==========================Singleton Class===========

**Java Singleton Pattern**

To implement Singleton pattern, we have different approaches but all of them have following common concepts.

* Private constructor to restrict instantiation of the class from other classes.
* Private static variable of the same class that is the only instance of the class.
* Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

|  |  |
| --- | --- |
| Eager initialization In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.  Note: while writing code only write final var | Static block initialization [Static block](https://www.journaldev.com/1365/static-keyword-in-java) initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for [exception handling](https://www.journaldev.com/1696/exception-handling-in-java). |
| package com.journaldev.singleton;  public class EagerInitializedSingleton {  private static final EagerInitializedSingleton instance = new EagerInitializedSingleton();  //private constructor to avoid client applications to use constructor  private EagerInitializedSingleton(){}  public static EagerInitializedSingleton getInstance(){  return instance;    }  } | package com.journaldev.singleton;  public class StaticBlockSingleton {  private static StaticBlockSingleton instance;  private StaticBlockSingleton(){}  //static block initialization for exception handling  static{  try{  instance = new StaticBlockSingleton();  }catch(Exception e){  throw new RuntimeException("Exception occured in creating singleton instance");  }  }    public static StaticBlockSingleton getInstance(){  return instance;  }  } |

### 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). Which classes are candidates of Singleton? Which kind of class do you make Singleton in Java?

Here they 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.

**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.**

|  |
| --- |
| Lazy Initialization [**Lazy Initialization**](http://en.wikipedia.org/wiki/Lazy_initialization) is a technique where one postpones the instantiation of an object until its first use. In other words the instance of a class is created when its required to be used for the first time. The idea behind this is to avoid unnecessary instance creation. |
| public class LazyInitializedSingleton {  private static LazyInitializedSingleton instance;    private LazyInitializedSingleton(){}    public static LazyInitializedSingleton getInstance(){  if(instance == null){  instance = new LazyInitializedSingleton();  }  return instance;  }  } |

### Give me some examples of Singleton pattern from Java Development Kit?

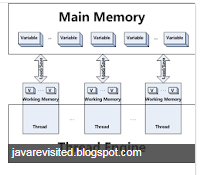
This is open question to all, please share which classes are Singleton in JDK. Answer to this question is java.lang.Runtime

Answer : There are many classes in Java Development Kit which is written using singleton pattern, here are few of them:

1. Java.lang.Runtime with getRuntime() method
2. Java.awt.Toolkit with getDefaultToolkit()
3. Java.awt.Desktop with getDesktop()

### 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.

Declaring a **volatile Java variable** means: The value of this **variable** will never be cached thread-locally: all reads and writes will go straight to "main memory"; Access to the **variable** acts as though it is enclosed in a synchronized block, synchronized on itself.  
  
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. See [here](http://javarevisited.blogspot.sg/2014/05/double-checked-locking-on-singleton-in-java.html) to learn more about double-checked-locking in Java.

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.

### 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 Cloneable 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”.

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

// **TODO** Auto-generated method stub

SingeltonCloneTest test1 = SingeltonCloneTest.*getInstance*();

**try** {

SingeltonCloneTest test2 = (SingeltonCloneTest) test1.clone();

} **catch** (CloneNotSupportedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

java.lang.CloneNotSupportedException: com.mindtree.singleton.pattern.SingeltonCloneTest

at java.lang.Object.clone(Native Method)

at com.mindtree.singleton.pattern.SingeltonCloneTest.main(SingeltonCloneTest.java:38)

## java.lang.Object[[edit](https://en.wikibooks.org/w/index.php?title=Java_Programming/API/java.lang.Object&action=edit&section=1" \o "Edit section: java.lang.Object)]

Object class is the superclass of all Java classes. All Java classes inherited from this class. This makes it possible that we can have methods that are available in all Java classes. This simplifies things compared to C++ where this is not the case.

|  |  |
| --- | --- |
| **Object class methods** | **Description** |
| [**boolean**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/boolean) equals( Object o ); | Gives generic way to compare objects |
| [Class](https://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Class) getClass(); | The [Class](https://en.wikibooks.org/wiki/Java_Programming/API/java.lang.Class) class gives us more information about the object |
| [**int**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/int) hashCode(); | Returns a hash value that is used to search objects in a collection |
| [**void**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/void) notify(); | Used in synchronizing threads |
| [**void**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/void) notifyAll(); | Used in synchronizing threads |
| [String](https://en.wikibooks.org/wiki/Java_Programming/API/java.lang.String) toString(); | Can be used to convert the object to String |
| [**void**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/void) wait(); | Used in synchronizing threads |
| [**protected**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/protected) Object clone() throws CloneNotSupportedException ; | Return a new object that are exactly the same as the current object |
| [**protected**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/protected) [**void**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/void) finalize() throws Throwable; | This method is called just before an object is garbage collected |

|  |
| --- |
| Thread Safe Singleton The easier way to create a thread-safe singleton class is to make the global access method [synchronized](https://www.journaldev.com/1061/thread-safety-in-java), so that only one thread can execute this method at a time. General implementation of this approach is like the below class.  private volatile static Singleton \_instance;  *public* *static* *Singleton* getInstanceDC() {  if (\_instance == **null**) { *// Single Checked*  *synchronized* (*Singleton*.class) {  if (\_instance == **null**) { *// Double checked*  \_instance = new *Singleton*();  }  }  }  return \_instance;  }  On surface this method looks perfect, as you only need to pay price for synchronized block one time, but it still broken, until you make \_instance variable [volatile](http://javarevisited.blogspot.sg/2011/06/volatile-keyword-java-example-tutorial.html).  Read more: <http://javarevisited.blogspot.com/2014/05/double-checked-locking-on-singleton-in-java.html#ixzz4w42Ct9Ca>  Without volatile modifier it's possible for another thread in Java to see half initialized state of \_instance variable, but with volatile variable guaranteeing happens-before relationship, all the write will happen on volatile \_instance before any read of \_instance variable.  **"Check and Act" race condition pattern**  Read more: <http://javarevisited.blogspot.com/2014/05/double-checked-locking-on-singleton-in-java.html#ixzz4w41saD3n> |
| public class ThreadSafeSingleton {  private static ThreadSafeSingleton instance;    private ThreadSafeSingleton(){}    public static synchronized ThreadSafeSingleton getInstance(){  if(instance == null){  instance = new ThreadSafeSingleton();  }  return instance;  }    }  Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances (Read: [Java Synchronization](https://www.journaldev.com/1061/thread-safety-in-java)). To avoid this extra overhead every time, **double checked locking** principle is used. In this approach, the synchronized block is used inside the if condition with an additional check to ensure that only one instance of singleton class is created.  Below code snippet provides the double checked locking implementation.  public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){  if(instance == null){  synchronized (ThreadSafeSingleton.class) {  if(instance == null){  instance = new ThreadSafeSingleton();  }  }  }  return instance;  } |

### Using Reflection to destroy Singleton Pattern

Reflection can be used to destroy all the above singleton implementation approaches. Let’s see this with an example class.

package com.journaldev.singleton;

import java.lang.reflect.Constructor;

public class ReflectionSingletonTest {

public static void main(String[] args) {

EagerInitializedSingleton instanceOne = EagerInitializedSingleton.getInstance();

EagerInitializedSingleton instanceTwo = null;

try {

Constructor[] constructors = EagerInitializedSingleton.class.getDeclaredConstructors();

for (Constructor constructor : constructors) {

//Below code will destroy the singleton pattern

constructor.setAccessible(true);

instanceTwo = (EagerInitializedSingleton) constructor.newInstance();

break;

}

} catch (Exception e) {

e.printStackTrace();

}

System.out.println(instanceOne.hashCode());

System.out.println(instanceTwo.hashCode());

}

}

When you run the above test class, you will notice that hashCode of both the instances are not same that destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate, do check out [**Java Reflection Tutorial**](https://www.journaldev.com/1789/java-reflection-example-tutorial).

### Enum Singleton

To overcome this situation with Reflection, Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program. Since [Java Enum](https://www.journaldev.com/716/java-enum) values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible; for example, it does not allow lazy initialization.

package com.journaldev.singleton;

public enum EnumSingleton {

INSTANCE;

public static void doSomething(){

//do something

}

}

## Making Singletons With Enum in Java

All of the above problems can be solved very easily by using the enum type to make singletons.

### Singleton With Enum

public enum Singleton {

INSTANCE;

}

The three lines above make a singleton without any of the problems discussed. Since enums are inherently serializable, we don't need to implement it with a serializable interface. The reflection problem is also not there. Therefore, it is 100% guaranteed that only one instance of the singleton is present within a JVM. Thus, this method is recommended as the **best method of making singletons in Java.**

### How to Use

public enum SingletonEnum {

INSTANCE;

int value;

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

}

Main class implementation:

public class EnumDemo {

public static void main(String[] args) {

SingletonEnum singleton = SingletonEnum.INSTANCE;

System.out.println(singleton.getValue());

singleton.setValue(2);

System.out.println(singleton.getValue());

}

}