volatile variables are used in Concurrent programming in Java**. When we declare a variable volatile, **every thread reads its value from main memory and don't used cached value available in every thread stack**. **volatile variable also prevents compiler from doing reordering** which can compromise synchronization.

Many Java programmer though know about volatile variables they are not sure where to use volatile modifier in Java. One of the popular **example of  using volatile variable is implementing double checked locking in Singleton,** where Singleton instance of class is declared volatile in Java.  
**Transient variables are used to prevent serialization** or a field while **volatile variables are used to prevent reordering and avoid reading cached value of field in multithreaded Java program**. Only similarity between transient and volatile keyword is that **they are only applicable to field** or properties of class. We can not use transient or volatile keyword during class or method declaration in Java.

**Ques. What is Volatile Variable in Java - When to use in Thread? Example**

**Ans.** volatile variable in Java is a special variable which is used to signal threads, a compiler that this particular variable value are going to be updated by multiple threads inside Java application. By making a variable volatile using the volatile keyword in Java, application programmer **ensures that its value should always be read from main memory** and **thread should not use cached value of that variable from their own stack.** With the introduction of Java memory model from Java 5 onwards along with introduction of CountDownLatch, CyclicBarrier, Semaphore and ConcurrentHashMap, volatile variable also guarantees "happens-before" relationship, which means not only another thread has visibility of latest value of volatile variable but also all the variable is seen by the thread which has updated value of volatile variable before these threads sees it.

**Important point related to volatile keyword in Java**

The volatile keyword **can only be applied to a variable, it can not be applied to class or method.** using volatile keyword along with class and method is a compiler error.

A volatile is also referred as modifier in Java.

When to use Volatile variable in Java  
Here are some of the scenario where we can use volatile variable in Java :  
  
1) Any **variable which is shared between multiple threads** should be made variable, in order **to ensure that all thread must see the latest value of the volatile variable.**  
  
2) A signal to compiler and JIT to ensure that compiler does not change ordering or volatile variable and moves them out of synchronized context.  
  
3) We want to **save the cost of synchronization as volatile variables are less expensive** than synchronization.

**Ques. How to Serialize Object in Java - Serialization Example**

**Ans**The Serialisation mechanism is provided by Java to save and restore the state of an object programmatically. Serializing an Object in Java means converting into a wire format so that we can either **persist its state in a file locally** or **transfer it to another client via the network**, hence it becomes an extremely important concept in distributed applications running across several JVMs.

**Java Program to Serialize an Object using Serializable interface**

Here is our Java program to demonstrate how to serialize and de-serialize an object in Java. This program contains two classes Shoe and SerializationDemo, the first class is our POJO, we will create object of this class and serialieze it. The second class is our application class which contains the main() method, all the code to create object, saving it, and finally restoring it written inside main method.

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

import java.util.logging.Logger;

/\*\*

\* Simple example of Serialization in Java. We first create a Shoe object, then

\* serializes it and finally restored it by using de-serialization.

\*

\* @author Javin Paul

\*/

public class Pattern {

public static void main(String args[]) throws IOException,

ClassNotFoundException {

Shoe whiteNikeShoe = new Shoe("Nike", 1000, 9, "WHITE", true);

System.out.println("Before Serialization");

whiteNikeShoe.print();

// serializing shoe object

writeShoe(whiteNikeShoe);

// creating another Shoe with different brand

Shoe blackAdidasShoe = new Shoe("Adidas", 2000, 8, "Black", true);

// deserializing shoe object

whiteNikeShoe = (Shoe) readShoe();

System.out.println("After DeSerialization");

whiteNikeShoe.print();

}

private static void writeShoe(Serializable shoe) throws IOException {

ObjectOutputStream oos = new ObjectOutputStream(

new FileOutputStream(new File("shoe.ser")));

oos.writeObject(shoe);

oos.close();

}

private static Object readShoe() throws IOException, ClassNotFoundException {

ObjectInputStream ois = new ObjectInputStream(

new FileInputStream(new File("shoe.ser")));

Object obj = ois.readObject();

return obj;

}

}

class Shoe implements Serializable {

// static final variable

private static final long serialVersionUID = 40L;

private static final Logger logger = Logger.getLogger(Shoe.class.getName());

// static variable but not final

private static String \_brand;

// instance variable

private int \_id;

private int \_size;

private String \_color;

// transient variable

**private transient boolean \_isRunningShoe;**

// non serializable field

Thread \_thread;

public Shoe(String brand, int id, int size, String color,

boolean isRunningShoe) {

System.out.println("Inside Constructor");

\_brand = brand;

\_id = id;

\_size = size;

\_color = color;

\_isRunningShoe = isRunningShoe;

}

public String brand() {

return \_brand;

}

public int id() {

return \_id;

}

public int size() {

return \_size;

}

public String color() {

return \_color;

}

public void print() {

System.out.println("SerialVersionUID (final static field) : "

+ serialVersionUID);

System.out.println("logger ((final static field) : " + logger);

System.out.println("\_brand (static field) : " + \_brand);

System.out.println("\_id (instance variable) : " + \_id);

System.out.println("\_size (instance variable) : " + \_size);

System.out.println("\_color (instance variable) : " + \_color);

System.out.println("\_isRunningShoed (transient variable) : "

+ \_isRunningShoe);

System.out.println("\_thread (non-serializable field) : " + \_thread);

}

}

Output

Inside Constructor

Before Serialization

SerialVersionUID (final static field) : 40

logger ((final static field) : java.util.logging.Logger@42aab87f

\_brand (static field) : Nike

\_id (instance variable) : 1000

\_size (instance variable) : 9

\_color (instance variable) : WHITE

\_isRunningShoed (transient variable) : true

\_thread (non-serializable field) : null

Inside Constructor

After DeSerialization

SerialVersionUID (final static field) : 40

logger ((final static field) : java.util.logging.Logger@42aab87f

\_brand (static field) : Adidas

\_id (instance variable) : 1000

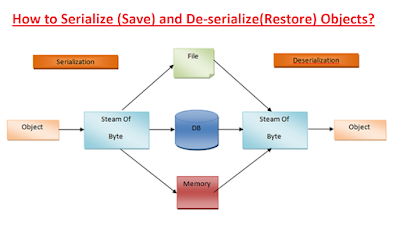
\_size (instance variable) : 9

\_color (instance variable) : WHITE

\_isRunningShoed (transient variable) : false

\_thread (non-serializable field) : null

A picture is worth a thousand word, so here is a diagram which explains the serialization and deserialization process from a 10K feet view:



Observation and Explanation

Now let's try to understand what happened when we serialize an instance of Shoe and later when we de-serialized it. I have created different types of variable in this class to show that whether their value is persisted or restored during Serialization or not. In Shoe class, we have two static final fields, SerialVersionUID, and Logger; their values are not persisted but because they are static. They are also initialized at the time of class loading, so they are fine. Also, because they are final, there is no danger of someone changing their value.  
  
The SerialVersionUID is very important field for a serializable class and we should always define it. If we don't define then JVM will calculate this value by reading structure of class e.g. number of instance variable, their types etc. This means, next time we add another instance variable or we rmeove old one, we risk of getting a different SerialVersionUID and if that happens we won't be able to restore object saved by previous version of your program.  
  
This has actually happend to us when one of the developer accidently removed the SerialVersionUID from one of the user preferences class and when user download and run the new version of our Java application, his preferences was all gone. He was a trader and for them their preferences means a lot, it was hard time to console and pacify him before we reverted back his GUI to previous version and restored his preferences. I am sure, we would never want to upset your clients and user.  
  
Now let's come to a simple, non-final static variable, its value is also not persisted during Serialization, that's why we see \_brand=Nike before Serialization and \_brand=Adidas after. Do we know why it happened? because when we created another instance of Shoe for Adidas, we reset value of this variable to "Adidas", and since the constructor is not called during deserialization, and class is not loaded again, its value remains "Adidas". It could have been null if de-serialization would have taken place at another JVM instance.  
  
Now let's see our three instance variables \_id, \_size, and \_color, their values are persisted during serialization and restored during de-serialization. This is why we see correct values for these three fields. Next is our transient variable isRunningShoe, this is tricky one, value of a transient variable is not stored during Serialization and that's why the value of isRunningShoe is incorrect after de-serialization.  
  
It was true before but it became false after de-serialization. We would not have noticed this had that boolean variable was initialized as false, and we would have thought that value of the transient variable was saved and restored, which is not true. So beware of default values during Serialization, variables like static and transient will be initialized to their default value after de-serialization.  
  
Last is our non-serializable instance variable \_thread, which holds the instance of java.lang.Thread, which doesn't implement the Serializable interface. It's really interesting that default Serialization process doesn't complain about this variable, this can also be tricky to understand if don't pay enough attention. The reason was that variable didn't hold any value.  
  
Now just initialize that variable as Thread \_thread = new Thread() and re-run the program again. This time,will get the following Exception :

Exception in thread "main" java.io.NotSerializableException: java.lang.Thread

at java.io.ObjectOutputStream.writeObject0(Unknown Source)

at java.io.ObjectOutputStream.defaultWriteFields(Unknown Source)

at java.io.ObjectOutputStream.writeSerialData(Unknown Source)

at java.io.ObjectOutputStream.writeOrdinaryObject(Unknown Source)

at java.io.ObjectOutputStream.writeObject0(Unknown Source)

at java.io.ObjectOutputStream.writeObject(Unknown Source)

at SerializationDemo.writeShoe(HelloHP.java:42)

at SerializationDemo.main(HelloHP.java:28)

Because Thread doesn't implement Serializable interface, we can not serialize it. This is a very common problem during maintenance of a legacy Java project. Suppose we have a Serializable class Employee, and later one developer introduced another instance variable Department, which is not Serializable. Do we know what will happen? The Employee can not be serialized anymore.  
  
That's why I recommend putting Serialization alert in the source file of a Serializable classes, reminding them about not adding any variable which is not serializable or making it transient if they really need it. We can see Java Coding Guidelines for more coding best pracitces while writing Java application. It contains 75 recommendations for reliable and secure Java programs.  
  
In short, we can say that :

The value of static variable is not persisted during Serialization.

Thetransient variables are not persisted as well.

Any NonSerializable field, which is not static or transient will break Serialization process by throwing ava.io.NotSerializableException.

The constructor of serializable class is not called during Serialization.

That's all about how to searialize an object in Java using Serializable interface. We have seen both saving and restoring object using serialization and de-serialization and also explored some key concepts related to how serialization works in Java. For example, transient and static variables are not saved during serialization, if we declare static variable, make sure it's not final, and to remember that constructor is not called during de-serialization process.  
  
All these concepts are very important to implement serialization correctly in your pgoram. We can further read Effective Java by Joshua Bloch to understand effective serialization e.g. with custom binary formats.  All the items related to Serialization in this book is must read for any serious Java developer.

**Q. What is Marker interface?**

**Ans**. A marker interface is **an empty interface without any method** but used to force some functionality in implementing classes by Java. Some of the well known marker interfaces are **Serializable and Cloneable.**

**Q. What is Java Reflection API? Why it’s so important to have?**

**Ans.** Java Reflection API provides the ability to inspect and modify the runtime behavior of java application. We can inspect a java class, interface, enum and get their methods and field details. Reflection API is an advanced topic and we should avoid it in normal programming. Reflection API usage can break the design pattern such as Singleton pattern by invoking the private constructor i.e violating the rules of access modifiers.

Even though we don’t use Reflection API in normal programming, it’s very important to have. We can’t have any frameworks such as Spring, Hibernate or servers such as Tomcat, JBoss without Reflection API. They invoke the appropriate methods and instantiate classes through reflection API and use it a lot for other processing.

**Q. What is composition in java?**

**Ans**. Composition is the design technique to **implement has-a relationship in classes**. We can use Object composition for code reuse.

Java composition is achieved by using instance variables that refer to other objects. The benefit of using composition is that we can control the visibility of other objects to client classes and reuse only what we need. Read more with example at Java Composition example.

**Q. What is the benefit of Composition over Inheritance?**

**Ans**. Any change in the superclass might affect subclass even though we might not be using the superclass methods. For example, if we have a method test() in the subclass and suddenly somebody introduces a method test() in the superclass, we will get compilation errors in the subclass. The composition will never face this issue because we are using only what methods we need.

Inheritance exposes all the superclass methods and variables to the client and if we have no control in designing superclass, it can lead to security holes. Composition allows us to provide restricted access to the methods and hence more secure.

We can get runtime binding in composition where inheritance binds the classes at compile time. So composition provides flexibility in the invocation of methods.

**Q. What is inner class in java?**

**Ans**. We can define a class inside a class and they are called nested classes. Any non-static nested class is known as an inner class. Inner classes are associated with the object of the class and **they can access all the variables and methods of the outer class**. Since inner classes are associated with the instance, we can’t have any static variables in them.

**Q. Exception handling Best Practices in Java Programming**

Ans.   
1) **Use Checked Exception for Recoverable error and Unchecked Exception for programming error.**

Choosing between checked and unchecked exception is always been confusing for Java programmers. Checked exceptions ensure that we provide an exception handling code for error conditions, which is away from language to enforcing we for writing robust code, but same time it also add lots of clutter into code and makes it unreadable.  
  
 Also, it seems reasonable to catch the exception and do something if we have alternatives or recovery strategies.

**2) Close or release resource in finally block**

This is a well known best practice in Java and quite a standard, while dealing with networking and IO classes. Closing resources in finally block guarantees that precious and scarce resource released properly in case of normal and aborted execution, guaranteed by finally block. From Java 7, language has a more interesting automatic resource management or ARM blocks, which can do this for we. Nevertheless, always remember to close resources in finally block, which is important to release limited resources like FileDescriptors, used in case of both socket and files.

**3) Including cause of Exception in stack-trace**

Many times Java library and open source code wraps one Exception into another, when one exception is thrown due to result of another exception. Its become extremely important to log or print cause of root exception. Java Exception class provides getCause() method to retrieve cause which can be used to provide more information about root cause of Exception. This Java best practice helps a lot while debugging or troubleshooting an issue. Always remember to pass original Exception, into constructor of new Exception, if we are wrapping one exception into another.

**4) Always provide meaning full message on Exception**

message of Exception is the most important place, where we can point out cause of problem because this is the first place every programmer looks upon. Always try to provide precise and factual information here. For example, compare these two Exception messages for IllegalArgumentException :

message 1: "Incorrect argument for method"

message 2: "Illegal value for ${argument}: ${value}

first one just says that argument is illegal or incorrect, but second one include both name of argument and its illegal value which is important to point out cause of error. Always follow this Java best practice, when writing code for handling exceptions and errors in Java.

**5) Avoid overusing Checked Exception**

Checked Exception has there advantage in terms of enforcement, but at same time it also litters the code and makes it unreadable by obscuring business logic. We can minimize this by not overusing checked Exception which result in much cleaner code. We can also use newer Java 7 features like one catch block for multiple exceptions and automatic resource management, to remove some duplication.

**6) Converting Checked Exception into RuntimeException**

This is one of the technique used to limit use of checked Exception in many of frameworks like Spring ,where most of checked Exception, which stem from JDBC is wrapped into DataAccessException, an unchecked Exception. This Java best practice provides benefits, in terms of  restricting specific exception into specific modules, like SQLException into DAO layer and throwing meaningful RuntimeException to client layer.

**7) Remember Exceptions are costly in terms of performance**

One thing which is worth remembering is that Exceptions are costly, and can slow down your code. Suppose we have method which is reading from ResultSet and often throws SQLException than move to next element, will be much slower than normal code which doesn't throw that Exception. So minimizing catching unnecessary Exception and moving on, without fixing there root cause. Don’t just throw and catch exceptions, if we can use boolean variable to indicate result of operation, which may result in cleaner and performance solution. Avoid unnecessary Exception handling by fixing root cause.

**8) Avoid empty catch blocks**

Nothing is more worse than empty catch block, because it not just hides the Errors and Exception, but also may leave your object in unusable or corrupt state. Empty catch block only make sense, if we absolutely sure that Exception is not going to affect object state on any ways, but still its better to log any error comes during program execution. This is not a Java best practice, but a most common practice, while writing Exception handling code in Java.

**9) Use Standard Exceptions**

Our ninth Java best practice advise on using standard and inbuilt Java Exceptions. Using standard Exception instead of creating own Exception every now and then is much better in terms of maintenance and consistency. Reusing standard exception makes code more readable, because most of  Java developers are familiar with standard RuntimeException from JDK like, IllegalStateException, IllegalArgumentException or NullPointerException, and they will immediately be able to know purpose of Exception, instead of looking out another place on code or docs to find out purpose of user defined Exceptions.

**10) Document Exception thrown by any method**

Java provides throw and throws keyword to throw exception and in javadoc we have @throw to document possible Exception thrown by any method. This becomes increasingly important if we are writing API or public interface. With proper documentation of Exception thrown by any method we can potentially alert anyone who is using it.

**Q. WHAT IS ENUM IN JAVA ?**

**Ans**. **Enum in Java is a keyword**, a feature which is used to represent fixed number of well- known values in Java, For example, Number of days in Week, Number of planets in Solar system etc. Enumeration (Enum) in Java was introduced in JDK 1.5 and it is one of my favorite features of J2SE 5 among Autoboxing and unboxing , Generics, varargs and static import. One of the common use of Enum which emerged in recent years is Using Enum to write Singleton in Java, which is by far easiest way to implement Singleton and handles several issues related to thread-safety and Serialization automatically. By the way, Java Enum as a type is more suitable to represent well known fixed set of things and state, for example representing the state of Order as NEW, PARTIAL FILL, FILL or CLOSED.

Enumeration(Enum) was not originally available in Java though it was available in another language like C and C++, but eventually, Java realized and introduced Enum on JDK 5 (Tiger) by keyword Enum.

In this Java Enum tutorial, we will see different Enum example in Java and learn using Enum in Java. Focus of this Java Enum tutorial will be on different features provided by Enum in Java and how to use them.

If we have used Enumeration before in C or C++ then we will not be uncomfortable with Java Enum but in my opinion, Enum in Java is more rich and versatile than in any other language.

**Q.How to represent enumerable value without Java enum ?**

**Ans**. Since Enum in Java is only available from Java 1.5 it's worth to discuss how we used to represent enumerable values in Java prior JDK 1.5 and without it. I use public static final constant to replicate enum like behavior. Let’s see an Enum example in Java to understand the concept better. In this example, we will use US Currency Coin as enumerable which has values like PENNY (1) NICKLE (5), DIME (10), and QUARTER (25).

public class CurrencyDenom {

public static final int PENNY = 1;

public static final int NICKLE = 5;

public static final int DIME = 10;

public static final int QUARTER = 25;

}

public class Currency {

private int currency; //CurrencyDenom.PENNY,CurrencyDenom.NICKLE,

// CurrencyDenom.DIME,CurrencyDenom.QUARTER

}

Though this can serve our purpose it has some serious limitations:

1) No Type-Safety: First of all it’s not type-safe; we can assign any valid int value to currency e.g. 99 though there is no coin to represent that value.

2) No Meaningful Printing: printing value of any of these constant will print its numeric value instead of meaningful name of coin e.g. when we print NICKLE it will print "5" instead of "NICKLE"

3) No namespace: to access the currencyDenom constant we need to prefix class name e.g. CurrencyDenom.PENNY instead of just using PENNY though this can also be achieved by using static import in JDK 1.5

Java Enum is the answer of all this limitation. Enum in Java is type-safe, provides meaningful String names and has their own namespace. Now let's see the same example using Enum in java:

public enum Currency {PENNY, NICKLE, DIME, QUARTER};

Here Currency is our enum and PENNY, NICKLE, DIME, QUARTER are enum constants. Notice curly braces around enum constants because Enum is a type like class and interface in Java. Also, we have followed the similar naming convention for enum like class and interface (first letter in Caps) and since Enum constants are implicitly static final we have used all caps to specify them like Constants in Java.

**Q. What is Enum in Java**

**Ans**. Now back to primary questions “What is Enum in java” simple answer Enum is a keyword in java and on more detail term Java Enum is a type like class and interface and can be used to define a set of Enum constants.

Enum constants are implicitly static and final and we can not change their value once created. Enum in Java provides type-safety and can be used inside switch statements like int variables.

Since enum is a keyword we can not use as a variable name and since it's only introduced in JDK 1.5 all your previous code which has an enum as a variable name will not work and needs to be refactored.

Benefits of using Enums in Java

1) Enum is type-safe we can not assign anything else other than predefined Enum constants to an Enum variable. It is a compiler error to assign something else, unlike the public static final variables used in Enum int pattern and Enum String pattern.

2) Enum has its own namespace.

3) The best feature of Enum is we can use Enum in Java inside Switch statement like int or char primitive data type. We will also see an example of using java enum in switch statement in this java enum tutorial.

4) Adding new constants on Enum in Java is easy and we can add new constants without breaking the existing code.

Important points about Enum in Java

1) Enums in Java are type-safe and have their own namespace. It means your enum will have a type for example "Currency" in the below example and we can not assign any value other than specified in Enum Constants.

**public enum Currency {**

**PENNY, NICKLE, DIME, QUARTER**

**};**

Currency coin = Currency.PENNY;

coin = 1; //compilation error

2) Enum in Java are reference types like class or interface and we can define constructor, methods, and variables inside java Enum which makes it more powerful than Enum in C and C++ as shown in next example of Java Enum type.

3) We can specify values of enum constants at the creation time as shown in below example:

public enum Currency {PENNY(1), NICKLE(5), DIME(10), QUARTER(25)};

But for this to work we need to define a member variable and a constructor because PENNY (1) is actually calling a constructor that accepts int value, see below example.

public enum Currency {

PENNY(1), NICKLE(5), DIME(10), QUARTER(25);

private int value;

private Currency(int value) {

this.value = value;

}

};

The constructor of enum in java must be private any other access modifier will result in compilation error. Now to get the value associated with each coin we can define a public getValue() method inside Java enum like any normal Java class. Also, the semicolon in the first line is optional.

4) Enum constants are implicitly static and final and can not be changed once created. For example, below code of java enum will result in compilation error:

Currency.PENNY = Currency.DIME;

The final field EnumExamples.Currency.PENNY cannot be reassigned.

5) Enum in java can be used as an argument on switch statement and with "case:" like int or char primitive type. This feature of java enum makes them very useful for switch operations. Let’s see an example of how to use java enum inside switch statement:

Currency usCoin = Currency.DIME;

switch (usCoin) {

case PENNY:

System.out.println("Penny coin");

break;

case NICKLE:

System.out.println("Nickle coin");

break;

case DIME:

System.out.println("Dime coin");

break;

case QUARTER:

System.out.println("Quarter coin");

}

from JDK 7 onwards we can also String in Switch case in Java code.

6) Since constants defined inside Enum in Java are final we can safely compare them using "==", the equality operator as shown in following example of Java Enum:

Currency usCoin = Currency.DIME;

if(usCoin == Currency.DIME){

System.out.println("enum in java can be compared using ==");

}

By the way comparing objects using == operator is not recommended, Always use equals() method or compareTo() method to compare Objects.

If we are not convinced than we should read this article to learn more about pros and cons of comparing two enums using equals() vs == operator in Java.

7) Java compiler automatically generates static values() method for every enum in java. Values() method returns array of Enum constants in the same order they have listed in Enum and we can use values() to iterate over values of Enum in Java as shown in below example:

for(Currency coin: Currency.values()){

System.out.println("coin: " + coin);

}

And it will print:

coin: PENNY

coin: NICKLE

coin: DIME

coin: QUARTER

Notice the order is exactly the same as defined order in the Enum.

8) In Java, Enum can override methods also. Let’s see an example of overriding toString() method inside Enum in Java to provide a meaningful description for enums constants.

public enum Currency {

........

@Override

public String toString() {

switch (this) {

case PENNY:

System.out.println("Penny: " + value);

break;

case NICKLE:

System.out.println("Nickle: " + value);

break;

case DIME:

System.out.println("Dime: " + value);

break;

case QUARTER:

System.out.println("Quarter: " + value);

}

return super.toString();

}

};

And here is how it looks like when displayed:

Currency usCoin = Currency.DIME;

System.out.println(usCoin);

Output:

Dime: 10

9) Two new collection classes EnumMap and EnumSet are added into collection package to support Java Enum. These classes are a high-performance implementation of Map and Set interface in Java and we should use this whenever there is any opportunity.

EnumSet doesn't have any public constructor instead it provides factory methods to create instance e.g. EnumSet.of() methods. This design allows EnumSet to internally choose between two different implementations depending upon the size of Enum constants.

If Enum has less than 64 constants than EnumSet uses RegularEnumSet class which internally uses a long variable to store those 64 Enum constants and if Enum has more keys than 64 then it uses JumboEnumSet. See my article the difference between RegularEnumSet and JumboEnumSet for more details.

10) We can not create an instance of enums by using new operator in Java because the constructor of Enum in Java can only be private and Enums constants can only be created inside Enums itself.

11) An instance of Enum in Java is created when any Enum constants are first called or referenced in code.

12) Enum in Java can implement the interface and override any method like normal class It’s also worth noting that Enum in java implicitly implements both Serializable and Comparable interface. Let's see and example of how to implement interface using Java Enum:

public enum Currency implements Runnable{

PENNY(1), NICKLE(5), DIME(10), QUARTER(25);

private int value;

............

@Override

public void run() {

System.out.println("Enum in Java implement interfaces");

}

}

13) We can define abstract methods inside Enum in Java and can also provide a different implementation for different instances of enum in java. Let’s see an example of using abstract method inside enum in java

public enum Currency {

PENNY(1) {

@Override

public String color() {

return "copper";

}

},

NICKLE(5) {

@Override

public String color() {

return "bronze";

}

},

DIME(10) {

@Override

public String color() {

return "silver";

}

},

QUARTER(25) {

@Override

public String color() {

return "silver";

}

};

private int value;

public abstract String color();

private Currency(int value) {

this.value = value;

}

}

In this example since every coin will have the different color we made the color() method abstract and let each instance of Enum to define their own color. We can get color of any coin by just calling the color() method as shown in below example of Java Enum:

System.out.println("Color: " + Currency.DIME.color());

So that was the comprehensive list of properties, behavior and capabilities of Enumeration type in Java. I know, it's not easy to remember all those powerful features and that's why I have prepared this small Microsoft powerpoint slide containing all important properties of Enum in Java. We can always come back and check this slide to revise important features of Java Enum.

Real world Examples of Enum in Java

So far we have learned what Enum can do for we in Java. We learned that enum can be used to represent well known fixed set of constants, enum can implement interface, it can be used in switch case like int, short and String and Enum has so many useful built-in metods like values(), vlaueOf(), name(), and ordinal(), but we didn't learn where to use the Enum in Java?

I think some real world examples of enum will do a lot of good to many pepole and that's why I am going to summarize some of the popular usage of Enum in Java world below.

Enum as Thread Safe Singleton

One of the most popular use of Java Enum is to impelment the Singleton design pattern in Java. In fact, Enum is the easieset way to create a thread-safe Singleton in Java. It offer so many advantage over traditional implementation using class e.g. built-in Serialization, guarantee that Singleton will always be Singleton and many more. I suggest we to check my article about Why Enum as Singelton is better in Java to larn more on this topic.

Strategy Pattern using Enum

We can also implement the Strategy design pattern using Enumeration type in Java. Since Enum can implement interface, it's a good candidate to implement the Strategy interface and define individual strategy. By keeping all related Strategy in one place, Enum offer better maintainence support. It also doesn't break the open closed design principle as per se because any error will be detected at compile time. See this tutorial to learn how to implement Strategy pattern using Enum in Java.

Enum as replacement of Enum String or int pattern

There is now no need to use String or integer constant to represent fixed set of things e.g. status of object like ON and OFF for a button or START, IN PROGRESS and DONE for a Task. Enum is much better suited for those needs as it provide compile time type safety and better debugging assistent than String or Integer.

Enum as State Machine

We can also use Enum to impelment State machine in Java. A State machine transition to predifine set of states based upon current state and given input. Since Enum can implement interface and override method, we can use it as State machine in Java. See this tutorial from Peter Lawrey for a working example.

Enum Java valueOf example

One of my readers pointed out that I have not mentioned about the valueOf method of enum in Java, which is used to convert String to enum in Java.

Here is what he has suggested, thanks @ Anonymous

“We could also include valueOf() method of enum in java which is added by compiler in any enum along with values() method. Enum valueOf() is a static method which takes a string argument and can be used to convert a String into an enum. One think though we would like to keep in mind is that valueOf(String) method of enum will throw "Exception in thread "main" java.lang.IllegalArgumentException: No enum const class" if we supply any string other than enum values.

Another of my reader suggested about ordinal() and name() utility method of Java enum Ordinal method of Java Enum returns the position of a Enum constant as they declared in enum while name()of Enum returns the exact string which is used to create that particular Enum constant.” name() method can also be used for converting Enum to String in Java.

That’s all on Java enum, Please share if we have any nice tips on enum in Java and let us know how we are using java enum in your work. We can also follow some good advice for using Enum by Joshua Bloch in his all time classic book Effective Java. That advice will give we more idea of using this powerful feature of Java programming language.

**Q. 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 forms 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, when one class owns other classes and other classes 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 its body part ceased to exist meaningfully, this is one example of Composition.  
  
Programmers often confuse between Association, Composition, and Aggregation in Object-oriented design discussions, this confusion also makes the difference between Association, Composition, and Aggregation one of the popular questions in Java Interviews, only after the difference between abstract class and interface.  
  
Another example of Composition is Car and it's part e.g. engines, wheels, etc. Individual parts of the car can not function when a car is destroyed.  While in the case of Aggregation, including object that can exist without being part of the main object like a Player which is part of a Team, can exist without a team and can become part of other teams as well.  
  
Another example of Aggregation is Student in School class when School closed, Student still exists and then can join another School or so.  In UML notation, a composition is denoted by a filled diamond, while aggregation is denoted by an empty diamond, which shows their obvious difference in terms of strength of the relationship.  
  
The composition is stronger than Aggregation.  In Short, a relationship between two objects is referred to as an association, and an association is known as composition when one object owns another while an association is known as aggregation when one object uses another object.  
  
In this OOPS tutorial, we will see a couple of more examples to understand the difference between Association, Composition, and Aggregation better.  
  
Btw, if we are new into the world of object-oriented programming and design then I also suggest we go through a hands-on course like UML and Object-Oriented Design Foundations by Karloy Neisztor on Udemy. It's a great course to understand the complete process of object-oriented analysis and design for creating quality software.

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, we can gauge the 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.

1. 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;

}

2. 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;

}

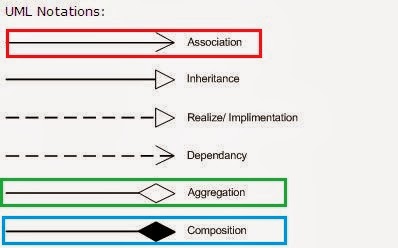
public class Person {

private String name;

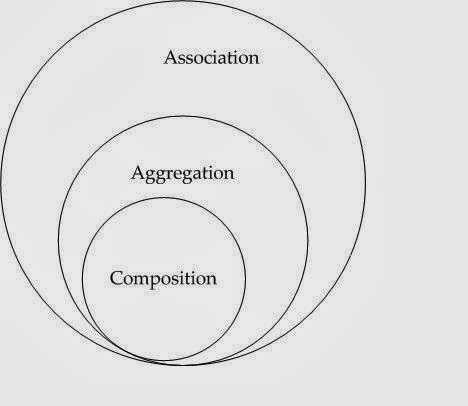
}

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 a filled diamond-head arrow.  
  
When we draw a UML diagram for two related classes 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 a different kind of dependency between two classes.

[](https://1.bp.blogspot.com/-VL_9cjhwEE4/UvJN__IvaBI/AAAAAAAABCc/IkDmShgM-Yc/s1600/Association,+Composition+UML.JPG)

As I said all three denotes relationship between object and only differ in their strength, we 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, 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, we can use final keyword 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.  
  
That's all on the difference between Association, Composition and Aggregation in UML, Java and Object oriented design. Since object oriented analysis is more about defining the relationship between object, it's important to know what kind of relationship exists between them, composition and aggregation is a two way of representing relationship between two objects.