## DRY (Don't repeat yourself)

Our first object oriented design principle is DRY, as name suggest **DRY (don't repeat yourself)** means don't write duplicate code, instead use [Abstraction](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html) to abstract common things in one place. If you have block of code in more than two place consider making it a separate method, or if you use a hard-coded value more than one time make them [public final constant](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html). Benefit of this Object oriented design principle is in maintenance. It's important  not to abuse it, duplication is not for code, but for functionality . It means, if you used common code to validate OrderID and SSN it doesn’t mean they are same or they will remain same in future. By using common code for two different functionality or thing you closely couple them forever and when your OrderID changes its format , your SSN validation code will break. So beware of such coupling and just don’t combine anything which uses similar code but are not related.

Encapsulate What Changes

Only one thing is constant in software field and that is "Change", So encapsulate the code you expect or suspect to be changed in future. Benefit of this OOPS Design principle is that Its easy to test and maintain proper encapsulated code. If you are coding in Java then follow principle of making variable and methods private by default and increasing access step by step e.g. from private to protected and not public. Several of **design pattern in Java** uses Encapsulation,[Factory design pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) is one example of Encapsulation which encapsulate object creation code and provides flexibility to introduce new product later with no impact on existing code.

Open Closed Design Principle

Classes, methods or functions should be Open for extension (new functionality) and Closed for modification. This is another beautiful SOLID design principle, which prevents some-one from changing already tried and tested code. Ideally if you are adding new functionality only than your code should be tested and that's the goal of [Open Closed Design principle](http://javarevisited.blogspot.com/2011/11/great-example-of-open-closed-design.html). By the way, Open Closed principle is "O" from SOLID acronym.

Single Responsibility Principle (SRP)

Single Responsibility Principle is another SOLID design principle, and represent  "S" on SOLID acronym. As per SRP, there should not be more than one reason for a class to change, or a class should always handle single functionality. If you put more than one functionality in one [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html)  it introduce **coupling** between two functionality and even if you change one functionality there is chance you broke coupled functionality,  which require another round of testing to avoid any surprise on production environment.

Dependency Injection or Inversion principle

Don't ask for dependency it will be provided to you by framework. This has been very well implemented in Spring framework, beauty of this **design principle** is that any class which is injected by DI framework is easy to test with mock object and easier to maintain because object creation code is centralized in framework and client code is not littered with that.There are multiple ways to  implemented **Dependency injection** like using  byte code instrumentation which some AOP (Aspect Oriented programming) framework like AspectJ does or by using proxies just like used in Spring. See this [example of IOC and DI design pattern](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html) to learn more about this SOLID design principle. It represent "D" on SOLID acronym.

Favor Composition over Inheritance

Always *favor composition over inheritance* ,if possible. Some of you may argue this, but I found that Composition is lot more flexible than [Inheritance](http://javarevisited.blogspot.sg/2012/10/what-is-inheritance-in-java-and-oops-programming.html). Composition allows to change behavior of a class at runtime by setting property during runtime and by using Interfaces to compose a class we use [polymorphism](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html) which provides flexibility of to replace with better implementation any time. Even Effective Java advise to favor composition over inheritance.

Liskov Substitution Principle (LSP)

According to Liskov Substitution Principle, Subtypes must be substitutable for super type i.e. methods or functions which uses super class type must be able to work with [object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html)of sub class without any issue". LSP is closely related **to Single responsibility principle** and **Interface Segregation Principle**. If a class has more functionality than subclass might not support some of the functionality ,and does violated LSP. In order to follow **LSP SOLID design principle**, derived class or sub class must enhance functionality, but not reduce them. LSP represent  "L" on SOLID acronym.

Interface Segregation principle (ISP)

Interface Segregation Principle stats that, a client should not implement an [interface](http://javarevisited.blogspot.com/2012/04/10-points-on-interface-in-java-with.html), if it doesn't use that. This happens mostly when one interface contains more than one functionality, and client only need one functionality and not other.Interface design is tricky job because once you release your interface you can not change it without breaking all implementation. Another benefit of this design principle in Java is, interface has disadvantage to implement all method before any class can use it so having single functionality means less method to implement.

Programming for Interface not implementation

Always *program for interface and not for implementation* this will lead to flexible code which can work with any new implementation of interface. So use interface type on variables, return types of method or argument type of methods in Java. This has been advised by many Java programmer including in Effective Java and head first design pattern book.

Delegation principle

Don't do all stuff  by yourself,  delegate it to respective class. Classical example of delegation design principle is [equals() and hashCode() method in Java](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html). In order to compare two object for equality we ask class itself to do comparison instead of Client class doing that check. Benefit of this design principle is no duplication of code and pretty easy to modify behavior.

All these **object oriented design principle**helps you write flexible and better code by striving high cohesion and low coupling. Theory is first step, but what is most important is to *develop ability to find out when to apply these design principle*. Find out, whether we are violating any design principle and compromising flexibility of code, but again as nothing is perfect in this world, don't always try to solve problem with **design patterns and design principle** they are mostly for large enterprise project which has longer maintenance cycle.

Read more: <http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html#ixzz2ziPvVjO2>

[**What is Factory method Design Pattern in Java with Example - Tutorial**](http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html)

**Factory design pattern in Java** one of the core design pattern which is used heavily not only in JDK but also in various Open Source framework such as Spring, Struts and Apache along with [decorator design pattern in Java](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html). Factory Design pattern is based on [**Encapsulation**](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html)object oriented concept. Factory method is used to create different object from factory often refereed as Item and it encapsulate the creation code. So instead of having object creation code on client side we encapsulate inside **Factory method in Java**. One of the best examples of factory pattern in Java is BorderFactory Class of Swing API. In this Design pattern tutorial we will see **What is Factory method design pattern in Java**, What are main *advantages of factory pattern in Java* , Code example of Factory design pattern and What problem **Factory pattern** solves in Java or when to use Factory design pattern.   
This article is in continuation of my design pattern article as [10 OOPS and SOLID design principles java programmer should know](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html)and [How to use Observer pattern in Java](http://javarevisited.blogspot.com/2011/12/observer-design-pattern-java-example.html)

What is static factory method or factory design pattern

[factory design patter in java tutorial with example](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)Factory design pattern is used to create objects or [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html) and it provides loose coupling and high cohesion. Factory pattern encapsulate object creation logic which makes it easy to change it later when you change how object gets created or you can even introduce new object with just change in one class. In GOF pattern list Factory pattern is listed as Creation design pattern. Factory should be an interface and clients first either creates factory or get factory which later used to create objects.

Example of static factory method in JDK

 Best Example of Factory method design pattern is valueOf() method which is there in String and wrapper classes like Integer and Boolean and used for type conversion i.e. from converting String to Integer or String to double in java..

Some more examples of factory method design pattern from JDK is :

valueOf() method which returns object created by factory equivalent to value of parameter passed.

getInstance() method which creates instance of Singleton class.

newInstance() method which is used to create and return new instance from factory method every time called.

getType() and newType() equivalent of getInstance() and newInstance() factory method but used when factory method resides in separate class.

**Problem which is solved by Factory method Pattern in Java**

Whenever we talk about **object oriented language** it will based upon some concept like [abstraction](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html), [polymorphism](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html) etc and on that [encapsulation](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html)and delegation are important concept any design will be called good if task are delegated to different object and some kind of encapsulation is there.

Some time our application or framework will not know that what kind of object it has to create at run-time it knows only the interface or abstract class and as we know we can not create object of interface or abstract class so main problem is frame work knows **when** it has to create but don’t know **what kind** of object.

Whenever we create object using new() we violate **principle of programming for interface rather than implementation** which eventually result in inflexible code and difficult to change in maintenance. By using Factory design pattern in Java we get rid of this problem.

Another problem we can face is class needs to contain objects of other classes or class hierarchies within it; this can be very easily achieved by just using the new keyword and the class constructor. The problem with this approach is that it is a very hard coded approach to create objects as this creates dependency between the two classes.

So **factory pattern** solve this problem very easily by model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate, Factory Pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code. The **factory methods** are typically implemented as virtual methods, so this pattern is also referred to as the “**Virtual Constructor**”. These methods create the objects of the products or target classes.

**When to use Factory design pattern in Java**

* Static Factory methods are common in frameworks where library code needs to create objects of types which may be sub classed by applications using the framework.
* Some or all concrete products can be created in multiple ways, or we want to leave open the option that in the future there may be new ways to create the concrete product.
* Factory method is used when Products don't need to know how they are created.
* We  can use factory pattern where we have to create an object of any one of sub-classes depending on the data provided

**Code Example of Factory Design Pattern in Java:**

Let’s see an example of how factory pattern is implemented in Code.We have requirement to create multiple currency e.g. INR, SGD, USD and code should be extensible to accommodate new Currency as well. Here we have made Currency as interface and all currency would be concrete implementation of Currency interface. Factory Class will create Currency based upon country and return concrete implementation which will be stored in interface type. This makes code dynamic and extensible.

Here is complete **code example of Factory pattern in Java**:

**interface** Currency {

       String getSymbol();

}

// Concrete Rupee Class code

**class** Rupee **implements** Currency {

       @Override

**public** String getSymbol() {

**return** "Rs";

       }

}

// Concrete SGD class Code

**class** SGDDollar **implements** Currency {

       @Override

**public** String getSymbol() {

**return** "SGD";

       }

}

// Concrete US Dollar code

**class** USDollar **implements** Currency {

       @Override

**public** String getSymbol() {

**return** "USD";

       }

}

// Factroy Class code

**class** CurrencyFactory {

**public** **static** Currency createCurrency (String country) {

**if** (country. equalsIgnoreCase ("India")){

**return** **new** Rupee();

       }**else** **if**(country. equalsIgnoreCase ("Singapore")){

**return** **new** SGDDollar();

       }**else** **if**(country. equalsIgnoreCase ("US")){

**return** **new** USDollar();

        }

**throw** **new** IllegalArgumentException("No such currency");

       }

}

// Factory client code

**public** **class** Factory {

**public** **static** **void** main(String args[]) {

              String country = args[0];

              Currency rupee = CurrencyFactory.*createCurrency*(country);

              System.*out*.println(rupee.getSymbol());

       }

}

**Advantage of Factory method Pattern in Java:**

**Factory pattern in Java** is heavily used everywhere including JDK, open source library and other frameworks.In following are main advantages of using Factory pattern in Java:

1*) Factory method design pattern* decouples the calling class from the target class, which result in less coupled and highly cohesive code?

E.g.: JDBC is a good example for this pattern; application code doesn't need to know what database it will be used with, so it doesn't know what database-specific driver classes it should use. Instead, it uses factory methods to get Connections, Statements, and other objects to work with. Which gives you flexibility to change your back-end database without changing your DAO layer in case you are using ANSI SQL features and not coded on DBMS specific feature?

2) Factory pattern in Java enables the subclasses to provide extended version of an object, because creating an object inside factory is more flexible than creating an object directly in the client. Since client is working on interface level any time you can enhance the implementation and return from Factory.

3) Another benefit of using *Factory design pattern in Java* is that it encourages [consistency in Code](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) since every time object is created using Factory rather than using different constructor at different client side.

4) Code written using Factory design pattern in Java is also [easy to debug](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html) and troubleshoot because you have a centralized method for object creation and every client is getting object from same place.

Some more advantages of factory method design pattern is:

1. **Static factory method** used in factory design pattern enforces use of Interface than implementation which itself a good practice. for example:

**Map** synchronizedMap = [**Collections**](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html).synchronizedMap(**new** **HashMap**());

2. Since static factory method have return type as Interface, it allows you to replace implementation with better performance version in newer release.

3. Another advantage of static factory method pattern is that they can cache frequently used object and eliminate duplicate object creation. Boolean.valueOf() method is good example which caches true and false boolean value.

4. Factory method pattern is also recommended by [Joshua Bloch in Effective Java.](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20)

5 Factory method pattern offers alternative way of creating object.

6. Factory pattern can also be used to hide information related to creation of object.

That’s all on **Factory design patten in Java** for now. This is one of the most used patterns in Java library and different Java frameworks. Summary is try to use **Factory pattern** whenever you see an opportunity to encapsulate object creation code and see chance of creating different object in near future.

# [Decorator design Pattern in Java with Example Java Tutorial](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html)

I was thinking to write on **decorator design pattern in Java** when I first wrote [10 interview questions on Singleton Pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html). Since design pattern is quite important while building software and it’s equally important on any Core Java Interview, It’s always good to have clear understanding of various design patterns in Java. In this article we will explore and learn **Decorator Design pattern in Java**which is a prominent core Java design pattern and you can see lot of its example in JDK itself. JDK use decorator pattern in IO package where it has decorated Reader and Writer Classes for various scenario, for example BufferedReader and BufferedWriter are example of decorator design pattern in Java. From design perspective its also good idea to learn how existing things work inside JDK itself for example [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html) or [How SubString method work in Java](http://javarevisited.blogspot.com/2011/10/how-substring-in-java-works.html), that will give you some idea of things you need to keep in mind while designing your Class or interface in Java. Now let’s Move on to **Decorator pattern in Java**.

## Java Decorator Design Pattern

In this Java tutorial we will see:

What is decorator pattern in Java?

When to use decorator pattern in Java?

How to use decorator pattern in Java?

Example of decorator design pattern

Advantage and Disadvantage of decorator pattern in Java

### What is decorator design pattern in Java?

          Decorator design pattern is used to **enhance the functionality of a particular object at run-time** or dynamically.

          At the same time **other instance of same class will not be affected by this** so individual object gets the new behavior.

          Basically we wrap the original object through decorator object.

          Decorator design pattern is based on abstract classes and we derive concrete implementation from that classes,

          It’s a structural design pattern and most widely used.

I prefer to answer *What is decorator design pattern* in point format just to stress on important point like this pattern operator at individual object level. This question also asked in many [Core Java interviews in Investment banks](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html)

### Problem which is solved by Decorator Pattern:

[decorator design pattern java example code](http://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html)Now the question is why this pattern has came into existence what is the problem with existing system, so the answer is if anyone wants to add some functionality to individual object or change the state of particular object at run time it is not possible what the possible is we can provide the specific behavior to all the object of that class at design time by the help of inheritance or using subclass, but **Decorator pattern** makes possible that we provide individual object of same class a specific behavior or state at run time. This doesn’t affect other object of same [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html).

**When to use Decorator pattern in Java**

          When sub classing is become impractical and we need large number of different possibilities to make independent object or we can say we have number of combination for an object.

          Secondly when we want to add functionality to individual object not to all object at run-time we use decorator design pattern.

**Code Example of decorator design pattern:**

To better understand concept of decorator design pattern let see a code example using Decorator Pattern in Java. You can also look inside JDK and find what are classes and packages which are using decorator pattern.

// Component on Decorator design pattern

**public** **abstract** **class** Currency **{**  
 String description = "Unknown currency";

**public** String getCurrencyDescription**()** **{**  
  **return** description;  
 **}**

 public abs**tract** **double** cost**(double** value**)**;

**}**

// Concrete Component

**public** **class**Rupee **extends** Currency**{**

**double**value**;**

**public** Rupee**()** **{**  
  description = "indian rupees";  
 **}**

 public double cost(double v){  
  value=v;

  returnvalue;  
 **}**

**}**

//Another Concrete Component

public class Dollar extends Currency{

**double**value**;**

 public Dollar () {  
  description = "Dollar”;  
 }

**public** **double** cost**(double v){**

**value=v;**

**return** value;

**}**

**}**

// Decorator

**public** **abstract** **class** Decorator **extends** Currency**{**

**public** **abstract** String getDescription**()**;

**}**

// Concrete Decorator

**public** **class** USDDecorator **extends** Decorator**{**

 Currency currency;

**public** USDDecorator**(**Currency currency**){**  
  **this**.currency = currency;  
 **}**

**public** String getDescription**(){**  
  **return** currency.getDescription**()**+" ,its US Dollar";  
 **}**

**}**

//Another Concrete Decorator

**public** **class** SGDDecorator **extends** Decorator**{**  
 Currency currency;

**public** SGDDecorator**(**Currency currency**){**  
  **this**.currency = currency;  
 **}**

**public** String getDescription**(){**  
  **return** currency.getDescription**()**+" ,its singapore Dollar";  
 **}**

**}**

Now its time to check currency.

**public** **class** CurrencyCheck **{**

**public** **static** **void** main**(**String**[]** args**)** **{**

  // without adding decorators

  Currency curr = **new** Dollar**()**;

  System.out.println**(**curr.getDescription**()** +" dollar. "+curr.cost**(2.0))**;

  //adding decorators

  Currency curr2 = **new** USDDecorator**(new** Dollar()**)**;

  System.out.println**(**curr2.getDescription**()** +" dollar. "+curr2.cost**(4.0))**;

Currency curr3 = **new** SGDDecorator**(new** Dollar()**)**;

  System.out.println**(**curr3.getDescription**()** +" dollar. "+curr3.cost**(4.0))**;

}

**Explanation of the code**:

We can understand this in following term;

1.      **Component Interface**: In our example Currency interface is component which used on its own or we need decorator for that.

2.      **Concrete Component: it** implements Component and we add new behavior to this object at dynamically. Dollar and Rupee are the concrete implementation of currency.

3.      **Decorator: Decorator** contains a HAS a Relationship in simple word we can say it has a instance variable that holds reference for component they implement same component which they are going to decorate. Here a Decorator is an abstract class which extends the currency.

4.      **Concrete Decorator:** it’s an implementation of Decorator So USD Dollar and SGD Dollar are the implementation of Decorator contains instance variable for component interface or the thing which they are going to decorate.

**Advantage of Decorator design Pattern in Java**

In brief we see what the main advantages of using decorator design patterns are.

1.      Decorator Pattern is flexible than inheritance because inheritance add responsibilities at compile time and it will add at run-time.

2.      Decorator pattern enhance or modify the object functionality

**Disadvantage**

Main disadvantage of using Decorator Pattern in Java is that the code maintenance can be a problem as it provides a lot of similar kind of small objects (each decorator).

That’s all on **decorator design pattern in Java**. To get mastery on decorator pattern I suggest looking inside JDK library itself and finding what classes are decorated, why they are decorated. Also think of scenario where inheritance is impractical and you look more flexibility and try to **use decorator pattern in Java** there.

# [10 Singleton Pattern Interview questions in Java - Answered](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html)

**Singleton pattern in Java** is one of the most common patterns available and it’s also used heavily in core Java libraries. Questions from Singleton pattern is very common in Java interviews and good knowledge of how to implement Singleton pattern certainly help.This is also one of my favorite [design pattern interview question](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html) and has lots of interesting follow-up to dig into details , this not only check the knowledge of design pattern but also check coding, multithreading aspect which is very important while working for a real life application.  In this post  have listed some of the most common question asked on Singleton pattern during a Java Interview. I have not provided the answers of these questions as they are easily available via google search but if you guys need I can try to  modify this tutorial to include answers as well. As promised earlier and having received lot of request for providing answers of these question, I have decided to update this post along with answers. By the way if you are preparing for interview on Java technology than you can check my collection on [Java interview questions](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) and  [multi-threading interview questions](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html). There are lot of resources in Javarevisited which can help you in your interview preparation. On the other hand if you are more interested on design pattern tutorials than you can check my post on [builder design pattern](http://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html) and [decorator pattern](http://javarevisited.blogspot.sg/2011/11/decorator-design-pattern-java-example.html).

10 interview question on Singleton Pattern in Java

Question starts with

**What is Singleton class? Have you used Singleton before?**  
Singleton is a class which has only one instance in whole application and provides a getInstance() method to access the singleton instance. There are many classes in JDK which is implemented using Singleton pattern like java.lang.Runtime which provides getRuntime() method to get access of it and used to get [free memory and total memory in Java](http://javarevisited.blogspot.sg/2012/01/find-max-free-total-memory-in-java.html).  
  
**1) Which classes are candidates of Singleton? Which kind of class do you make Singleton in Java?**Here they will check whether candidate has enough experience on usage of singleton or not. Does he is familiar of advantage/disadvantage or alternatives available for singleton in Java or not.

**Answer:** Any class which you want to be available to whole application and whole only one instance is viable is candidate of becoming Singleton. One example of this is Runtime class , since on whole java application only one runtime environment can be possible making Runtime Singleton is right decision. Another example is a utility classes like Popup in GUI application, if you want to show popup with message you can have one PopUp class on whole GUI application and anytime just get its instance, and call show() with message.   
  
  
**2) Can you write code for getInstance() method of a Singleton class in Java?**Most of the java programmer fail here if they have mugged up the singleton code because you can ask lots of follow-up question based upon the code they have written. I have seen many programmer write Singleton getInstance() method with double checked locking but they are not really familiar with the caveat associated with double checking of singleton prior to Java 5.

**Answer:** Until asked don’t write code using double checked locking as it is more complex and chances of errors are more but if you have deep knowledge of double checked locking, [volatile variable](http://javarevisited.blogspot.sg/2011/06/volatile-keyword-java-example-tutorial.html) and lazy loading than this is your chance to shine. I have shared code examples of writing singleton classes using enum, using static factory and with double checked locking in my recent post [Why Enum Singletons are better in Java](http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html), please see there.

**3) Is it better to make whole getInstance() method synchronized or just critical section is enough? Which one you will prefer?**This is really nice question and I mostly asked to just quickly check whether candidate is aware of performance trade off of unnecessary locking or not. Since locking only make sense when we need to create instance and rest of the time its just read only access so locking of critical section is always better option. read more about synchronization on [How Synchronization works in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html)  
**Answer:** This is again related to double checked locking pattern, well synchronization is costly and when you apply this on whole method than call to getInstance() will be synchronized and contented. Since synchronization is only needed during initialization on singleton instance, to prevent creating another instance of Singleton,  It’s better to only synchronize critical section and not whole method. Singleton pattern is also closely related to [factory design pattern](http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html) where getInstance() serves as static factory method.  
  
**4) What is lazy and early loading of Singleton and how will you implement it?**This is another great Singleton interview question in terms of understanding of concept of loading and cost associated with class loading in Java. Many of which I have interviewed not really familiar with this but its good to know concept.

**Answer:** As there are many ways to implement Singleton like using **double checked locking** or Singleton class with [static](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html) [final](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) instance initialized during class loading. Former is called lazy loading because Singleton instance is created only when client calls getInstance() method while later is called early loading because Singleton instance is created when class is loaded into memory.  
  
  
**5) Example of Singleton in standard Java Development Kit?**This is open question to all, please share which classes are Singleton in JDK. Answer to this question is java.lang.Runtime  
**Answer:** There are many classes in Java Development Kit which is written using singleton pattern, here are few of them:

* Java.lang.Runtime with getRuntime() method
* Java.awt.Toolkit with getDefaultToolkit()
* Java.awt.Desktop with  getDesktop()

**6) What is double checked locking in Singleton?**One of the most hyped question on Singleton pattern and really demands complete understanding to get it right because of Java Memory model caveat prior to Java 5. If a guy comes up with a solution of using [volatile keyword](http://javarevisited.blogspot.sg/2012/03/difference-between-transient-and.html) with Singleton instance and explains it then it really shows it has in depth knowledge of Java memory model and he is constantly updating his Java knowledge.

**Answer**: Double checked locking is a technique to prevent creating another instance of Singleton when call to getInstance() method is made in multi-threading environment. In Double checked locking pattern as shown in below example, singleton instance is checked two times before initialization.

**public** static Singleton getInstance(){  
     **if**(\_INSTANCE == **null**){  
         **synchronized**(Singleton.**class**){  
         *//double checked locking - because second check of Singleton instance with lock*  
                **if**(\_INSTANCE == **null**){  
                    \_INSTANCE = **new** Singleton();  
                }  
            }  
         }  
     **return** \_INSTANCE;  
}

Double checked locking should only be used when you have requirement for lazy initialization otherwise [use Enum to implement singleton](http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html) or simple static final variable.  
  
**7) How do you prevent for creating another instance of Singleton using clone() method?**This type of questions generally comes some time by asking how to break singleton or when Singleton is not Singleton in Java.  
**Answer:** Preferred way is not to implement Clonnable interface as why should one wants to create clone() of Singleton and if you do just throw Exception from clone() method as  “Can not create clone of Singleton class”.  
  
**8) How do you prevent for creating another instance of Singleton using reflection?**Open to all. In my opinion throwing exception from constructor is an option.   
**Answer:** This is similar to previous interview question. Since constructor of Singleton class is supposed to be [private](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html) it prevents creating instance of Singleton from outside but [Reflection can access private fields and methods](http://javarevisited.blogspot.sg/2012/05/how-to-access-private-field-and-method.html), which opens a threat of another instance. This can be avoided by throwing Exception from constructor as “Singleton already initialized”

**9) How do you prevent for creating another instance of Singleton during serialization?**Another great question which requires knowledge of [Serialization in Java](http://javarevisited.blogspot.com/2011/04/top-10-java-serialization-interview.html) and how to use it for persisting Singleton classes. This is open to you all but in my opinion use of readResolve() method can sort this out for you.  
Answer: You can prevent this by using readResolve() method, since during serialization readObject() is used to create instance and it return new instance every time but by using readResolve you can replace it with original Singleton instance.  I have shared code on how to do it in my post [Enum as Singleton in Java](http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html). This is also one of the reason I have said that use Enum to create Singleton because serialization of enum is taken care by JVM and it provides guaranteed of that.  
  
**10) When is Singleton not a Singleton in Java?**There is a very good article present in Sun's Java site which discusses various scenarios when a Singleton is not really remains Singleton and multiple instance of Singleton is possible. Here is the link of that article <http://java.sun.com/developer/technicalArticles/Programming/singletons/>

Apart from these questions on Singleton pattern, some of my reader contribute few more questions, which I included here. Thank you guys for your contribution.

**11) Why you should avoid the singleton anti-pattern at all and replace it with DI?**

Answer: Singleton Dependency Injection: every class that needs access to a singleton gets the object through its constructors or with a DI-container.

Singleton Anti-Pattern: with more and more classes calling getInstance the code gets more and more tightly coupled, monolithic, not testable and hard to change and hard to reuse because of not configurable, hidden dependencies. Also, there would be no need for this clumsy double checked locking if you call getInstance less often (i.e. once).

**12) How many ways you can write Singleton Class in Java?**  
Answer:  I know at least four ways to implement Singleton pattern in Java  
1) Singleton by synchronizing getInstance() method  
2) Singleton with public static final field initialized during class loading.  
3) Singleton generated by static nested class, also referred as Singleton holder pattern.  
4) From Java 5 on-wards using Enums

**13) How to write thread-safe Singleton in Java?**

Answer: Thread safe Singleton usually refers to write [thread safe code](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html) which creates one and only one instance of Singleton if called by multiple thread at same time. There are many ways to achieve this like by using double checked locking technique as shown above and by using [Enum](http://javarevisited.blogspot.in/2011/08/enum-in-java-example-tutorial.html) or Singleton initialized by class loader.

At last few more questions for your practice, contributed by Mansi:

14) Singleton vs Static Class?  
15) When to choose Singleton over Static Class?  
16) Can you replace Singleton with Static Class in Java?  
17) Difference between Singleton and Static Class in java?  
18) Advantage of Singleton over Static Class?

I have covered answers of couple of these questions in my post, [Singleton vs Static Class in Java - Pros and Cons](http://javarevisited.blogspot.com/2013/03/difference-between-singleton-pattern-vs-static-class-java.html).   
  
If you like to read more Java interview questions you can have a look on some of my favorites below:

Read more: <http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html#ixzz2gmGNQFQP>