**10. What is Singleton? is it better to make whole method synchronized or only critical section synchronized ?**

Singleton in Java is a class with just one instance in whole Java application, for example java.lang.Runtime is a Singleton class. Creating Singleton was tricky prior Java 4 but once Java 5 introduced Enum its very easy. see my article [How to create thread-safe Singleton in Java](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) for more details on writing Singleton using enum and double checked locking which is purpose of this Java interview question.

Read more: <http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html#ixzz3KTsCklfG>

[**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?

Read more: <http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html#ixzz3KTsJZ13v>

# [Difference between Singleton Pattern vs Static Class in Java](http://javarevisited.blogspot.com/2013/03/difference-between-singleton-pattern-vs-static-class-java.html)

Singleton pattern  vs  Static Class (a class, having all static methods) is another interesting questions, which I missed while blogging about [Interview questions on Singleton pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html). Since both Singleton pattern and static class 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. Because of high level similarities, interviewer normally ask questions like, *Why you use Singleton instead of Static Methods,* or Can you replace Singleton with static class, and  what are differences between [Singleton pattern](http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html) and [static in Java](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html). In order to answer these question, it’s important to remember fundamental difference between Singleton pattern and static class, former gives you an [Object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html), while later just provide static methods. 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.

## When to use Static Class in place of Singleton in Java

[When to choose Singleton pattern vs static in Java](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)Indeed there are some situations, where static classes makes sense than Singleton. 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.

## Difference between Singleton vs Static in Java

This is answer of our second interview question about Singleton over static. As I said earlier, fundamental difference between them is, one represent object while other represent a method. Here are few more differences between static and singleton 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 its 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.

These are some differences between static class and singleton pattern, this will help to decide between two, which situation arises. In next section we will when to choose Singleton pattern over static class in Java.

## 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.

Read more: <http://javarevisited.blogspot.com/2013/03/difference-between-singleton-pattern-vs-static-class-java.html#ixzz3KTsOTo5M>

# [Why Enum Singleton are better in Java](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html)

**Enum Singletons** are new way to implement Singleton pattern in Java by using Enum with just one instance. Though Singleton pattern in Java exists from long time Enum Singletons are relatively new concept and in practice from Java 5 onwards after introduction of Enum as keyword and feature. This article is somewhat related to my earlier post on Singleton, [10 interview questions on Singleton pattern in Java](http://javarevisited.blogspot.sg/2011/03/10-interview-questions-on-singleton.html) where we have discussed common questions asked on interviews about Singleton pattern and [10 Java enum examples](http://javarevisited.blogspot.in/2011/08/enum-in-java-example-tutorial.html), where we have seen how versatile enum can be. This post is about **why should we use Enum as Singleton in Java**, What benefit it offers compared to conventional singleton methods etc.

## Java Enum and Singleton Pattern

[Enum Singleton pattern in Java](http://javarevisited.blogspot.sg/2011/11/collection-interview-questions-answers.html)Following are some reasons which make sense to me for using Enum to implement Singleton pattern in Java. By the way If you like articles on design pattern than you can also check my post on [Builder design pattern](http://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html) and [Decorator design pattern](http://javarevisited.blogspot.sg/2011/11/decorator-design-pattern-java-example.html) .

**1) Enum Singletons are easy to write**

This is by far biggest advantage, if you have been writing Singletons prior ot Java 5 than you know that even with double checked locking you can have more than one instances. though that issue is fixed with Java memory model improvement and gurantee provided by volatile variables from Java 5 onwards but it still tricky to write for many beginners. compared to double checked locking with synchronization Enum singletons are cake walk. If you don't believe than just compare below code for conventional singleton with double checked locking and Enum Singletons:

**Singleton using Enum in Java**

This is the way we generally declare Enum Singleton , it may contain instace variable and instance method but for sake of simplicity I haven’t used any, just beware that if you are using any instance method than you need to ensure thread-safety of that method if at all it affect the state of object. By default creation of Enum instance is thread safe but any other method on Enum is programmers responsibility.

***/\*\*  
\* Singleton pattern example using Java Enumj  
\*/***

**public** **enum** EasySingleton{  
    INSTANCE;  
}

You can acess it by EasySingleton.INSTANCE, much easier than calling getInstance() method on Singleton.

**Singleton example with double checked locking**

Below code is an example of double checked locking in Singleton pattern, here getInstance() method checks two times to see whether INSTANCE is null or not and that’s why it’s called double checked locking pattern, remember that double checked locking is broker before Java 5 but with the guranteed of [volatile variable in Java 5](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html) memory model, it should work perfectly.

***/\*\*  
\* Singleton pattern example with Double checked Locking  
\*/***

**public** **class** DoubleCheckedLockingSingleton{  
     **private** **volatile** DoubleCheckedLockingSingleton INSTANCE;  
    
     **private** DoubleCheckedLockingSingleton(){}  
    
     **public** DoubleCheckedLockingSingleton getInstance(){  
         **if**(INSTANCE == **null**){  
            **synchronized**(DoubleCheckedLockingSingleton.**class**){  
                *//double checking Singleton instance*  
                **if**(INSTANCE == **null**){  
                    INSTANCE = **new** DoubleCheckedLockingSingleton();  
                }  
            }  
         }  
         **return** INSTANCE;  
     }  
}

You can call DoubleCheckedLockingSingleton.getInstance() to get access of this Singleton class.

Now Just look at amount of code needed to create a **lazy loaded thread-safe Singleton**. With Enum Singleton pattern you can have that in one line because creation of Enum instance is [thread-safe](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html) and guranteed by JVM.

People may argue that there are better way to write Singleton instead of Double checked locking approach but every approach has there own advantages and disadvantages like I mostly prefer static field Singleton intialized during classloading as shwon in below example, but keep in mind that is not a **lazy loaded Singleton**:

**Singleton pattern with static factory method**

This is one of my favorite method to impelemnt Singleton pattern in Java, Since Singleton instance is [static](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html) and [final variable](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) it initialized when class is first loaded into memeory so creation of instance is inherently thread-safe.

***/\*\*  
\* Singleton pattern example with static factory method  
\*/***  
  
**public** **class** Singleton{  
    *//initailzed during class loading*  
    **private** **static** **final** Singleton INSTANCE = **new** Singleton();  
    
    *//to prevent creating another instance of Singleton*  
    **private** Singleton(){}  
  
    **public** **static** Singleton getSingleton(){  
        **return** INSTANCE;  
    }  
}

You can call Singleton.getSingleton() to get access of this class.

**2) Enum Singletons handled Serialization by themselves**

Another problem with conventional Singletons are that once you implement [serializable interface](http://javarevisited.blogspot.sg/2011/04/top-10-java-serialization-interview.html) they are no longer remain Singleton because readObject() method always return a new instance just like constructor in Java. you can avoid that by using readResolve() method and discarding newly created instance by replacing with Singeton as shwon in below example :

*//readResolve to prevent another instance of Singleton*  
    **private** Object readResolve(){  
        **return** INSTANCE;  
    }

This can become even more complex if your Singleton Class maintain state, as you need to make them [transient](http://javarevisited.blogspot.sg/2012/03/difference-between-transient-and.html), but witn **Enum Singleton**, Serialization is guarnateed by JVM.

**3) Creation of Enum instance is thread-safe**

As stated in point 1 since creatino of Enum instance is thread-safe by default you don't need to worry about double checked locking.

In summary, given the **Serialzation and thraead-safety guaranteed** and with couple of line of code enum Singleton pattern is best way to create Singleton in Java 5 world. you can still use other popular methods if you feel so but I still have to find a *convincing reason not to use Enum as Singleto*n, let me know if you got any.

Read more: <http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html#ixzz3KTsS7J3P>

# [Java Enum Tutorial: 10 Examples of Enum in Java](http://javarevisited.blogspot.in/2011/08/enum-in-java-example-tutorial.html)

**What is Enum in Java**  
Enum in Javais 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](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html), 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 type is more suitable to represent well known fixed set of things and state,  for example representing state of Order as NEW, PARTIAL FILL, FILL or CLOSED. Enumeration(Enum) was not originally available in Java though it was available in other 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 you have used Enumeration before in C or C++ than you will not be uncomfortable with Java Enum but in my opinion Enum in Java is more rich and versatile than in any other language. . By the way, if you like to learn new concepts using book than you can also see Java 5.0 Tiger : A Developers notebook, I had followed this book while learning Enum, when Java 1.5 was first launched. This book has excellent chapter not only on Enum but also on key features of Java 1.5 and  worth reading. 

## How to represent enumerable value without Java enum

[java enum example, enum in java tutorial](http://javarevisited.blogspot.com/2011/08/convert-string-to-integer-to-string.html)Since **Enum in Java** is only available from **Java 1.5** its 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](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) 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 server our purpose it has some serious limitations:  
  
**1) No Type-Safety**: First of all it’s not [type-safe](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html); you 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 you 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](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html)  
  
**Java Enum** is answer of all this limitation. Enum in Java is type-safe, provides meaningful String names and has there own namespace. Now let's see 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 are type like [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html)and[interface in Java](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html). Also we have followed 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.

## What is Enum in Java

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 type like class and interface and can be used to define a set of Enum constants. Enum constants are [implicitly static and final](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) and you can not change there value once created. Enum in Java provides type-safety and can be used inside switch statment like int variables. Since enum is a keyword you can not use as variable name and since its only introduced in JDK 1.5 all your previous code which has enum as variable name will not work and needs to be re-factored.

### Benefits of Enums in Java:

1) **Enum is type-safe** you can not assign anything else other than predefined Enum constants to an Enum variable. It is 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 name-space.  
  
3) Best feature of Enum is **you can use Enum in Java inside Switch statement** like int or char primitive data type.we will also see example of using java enum in switch statement in this java enum tutorial.  
  
4) Adding new constants on Enum in Java is easy and you can add new constants without breaking existing code.

## Important points about Enum in Java

1) **Enums in Java are type-safe** and has there own name-space. It means your enum will have a type for example "Currency" in below example and you 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 type**like [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html)or [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html)and you 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) You 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 you need to define a member variable and a constructor because PENNY (1) is actually [calling a constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) which 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;

        }

};     
**Constructor of enum in java** must be [**private**](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html)any other access modifier will result in compilation error. Now to get the value associated with each coin you can define a public getValue() method inside java enum like any normal java class. Also semi colon in the first line is optional.  
  
  
4) Enum constants are implicitly [static](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html)and [final](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html)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 re assigned.  
  
    
    
5) **Enum in java can be used as an argument on switch statment** 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 you can also [String in Switch case in Java](http://javarevisited.blogspot.sg/2011/08/string-switch-case-jdk7-example.html) code.  
  
6) Since **constants defined inside Enum in Java are final you can safely compare them using "==" 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](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) or [compareTo() method](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) to compare Objects.

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 you can use values() to [iterate](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html)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 its exactly same **with defined order in enums**.  
  
  
    
8) In Java Enum can override methods also. Let’s see an example of overriding toString() method **inside Enum in Java** to provide **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 high performance implementation of [Map and Set interface in Java](http://javarevisited.blogspot.sg/2012/07/create-read-only-list-map-set-example-java.html)and we should use this whenever there is any opportunity.  
  
  
  
10**) You can not create instance of enums by using new operator** in Java because constructor of Enum in Java can only be private and Enums constants can only be created inside Enums itself.  
  
  
11) 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 implement both [Serializable](http://javarevisited.blogspot.sg/2012/01/serializable-externalizable-in-java.html)and [Comparable](http://javarevisited.blogspot.sg/2011/06/comparator-and-comparable-in-java.html)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) **You can define abstract methods inside Enum in Java** and can also provide different implementation for different instances of enum in java.  Let’s see an *example of using*[*abstract method*](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html)*inside enum in java*

**public** **enum** Currency **implements** Runnable{

          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 different color we made the color() method abstract and let each instance of Enum to define   there own color. You can get color of any coin by just calling color() method as shown in below example of java enum:

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

    
**Enum Java valueOf example**  
One of my reader pointed out that I have not mention about valueOf method of enum in Java, which is used to convert String to enum in java.  Here is what he has suggested, thanks @ Anonymous  
“You 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 enum. One think though you 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 you 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 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](http://javarevisited.blogspot.sg/2011/12/convert-enum-string-java-example.html).  
  
That’s all on Java enum , Please share if you have any nice tips on enum in Java  and let us know how you are using java enum in your work. You can also follow some good advice for using Enum by Joshua Bloch in his all time classic book Effective Java. Those advice will give you more idea of using this powerful feature of Java programming language

**Further Reading on Java Enum**

If you like to learn more about this cool feature, I suggest reading following books. Books are one of the best resource to completely understand any topic and I personally follow them as well. Enumeration types chapter from Thinking in Java is particularly useful.

Thinking in Java (4th Edition) By Bruce Eckel

Effective Java by Joshua Bloch

Java 5.0 Tiger: A Developers notebook  
Java 7 Recipes  
    
Last book is suggested by one of our reader @Anonymous, you can see his comment

Read more: <http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html#ixzz3KTyKaMdj>