**Singleton Design Pattern**

Singleton pattern is a design solution where an application wants to have one and only one instance of any class, in all possible scenarios without any exceptional condition. It has been debated long enough in java community regarding possible approaches to make any class singleton. Still, you will find people not satisfied with any solution you give. They can not be overruled either. In this post, we will discuss some good approaches and will work towards our best possible effort.

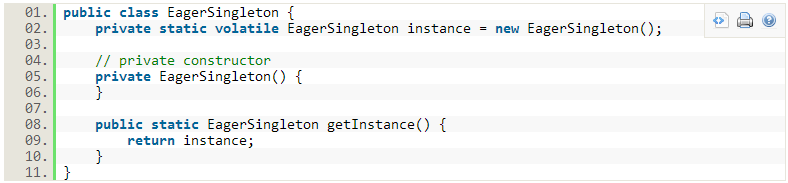
**Sections in this post:**

* Eager initialization
* Lazy initialization
* Static block initialization
* Bill pugh solution
* Using Enum
* Adding readResolve()
* Adding serial version id
* Conclusion

Singleton term is derived from its [mathematical counterpart](http://en.wikipedia.org/wiki/Singleton_%28mathematics%29). It wants us, as said above, to have only one instance. Lets see the possible solutions:

### Eager initialization

This is a design pattern where an instance of a class is created much before it is actually required. Mostly it is done on system start up. In singleton pattern, it refers to create the singleton instance irrespective of whether any other class actually asked for its instance or not.

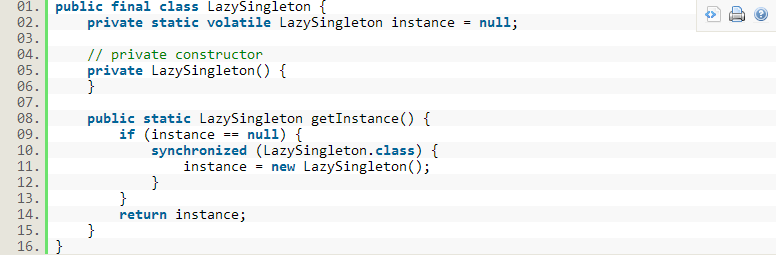
Above method works fine, but has one drawback. Instance is created irrespective of it is required in runtime or not. If this instance is not big object and you can live with it being unused, this is best approach.

Lets solve above problem in next method.

### Lazy initialization

In computer programming, [lazy initialization](http://en.wikipedia.org/wiki/Lazy_initialization) is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. In singleton pattern, it restricts the creation of instance until requested first time. Lets see in code:

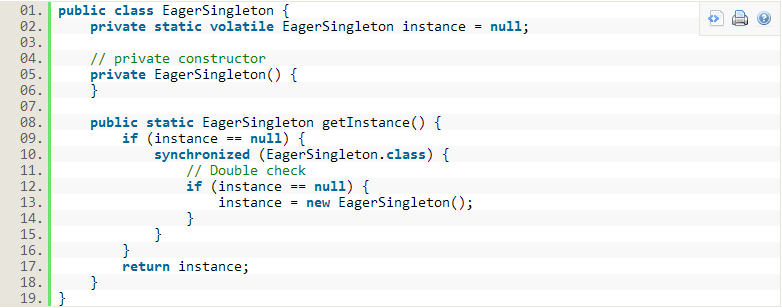
[view source](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#viewSource)[print](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#printSource)[?](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#about)

On first invocation, above method will check if instance is already created using instance variable. If there is no instance i.e. instance is null, it will create an instance and will return its reference. If instance is already created, it will simply return the reference of instance.

But, this method also has its own drawbacks. Lets see how. Suppose there are two threads T1 and T2. Both comes to create instance and execute “instance==null”, now both threads have identified instance variable to null thus assume they must create an instance. They sequentially goes to synchronized block and create the instances. At the end, we have two instances in our application.

This error can be solved using [double-checked locking](http://en.wikipedia.org/wiki/Double_checked_locking_pattern#Usage_in_Java). This principle tells us to recheck the instance variable again in synchronized block in given below way:

[view source](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#viewSource)[print](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#printSource)[?](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#about)

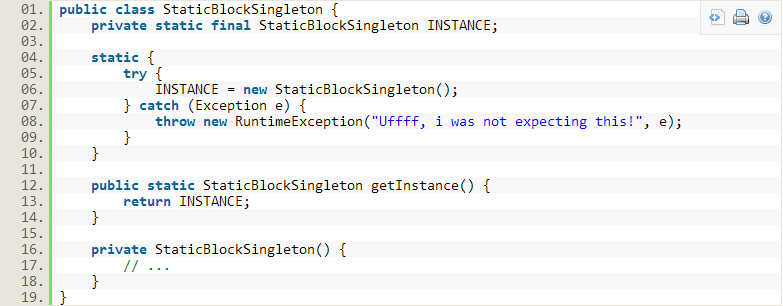


Above code is the correct implementation of singleton pattern.

Please ensure to use “[**volatile**](http://en.wikipedia.org/wiki/Volatile_variable#In_Java)” keyword with instance variable otherwise you can run into out of order write error scenario, where reference of instance is returned before actually the object is constructed i.e. JVM has only allocated the memory and constructor code is still not executed. In this case, your other thread, which refer to uninitialized object may throw null pointer exception and can even crash the whole application.

### Static block initialization

If you have little idea about class loading sequence, you can connect to the fact that static blocks are executed during the loading of class and even before the constructor is called. We can use this feature in our singleton pattern also like this:

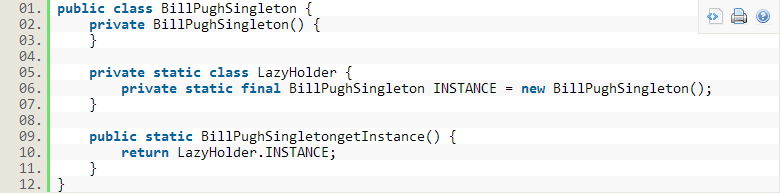
Above code has one drawback. Suppose there are 5 static fields in class and application code needs to access only 2 or 3, for which instance creation is not required at all. So, if we use this static initialization. we will have one instance created though we require it or not.

Next section will overcome this problem.

### Bill pugh solution

Bill pugh was main force behind [java memory model](http://en.wikipedia.org/wiki/Java_Memory_Model) changes. His principle “[Initialization-on-demand holder idiom](http://en.wikipedia.org/wiki/Initialization_on_demand_holder_idiom)” also uses static block but in different way. It suggest to use static inner class.

[view source](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#viewSource)[print](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#printSource)[?](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#about)

As you can see, until we need an instance, the LazyHolder class will not be initialized until required and you can still use other static members of BillPughSingleton class. ***This is the solution, i will recommend to use. I also use it in my all projects.***

### Using Enum

This type of implementation recommend the use of enum. [Enum](http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html" \o "enum in java), as written in java docs, provide implicit support for thread safety and only one instance is guaranteed. This is also a good way to have singleton with minimum effort.

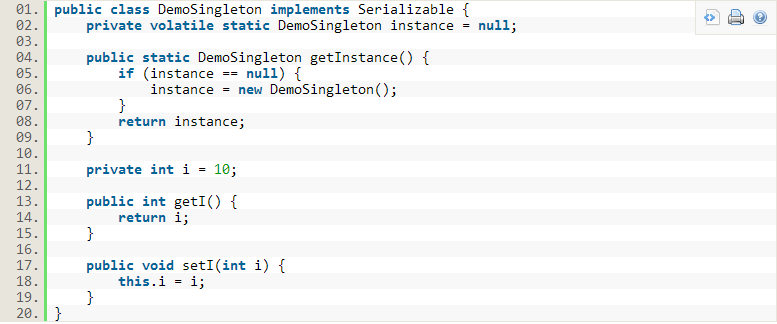
[view source](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#viewSource)[print](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#printSource)[?](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#about)

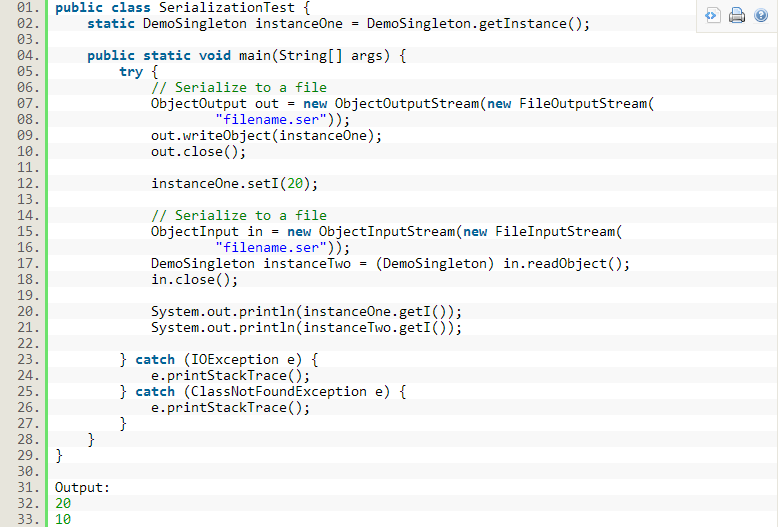
### Adding readResolve()

So, till now you must have taken your decision that how you would like to implement your singleton. Now let’s see other problems that may arise even in interviews also.  
Let’s say your application is distributed and it frequently serializes the objects in file system, only to read them later when required. Please note that, de-serialization always creates a new instance. Let’s understand using an example:

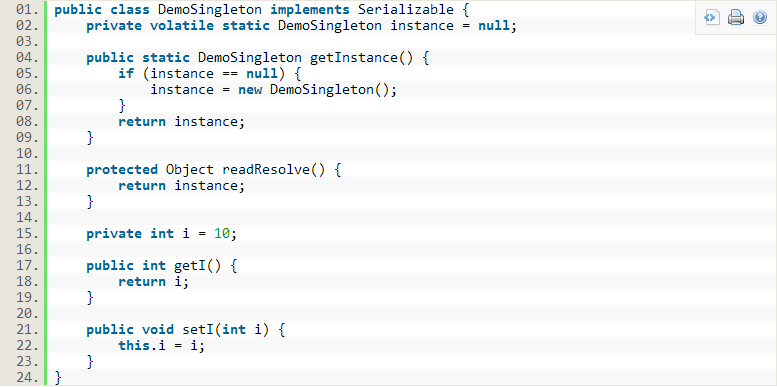
Our singleton class is:

[view source](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#viewSource)[print](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#printSource)[?](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#about)

Let’s serialize this class and de-serialize it after making some changes:

Unfortunately, both variables have different value of variable “i”. Clearly, there are two instances of our class. So, again we are in same problem of multiple instances in application.  
To solve this issue, we need to include [readResolve()](http://docs.oracle.com/javase/1.4.2/docs/api/java/io/ObjectInputStream.html" \o "read resolve) method in our DemoSingleton class. This method will be invoked when you will de-serialize the object. Inside this method, you must return the existing instance to ensure single instance application wide.

[view source](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#viewSource)[print](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#printSource)[?](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#about)

Now when you execute the class SerializationTest, it will give you correct output.

[view source](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#viewSource)[print](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#printSource)[?](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#about)

1.20

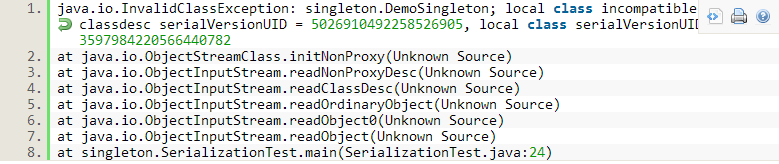
2.20

### Adding serial version id

So far so good. Till now, we have solved the problem of synchronization and serialization both. Now, we are just one step behind our correct and complete implementation. And missing part is serial version id.

This is required in condition when you class structure can change in between you serialize the instance and go again to de-serialize it. Changed structure of class will cause JVM to give exception while de-serializing process.

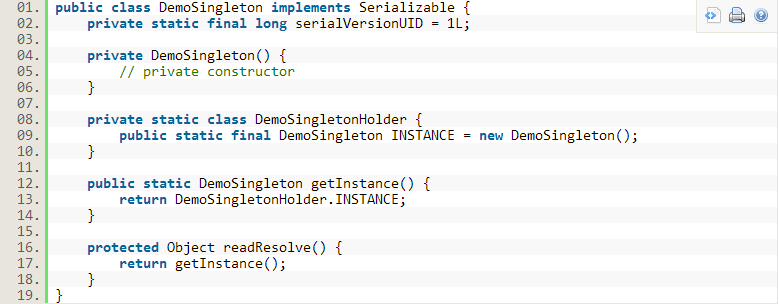
[view source](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#viewSource)[print](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#printSource)[?](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#about)

This problem can be solved only by adding a unique serial version id to class. It will prevent the compiler to throw the exception by telling that both classes are same, and will load the available instance variables only.

### Conclusion

After having discussed so many possible approaches and other possible error cases, i will recommend you below code template to design your singleton class which shall ensure only one instance of class in whole application in all above discussed scenarios.

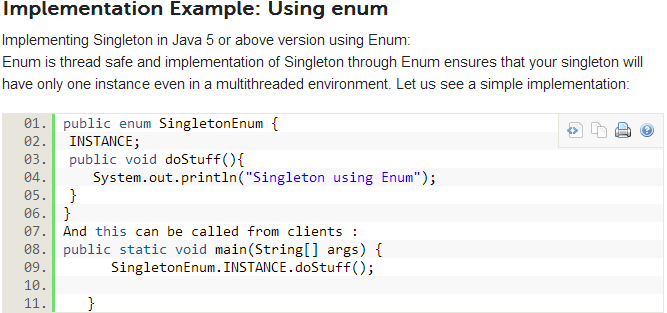
[view source](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#viewSource)[print](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#printSource)[?](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/#about)

I hope, this post has enough information to make you understand the most common approaches for singleton pattern. Let me know of your thoughts please.

Happy Learning!!

**Update:**I just thought to add some examples which can be referred for further study and mention in interviews:

* [java.awt.Desktop#getDesktop()](http://docs.oracle.com/javase/6/docs/api/java/awt/Desktop.html#getDesktop%28%29)
* [java.lang.Runtime#getRuntime()](http://docs.oracle.com/javase/6/docs/api/java/lang/Runtime.html#getRuntime%28%29)



**Question: Why can’t we use a static class instead of singleton?**Answer:

* One of the key advantages of singleton over static class is that it can implement interfaces and extend classes while the static class cannot (it can extend classes, but it does not inherit their instance members). If we consider a static class it can only be a nested static class as top level class cannot be a static class. Static means that it belongs to a class it is in and not to any instance. So it cannot be a top level class.
* Another difference is that static class will have all its member as static only unlike Singleton.
* Another advantage of Singleton is that it can be lazily loaded whereas static will be initialized whenever it is first loaded.
* Singleton object stores in Heap but, static object stores in stack.
* We can clone the object of Singleton but, we can not clone the static class object.
* Singleton can use the Object Oriented feature of polymorphism but static class cannot.

#### Uses of Singleton design pattern:

Various usages of Singleton Patterns:

* Hardware interface access: The use of singleton depends on the requirements. However practically singleton can be used in case external hardware resource usage limitation required e.g. Hardware printers where the print spooler can be made a singleton to avoid multiple concurrent accesses and creating deadlock.
* Logger : Similarly singleton is a good potential candidate for using in the log files generation. Imagine an application where the logging utility has to produce one log file based on the messages received from the users. If there is multiple client application using this logging utility class they might create multiple instances of this class and it can potentially cause issues during concurrent access to the same logger file. We can use the logger utility class as a singleton and provide a global point of reference.
* Configuration File: This is another potential candidate for Singleton pattern because this has a performance benefit as it prevents multiple users to repeatedly access and read the configuration file or properties file. It creates a single instance of the configuration file which can be accessed by multiple calls concurrently as it will provide static config data loaded into in-memory objects. The application only reads from the configuration file at the first time and there after from second call onwards the client applications read the data from in-memory objects.
* Cache: We can use the cache as a singleton object as it can have a global point of reference and for all future calls to the cache object the client application will use the in-memory object.

**Question**: **What is the impact if we are creating another instance of singleton using serialization and deserialization?**

**Answer**: When we serialize a class and deserialize it then it creates another instance of the singleton class. Basically as many times as you deserialize the singleton instance it will create multiple instance. Well in this case the best way is to make the singleton as enum. In that way the underlying Java implementation takes care of all the details. If this is not possible then we will need to override the*readobject()* method to return the same singleton instance.

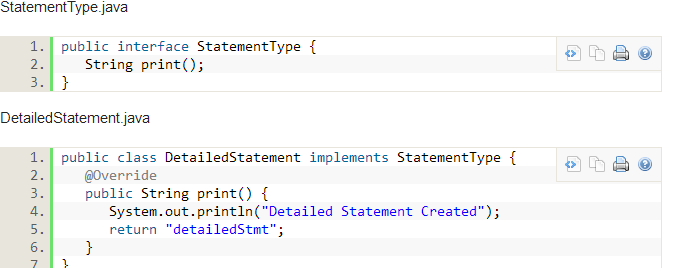
**Problem Statement:**  
Design a small ATM printing application which can generate multiple types of statements of the transaction including Mini Statement, Detailed statement etc. However the customer should be aware of the creation of these statements. Ensure that the memory consumption is minimized.  
  
**Design Solution:**  
The above requirement can be addressed using two core Gang of four design pattern – Factory design pattern and Singleton design pattern. In order to generate multiple types of statements for the ATM transactions in the ATM machine we can create a Statement Factory object which will have a factory method of *createStatements()*. The createStatements will create DetailedStatement or MiniStatement objects.   
  
The client object will be completely unware of the object creation since it will interact with the Factory interface only. We will also create an interface called StatementType. This will allow further statement type objects e.g. Credit card statement etc to be added. So the solution is scalable and extensible following the object oriented Open/Closed design principle.  
  
The second requirement of reducing the memory consumption can be achieved by using Singleton design pattern. The Statement Factory class need not be initiated multiple times and a single factory can create multiple statement objects. Singleton pattern will create a single instance of the StatementFactory class thus saving memory.

ATM example

* Factory: Factory is an abstract class which is a single point of contact for the client. All the concrete factory classes needs to implement the abstract factory method.
* StatementFactory: This is the Factory implementation class which consist of the creator method. This class extends from the Factory abstract class.This is the main creator class for all the products e.g. Statements.
* StatementType: This is a product interface which provides abstraction to the various types of products which needs to be created by the Factory class.
* DetailedStatement: This is a concrete implementation of the StatementType interface which will print the detailed statements.
* MiniStatement: This is a concrete implementation of the StatementType interface which will print the mini statements.
* Client: This is the client class which will call the StatementFactory and StatementType and request for object creation.

**Assumption:**  
The design solution caters to only single ATM machine.







**Conclusion:**

In the above articles we have gone into the details of the Singleton pattern, how to implement singleton pattern along with a practical application. Though singleton pattern looks a simple implementation we should resist ourselves from using it until and unless there is a strong requirement for it. You can blame the unpredictable nature of the results in the multi-threading environment. Though we can use *enum* in Java 5 and above it sometimes get difficult to implement your logic always in *enum* or there might be legacy code before Java 5. Hope our readers have liked this article.

**Singleton Pattern vs. Static Class in Java**

Singleton pattern and static class (a class, having all static methods) provides good accessibility, and they share some similarities e.g. both can be used without creating object and both provide only one instance, at very high level it looks that they both are intended for same task.

It’s important to remember fundamental difference between Singleton pattern and static class, one represent object while other represent a method. Since an object is always much more capable than a method, it can guide you when to use Singleton pattern vs. static methods. 

In this Java article we will learn, where to use Singleton pattern in Java, and when static class is better alternative. By the way, JDK has examples of both singleton and static, and that too very intelligently e.g. java.lang.Math is a [final class](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html) with full of [static methods](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html), on the other hand java.lang.Runtime is a Singleton class in Java. For those who are not familiar with Singleton design pattern or static class,  static class is a [Java class](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html), which only contains static methods, good examples of static class is java.lang.Math,which contains lots of utility methods for various maths function e.g. sqrt(). While [Singleton classes](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) are those, which has only one instance during application life cycle like java.lang.Runtime.  
  
Prime example of this is java.lang.Math which is not Singleton, instead a class with all static methods. Here are few situation where I think using static class over Singleton pattern make sense:

1) If your Singleton is not maintaining any state, and just providing global access to methods, than consider using static class, as static methods are much faster than Singleton, because of [static binding](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html) during compile time. But remember its not advised to maintain state inside static class, especially in concurrent environment, where it could lead subtle [race conditions](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html) when modified parallel by multiple threads without adequate synchronization.

You can also choose to use static method, if you need to combine bunch of utility method together. Anything else, which requires singles access to some resource, should use Singleton design pattern.

## Singleton vs Static in Java:

1) Static class provides better performance than Singleton pattern, because static methods are bonded on compile time.

2) One more difference between Singleton and static is, ability to override. Since [static methods in Java cannot be overridden](http://java67.blogspot.com/2012/08/can-we-override-static-method-in-java.html), they leads to inflexibility. On the other hand, you can override methods defined in Singleton class by extending it.

3) Static classes are hard to mock and consequently hard to test than Singletons, which are pretty easy to mock and thus easy to test. It’s easier to write [JUnit test](http://javarevisited.blogspot.com/2013/03/how-to-write-unit-test-in-java-eclipse-netbeans-example-run.html) for Singleton than static classes, because you can pass mock object whenever Singleton is expected, e.g. into constructor or as method arguments.

4) If your requirements needs to maintain state than Singleton pattern is better choice than static class, because maintaining  state in later case is nightmare and leads to subtle bugs.

5) Singleton classes can be [lazy loaded](http://javarevisited.blogspot.sg/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) if it’s an heavy object, but static class doesn't have such advantages and always eagerly loaded.

6) Many [Dependency Injection framework](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html) manages Singleton quite well e.g. Spring, which makes using them very easy.

## Advantage of Singleton Pattern over Static Class in Java:

Main advantage of Singleton over static is that former is more object oriented than later. With Singleton, you can use [Inheritance](http://javarevisited.blogspot.com/2012/10/what-is-inheritance-in-java-and-oops-programming.html) and [Polymorphism](http://javarevisited.blogspot.com.au/2011/08/what-is-polymorphism-in-java-example.html) to extend a base class, implement an interface and capable of providing different implementations. If we talk about java.lang.Runtime, which is a Singleton in Java, call to getRuntime() method return different implementations based on different JVM, but guarantees only one instance per JVM, had java.lang.Runtime an static class, it’s not possible to return different implementation for different JVM.

That’s all on difference between Singleton and static class in Java. When you need a class with full OO capability, chose Singleton, while if you just need to store bunch of static methods together, than use static class.