Contents

[1. jdk vs jre vs jvm 2](#_Toc510006046)

[2. Constructors 2](#_Toc510006047)

[3. Access Modifiers in Java 3](#_Toc510006048)

[4. Final Modifiers in Java 5](#_Toc510006049)

[Final modifiers are applicable to: 5](#_Toc510006050)

[5. Abstract Class 8](#_Toc510006051)

[6. Interface 9](#_Toc510006052)

[Why Java doesn't support multiple inheritance 10](#_Toc510006053)

[d. Overloading vs. Overriding 12](#_Toc510006054)

[7. Static Keyword 16](#_Toc510006055)

[8. Serialization 18](#_Toc510006056)

[9. Strings 22](#_Toc510006057)

[10. How to resolve java.lang.UnsupportedClassVersionError with example 27](#_Toc510006058)

[11. Does Java pass by value or pass by reference 30](#_Toc510006059)

[12. What is Enum in Java 32](#_Toc510006060)

# jdk vs jre vs jvm

**JDK (Java Development Kit)**

It contains everything that will be required to***develop and run*** any Java Application.

**JRE (Java Run time Environment)**

It contains everything required to ***run*** any Java Application which is already compiled. It doesn’t contain library which is required to develop java application.

**JVM (Java Virtual Machine)**

It is a virtual machine which works over your operating system to provide proper environment for your compiled Java code. JVM only works with bytecode. Hence you need to compile your java application (.java) so that it can be converted to bytecode format (.class file). This then will be used by JVM to run application. JVM only provide environment it needs other library to run application properly.

# Constructors

Constructors have only one purpose in life and that is to **create an Instance of a Class**. This instantiation includes memory allocation and member initialization (*Optional*).

* Constructor cannot have a return type.
* Constructor must have the same name as that of the class.
* Constructors cannot be marked static
* Constructor cannot be marked abstract
* Constructor cannot be overridden.
* Constructor cannot be Final.
* **Public constructor**: Objects can be created anywhere.
* **Default constructor**: Objects can be created only in the same package.
* **Protected constructor**: Objects can be created by classes outside the package only if it's a subclass.
* **Private constructor**: Object can only be created inside the class (e.g., when implementing a singleton).

The static, final and abstract keywords are not meaningful for a constructor because:

* Static members belong to a class, but the constructor is needed to create an object.
* An abstract class is a partially implemented class, which contains abstract methods to be implemented in child class.
* Final: Because you can't override/extend a constructor anyway. You can extend a class (to prevent that you make it final) or override a method (to prevent that you make it final), but there is nothing like this for constructors.
* A constructor **belongs** to the class in which it is declared. A sub class is a different class and must have its own constructor. So, constructors simply can't be overridden.

**Constructor Chaining:**

Every constructor calls its superclass constructor. **An implied super() is therefore included in each constructor which does not include either this() or an explicit super() call as its first statement.** The super() statement invokes a constructor of the super class.

The implicit super() can be replaced by an explicit super(). The super statement must be the first statement of the constructor. The explicit super allows parameter values to be passed to the constructor of its superclass and must have matching parameter types A super() call in the constructor of a subclass will result in the call of the relevant constructor from the superclass, based on the signature of the call. This is called constructor chaining.

The super() construct as with this() construct: if used, must occur as the first statement in a constructor, and it can only be used in a constructor declaration. This implies that this() and super() calls cannot both occur in the same constructor. Just as the this() construct leads to chaining of constructors in the same class, the super() construct leads to chaining of subclass constructors to superclass constructors. if a constructor has neither a this() nor a super() construct as its first statement, then a super() call to the default constructor in the superclass is inserted.

# Access Modifiers in Java

***Access Modifiers for Class***

Classes in java can use only public and default access modifiers.

* Public: When set to public, the given class will be accessible to all the classes available in Java world.
* Default: When set to default, the given class will be accessible to the classes which are defined in the same package.

|  |  |  |
| --- | --- | --- |
| **Visibility** | **Public** | **Default** |
| **Within Same Package** | Yes | Yes |
| **From Outside the Same Package** | Yes | No |

Private and protected are permitted, but only as inner or nested classes.  
  
If you have a private inner or nested class, then access is restricted to the scope of that outer class. If you have a private class on its own as a top-level class, then you can't get access to it from anywhere.  
  
If you have a protected inner or nested class, then access is permitted from inside the same package or subclasses of the outer class or something similar. If you have a protected top-level class there isn't an outer class to have subclasses to gain access from, so protected is meaningless.  
  
Making a class final and giving it a private constructor mean that you have to work out some other way of giving access, maybe via public static methods, or a public static getInstance() method.  
The final bit really means you are not allowed to extend the class. It is not usually possible to extend a top-level class with only private constructors.

***Access Modifiers for Variable (Instance / Static Variable)***

Variables are eligible for all of the above mentioned modifiers.

* Default
* Public
* Protected
* Private

***Note\*:****Visibility of the class should be checked before checking the visibility of the variable defined inside that class. If the class is visible only then the variables defined inside that class will be visible. If the class is not visible then no variable will be accessible, even if it is set to public.*

**Default -** If a variable is set to default, it will be accessible to the classes which are defined in the same package. Any method in any class which is defined in the same package can access the variable via**Inheritance**or**Direct access.**

**Public -** If a variable is set to public it can be accessible from any class available in the Java world. Any method in any class can access the given variable via**Inheritance**or**Direct access.**

**Protected -** If a variable is set to protected inside a class, it will be accessible from its sub classes defined in the same or different package only via ***Inheritance***.

***Note:*** *The only difference between protected and default is that protected access modifiers respect****class subclass relation****while default does not.*

**Private -** A variable if defined private will be accessible only from within the class it is defined. Such variables are not accessible from outside the defined class, not even its subclass.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Visibility** | **Public** | **Private** | **Protected** | **Default** |
| **Within Same Class** | Yes | Yes | Yes | Yes |
| **From Any Class in Same Package** | Yes | No | Yes | Yes |
| **From Any Sub Class in Same Package** | Yes | No | Yes | Yes |
| **From Any Sub Class from Different Package** | Yes | No | Yes (**Inheritance**) | No |
| **From Any Non Sub Class in Different Package** | Yes | No | No | No |

***Access Modifiers for Methods***

Methods are eligible for all of the above mentioned modifiers.

**Default -** When a method is set to default it will be accessible to the classes which are defined in the same package. Any method in any class which is defined in the same package can access the given method via***Inheritance***or***Direct access***.

**Public -** When a method is set to public it will be accessible from any class available in the Java world. Any method in any class can access the given method via***Inheritance***or***Direct access*** depending on class level access.

**Protected -** If a method is set to protected inside a class, it will be accessible from its sub classes defined in the same or different package.

**Note:\***The only difference between protected and default is that protected access modifiers respect**class subclass relation** while default does not.

**Private -** A method if defined private will be accessible only from within the class it is defined. Such methods are not accessible from outside the defined class, not even its subclass.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Visibility** | **Public** | **Private** | **Protected** | **Default** |
| **Within Same Class** | Yes | Yes | Yes | Yes |
| **From Any Class in Same Package** | Yes | No | Yes | Yes |
| **From Any Sub Class in Same Package** | Yes | No | Yes | Yes |
| **From Any Sub Class from Different Package** | Yes | No | Yes (**Inheritance**) | No |
| **From Any Non Sub Class in Different Package** | Yes | No | No | No |

***Access Modifier for Local Variable***

No Access Modifiers can be applied to local variables. Only final can be applied to a local variable which is a [Non Access Modifier](http://javabeginnerstutorial.com/core-java-tutorial/non-access-modifiers-in-java/) .

# Final Modifiers in Java

### Final modifiers are applicable to:

* Class
* Method
* Instance Variable
* Local Variable
* Method arguments

**Final in java** is very important keyword and can be applied to class, method, and variables in Java. In this java final tutorial we will see **what is final keyword in Java**, *what does it mean by making final variable*, final method and final class in java and *what are primary benefits of using final keywords in Java* and finally some examples of final in Java. Final is often used along with static keyword in Java to make static final constant and you will see how final in Java can increase performance of Java application.

**Example of Final variable, final method and Class in Java**

**What is final keyword in Java?**

**Final is a keyword** or reserved word in java and can be applied to member variables, methods, class and local variables in Java. Once you make a reference final you are not allowed to change that reference and compiler will verify this and raise **compilation error** if you try to re-initialized **final variables in java**.

**What is final variable in Java?**

Any variable either member variable or local variable (declared inside method or block) modified by final keyword is called final variable. **Final variables are often declare with static keyword** in java and treated as constant. Here is an example of final variable in Java

public static final String LOAN = "loan";

LOAN = new String("loan") **//invalid compilation error**

Final variables are by default read-only.

**What is final method in Java**

Final keyword in java can also be applied to methods. **A java method with final keyword is called final method and it cannot be overridden in sub-class**. You should make a method final in java if you think it’s complete and its behavior should remain constant in sub-classes. Final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded on compile time. Here is an *example of final method in Java***:**

class **PersonalLoan**{

 public final String getName(){

     return "personal loan";

 }

}

class **CheapPersonalLoan** extends **PersonalLoan**{

    @Override

    public final String getName(){

        return "cheap personal loan"; **//compilation error: overridden method is final**

    }

}

**What is final Class in Java**

**Java class with final modifier is called final**[**class in Java**](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html)**. Final class is complete in nature and cannot be sub-classed or inherited**. **Several classes in Java are final e.g. String, Integer and other wrapper classes.** Here is an *example of final class in java*

final class **PersonalLoan**{

}

class **CheapPersonalLoan** extends **PersonalLoan**{  //compilation error: cannot inherit from final class

}

**Benefits of final keyword in Java**

Here are few benefits or advantage of using final keyword in Java:

1. Final keyword improves performance. Not just JVM can cache **final variable** but also application can cache frequently use final variables.

2. Final variables are safe to share in [multi-threading](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) environment without additional synchronization overhead.

3. **Final keyword**allows [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) to optimized method, variable or class.

**Final and Immutable Class in Java**

Final keyword helps to write immutable class. Immutable classes are the one which cannot be modified once it gets created and String is primary example of immutable and final class which I have discussed in detail on [Why String is final or immutable in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html). Immutable classes offer several benefits one of them is that they are effectively read-only and can be safely shared in between multiple threads without any synchronization overhead. You cannot make a class immutable without making it final and hence final keyword is required to make a class immutable in java.

Example of Final in Java

Java has several system classes in JDK which are final, some example of final classes are String, Integer, Double and other wrapper classes. You can also use final keyword to make your code better whenever it required. See relevant section of **java final tutorial** for *example of final variable***,** *final method* and *final class in Java*.

**Important points on final in Java**

* Final keyword can be applied to member variable, local variable, method or [class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html).
* **Final member variable must be initialized at the time of declaration or inside constructor, failure to do so will result in compilation error**.
* You cannot reassign value to final variable in Java.
* Local final variable must be initializing during declaration.
* Only final variable is accessible inside anonymous class in Java.
* Final method cannot be [overridden in Java](http://javarevisited.blogspot.com/2011/12/method-overloading-vs-method-overriding.html).
* Final class cannot be inheritable in Java.
* All variable declared inside java interface are implicitly final.
* Final and abstract are two opposite keyword and a final class cannot be [abstract in java](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html).
* Final methods are bonded during compile time also called static binding.
* Final variables which is not initialized during declaration are called blank final variable and must be initialized on all constructor either explicitly or by calling this(). Failure to do so compiler will complain as "final variable (name) might not be initialized".
* Making a class, method or variable final in Java helps to improve performance because JVM gets an opportunity to make assumption and optimization.
* As per Java code convention final variables are treated as constant and written in all Caps e.g. private final int COUNT=10;
* Making a collection reference variable final means only reference cannot be changed but you can add, remove or change object inside collection. For example:

private final List loans = new ArrayList();

loans.add(“home loan”);  **//valid**

loans.add("personal loan"); **//valid**

loans = new Vector();  **//not valid**

# Abstract Class

An **abstract class**is something which is incomplete and you cannot create instance of abstract class. If you want to use it you need to make it complete or concrete by extending it. A class is called concrete if it does not contain any abstract method and implements all abstract method inherited from abstract class or interface it has implemented or extended.By the way Java has concept of abstract classes, abstract method but a variable cannot be abstract in Java. Popular example of abstract class in Java is ActionListener which has abstract method called actionPerformed(ActionEvent ae). This method is called when an ActionEvent is fired like when you click on JButton. Its common in java to attach ActionListener with JButtonby implementing abstract method actionPerformed(ActionEvent ae) using Anonymous class, as shown in below Example :

JButton ok = **new** JButton("OK");

ok.addActionListener(**new** ActionListener(){

**public** **void** **actionPerformed**(ActionEvent ae){

//code to handle event

}

});

***An abstract method in Java doesn't have body, it’s just a declaration.***In order to use abstract method you need to [override](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html)that method in sub class***.***

So ***when do you use abstraction***? (Most important in my view)

When you know something needs to be there but not sure how exactly it should look like. e.g. when I am creating a class called Vehicle, I know there should be methods like start() and stop() but don't know how that start and stop method should work, because every vehicle can have different start and stop mechanism e.g. some can be started by kicking or some can be by pressing buttons . Same concept applies to interface in Java as well, which we will discuss in some other post.

So implementation of those start() and stop() methods should be left to there concrete implementation e.g. Scooter , MotorBike , Car etc.

Abstraction: Things to Remember

1) Use abstraction if you know something needs to be in class but implementation of that varies. Abstraction is actually result of thought process and it really need good experience of both domain and Object oriented analysis and design to come up with good abstraction for your project.

2) In Java you cannot create instance of abstract class using new operator, its compiler error. **Though abstract class can have constructor.**

3) Abstract is a keyword in Java, which can be used with both class and method.  Abstract class can contain both abstract and concrete method. Abstract method doesn't have body, just declaration.

4) A class automatically becomes abstract class when any of its method declared as abstract.

5) Abstract method doesn't have method body.

6) In Java, variable cannot be made abstract , its only class or methods which would be abstract.  
7) If a class extends an abstract class or interface it has to provide implementation to all its abstract method to be a concrete class. Alternatively this class can also be abstract.

# Interface

**Interface in java** is core part of Java programming language and one of the ways to achieve [abstraction in Java](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html)along with abstract class. Even though interface is fundamental [object oriented concept](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html); Many Java programmers thinks Interface in Java as advanced concept and refrain using interface from early in programming career. At very basic level **interface in java**is a keyword but same time it is an **object oriented term to define contracts**and**abstraction**, this contract is followed by any implementation of Interface in Java. Since [multiple inheritance is not allowed in Java](http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html), interface is only way to implement multiple inheritance at Type level. In this **Java tutorial** we will see **what is an interface in Java**, How to use interface in Java and where to use interface in Java and some important points related to Java interface. *What is an interface in Java* is also a [common core Java question](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) which people asked on various programming exams and interviews.

**Key Points about Interface in Java**

* All Interface methods are implicitly public and abstract. Even if you write these keyword it will not create problem as you can see in second method declaration.
* Interfaces can declare only constant. Instance variables are not allowed. It means all variables inside Interface must be public, static, final. Variables inside interface are implicitly public static final.
* Interface methods cannot be static.
* Interface methods cannot be final, strictfp or native.
* Interface can extend one or more other interface. Note: Interface can only extend other interface.
* Interface is 100% abstract class (***Implicitly***).
* Interfaces can be implemented by any class from any inheritance tree.
* All methods in Interfaces are abstract.
* Interface can have constants, these constants are public, static and final(***Implicitly***).
* Interface methods are implicitly ***public & abstract.***
* Class implementing an interface can also be an abstract class.
* An abstract class which is implementing an interface need not implement all abstract method.
* A class can extend *more than one* Interface.
* Interfaces cannot extend a class or implement an Interface.
* Interface can extend another Interface.
* A non-abstract class which is implementing an Interface needs to follow some rules

.

* 1. This class needs to provide concrete implementation of all abstract method.
  2. All rules of Overriding needs to be followed.
  3. It must maintain the exact signature of method.

Java standard library itself has many inbuilt interfaces like [Serializable](http://javarevisited.blogspot.sg/2011/04/top-10-java-serialization-interview.html), Clonnable, [Runnable](http://www.blogger.com/goog_1507016574)or Callable interface in Java.  ***Declaring interface is easy but making it correct in first attempt is hard*** but if you are in business of designing API then you need to get it right in first attempt because its not possible to modify interface once it released without breaking all its implementation. Here is an example of declaring interface in Java:

**interface** SessionIDCreator **extends** **Serializable**, **Cloneable**{  
        [**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) TYPE = "AUTOMATIC";  
        **int** createSessionId();  
    }  
    
 **class** SerialSessionIDCreator **implements** SessionIDCreator{  
 **private** **int** lastSessionId;  
 @**Override**  
        **public** **int** createSessionId() {  
            **return** lastSessionId++;

        }      
 }

In above *example of interface in Java*, SessionIDCreator is an interface while SerialSessionIDCreator is a implementation of interface. **@Override annotation can be used on interface method from Java 6 onwards**, so always try to use it. It’s one of those coding practice which should be in your [code review checklist](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html).

**When to use interface in Java**

*Interface* is best choice for **Type declaration** or defining contract between multiple parties. If multiple programmers are working in different module of project they still use each other’s API by defining interface and not waiting for actual implementation to be ready. This brings us lot of flexibility and speed in terms of coding and development. Use of Interface also ensures [best practices](http://javarevisited.blogspot.com/2011/08/code-comments-java-best-practices.html) like "**programming for interfaces than implementation**" and results in more flexible and maintainable code. Though interface in Java is not the only one who provides higher level abstraction, you can also use abstract class but choosing between Interface in Java and abstract class is a skill. Difference between Interface in Java and abstract class in java is also a very [popular java interview question](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html)

That's it for now on *what is Interface in Java,* specifics of Java interface and **How and when to use Interface in Java**.  Interface is key to write flexible and maintainable code. If you are not yet using interface in your code than start thinking in terms of interfaces and use it as much possible. You will learn more about interfaces when you start using design patterns. Many design patterns like [decorator pattern](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html), [Factory method pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html)  or [Observer design pattern](http://javarevisited.blogspot.com/2011/12/observer-design-pattern-java-example.html)  makes very good use of Java interfaces.

## Why Java doesn't support multiple inheritance

1) First reason is **ambiguity around Diamond problem**, consider a class A has foo() method and then B and C derived from A and has their own foo() implementation and now class D derive from B and C using multiple [inheritance](http://javarevisited.blogspot.com/2012/10/what-is-inheritance-in-java-and-oops-programming.html) and if we refer just foo() compiler will not be able to decide which foo() it should invoke. This is also called Diamond problem because structure on this inheritance scenario is similar to 4 edge diamond, see below

           A foo()

           / \

          /   \

   foo() B     C foo()

          \   /

           \ /

            D

           foo()

In my opinion even if we remove the top head of diamond class A and allow multiple inheritances we will see this problem of ambiguity.  
  
Sometimes if you give this reason to interviewer he asks if C++ can support *multiple inheritance* than why not Java. hmmmmm in that case I would try to explain him the second reason which I have given below that it’s not because of technical difficulty but more to maintainable and clearer design was driving factor though this can only be confirmed by any of java designer and we can just speculate. [Wikipedia link](http://en.wikipedia.org/wiki/Diamond_problem) has some good explanation on how different language address problem arises due to diamond problem while using multiple inheritances.  
  
2) Second and more convincing reason to me is that **multiple inheritances does complicate the design and creates problem during casting, constructor chaining etc** and given that there are not many scenario on which you need multiple inheritance its wise decision to omit it for the sake of simplicity. Also java avoids this ambiguity by supporting single inheritance with interfaces. Since interface only have method declaration and doesn't provide any implementation there will only be just one implementation of specific method hence there would not be any ambiguity.

**Difference between abstract class and interface in Java**

While deciding *when to use interface and abstract class*, it’s important to know difference between abstract class and interface in Java. In my opinion, following two differences between them drives decision about when to use abstract class or interface in Java.

1) Interface in Java can only contain declaration. You cannot declare any concrete methods inside interface. On the other hand abstract class may contain both abstract and concrete methods, which makes abstract class an ideal place to provide common or default functionality. I suggest reading my post [10 things to know about interface in Java](http://javarevisited.blogspot.com/2012/04/10-points-on-interface-in-java-with.html) to know more about interfaces, particularly in Java programming language

2) Java interface can extend multiple interface also Java class can implement multiple interfaces, Which means interface can provide more Polymorphism support than abstract class . By extending abstract class, a class can only participate in one Type hierarchy but by using interface it can be part of multiple type hierarchies. E.g. a class can be Runnable and Displayable at same time. One example I can remember of this is writing GUI application in J2ME, where  class extends Canvas and implements CommandListener to provide both graphic and event-handling functionality..

3) In order to implement interface in Java, until your class is abstract, you need to provide implementation of all methods, which is very painful. On the other hand abstract class may help you in this case by providing default implementation. Because of this reason, I prefer to have minimum methods in interface, starting from just one, I don't like idea of [marker interface](http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html), once annotation is introduced in Java 5. If you look JDK or any framework like Spring, which I does to understand OOPS and design patter better, you will find that most of interface contains only one or two methods e.g. Runnable, Callable, ActionListener etc.

# Overloading vs. Overriding

Method Overloading is a feature that allows a class to have two or more methods having same name, if their argument lists are different. In the last tutorial we discussed constructor overloading that allows a class to have more than one constructor having different argument lists.

Argument lists could differ in –  
1. Number of parameters.  
2. Data type of parameters.  
3. Sequence of Data type of parameters.

Method overloading is also known as **Static Polymorphism**.

Points to Note:   
1. [Static Polymorphism](http://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) is also known as compile time binding or early binding.  
2. [Static binding](http://beginnersbook.com/2013/04/java-static-dynamic-binding/) happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

**Overloading Method Rules**

There are some rules associated with overloaded method.

**Overloaded methods**

* **Must change the argument list**
* Can change the return type
* Can change the access modifier(Broader)
* Can declare new or broader checked exception

A method can be overloaded in Class or in SubClass.

Overloading Method Example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //Overloaded method with one argument  public void add(int input1, int input2) {  System.out.println("In method with two argument");  }            //Overloaded method with one argument  public void add(int input1) {  System.out.println("In method with one argument");  } |

Invoking Overloaded Method

Out of several available overloaded method which method to invoke is based on the arguments.

|  |  |
| --- | --- |
| 1  2 | add(3,4);       add(5); |

First call will execute the first method and second will execute the second method.

**Overloading Cheat sheet**

* Using the same method name but with different argument is called overloading.
* **Constructors**can also be overloaded
* Overloaded methods must have different argument set.
* Overloaded methods ***may*** have **different return type.**
* Overloaded methods ***may*** have different access modifier.
* Overloaded methods ***may*** throw different exception ***broader or narrow*** no restriction
* Methods from **super class**can also be overloaded in subclass.
* Polymorphism applies to overriding not Overloading
* Which overloaded method will be invoked is decided on***compile time*** on the basis of **reference type**.

**Method Overriding**

Class inheriting the method from its **super class** has the option to **override** it. Benefit of overriding is the ability to define behavior specific to particular class. In case of concrete subclass it is forced to implement all methods defined in abstract class if no other super class implemented it in hierarchy. Overriding sometimes referred as Run time Binding. It means which overridden method is to be invoked will be determined by reference type and not the instance type.

Method Overriding Example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class ParentClass  {  public void show()  {  System.out.println("Show method of Super class");  }  }    public class SubClass extends ParentClass  {  //below method is overriding the ParentClass version of show method  public void show()  {  System.out.println("Show method of Sub class");  }  } |

**Method Override Rules**

* Overriding method cannot have more restrictive access modifier than the method being overridden but it can be less.
* The argument list must exactly match that of the overridden method, if they don’t it is more likely that you are overloading the method.
* Return type must be the same as, or subtype of the return type declared in overridden method in Super class.
* Overriding method can throw any unchecked exception(Runtime) but it can throw checked exception which is broader or new than those declared by the overridden method but it cannot throw fewer or narrow checked exception.
* Final method cannot be overridden.
* Static methods cannot be overridden. Static method looks to overridden but it is hidden.
* If a method cannot be inherited then it cannot be overridden.

**Invoke Overridden method from Super class**

What if you want to invoke super class overridden method before executing the subclass method. You can use SUPER keyword.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class SubClass extends superclass {    void method() {  super.method();  System.out.println("In Sub Class");  }    public static void main(String[] args) {  SubClass obj = new SubClass();  obj.method();  }  }    class superclass {  void method() {  System.out.println("In Super Class");  }  } |

Output

|  |  |
| --- | --- |
| 1  2 | In Super Class  In Sub Class |

Static Method cannot be overridden

Static method cannot be overridden. It might looks like it is overridden but it is not. Static method can be hidden.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class SubClass extends superclass {  static void method() {  // super.method(); // Super keyword will not work here. As it is not overriden method  System.out.println("In Sub Class");  }  *@SuppressWarnings*("static-access") // The static method method() from the type SubClass should be accessed in a static way  public static void main(String[] args) {  SubClass obj = new SubClass();  obj.method();                  SubClass.method();// It is same as above. Same method will be invoked  }  }  class superclass {  static void method() {  System.out.println("In Super Class");  }  } |

Here super keyword cannot be used to invoke super class method. As it is not overridden method from super class.

**Cheat-sheet**

* Constructorcannot be overridden.
* Overriding methods must have the same argument set.
* Overriden methods must have same return type. These return type can also be the subclass (***covariant return***).
* Overriden method **cannot** have more restrictive access modifier.
* Overriden method **cannot** throw new or broader exception (***Checked***).
* Overriden method **can throw** any unchecked exception.
* **Final** methods ***cannot*** be overriden.
* ***Private methods*** are not inherited to subclass hence it cannot be overriden in subclass.
* Polymorphism applies to overriding.
* ***Object type*** determines which overridden method will be invoked and that will be decided at the runtime.

**Overloading vs. Overriding in Java**

* Overloading happens at [compile-time](http://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) while Overriding happens at [runtime](http://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/): The binding of overloaded method call to its definition has happens at compile-time however binding of overridden method call to its definition happens at runtime.
* Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
* The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
* [Static binding](http://beginnersbook.com/2013/04/java-static-dynamic-binding/) is being used for overloaded methods and dynamic is being used for overridden/overriding methods.
* Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.
* Private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.
* Return type of method does not matter in case of method overloading; it can be same or different. However in case of method overriding the overriding method can have more specific return type ([refer this](http://stackoverflow.com/questions/14694852/can-overridden-methods-differ-in-return-type)).
* Argument list should be different while doing method overloading. Argument list should be same in method Overriding.

# Static Keyword

**What is Static?**

Static is a Non Access Modifier.

**Applicable to**

Static keyword can be applied on

* *Method*
* *Variable*
* *Class nested within another Class*
* *Initialization Block*

**Not Applicable to**

Static keyword cannot be applied to

* Class (Not Nested)
* Constructor
* Interfaces
* Method Local Inner Class(Difference then  nested class)
* Inner Class methods
* Instance Variables
* Local Variables

**Purpose of Static Keyword**

Used to attach Variable / Method to class. Variable and Method marked static belong to the class rather than to any particular instance.

**How to Invoke**

Static variable / methods can be used without having any Instance of the class. Only class is required to invoke a static method or static variable.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | /\*  \* Here we will learn to access Static method and Static Variable.  \*/  public class JavaStaticExample {  static int i = 10;  static void method() {  System.out.println("Inside Static method");  }  public static void main(String[] args) {  // Accessing Static method  JavaStaticExample.method();    // Accessing Static Variable  System.out.println(JavaStaticExample.i);    /\*  \* No Instance is required to access Static Variable or Method as we  \* have seen above. Still we can access the same static variable and  \* static method using Instace references as below.  \*/  JavaStaticExample obj1 = new JavaStaticExample();  JavaStaticExample obj2 = new JavaStaticExample();    /\*  \* Accessing static variable in Non Static way. Compiler will warn you  \* with below warning.  \*  \* The static field JavaStaticExample.i should be accessed in a static  \* way.  \*/  System.out.println(obj1.i);  // Accessing satic method using reference.  // Warning by compiler  // "The static method method() from the type JavaStaticExample should be accessed in a static way"  obj1.method();  }  } |

Output of the above program

|  |  |
| --- | --- |
| 1  2  3  4 | Inside Static method  10  10  Inside Static method |

**Note\*:** Static keyword can be used with Variables & Methods. It is ***not applicable to class***.

**Class Variables – Static Fields**

Class variables also known as static fields share characteristics across all objects within a class. When you declare a field to be static, only a single instance of the associated variable is created, which is common to all the objects of that class. Hence when one object changes the value of a class variable, it affects all the objects of the class. We can access a class variable by using the name of the class, and not necessarily using a reference to an individual object within the class. Static variables can be accessed even though no object of that class exists. It is declared using the static keyword.

**Class Methods – Static Methods**

Class methods, similar to Class variables can be invoked without having an instance of the class. Class methods are often used to provide global functions for Java programs. For example, methods in the java.lang.Math package are class methods. You cannot call non-static methods from inside a static method.

**Static Keyword Rules**

* ***Variable / Methods***marked static belong to the **Class** rather than to any particular Instance.
* **Static method/variables** can be used without creating any instance of the class.
* If there are instances, a static variable of a class will be shared by all instances of that class; there will be **only one copy**.
* A static method can’t access non static variable and also it cannot directly invoke non static method (But it can invoke/access method/variable via *instances*).

**Cheat-sheet**

* ***Static***is a Non Access Modifier.
* **Static** modifier can be applied to Variable / Method / Block / Inner Class.
* ***Static members*** belong to class only not any instance.
* Static method **cannot** access *instance variable*.
* Static methods **cannot** be *overriden*. As they are class specific and doesn’t belong to any Instance.
* Static methods can be ***redefined***.
* Static variable cannot be directly accessed by Non Static Methods.
* If a class contain any static block then that block will get executed only when class get loaded in JVM. Creating multiple instances doesn’t mean that Static block will get executed multiple time. Only Constructor will get executed multiple times.
* If Class.forName(“*class\_name*“) is call0065d then static block of the class will get executed.

# Serialization

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Serialization in java** is a mechanism of writing the state of an object into a byte stream.  It is mainly used in Hibernate, RMI, JPA, EJB, JMS technologies.  The reverse operation of serialization is called *deserialization*.  The String class and all the wrapper classes implements*java.io.Serializable* interface by default.  Advantage of Java Serialization  It is mainly used to travel object's state on the network (known as marshaling).  java.io.Serializable interface  Serializable is a marker interface (has no body). It is just used to "mark" java classes which support a certain capability. It must be implemented by the class whose object you want to persist. Let's see the example given below:  **import** java.io.Serializable;  **public** **class** Student **implements** Serializable{  **int** id;  String name;  **public** Student(**int** id, String name) {  **this**.id = id;  **this**.name = name;   }  }  ObjectOutputStream class  The ObjectOutputStream class is used to write primitive data types and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.  Constructor   |  | | --- | | 1) public ObjectOutputStream(OutputStream out) throws IOException {}creates an ObjectOutputStream that writes to the specified OutputStream. |   Important Methods   |  |  | | --- | --- | | Method | Description | | 1) public final void writeObject(Object obj) throws IOException {} | writes the specified object to the ObjectOutputStream. | | 2) public void flush() throws IOException {} | flushes the current output stream. | | 3) public void close() throws IOException {} | closes the current output stream. |   Example of Java Serialization  In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.  **import** java.io.\*;  **class** Persist{  **public** **static** **void** main(String args[])**throws** Exception{  Student s1 =**new** Student(211,"ravi");    FileOutputStream fout=**new** FileOutputStream("f.txt");  ObjectOutputStream out=**new** ObjectOutputStream(fout);    out.writeObject(s1);  out.flush();  System.out.println("success");   }  }  **success**  **Deserialization in java**  Deserialization is the process of reconstructing the object from the serialized state.It is the reverse operation of serialization.  ObjectInputStream class  An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.  Constructor   |  |  | | --- | --- | | **1) public ObjectInputStream(InputStream in) throws IOException {}** | creates an ObjectInputStream that reads from the specified InputStream. |   Important Methods   |  |  | | --- | --- | | Method | Description | | 1) public final Object readObject() throws IOException, ClassNotFoundException{} | reads an object from the input stream. | | 2) public void close() throws IOException {} | closes ObjectInputStream. |   Example of Java Deserialization  **import** java.io.\*;  **class** Depersist{  **public** **static** **void** main(String args[])**throws** Exception{      ObjectInputStream in=**new** ObjectInputStream(**new** FileInputStream("f.txt"));    Student s=(Student)in.readObject();    System.out.println(s.id+" "+s.name);      in.close();   }  }  211 ravi  **Java Serialization with Inheritance (IS-A Relationship)**  If a class implements serializable then all its sub classes will also be serializable. Let's see the example given below:  **import** java.io.Serializable;  **class** Person **implements** Serializable{  **int** id;   String name;   Person(**int** id, String name) {  **this**.id = id;  **this**.name = name;   }  }  **class** Student **extends** Person{   String course;  **int** fee;  **public** Student(**int** id, String name, String course, **int** fee) {  **super**(id,name);  **this**.course=course;  **this**.fee=fee;   }  }  Now you can serialize the Student class object that extends the Person class which is Serializable.Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.  **Java Serialization with Aggregation (HAS-A Relationship)**  If a class has a reference of another class, all the references must be Serializable otherwise serialization process will not be performed. In such case, *NotSerializableException* is thrown at runtime.  **class** Address{   String addressLine,city,state;  **public** Address(String addressLine, String city, String state) {  **this**.addressLine=addressLine;  **this**.city=city;  **this**.state=state;   }  }  **import** java.io.Serializable;  **public** **class** Student **implements** Serializable{  **int** id;   String name;   Address address;//HAS-A  **public** Student(**int** id, String name) {  **this**.id = id;  **this**.name = name;   }  }  Since Address is not Serializable, you can not serialize the instance of Student class.  *Note: All the objects within an object must be Serializable.*  **Java Serialization with static data member**  If there is any static data member in a class, it will not be serialized because static is the part of class not object.  **class** Employee **implements** Serializable{  **int** id;   String name;  **static** String company="SSS IT Pvt Ltd";//it won't be serialized  **public** Student(**int** id, String name) {  **this**.id = id;  **this**.name = name;   }  }  **Java Serialization with array or collection**  Rule: In case of array or collection, all the objects of array or collection must be serializable. If any object is not serialiizable, serialization will be failed.  **Externalizable in java**  The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format. It is not a marker interface.  The Externalizable interface provides two methods:  **public void writeExternal(ObjectOutput out) throws IOException**  **public void readExternal(ObjectInput in) throws IOException** |

**Java Transient Keyword**

If you don't want to serialize any data member of a class, you can mark it as transient.

# Strings

String in Java is very special class and most frequently used class as well. There are lot many things to learn about String in Java than any other class, and having a good knowledge of different String functionalities makes you to use it properly. Given heavy use of Java String in almost any kind of project, it become even more important to know subtle detail about String. Though I have shared lot of String related article already here in **Java revisited**, this is an effort to bring some of String feature together. In this tutorial we will see some important points about Java String, which is worth remembering. You can also refer my earlier post [10 advanced Java String questions](http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html) to know more about String. Though I tried to cover lot of things, there are definitely few things, which I might have missed; please let me know if you have any question or doubt on java.lang.String functionality and I will try to address them here.

**1) Strings are not null terminated in Java.**

Unlike C and C++, String in Java doesn't terminate with null character. Instead String are Object in Java and backed by character array. You can get the character array used to represent String in Java by calling toCharArray() method of java.lang.String class of JDK.

**2) Strings are immutable and final in Java**

Strings are immutable in Java it means once created you cannot modify content of String. If you modify it by using toLowerCase(), toUpperCase() or any other method,  It always result in new String. Since String is final there is no way anyone can extend String or override any of String functionality. Now if you are puzzled [why String is immutable or final in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html). Checkout the link.

**3) Strings are maintained in String Pool**

As I Said earlier String is special class in Java and all String literal e.g. "abc"  (anything which is inside double quotes are String literal in Java) are maintained in a separate String pool, special memory location inside Java memory, more precisely inside [PermGen Space](http://javarevisited.blogspot.com/2012/01/tomcat-javalangoutofmemoryerror-permgen.html). Any time you create a new String object using String literal, JVM first checks String pool and if an object with similar content available, than it returns that and doesn't create a new object. JVM doesn't perform String pool check if you create object using new operator.

You may face subtle issues if you are not aware of this String behaviour , here is an example

String name = "Scala"; //1st String object

String name\_1 = "Scala"; //same object referenced by name variable

String name\_2 = **new** String("Scala") //different String object

//this will return true

**if**(name==name\_1){

System.out.println("both name and name\_1 is pointing to same string object");

}

//this will return false

**if**(name==name\_2){

System.out.println("both name and name\_2 is pointing to same string object");

}

if you compare name and name\_1 using equality operator "==" it will return true because both are pointing to same object. While name==name\_2 will return false because they are pointing to different string object. It's worth remembering that [equality "==" operator compares object memory location](http://javarevisited.blogspot.sg/2012/12/difference-between-equals-method-and-equality-operator-java.html) and not characters of String. By default Java puts all string literal into string pool, but you can also put any string into pool by calling intern() method of java.lang.String class, like string created using new() operator.

**4) Use Equals methods for comparing String in Java**

String class overrides equals method and provides a content equality, which is based on characters, case and order. So if you want to compare two String object, to check whether they are same or not, always use equals() method instead of equality operator. Like in earlier example if  we use [equals method](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) to compare objects, they will be equal to each other because they all contains same contents. Here is example of comparing String using equals method.

String name = "Java"; //1st String object

String name\_1 = "Java"; //same object referenced by name variable

String name\_2 = **new** String("Java") //different String object

**if**(name.equals(name\_1)){

System.out.println("name and name\_1 are equal String by equals method");

}

//this will return false

**if**(name==name\_2){

System.out.println("name\_1 and name\_2 are equal String by equals method");

}

You can also check my earlier post [difference between equals() method and == operator](http://javarevisited.blogspot.com/2012/12/difference-between-equals-method-and-equality-operator-java.html) for more detail discussion on consequences of comparing two string using == operator in Java.

**5) Use indexOf() and lastIndexOf() or matches(String regex) method to search inside String**

String class in Java provides convenient method to see if a character or sub-string or a pattern exists in current String object. You can use indexOf() which will return position of character or String, if that exist in current String object or -1 if character doesn't exists in String. lastIndexOf is similar but it searches from end. String.match(String regex) is even more powerful, which allows you to search for a [regular expression pattern](http://javarevisited.blogspot.com/2012/10/regular-expression-example-in-java-to-check-String-number.html) inside String. here is examples of indexOf, lastIndexOf and matches method from java.lang.String class.

String str = "Java is best programming language";

**if**(str.indexOf("Java") != -**1**){

     System.out.println("String contains Java at index :" + str.indexOf("Java"));

}

**if**(str.matches("J.\*")){

     System.out.println("String Starts with J");

}

str ="Do you like Java ME or Java EE";

**if**(str.lastIndexOf("Java") != -**1**){

      System.out.println("String contains Java lastly at: " + str.lastIndexOf("Java"));

}

As expected indexOf will return 0 because characters in String are indexed from zero. lastIndexOf returns index of second “Java”, which starts at 23 and matches will return true because J.\* pattern is any String starting with character J followed by any character because of dot(.) and any number of time due to asterick (\*).

Remember matches() is tricky and some time non-intuitive. If you just put "Java" in matches it will return false because String is not equals to "Java" i.e. in case of plain text it behaves like equals method. See [here](http://java67.blogspot.sg/2012/09/java-string-matches-example-regular-expression.html) for more examples of String matches() method.

Apart from indexOf(), lastIndexOf() and matches(String regex) String also has methods like startsWith() and endsWidth(), which can be used to check an String if it starting or ending with certain character or String.

**6) Use SubString to get part of String in Java**

Java String provides another useful method called substring(), which can be used to get parts of String. basically you specify start and end index and substring() method returns character from that range. Index starts from 0 and goes till String.length()-1. By the way String.length() returns you number of characters in String, including white spaces like tab, space. One point which is worth remembering here is that substring is also backed up by character array, which is used by original String. This can be dangerous if original string object is very large and substring is very small, because even a small fraction can hold reference of complete array and prevents it from being garbage collected even if there is no other reference for that particular String. Read[How Substring works in Java](http://javarevisited.blogspot.com/2011/10/how-substring-in-java-works.html) for more details. Here is an example of using SubString in Java:

String str = "Java is best programming language";

//this will return part of String str from index 0 to 12

String subString = str.substring(**0**,**12**);

System.out.println("Substring: " + subString);

**7) "+" is overloaded for String concatenation**

*Java doesn't support Operator overloading* but String is special and + operator can be used to concatenate two Strings. It can even used to convert int, char, long or double to convert into String by simply concatenating with empty string "". internally + is implemented using StringBuffer prior to Java 5 and StringBuilder from Java 5 onwards. This also brings point of usingStringBuffer or StringBuilder for manipulating String. Since both represent mutable object they can be used to reduce string garbage created because of temporary String. Read more about [StringBuffer vs StringBuilder](http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) here.

**8) Use trim() to remove white spaces from String**

String in Java provides trim() method to remove white space from both end of String. If trim() removes white spaces it returns a new String otherwise it returns same String. Along with trim() String also provides replace() and replaceAll() method for replacing characters from String. replaceAll method even support regular expression. Read more about How to replace String in Java [here](http://javarevisited.blogspot.com/2011/12/java-string-replace-example-tutorial.html).

**9) Use split() for splitting String using Regular expression**

String in Java is feature rich. it has methods like split(regex) which can take any String in form of regular expression and split the String based on that. particularly useful if you dealing with comma separated file (CSV) and wanted to have individual part in a String array. There are other methods also available related to splitting String, see this [Java tutorial to split string](http://javarevisited.blogspot.com/2011/09/string-split-example-in-java-tutorial.html) for more details.

**10) Don't store sensitive data in String**

String pose security threat if used for storing sensitive data like passwords, SSN or any other sensitive information. Since String is immutable in Java there is no way you can erase contents of String and since they are kept in String pool (in case of String literal) they stay longer on Java heap ,which exposes risk of being seen by anyone who has access to Java memory, like reading from memory dump. Instead char[] should be used to store password or sensitive information. See [Why char[] is more secure than String for storing passwords in Java](http://javarevisited.blogspot.com.br/2012/03/why-character-array-is-better-than.html) for more details.  
  
  
**11) Character Encoding and String**  
Apart from all these 10 facts about String in Java, the most critical thing to know is *what encoding your String is using*. It does not make sense to have a String without knowing what encoding it uses. There is no way to interpret an String if you don't know the encoding it used. You can not assume that "plain" text is ASCII. If you have a String, in memory or stored in file, you must know what encoding it is in, or you cannot display it correctly. By default Java uses platform encoding i.e. character encoding of your server, and believe me this can cause huge trouble if you are handling Unicode data, especially if you are [converting byte array to XML String](http://javarevisited.blogspot.sg/2013/03/convert-and-print-byte-array-to-hex-string-java-example-tutorial.html). I have faced instances where our program fail to interpret Strings from European language e.g. German, French etc. because our server was not using Unicode encodings like UTF-8 or UTF-16. Thankfully, Java allows you to specify default character encoding for your application using system property file.encoding. See [here](http://javarevisited.blogspot.com/2012/01/get-set-default-character-encoding.html)to read more about character encoding in Java

That's all about String in Java. As I have said String is very special in Java, sometime even refer has God class. It has some unique feature like immutability, concatenation support, caching etc, and to become a serious Java programmer, detailed knowledge of String is quite important. Last but not the least don't forget about [character encoding](http://javarevisited.blogspot.com/2012/01/get-set-default-character-encoding.html) while converting a byte array into String in Java. Good knowledge of java.lang.String is must for good Java developers.

**Difference between StringBuffer and StringBuilder**

There are many differences between StringBuffer and StringBuilder. A list of differences between StringBuffer and StringBuilder are given below:

|  |  |  |
| --- | --- | --- |
| No. | StringBuffer | StringBuilder |
| 1) | StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is less efficient than StringBuilder. | StringBuilder is more efficient than StringBuffer. |

StringBuffer Example

**public** **class** BufferTest{

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

        StringBuffer buffer=**new** StringBuffer("hello");

        buffer.append("java");

        System.out.println(buffer);

    }

}

hellojava

StringBuilder Example

**public** **class** BuilderTest{

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

        StringBuilder builder=**new** StringBuilder("hello");

        builder.append("java");

        System.out.println(builder);

    }

}

hellojava

**Performance Test of StringBuffer and StringBuilder**

Let's see the code to check the performance of StringBuffer and StringBuilder classes.

**public** **class** ConcatTest{

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

**long** startTime = System.currentTimeMillis();

        StringBuffer sb = **new** StringBuffer("Java");

**for** (**int** i=0; i<10000; i++){

            sb.append("Tpoint");

        }

        System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() - startTime) + "ms");

        startTime = System.currentTimeMillis();

        StringBuilder sb2 = **new** StringBuilder("Java");

**for** (**int** i=0; i<10000; i++){

            sb2.append("Tpoint");

        }

        System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() - startTime) + "ms");

    }

}

Time taken by StringBuffer: 16ms

Time taken by StringBuilder: 0ms

# [How to resolve java.lang.UnsupportedClassVersionError with example](http://javarevisited.blogspot.in/2011/07/javalangunsupportedclassversionerror.html)

[How to resolve java.lang.UnsupportedClassVersionError with example](http://javarevisited.blogspot.in/2011/07/javalangunsupportedclassversionerror.html" \o "How to resolve java.lang.UnsupportedClassVersionError  with example)

java.lang.UnsupportedClassVersionError **is a quite common error after [NoClassDefFoundError](http://javarevisited.blogspot.com/2011/06/noclassdeffounderror-exception-in.html) or ClassNotFoundException they all seems to related to class files but they all are different and there cause and resolution are different. In this java tutorial we will see what is**UnsupportedClassVersionError **in Java? Why UnsupportedClassVersionError** **comes in Java? What is class file format and version numbers associated with it and finally how to resolve**UnsupportedClassVersionError **in Java.**

**This article is in continuation of debugging tutorials like**[**How to remote debug Java program in Eclipse**](http://javarevisited.blogspot.com/2011/02/how-to-setup-remote-debugging-in.html)**and**[**10 Java debugging tips in Eclipse**](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html)**. If you have not read those article you may find them useful.**

**How to resolve UnsupportedClassVersionError in Java**

**What is UnSupportedClassVersionError in Java?**

Java.lang.UnsupportedClassVersionError is a subclass of java.lang.ClassFormatError. This is a kind of linking error which occurs during linking phase accordingly java.lang.ClassFormatError has also derived from java.lang.LinkageError. As the name suggests "UnSupportedClassVersionError" so it’s related to unsupported class version, now questions comes what is class version in Java? Well every source file is compiled into class file and each class file has two versions associated with it, major version and minor version.   
 **The Version of class file is represented as major\_version.minor\_version**. This version is used to determine *format of class file in Java*.

According to Java Virtual Machine specification, “A JVM implementation can support a class file format of version v if and only if v lies in some contiguous range Mi.0 v Mj.m. Only Sun can specify what range of versions a JVM implementation conforming to a certain release level of the Java platform may support.” For example: JDK 1.2 supports class file formats from version 45.0 to version 46.0 inclusive. So if a class file has version 48.0 it means that major version of class file is "48" and minor version is "0", which tells us that JDK 1.4 has been used to compile and generate that class file.

When UnSupportedClassVersionError in Java comes:

So now we got the theory behind class file format and major and minor version of class file in Java. Now a million dollar question is when UnSupportedClassVersionError in Java does occur?  precise answer of this is "When JVM tries to load a class and found that class file version is not supported it throws *UnSupportedClassVersionError* and it generally occurs if a **higher JDK version  is used to compile the source file and  a lower JDK version is used to run the program**. for example if you compile your java source file in JDK 1.5 and you will try to run it on JDK 1.4 you will get error **"java.lang.UnsupportedClassVersionError: Bad version number in .class file** [at java.lang.ClassLoader.defineClass1(Native Method)]".  
  
But its important to note is that vice-versa is not true "you can compile your program in J2SE 1.4 and run on J2SE 1.5 and you will not get any UnSupportedClassVersionError". When a higher JDK is used for compilation it creates class file with higher version and when a lower JDK is used to run the program it found that higher version of class file not supported at JVM level and results in java.lang.UnsupportedClassVersionError.

How to fix UnSupportedClassVersionError

Now we know the *root cause of UnSupportedClassVersionError* that we are using a lower JVM for running the program. But major problem is that stack trace of UnSupportedClassVersionError will not tell you for which class it’s coming. So if you are using multiple third party jars in your application you find that it comes at a particular part when JVM tries to load a class from a particular jar. anyway we all know that latest version of JDK is 1.6 so maximum version of class file could be generated by JDK 6, so by using JDK 6 we can solve UnSupportedClassVersionError, but many times its not easy to just move to higher JDK version. So I would suggest:

1) Find out due to which jar or class file this UnSupportedClassVersionError is coming?

2) Try to compile source code of that jar with the JDK version you are using to run your program, if source is available.

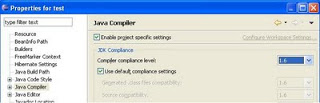
3) If you don't have source try to find the compatible version of that library.

4) Increase the JRE version you are using to run your program.

You can go by any approach to resolve UnSupportedClassVersionError based upon your need. Generally a higher JVM version is ok and does not cause any problem unless the class file format is quite old and no more supported by Sun in higher JVMs. The best way to deal with UnSupportedClassVersionError in Java is to use same version or JDK and JRE for compiling and running your program.

Example of UnSupportedClassVersionError in Java

You can easily reproduce UnSupportedClassVersionError by using javac of higher JDK and "java" from lower Java version. Let’s see some of examples of UnSupportedClassVersionError in Java:  
  
1) **java.lang.UnsupportedClassVersionError: EquityTradingManager (Unsupported major.minor version 49.0)**  
      at java.lang.ClassLoader.defineClass0(Native Method)  
      at java.lang.ClassLoader.defineClass(ClassLoader.java:539)  
==>Since we know that major version 49 is supported by JDK 1.5, so these will "java.lang.UnsupportedClassVersionError” will come if JVM used to run this program is lower than Java 1.5.  
  
2**) Java.lang.UnsupportedClassVersionError: Bad version number in .class file**  
  
3) **java.lang.unsupportedclassversionerror unsupported classversion 50.0**  
==> Compile in JDK 1.6 and running on lower version than Java 6.  
  
4) **java.lang.unsupportedclassversionerror unsupported classversion 49.0**  
==> compiled in Java 5 and running on lower JVM than JDK 5.  
  
5) **java.lang.unsupportedclassversionerror bad version number in eclipse.**

[](http://1.bp.blogspot.com/-w6p4e0W4E-Q/Tg6ZWAtmUZI/AAAAAAAAAMU/1LZYAVb3Yvg/s1600/unsupportedclassversionerror_in_java.GIF)

==> Most of us use eclipse for building and running project some of us also use ant for building project. In eclipse there is some setting related to java source version which if you got incorrect can result in "java.lang.unsupportedclassversionerror bad version number". so make sure you have correct configuration. For example if you compile with source compatible 1.6 you need JRE 6 to execute the program. To check the compiler setting in eclipse go to project  ==>Properties==>Java Compiler as shown in image

Important point about UnSupportedClassVersionError in Java:

1) If you encounter UnSupportedClassVersionError, check the JRE version you are using to run program and switch to higher version for quick solution.  
2) java.lang.UnsupportedClassVersionError is derived from java.lang.LinkageError, so it will not be detected in compile time and it will only come on runtime, precisely when JVM tries to load a class.  
3) Class file format which is identified using major version and minor version. Class file format is assigned when you compile source file and its depends on JDK version used to compile.  
4) Its always best practice to use same version of java for compilation and execution to avoid any chance of UnSupportedClassVersionError.  
5) *UnSupportedClassVersionError* is not related to [java classpath](http://javarevisited.blogspot.com/2011/01/how-classpath-work-in-java.html) , so don't confuse this with NoClassDefFoundError or ClassNotFoundException.

Major Class Versions of Various JDK

Following are the major version of *class file format* in standard JDK environment.  
  
JDK 1.1 = 45  
JDK 1.2 = 46  
JDK 1.3 = 47  
JDK 1.4 = 48  
JDK 1.5 = 49  
JDK 1.6 = 50  
  
You can also get version of "javac" (used for compilation) and version of "java" (used for execution) as below  
**C:\equity trading\stocks>javac -version**  
javac 1.6.0-beta2  
  
**C:\equity trading\stocks>java -version**  
java version "1.6.0-beta2"  
Java(TM) SE Runtime Environment (build 1.6.0-beta2-b86)  
Java HotSpot(TM) Client VM (build 1.6.0-beta2-b86, mixed mode, sharing)  
  
Now you can identify your JDK version based on class file format version whenever you see java.lang.UnsupportedClassVersionError :)  
  
So next time when you see **UnsupportedClassVersionError** don't be afraid and follow the best approach based upon your need.  
Read more: <http://javarevisited.blogspot.com/2011/07/javalangunsupportedclassversionerror.html#ixzz46oKASK2E>

# Does Java pass by value or pass by reference

Does Java is pass by value or pass by reference is one of the [tricky Java question](http://java67.blogspot.com/2012/09/top-10-tricky-java-interview-questions-answers.html) mostly asked on fresher level interviews. Before debating whether Java is pass by value or pass by reference lets first clear what is pass by value and what is pass by reference. This question has its origin on C and C++ where you can pass function parameter either value or memory address, where value is stored (pointer). As per Java specification everything in Java is pass by value whether its primitive value or objects and it does make sense because Java doesn't support pointers or pointer arithmetic, Similarly [multiple inheritance](http://javarevisited.blogspot.sg/2011/07/why-multiple-inheritances-are-not.html) and [operator overloading](http://javarevisited.blogspot.sg/2011/08/why-java-does-not-support-operator.html) is also not supported in Java. This question becomes confusing when interviewer ask about how object is passed in Java ? Answer to this question is simple whenever a method parameter expect object, reference of that object is passed. Many programmer confuses reference with pointers here which is not correct, reference is a kind of handle which is used to locate object or change object, but it doesn’t allows any pointer arithmetic i.e. you can not increase or decrease memory address and locate a different [object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html) using reference in Java.

**Pass by Value and Pass by Reference Example in Java**

Let’s see two example of calling method and passing parameter this will clear any doubt whether Java is pass by value or pass by reference. consider following example:

**public** **class** PassByValueExample {  
    
    **public** **static** **void** main(**String** args[]) {  
       **int** number = 3;  
       printNext(number);  
       **System**.out.println("number Inside main(): "+number);  
    }    
    **public** **static** **void** printNext(**int** number){  
        number++;  
        **System**.out.println("number Inside printNext(): "+number);  
    }    
}  
**Output:**  
number Inside printNext(): 4  
number Inside main(): 3

Above example clearly shows that primitives are passed as pass by value to method parameters, had Java pass by reference both [main method](http://javarevisited.blogspot.sg/2011/12/main-public-static-java-void-method-why.html) and printNext() would have printed same value. Now look at another example of passing object as method parameter which will confuse you that Java is pass by reference, which Java is not.

**public** **class** PassByReferenceConfusion {    
    **public** **static** **void** main(**String** args[]) {  
       Car car = **new** Car("BMW");  
       **System**.out.println("Brand of Car Inside main() before: "+ car.brand);  
       printBrand(car);  
       **System**.out.println("Brand of Car Inside main()after: "+ car.brand);  
    }  
    
    **public** **static** **void** printBrand(Car car){  
        car.brand = "Maruti";  
        **System**.out.println("Brand of Car Inside printBrand(): "+car.brand);  
    }    
    **private** **static** **class** Car{  
        **private** **String** brand;        
        **public** Car(**String** brand){  
            **this**.brand = brand;  
        }  
  
    }  
}  
  
**Output:**  
Brand of Car Inside main() before: BMW  
Brand of Car Inside printBrand(): Maruti  
Brand of Car Inside main()after: Maruti

If you see **change made in method parameter is reflected globally** i.e. brand of car is changed in all places it means one object is used in both method. Well in reality if you pass object as method parameter in Java  it passes "value of reference" or in simple term object reference or handle to [Object in Java](http://javarevisited.blogspot.sg/2012/12/what-is-object-in-java-or-oops-example.html). Here reference term is entirely different than reference term used in C and C+ which directly points to memory address of variable and subject to pointer arithmetic. in Java object can only be accessed by its reference as you can not get memory address where object is stored or more precisely there is no method to get value of object by passing memory address.

To conclude everything in Java including primitive and objects are pass by value. In case of object value of reference is passed.

Read more: <http://javarevisited.blogspot.com/2012/12/does-java-pass-by-value-or-pass-by-reference.html#ixzz46oNwFJNv>

# 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 a type is more suitable to represent well known fixed set of things and state,  for example representing the state of Order as NEW, PARTIAL FILL, FILL or CLOSED.   
  
Enumeration(Enum) was not originally available in Java though it was available in another language like C and C++, but eventually, Java realized and introduced Enum on JDK 5 (Tiger) by **keyword Enum**.   
  
In this **Java Enum tutorial**, we will see different *Enum example in Java* and learn using Enum in Java. Focus of this Java Enum tutorial will be on different features provided by Enum in Java and how to use them.   
  
If you have used Enumeration before in C or C++ then 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 then you can also see [Head First Java 2nd Edition](http://www.amazon.com/dp/0596009208/?tag=javamysqlanta-20), 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

Since **Enum in Java** is only available from **Java 1.5** it's worth to discuss how we used to represent enumerable values in Java prior JDK 1.5 and without it. I use public static [final constant](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 serve 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 the answer of all this limitation. Enum in Java is type-safe, provides meaningful String names and has their own namespace. Now let's see the same example using Enum in Java:

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

Here Currency is our **enum** and PENNY, NICKLE, DIME, QUARTER are **enum constants**. Notice **curly braces around enum constants** because Enum is a type like [class](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 the similar naming convention for enum like class and interface (first letter in Caps) and since *Enum constants are implicitly static final* we have used all caps to specify them like Constants in Java.  
  
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 a 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 cannot change their value once created. Enum in Java provides type-safety and can be used inside switch statement like int variables.   
  
Since enum is a keyword you can not use as a variable name and since it’s only introduced in JDK 1.5 all your previous code which has an enum as a variable name will not work and needs to be refactored.

Benefits of using Enums in Java

1) **Enum is type-safe** you can not assign anything else other than predefined Enum constants to an Enum variable. It is a compiler error to assign something else, unlike the public static final variables used in [Enum int pattern](http://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html" \t "_blank) and [Enum String pattern](http://java67.blogspot.com/2015/03/top-40-core-java-interview-questions-answers-telephonic-round.html" \t "_blank).  
  
2) Enum has its own namespace.  
  
3) The 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 an example of [using java enum in switch statement](http://java67.blogspot.com/2012/09/how-to-use-java-enum-in-switch-case-example.html) in this java enum tutorial.  
  
4) Adding new constants on [Enum in Java](http://java67.blogspot.com/2014/04/what-java-developer-should-know-about-Enumeration-type-in-Java.html) is easy and you can add new constants without breaking the existing code.

Important points about Enum in Java

1) **Enums in Java are type-safe** and has their own namespace. 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 types**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;

}

};

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

**Currency** usCoin **=** **Currency.DIME**;

**switch** (usCoin) {

**case** **PENNY:**

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

**break**;

**case** **NICKLE:**

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

**break**;

**case** **DIME:**

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

**break**;

**case** **QUARTER:**

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

}

from JDK 7 onwards 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 "==", the equality operator** as shown in following example of  Java Enum:

**Currency** usCoin **=** **Currency.DIME**;

**if**(usCoin **==** **Currency.DIME**){

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

}

By the way comparing objects using == operator is not recommended, Always use [equals() method](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.  
  
If you are not convinced than you should read [this article](http://javarevisited.blogspot.com/2013/04/how-to-compare-two-enum-in-java-equals.html) to learn more about pros and cons of comparing two enums using equals() vs == operator in Java.

7) Java compiler automatically generates static [values() method](http://java67.blogspot.com/2013/03/how-to-iterate-over-java-enum-using.html) 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 is exactly the same **as defined order in the Enum**.

8) In Java, [Enum can override methods](http://java67.blogspot.com/2012/08/how-to-convert-enum-to-string-in-java.html) also. Let’s see an example of overriding toString() method **inside Enum in Java** to provide a **meaningful description** for enums constants.

**public** **enum** Currency {

.**......**.

**@Override**

**public** **String** **toString**() {

**switch** (this) {

**case** **PENNY:**

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

**break**;

**case** **NICKLE:**

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

**break**;

**case** **DIME:**

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

**break**;

**case** **QUARTER:**

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

}

**return** super**.**toString();

}

};

And here is how it looks like when displayed:

**Currency** usCoin **=** **Currency.DIME**;

**System.**out**.**println(usCoin);

**Output:**

**Dime:** 10  
        
9) Two new collection classes **[EnumMap](http://javarevisited.blogspot.com/2012/09/what-is-enummap-in-java-example-tutorial.html) and [EnumSet](http://javarevisited.blogspot.com/2014/03/how-to-use-enumset-in-java-with-example.html)**are added into collection package to **support Java Enum**. These classes are a 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.  
  
EnumSet doesn't have any public constructor instead it provides factory methods to create instance e.g. EnumSet.of() methods. This design allows EnumSet to internally choose between two different implementations depending upon the size of Enum constants.  
  
If Enum has less than 64 constants than EnumSet uses RegularEnumSet class which internally uses a long variable to store those 64 Enum constants and if Enum has more keys than 64 then it uses JumboEnumSet. See my article the [difference between RegularEnumSet and JumboEnumSet](http://java67.blogspot.com/2013/11/difference-between-regularenumset-and-jumboenumset-java.html) for more details.  
  
  
10)**You can not create an instance of enums by using new operator** in Java because the [constructor of Enum in Java can only be private](http://java67.blogspot.com/2012/11/java-enum-example-with-constructor.html) and Enums constants can only be created inside Enums itself.  
  
  
11) An instance of Enum in Java is created when any Enum constants are first called or referenced in code.

12) **Enum in Java can implement the interface** and override any method like normal class It’s also worth noting that Enum in java implicitly implements both [Serializable](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 a 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 {

**PENNY**(1) {

**@Override**

**public** **String** **color**() {

**return** "copper";

}

},

**NICKLE**(5) {

**@Override**

**public** **String** **color**() {

**return** "bronze";

}

},

**DIME**(10) {

**@Override**

**public** **String** **color**() {

**return** "silver";

}

},

**QUARTER**(25) {

**@Override**

**public** **String** **color**() {

**return** "silver";

}

};

**private** **int** value;

**public** **abstract** **String** **color**();

**private** **Currency**(**int** *value*) {

this**.**value **=** value;

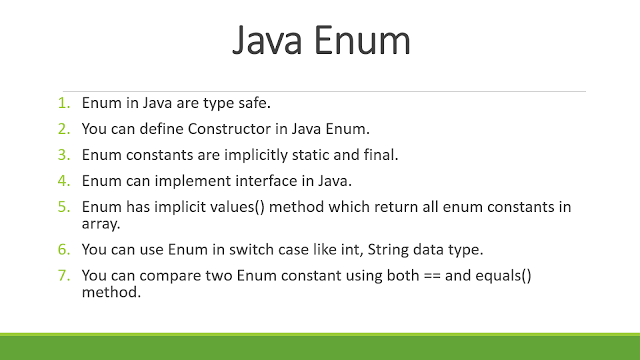
}

}

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

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

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

[](http://java67.blogspot.com/2013/07/15-java-enum-interview-questions-amswers-for-experienced-programmers.html)

Real world Examples of Enum in Java

So far you have learned what Enum can do for you in Java. You learned that enum can be used to represent well known fixed set of constants,  [enum can implement interfac](http://java67.blogspot.com/2013/07/java-enum-code-example-softdrinks-how.html" \t "_blank)e, it can be used in switch case like int, short and String and Enum has so many useful built-in metods like values(), vlaueOf(), name(), and ordinal(), but we didn't learn where to use the Enum in Java?   
  
I think some real world examples of enum will do a lot of good to many pepole and that's why I am going to summarize some of the popular usage of Enum in Java world below.   
  
  
**Enum as Thread Safe Singleton**  
One of the most popular use of Java Enum is to impelment the Singleton design pattern in Java. In fact, Enum is the easieset way to create a [thread-safe Singleton in Java](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html). It offer so many advantage over traditional implementation using class e.g. built-in Serialization, guarantee that Singleton will always be Singleton and many more. I suggest you to check my article about [Why Enum as Singelton is better in Java](http://javarevisited.blogspot.com/2012/07/why-enum-singleton-are-better-in-java.html) to larn more on this topic.   
  
  
**Strategy Pattern using Enum**  
You can also implement the Strategy design pattern using Enumeration type in Java. Since Enum can implement interface, it's a good candidate to implement the [Strategy interface](http://java67.blogspot.com/2014/12/strategy-pattern-in-java-with-sample.html) and define individual strategy. By keeping all related Strategy in one place, Enum offer better maintainence support. It also doesn't break the open closed design principle as per se because any error will be detected at compile time. See this [tutorial](http://javarevisited.blogspot.com/2014/11/strategy-design-pattern-in-java-using-Enum-Example.html)to learn how to implement Strategy pattern using Enum in Java.  
  
  
**Enum as replacement of Enum String or int pattern**  
There is now no need to use String or integer constant to represent fixed set of things e.g. status of object like ON and OFF for a button or START, IN PROGRESS and DONE for a Task. Enum is much better suited for those needs as it provide compile time type safety and better debugging assistent than String or Integer.  
  
  
**Enum as State Machine**  
You can also use Enum to impelment State machine in Java. A State machine transition to predifine set of states based upon current state and given input. Since Enum can implement interface and override method, you can use it as State machine in Java. See this [tutorial](http://vanillajava.blogspot.sg/2011/06/java-secret-using-enum-as-state-machine.html)from Peter Lawrey for a working example.  
  
 **Enum Java valueOf example**  
One of my readers pointed out that I have not mentioned about the valueOf method of enum in Java, which is used to [convert String to enum in Java](http://java67.blogspot.com/2012/08/how-to-create-enum-from-string-in-java.html).  
  
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 an 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 the position of a Enum constant as they declared in enum while name()of Enum returns the exact string which is used to create that particular Enum constant.” name() method can also be used for [converting Enum to String in Java](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. That advice will give you more idea of using this powerful feature of Java programming language  
  
Read more: <http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html#ixzz47TvMGNBX>