1)Difference between final,finally,finalize?

\*\*final is a modifier. If a class is final we cannot inherit from that, if a method is final we cannot override that and if a variable is final that becomes constant and we cannot reassign it.

\*\*finally is associated with try catch block to maintain cleanup code.

\*\*finalize is a method which is always invoked by garbage collector on the object just before destroying an object to perform cleanup activities.

2)Difference between String and StringBuffer ?

|  |  |
| --- | --- |
| String | StringBuffer |
| String s=new String(“Durga”); s.concat(“Software”) sop(s);  A new object will be created when we try to concat on s and s will point to new object.  So string is immutable. | StringBuffer sb=new StringBuffer(“Durga”);  Sb.append(“Software”);sop(sb);  No new object will be created when we call append.  So it is immutable. |

3)Difference between == and .equals() in java ::🡪

We use == for reference comparison whereas .equals() method is used for content comparison.

String s1=new String(“Durga”);

String s2=new String(“Durga”);

Sop(s1==s2) //returns false;

Sop(s1.equals(s2)) //returns true;

\*\*The equals method is defined in the Object class, from which every class is either a direct or indirect descendant. By *default*, the equals() method actually behaves the same as the “==” operator – meaning it checks to see if both objects reference the same place in memory. But, the equals method is actually meant to compare the contents of 2 objects, and not their location in memory.

\*\*So, how is that behavior actually accomplished? Simple – the equals class is overridden to get the desired functionality whereby the object contents are compared instead of the object locations. This is the Java best practice for overriding the equals method – you should compare the values inside the object to determine equality.

\*\*In String class, all wrapper class and all collection classes .equals() method is overridden for content comparison.

4) Modifier available in java ?

12 modifiers are there in java.

5)StringBuffer vs StringBuilder ?

StringBuffer and StringBuilder are same including constructor but following are the difference

|  |  |
| --- | --- |
| StringBuffer | StringBuilder |
| StringBuffer object every method is synchronized | No method is synchronized |
| ThreadSafe but not performancewise good. | Not threadsafe and performanewise good. |
| Introduced in 1.0 version | Introduced in 1.5 version |

6) String vs StringBuffer vs StringBuilder?

|  |  |  |
| --- | --- | --- |
| String | StringBuffer | StringBuilder |
| If the content is fixed and won’t change frequently then use String. | If content is not fixed keeps changing but the thread safety is required then use StringBuffer. | If content is not fixed keeps changing and the thread safety is not required then use StringBuilder. |

7) Interface vs Abstact Class vs Concrete Class?

|  |  |  |
| --- | --- | --- |
| Interface | Abstract Class | Concrete Class |
| If we don’t know anything about implementation but we have requirement specification we go for Interface. | If we are talking about implementation but not completely then we go for Abstract Class. | We are talking about implementation completely and ready to provide service then we go for Concrete Class. |

8)Access Specifier vs Access Modifier?

|  |  |
| --- | --- |
| Access Specifier | Access Modifier |
| In java there is no terminology like Access Specifiers. | In java all are considered as Modifier only. |
| In old languages like c++ public,private,protected,default these are called Access Specifier. |  |

9)Interface vs Abstract Class?

|  |  |
| --- | --- |
| Interface | Abstract Class |
| If we don’t know anything about implementation but we have requirement specification we go for Interface. | If we are talking about implementation but not completely then we go for Abstract Class. |
| Every method inside it is abstract and public by default whether we declare or not. | No such boundary. |
| We can’t declare interface methods with these modifiers :🡪private,protected,final,synchronized,static,native,strictfp | No restriction with abstract class modifiers. |
| Every variable is by default public,static and final | No such restrictions. |
| We can’t declare interface variables with these modifier :🡪  private,protected,volatile,transient. | No such restrictions. |
| It is mandatory to initialize at the time of declaration interface variables otherwise we will get compile time error. | No such restriction. |
| We cannot declare instance block/static block inside interface otherwise will get compiler time error | No such restriction. |
| Inside interface we cannot declare constructor. | No such restriction. |

10) What does System.out.println() do ?

System is a class in java.lang package.

Static variable out of type PrintStream is in System class.

Println() is a method of PrintStream class.

Class System{

Static PrintStream out;

}

11) Public static void main(String[] args) explain ?

Whether class contains main() method or not this won’t be checked in compile time. At runtime if JVM is unable to find required main() method then we will get runtime exception saying **NoSuchMethodError:main.**

**\*public🡪**to call by JVM from anywhere.

\*static🡪without any existing object main method has to be called by JVM and main is not related to any object.

\*void🡪main method won’t return anything to JVM.

\*main🡪this is the name which is configured inside JVM which should be called first.

\*String[] args🡪 command line arguments.

The above syntax we have to strictly followed else we will get error **NoSuchMethodError:main.**

But following changes can be acceptable🡪

a)instead of “public static” we can use “static public” also.

b)we can declare String[] array in any format incase of String[] args.

c)we can replace main(String[] args)🡺 main(String… args);

d)We can declare main method with the following modifiers also🡪final,synchronized,strictfp.

\*\*\*If main method syntax is not correct we won’t get compile time error we will get runtime error.

CASE 1:🡪 overloading of the main method is possible but JVM is always going to call the method with String arguments. Other method we can call explicitly like normal method call.

CASE2:🡪Inheritance concept is applicable to main method .While executing child class if child class doesnot contain main method then parent class’s main method will be called.

CASE 3:🡪we can have the same syntax of main method in the child class as in the parent class but it will be method hiding not overriding as it is static.

**Some enhancement of main method in 1.7v onwards:🡪**

a)If a class doesn’t contain main method and we try to run it then upto 1.6 version we use to get error like **NoSuchMethodError:main.** But from 1.7 onwards we will get more meaningful error information.

b)From 1.7 v onwards to run a java program main method is mandatory. Hence, even though class contain static blocks they won’t be executed if the class doesn’t contain main method.

c)If the class contains main() method whether it is 1.6 or 1.7 version there is no change in execution sequence i.e first static block then main .

d) from 1.7 version we cannot run the code without main method.So to print anything in console we should have main method mandatorily.

12)Overloading vs Overridding

\*\*Overloading is method with same name but different signature. This is called static polymorphism.In overloading the method return type doesnot matter. it can be same/different.

\*\*Overridding is when child class override/change the implementation of the parent class method. This is dynamic polymorphism. In overriding the return type of the child class method must be same as parent class method’s return type or child class of the parent class method’s return type.

For private,static and final method overriding concept is not there. For static method method hiding is there instead of overriding.

\*\*for overloading no restriction on access specifier but in case of overriding child class’s method should have same access specifier as parent’s method or it can be less restrictive than the parent’s method’s specifier.

\*\*If the child class’s method throw checked exception then parent class’s method should throw the same exception or the parent of the exception class.

\*\*If we use the parent class reference to hold child class object then the child class’s uniquely defined method we can’t call by parent class reference.

13)Singletone Class and FactoryMethod??

**Singletone Class Pattern**

In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), the **singleton pattern** is a [design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) that restricts the [instantiation](https://en.wikipedia.org/wiki/Instantiation_(computer_science)) of a class to one [object](https://en.wikipedia.org/wiki/Object_(computer_science)). This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the [mathematical concept of a singleton](https://en.wikipedia.org/wiki/Singleton_(mathematics)).

**1)Lazy initialization**

*// Single-threaded version*

**class** **Foo** {

**private** Helper helper;

**public** Helper getHelper() {

**if** (helper == **null**) {

helper = **new** Helper();

}

**return** helper;

}

*// other functions and members...*

}

The problem is that this does not work when using multiple threads. A [lock](https://en.wikipedia.org/wiki/Lock_(computer_science)) must be obtained in case two threads call getHelper() simultaneously. Otherwise, either they may both try to create the object at the same time, or one may wind up getting a reference to an incompletely initialized object.

The lock is obtained by expensive synchronizing, as is shown in the following example.

*// Correct but possibly expensive multithreaded version*

**class** **Foo** {

**private** Helper helper;

**public** **synchronized** Helper getHelper() {

**if** (helper == **null**) {

helper = **new** Helper();

}

**return** helper;

}

*// other functions and members...*

}

However, the first call to getHelper() will create the object and only the few threads trying to access it during that time need to be synchronized; after that all calls just get a reference to the member variable. Since synchronizing a method could in some extreme cases decrease performance by a factor of 100 or higher,[[3]](https://en.wikipedia.org/wiki/Double-checked_locking" \l "cite_note-Boehm2005-3) the overhead of acquiring and releasing a lock every time this method is called seems unnecessary: once the initialization has been completed, acquiring and releasing the locks would appear unnecessary. Many programmers have attempted to optimize this situation in the following manner:

1. Check that the variable is initialized (without obtaining the lock). If it is initialized, return it immediately.
2. Obtain the lock.
3. Double-check whether the variable has already been initialized: if another thread acquired the lock first, it may have already done the initialization. If so, return the initialized variable.
4. Otherwise, initialize and return the variable.

*// Broken multithreaded version*

*// "Double-Checked Locking" idiom*

**class** **Foo** {

**private** Helper helper;

**public** Helper getHelper() {

**if** (helper == **null**) {

**synchronized**(**this**) {

**if** (helper == **null**) {

helper = **new** Helper();

}

}

}

**return** helper;

}

*// other functions and members...*

}

Intuitively, this algorithm seems like an efficient solution to the problem. However, this technique has many subtle problems and should usually be avoided. For example, consider the following sequence of events:

1. Thread *A* notices that the value is not initialized, so it obtains the lock and begins to initialize the value.
2. Due to the semantics of some programming languages, the code generated by the compiler is allowed to update the shared variable to point to a partially constructed object before *A* has finished performing the initialization. For example, in Java if a call to a constructor has been inlined then the shared variable may immediately be updated once the storage has been allocated but before the inlined constructor initializes the object.[[4]](https://en.wikipedia.org/wiki/Double-checked_locking#cite_note-IBM-4)
3. Thread *B* notices that the shared variable has been initialized (or so it appears), and returns its value. Because thread *B* believes the value is already initialized, it does not acquire the lock. If *B* uses the object before all of the initialization done by *A* is seen by *B* (either because *A* has not finished initializing it or because some of the initialized values in the object have not yet percolated to the memory *B* uses ([cache coherence](https://en.wikipedia.org/wiki/Cache_coherence))), the program will likely crash.

One of the dangers of using double-checked locking in [J2SE 1.4](https://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition) (and earlier versions) is that it will often appear to work: it is not easy to distinguish between a correct [implementation](https://en.wikipedia.org/wiki/Implementation) of the technique and one that has subtle problems. Depending on the [compiler](https://en.wikipedia.org/wiki/Compiler), the interleaving of threads by the [scheduler](https://en.wikipedia.org/wiki/Scheduling_(computing)) and the nature of other [concurrent system activity](https://en.wikipedia.org/wiki/Concurrency_(computer_science)), failures resulting from an incorrect implementation of double-checked locking may only occur intermittently. Reproducing the failures can be difficult.

As of [J2SE 5.0](https://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition), this problem has been fixed. The [volatile](https://en.wikipedia.org/wiki/Volatile_variable) keyword now ensures that multiple threads handle the singleton instance correctly. This new idiom is described in [[3]](http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html) and [[4]](http://www.oracle.com/technetwork/articles/javase/bloch-effective-08-qa-140880.html).

*// Works with acquire/release semantics for volatile*

*// Broken under Java 1.4 and earlier semantics for volatile*

**class** **Foo** {

**private** **volatile** Helper helper;

**public** Helper getHelper() {

Helper result = helper;

**if** (result == **null**) {

**synchronized**(**this**) {

result = helper;

**if** (result == **null**) {

helper = result = **new** Helper();

}

}

}

**return** result;

}

*// other functions and members...*

}

Note the local variable result, which seems unnecessary. This ensures that in cases where helper is already initialized (i.e., most of the time), the volatile field is only accessed once (due to "return result;" instead of "return helper;"), which can improve the method's overall performance by as much as 25 percent.[[5]](https://en.wikipedia.org/wiki/Double-checked_locking#cite_note-5)

**2)Eager initialization**

If the program will always need an instance, or if the cost of creating the instance is not too large in terms of time/resources, the programmer can switch to eager initialization, which always creates an instance:

**public** **class** **Singleton** {

**private** **static** **final** Singleton INSTANCE = **new** Singleton();

**private** Singleton() {}

**public** **static** Singleton getInstance() {

**return** INSTANCE;

}

}

This method has a number of advantages:

* The static initializer is run when the class is initialized, after class loading but before the class is used by any thread.
* There is no need to synchronize the getInstance() method, meaning all threads will see the same instance and no (expensive) locking is required.
* The final keyword means that the instance cannot be redefined, ensuring that one (and only one) instance ever exists.

**3)Static block initialization**[[edit](https://en.wikipedia.org/w/index.php?title=Singleton_pattern&action=edit&section=7)]

Some authors refer to a similar solution allowing some pre-processing (e.g. for error-checking).[[6]](https://en.wikipedia.org/wiki/Singleton_pattern#cite_note-6) In this sense, the traditional approach could be seen as a particular case of this one, as the class loader would do exactly the same processing.

**public** **class** **Singleton** {

**private** **static** **final** Singleton instance;

**static** {

**try** {

instance = **new** Singleton();

} **catch** (Exception e) {

**throw** **new** RuntimeException("Darn, an error occurred!", e);

}

}

**public** **static** Singleton getInstance() {

**return** instance;

}

**private** Singleton() {

*// ...*

}

}

**4)The enum way**[[edit](https://en.wikipedia.org/w/index.php?title=Singleton_pattern&action=edit&section=9)]

In the second edition of his book *Effective Java*, [Joshua Bloch](https://en.wikipedia.org/wiki/Joshua_Bloch) claims that "a single-element enum type is the best way to implement a singleton"[[8]](https://en.wikipedia.org/wiki/Singleton_pattern#cite_note-8) for any Java that supports [enums](https://en.wikipedia.org/wiki/Enums" \o "Enums). The use of an enum is very easy to implement and has no drawbacks regarding serializable objects, which have to be circumvented in the other ways.

**public** **enum** Singleton {

INSTANCE;

**public** void execute (String arg) {

*// Perform operation here*

}

}

The public method can be written to take any desired types of arguments; a single String argument is used here as an example.

This approach implements the singleton by taking advantage of Java's guarantee that any enum value is instantiated only once in a Java program. Since Java enum values are globally accessible, so is the singleton, initialized lazily by the class loader. The drawback is that the enum type is somewhat inflexible.

public enum MySingleton {

INSTANCE;

}

has an implicit empty constructor. Let's be explicit instead,

public enum MySingleton {

INSTANCE;

private MySingleton() {

System.out.println("Here");

}

}

If you then added another class with a main() method like

public static void main(String[] args) {

System.out.println(MySingleton.INSTANCE);

}

You would see

Here

INSTANCE

enum fields are compile time constants, but they are instances of their enum type. And, they're constructed when the enum type is referenced for the first time.

# 14) Why always override hashcode() if overriding equals()?

In Java, every object has access to the equals() method because it is inherited from theObject class. However, this default implementation just simply compares the memory addresses of the objects. You can override the default implementation of the equals() method defined in java.lang.Object. If you override the equals(), you MUST also override hashCode(). Otherwise a violation of the general contract for Object.hashCode will occur, which can have unexpected repercussions when your class is in conjunction with all hash-based collections.

Here is the contract, copied from the java.lang.Object specialization:

### public int hashCode()

Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable.

The general contract of hashCode is:

* Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
* **If two objects are equal according to the equals(Object) method, then calling thehashCode method on each of the two objects must produce the same integer result.**
* It is not required that if two objects are unequal according to the equals(java.lang.Object)method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

The default implementation of equals() method checks to see if the two objects have the same identity. Similarly, the default implementation of the hashCode() method returns an integer based on the object's identity and is not based on the values of instance (and class) variables of the object. No matter how many times the values of its instance variables (data fields) change, the hash code calculated by the default hashCode implementation does not change during the life of the object.

Consider the following code, we have overridden equals() method to check if two objects are equal based on the values of their instance variables. Two objects may be stored at different memory addresses but may still be equal base on their instance variable.

public class CustomerID {  
 private long crmID;  
 private int nameSpace;  
  
 public CustomerID(long crmID, int nameSpace) {  
 super();  
 this.crmID = crmID;  
 this.nameSpace = nameSpace;  
 }  
  
 public boolean equals(Object obj) {  
 //null instanceof Object will always return false  
 if (!(obj instanceof CustomerID))  
 return false;  
 if (obj == this)  
 return true;  
 return this.crmID == ((CustomerID) obj).crmID &&  
 this.nameSpace == ((CustomerID) obj).nameSpace;  
 }  
  
 public static void main(String[] args) {  
 Map m = new HashMap();  
 m.put(new CustomerID(2345891234L,0),"Jeff Smith");  
 System.out.println(m.get(new CustomerID(2345891234L,0)));  
 }  
  
}

Compile and run the above code, the output result is

null

What is wrong? The two instances of CustomerID are logically equal according to the class'sequals method. Because the hashCode() method is not overridden, these two instances' identities are not in common to the default hashCode implementation. Therefore, theObject.hashCode returns two seemingly random numbers instead of two equal numbers. Such behavior violates "Equal objects must have equal hash codes" rule defined in the hashCode contract.

Let's provide a simple hashCode() method to fix this problem:

public class CustomerID {  
 private long crmID;  
 private int nameSpace;  
  
 public CustomerID(long crmID, int nameSpace) {  
 super();  
 this.crmID = crmID;  
 this.nameSpace = nameSpace;  
 }  
  
 public boolean equals(Object obj) {  
 //null instanceof Object will always return false  
 if (!(obj instanceof CustomerID))  
 return false;  
 if (obj == this)  
 return true;  
 return this.crmID == ((CustomerID) obj).crmID &&  
 this.nameSpace == ((CustomerID) obj).nameSpace;  
 }  
  
 public int hashCode() {  
 int result = 0;  
 result = (int)(crmID/12) + nameSpace;  
 return result;  
 }  
  
 public static void main(String[] args) {  
 Map m = new HashMap();  
 m.put(new CustomerID(2345891234L,0),"Jeff Smith");  
 System.out.println(m.get(new CustomerID(2345891234L,0)));  
 }  
  
}

Compile and run the above code, the output result is

Jeff Smith

The hashcode distribution for instances of a class should be random. This is exactly what is meant by the third provision of the hashCode contract. Write a correct hashCode method is easy, but to write an effective hashCode method is extremely difficult.

For example, From [How to Avoid Traps and Correctly Override Methods From java.lang.Object](http://www.javaworld.com/javaworld/jw-01-1999/jw-01-object.html): If you are unsure how to implement hashCode(), just always return 0 in your implementations. So all of your custom objects will return the same hash code. Yes, it turns hashtable of your objects into one (possibly) long linked-list, but you have implemented hashCode() correctly!

public int hashCode(){  
 return 0;  
}

It's legal because it ensures that equal objects have the same hash code, but it also indicates that every object has the same hash code. So every object will be hashed into the same bucket, and hash tables degenerate to linked lists. The performance is getting worse when it needs to process a large number of objects. How to implement a good hash function is a big topic and we will not cover here.

(Have To Study Factory Methods)

\*\* Fail-safe vs Fail-fast iterator :🡪

Difference between Fail fast and fail safe iterator  or  Fail fast vs Fail Safe iterator is one of those questions which are  used to test your knowledge about the topic Concurrency.  
Before we discuss in detail about fail safe iterator and fail fast iterator in addition to  their comparison , we should understand the term *Concurrent Modification*.  
  
**What is Concurrent Modification ?**  
  
When one or more thread is iterating over the collection, in between, one thread changes the structure of the collection (either adding the element to the collection or by deleting the element in the collection or by updating the value at particular position in the collection) is known as Concurrent Modification  
  
**Difference between Fail Fast iterator and Fail Safe iterator**  
  
**Fail fast Iterator**  
  
Fail fast iterator while iterating through the collection , instantly throws Concurrent Modification Exception if there is structural modification  of the collection . Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.   
  
Fail-fast iterator can throw ConcurrentModificationException in two scenarios :

*Single Threaded Environment*  
   
After the creation of the iterator , structure is modified at any time by any method other than iterator's own remove method.   
   
*Multiple Threaded Environment*  
  
 If one thread is modifying the structure of the collection while other thread is iterating over it .  
  
  
According to  [Oracle docs](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html" \t "_blank) , **the fail-fast behavior of an iterator cannot be guaranteed** as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness:**the fail-fast behavior of iterators should be used only to detect bugs.**  
  
  
**Interviewer : How  Fail  Fast Iterator  come to know that the internal structure is modified ?**  
Iterator read internal data structure (object array) directly . The internal data structure(i.e object array) should not be modified while iterating through the collection. To ensure this it maintains an internal  flag *"mods" .*Iterator checks the *"mods" flag*whenever it gets the next value (using hasNext() method and next() method). Value of*mods* flag changes whenever there is an structural modification. Thus indicating iterator to throw ConcurrentModificationException.  
  
  
**Fail Safe Iterator :**  
  
Fail Safe Iterator makes copy of the internal data structure (object array) and iterates over the copied data structure.Any structural modification done to the iterator affects the copied data structure.  So , original data structure remains  structurally unchanged .Hence , no ConcurrentModificationException throws by the fail safe iterator.  
  
Two  issues associated with Fail Safe Iterator are :  
  
1. Overhead of maintaining the copied data structure i.e memory.  
  
2.  Fail safe iterator does not guarantee that the data being read is the data currently in the original data structure.   
  
According to [Oracle docs](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html) , fail safe iterator is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber mutations, and is useful when you cannot or don’t want to synchronize traversals, yet need to preclude interference among concurrent threads. The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created. This **array never changes during the lifetime of the iterator, so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException**.The iterator will not reflect additions, removals, or changes to the list since the iterator was created. Element-changing operations on iterators themselves (remove(), set(), and add()) are not supported. These methods throw UnsupportedOperationException.  
  
  
  
**Example of Fail Fast Iterator and Fail Safe Iterator**

**import** **java.util.HashMap**;

**import** **java.util.Iterator**;

**import** **java.util.Map**;

**public** **class** **FailFastExample**

{

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

{

Map<String,String> premiumPhone = **new** HashMap<String,String>();

premiumPhone.put("Apple", "iPhone");

premiumPhone.put("HTC", "HTC one");

premiumPhone.put("Samsung","S5");

Iterator iterator = premiumPhone.keySet().iterator();

**while** (iterator.hasNext())

{

System.out.println(premiumPhone.get(iterator.next()));

premiumPhone.put("Sony", "Xperia Z");

}

}

}

**Output :**

iPhone

Exception in thread "main" java.util.ConcurrentModificationException

at java.util.HashMap$HashIterator.nextEntry(Unknown Source)

at java.util.HashMap$KeyIterator.next(Unknown Source)

at FailFastExample.main(FailFastExample.java:**20**)

**Fail Safe Iterator Example :**

**import** **java.util.concurrent.ConcurrentHashMap**;

**import** **java.util.Iterator**;

**public** **class** **FailSafeExample**

{

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

{

ConcurrentHashMap<String,String> premiumPhone =

**new** ConcurrentHashMap<String,String>();

premiumPhone.put("Apple", "iPhone");

premiumPhone.put("HTC", "HTC one");

premiumPhone.put("Samsung","S5");

Iterator iterator = premiumPhone.keySet().iterator();

**while** (iterator.hasNext())

{

System.out.println(premiumPhone.get(iterator.next()));

premiumPhone.put("Sony", "Xperia Z");

}

}

}

**Output :**

S5

HTC one

iPhone

**Recap : Difference between Fail Fast Iterator and Fail Safe Iterator**

|  |  |  |
| --- | --- | --- |
|  | **Fail Fast Iterator** | **Fail Safe Iterator** |
| Throw ConcurrentModification Exception | Yes | No |
| Clone object | No | Yes |
| Memory Overhead | No | Yes |
| Examples | HashMap,Vector,ArrayList,HashSet | CopyOnWriteArrayList, ConcurrentHashMap |

# [How to create Immutable Class and Object in Java - Tutorial Example](http://javarevisited.blogspot.in/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html)

Writing or creating immutable classes in Java is becoming popular day by day, because of concurrency and multithreading advantage provided by immutable objects. Immutable objects offers several benefits over conventional mutable object, especially while creating concurrent Java application. Immutable object not only guarantees safe publication of object’s state, but also can be shared among other threads without any external [synchronization](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html). In fact JDK itself contains several immutable classes like [String](http://javarevisited.blogspot.com/2012/03/how-to-compare-two-string-in-java.html), [Integer](http://javarevisited.blogspot.com/2011/08/convert-string-to-integer-to-string.html) and other wrapper classes. For those, who doesn’t know what is immutable class or object, Immutable objects are those, whose state can not be changed once created e.g. java.lang.String, once created can not be modified e.g. trim, uppercase, lowercase. All modification in String result in new object, see [why String is immutable in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html) for more details. In this Java programming tutorial, we will learn, how to write immutable class in Java or how to make a class immutable. By the way making a class immutable is not difficult on code level, but its the decision to make, which class mutable or immutable which makes difference. I also suggest reading, Java Concurrency in Practice to learn more about concurrency benefit offered by Immutable object.

## What is immutable class in Java

[What is Immutable class and object, how to create Immutable in Java with example](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)As said earlier, Immutable classes are those class, whose [object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html) can not be modified once created, it means any modification on immutable object will result in another immutable object. best example to understand immutable and mutable objects are, [String and StringBuffer](http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html). Since String is immutable class, any change on existing string object will result in another string e.g. replacing a character into String, [creating substring from String](http://javarevisited.blogspot.in/2011/10/how-substring-in-java-works.html), all result in a new objects. While in case of mutable object like StringBuffer, any modification is done on object itself and no new objects are created. Some times this immutability of String can also cause security hole, and that the reason [why password should be stored on char array instead of String](http://javarevisited.blogspot.com/2012/03/why-character-array-is-better-than.html).

## How to write immutable class in Java

Despite of few disadvantages, Immutable object still offers several benefits in multi-threaded programming and it’s a great choice to achieve [thread safety](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) in Java code. here are few rules, which helps to make a class immutable in Java :

1. State of immutable object can not be modified after construction, any modification should result in new immutable object.

2. All fields of Immutable class should be final.

3. Object must be properly constructed i.e. object reference must not leak during construction process.

4. Object should be final in order to restrict sub-class for altering immutability of parent class.

By the way, you can still create immutable object by violating few rules, like String has its [hashcode](http://javarevisited.blogspot.com/2011/10/override-hashcode-in-java-example.html) in non final field, but its always guaranteed to be same. No matter how many times you calculate it, because it’s calculated from final fields, which is guaranteed to be same. This required a deep knowledge of Java memory model, and can create subtle [race conditions](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html) if not addressed properly. In next section we will see simple example of writing immutable class in Java. By the way, if your Immutable class has lots of optional and mandatory fields, then you can also use [Builder design pattern](http://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html) to make a class Immutable in Java.

## Immutable Class Example in Java

Here is complete code example of writing immutable class in Java. We have followed simplest approach and all rules for making a class immutable, including it [making class final](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html) to avoid putting immutability at risk due to [Inheritance](http://javarevisited.blogspot.com/2012/10/what-is-inheritance-in-java-and-oops-programming.html) and [Polymorphism](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html).

public final class Contacts {

    private final String name;

    private final String mobile;

    public Contacts(String name, String mobile) {

        this.name = name;

        this.mobile = mobile;

    }

    public String getName(){

        return name;

    }

    public String getMobile(){

        return mobile;

    }

}

This Java class is immutable, because its state can not be changed once created. You can see that all of it’s fields are final. This is one of the most simple way of creating immutable class in Java, where all fields of class also remains immutable like String in above case. Some time you may need to write immutable class which includes mutable classes like [java.util.Date](http://javarevisited.blogspot.com/2012/04/difference-between-javautildate-and.html), **despite storing Date into final field it can be modifiedinternally,** if internal date is returned to the client. In order to preserve immutability in such cases, its advised to **return copy of original object**, which is also one of the [Java best practice](http://javarevisited.blogspot.co.uk/2012/08/top-10-jdbc-best-practices-for-java.html). here is another example of making a class immutable in Java, which includes mutable member variable.

public final class ImmutableReminder{

    private final Date remindingDate;

    public ImmutableReminder (Date remindingDate) {

        if(remindingDate.getTime() < System.currentTimeMillis()){

            throw new IllegalArgumentException("Can not set reminder” +

                        “ for past time: " + remindingDate);

        }

        this.remindingDate = new Date(remindingDate.getTime());

    }

    public Date getRemindingDate() {

        return (Date) remindingDate.clone();

    }

}

In above example of creating immutable class, [Date](http://javarevisited.blogspot.com/2011/09/convert-date-to-string-simpledateformat.html) is a mutable object. If getRemindingDate() returns actual Date object than despite remindingDate being final variable, internals of Date can be modified by client code. By returning clone() or copy of remindingDate, we avoid that danger and preserves immutability of class.

## Benefits of Immutable Classes in Java

As I said earlier Immutable classes offers several benefits, here are few to mention:

1) Immutable objects are by default [thread safe](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html), can be shared without synchronization in concurrent environment.

2) Immutable object simplifies development, because its easier to share between multiple threads without external synchronization.

3) Immutable object boost performance of Java application by reducing [synchronization](http://java67.blogspot.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html) in code.  
  
4) Another important benefit of Immutable objects is **reusability**, you can cache Immutable object and reuse them, much like String literals and Integers.  You can use [static factory methods](http://javarevisited.blogspot.it/2011/12/factory-design-pattern-java-example.html) to provide methods like valueOf(), which can return an existing Immutable object from cache, instead of creating a new one.

Apart from above advantages, immutable object has disadvantage of creating garbage as well. Since immutable object can not be reused and they are just a use and throw. String being a prime example, which can create lot of garbage and can potentially slow down application due to [heavy garbage collection](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html), but again that's extreme case and if used properly Immutable object adds lot of value.

That's all on **how to write immutable class in Java**. we have seen rules of writing immutable classes, benefits offered by immutable objects and how we can create immutable class in Java which involves mutable fields. Don’t forget to read more about concurrency benefit offered by Immutable object in one of the best Java book recommended to Java programmers, Concurrency Practice in Java.

**Abstraction vs Encapsulation – Java OOPS**

Abstraction and Encapsulation in Java are two important [Object oriented programming concept](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html) and they are completely different to each other. Only similarity between Abstraction and Encapsulation is that they are OOPS concept, other than that they mean two different things. Abstraction represent taking out the behavior from How exactly its implemented, one example of [abstraction in Java](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html) is interface while Encapsulation means hiding details of implementation from outside world so that when things change no body gets affected. One example of [Encapsulation in Java](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) is private methods; clients  don't care about it, You can change, amend or even remove that method  if that method is not encapsulated and it were public all your clients would have been affected. Apart from this main difference in behavior here are couple of more *differences between Abstraction and Encapsulation in Java*.

## Abstraction vs Encapsulation in Java

Here are some of the main differences between Abstraction vs Encapsulation in Java and OOPS(Object Oriented programming) concept. Abstraction and Encapsulation along with [Inheritance](http://java67.blogspot.sg/2012/08/what-is-inheritance-in-java-oops-programming-example.html) and [polymorphism](http://javarevisited.blogspot.sg/2011/08/what-is-polymorphism-in-java-example.html) forms basis of Object oriented programming in Java.

1) First difference between Abstraction and Encapsulation is that, Abstraction is implemented in Java using [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) and abstract class while Encapsulation is implemented using [private](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html), package-private and protected access modifier.

2) Encapsulation is also called data hiding.

3) Design principles "[programming for interface than implementation](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html)" is based on abstraction and "encapsulate whatever changes" is based upon Encapsulation.

That's all from my side on differences between Abstraction and Encapsulation in Java. Correct understanding of Encapsulation and Abstraction is must for any Java developer. Head first Object oriented Analysis and design is a great book to learn more about Abstraction, Encapsulation and other OOPS concept.