# Java 8 Features with Examples

## Java 8 Features

Some of the important Java 8 features are;

1. [forEach() method in Iterable interface](https://www.journaldev.com/2389/java-8-features-with-examples#iterable-forEach)
2. [default and static methods in Interfaces](https://www.journaldev.com/2389/java-8-features-with-examples#interface-default-static-method)
3. [Functional Interfaces and Lambda Expressions](https://www.journaldev.com/2389/java-8-features-with-examples#functional-interface-lambdas)
4. [Java Stream API for Bulk Data Operations on Collections](https://www.journaldev.com/2389/java-8-features-with-examples#java-stream-api)
5. [Java Time API](https://www.journaldev.com/2389/java-8-features-with-examples#java8-time)
6. [Collection API improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-collection)
7. [Concurrency API improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-concurrency)
8. [Java IO improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-io)
9. [Miscellaneous Core API improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-core)

Let’s have a brief look on these Java 8 features. I will provide some code snippets for better understanding, so if you want to run programs in Java 8, you will have to setup Java 8 environment by following steps.

* [Download JDK8](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html) and install it. Installation is simple like other java versions. JDK installation is required to write, compile and run the program in Java.
* Download latest Eclipse IDE, it provides support for java 8 now. Make sure your projects build path is using Java 8 library.

### forEach() method in Iterable interface

Whenever we need to traverse through a Collection, we need to create an Iterator whose whole purpose is to iterate over and then we have business logic in a loop for each of the elements in the Collection. We might get [ConcurrentModificationException](https://www.journaldev.com/378/java-util-concurrentmodificationexception) if iterator is not used properly.

Java 8 has introduced forEach method in java.lang.Iterable interface so that while writing code we focus on business logic only. forEach method takes java.util.function.Consumer object as argument, so it helps in having our business logic at a separate location that we can reuse. Let’s see forEach usage with simple example.

package com.journaldev.java8.foreach;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

import java.util.function.Consumer;

import java.lang.Integer;

public class Java8ForEachExample {

public static void main(String[] args) {

//creating sample Collection

List<Integer> myList = new ArrayList<Integer>();

for(int i=0; i<10; i++) myList.add(i);

//traversing using Iterator

Iterator<Integer> it = myList.iterator();

while(it.hasNext()){

Integer i = it.next();

System.out.println("Iterator Value::"+i);

}

//traversing through forEach method of Iterable with anonymous class

myList.forEach(new Consumer<Integer>() {

public void accept(Integer t) {

System.out.println("forEach anonymous class Value::"+t);

}

});

//traversing with Consumer interface implementation

MyConsumer action = new MyConsumer();

myList.forEach(action);

}

}

//Consumer implementation that can be reused

class MyConsumer implements Consumer<Integer>{

public void accept(Integer t) {

System.out.println("Consumer impl Value::"+t);

}

}

The number of lines might increase but forEach method helps in having the logic for iteration and business logic at separate place resulting in higher separation of concern and cleaner code.

### default and static methods in Interfaces

If you read forEach method details carefully, you will notice that it’s defined in Iterable interface but we know that interfaces can’t have method body. From Java 8, interfaces are enhanced to have method with implementation. We can use default and static keyword to create interfaces with method implementation. forEach method implementation in Iterable interface is:

default void forEach(Consumer<? super T> action) {

Objects.requireNonNull(action);

for (T t : this) {

action.accept(t);

}

}

We know that Java doesn’t provide [multiple inheritance in Classes](https://www.journaldev.com/1775/multiple-inheritance-in-java) because it leads to **Diamond Problem**. So how it will be handled with interfaces now, since interfaces are now similar to abstract classes. The solution is that compiler will throw exception in this scenario and we will have to provide implementation logic in the class implementing the interfaces.

package com.journaldev.java8.defaultmethod;

@FunctionalInterface

public interface Interface1 {

void method1(String str);

default void log(String str){

System.out.println("I1 logging::"+str);

}

static void print(String str){

System.out.println("Printing "+str);

}

//trying to override Object method gives compile time error as

//"A default method cannot override a method from java.lang.Object"

// default String toString(){

// return "i1";

// }

}

package com.journaldev.java8.defaultmethod;

@FunctionalInterface

public interface Interface2 {

void method2();

default void log(String str){

System.out.println("I2 logging::"+str);

}

}

Notice that both the interfaces have a common method log() with implementation logic.

package com.journaldev.java8.defaultmethod;

public class MyClass implements Interface1, Interface2 {

@Override

public void method2() {

}

@Override

public void method1(String str) {

}

//MyClass won't compile without having it's own log() implementation

@Override

public void log(String str){

System.out.println("MyClass logging::"+str);

Interface1.print("abc");

}

}

As you can see that Interface1 has static method implementation that is used in MyClass.log() method implementation. Java 8 uses **default** and **static** methods heavily in [**Collection API**](https://www.journaldev.com/1260/collections-in-java-tutorial) and default methods are added so that our code remains backward compatible.

If any class in the hierarchy has a method with same signature, then default methods become irrelevant. Since any class implementing an interface already has Object as superclass, if we have equals(), hashCode() default methods in interface, it will become irrelevant. Thats why for better clarity, interfaces are not allowed to have Object class default methods.

For complete details of interface changes in Java 8, please read [Java 8 interface changes](https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method).

## ****Functional Interface****:

The functional interface is an interface with one abstract method, but it can have any number of default and static methods. The runnable and comparable interface are examples of a functional interface.

Example:

public interface TestInterface {

public abstract void show(int n);

public static void hello() {

System.out.println("static method");

}

public default void test() {

System.out.println("default method");

}

}

### @FunctionalInterface Annotation:

We use the **@Functional annotation** with a functional interface. This ensures that the interface can’t have more than one abstract method. If we define more than one abstract method in this interface, then the compiler shows an error ‘**Invalid '@FunctionalInterface' annotation.**’

@FunctionalInterface

public interface TestInterface {

public abstract void show(int n);

public static void hello() {

System.out.println("static method");

}

public default void test() {

System.out.println("default method");

}

}

## Why the Default Method?

Reengineering an existing JDK framework is always very complex. Modifying one interface in a JDK framework breaks all classes that extend the interface, which means that adding any new method could break millions of lines of code. Therefore, default methods have introduced as a mechanism to extend interfaces in a backward-compatible way.

Default methods can be provided to an interface without affecting implementing classes as it includes an implementation. If each added method in an interface is defined with implementation, then no implementing class is affected. An implementing class can override the default implementation provided by the interface.

For Java 8, the JDK collections have been extended and the forEach method is added to the entire collection (which work in conjunction with [lambdas](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)). With the conventional way, the code looks like below:

[?](http://muhammadkhojaye.blogspot.com/2014/03/interface-default-methods-in-java-8.html)

public interface Iterable<T> {

public void forEach(Consumer<? super T> consumer);

}

Since this result each implementing class with compile errors, a default method added with a required implementation in order that the existing implementation should not be changed.

The [Iterable](http://download.java.net/jdk8/docs/api/java/lang/Iterable.html" \t "_blank" \o "Interface Iterable) interface with the Default method is below:

[?](http://muhammadkhojaye.blogspot.com/2014/03/interface-default-methods-in-java-8.html)

public interface Iterable<T> {

public default void forEach(Consumer<? super T> consumer) {

for (T t : this) {

consumer.accept(t);

}

}

}

The same mechanism has been used to adda  [Stream](http://download.java.net/jdk8/docs/api/java/util/stream/Stream.html) in the JDK interface without breaking the implementing classes.

## When to Use Default Method Over Abstract Classes

### Abstract Classes Versus Interfaces in Java 8

After introducing Default Method, it seems that interfaces and abstract classes are the same. However, they are still a different concept in Java 8.

The abstract class can define constructors. They are more structured and can have a state associated with them. While in contrast, default method can be implemented only in the terms of invoking other interface methods, with no reference to a particular implementation's state. Hence, both are used for different purposes and choosing between two really depends on the scenario context.

## Default Method and Multiple Inheritance Ambiguity Problems

Since Java classes can implement multiple interfaces and each interface can define a default method with the same method signature, the inherited methods can conflict with each other.

Consider the example below:

[?](http://muhammadkhojaye.blogspot.com/2014/03/interface-default-methods-in-java-8.html)

public interface InterfaceA {

default void defaultMethod(){

System.out.println("Interface A default method");

}

}

public interface InterfaceB {

default void defaultMethod(){

System.out.println("Interface B default method");

}

}

public class Impl implements InterfaceA, InterfaceB {

}

The above code will fail to compile with the following error:

java: class Impl inherits unrelated defaults for defaultMethod() from types InterfaceA and InterfaceB

In order to fix this class, we need to provide a default method implementation:

[?](http://muhammadkhojaye.blogspot.com/2014/03/interface-default-methods-in-java-8.html)

public class Impl implements InterfaceA, InterfaceB {

public void defaultMethod(){

}

}

Further, if we want to invoke default implementation provided by any super interface rather than our own implementation, we can do so as follows:

[?](http://muhammadkhojaye.blogspot.com/2014/03/interface-default-methods-in-java-8.html)

public class Impl implements InterfaceA, InterfaceB {

public void defaultMethod(){

// existing code here..

InterfaceA.super.defaultMethod();

}

}

We can choose any default implementation or both as part of our new method.

**Consumer Functional Interface**

## Interface Consumer<T>

* **Type Parameters:**

T - the type of the input to the operation

**All Known Subinterfaces:**

[Stream.Builder](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.Builder.html)<T>

**Functional Interface:**

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **Consumer<T>**

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects.

This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [accept(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-).

**Since:**

1.8

### *Method Summary*

|  |  |
| --- | --- |
| **All Methods**[**Instance Methods**](javascript:show(2);)[**Abstract Methods**](javascript:show(4);)[**Default Methods**](javascript:show(16);) | |
| **Modifier and Type** | **Method and Description** |
| Void | [**accept**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t)  Performs this operation on the given argument. |
| default [**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)> | [**andThen**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#andThen-java.util.function.Consumer-)([**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)> after)  Returns a composed Consumer that performs, in sequence, this operation followed by the after operation. |

### *Method Detail*

#### accept

void accept([T](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t)

Performs this operation on the given argument.

**Parameters:**

t - the input argument

#### andThen

default [Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<[T](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)> andThen([Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)> after)

Returns a composed Consumer that performs, in sequence, this operation followed by the after operation. If performing either operation throws an exception, it is relayed to the caller of the composed operation. If performing this operation throws an exception, the after operation will not be performed.

**Parameters:**

after - the operation to perform after this operation

**Returns:**

a composed Consumer that performs in sequence this operation followed by the after operation

**Throws:**

[NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) - if after is null

## Difference Between Default Method and Regular Method

Default Method is different from the regular method in the sense that default method comes with default modifier. Additionally, methods in classes can use and modify method arguments as well as the fields of their class, but default method, on the other hand, can only access its arguments as interfaces do not have any state.

In summary, Default methods enable us to add new functionality to existing interfaces without breaking older implementation of these interfaces.

When we extend an interface that contains a default method, we can perform following,

* Not override the default method and will inherit the default method.
* Override the default method similar to other methods we override in subclass..
* Redeclare default method as abstract, which force subclass to override it.

## ****Lambda Expression:****

The lambda expression is an anonymous method (a method without a name) that is used to implement the abstract method of the functional interface.

The lambda expression introduces a new operator in Java called the ARROW ( -> )   operator. It divides the lambda expression into two parts:

(n) - > System.out.println(n);

In regards to the left-side, specify-required parameter of the implemented method, it can be empty if no parameter is required. As far as the right-side of the lambda expression, you will need to specify the action of the lambda expression (body of the implemented method);

In Java 7, when we want to use functional interface in our class, we have to implement the interface and either override the abstract method or use the anonymous class of that interface.

### ****Example 1:  Let’s Create a Thread Without the Lambda Expression.****

public class LambdaTest implements Runnable{

public void run() {

System.out.println("thread running");

}

public static void main( String[] args ){

Thread t = new Thread(new LambdaTest());

t.start();

}

}

But, in Java 8, it is not necessary to implement the functional interface in our class. We can use it with a lambda expression, as shown below.

### ****Example 2: Thread Creation With the Lambda Expression****

public class LambdaTest{

public static void main( String[] args ){

Runnable run = () -> {

System.out.println("Thread");

};

/\* you can also write like this

Runnable run = () -> System.out.println("Thread");

\*/

Thread t = new Thread(run);

t.start();

}

}

From the above example, we have noticed that the thread creation with the lambda expression results in simpler, clean code.

# Functional Interfaces In Java

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, [lambda expressions](https://www.geeksforgeeks.org/lambda-expressions-java-8/) can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. ***Runnable***, ***ActionListener***, ***Comparable*** are some of the examples of functional interfaces.  
Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

|  |
| --- |
| // Java program to demonstrate functional interface   class Test  {      public static void main(String args[])      {          // create anonymous inner class object          new Thread(new Runnable()          {              @Override              public void run()              {                  System.out.println("New thread created");              }          }).start();      }  } |

***chevron\_right***

*filter\_none*

Output:

New thread created

Java 8 onwards, we can assign [lambda expression](https://www.geeksforgeeks.org/lambda-expressions-java-8/) to its functional interface object like this:

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| // Java program to demonstrate Implementation of  // functional interface using lambda expressions    class Test  {    public static void main(String args[])    {        // lambda expression to create the object      new Thread(()->         {System.out.println("New thread created");}).start();    }  } |

***chevron\_right***

*filter\_none*

**java.util.function Package:**  
The java.util.function package in Java 8 contains many builtin functional interfaces like-

* **Predicate:** The Predicate interface has an abstract method test which gives a Boolean value as a result for the specified argument. Its prototype is
* public Predicate
* {
* public boolean test(T t);

}

* **BinaryOperator:** The BinaryOperator interface has an abstract method apply which takes two argument and returns a result of same type. Its prototype is
* public interface BinaryOperator
* {
* public T apply(T x, T y);

}

* **Function:** The Function interface has an abstract method apply which takes argument of type T and returns a result of type R. Its prototype is
* public interface Function
* {
* public R apply(T t);

}

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| // A simple program to demonstrate the use  // of predicate interface  import java.util.\*;  import java.util.function.Predicate;    class Test  {      public static void main(String args[])      {            // create a list of strings          List<String> names =              Arrays.asList("Geek","GeeksQuiz","g1","QA","Geek2");            // declare the predicate type as string and use          // lambda expression to create object          Predicate<String> p = (s)->s.startsWith("G");            // Iterate through the list          for (String st:names)          {              // call the test method              if (p.test(st))                  System.out.println(st);          }      }  } |

***chevron\_right***

*filter\_none*

Output:

Geek

GeeksQuiz

Geek2

**Important Points/Observations:**

1. A functional interface has only one abstract method but it can have multiple default methods.
2. @FunctionalInterface annotation is used to ensure an interface can’t have more than one abstract method. The use of this annotation is optional.
3. The java.util.function package contains many builtin functional interfaces in Java 8.

### Java Stream API for Bulk Data Operations on Collections

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-  
**Intermediate Operations:**

1. **map:**The map method is used to map the items in the collection to other objects according to the Predicate passed as argument.  
   List number = Arrays.asList(2,3,4,5);  
   List square = number.stream().map(x->x\*x).collect(Collectors.toList());
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
3. **sorted:** The sorted method is used to sort the stream.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List number = Arrays.asList(2,3,4,5,3);  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

**Program to demonstrate the use of Stream**

|  |
| --- |
| //a simple program to demonstrate the use of stream in java  import java.util.\*;  import java.util.stream.\*;    class Demo  {    public static void main(String args[])    {        // create a list of integers      List<Integer> number = Arrays.asList(2,3,4,5);        // demonstration of map method      List<Integer> square = number.stream().map(x -> x\*x).                             collect(Collectors.toList());      System.out.println(square);        // create a list of String      List<String> names =                  Arrays.asList("Reflection","Collection","Stream");        // demonstration of filter method      List<String> result = names.stream().filter(s->s.startsWith("S")).                            collect(Collectors.toList());      System.out.println(result);        // demonstration of sorted method      List<String> show =              names.stream().sorted().collect(Collectors.toList());      System.out.println(show);        // create a list of integers      List<Integer> numbers = Arrays.asList(2,3,4,5,2);        // collect method returns a set      Set<Integer> squareSet =           numbers.stream().map(x->x\*x).collect(Collectors.toSet());      System.out.println(squareSet);        // demonstration of forEach method      number.stream().map(x->x\*x).forEach(y->System.out.println(y));        // demonstration of reduce method      int even =         number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);        System.out.println(even);    }  } |

Output:

[4, 9, 16, 25]

[Stream]

[Collection, Reflection, Stream]

[16, 4, 9, 25]

4

9

16

25

6

**Important Points/Observations:**

1. A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
2. Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

### Java Time API

New date-time API is introduced in Java 8 to overcome the following drawbacks of old date-time API :

1. **Not thread safe :**Unlike old java.util.Date which is not thread safe the new date-time API is *immutable* and doesn’t have setter methods.
2. **Less operations :**In old API there are only few date operations but the new API provides us with many date operations.

Java 8 under the package java.time introduced a new date-time API, most important classes among them are :

1. **Local :**Simplified date-time API with no complexity of timezone handling.
2. **Zoned :**Specialized date-time API to deal with various timezones.
3. **LocalDate/LocatTime**and **LocalDateTime API :**Use it when time zones are NOT required.

|  |
| --- |
| // Java code for LocalDate  // / LocalTime Function  import java.time.\*;  import java.time.format.DateTimeFormatter;    public class Date {    public static void LocalDateTimeApi()  {        // the current date      LocalDate date = LocalDate.now();      System.out.println("the current date is "+                          date);          // the current time      LocalTime time = LocalTime.now();      System.out.println("the current time is "+                          time);          // will give us the current time and date      LocalDateTime current = LocalDateTime.now();      System.out.println("current date and time : "+                          current);          // to print in a particular format      DateTimeFormatter format =        DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");        String formatedDateTime = current.format(format);        System.out.println("in foramatted manner "+                          formatedDateTime);          // printing months days and seconds      Month month = current.getMonth();      int day = current.getDayOfMonth();      int seconds = current.getSecond();      System.out.println("Month : "+month+" day : "+                          day+" seconds : "+seconds);        // printing some specified date      LocalDate date2 = LocalDate.of(1950,1,26);      System.out.println("the repulic day :"+date2);        // printing date with current time.      LocalDateTime specificDate =          current.withDayOfMonth(24).withYear(2016);        System.out.println("specfic date with "+                         "current time : "+specificDate);  }        // Driver code      public static void main(String[] args)      {          LocalDateTimeApi();      }  } |

**Output:**

the current date is 2018-04-09

the current time is 06:21:10.409

current date and time : 2018-04-09T06:21:10.410

in foramatted manner 09-04-2018 06:21:10

Month : APRIL day : 9 seconds : 10

the repulic day :1950-01-26

specfic date with current time : 2016-04-24T06:21:10.410

* 1. **Zoned date-time API**: Use it when time zones are to be considered

|  |
| --- |
| // Java code for Zoned date-time API  import java.time.LocalDateTime;  import java.time.ZoneId;  import java.time.ZonedDateTime;  import java.time.format.DateTimeFormatter;    public class Zone {    // Function to get Zoned Date and Time  public static void ZonedTimeAndDate()  {      LocalDateTime date = LocalDateTime.now();      DateTimeFormatter format1 =        DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");        String formattedCurrentDate = date.format(format1);        System.out.println("formatted current Date and"+                        " Time : "+formattedCurrentDate);        // to get the current zone      ZonedDateTime currentZone = ZonedDateTime.now();      System.out.println("the current zone is "+                          currentZone.getZone());        // getting time zone of specific place      // we use withZoneSameInstant(): it is      // used to return a copy of this date-time      // with a different time-zone,      // retaining the instant.      ZoneId tokyo = ZoneId.of("Asia/Tokyo");        ZonedDateTime tokyoZone =              currentZone.withZoneSameInstant(tokyo);        System.out.println("tokyo time zone is " +                          tokyoZone);        DateTimeFormatter format =          DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");        String formatedDateTime = tokyoZone.format(format);        System.out.println("formatted tokyo time zone "+                          formatedDateTime);    }        // Driver code      public static void main(String[] args)      {            ZonedTimeAndDate();        }  } |

**Output:**

formatted current Date and Time : 09-04-2018 06:21:13

the current zone is Etc/UTC

tokyo time zone is 2018-04-09T15:21:13.220+09:00[Asia/Tokyo]

formatted tokyo time zone 09-04-2018 15:21:13

1. **Period**and **Duration** classes :  
   *Period :*It deals with *date* based amount of time.  
   *Duration :* It deals with*time*based amount of time.

|  |
| --- |
| // Java code for period and duration  import java.time.LocalDate;  import java.time.LocalTime;  import java.time.Month;  import java.time.Duration;  import java.time.Period;    public class Geekforgeeks {        public static void checkingPeriod()      {          LocalDate date1 = LocalDate.now();            LocalDate date2 =              LocalDate.of(2014, Month.DECEMBER, 12);            Period gap = Period.between(date2, date1);          System.out.println("gap between dates "+                             "is a period of "+gap);  }        // Function to check duration      public static void checkingDuraion()      {            LocalTime time1 = LocalTime.now();          System.out.println("the current time is " +                              time1);            Duration fiveHours = Duration.ofHours(5);            // adding five hours to the current          // time and storing it in time2          LocalTime time2 = time1.plus(fiveHours);            System.out.println("after adding five hours " +                             "of duration " + time2);            Duration gap = Duration.between(time2, time1);          System.out.println("duraion gap between time1" +                             " & time2 is " + gap);  }        // Driver code      public static void main(String[] args)      {          checkingPeriod();          checkingDuraion();      }  } |

**Output:**

gap between dates is a period of P3Y3M28D

the current time is 06:21:18.248

after adding five hours of duration 11:21:18.248

duraion gap between time1 & time2 is PT-5H

1. **ChronoUnits Enum :**java.time.temporal.ChronoUnit enum is added in Java 8 to replace integer values used in old API to represent day, month etc.

|  |
| --- |
| // Java code for ChronoUnits Enum  import java.time.LocalDate;  import java.time.temporal.ChronoUnit;    public class Geeksforgeeks {        // Function to check ChronoUnit      public static void checkingChronoEnum()      {          LocalDate date = LocalDate.now();          System.out.println("current date is :" +                              date);            // adding 2 years to the current date          LocalDate year =               date.plus(2, ChronoUnit.YEARS);            System.out.println("next to next year is " +                              year);            // adding 1 month to the current data          LocalDate nextMonth =                    date.plus(1, ChronoUnit.MONTHS);            System.out.println("the next month is " +                              nextMonth);            // adding 1 week to the current date          LocalDate nextWeek =                    date.plus(1, ChronoUnit.WEEKS);            System.out.println("next week is " + nextWeek);            // adding 2 decades to the current date          LocalDate Decade =                    date.plus(2, ChronoUnit.DECADES);            System.out.println("20 years after today " +                              Decade);      }        // Driver code      public static void main(String[] args) {            checkingChronoEnum();        }  } |

**Output:**

current date is :2018-04-09

next to next year is 2020-04-09

the next month is 2018-05-09

next week is 2018-04-16

20 years after today 2038-04-09

1. **TemporalAdjuster :**It is used to perform various date related operations.

|  |
| --- |
| // Java code Temporal Adjuster  import java.time.LocalDate;  import java.time.temporal.TemporalAdjusters;  import java.time.DayOfWeek;    public class Geek  {        // Function to check date and time      // according to our requirement      public static void checkingAdjusters()      {            LocalDate date = LocalDate.now();          System.out.println("the current date is "+                              date);            // to get the first day of next month          LocalDate dayOfNextMonth =                date.with(TemporalAdjusters.                          firstDayOfNextMonth());            System.out.println("firstDayOfNextMonth : " +                              dayOfNextMonth );            // get the next saturday          LocalDate nextSaturday =                  date.with(TemporalAdjusters.                            next(DayOfWeek.SATURDAY));            System.out.println("next satuday from now is "+                              nextSaturday);            // first day of current month          LocalDate firstDay =                    date.with(TemporalAdjusters.                    firstDayOfMonth());            System.out.println("firstDayOfMonth : " +                              firstDay);            // last day of current month          LocalDate lastDay =                    date.with(TemporalAdjusters.                              lastDayOfMonth());            System.out.println("lastDayOfMonth : " +                              lastDay);  }        // Driver code      public static void main(String[] args)      {            checkingAdjusters();      }  } |

**Output:**

the current date is 2018-04-09

firstDayOfNextMonth : 2018-05-01

next satuday from now is 2018-04-14

firstDayOfMonth : 2018-04-01

lastDayOfMonth : 2018-04-30

### Collection API improvements

We have already seen forEach() method and Stream API for collections. Some new methods added in Collection API are:

* + Iterator default method forEachRemaining(Consumer action) to perform the given action for each remaining element until all elements have been processed or the action throws an exception.
  + Collection default method removeIf(Predicate filter) to remove all of the elements of this collection that satisfy the given predicate.
  + Collection spliterator() method returning Spliterator instance that can be used to traverse elements sequentially or parallel.
  + Map replaceAll(), compute(), merge() methods.
  + Performance Improvement for HashMap class with Key Collisions

### Concurrency API improvements

Some important concurrent API enhancements are:

* + ConcurrentHashMap compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
  + CompletableFuture that may be explicitly completed (setting its value and status).
  + Executors newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.

### Java IO improvements

Some IO improvements known to me are:

* + Files.list(Path dir) that returns a lazily populated Stream, the elements of which are the entries in the directory.
  + Files.lines(Path path) that reads all lines from a file as a Stream.
  + Files.find() that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
  + BufferedReader.lines() that return a Stream, the elements of which are lines read from this BufferedReader.

### Miscellaneous Core API improvements

Some misc API improvements that might come handy are:

* + [ThreadLocal](https://www.journaldev.com/1076/java-threadlocal-example) static method withInitial(Supplier supplier) to create instance easily.
  + [Comparator](https://www.journaldev.com/780/comparable-and-comparator-in-java-example) interface has been extended with a lot of default and static methods for natural ordering, reverse order etc.
  + min(), max() and sum() methods in Integer, Long and Double wrapper classes.
  + logicalAnd(), logicalOr() and logicalXor() methods in Boolean class.
  + [ZipFile](https://www.journaldev.com/957/java-zip-file-folder-example).stream() method to get an ordered Stream over the ZIP file entries. Entries appear in the Stream in the order they appear in the central directory of the ZIP file.
  + Several utility methods in Math class.
  + jjs command is added to invoke Nashorn Engine.
  + jdeps command is added to analyze class files
  + JDBC-ODBC Bridge has been removed.
  + PermGen memory space has been removed

That’s all for Java 8 features with example programs. If I have missed some important features of Java 8, please let me know through comments.

# Java 8 method reference example

In [Java 8](https://howtodoinjava.com/java-8-tutorial/), we can refer a method from class or object using **class::methodName** type syntax. Let’s learn about different types of available **method references in java 8**.

Table of Contents

1. [Types of Method References](https://howtodoinjava.com/java8/lambda-method-references-example/#method-references)

2. [Reference to static method - Class::staticMethodName](https://howtodoinjava.com/java8/lambda-method-references-example/#static-methods)

3. [Reference to instance method from instance - ClassInstance::instanceMethodName](https://howtodoinjava.com/java8/lambda-method-references-example/#instance-method)

4. [Reference to instance method from class type - Class::instanceMethodName](https://howtodoinjava.com/java8/lambda-method-references-example/#class-method)

5. [Reference to constructor - Class::new](https://howtodoinjava.com/java8/lambda-method-references-example/#constructor)

## 1. Types of method references

Java 8 allows four types of method references.

|  |  |  |
| --- | --- | --- |
| **Method Reference** | **Description** | **Method reference example** |
| Reference to **static method** | Used to refer static methods from a class | Math::max equivalent to Math.max(x,y) |
| Reference to **instance method from instance** | Refer to an instance method using a reference to the supplied object | System.out::println equivalent to System.out.println(x) |
| Reference to **instance method from class type** | Invoke the instance method on a reference to an object supplied by the context | String::length equivalent to str.length() |
| Reference to **constructor** | Reference to a constructor | ArrayList::new equivalent to new ArrayList() |

## Rules for method reference

## 1.Method name can be anything but only method parameter should be equal to functional interface and reffered class instance method.

## 2. Return type can be anything there is no restriction with method return type.

## 2. Method reference to static method – Class::staticMethodName

An example to use Math.max() which is static method.

|  |
| --- |
| List<Integer> integers = Arrays.asList(1,12,433,5);  Optional<Integer> max = integers.stream().reduce( Math::max );  max.ifPresent(value -> System.out.println(value)); |

Output:

433

## 3. Method reference to instance method from instance – ClassInstance::instanceMethodName

In above example, we use System.out.println(value) to print the max value found. We can use System.out::println to print the value.

|  |
| --- |
| List<Integer> integers = Arrays.asList(1,12,433,5);  Optional<Integer> max = integers.stream().reduce( Math::max );  max.ifPresent( System.out::println ); |

Output:

433

## 4. Method reference to instance method from class type – Class::instanceMethodName

In this example, s1.compareTo(s2) is referred as String::compareTo.

|  |
| --- |
| List<String> strings = Arrays          .asList("how", "to", "do", "in", "java", "dot", "com");  List<String> sorted = strings          .stream()          .sorted((s1, s2) -> s1.compareTo(s2))          .collect(Collectors.toList());  System.out.println(sorted);  List<String> sortedAlt = strings          .stream()          .sorted(String::compareTo)          .collect(Collectors.toList());  System.out.println(sortedAlt); |

Output:

[com, do, dot, how, in, java, to]

[com, do, dot, how, in, java, to]

## 5. Reference to constructor – Class::new

The first method can be updated to create a list of integers from 1 to 100. Using [lambda expression](https://howtodoinjava.com/java8/complete-lambda-expressions-tutorial-in-java/) is rather easy. To create a new instance of ArrayList, we have use ArrayList::new.

|  |
| --- |
| List<Integer> integers = IntStream                  .range(1, 100)                  .boxed()                  .collect(Collectors.toCollection( ArrayList::new ));    Optional<Integer> max = integers.stream().reduce(Math::max);  max.ifPresent(System.out::println); |

Output:

99

**Java8 features end---**

# 5 Different Ways to Create Objects in Java

### A list of five ways to create objects in Java, how they interact with constructors, and an example of how to utilize all of these methods.

Being Java developers, we usually create lots of objects daily, but we always use dependency management systems e.g. Spring to create these objects. However, there are more ways to create objects, which we will study in this article.

There are five total ways to create objects in Java, which are explained below with their examples followed by bytecode of the line which is creating the object.

|  |  |
| --- | --- |
| Using new keyword | } → constructor gets called |
| Using [newInstance()](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#newInstance--) method of Class class | } → constructor gets called |
| Using [newInstance()](https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Constructor.html#newInstance-java.lang.Object...-) method of Constructor class | } → constructor gets called |
| Using clone() method | } → no constructor call |
| Using deserialization | } → no constructor call |

If you will execute program given in end, you will see method 1, 2, 3 uses the constructor to create the object while 4, 5 doesn’t call the constructor to create the object.

### ****1. Using new keywords****

It is the most common and regular way to create an object and a very simple one also. By using this method we can call whichever constructor we want to call (no-arg constructor as well as parameterized).

Employee emp1 = new Employee();

0: new #19 // class org/programming/mitra/exercises/Employee

3: dup

4: invokespecial #21 // Method org/programming/mitra/exercises/Employee."":()V

### ****2. Using newInstance() method of Class class****

We can also use the newInstance() method of a Class class to create an object. This newInstance() method calls the no-arg constructor to create the object.

We can create an object by newInstance() in the following way:

Employee emp2 = (Employee) Class.forName("org.programming.mitra.exercises.Employee").newInstance();

Or

Employee emp2 = Employee.class.newInstance();

51: invokevirtual #70 // Method java/lang/Class.newInstance:()Ljava/lang/Object;

### ****4. Using newInstance() method of Constructor class****

Similar to the newInstance() method of Class class, There is one newInstance() method in the java.lang.reflect.Constructor class which we can use to create objects. We can also call parameterized constructor, and private constructor by using this newInstance() method.

Constructor<Employee> constructor = Employee.class.getConstructor();

Employee emp3 = constructor.newInstance();

111: invokevirtual #80 // Method java/lang/reflect/Constructor.newInstance:([Ljava/lang/Object;)Ljava/lang/Object;

Both newInstance() methods are known as reflective ways to create objects. In fact newInstance() method of Class class internally uses newInstance() method of Constructor class. That's why the later one is preferred and also used by different frameworks like Spring, Hibernate, Struts etc. To know differences between both newInstance() methods read [Creating objects through Reflection in Java with Example](https://programmingmitra.blogspot.in/2016/05/creating-objects-through-reflection-in-java-with-example.html).

### ****4. Using clone() method:****

Whenever we call clone() on any object, the JVM actually creates a new object for us and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.

To use clone() method on an object we need to implement Cloneable and define the clone() method in it.

Employee emp4 = (Employee) emp3.clone();

162: invokevirtual #87 // Method org/programming/mitra/exercises/Employee.clone ()Ljava/lang/Object;

Java cloning is the most debatable topic in Java community and it surely does have its drawbacks but it is still the most popular and easy way of creating a copy of any object until that object is full filling mandatory conditions of Java cloning. I have covered cloning in details in a 3 article long [Java Cloning Series](https://programmingmitra.blogspot.in/search/label/Java%20Cloning) which includes ([Java Cloning And Types Of Cloning (Shallow And Deep) In Details With Example](https://programmingmitra.blogspot.in/2016/11/Java-Cloning-Types-of-Cloning-Shallow-Deep-in-Details-with-Example.html), [Java Cloning - Copy Constructor Versus Cloning](https://programmingmitra.blogspot.in/2017/01/Java-cloning-copy-constructor-versus-Object-clone-or-cloning.html), [Java Cloning - Even Copy Constructors Are Not Sufficient](https://programmingmitra.blogspot.in/2017/01/java-cloning-why-copy-constructors-are-not-sufficient-or-good.html)), go ahead and read them if you want to know more about cloning.

### ****5. Using deserialization:****

Whenever we serialize and deserialize an object, the JVM creates a separate object for us. In deserialization, the JVM doesn’t use any constructor to create the object.

To deserialize an object we need to implement a Serializable interface in our class.

ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj"));

Employee emp5 = (Employee) in.readObject();

261: invokevirtual #118 // Method java/io/ObjectInputStream.readObject:()Ljava/lang/Object;

As we can see in the above bytecode snippets, all 4 methods are called and get converted to invokevirtual (object creation is directly handled by these methods) except the first one, which got converted to two calls: one is new and other is invokespecial (call to constructor).

**Example**

Let’s consider an Employee class for which we are going to create the objects:

class Employee implements Cloneable, Serializable {

private static final long serialVersionUID = 1L;

private String name;

public Employee() {

System.out.println("Employee Constructor Called...");

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + ((name == null) ? 0 : name.hashCode());

return result;

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Employee other = (Employee) obj;

if (name == null) {

if (other.name != null)

return false;

} else if (!name.equals(other.name))

return false;

return true;

}

@Override

public String toString() {

return "Employee [name=" + name + "]";

}

@Override

public Object clone() {

Object obj = null;

try {

obj = super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return obj;

}

}

In the below Java program we are going to create Employee objects in all 5 ways. You can also find the source code at [GitHub](https://github.com/njnareshjoshi/exercises/tree/master/src/org/programming/mitra/exercises).

public class ObjectCreation {

public static void main(String... args) throws Exception {

// By using new keyword

Employee emp1 = new Employee();

emp1.setName("Naresh");

System.out.println(emp1 + ", hashcode : " + emp1.hashCode());

// By using Class class's newInstance() method

Employee emp2 = (Employee) Class.forName("org.programming.mitra.exercises.Employee")

.newInstance();

// Or we can simply do this

// Employee emp2 = Employee.class.newInstance();

emp2.setName("Rishi");

System.out.println(emp2 + ", hashcode : " + emp2.hashCode());

// By using Constructor class's newInstance() method

Constructor<Employee> constructor = Employee.class.getConstructor();

Employee emp3 = constructor.newInstance();

emp3.setName("Yogesh");

System.out.println(emp3 + ", hashcode : " + emp3.hashCode());

// By using clone() method

Employee emp4 = (Employee) emp3.clone();

emp4.setName("Atul");

System.out.println(emp4 + ", hashcode : " + emp4.hashCode());

// By using Deserialization

// Serialization

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("data.obj"));

out.writeObject(emp4);

out.close();

//Deserialization

ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj"));

Employee emp5 = (Employee) in.readObject();

in.close();

emp5.setName("Akash");

System.out.println(emp5 + ", hashcode : " + emp5.hashCode());

}

}

This program will give the following output:

Employee Constructor Called...

Employee [name=Naresh], hashcode : -1968815046

Employee Constructor Called...

Employee [name=Rishi], hashcode : 78970652

Employee Constructor Called...

Employee [name=Yogesh], hashcode : -1641292792

Employee [name=Atul], hashcode : 2051657

Employee [name=Akash], hashcode : 63313419

# Java synchronized keyword

By Lokesh Gupta | Filed Under: [Multi Threading](https://howtodoinjava.com/java/multi-threading/)

**Java synchronized keyword** marks a block or method a critical section. A critical section is where one and only one thread is executing at a time, and the thread holds the lock for the synchronized section.

**synchronized** keyword helps in writing [concurrent](https://howtodoinjava.com/java-concurrency-tutorial/) parts of the applications, to protect shared resources within this block.

The synchronized keyword can be use with –

* a code block
* a method

## 1. Java synchronized block

#### 1.1. Syntax

The general syntax for writing a synchronized block is as follows. Here **lockObject** is a reference to an object whose lock associates with the monitor that the synchronized statements represent.

|  |
| --- |
| Syntax |
| synchronized( lockObject )  {     // synchronized statements  } |

#### 1.2. Internal working

When a thread wants to execute synchronized statements inside the synchronized block, it MUST acquire the lock on lockObject‘s monitor. At a time, only one thread can acquire the monitor of a lock object. So all other threads must wait till this thread, currently acquired the lock, finish it’s execution.

In this way, synchronized keyword guarantees that only one thread will be executing the synchronized block statements at a time, and thus prevent multiple threads from corrupting the shared data inside the block.

Keep in mind that if a thread is put on sleep (using sleep() method) then it does not release the lock. At this sleeping time, no thread will be executing the synchronized block statements.

Java synchronization will throw **NullPointerException** if lock object used in 'synchronized (lock)' is null.

#### 1.3. Java synchronized block example

Java program to demonstrate the usage of synchronized block. In given example, we have a MathClasswith a method printNumbers(). This method will print the numbers starting from 1 to the argument number N.

Notice that the code in printNumbers() method is inside synchronized block.

|  |
| --- |
| MathClass.java |
| public class MathClass  {      void printNumbers(int n) throws InterruptedException      {          synchronized (this)          {              for (int i = 1; i <= n; i++)              {                  System.out.println(Thread.currentThread().getName() + " :: "+  i);                  Thread.sleep(500);              }          }      }  } |

I have created two threads which start executing the printNumbers() method exactly at same time. Due to block being synchronized, only one thread is allowed access and other thread has to wait until first thread is finished.

|  |
| --- |
| Main.java |
| public class Main  {      public static void main(String args[])      {          final MathClass mathClass = new MathClass();            //first thread          Runnable r = new Runnable()          {              public void run()              {                  try {                      mathClass.printNumbers(3);                  } catch (InterruptedException e) {                      e.printStackTrace();                  }              }          };            new Thread(r, "ONE").start();          new Thread(r, "TWO").start();      }  } |

Program output.

|  |
| --- |
| Console |
| ONE :: 1  ONE :: 2  ONE :: 3    TWO :: 1  TWO :: 2  TWO :: 3 |

## 2. Java synchronized method

#### 2.1. Syntax

The general syntax for writing a synchronized method is as follows. Here **lockObject** is a reference to an object whose lock associates with the monitor that the synchronized statements represent.

|  |
| --- |
| Syntax |
| <access modifier> synchronized method( parameters )  {      // synchronized code  } |

#### 2.2. Internal working

Similar to synchronized block, a thread MUST acquire the lock on the associated monitor object with synchronized method. In case of synchronized method, the lock object is –

* **‘.class’ object** – if the method is static.
* **‘this’ object** – if the method is not static. ‘this’ refer to reference to current object in which synchronized method is invoked.

Read More : [Object level lock vs Class level lock in Java](https://howtodoinjava.com/java/multi-threading/object-vs-class-level-locking/)

Java synchronized keyword is **re-entrant** in nature it means if a synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.

#### 2.3. Java synchronized method example

Similar to synchronized block example, we can apply synchronized keyword at printNumber() method and it will make the method as synchronized. Now if we again run the example, we will get the similar output.

|  |
| --- |
| MathClass.java |
| public class MathClass  {      synchronized void printNumbers(int n) throws InterruptedException      {          for (int i = 1; i <= n; i++)          {              System.out.println(Thread.currentThread().getName() + " :: "+  i);              Thread.sleep(500);          }      }  } |

Program output.

|  |
| --- |
| Console |
| ONE :: 1  ONE :: 2  ONE :: 3    TWO :: 1  TWO :: 2  TWO :: 3 |

**Exception interview Question 1. What is exception in java?**

**Answer**. It’s the basic Exception framework interview question. Freshers must know about this. Java Exception handling provides a mechanism to handle compile and runtime errors in java.

* To make application robust Exception must be handled appropriately,
* by handling exceptions we end up giving some meaningful message to end user rather than giving meaningless message in java.

**Exception interview Question 2. Explain exception hierarchy in java?**

**Answer**. It’s also the basic Exception handling interview question. Freshers must know about this.

**Exception hierarchy in java >** ****

java.lang.**Object** is superclass of all classes in java.

java.lang.**Throwable** is superclass of java.lang.**Exception** and java.lang.**Error**

java.lang.**Exception** is superclass of java.lang.**RuntimeException, IOException, SQLException,** [**BrokenBarrierException**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html)and many more other classes in java.

java.lang.[**RuntimeException**](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)is superclass of java.lang.[**NullPointerException**](http://www.javamadesoeasy.com/2015/05/nullpointerexception-in-java.html)**, ArithmeticException** and many more other classesin java.

java.lang.[**Error**](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html)is superclass of java.lang.**VirtualMachineError, IOError, AssertionError,** [**ThreadDeath**](http://www.javamadesoeasy.com/2015/04/threaddeath-error-calling-stop-method.html)and many more other classesin java.

java.lang.**VirtualMachineError** is superclass of java.lang.**OutOfMemoryError, StackOverflowError** and many more other classesin java.

**Exception interview Question 3. What are differences between checked and unchecked exceptions in java?**

**Answer**. This is very important Exception handling interview question in java.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***checked exception*** | ***unchecked exception*** |
| 1 | Also known as | **checked** exceptions are also known as **compileTime** exceptions in java. | **unchecked** exceptions are also known as **runtime** exceptions in java. |
| 2 | Should be solved at compile or runtime? | Checked exceptions are those which need to be taken care at compile time in java. | Unchecked exceptions are those which need to be taken care at runtime in java. |
| 3 | Benefit/ Advantage | We cannot proceed until we fix compilation issues which are most likely to happen in program, this helps us in avoiding runtime problems upto lot of extent in java. | Whenever runtime exception occurs execution of program is interrupted, but by handling these kind of exception we avoid such interruptions and end up giving some meaningful message to user in java. |
| 4 | Creating custom/own exception | |  | | --- | | **class** UserException **extends Exception** {     UserException(String s) {  **super**(s);     }  } |   By extending java.lang.**Exception**, we can create checked exception. | |  | | --- | | **class** UserException **extends RuntimeException** {     UserException(String s) {  **super**(s);     }  } |   By extending java.lang.**RuntimeException**, we can create unchecked exception. |
| 5 | [Exception propagation](http://www.javamadesoeasy.com/2015/05/exception-propagation-in-java-deep.html) | For **propagating checked** exceptions method must throw exception by using **throws** keyword. | **unchecked** exceptions are [**automatically propagated**](http://www.javamadesoeasy.com/2015/05/exception-propagation-in-java-deep.html)in java. |
| 6 | handling checked and unchecked exception while overriding superclass method | *If superclass method throws/declare* ***checked******exception*** *>*   * overridden method of subclass **can** declare/**throw** **narrower** (subclass of) **checked exception** (As shown in [**Program**](http://www.javamadesoeasy.com/2015/05/program-to-show-overridden-method-of_93.html)), or * overridden method of subclass **cannot** declare/**throw** **broader** (superclass of) **checked exception** (As shown in [**Program**](http://www.javamadesoeasy.com/2015/05/program-to-show-overridden-method-of_94.html)), or * overridden method of subclass **can** declare/**throw any unchecked /RuntimeException** (As shown in [**Program**](http://www.javamadesoeasy.com/2015/05/program-to-show-overridden-method-of_37.html)) | *If superclass method throws/declare* ***unchecked*** *>*   * overridden method of subclass **can** declare/**throw any unchecked /RuntimeException (superclass or subclass)** (As shown in [**Program**](http://www.javamadesoeasy.com/2015/05/program-to-show-overridden-method-of.html)), or * overridden method of subclass **cannot** declare/**throw** **any checked exception** (As shown in [**Program**](http://www.javamadesoeasy.com/2015/05/program-to-show-overridden-method-of_6.html)), |
|  | Which classes are which type of exception?  either  **checked or** **unchecked** exception? | The class **Exception and all its subclasses** that are **not also subclasses of RuntimeException** are checked exceptions in java. | The class **RuntimeException and all its subclasses** are unchecked exceptions.  Likewise,  The class **Error and all its subclasses** are unchecked exceptions in java. |
| 7 | Most frequently faced exceptions | [SQLException](http://www.javamadesoeasy.com/2015/05/sqlexception-in-java.html),  IOException,  ClassNotFoundException | [NullPointerException](http://www.javamadesoeasy.com/2015/05/nullpointerexception-in-java.html),  ArithmeticException ArrayIndexOutOfBoundsException. |

**Exception interview Question 4. What are 5 exception handling keywords in java?**

**Answer**. This is another very important exception handling interview question in java.

***5*** [***keyword***](http://www.javamadesoeasy.com/2015/05/keywords-in-java-language.html) ***in java exception handling in java***

* + [**try**](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html) **- Any exception occurring in try block is catched by catch block.**
  + [**catch**](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html) **-** catch block is always followed by try block in java.
  + [**finally**](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html) ***finally block*** *can can only exist if try or try-catch block is there, finally block can’t be used alone in java.*

***Features*** *of finally >*

* finally block is **always executed** irrespective of exception is thrown or not.
  + finally is **keyword** in java.
  + finally block is optional in java, we may use it or not.

*finally block is* ***not executed*** *in following scenarios >*

* finally is not executed when **System.exit** is called.
* if in case **JVM crashes** because of some java.util.[**Error**](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html).
  + [**throw**](http://www.javamadesoeasy.com/2015/05/throw-exception-in-java.html) **throw** is a **keyword** in java.
    - **throw** keyword allows us to throw [**checked** or **unchecked**](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)[exception](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html).
  + [**throws**](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html) **throws** is written in method’s definition to indicate that method can throw [exception](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html) in java.

**Exception interview Question 5. Explain what is Error in java?**

**Answer**. Experienced developers must know in detail about Exception handling interview question in java. ***java.lang.***[***Error***](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html)

* Error is a subclass of **Throwable  in java.**
* Error **indicates some serious problems** that our **application should not try to catch in java.**
* Errors are **abnormal conditions in application**.
* Error and its subclasses are regarded as **unchecked** exceptions in java

Must know :

[ThreadDeath](http://www.javamadesoeasy.com/2015/04/threaddeath-error-calling-stop-method.html) is an error which application must not try to catch but it is normal condition in java.

**Exception interview Question 6. What are differences between Exception and Error in java?**

**Answer**. It is another very important exception interview question to differentiate between Exception and Error in java.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | [**Exception**](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html) | [**Error**](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html) |
| 1 | serious problem? | Exception does **not indicate any serious problem**. | Error **indicates some serious problems** that our **application should not try to catch.** |
| 2 | divided into  [**checked** and **unchecked**](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html) | Exception are divided into **checked** and **unchecked exceptions in java**. | Error are **not divided** further into such classifications in java. |
| 3 | Which classes are which type of exception? either  **checked or** **unchecked** exception? | The class **Exception and all its subclasses** that are **not also subclasses of RuntimeException** are checked exceptions.  The class **RuntimeException and all its subclasses** are unchecked exceptions.  Likewise,  The class **Error and all its subclasses** are unchecked exceptions in java. | Error and its subclasses are regarded as **unchecked** exceptions in java |
| 4 | Most frequently faced exception and errors | **checked exceptions>**  SQLException,  IOException,  ClassNotFoundException  **unchecked exceptions>**  [NullPointerException](http://www.javamadesoeasy.com/2015/05/nullpointerexception-in-java.html), ArithmeticException, | **VirtualMachineError, IOError, AssertionError,** [**ThreadDeath**](http://www.javamadesoeasy.com/2015/04/threaddeath-error-calling-stop-method.html),  **OutOfMemoryError, StackOverflowError.** |
| 5 | Why to catch or not to catch? | Application **must catch** the Exception because they does not cause any major threat to application in java. | Application **must not catch** the Error because they does cause any major threat to application.  Example >  Let’s say errors like OutOfMemoryError and StackOverflowError occur and are caught then JVM might not be able to free up memory for rest of application to execute, so it will be better if application don’t catch these errors and is allowed to terminate in java. |

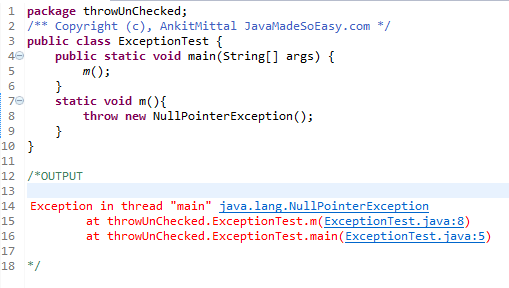
**Exception interview Question 7. Explain throw keyword in java?**

**Answer**. This is also frequently asked exception handling interview question.

* [**throw**](http://www.javamadesoeasy.com/2015/05/throw-exception-in-java.html)is a **keyword** in java.
* **throw** keyword allows us to throw [**checked** or **unchecked**](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)[exception](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html).

***throw unchecked*** *exception in java >*

* We need **not to handle** unChecked exception either by catching it or throwing it in java.



We throw NullPointerException (unChecked exception) and didn’t handled it, no compilation error was thrown in java.

***throw checked*** *exception >*

* We need to handle checked exception either by catching it, or
* throwing it by using **throws** keyword. (When thrown, exception must be handled in calling environment)

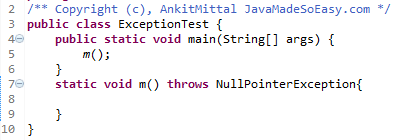
**Exception interview Question 8. Explain throws keyword in java?**

**Answer**. This exception interview question will be covered in below question but it give you more detailed information about throw keyword in java.

[throws](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html) is written in method’s definition to indicate that method can throw [exception](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html).

***throws*** [***unChecked***](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)*exception in java >*

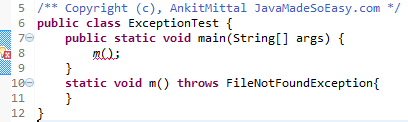
* We need **not to handle** unChecked exception either by catching it or throwing it.



Above code **throws** NullPointerException (unChecked exception) and didn’t handled it from where method m() was called and no compilation error was thrown.

***throws Checked*** *exception in java >*

* We need **to handle** checked exception either by catching it or throwing it further, if not handled we will face compilation error.



**Exception interview Question 9. What is difference between throw and throws in java?**

**Answer**. This is also another important and frequently asked exception handling interview question. To confuse interviewees Interviewers might give you code snippet and ask you to insert throw or throws keyword in java.

|  |  |  |
| --- | --- | --- |
|  | [**throw**](http://www.javamadesoeasy.com/2015/05/throw-exception-in-java.html) | [**throws**](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html) |
| 1 | **throw** [keyword](http://www.javamadesoeasy.com/2015/05/keywords-in-java-language.html) is used to throw an [exception](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html) explicitly in java. | **throws** keyword is used to declare an exception in java. |
| 2 | **throw** is used **inside method**.  Example in java >  **static** **void** m(){  **throw** **new** FileNotFoundException();  } | **throws** is used **in** **method declaration**.  Example in java >  **static** **void** m() throws FileNotFoundException{  } |
| 3 | **throw** is always **followed by** **instanceof** Exception class in java.  Example >  **throw** **new** FileNotFoundException() | **throws** is always **followed by name of Exception class in java**.  Example >  **throws** FileNotFoundException |
| 4 | **throw** can be used to throw **only one exception at time**.  Example >  **throw** **new** FileNotFoundException() | **throws** can be used to throw **multiple exception at time**.  Example >  **throws** FileNotFoundException, NullPointerException  and many more... |
| 5 | **throw** cannot propagate exception to calling method in java.  https://lh4.googleusercontent.com/gO7Xp7MlqokGsbfiqCTCq0Px1YrtSGeLXjSP53nxwSHuKsTKmTmY2vd2g5Fd7shnyP3YRLNGrJcP7J5S1jfJPkdEWLOywBluAEoMQPKp3l4oIDHDK2ugUU4tvSmt_HDX5Q3DMBI | **throws** can [propagate exception](http://www.javamadesoeasy.com/2015/05/exception-propagation-in-java-deep.html) to calling method.  Please see these programs to understand how exception is propagated to calling method.  [**Program 1**](http://www.javamadesoeasy.com/2015/05/exception-propagation-in-java-deep.html) - Handling Exception by throwing it from m() method (using throws keyword) and handling it in try-catch block from where call to method m() was made.  [**Program 2**](http://www.javamadesoeasy.com/2015/05/exception-propagation-in-java-deep.html)- Throwing Exception from m() method and then again throwing it from calling method [ i.e. main method] |

**Exception interview Question 10. How to create user defined checked and unchecked Exception in java?**

**Answer**. Very important exception handling interview question. Interviewers generally expects interviewees  to write code to create checked and unchecked Exception in java.

*Creating user defined* [***checked***](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)*exception in java >*

|  |
| --- |
| **class** UserDefinedException **extends Exception** {     UserDefinedException(String s) {  **super**(s);     }  } |

By extending java.lang.**Exception**, we can create checked exception.

*Creating user defined* [***unchecked***](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)*exception in java >*

|  |
| --- |
| **class** UserDefinedException **extends RuntimeException** {     UserDefinedException(String s) {  **super**(s);     }  } |

By extending java.lang.**RuntimeException**, we can create unchecked exception.

**Exception interview Question 11. How to use try-catch-finally in java? Can we use try,catch or finally block alone in java?**

**Answer**. This exception handling interview question will test your practical/basic understanding of Exception handling in java.

*We can enclose exception prone code in >*

* [**try-catch** block](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html), or
* **try-finally** block, or
* **try-catch-finally** block.

*Using try-catch block  in java*

|  |
| --- |
| **try{**  **//Code to be enclosed in try-catch block**  **}catch(Exception e){**  **}** |

Using try-finally block  in java

|  |
| --- |
| **try{**  **//Code to be enclosed in try-finally block**  **}finally{**  **}** |

We cannot use **try block** alone, it must be followed by either **catch** or **finally**.

Using only try block will cause *compilation error*

|  |
| --- |
| **try{**  **//only try block will cause compilation error**  **}** |

*Likewise, we cannot use* ***catch block*** *alone, it always follows* ***try block.***

Using only catch block will cause *compilation error*

|  |
| --- |
| **catch{**  **//only catch block will cause compilation error**  **}** |

*Likewise, we cannot use* ***finally block*** *alone, it always follows* ***try block.***

Using only finally block will cause *compilation error*

|  |
| --- |
| **finally{**  **//only finally block will cause compilation error**  **}** |

**Exception interview Question 12. Is it allowed to use multiple catch block in java?**

**Answer**. Another exception handling interview question which will test your practical knowledge and understanding of Exception handling in java. [Java exception handling](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html) allows us to use [multiple catch block](http://www.javamadesoeasy.com/2015/05/multiple-catch-block-in-java.html) in java.

**Important** Point  about **multiple catch block in java** >

1. **Exception class handled in starting catch block must be subclass of Exception class handled in following catch blocks (otherwise we will face compilation error).**
2. Either one of the multiple catch block will handle exception at time in java.

Program - Let’s understand the concept of multiple catch block in java>

|  |
| --- |
| /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  public class ExceptionTest {     public static void main(String[] args) {              try{                   int i=10/0; //will throw ArithmeticException            }**catch**(**ArithmeticException** ae){                   System.*out*.println("Exception handled - ArithmeticException");            }**catch**(**RuntimeException** re){                   System.*out*.println("Exception handled - RuntimeException");            }**catch**(**Exception** e){                   System.*out*.println("Exception handled - Exception");            }     }  }  /\*OUTPUT  Exception handled - ArithmeticException  \*/ |

In the above above >

**ArithmeticException** has been used in **first** catch block

**RuntimeException** has been used in **second** catch block

**Exception** has been used in **third** catch block

**Exception** is superclass of **RuntimeException** and

**RuntimeException** is superclass of **ArithmeticException.**

**Exception interview Question 13. What is Automatic resource management in java 7?**

**Answer**.Experienced java developers must be well versed with this exception interview question. As we know java allows us to handle multiple exceptions by using [multiple catch blocks](http://www.javamadesoeasy.com/2015/05/multiple-catch-block-in-java.html).   
  
Now, java 7 has done improvements in multiple exception handling by introducing **multi catch syntax** which helps in [**automatic resource management**](http://www.javamadesoeasy.com/2015/05/catch-block-and-automatic-resource.html)**.**

*Features of* ***multi catch syntax in java*** *>*

* Has **improved way of catching multiple** [**exceptions**](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html)**.**
* This syntax does **not looks clumsy in java**.
* **Reduces developer efforts** of writing multiple catch blocks in java.
* Allows us to **catch more than one exception in one catch block**.

Here is the **multi catch syntax** >

|  |
| --- |
| **try**{                   //code . . . . .            }**catch**(IOException **|** SQLException ex){                   //code . . . . .            } |

We could separate different exceptions using **pipe** ( **|** ) in java.

**Exception interview Question 14. Explain try-with-resource in java?**

**Answer**. Again experienced java developers must be well versed with this exception interview question. **Before java 7**, we used to write **explicit code for closing file in** [**finally**](http://www.javamadesoeasy.com/2015/05/finally-block-in-java.html) **block by using** [**try-finally block**](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html)like this >

|  |
| --- |
| /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** TryWithResourseTest {  **public** **static** **void** main(String[] args) **throws** IOException {            InputStream inputStream = **null**;  **try**{                   inputStream = **new** FileInputStream("c:/txtFile.txt");                   //code......            }**finally**{  **if**(inputStream!=**null**)  **inputStream.close();**            }     }  } |

**In java 7**, using **Try-with-resources >**

* we need not to write **explicit code for closing file**.

|  |
| --- |
| **import** java.io.FileInputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** TryWithResourseTest {  **public** **static** **void** main(String[] args) **throws** IOException {  **try (InputStream inputStream = new FileInputStream("c:/txtFile.txt"))** {               //code...           }    }  } |

*Using multiple resources inside* ***Try-with-resources is also allowed in java.***

**Exception interview Question  15. Now, question comes why we need not to close file when we are using Try-with-resources in java?**

**Answer**.  Again experienced java developers must be well versed with this exception interview question. Because **FileInputStream** implements java.lang.**AutoCloseable** **interface** (**AutoCloseable** interface’s close method automatically closes resources which are no longer needed) in java.

Which classes can be used inside **Try-with-resources in java?**

All the classes which implements **AutoCloseable** interface can be used inside **Try-with-resources in java.**

**Exception interview Question 16. Explain finally keyword in java?**

**Answer**. Fresher and experienced java developers must be well versed with this exception handling interview question in java.

*try or* [*try-catch*](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html) *block can be followed by finally block in java >*

* try**-finally** block, or

|  |
| --- |
| **try{**  **//Code to be enclosed in try-finally block**  **}finally{**  **}** |

* try-catch**-finally** block.

|  |
| --- |
| **try{**  **//Code to be enclosed in try-catch-finally block**  **}catch(Exception e){**  **}finally{**  **}** |

***finally block*** *can can only exist if try or try-catch block is there, finally block can’t be used alone in java.*

***Features*** *of finally in java >*

* finally block is **always executed** irrespective of exception is thrown or not.
* finally is [**keyword**](http://www.javamadesoeasy.com/2015/05/keywords-in-java-language.html)in java.

*finally block is* ***not executed*** *in following scenarios in java >*

* finally is not executed when **System.exit** is called.
* if in case **JVM crashes** because of some java.util.[**Error**](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html).

*Application of finally block in java programs in java >*

* We may use finally block to execute code for **database connection closing**, because closing connection in try or catch block may not be safe.
  + **Why closing connection in try block may not be safe?**
  + Because exception may be thrown in try block before reaching connection closing statement.
  + **Why closing connection in catch block may not be safe?**
  + Because inappropriate exception may be thrown in try block and we might not enter catch block to close connection in java.

For programs to demonstrate finally. [Please refer this post](http://www.javamadesoeasy.com/2015/05/finally-block-in-java.html).

**Exception interview Question 17. Is it allowed to use nested try-catch in java?**

**Answer**. It’s basic java exception handling interview question.

java exception handling allows us to use [nested try-catch block](http://www.javamadesoeasy.com/2015/05/nested-try-catch-block-in-java.html).

Nested [try-catch](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html) block means using try-catch block inside another try-catch block in java.

|  |
| --- |
| /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** ExceptionTest {  **public** **static** **void** main(String[] args) {    **try**{  **int** i=10/0; //will throw ArithmeticException            }**catch**(ArithmeticException ae){                   System.*out*.println("try-catch block handled - ArithmeticException");    **//using nested try-catch block**  **try{**  **String s=null;**  **s.charAt(0); //will throw NullPointerException**  **}catch(NullPointerException npe){**  **System.*out*.println("NESTED try-catch block handled - "**  **+ "NullPointerException");**  **}**              }     }  } |

**Exception interview Question 18. Discuss which checked and unchecked exception can be thrown/declared by subclass method while overriding superclass method in java?**

**Answer**. It’s very very important exception handling interview question. Experienced and freshers all must be able to answer this question.

*If superclass method throws/declare* ***unchecked/RuntimeException in java*** *>*

* overridden method of subclass **can** declare/**throw any unchecked /RuntimeException (superclass or subclass)**, or
* overridden method of subclass **cannot** declare/**throw** **any checked exception in java**, or
* overridden method of subclass **can** declare/**throw** **same exception in java**, or
* overridden method of subclass **may not** declare/**throw any exception in java**.

*If superclass method throws/declare* ***checked****/****compileTime******exception in java*** *>*

* overridden method of subclass **can** declare/**throw** **narrower** (subclass of) **checked exception**, or
* overridden method of subclass **cannot** declare/**throw** **broader** (superclass of) **checked exception**, or
* overridden method of subclass **can** declare/**throw any unchecked /RuntimeException**, or
* overridden method of subclass **can** declare/**throw** **same exception**, or
* overridden method of subclass **may not** declare/**throw any exception in java**.

*If superclass method does* ***not throw****/declare any* ***exception in java*** *>*

* overridden method of subclass **can** declare/**throw any unchecked /RuntimeException** , or
* overridden method of subclass **cannot** declare/**throw** **any checked exception**, or
* overridden method of subclass **may not** declare/**throw any exception in java**.

For programs please refer > [**Throw/declare checked and unchecked exception while overriding superclass method in java**](http://www.javamadesoeasy.com/2015/05/throwdeclare-checked-and-unchecked.html)

**Exception interview Question 19. What will happen when catch and finally block both return value, also when try and finally both return value in java?**

**Answer**. This is very important exception handling interview question for experienced developers.

When **catch and finally block** both return value, **method will ultimately return value returned by** [**finally**](http://www.javamadesoeasy.com/2015/05/finally-block-in-java.html) block irrespective of value returned by [catch](http://www.javamadesoeasy.com/2015/05/catch-block-and-automatic-resource.html) block.

|  |
| --- |
| /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  **public** **class** ExceptionTest {  **public** **static** **void** main(String[] args) {            System.*out*.println("method return -> "+*m*());     }    **static** String m(){  **try**{  **int** i=10/0; //will throw ArithmeticException            }**catch**(ArithmeticException e){  **return** "catch";            }**finally**{  **return** "finally";            }       }  }  /\*OUTPUT  method return -> finally  \*/ |

In above program, i=10/0 will throw ArithmeticException and enter catch block to return "catch", but ultimately control will enter finally block to return "finally".

Likewise, when [**try and finally**](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html) **block** both return value, **method will ultimately return value returned by finally block** irrespective of value returned by try block. For program [please refer](http://www.javamadesoeasy.com/2015/05/what-will-happen-when-catch-and-finally.html).

**Exception interview Question 20. What is exception propagation in java?**

**Answer**.

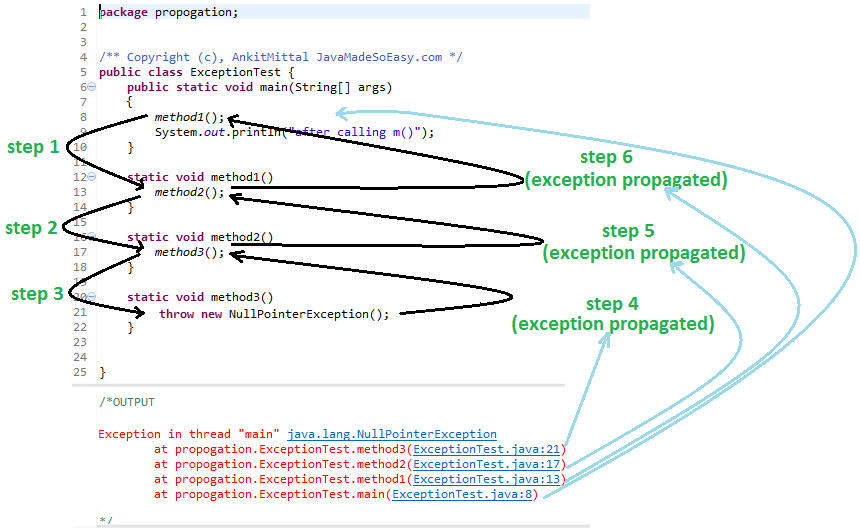
Experienced developers must know in detail about Exception handling interview question in java. Even freshers must try and understand this in depth concept of exception propagation in java.

Whenever methods are called [stack](http://javamadesoeasy.com/2015/01/stacks.html) is formed and an exception is first thrown from the top of the stack and if it is not caught, it starts coming down the stack to previous methods until it is not caught.

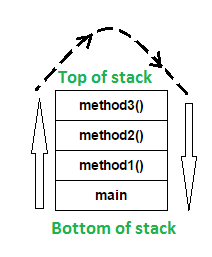
If exception remains uncaught even after reaching bottom of the stack it is propagated to JVM and program is terminated in java.

*Propagating* [***unchecked***](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html)*exception (NullPointerException) >*

**unchecked** exceptions are **automatically propagated** in java.

[](http://www.javamadesoeasy.com/2015/05/exception-propagation-in-java-deep.html)

[***stack***](http://javamadesoeasy.com/2015/01/stacks.html)*of methods is formed >*

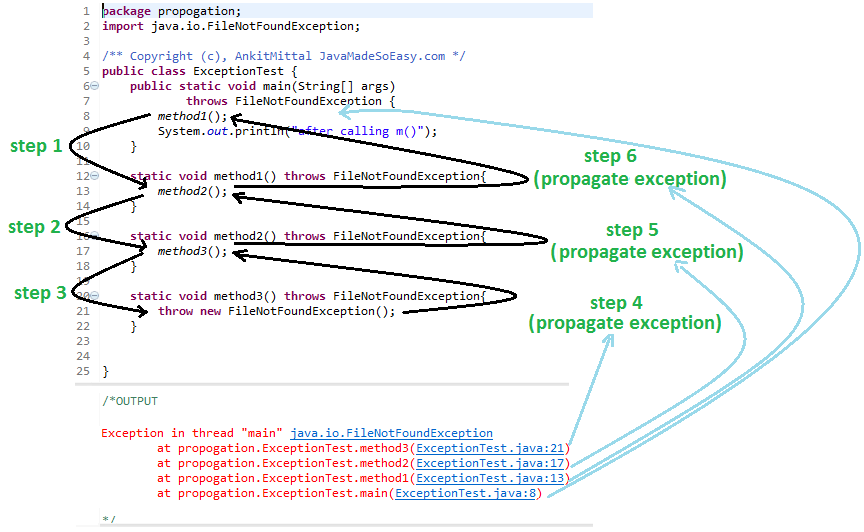


In the above program, stack is formed and an exception is first thrown from the top of the stack [ **method3()** ] and it remains uncaught there, and starts coming down the stack to previous methods to **method2()**,then to **method1()**,than to **main()** and it remains uncaught throughout.

exception remains uncaught even after reaching bottom of the stack [ **main()** ] so it is propagated to JVM and ultimately program is terminated by throwing exception [ as shown in output ] in java.

*Propagating* ***checked*** *exception (FileNotFoundException) using throws keyword >*

For **propagating checked** exceptions method must throw exception by using [**throws**](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html)keyword.



**Exception interview Question 21. Can a catch or finally block throw exception in java?**

**Answer**. Yes, catch or finally block can throw checked or unchecked exception but it must be handled accordingly. Please refer this post for [handling checked and unchecked exceptions](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html) in java.

**Exception interview Question 22. Why shouldn’t you use Exception for catching all exceptions in java?**

**Answer**. Catching Exception rather than handling specific exception can be vulnerable to our application. [Multiple catch blocks](http://www.javamadesoeasy.com/2015/05/multiple-catch-block-in-java.html) must be used to catch specific exceptions, because handling specific exception gives developer the liberty of taking appropriate action and develop robust application.

**Exception interview Question 23. What is Difference between** [**multiple catch block**](http://www.javamadesoeasy.com/2015/05/multiple-catch-block-in-java.html) **and** [**multi catch syntax**](http://www.javamadesoeasy.com/2015/05/catch-block-and-automatic-resource.html)**?**

**Answer**. Experienced developers must know in detail about this Exception handling interview question in java

|  |  |  |
| --- | --- | --- |
|  | **multiple catch block** | **multi catch syntax** |
| 1 | multiple catch blocks were introduced in prior versions of Java 7 and does not provide any automatic resource management in java. | **multi catch syntax was introduced in** java 7 for improvements in multiple exception handling which helps in **automatic resource management in java.** |
| 2 | Here is the syntax for writing **multiple catch block in java** >   |  | | --- | | **try**{  //code . . . . .  }**catch**(**IOException** ex1){  //code . . . . .  } **catch**(**SQLException** ex2){  //code . . . . .  } | | Here is the **multi catch syntax in java** >   |  | | --- | | **try**{  //code . . . . .  }**catch**(IOException **|** SQLException ex){  //code . . . . .  } |   We could separate different exceptions using **pipe** ( **|** ) |
| 3 | For catching IOException and SQLException we need to write **two catch block** like this >  https://lh4.googleusercontent.com/Y8Pt2V80aY5BXlcs1viypo_8NBeNoVmA1Awad1o9oGIQhY02xSTW3M2fGrPGK0THNJP6yBczP-QuNHAiJjR9fMSGm1uawbwgaslhJh4KOihK-gbhHaBsPD7UYSsU2inbOxb4JPdn | with the help of multi catch syntax we can catch IOException and SQLException in one catch block using **multi catch syntax** like this >  https://lh5.googleusercontent.com/EDfjTrfY38x6H8uhRZ4ebbMwEdssvXcHaVb7dRVIr7r58vkNW4rtA8dpntIaHul_whEsS5no1B2EaawfQPTwwmOIyhjX-f0gAaRPzIHCpY36LafMmBjFpb6qPM4rQEhVb-XCEk0W |
| 4 | **When multiple catch blocks** are used , first catch block could be subclass of Exception class handled in following catch blocks like this >  IOException is subclass of Exception in java.https://lh5.googleusercontent.com/dDTyOichrPKc2g2d_KU_BLjUdPs1LpRqNum51I_x0iSD4KRaXe4T-gHeXLHEUELu4vE3W1jYt2ifV80dl0ZMaZZeRBYuIqSekdNpbUW_LdoQ30ms9gSi5Oj8kY43yAzcly5m29UC | If **Multi catch syntax** is used to catch subclass and its superclass than compilation error will be thrown.  IOException and Exception in **multi catch syntax** will cause compilation error “The exception **IOException** is already caught by the alternative **Exception**”.  https://lh4.googleusercontent.com/SFnwBVq0EsP5hKSaU7EdH35tTemyCbiqh35H-A_yd2KPEvoyzVJ7WU0y3yyafvAa0lFqSFW40Gm4b64_YaiopsmSx-hObaF6EoTNEhKihbyCIFwCb0k0lTyLU3F1pPjd5YNxuimg  **Solution >**  We must use only **Exception** to catch its subclass like this >  https://lh3.googleusercontent.com/ClWKvKo_sJKavB7eyyAgwQOHmkx_uCo3xZGxhX3_9kRUaavmPVd6dN6MM3Ix7HJ7m9ZXd5pbwyLwZ3ex8raeVavEkunmc3OyWemcUmra2fuaNtSZG1pNqz58wmSepXcuaUnaU2QF |
| 5 | Does not provide such features. | *Features of* ***multi catch syntax*** *>*   * Has **improved way of catching multiple** [**exceptions**](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html)**.** * This syntax does **not looks clumsy**. * **Reduces developer efforts** of writing multiple catch blocks. * Allows us to **catch more than one exception in one catch block**. * Helps in **automatic resource management.** |

*For more read :*[***Difference between multiple catch block and multi catch syntax***](http://www.javamadesoeasy.com/2015/05/difference-between-multiple-catch-block.html)

**Exception interview Question 24.  can a method be overloaded on basis of  exceptions in java ?**

**Answer**.

Another Exception handling interview question which will test your practical understanding of exception in java.

Yes a method be overloaded on basis of  exceptions in java.

But now question which overloaded exception will be called.

Let’s take an example :

***Ques****. Let's say one method handles Exception and other handles ArithmeticException. Which method will be invoked when ArithmeticException is thrown?*

**Ans**. Method which handles more specific exception will be called.

Program >

|  |
| --- |
| **import** java.io.IOException;  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/)  \* Main class \*/  **public** **class** ExceptionTest {    **void** method(Exception e){            System.*out*.println(e+" caught in Exception method");     }  **void** method(ArithmeticException ae){            System.*out*.println(ae+" caught in ArithmeticException method");     }    **public** **static** **void** main(String[] args) {            ExceptionTest obj=**new** ExceptionTest();            obj.method(**new** ArithmeticException());            obj.method(**new** IOException());     }  }  /\* OUTPUT  java.lang.ArithmeticException caught in ArithmeticException method  java.io.IOException caught in Exception method  \*/ |

**Exception interview Question 25.  Mention few exception handling best practices in java?**

**Answer**. Experienced developers must be able to answer this Exception handling interview question in detail in java.

* [**Throw**](http://www.javamadesoeasy.com/2015/05/throw-exception-in-java.html)[**exceptions**](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html) **when the method cannot handle the exception**, and more **importantly**, must be **handled by the caller**.

**Example -**

In Servlets - doGet() and doPost() throw ServletException or IOException in certain circumstances where the request could not be read correctly. Neither of these methods are in a position to handle the exception, but the container is (which generally results in the 404 error page in most cases).

* **Bubble the exception if the method cannot handle it**. This is a corollary of the above point, but applicable to methods that must catch the exception. If the caught exception cannot be handled correctly by the method, then it is preferable to bubble it.
* **Throw the exception right away**. If an exception scenario is encountered, then it is a good practice to throw an exception indicating the original point of failure, instead of attempting to handle the failure via error codes, until a point deemed suitable for throwing the exception. In other words, attempt to minimize mixing exception handling with error handling.
* **Either log the exception or bubble it, but don't do both**. Logging an exception often indicates that the exception stack has been completely unwound, indicating that no further bubbling of the exception has occurred. Hence, it is not recommended to do both at the same time, as it often leads to a frustrating experience in debugging.
* **Application should not try to catch** [**Error**](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html) **-** Because, in most of cases recovery from an Error is almost impossible. So, application must be allowed to terminate.

**Example>**

Let’s say errors like OutOfMemoryError and StackOverflowError occur and are caught then JVM might not be able to free up memory for rest of application to execute, so it will be better if application don’t catch these errors and is allowed to terminate.

Read : Complete list of [**Exception handling best practices and guidelines for using exceptions in java**](http://www.javamadesoeasy.com/2015/05/exception-handling-best-practices-and.html)

**Exception interview Question 26.  Difference between Final, Finally and Finalize in java?**

**Answer**. It is another very very important exception interview question to differentiate between final, finally and finalize in java.

|  |  |  |  |
| --- | --- | --- | --- |
|  | [***final***](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) | [***finally***](http://www.javamadesoeasy.com/2015/05/finally-block-in-java.html) | [***finalize***](http://www.javamadesoeasy.com/2015/05/finalize-method-in-java-10-salient.html) |
| **1** | [final](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) can be applied to **variable**, **method** and **class** in java. | [finally](http://www.javamadesoeasy.com/2015/05/finally-block-in-java.html) is a block. | [finalize](http://www.javamadesoeasy.com/2015/05/finalize-method-in-java-10-salient.html) is a method. |
| **2** | ***2.1) Final variable***  **final memberVariable**  **final local variable**  **final static variable**  **Final memberVariable** of class must be initialized at time of declaration, once initialized final memberVariable cannot be assigned a new value.  Final variables are called **constants** in java.   |  | | --- | | **class** FinalTest {  **final** **int** x=1; //memberVariable/instanceVariable  } |   If constructor is defined then final memberVariable can be initialized in constructor but  once initialized cannot be assigned a new value.   |  | | --- | | **class** FinalTest {  **final** **int** x; //memberVariable/instanceVariable     FinalTest() {            x = 1; //final memberVariable can be initialized in constructor.     }  } |   **Final local variable** can be left uninitialized at time of declaration and can be initialized later, but once initialized cannot be assigned a new value in java.   |  | | --- | | **class** FinalTest {  **void** method(){  **final** **int** x; //uninitialized at time of declaration        x=1;    }  } |   **Final static variable** of class must be initialized at time of declaration or can be initialized in static block, once initialized final static variable cannot be assigned a new value.  If static block is defined then final static variable can be initialized in static block, once initialized final static variable cannot be assigned a new value.   |  | | --- | | **class** FinalTest {  **final** **static** **int** *x*; //static variable  **static**{ //static block  *x*=1;   }  } |   ***2.2) Final method***  **Final method** cannot be overridden, any attempt to do so will cause compilation error.  https://lh6.googleusercontent.com/9O1O5MRwg1NVPsVCuL-i3-M2HsZ4vFfyEkajayflbtUXS3PMil96r-z69VJigDDJKXyy4KE8UJbosx66qyqbojmEcvGAW__dHg6oKVB8-_cus4a6r_peqKFO-hmqx8CWgoWIOgh8  Runtime polymorphism is not applicable on final methods because they cannot be inherited.  ***2.3) Final class***  **Final class** cannot be extended, any attempt to do so will cause compilation error.  https://lh3.googleusercontent.com/OWVRfY8Eqw9fRdxNxyFG6OAnxBnLo79ffiMTMK6vWzNgcb0yjyW9d5FzlIaRSNH9i4TazrkZxRnlN3spCOmw54iXR6AHwegwwF-3R1tZKHB0I3K88aCVaqHxvsWlr9ZFL5TYDbaK | *try or* [*try-catch*](http://www.javamadesoeasy.com/2015/05/try-catch-finally-block-in-java.html) *block can be followed by finally block >*  try**-finally** block, or   |  | | --- | | **try{**  **//Code to be enclosed in try-finally block**  **}finally{**  **}** |   try-catch**-finally** block.   |  | | --- | | **try{**  **//Code to be enclosed in try-catch-finally block**  **}catch(Exception e){**  **}finally{**  **}** |   ***finally block*** *can can only exist if try or try-catch block is there, finally block can’t be used alone in java.*  ***finally block is not executed in following scenarios >***  finally is not executed when **System.exit** is called.  if in case **JVM crashes** because of some java.util.[**Error**](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html). | finalize method is called before garbage collection by JVM,  finalize method is called for any cleanup action that may be required before garbage collection.   |  | | --- | | /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  @Override  **protected void finalize() throws Throwable** {  **try** {        System.*out*.println("in   finalize() method, "                                +   "doing cleanup activity");    } **catch** (Throwable throwable) {  **throw** throwable;  }  } |   finalize() method is defined **in java.lang.Object** |
| **3** | - | finally block can only exist if try or try-catch block is there, finally block can’t be used alone in java. | We can *force early garbage collection in java* by using following methods >  **System.*gc*(); Runtime.*getRuntime*().gc();**  **System.*runFinalization*(); Runtime.*getRuntime*().runFinalization();** |
| **4** | - | finally is always executed irrespective of exception thrown in java. | If any uncaught [exception](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html) is thrown inside finalize method -  **exception is ignored,**  **thread is terminated and**  **object is discarded.**  **Note :** Any exception thrown by the finalize method causes the finalization of this object to be halted, but is otherwise ignored. |
| **5** | - | Currently executing thread calls finally method in java. | JVM does not guarantee which [***daemon***](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html)[thread](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) will invoke the finalize method for an object. |
| **6** | final is a keyword in java. | finally Is a keyword in java. | finalize is not a keyword in java. |

For more detail read : [**Final, Finally and Finalize - difference and similarity in java**](http://www.javamadesoeasy.com/2015/05/final-finally-and-finalize-difference.html)

Another Exception interview Question.  **What are the differences between** [**between ClassNotFoundException and NoClassDefFoundError in java ?**](http://www.javamadesoeasy.com/2015/12/what-is-difference-between.html)

**Answer**.

|  |  |  |
| --- | --- | --- |
|  | ***ClassNotFoundException*** | ***NoClassDefFoundError*** |
| 1 | ClassNotFoundException is [Checked (compile time) **Exception** in java.](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html) | NoClassDefFoundError is a [Error](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html) in java. Error and its subclasses are regarded as unchecked exceptions in java. |
| 2 | Here is the hierarchy of java.lang.ClassNotFoundException - -java.lang.Object  -java.lang.Throwable   -java.lang.[Exception](http://www.javamadesoeasy.com/2015/05/exception-handling-exception-hierarchy.html)    -java.lang.ReflectiveOperationException     -java.lang.ClassNotFoundException | Here is the hierarchy of java.lang.NoClassDefFoundError - -java.lang.Object  -java.lang.Throwable   -java.lang.[Error](http://www.javamadesoeasy.com/2015/05/javalangerror-in-exception-handling-in.html)    -java.lang.LinkageError     -java.lang.NoClassDefFoundError |
| 3 | **ClassNotFoundException** is thrown when JVM tries to class from classpath but it does not find that class. | **NoClassDefFoundError** is thrown when JVM tries to load class which >   * **was NOT** available at **runtime** but * **was** available at **compile** time. |
|  | **ExceptionInInitializerError** has got nothing to do with **ClassNotFoundException**. | You must ensure that class does not throws **java.lang.ExceptionInInitializerError** because that is likely to be followed by **NoClassDefFoundError**. |

# ThreadPoolExecutor – Java Thread Pool Example: Start

**Java thread pool** manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed. We can use ThreadPoolExecutor to create thread pool in java.

Java thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue. **java.util.concurrent.Executors** provide implementation of **java.util.concurrent.Executor**interface to create the thread pool in java. Let’s write a simple program to explain it’s working.

First, we need to have a Runnable class, named WorkerThread.java

package com.journaldev.threadpool;

public class WorkerThread implements Runnable {

private String command;

public WorkerThread(String s){

this.command=s;

}

@Override

public void run() {

System.out.println(Thread.currentThread().getName()+" Start. Command = "+command);

processCommand();

System.out.println(Thread.currentThread().getName()+" End.");

}

private void processCommand() {

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public String toString(){

return this.command;

}

}

## ExecutorService Example

Here is the test program class SimpleThreadPool.java, where we are creating fixed thread pool from **Executors framework**.

package com.journaldev.threadpool;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class SimpleThreadPool {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(5);

for (int i = 0; i < 10; i++) {

Runnable worker = new WorkerThread("" + i);

executor.execute(worker);

}

executor.shutdown();

while (!executor.isTerminated()) {

}

System.out.println("Finished all threads");

}

}

In above program, we are creating fixed size thread pool of 5 worker threads. Then we are submitting 10 jobs to this pool, since the pool size is 5, it will start working on 5 jobs and other jobs will be in wait state, as soon as one of the job is finished, another job from the wait queue will be picked up by worker thread and get's executed.

Here is the output of the above program.

pool-1-thread-2 Start. Command = 1

pool-1-thread-4 Start. Command = 3

pool-1-thread-1 Start. Command = 0

pool-1-thread-3 Start. Command = 2

pool-1-thread-5 Start. Command = 4

pool-1-thread-4 End.

pool-1-thread-5 End.

pool-1-thread-1 End.

pool-1-thread-3 End.

pool-1-thread-3 Start. Command = 8

pool-1-thread-2 End.

pool-1-thread-2 Start. Command = 9

pool-1-thread-1 Start. Command = 7

pool-1-thread-5 Start. Command = 6

pool-1-thread-4 Start. Command = 5

pool-1-thread-2 End.

pool-1-thread-4 End.

pool-1-thread-3 End.

pool-1-thread-5 End.

pool-1-thread-1 End.

Finished all threads

The output confirms that there are five threads in the pool named from "pool-1-thread-1" to "pool-1-thread-5" and they are responsible to execute the submitted tasks to the pool.

## ThreadPoolExecutor Example

**Executors** class provide simple implementation of **ExecutorService** using **ThreadPoolExecutor** but ThreadPoolExecutor provides much more feature than that. We can specify the number of threads that will be alive when we create ThreadPoolExecutor instance and we can limit the size of thread pool and create our own **RejectedExecutionHandler** implementation to handle the jobs that can't fit in the worker queue.

Here is our custom implementation of RejectedExecutionHandler interface.

package com.journaldev.threadpool;

import java.util.concurrent.RejectedExecutionHandler;

import java.util.concurrent.ThreadPoolExecutor;

public class RejectedExecutionHandlerImpl implements RejectedExecutionHandler {

@Override

public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {

System.out.println(r.toString() + " is rejected");

}

}

ThreadPoolExecutor provides several methods using which we can find out the current state of executor, pool size, active thread count and task count. So I have a monitor thread that will print the executor information at certain time interval.

package com.journaldev.threadpool;

import java.util.concurrent.ThreadPoolExecutor;

public class MyMonitorThread implements Runnable

{

private ThreadPoolExecutor executor;

private int seconds;

private boolean run=true;

public MyMonitorThread(ThreadPoolExecutor executor, int delay)

{

this.executor = executor;

this.seconds=delay;

}

public void shutdown(){

this.run=false;

}

@Override

public void run()

{

while(run){

System.out.println(

String.format("[monitor] [%d/%d] Active: %d, Completed: %d, Task: %d, isShutdown: %s, isTerminated: %s",

this.executor.getPoolSize(),

this.executor.getCorePoolSize(),

this.executor.getActiveCount(),

this.executor.getCompletedTaskCount(),

this.executor.getTaskCount(),

this.executor.isShutdown(),

this.executor.isTerminated()));

try {

Thread.sleep(seconds\*1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

Here is the thread pool implementation example using **ThreadPoolExecutor**.

package com.journaldev.threadpool;

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.Executors;

import java.util.concurrent.ThreadFactory;

import java.util.concurrent.ThreadPoolExecutor;

import java.util.concurrent.TimeUnit;

public class WorkerPool {

public static void main(String args[]) throws InterruptedException{

//RejectedExecutionHandler implementation

RejectedExecutionHandlerImpl rejectionHandler = new RejectedExecutionHandlerImpl();

//Get the ThreadFactory implementation to use

ThreadFactory threadFactory = Executors.defaultThreadFactory();

//creating the ThreadPoolExecutor

ThreadPoolExecutor executorPool = new ThreadPoolExecutor(2, 4, 10, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(2), threadFactory, rejectionHandler);

//start the monitoring thread

MyMonitorThread monitor = new MyMonitorThread(executorPool, 3);

Thread monitorThread = new Thread(monitor);

monitorThread.start();

//submit work to the thread pool

for(int i=0; i<10; i++){

executorPool.execute(new WorkerThread("cmd"+i));

}

Thread.sleep(30000);

//shut down the pool

executorPool.shutdown();

//shut down the monitor thread

Thread.sleep(5000);

monitor.shutdown();

}

}

Notice that while initializing the ThreadPoolExecutor, we are keeping initial pool size as 2, maximum pool size to 4 and work queue size as 2. So if there are 4 running tasks and more tasks are submitted, the work queue will hold only 2 of them and rest of them will be handled by RejectedExecutionHandlerImpl.

Here is the output of above program that confirms above statement.

pool-1-thread-1 Start. Command = cmd0

pool-1-thread-4 Start. Command = cmd5

cmd6 is rejected

pool-1-thread-3 Start. Command = cmd4

pool-1-thread-2 Start. Command = cmd1

cmd7 is rejected

cmd8 is rejected

cmd9 is rejected

[monitor] [0/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated: false

[monitor] [4/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated: false

pool-1-thread-4 End.

pool-1-thread-1 End.

pool-1-thread-2 End.

pool-1-thread-3 End.

pool-1-thread-1 Start. Command = cmd3

pool-1-thread-4 Start. Command = cmd2

[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated: false

[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated: false

pool-1-thread-1 End.

pool-1-thread-4 End.

[monitor] [4/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated: true

[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated: true

Notice the change in active, completed and total completed task count of the executor. We can invoke **shutdown()** method to finish execution of all the submitted tasks and terminate the thread pool.

**Why Do We Need ThreadPool ?**

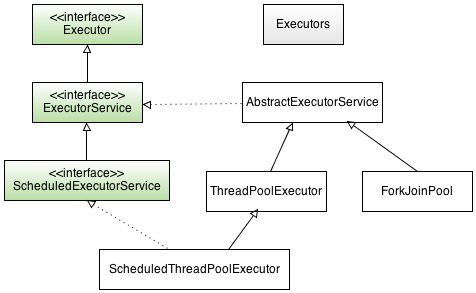
In a large scale web application, if only one thread / process is used to process all client requests, then only one client can access the server at a time.  In order to support large number of clients, we need to use one thread per request paradigm i.e. we need to use separate threads to process each client request. But this require new threads to be created, when request arrived.

Creating a thread in java is a very expensive process which includes memory overhead. So, it’s a good idea if we can re-use these threads once created, to run our future runnables. Thread pool solves this problem for us. It creates Thread and manages them. Instead of creating Thread and discarding them once task is done, thread-pool reuses threads in form of worker thread.

**Executor Framework**

Java 1.5 introduced Thread pool in Java in the form of Executor framework, which allows Java programmer to decouple submission of a task to execution of the task.

java.util.concurrent package hierarchy :



**1) Executor**

Executor interface is the core of executor framework. It has two sub interfaces ExecutorService and ScheduledExecutorService.  An object of type Executor can execute Runnable and Callable tasks.

The ExecutorService feature came with Java 5. It extends the Executor interface and provides a thread pool feature to execute asynchronous short tasks.   
  
There are five ways to execute the tasks asyncronously by using the ExecutorService interface provided Java 6.   
  
**ExecutorService execService = Executors.newCachedThreadPool();**   
  
This approach creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for 60 seconds are terminated and removed from the cache.   
  
**ExecutorService execService = Executors.newFixedThreadPool(10);**   
  
This approach creates a thread pool that reuses a fixed number of threads. Created nThreads will be active at the runtime. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available.   
  
**ExecutorService execService = Executors.newSingleThreadExecutor();**   
  
This approach creates an Executor that uses a single worker thread operating off an unbounded queue. Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time.   
  
**Methods of the ExecutorService :**   
  
**execute(Runnable) :** Executes the given command at some time in the future.   
  
**submit(Runnable) :** Submit method returns a Future Object which represents executed task. Future Object returns null if the task has finished correctly.   
  
**shutdown() :** Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. Invocation has no additional effect if already shut down.   
  
**shutdownNow() :** Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.   
  
There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. For example, typical implementations will cancel via Thread.interrupt, so any task that fails to respond to interrupts may never terminate.   
  
A sample application is below : 

**Benefits Of Using The Thread Pool**

1. **Better performance** : Creating a brand new thread each time a new task arrives is expensive. Threadpool saves time because it re-uses the threads once created, to run future runnables.
2. You do not have to create, manage, schedule, and terminate your thread, Executor framework does all of this for you. In other words, Thread Pool frees application developer from thread management stuff and allows to focus on business logic.
3. Use of Threadpool allows you to change your execution policy as you need. you can go from single thread to multiple threads by just replacing ExecutorService implementation.

**Limitations Of Using The Thread Pool**

While the thread pool is a powerful mechanism for structuring multithreaded applications, it is not without risk such as pool-related deadlock, resource thrashing and thread leakage.

1. **Resource thrashing**

If the thread pool size is not tuned properly threads consume numerous resources, including memory and other system resources. While the scheduling overhead of switching between threads is small, with many threads context switching can become a significant drag on your program’s performance.

1. **Thread leakage**

A significant risk in all kinds of thread pools is thread leakage, which occurs when a thread is removed from the pool to perform a task, but is not returned to the pool when the task completes.

One way this happens is when the task throws a RuntimeException or an Error. If the pool class does not catch these, then the thread will simply exit and the size of the thread pool will be permanently reduced by one.

When this happens enough times, the thread pool will eventually be empty, and the system will stall because no threads are available to process tasks.

While using Executor framework, we don't need to worry about thread leakages anymore.

# ThreadPoolExecutor – Java Thread Pool Example: End

# static keyword in java

[**2.3**](http://www.geeksforgeeks.org/easy/)

static is a non-access modifier in Java which is applicable for the following:

1. blocks
2. variables
3. methods
4. nested classes

To create a static member(block,variable,method,nested class), precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. For example, in below java program, we are accessing static method m1() without creating any object of Test class.

|  |
| --- |
| // Java program to demonstrate that a static member  // can be accessed before instantiating a class  class Test  {      // static method      static void m1()      {          System.out.println("from m1");      }        public static void main(String[] args)      {            // calling m1 without creating            // any object of class Test             m1();      }  } |

Run on IDE

Output:

from m1

**Static blocks**

If you need to do computation to initialize your **static variables**, you can declare a static block that gets executed exactly once, when the class is first loaded. Consider the following java program demonstrating use of static blocks.

|  |
| --- |
| // Java program to demonstrate use of static blocks  class Test  {      // static variable      static int a = 10;      static int b;        // static block      static {          System.out.println("Static block initialized.");          b = a \* 4;      }        public static void main(String[] args)      {         System.out.println("from main");         System.out.println("Value of a : "+a);         System.out.println("Value of b : "+b);      }  } |

Run on IDE

Output:

Static block initialized.

from main

Value of a : 10

Value of b : 40

For Detailed article on static blocks, see [static blocks](http://www.geeksforgeeks.org/g-fact-79/)

**Static variables**

When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

**Important points for static variables:-**

* We can create static variables at class-level only. See [here](http://www.geeksforgeeks.org/g-fact-47/)
* static block and static variables are executed in order they are present in a program.

Below is the java program to demonstrate that static block and static variables are executed in order they are present in a program.

|  |
| --- |
| // java program to demonstrate execution  // of static blocks and variables  class Test  {      // static variable      static int a = m1();        // static block      static {          System.out.println("Inside static block");      }        // static method      static int m1() {          System.out.println("from m1");          return 20;      }        // static method(main !!)      public static void main(String[] args)      {         System.out.println("Value of a : "+a);         System.out.println("from main");      }      } |

Run on IDE

Output:

from m1

Inside static block

Value of a : 20

from main

**Static methods**

When a method is declared with static keyword, it is known as static method. The most common example of a static method is main( ) method.As discussed above, Any static member can be accessed before any objects of its class are created, and without reference to any object.Methods declared as static have several restrictions:

* They can only directly call other static methods.
* They can only directly access static data.
* They cannot refer to [this](http://www.geeksforgeeks.org/this-reference-in-java/) or [super](http://www.geeksforgeeks.org/super-keyword/) in any way.

Below is the java program to demonstrate restrictions on static methods.

|  |
| --- |
| // java program to demonstrate restriction on static methods  class Test  {      // static variable      static int a = 10;        // instance variable      int b = 20;        // static method      static void m1()      {          a = 20;          System.out.println("from m1");             // Cannot make a static reference to the non-static field b           b = 10; // compilation error             // Cannot make a static reference to the                   // non-static method m2() from the type Test           m2();  // compilation error             //  Cannot use super in a static context           System.out.println(super.a); // compiler error      }        // instance method      void m2()      {          System.out.println("from m2");      }            public static void main(String[] args)      {          // main method      }  } |

Run on IDE

**When to use static variables and methods?**

Use the static variable for the property that is common to all objects. For example, in class Student, all students shares the same college name. Use static methods for changing static variables.

Consider the following java program, that illustrate the use of static keyword with variables and methods.

### How ArrayList Works Internally in Java

ArrayList arguably would be the most used collection along with the [HashMap](https://netjs.blogspot.com/2015/11/difference-between-hashmap-and-hashtable-java.html). Many of us programmers whip up code everyday which contains atleast one of these data structures to hold objects. I have already discussed [how HashMap works internally in Java](https://netjs.blogspot.com/2015/05/how-hashmap-internally-works-in-java.html), in this post I'll try to explain **how ArrayList internally works in Java**.

As most of us would already be knowing that **ArrayList is a Resizable-array implementation** of the List interface i.e. ArrayList grows dynamically as the elements are added to it. So let's try to get clear idea about the following points-

### Where does ArrayList internally store elements

Basic data structure used by Java ArrayList to store objects is an [array](http://netjs.blogspot.com/2017/02/array-in-java.html) of [Object class](http://netjs.blogspot.com/2017/06/object-class-in-java.html), which is defined as follows -

transient Object[] elementData;

I am sure many of you would be thinking why [transient](http://netjs.blogspot.com/2017/04/transient-in-java.html) and how about [serializing](https://netjs.blogspot.com/2017/04/serialization-in-java.html) an ArrayList then?  
ArrayList provides its own version of **readObject** and **writeObject** methods so no problem in serializing an ArrayList and that is the reason, I think, of making this Object array as **transient**.

### What happens when ArrayList is created

ArrayList class in Java provides **3 constructors** to create an ArrayList.

* **public ArrayList(int initialCapacity)** - When this [constructor](http://netjs.blogspot.com/2015/04/constructor-in-java.html) is used we can provide some initial capacity rather than depending on the default capacity as defined in the ArrayList class.  
  **As example -**
* List<String> myList = new ArrayList<String>(7);

Code in the ArrayList class is as -

public ArrayList(int initialCapacity) {

if (initialCapacity > 0) {

this.elementData = new Object[initialCapacity];

} else if (initialCapacity == 0) {

this.elementData = EMPTY\_ELEMENTDATA;

} else {

throw new IllegalArgumentException("Illegal Capacity: "+

initialCapacity);

}

}

Where **EMPTY\_ELEMENTDATA** is defined as -

private static final Object[] EMPTY\_ELEMENTDATA = {};

It is easy to see that, if provided capacity is greater than zero then the elementData array will be created with that capacity, in case provided capacity is zero then elementData array is initialized with an empty Object array. In that case ArrayList will grow when first element is added.

* **public ArrayList()** - In case **default constructor** is used i.e. ArrayList is created like -
* myList = new ArrayList();

Code in the ArrayList class is as -

public ArrayList() {

this.elementData = DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA;

}

Where **DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA** is defined as

/\*\*

\* Shared empty array instance used for default sized empty instances. We

\* distinguish this from EMPTY\_ELEMENTDATA to know how much to inflate when

\* first element is added.

\*/

private static final Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {};

So you can see initially it will be initialized with an empty array, it will grow only when first element is added to the list.

* **public ArrayList(Collection<? extends E> c)** - If we want to construct a list containing the elements of the specified collection we can use this [constructor](http://netjs.blogspot.com/2015/04/constructor-chaining-in-java-calling-one-constructor-from-another.html). In this constructor implementation checks for the length of the collection passed as parameter, if length is greater than zero then **Arrays.copyOf** method is used to copy the collection to the elementData array.
* elementData = Arrays.copyOf(elementData, size, Object[].class);

### How does ArrayList grow dynamically

When we add an element to an ArrayList it first verifies whether it has that much capacity in the array to store new element or not, in case there is not then the new capacity is calculated which is 50% more than the old capacity and the array is increased by that much capacity (Actually uses Arrays.copyOf which returns the original array increased to the new length).

Code in the Java ArrayList implementation is like this-

public boolean add(E e) {

ensureCapacityInternal(size + 1); // Increments modCount!!

elementData[size++] = e;

return true;

}

private void ensureCapacityInternal(int minCapacity) {

if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {

minCapacity = Math.max(DEFAULT\_CAPACITY, minCapacity);

}

ensureExplicitCapacity(minCapacity);

}

Where DEFAULT\_CAPACITY is defined as -

private static final int DEFAULT\_CAPACITY = 10;

private void ensureExplicitCapacity(int minCapacity) {

modCount++;

// overflow-conscious code

if (minCapacity - elementData.length > 0)

grow(minCapacity);

}

You can see here it is determined if there is a need to increase the size of the array, if yes then grow method is called.

private void grow(int minCapacity) {

// overflow-conscious code

int oldCapacity = elementData.length;

int newCapacity = oldCapacity + (oldCapacity >> 1);

if (newCapacity - minCapacity < 0)

newCapacity = minCapacity;

if (newCapacity - MAX\_ARRAY\_SIZE > 0)

newCapacity = hugeCapacity(minCapacity);

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.copyOf(elementData, newCapacity);

}

Note that till **Java 6** the new capacity calculation used to be like this -

int newCapacity = (oldCapacity \* 3)/2 + 1;

Which is changed in **Java 7** to use right shift operator. With right shift operator also it will grow by 50% of old capacity.  
Let's see it with the help of a small program

public class Test {

public static void main(String args[]) {

int a = 10;

System.out.println(a>>1);

}

}

**Output**

5

If the default capacity was 10 then

int newCapacity = oldCapacity + (oldCapacity >> 1);

will return 15.

### What happens when an element is removed from ArrayList

When elements are removed from an ArrayList in Java using either **remove(int i)** (i.e using index) or **remove(Object o)**, gap created by the removal of an element has to be filled in the underlying array. That is done by Shifting any subsequent elements to the left (subtracts one from their indices). **System.arrayCopy** method is used for that.

System.arraycopy(elementData, index+1, elementData, index, numMoved);

Here index+1 is the source position and index is the destination position. Since element at the position index is removed so elements starting from index+1 are copied to destination starting from index.

**Points to note**

1. ArrayList in Java is a Resizable-array implementation of the List interface.
2. Internally ArrayList class uses an array of Object class to store its elements.
3. When initializing an ArrayList you can provide initial capacity then the array would be of the size provided as initial capacity.
4. If initial capacity is not specified then default capacity is used to create an array. Default capacity is 10.
5. When an element is added to an ArrayList it first verifies whether it can accommodate the new element or it needs to grow, in case capacity has to be increased then the new capacity is calculated which is 50% more than the old capacity and the array is increased by that much capacity.
6. When elements are removed from an ArrayList space created by the removal of an element has to be filled in the underlying array. That is done by Shifting any subsequent elements to the left.

# HashMap internal implementation

How a HashMap works internally has become a popular question in almost all interviews. Almost everybody knows how to use a HashMap or the [difference between HashMap and Hashtable](http://www.javainterviewpoint.com/differences-betwen-hashmap-hashtable/). However, many people fail when the question is "How does a HashMap work internally?"

The answer to this question is that it works based on the hashing principle, but it is not as simple as it sounds. Hashing is the mechanism of assigning unique code to a variable or attribute using an algorithm to enable easy retrieval. A true hashing mechanism should always return the same hashCode() when it is applied to the same object.

Then comes the question, "How does hashing help in storing and retrieving the value in HashMap?" Many will say that the value will be stored in the bucket and retrieved using the key. If you think that is how it works, then you are absolutely wrong. To prove it, let's take a look at the HashMap class:

/\*\*

\* The table, resized as necessary. Length MUST Always be a power of two.

\*/

transient Entry[] table;

So what is the use of Entry[] in a HashMap for? The HashMap stores the Objects as Entry instances, not as key and value.

## What Is Entry Class?

HashMap has an inner class called an Entry Class which holds the key and values. There is also something called next, which you will get to know a bit later.

static class Entry<K,V> implements Map.Entry<K,V>

{

final K key;

V value;

Entry<K,V> next;

final int hash;

........

}

You know that the HashMap stores the Entry instances in an array and not as key-value pairs. In order to store a value, you will use the put() method of the HashMap. Let's dig into that and see how it works.

## How Does the Put() Method Work Internally?

The code will look like this:

public V put(K key, V value)

{

if (key == null)

return putForNullKey(value);

int hash = hash(key.hashCode());

int i = indexFor(hash, table.length);

for (Entry<K,V> e = table[i]; e != null; e = e.next)

{

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

{

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

First, it checks if the key given is null or not. If the given key is null, it will be stored in the zero position, as the hashcode of null will be zero.

Then it applies the hashcode to the key .hashCode() by calling the hashcode method. In order to get the value within the limits of an array, the hash(key.hashCode()) is called, which performs some shifting operations on the hashcode.

The indexFor() method is used to get the exact location to store the Entry object.

Then comes the most important part — if two different objects have the same hashcode (e.g. Aa and BB will have the same hashcode), will it be stored in the same bucket? To handle this, let's think of the LinkedList in the data structure. It will have a "next" attribute, which will always point to the next object, the same way the next attribute in the Entry class points to the next object. Using this different objects with the same hashcode will be placed next to each other.

In the case of the Collision, the HashMap checks for the value of the next attribute. If it is null, it inserts the Entry object in that location. If the next attribute is not null, then it keeps the loop running until the next attribute is null then stores the Entry object there.

## How Are Duplicate Keys Prevented in HashMap?

As we all know, HashMap doesn't allow duplicate keys, even though when we insert the same key with different values, only the latest value is returned.

import java.util.HashMap;

import java.util.Map;

public class HashMapEg {

public static void main(String[] args) {

Map map = new HashMap();

map.put(1, "sam");

map.put(1, "Ian");

map.put(1, "Scott");

map.put(null, "asdf");

System.out.println(map);

}

}

For the above code, you will get the output {null=asdf, 1=Scott}, as the values sam  and Ian  will be replaced by Scott. So, how does this happen?

All the Entry Objects in the LinkedList will have the same hashcode, but HashMap uses equals() . This method checks the equality, so if key.equals(k) is true, it will replace the value object inside the Entry class and not the key. This way, it prevents the duplicate key from being inserted.

## How Does the Get() Method Work Internally?

Almost the same logic applied in the put() method will be used to retrieve the value.

public V get(Object key)

{

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<K,V> e = table[indexFor(hash, table.length)];e != null;e = e.next)

{

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

1. First, it gets the hash code of the key object, which is passed, and finds the bucket location.
2. If the correct bucket is found, it returns the value.
3. If no match is found, it returns null.

## What Happens If Two Keys Have the Same Hashcode?

The same collision resolution mechanism will be used here. key.equals(k) will check until it is true, and if it is true, it returns the value of it.

# HashSet Internal implementation in Java

Not many programmer know that HashSet is internally implemented using HashMap in Java, so if you know [How HashMap works internally in Java](http://java67.blogspot.sg/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html), more likely you can figure out *how HashSet works in Java*. But, now a curious Java developer can question that, how come HashSetuses HashMap, because you need a key value pair to use with Map, while in HashSet we only store one object. Good question, isn't it? If you remember some functionality of earlier class, then you know that [HashMap allows duplicate values](http://java67.blogspot.sg/2013/02/10-examples-of-hashmap-in-java-programming-tutorial.html) and this property is exploited while implementing HashSet in Java. Since HashSet implements Set interface it needs to guarantee uniqueness and this is achieved by storing elements as keys with same value always. Things gets clear by checking HashSet.java from JDK source code. All you need to look at is, how elements are stored in HashSet and how they are retrieved from HashSet. Since HashSet doesn't provide any direct method for retrieving object e.g. get(Key key) from HashMap or get(int index) from List, only way to get object from HashSet is via Iterator. See [here](http://java67.blogspot.sg/2012/10/how-to-iterate-over-hashset-in-java.html)for code example of iterating over HashSet in Java. When you create an object of HashSet in Java, it internally create instance of backup Map with default initial capacity 16 and default load factor 0.75 as shown below :  
  
/\*\*

\* Constructs a new, empty set; the backing <tt>HashMap</tt> instance has

\* default initial capacity (16) and load factor (0.75).

\*/

public **HashSet**() {

map **=** **new** **HashMap**<>();

}

Now let's see the code for add() and iterate() method from java.util.HashSet in Java to understand *how HashSet works internally in Java*.  
  
  
  
**How Object is stored in HashSet**  
As you can see below, a call to add(Object) is delegate to put(Key, Value) internally, where Key is the object you have passed and value is another object,  called PRESENT, which is a constant in java.util.HashSet as shown below :

**private** transient **HashMap**<E,**Object**> map;

// Dummy value to associate with an Object in the backing Map

**private** static **final** **Object** **PRESENT** **=** **new** **Object**();

public boolean add(E e) {

**return** map.put(e, **PRESENT**)==**null**;

}

Since PRESENT is a constant, for all keys we have same value in backup HashMap called map.  
  
  
**How Object is retrieved from HashSet**  
Now let's see the code for getting iterator for traversing over HashSet in Java. iterator() method from java.util.HashSet class returns iterator for backup Map returned by map.keySet().iterator() method.  
  
       /\*\*

\* Returns an iterator over the elements in this set. The elements

\* are returned in no particular order.

\*

\* @return an Iterator over the elements in this set

\* @see ConcurrentModificationException

\*/

public **Iterator**<E> iterator() {

**return** map.keySet().iterator();

}

## How to use HashSet in Java

Using HashSet in Java is very simple, don't think it is Map but think more like Collection i.e. add elements by using add() method, check its return value to see if object already existed in HashSet or not. Similarly use iterator for retrieving element from HashSet in Java. You can also use contains() method to check if any object already exists in HashSet or not. This method use [equals() method](http://java67.blogspot.sg/2012/11/difference-between-operator-and-equals-method-in.html) for comparing object for matching. You can also use remove() method to remove object from HashSet. Since element of HashSet is used as key in backup HashMap, they must implement [equals() and hashCode()](http://java67.blogspot.com/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html) method. Immutability is not requirement but if its immutable then you can assume that object will not be changed during its stay on set. Following example demonstrate basic usage of HashSet in Java, for more advanced example, you can check [this](http://javarevisited.blogspot.sg/2012/06/hashset-in-java-10-examples-programs.html)tutorial.

**import** **java.util.HashSet**;

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

/\*\*

\* Java Program to demonstrate How HashSet works internally in Java.

\* @author http://java67.blogspot.com

\*/

**public** **class** **HashSetDemo**{

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

HashSet<String> supportedCurrencies = **new** HashSet<String>();

// adding object into HashSet, this will be translated to put() calls

supportedCurrencies.add("USD");

supportedCurrencies.add("EUR");

supportedCurrencies.add("JPY");

supportedCurrencies.add("GBP");

supportedCurrencies.add("INR");

supportedCurrencies.add("CAD");

// retrieving object from HashSet

Iterator<String> itr = supportedCurrencies.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

}

}

Output

JPY

EUR

INR

USD

CAD

GBP

That's all about **How HashSet is implemented in Java** and **How HashSet works internally**. As I said, If you how HashMap internally in Java, you can explain working of HashSet provided,  you know it uses same values for all keys. Remember to override equals() and hashCode() for any object you are going to store in HashSet, since your object is used as key in backup Map, it must override those method. Make your object Immutable or effective immutable if possible.

Let us first define Overloading and Overriding.

[**Overriding**](http://www.geeksforgeeks.org/overriding-in-java/): Overriding is a feature of OOP languages like Java that is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in superclass (or base class).  
The implementation to be executed is decided at run-time and decision is made according to the object used for call. Note that signatures of both methods must be same. Refer [Overriding in Java](http://www.geeksforgeeks.org/overriding-in-java/) for details.

[**Overloading**](http://www.geeksforgeeks.org/overloading-in-java/): Overloading is also a feature of OOP languages like Java that is related to compile time (or static) polymorphism. This feature allows different methods to have same name, but different signatures, especially number of input parameters and type of input paramaters. Note that in both C++ and Java, [methods cannot be overloaded according to return type.](http://www.geeksforgeeks.org/g-fact-75/)

**Can we overload static methods?**  
The answer is ‘Yes’. We can have two ore more static methods with same name, but differences in input parameters. For example, consider the following Java program.

|  |
| --- |
| // filename Test.java  public class Test {      public static void foo() {          System.out.println("Test.foo() called ");      }      public static void foo(int a) {          System.out.println("Test.foo(int) called ");      }      public static void main(String args[])      {          Test.foo();          Test.foo(10);      }  } |

Run on IDE

Output:

Test.foo() called

Test.foo(int) called

**Can we overload methods that differ only by static keyword?**  
We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. This behaviour is same in C++ (See point 2 of [this](http://www.geeksforgeeks.org/function-overloading-in-c/)).

|  |
| --- |
| // filename Test.java  public class Test {      public static void foo() {          System.out.println("Test.foo() called ");      }      public void foo() { // Compiler Error: cannot redefine foo()          System.out.println("Test.foo(int) called ");      }      public static void main(String args[]) {          Test.foo();      }  } |

Run on IDE

Output: Compiler Error, cannot redefine foo()

# [12 Rules of Overriding in Java You Should Know](https://www.codejava.net/java-core/the-java-language/12-rules-of-overriding-in-java-you-should-know)

Overriding is a core concept in *Object Oriented Programming* as well as in Java programming language. Understanding what it is, how it works and what the rules are is very important for every Java programmer, especially those who are preparing for the [OCPJP exam](https://www.codejava.net/books/good-books-for-ocpjp-7-exams-1z0-804-and-1z0-805) (exam topic: Java Class Design > Override methods).Therefore, this tutorial compiles a comprehensive summary of concepts and rules regarding overriding in the Java language, with easy-to-understand code examples.

# 1.    What is Overriding?

**Overriding** refers to the ability of a subclass to re-implement an instance method inherited from a superclass. Let’s take a look at the following class diagram:

Here, Animal is the superclass and Dog is the subclass, thus Dog inherits the move() method from Animal. However, Dogre-implements the move() method for some behaviors which are specific to only dogs (like walk and run). In this respect:

* The Dog’s move() method is called the **overriding method**.
* The Animal’s move() method is called the **overridden method**.

Basically, the overriding method must have same name and same arguments list as the overridden one. It’s the way by which a subtype extends or re-defines behaviors of its supertype.

# 2.    What methods can be overridden?

**Rule #1:Only inherited methods can be overridden.**

Because overriding happens when a subclass re-implements a method inherited from a superclass, so only inherited methods can be overridden, that’s straightforward. That means only methods declared with the following access modifiers: **public**, **protected** and default (in the same package) can be overridden. That also means **private** methods cannot be overridden. Let’s see some examples:

* The Dog class overrides both the move() (public) and eat() (protected) methods from the Animal class (regardless of packages where the both classes are declared):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10 | public class Animal {        public void move() {          // animal moving code...      }        protected void eat() {          // animal eating code...      }  } | | |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10 | public class Dog extends Animal {        public void move() {          // dog moving code...      }        protected void eat() {          // dog eating code...      }  } | |

* In the following example, the Dog class perfectly overrides the move() method which is declared with default access modifier in the Animal class, as long as both the classes are in the same package:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | package net.codejava.core;    public class Animal {        void move() {          // Animal moving code...      }  } | | |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | package net.codejava.core;    public class Dog extends Animal {        void move() {          // Dog moving code...      }  } | |

* In the following example, the Dog and Animal classes are in different packages. Thus it isn’t considered an overriding because the Dog class does not inherit the Animal’s move() method:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | package net.codejava.animal;    public class Animal {        void move() {          // Animal moving code...      }  } | | |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10 | package net.codejava.dog;    import net.codejava.core.animal.Animal;    public class Dog extends Animal {        void move() {          // Dog moving code...      }  } | |

* Here, the Dog’s move() method is just a new method, not an overriding one.
* In the following example, the Animal’s move() method is private, so the Dog’s move() method is just a new method, not an overriding one:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Animal {        private void move() {          // Animal moving code...      }  } | | |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Dog extends Animal {        public void move() {          // Dog moving code...      }  } | |

# 3.    What methods that cannot be overridden?

**Rule #2:Final and static methods cannot be overridden.**

A final method means that it cannot be re-implemented by a subclass, thus it cannot be overridden. Consider the following example:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Animal {        final void sleep() {          // animal sleeping code...      }  } | | |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Dog extends Animal {        public void sleep() {          // Dog sleeping code...      }  } | |

Here, the Animal’s sleep() method is marked as final, therefore the Dog class won’t compile. The compiler will complain:

|  |  |
| --- | --- |
| 1  2  3  4  5 | error: sleep() in Dog cannot override sleep() in Animal      public void sleep() {                  ^    overridden method is final  1 error |

In case of static method, because a static method is available to all instances of the superclass and its subclasses, so it’s not permissible to re-implement the static method in a particular subclass. Consider the following example:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Animal {        static void sleep() {          // animal sleeping code...      }  } | | |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Dog extends Animal {        public void sleep() {          // Dog sleeping code...      }  } | |

The compiler will issue the following complaint when trying to compile the Dog class:

|  |  |
| --- | --- |
| 1  2  3  4  5 | error: sleep() in Dog cannot override sleep() in Animal      public void sleep() {                  ^    overridden method is static  1 error |

# 

This book helps you improve your Java programming skills to a new level: [**Effective Java (2nd Edition)**](https://www.amazon.com/gp/product/0321356683/ref=as_li_tf_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321356683&linkCode=as2&tag=code0ac-20)**https://ir-na.amazon-adsystem.com/e/ir?t=code0ac-20&l=am2&o=1&a=0321356683**

# 4.    Requirements for the overriding method

With respect to the overridden method, the overriding method must obey the following rules:

**Rule #3**: **The overriding method must have same argument list.**

Let’s see the following example:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Animal {        protected void eat(String food) {          // animal eating code...      }  } | | |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Dog extends Animal {        protected void eat(String food) {          // dog eating code...      }  } | |

The eat() method of the Dog class is a legal overriding, as it keeps the same argument (String food) as the superclass’ version. If we add a new argument to the method like this:

|  |  |
| --- | --- |
| 1  2  3 | protected void eat(String food, int amount) {      // dog eating code...  } |

Then this method is not an overriding, it is an overload instead.

**Rule #4**: **The overriding method must have same return type (or subtype).**

Suppose that a Food class has a subclass called DogFood, the following example shows a correct overriding:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9 | public class Animal {        protected Food seekFood() {            // animal seeking for food code...            return new Food();      }  } | | |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9 | public class Dog extends Animal {        protected Food seekFood() {            // dog seeking for food code...            return new DogFood();      }  } | |

It’s possible to modify the return type of the Dog’s seekFood() method to DogFood - a subclass of Food, as shown below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | protected DogFood seekFood() {        // dog seeking for food code...        return new DogFood();  } |

That’s perfectly a legal overriding, and the return type of Dog’s seekFood() method is known as **covariant return type**.

The Dog class won’t compile if we change the seekFood() method’s return type to another, as shown below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | protected String seekFood() {        // dog seeking for food code...        return new String();  } |

As the complier issues this error:

|  |  |
| --- | --- |
| 1  2  3  4  5 | error: seekFood() in Dog cannot override seekFood() in Animal      protected String seekFood() {                       ^    return type String is not compatible with Food  1 error |

**Rule #5**: **The overriding method must not have more restrictive access modifier.**

This rule can be understood as follows:

* If the overridden method is has default access, then the overriding one must be default, protected or public.
* If the overridden method is protected, then the overriding one must be protected or public.
* If the overridden method is public, then the overriding one must be only public.

In other words, the overriding method may have less restrictive (more relaxed) access modifier. The following example shows a legal overriding:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Animal {        protected void move() {          // animal moving code...      }  } | | |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Dog extends Animal {        public void move() {          // Dog moving code...      }  } | |

However, in the following example, the Dog class won’t compile:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Dog extends Animal {        void move() {          // Dog moving code...      }  } |

It is because the move() method now has default access, which is more restrictive than the protected access of the superclass’ version.

**Rule #6**: **The overriding method must not throw new or broader checked exceptions.**

In other words, the overriding method may throw fewer or narrower checked exceptions, or any unchecked exceptions.

Consider the following superclass - Animal:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Animal {        protected void move() throws IOException {          // animal moving code...      }  } |

The following subclass - Dog, correctly overrides the move() method because the FileNotFoundException is a subclass of the FileIOException:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Dog extends Animal {        protected void move() throws FileNotFoundException {          // Dog moving code...      }  } |

The following example shows an illegal overriding attempt because the InterruptedException is a new and checked exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Dog extends Animal {        protected void move() throws IOException, InterruptedException {          // Dog moving code...      }  } |

However, the following example is a legal overriding, because the IllegalArgumentException is an unchecked exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Dog extends Animal {        protected void move() throws IOException, IllegalArgumentException {          // Dog moving code...      }  } |

And in the example below, the Dog class won’t compile because its move() method throws Exception which is superclass (broader) of the IOException:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Dog extends Animal {        protected void move() throws Exception {          // Dog moving code...      }  } |

# [Can overridden methods differ in return type?](https://stackoverflow.com/questions/14694852/can-overridden-methods-differ-in-return-type)

# Java supports\* covariant return types for overridden methods. This means an overridden method may have a more specific return type. That is, as long as the new return type is assignable to the return type of the method you are overriding, it's allowed.

# Before Java 5.0, when you override a method, both parameters and return type must match exactly. In Java 5.0, it introduces a new facility called covariant return type. You can override a method with the same signature but returns a subclass of the object returned. In another words, a method in a subclass can return an object whose type is a subclass of the type returned by the method with the same signature in the superclass.

Broadly speaking yes return type of overriding method can be different. But its not straight forward there are some case involved in this.

Case 1: If the return type is primitive data type or void.

Output: If the return type is void or primitive then the data type of parent class method and overriding method should be same. e.g. if return type is int, float, string then it should be same

Case 2: If the return type is derived data type:

Output: If the return type of the parent class method is derived type then the return type of the overriding method is same derived data type of sub class to the derived data type. e.g. Suppose i have a class A B is subclass to A C is subclass to B D is subclass to C then if if the super class is returning type A then the overriding method is subclass can return A, B, C or D type i.e its sub types. This is also called as covarience.

yes It is possible.. returns type can be different only if parent class method return type is  
a super type of child class method return type..  
means

class ParentClass {

public Circle() method1() {

return new Cirlce();

}

}

class ChildClass extends ParentClass {

public Square method1() {

return new Square();

}

}

Class Circle {

}

class Square extends Circle {

}

# 5.    Invoking the overridden method

**Rule #7:Use the super keyword to invoke the overridden method from a subclass.**

It’s very common that a subclass extends a superclass’ behavior rather than re-implementing the behavior from scratch. In such case, invoke the superclass’ method in the following form:

**super.overriddenMethodName()**

Consider the following example:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Animal {        protected void move() {          // animal moving code...      }  } | | |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | public class Dog extends Animal {        protected void move() {          super.move();   // Animal movement            // Dog-specific moving code...      }  } | |

Here, the Dog class overrides the move() method from the Animal class. Then in the Dog’s move() method, it calls the superclass’ version of the method first, then add behaviors specific to only dogs.

# 6.    Overriding and constructor

**Rule #8:Constructors cannot be overridden.**

Because constructors are not methods and a subclass’ constructor cannot have same name as a superclass’ one, so there’s nothing relates between constructors and overriding.

# 7.    Overriding and abstract method

**Rule #9: Abstract methods must be overridden by the first concrete (non-abstract) subclass.**

Consider the following interface:

|  |  |
| --- | --- |
| 1  2  3 | public interface Animal {      void move();  } |

If an abstract class implements the above interface, then it doesn’t require the subclass to override the move() method, as shown in the following AbstractDog class:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public abstract class AbstractDog implements Animal {        protected abstract void bark();    } |

But if a concrete (non-abstract) class, says BullDog, is a subclass of the AbstractDog class or the Animal interface, then it must override all the inherited abstract methods, as shown below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class BullDog extends AbstractDog {        public void move() {            // Bulldog moves...        }        protected void bark() {            // Bulldog barks...        }    } |

In this respect, the BullDog class is said to *implement* the move() and bark() abstract methods of its supertypes - the Animal interface and the AbstractDog class. Although all the rules of overriding must be obeyed in this context, the term *implement* is more exact then the term *override*, since the overridden method is abstract.

Recommended Book: [**Core Java, Volume II--Advanced Features (9th Edition) (Core Series)**](https://www.amazon.com/gp/product/013708160X/ref=as_li_qf_sp_asin_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=013708160X&linkCode=as2&tag=codejava-article-20)

# 8.    Overriding and static method

**Rule #10: A static method in a subclass may hide another static one in a superclass, and that’s called hiding.**

Consider the following example:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7 | public class Animal {        static void sleep() {          System.out.println("Animal sleeps");      }    } | | |  |  | | --- | --- | | 1  2  3  4  5  6 | public class Dog extends Animal {        static void sleep() {          System.out.println("Dog sleeps");      }  } | |

Here, the sleep() method of the Dog class is said to *hide* the sleep() method of the Animal class. When a static method of the superclass is hidden, it requires the subclass to use a fully qualified class name of the superclass to invoke the hidden method, as shown in the doSomething() method of the Dog class below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class Dog extends Animal {        static void sleep() {          System.out.println("Dog sleeps");      }        void doSomething() {          sleep();    // this calls the hiding method            // because the Animal's sleep() is hidden, it requires to use          // a fully qualified class name to access it.          Animal.sleep();      }  } |

Note that the rules of overriding are still applied for the hiding method.

# 9.    Overriding and synchronized method

**Rule #11: The synchronized modifier has no effect on the rules of overriding.**

The [**synchronized**](https://www.codejava.net/java-core/the-java-language/synchronized-keyword) modifier relates to the acquiring and releasing of a monitor object in multi-threaded context, therefore it has totally no effect on the rules of overriding. That means a synchronized method can override a non-synchronized one and vice versa.

# 10.    Overriding and strictfp method

**Rule #12: The strictfp modifier has no effect on the rules of overriding.**

That means the presence or absence of the [**strictfp**](https://www.codejava.net/java-core/the-java-language/java-keyword-strictfp) modifier has absolutely no effect on the rules of overriding: it’s possible that a FP-strict method can override a non-FP-strict one and vice-versa.

# Rules for Exception handling w.r.t MethodOverriding

# Exception-Hierarchy-Diagram

* Let us see each case with an example

### ****Rule 1:**** If parent-class method ****doesn’t****de=clare any exception

#### Then child-class overriding-method can declare **any type of unchecked exception** Note: this is the only possibility<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-no-exception-child-declares-unchecked-exception-1a.png>

#### If child-class overriding-method declares checked-exception, then compiler throws compile-time error stating CTE – “**Exception IOException is not compatible with throws clause in ParentClass.testMethod()**”<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-no-exception-child-declares-checked-exception-1b.png>

#### Then child-class overriding-method can declare **no exception** in the overriding-method of child-class **This is very much same as that of overridden-method of parent-class (exactly same method-signature)** <http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-no-exception-child-declares-no-exception-1c.png>

### ****Rule 2:**** If parent-class method declares****unchecked*–*****exception

#### Then child-class overriding-method can declare **any type of unchecked exception** Not necessarily same exception as that of parent-class’ method (only for unchecked exception)<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-unchecked-exception-child-declares-unchecked-exception-2a.png>

#### If child-class overriding-method declares any checked-exception, then compiler throws compile-time error stating CTE – “**Exception IOException is not compatible with throws clause in ParentClass.testMethod()**”<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-unchecked-exception-child-declares-checked-exception-2b.png>

#### Then child-class overriding-method can declare **no exception** in the overriding-method of child-class<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-unchecked-exception-child-declares-no-exception-2c.png>

### ****Rule 3:**** If parent-class method declares****checked****exception

#### Then child-class overriding-method can declare **any type of unchecked exception**<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-checked-exception-child-declares-unchecked-exception-3a.png>

#### Then child-class overriding-method can declare **same type of checked exception or one of its sub-class or no exception**<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-checked-exception-child-declares-same-checked-exception-3b-1.png>

#### OR, sub-type of declared checked exception<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-checked-exception-child-declares-sub-class-checked-exception-3b-2.png>

#### Then child-class overriding-method can declare **no exception** in the overriding-method of child-class<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-checked-exception-child-declares-no-exception-3c.png>

### ****Rule 4:**** If parent-class method declares combination of both****checked****&****unchecked****exceptions

#### Then child-class overriding-method can declare **any type of unchecked exception**<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-checked-and-unchecked-exception-child-declares-unchecked-exception-4a.png>

#### Then child-class overriding-method can declare **same type of checked-exception or one of its sub-class or no exception**<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-checked-and-unchecked-exception-child-declares-checked-exception-4b.png>

#### Then child-class overriding-method can declare **no exception** in the overriding-method of child-class<http://www.benchresources.net/wp-content/uploads/2017/02/Parent-declares-checked-and-unchecked-exception-child-declares-no-exception-4c.png>

**🔁 Summary Table**

| **Case** | **Superclass throws** | **Subclass throws** | **Valid?** |
| --- | --- | --- | --- |
| 1 | Checked Exception | Same Exception | ✅ |
| 2 | Checked Exception | Subclass | ✅ |
| 3 | Checked Exception | Broader Exception | ❌ |
| 4 | Checked Exception | None | ✅ |
| 5 | None | Checked Exception | ❌ |
| 6 | None | Unchecked | ✅ |
| 7 | Unchecked | Any Exception | ✅ |

**Conclusion:**

* When parent-class method declares **no exception**, then child-class overriding-method can declare,  
  **1. No exception or**  
  **2. Any number of unchecked exception**  
  **3. but strictly no checked exception**
* When parent-class method declares **unchecked exception**, then child-class overriding-method can declare,  
  **1. No exception or**  
  **2. Any number of unchecked exception**  
  **3. but strictly no checked exception**
* When parent-class method declares **checked exception**, then child-class overriding-method can declare,  
  **1. No exception or**  
  **2. Same checked exception or**  
  **3. Sub-type of checked exception or**  
  **4. any number of unchecked exception**
* All above conclusion hold true, even if combination of both checked & unchecked exception is declared in parent-class’ method

**References**

Java – Convert String to int

By [mkyong](http://www.mkyong.com/author/mkyong/) | June 1, 2015 | Updated : July 1, 2015 | Viewed : 328,462 times +7,780 pv/w

In Java, you can use Integer.parseInt() to convert a String to int.

1. Integer.parseInt() Examples

Example to convert a String “10” to an primitive int.

String number = "10";

int result = Integer.parseInt(number);

System.out.println(result);

Output

10

2. Integer.valueOf() Examples

Alternatively, you can use Integer.valueOf(), it will returns an Integer object.

String number = "10";

Integer result = Integer.valueOf(number);

System.out.println(result);

Output

10

Java – Convert comma-separated String to a List

1. Comma-separated String to List

TestApp1.java

package com.mkyong.utils;

import java.util.Arrays;

import java.util.List;

public class TestApp1 {

public static void main(String[] args) {

String alpha = "A, B, C, D";

//Remove whitespace and split by comma

List<String> result = Arrays.asList(alpha.split("\\s\*,\\s\*"));

System.out.println(result);

}

}

Output

[A, B, C, D]

2. List to Comma-separated String

No need to loop the List, uses the new Java 8 String.join

TestApp2.java

package com.mkyong.utils;

import java.util.Arrays;

import java.util.List;

public class TestApp2 {

public static void main(String[] args) {

List<String> list = Arrays.asList("A", "B", "C", "D");

String result = String.join(",", list);

System.out.println(result);

}

}

Output

A,B,C,D

1. Sort by Key

1.1 Uses java.util.TreeMap, it will sort the Map by keys automatically.

SortByKeyExample1.java

package com.mkyong.test;

import java.util.HashMap;

import java.util.Map;

import java.util.TreeMap;

public class SortByKeyExample1 {

public static void main(String[] args) {

Map<String, String> unsortMap = new HashMap<String, String>();

unsortMap.put("Z", "z");

unsortMap.put("B", "b");

unsortMap.put("A", "a");

unsortMap.put("C", "c");

unsortMap.put("D", "d");

unsortMap.put("E", "e");

unsortMap.put("Y", "y");

unsortMap.put("N", "n");

unsortMap.put("J", "j");

unsortMap.put("M", "m");

unsortMap.put("F", "f");

System.out.println("Unsort Map......");

printMap(unsortMap);

System.out.println("\nSorted Map......By Key");

Map<String, String> treeMap = new TreeMap<String, String>(unsortMap);

printMap(treeMap);

}

//pretty print a map

public static <K, V> void printMap(Map<K, V> map) {

for (Map.Entry<K, V> entry : map.entrySet()) {

System.out.println("Key : " + entry.getKey()

+ " Value : " + entry.getValue());

}

}

}

Output

Unsort Map......

Key : A Value : a

Key : B Value : b

Key : C Value : c

Key : D Value : d

Key : E Value : e

Key : F Value : f

Key : Y Value : y

Key : Z Value : z

Key : J Value : j

Key : M Value : m

Key : N Value : n

Sorted Map......By Key

Key : A Value : a

Key : B Value : b

Key : C Value : c

Key : D Value : d

Key : E Value : e

Key : F Value : f

Key : J Value : j

Key : M Value : m

Key : N Value : n

Key : Y Value : y

Key : Z Value : z

1.2 Yet another java.util.TreeMap example, provide a custom Comparator to sort the key in descending order.

SortByKeyExample2.java

package com.mkyong.test;

import java.util.Comparator;

import java.util.HashMap;

import java.util.Map;

import java.util.TreeMap;

public class SortByKeyExample2 {

public static void main(String[] args) {

Map<Integer, String> unsortMap = new HashMap<Integer, String>();

unsortMap.put(10, "z");

unsortMap.put(5, "b");

unsortMap.put(6, "a");

unsortMap.put(20, "c");

unsortMap.put(1, "d");

unsortMap.put(7, "e");

unsortMap.put(8, "y");

unsortMap.put(99, "n");

unsortMap.put(50, "j");

unsortMap.put(2, "m");

unsortMap.put(9, "f");

System.out.println("Unsort Map......");

printMap(unsortMap);

System.out.println("\nSorted Map......By Key");

Map<Integer, String> treeMap = new TreeMap<Integer, String>(

new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

return o2.compareTo(o1);

}

});

/\* For Java 8, try this lambda

Map<Integer, String> treeMap = new TreeMap<>(

(Comparator<Integer>) (o1, o2) -> o2.compareTo(o1)

);

\*/

treeMap.putAll(unsortMap);

printMap(treeMap);

}

public static <K, V> void printMap(Map<K, V> map) {

for (Map.Entry<K, V> entry : map.entrySet()) {

System.out.println("Key : " + entry.getKey()

+ " Value : " + entry.getValue());

}

}

}

Output

Unsort Map......

Key : 1 Value : d

Key : 50 Value : j

Key : 2 Value : m

Key : 99 Value : n

Key : 20 Value : c

Key : 5 Value : b

Key : 6 Value : a

Key : 7 Value : e

Key : 8 Value : y

Key : 9 Value : f

Key : 10 Value : z

Sorted Map......By Key

Key : 99 Value : n

Key : 50 Value : j

Key : 20 Value : c

Key : 10 Value : z

Key : 9 Value : f

Key : 8 Value : y

Key : 7 Value : e

Key : 6 Value : a

Key : 5 Value : b

Key : 2 Value : m

Key : 1 Value : d

***10 Differences*** *between java.util.*[***List***](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html)*, java.util.*[***Set***](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html) *and java,util.*[***Map***](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) *in java >*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Property | ***java.util.List*** | ***java.util.Set*** | ***java.util.Map*** |
| 1 | Duplicate elements | List **allows to store duplicate elements** in java. | *Set does* ***not allow to store duplicate elements*** in java*.* | *Map stores data in form of* ***key-value pair*** *it does not allow to store duplicate keys but allows duplicate values* in java*.* |
| 2 | Insertion order | java.util.List is ordered collection it **maintain insertion order** in java. | *Most of the java.util.Set implementation* does not **maintain insertion order**.  HashSet does not maintains insertion order in java.  Thought LinkedHashSet maintains insertion order in java.    TreeSet is sorted by natural order in java. | *Most of the java.util.Map implementation* does not **maintain insertion order**.  HashMap does not maintains insertion order in java.  Thought LinkedHashMap maintains insertion order of keys in java.    TreeMap is sorted by natural order of keys in java. |
|  |  |  |  |  |
| 3 | Null keys | List allows to store **many null keys** in java. | Most of the Set implementations allow to add only **one null** in java**.**  [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html) and [**ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html) does not allow to add null in java. | Lets look at Map implementations -  [HashMap](http://www.javamadesoeasy.com/2015/04/hashmap-in-java.html) allows one null key and many null values.  [LinkedHashMap](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html) allows one null key and many null values.  [TreeMap](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html) doesn't allow null key but allow many null values.  [Hashtable](http://www.javamadesoeasy.com/2015/04/hashmap-and-hashtable-similarity-and.html) doesn't allow null key or null values.  [ConcurrentHashMap](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html) doesn't allow null key or null values.  [ConcurrentSkipListMap](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html) doesn't allow null key or null values. |
| 4 | Getting element on specific **index** | List implementations provide get method to get element on specific index in java.  ArrayList, Vector, copyOnWriteArrayList and LinkedList provides -  *get(int index)*  Method returns element on specified *index*.  **Get method directly gets element on specified index. Hence, offering O(1) complexity.** | Set implementations does not provide any such get method to get element on specified index in java. | Map implementations does not provide any such get method to get element on specified index in java. |
| 5 | Implementing classes | [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)***,*** [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html)***,*** [**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)***,*** [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html) classes implements [**List**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface in java. | [**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)***,*** [**CopyOnWriteArraySet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html)***,*** [**LinkedHashSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html)***,*** [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [**ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html), [**EnumSet**](http://www.javamadesoeasy.com/2015/04/enumset-in-java-with-program.html) classes implements [**Set**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface in java. | [HashMap, Hashtable, ConcurrentHashMap,  LinkedHashMap,  TreeMap,  ConcurrentSkipListMap,  IdentityHashMap,WeakHashMap,  EnumMap classes](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) implements [Map](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) interface in java. |
| 6 | listIterator | **listIterator** method returns listIterator to iterate over elements in List in java.  **listIterator provides** additional methods as compared to iterator like  **hasPrevious(), previous(), nextIndex(), previousIndex(), add(E element), set(E element)** | Set does not provide anything like listIterator. It simply return Iterator in java. | Map provides three type of iterators -  *map.keySet().iterator()* method returns iterator to iterate over keys in HashMap  *map.values().iterator()* method returns iterator to iterate over keys in HashMap in java.  *map.entrySet().iterator()* method returns iterator to iterate over keys in HashMap. |
| 7 | Structure and resizable | **List** are Resizable-array implementation of the java.util.**List** interface in java. | Set uses [**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html)for their implementation.  Hence, structure is map based and resizing depends on Map implementation.  *Example >* [***HashSet***](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html) *internally uses* [*HashMap*](http://javamadesoeasy.com/2015/02/hashmap-custom-implementation.html)*.* | [**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) **uses hashing technique for storing** key-value pairs. |
| 8 | Index based structure /RandomAccess | As **ArrayList** uses array for implementation it is index based structure, hence provides random access to elements.  But [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html) is not indexed based structure in java. | Set is not index based structure at all in java. | Map is not index based structure at all in java. |
| 9 | unsynchronized implementations | [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)***,*** [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html) | [**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)***,*** [**LinkedHashSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html)***,*** [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [**EnumSet**](http://www.javamadesoeasy.com/2015/04/enumset-in-java-with-program.html) | [HashMap,  LinkedHashMap,  TreeMap,  IdentityHashMap, WeakHashMap,  EnumMap](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) |
| 10 | synchronized implementations | [**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)***,*** [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html) | [**CopyOnWriteArraySet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html), [**ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html) | [Hashtable](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html), [ConcurrentHashMap](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html), [ConcurrentSkipListMap](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html), |

[**java.util.List hierarchy in java**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html)

[**java.util.Set hierarchy in java**](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html)

[**java.util.Map hierarchy in java**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html)

*Differences between java.util.*[*ArrayList*](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html) *and java.util.*[*LinkedList*](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html) *in java >*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.ArrayList*** | **java.util.LinkedList** |
| 1 | Structure | java.util.ArrayList is index based structure in java.  [https://lh6.googleusercontent.com/f8eJoX-FNynX9xx_rTl8IqaAGjMMtDYnZVgg6Hkco7C7uYfoHpCk-5-EjhsyIPm5OxjMI7d0JkALWUYnXP7tgqWTUrYCoFQBBbjQrg6fRHdCNyeglftSiWg-tB0R1-SOWfE1tw0](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html) | A java.util.**LinkedList** is a data structure consisting of a group of **nodes** which together represent a sequence.  node is composed of a data and a reference (in other words, a **link**) to the next node in the sequence in java.  [https://lh3.googleusercontent.com/iS8cWH4MmnPfJHvBJRj9EonXJ2d2NQXuZf6_5vA8n9DzFWouOJI0O7xB0Ed115OpPhGvSXIx5X7BU28PXg4YFrhkp6RHX0RtwwlcboamgUM3_YaBSQlN2VmjYzt-KfY8lEvTnNk](http://www.javamadesoeasy.com/2015/01/doublylinkedlist-insert-and-delete-at.html) |
| 2 | **Resizable** | **ArrayList is Resizable-array in java.** | New node is created for storing new element in LinkedList in java. |
| 3 | **Initial capacity** | java.util.ArrayList is created with initial capacity of 10 in java. | For storing every element node is created in LinkedList, so linkedList’s initial capacity is 0 in java. |
| 4 | Ensuring **Capacity**/ resizing. | ArrayList is created with initial capacity of 10.  ArrayList’s size is **increased by 50%** i.e. after resizing it’s size become 15 in java. | For storing every element node is created, so linkedList’s initial capacity is 0, it’s size grow with addition of each and every element in java. |
| 5 | RandomAccess interface | ArrayList implements RandomAccess(Marker interface) to indicate that they support fast random access (i.e. index based access) in java. | LinkedList does not implement RandomAccess interface in java. |
| 6 | AbstractList and AbstractSequentialList | ArrayList extends AbstractList (abstract class) which provides implementation to  List interface to minimize the effort required to implement this interface backed by RandomAccess interface. | LinkedList extends AbstractSequentialList (abstract class), AbstractSequentialList extends AbstractList.  In LinkedList, data is accessed sequentially, so for obtaining data at specific index, iteration is done on nodes sequentially in java. |
| 7 | How **get(index)** method works?  (Though difference has been discussed briefly in above 2 points but in this in point we will figure difference in detail.) | Get method of ArrayList directly gets element on specified index. Hence, offering O(1) complexity in java. | Get method of LinkedList iterates on nodes sequentially to get element on specified index. Hence, offering O(n) complexity in java. |
| **8** | **When to use** | **Use ArrayList when get operations is more frequent than add and remove operations in java.** | **Use LinkedList when add and remove operations are more frequent than get operations in java.** |
| **9** | **Complexity** offered by methods are different | |  |  |  | | --- | --- | --- | | Operation/ method | Worst case | Best case | | *add* | O(n), when array is full it needs restructuring,  operation runs in *amortized constant time in java.* | O(1), when array does not need any restructuring in java. | | *remove* | O(n), when removal is done from between restructuring is needed. | O(1), when removal is done at last position, no restructuring is needed. | | *get* | O(1), it is index based structure. So, complexity of  get operation is always done in O(1) in java. | O(1) it is index based structure. So, complexity of  get operation is always done in O(1) in java. | | *set* | O(1), it is index based structure, no restructuring is needed in set operation. So, complexity of operation is always O(1) | O(1), it is index based structure, no restructuring is needed in set operation. So, complexity of operation is always O(1) | | *iterator* | O(n), because iteration is done over each and every element in java. | O(n), because iteration is done over each and every element in java. | | *listIterator* | O(n), its same as iterator in java. | O(n), its same as iterator in java. | | *enumeration* | O(n), its same as iterator in java. | O(n), its same as iterator in java. | | |  |  |  | | --- | --- | --- | | Operation/ method | Worst case | Best case | | *add(E element)* | **O(1),** Adds specified *element* to the end of LinkedList in java. | **O(1),** Adds specified *element* to the end of LinkedList in java. | | *add(int index, E element)* | **O(n)**, because iteration is done on all elements one by one to find out specified index.  Current *element* is placed at specified *index* and one is added to indices of subsequent elements on right in java. | **O(n)** | | *addFirst(E element)* | **O(1)** | **O(1)** | | *addLast(E element)* | **O(1)** | **O(1)** | |  |  |  | | *remove()* | **O(1),** Method retrieves and removes the first element (head) of this list in java. | **O(1)** | | *remove(int index)* | **O(n)**, because iteration is done on all elements one by one to find out specified index.  one is subtracted from indices of subsequent elements on right. | **O(n)** | | *remove(Object object)* | **O(n)**, because iteration is done on all elements one by one to find out specified object.  one is subtracted from indices of subsequent elements on right. | **O(n)** | | *removeFirst()* | O(1) | O(1) | | *removeLast()* | O(1) | O(1) | | iterator | O(n), because iteration is done over each and every element. | O(n), because iteration is done over each and every element. | | listIterator | O(n), its same as iterator in java. | O(n), its same as iterator in java. | | enumeration | O(n), its same as iterator in java. | O(n), its same as iterator in java. | |

*So far we have learned what are differences between ArrayList and LinkedList in java. Now we will learn similarities in ArrayList and LinkedList in Collection framework in java.*

*Read :* [**Collection - List, Set and Map all properties in tabular form in java**](http://www.javamadesoeasy.com/2015/04/collection-list-set-and-map-all.html)

***Similarity*** *in java.util.****ArrayList*** *and java.util.****LinkedList*** *in java >*

|  |  |  |
| --- | --- | --- |
|  | Property | *java.util.****ArrayList*** *and java.util.***LinkedList** |
| 1 | synchronization | ***ArrayList*** *and* **LinkedList both** are **not synchronized**  (because 2 threads on same ArrayList/LinkedList object can access it at same time) in java.  I have created **program** to show see consequence of using ArrayList in multithreading environment.  In the program i will implement our own arrayList in java. |
| 2 | Iterator and listIterator are Fail-fast | Iterator and listIterator returned by ***ArrayList*** *and* **LinkedList both** are [**Fail-fast**](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html) in java. |
| 3 | Enumeration is fail-fast | **Enumeration** of ***ArrayList*** *and* **LinkedList both** is **fail-fast**, means any modification made to ArrayList during iteration using Enumeration will throw ConcurrentModificationException in java.  Example/Code to throw ConcurrentModificationException in ArrayList (we may replace arrayList with linkedList )-   |  | | --- | | Enumeration<String> listEnum= Collections.*enumeration*(arrayList);  **while**(listEnum.hasMoreElements()){  //adding element will throw   ConcurrentModificationException        System.*out*.println(listEnum.nextElement());  } | |
| 4 | Insertion order | ***ArrayList*** *and* **LinkedList both maintains insertion order** in java. |
| 5 | Allows null | ***ArrayList*** *and* **LinkedList** both **allows to store null** in java. |
| 6 | Implements java.util.List | ***ArrayList*** *and* **LinkedList** both are implementation of the java.util.**List** interface. |
| 7 | Introduced in which java version | ***ArrayList*** *and* **LinkedList** both were introduced in second version of java (1.2) i.e. **JDK 2.0** |

### [Differences and Similarities between ArrayList and vector in java](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)

*You are here :* [***Home***](http://www.javamadesoeasy.com/) ***/*** [***Core Java Tutorials***](http://www.javamadesoeasy.com/2015/04/core-java-tutorial.html) ***/*** [*Collection framework Tutorial in java*](http://www.javamadesoeasy.com/2015/05/collection-framework-tutorial-in-java.html)

*It’s very important to differentiate between ArrayList and Vector, so in this Collection framework tutorial we will learn what are differences and similarities between java.util.ArrayList and java.util.Vector in java.*

*Also Read :*[**java.util.List hierarchy in java**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html)

[**java.util.Set hierarchy in java**](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html)

[**java.util.Map hierarchy in java**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html)

***Differences*** *between java.util.*[***ArrayList***](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html) *and java.util.****Vector*** *in java>*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.ArrayList*** | ***java.util.Vector*** |
| 1 | synchronization | java.util.ArrayList is **not synchronized**  (because 2 threads on same ArrayList object can access it at same time).  I have created [**program**](http://www.javamadesoeasy.com/2015/05/consequence-of-using-arraylist-in.html)to show consequence of using ArrayList in multithreading environment.  In the program we will implement our own arrayList in java. | java.util.Vector is **synchronized** (because 2 threads on same Vector object cannot  access it at same time).  I have created [**program**](http://www.javamadesoeasy.com/2015/05/advantage-of-using-vector-in.html)to show advantage of using Vector in multithreading environment.  In the program we will implement our own vector in java. |
| 2 | Performance | ArrayList is not synchronized, hence its operations are **faster** as compared to Vector in java. | Vector is synchronized, hence its operations are **slower** as compared to ArrayList in java.  If we are working not working in multithreading environment jdk recommends us to use ArrayList. |
| 3 | Enumeration | **Enumeration** is [**fail-fast**](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html), means any modification made to ArrayList during iteration using Enumeration will throw ConcurrentModificationException in java. | **Enumeration** is **fail-safe**, means any modification made to Vector during iteration using Enumeration don’t throw any exception in java. |
| 4 | Introduced  in which java version | ArrayList was introduced in second version of java i.e. **JDK 2.0** | Vector was introduced in first version of java i.e. **JDK 1.0**  But it was refactored in java 2 i.e. JDK 1.2 to implement the List interface, hence making it a member of member of the [Java Collections Framework](http://www.javamadesoeasy.com/2015/04/collection-in-java.html). |
| 5 | Ensuring Capacity/ resizing. | ArrayList is created with initial capacity of 10.  When its full size is **increased by 50%** i.e. after resizing it’s size become 15 in java. | Vector is created with initial capacity of 10.  Vector’s size is **increased by 100%** i.e. after resizing it’s size become 20 in java. |
| 6 | Custom implementation | [https://lh4.googleusercontent.com/FxlXU1zjlhbC03GLS5VMI9ubd2k_1ToIAsVmXXFYwEzF93-gTYCcJUkmkikYTeLiApsdx62z2NZXn4CAeTBdaAGSWDZZe_TPO9Pyz3GtW-PdbqkTQv8ZwoOR3rFSEb_CHbBZVng](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html)Read : [ArrayList custom implementation](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html) | [https://lh3.googleusercontent.com/-RcOB6IdgA9EdNDV1BU77aAYgZ29vPcz59SYkwvjtapRDlEvmOVny8YU-sjAtqFr1HBoZNk8qJShPDxlzmKb-MrhLKQQuyfZOMqp_OxeWni3x8qzY1M4V9XxOnMpd195WoCLqCY](http://javamadesoeasy.com/2015/02/vector-custom-implementation.html)Read :  [Vector custom implementation](http://javamadesoeasy.com/2015/02/vector-custom-implementation.html) |

*So far we have learned what are differences between ArrayList and Vector in java. Now we will learn similarities in ArrayList and Vector in Collection framework in java.*

*Read :* [**Collection - List, Set and Map all properties in tabular form in java**](http://www.javamadesoeasy.com/2015/04/collection-list-set-and-map-all.html)

***Similarity*** *in java.util.****ArrayList*** *and java.util.****Vector*** *in java >*

|  |  |  |
| --- | --- | --- |
|  | Property | *java.util.****ArrayList and*** *java.util.****Vector*** |
| 1 | Iterator and listIterator are Fail-fast | Iterator and listIterator returned by *ArrayList and Vector* both are **Fail-fast** in java. |
| 2 | Insertion order | *ArrayList and Vector* both **maintains insertion order** in java. |
| 3 | Duplicate | *ArrayList and Vector* both **allow to store duplicate elements** in java**.** |
| 4 | Allows null | *ArrayList and Vector* both **allows to store null** in java. |
| 5 | Implements java.util.List | *ArrayList and Vector* both are implementation of the java.util.**List** interface. |
| 6 | Index based structure | *ArrayList and Vector* both are **index based structures** in java. |
| 7 | Complexity | Complexity offered by **methods of** *ArrayList and Vector* **both of these is same** in java.   |  |  |  | | --- | --- | --- | | Operation/ method | ***Worst case*** | ***Best case*** | | add | O(n), when array is full it needs restructuring,  operation runs in *amortized constant time.* | O(1), when array does not need any restructuring. | | remove | O(n), when removal is done from between restructuring is needed. | O(1), when removal is done at last position, no restructuring is needed. | | get | O(1), it is index based structure. So, complexity of  get operation is always done in O(1). | O(1) it is index based structure. So, complexity of  get operation is always done in O(1). | | set | O(1), it is index based structure, no restructuring is needed in set operation. So, complexity of operation is always O(1) | O(1), it is index based structure, no restructuring is needed in set operation. So, complexity of operation is always O(1) | | iterator | O(n), because ireation is done over each and every element. | O(n), because ireation is done over each and every element. | | listIterator | O(n), its same as iterator. | O(n), its same as iterator. | | enumeration | O(n), its same as iterator. | O(n), its same as iterator. | |

### [Differences and Similarities between Iterator and ListIterator in java](http://www.javamadesoeasy.com/2015/04/iterator-vs-listiterator-similarity-and.html)

*In this Collection framework tutorial we will learn what are differences and similarities between java.util.****Iterator*** *and java.util.****ListIterator*** *in java.*

***Differences*** *between java.util.****Iterator*** *and java.util.****ListIterator*** *>*

|  |  |  |
| --- | --- | --- |
|  | ***java.util.ListIterator*** | ***java.util.Iterator*** |
| 1 | **hasPrevious()**  method returns true if this listIterator has more elements when traversing the list in the reverse direction. | **No such method** in java.util.Iterator. |
| 2 | **previous()**  returns previous element in iteration (traversing in backward direction).  if the iteration has no previous elements than NoSuchElementException is thrown. | **No such method** in java.util.Iterator. |
| 3 | **nextIndex()**  method returns the index of the element that would be returned by a subsequent call to next() method. If listIterator is at the end of the list than method returns size of list. | **No such method** in java.util.Iterator. |
| 4 | **previousIndex()**  method returns the index of the element that would be returned by a subsequent call to previous() method. If listIterator is at the start of the list than method returns -1. | **No such method** in java.util.Iterator. |
| 5 | **add(E element)**  Method inserts the specified **element** into the list.  The element is inserted immediately before the element that would be returned by next (So, subsequent call to next would be unaffected), if any, and after the element that would be returned by previous (So,subsequent call to previous would return the new **element**), if any.  If the list does not contain any element than new **element** will be the sole element in the list. | **No such method** in java.util.Iterator. |
| 6 | **set(E element)**  Method replaces the last element returned by next() or previous() method with the specified **element**. This call can be made only if neither remove nor add have been called after the last call to next or previous.  If call to set() method is followed up by any call made to remove() or add() method after next() or previous() than UnsupportedOperationException is thrown. | **No such method** in java.util.Iterator. |
| 7 | All the implementations of [**List**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html) interface like [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)***,*** [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html), [**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)***,*** [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html) classes returns listIterator. | All Implementation classes of [**Collection**](http://www.javamadesoeasy.com/2015/04/collection-in-java.html) interface’s subinterfaces like [Set and List](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) return iterator. |

*Also Read :* [**Collection - List, Set and Map all properties in tabular form in java**](http://www.javamadesoeasy.com/2015/04/collection-list-set-and-map-all.html)

*So far we have learned what are differences between java.util.****Iterator*** *and java.util.****ListIterator*** *in java.*

*Now we will learn similarities between java.util.****Iterator*** *and java.util.****ListIterator*** *in Collection framework in java.*

***Similarity*** *between java.util.****Iterator*** *and java.util.****ListIterator*** *>*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | **java.util.Iterator** | **java.util.ListIterator** |
| 1 | Definition | *Iterator* method returns iterator to iterate over elements in ArrayList. | *ListIterator* method returns listIterator to iterate over elements in ArrayList. |
| 2 | Common important methods | Iterator important methods :  **hasNext()** method returns true if the iteration has more elements. (Traversing/Iteration is  done in forward direction).  **next()** method - returns next element in iteration.  if the iteration has no more elements than NoSuchElementException is thrown.  **remove()** method - removes last element returned by iterator.  Method must always be called after call to next() method else IllegalStateException is thrown. | ListIterator provide all 3 methods.  Apart from hasNext(), next() and remove() methods provided by Iterator, **ListIterator additionally provides(We will discuss that in differences).** |
| 3 | **Complexity** | Complexity of Iterator is **O(n)**, because iteration/traversal is done over each and every element. | Complexity of ListIterator is also **O(n)**, because iteration/traversal is done over each and every element. |

### [Differences between Collection and Collections in java](http://www.javamadesoeasy.com/2015/04/collection-vs-collections-differences.html)

*In this Collection framework tutorial we will learn what are differences between java.util.****Collection*** *and java.util.****Collections*** *in java.*

java.util.[***Collection***](http://www.javamadesoeasy.com/2015/04/collection-in-java.html) ***​*** *is the* root **interface** in the ​*hierarchy of Java Collection framework​*.

The JDK does not provide any classes which directly implements java.util.Collection interface, but it  provides classes such as [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html), [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html), [**vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html), [**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html), [**EnumSet**](http://www.javamadesoeasy.com/2015/04/enumset-in-java-with-program.html), [**LinkedHashSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [CopyOnWriteArrayList](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html), [CopyOnWriteArraySet](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html), [ConcurrentSkipListSet](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html)  which implements more specific subinterfaces like ​[Set and List​](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) in java.

java.util.**Collections** is a utility **class** which **consists** of **static methods** that **operate on** or return **Collection** in java.

**java.util.Collections provides method like >**

* **reverse** method for reversing [**List**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html) in java.
* **shuffle** method for shuffling elements of **List** in java.
* **unmodifiableCollection**, [**unmodifiableSet**](http://www.javamadesoeasy.com/2015/04/hashset-making-set-unmodifiable-using.html), [**unmodifiableList**](http://www.javamadesoeasy.com/2015/04/arraylist-making-list-unmodifiable.html), [**unmodifiableMap**](http://www.javamadesoeasy.com/2015/04/hashmap-making-map-unmodifiable-using.html) methods for making **List**, [**Set**](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html) and [**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) unmodifiable in java.
* **min** method to return smallest element in **Collection** in java.
* **max** method to return smallest element in **Collection**.
* **sort** method for sorting **List**.
* **synchronizedCollection**, [**synchronizedSet**](http://www.javamadesoeasy.com/2015/04/hashset-synchronizing-using.html), [**synchronizedList**](http://www.javamadesoeasy.com/2015/04/arraylist-synchronizing-using.html), [**synchronizedMap**](http://www.javamadesoeasy.com/2015/04/hashmap-synchronizing-map-using.html)methods for synchronizing **List**, **Set** and **Map** respectively in java**.**

### These very important questions have been framed to test your basic and in depth knowledge of Collection, these questions covers vast variety of questions that touches almost all concepts of Collection in java. Questions ranges from easy to hard for fresher/beginner to experienced developers . Go, have a crack on these!!

### Collection interview Question 76.

### Collection Output interview question 26.

|  |
| --- |
| import **java.util.HashMap;**  **import** java.util.Map;  /\*\* \*/  **public** **class** MyClass {  **public** **static** **void** main(String args[]) {            Map<String, String> hashMap = **new** HashMap<String, String>();            hashMap.put(**new** String("a"), "audi");            hashMap.put(**new** String("a"), "ferrari");            System.*out*.println(hashMap);     }  }  /\*OUTPUT  {a=ferrari}  \*/ |

### Answer. HashMap does not allow duplicate keys. HashMap when comparing keys (and values) performs object-equality not reference-equality. In an HashMap, two keys k1 and k2 are equal if and only if (k1==null ? k2==null : k1.equals(k2))

### new String("a") & new String("a") are different by reference but equal by value.

### 

### Collection interview Question 77.

### Collection Output interview question 27.

|  |
| --- |
| import **java.util.IdentityHashMap;**  **import** java.util.Map;  /\*\* \*/  **public** **class** MyClass {  **public** **static** **void** main(String args[]) {            Map<String, String> identityHashMap = **new** IdentityHashMap<String, String>();            identityHashMap.put(**new** String("a"), "audi");            identityHashMap.put(**new** String("a"), "ferrari");            System.*out*.println(identityHashMap);     }  }  /\*OUTPUT  {a=audi, a=ferrari}  \*/ |

### Answer.

### [IdentityHashMap](http://www.javamadesoeasy.com/2015/04/identityhashmap-in-java.html) when comparing keys (and values) performs reference-equality in place of object-equality. In an IdentityHashMap, two keys k1 and k2 are equal if and only if (k1==k2). (In normal Map implementations (like HashMap) two keys k1 and k2 are considered equal if and only if (k1==null ? k2==null : k1.equals(k2)).)

### new String("a") & new String("a") are different by reference.

### Must read : [Differences and Similarities between HashMap and IdentityHashMap with program in java](http://www.javamadesoeasy.com/2015/04/hashmap-vs-identityhashmap-similarity.html)

# Difference Between HashMap and IdentityHashMap

### Most of the time I use HashMap whenever a map kinda object is needed. When reading some blog I came across IdentityHashMap in Java.

Most of the time I use HashMap whenever a map kinda object is needed. When reading some blog I came across [IdentityHashMap](http://java.sun.com/j2se/1.4.2/docs/api/java/util/IdentityHashMap.html)in Java. It is good to understand the differences between the two because you never know when you will see them flying across your code and you trying to find out why is  this kinda Map is used here.

IdentityHashMap as name suggests uses the equality operator(==) for comparing the keys. So when you put any Key Value pair in it the Key Object is compared using == operator.

import java.util.HashMap; import java.util.IdentityHashMap; import java.util.Map;public class IdentityMapDemo {public static void main(String[] args) { Map identityMap = new IdentityHashMap(); Map hashMap = new HashMap(); identityMap.put("a", 1); identityMap.put(new String("a"), 2); identityMap.put("a", 3);hashMap.put("a", 1); hashMap.put(new String("a"), 2); hashMap.put("a", 3);System.out.println("Identity Map KeySet Size :: " + identityMap.keySet().size()); System.out.println("Hash Map KeySet Size :: " + hashMap.keySet().size()); } }

On the other hand HashMap uses equals method to determine the uniqueness of the Key.

k1.equals(k2)

instead of equality operator.

When you run the above code the result will be

Identity Map KeySet Size :: 2Hash Map KeySet Size :: 1

The Keysize of Identity Map is 2 because here **a** and **new String(“a”)** are considered two different Object. The comparison is done using == operator.

For HashMap the keySize is 1 because K1.equals(K2) returns true for all three Keys and hence it keep on removing the old value and updating it with the new one.

These both Maps will behave in same manner if they are used for Keys which are user defined Object and doesn’t overrides equals method.

### 

### Collection interview Question 78.

### Collection Output interview question 28.

|  |
| --- |
| import **java.util.Map;**  **import** java.util.TreeMap;  /\*\* \*/  **public** **class** TreeMapTest {  **public** **static** **void** main(String args[]) {            Map<Integer, String> m = **new** TreeMap<Integer, String>();            m.put(11, "audi");            m.put(**null**, **null**);            m.put(11, "bmw");            m.put(**null**, "fer");            System.*out*.println(m.size());            System.*out*.println(m);     }  } |

### Answer.  NullPointerException

### [TreeMap](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html) does not any null key or null value.

### 

### Collection interview Question 79.

### Collection Output interview question 29.

|  |
| --- |
| import **java.util.Arrays;**  **import** java.util.Comparator;  /\*\* \*/  **public** **class** MyClass {  **public** **static** **void** main(String[] args) {            String[] ar = { "c", "d", "b", "a", "e" };            NestedClass in = **new** NestedClass();            Arrays.*sort*(ar, in);  **for** (String str : ar)                   System.*out*.print(str + " ");            System.*out*.println(Arrays.*binarySearch*(ar, "b"));     }  **static** **class** NestedClass **implements** Comparator<String> {  **public** **int** compare(String s1, String s2) {  **return** s2.compareTo(s1);            }     }  } |

### Answer.

### /\*

### e d c b a -1

### \*/

### >compareTo() method will do the reverse sorting.

### >binarySearch() gives –1 because it should have been invoked using the same Comparator as was used during reverse sorting of the array.

### 

### Read:

### [COLLECTION - Top 100 interview questions and answers in java for fresher and experienced in detail - Set-1 > Q1- Q50](http://www.javamadesoeasy.com/2015/05/collection-top-50-interview-questions.html)

### [COLLECTION - Top 100 important interview OUTPUT questions and answers in java, Set-2 > Q51- Q75](http://www.javamadesoeasy.com/2015/07/collection-top-100-important-interview.html)

### 

### Collection interview Question 80.

### Collection Output interview question 30.

|  |
| --- |
| import **java.util.EnumSet;**  **import** java.util.Set;  /\*\* \*/  **public** **class** EnumSetTest {  **private** **enum** Days {  *Monday*, *Tuesday*, *Wednesday*, *Thursday*, *Friday*, *Saturday*, *Sunday*;  **public** **static** Set<Days> *allDays* = EnumSet.*allOf*(Days.**class**);    **public** **static** Set<Days> *weekDays* = EnumSet.*range*(*Monday*, *Friday*);    **public** **boolean** isWeekDay() {  **return** *weekDays*.contains(**this**);     }     }     /\*\* Main \*/  **public** **static** **final** **void** main(**final** String args[]) {            System.*out*.println(Days.*weekDays*.size());              Days day=Days.*Monday*;            System.*out*.println( (day.isWeekDay() ? "is WeekDay" : "is weekEnd"));              day=Days.*Sunday*;            System.*out*.println( (day.isWeekDay() ? "is WeekDay" : "is weekEnd"));         day=Days.*Monday*;     System.*out*.println(Days.*allDays*.contains(day));     System.*out*.println(day.ordinal());     }  } |

### Answer.

### /\*OUTPUT

### 5

### is WeekDay

### is weekEnd

### true

### 0

### \*/

### *Read :* [*EnumSet in java*](http://www.javamadesoeasy.com/2015/04/enumset-in-java-with-program.html)

### 

### Collection interview Question 81.

### Collection Output interview question 31.

|  |
| --- |
| import **java.util.LinkedHashSet;**  **import** java.util.Set;  /\*\* \*/  **public** **class** LinkedHashSetTest {  **public** **static** **void** main(String args[]) {            Set s = **new** LinkedHashSet();            s.add("1");            s.add(1);            s.add(3);            s.add(2);            System.*out*.println(s);     }  } |

### Answer.

### /\* OUTPUT

### [1, 1, 3, 2]

### \*/

### [LinkedHashSet](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html) maintains insertion order and does not allow duplicates.

### *Read :* [*HashSet vs LinkedHashSet vs TreeSet - Similarity and Differences*](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html)

### 

### Collection interview Question 82.

### Collection Output interview question 32.

|  |
| --- |
| import **java.util.ArrayList;**  **import** java.util.Collections;  **class** Employee **implements** Comparable<Employee>{     String name;     String id;  **public** Employee(String name, String id) {  **this**.name = name;  **this**.id = id;     }     @Override  **public** **int** compareTo(Employee otherEmployee) {  **return** **this**.name.compareTo(otherEmployee.name);     }     @Override  **public** String toString() {  **return** "{" + "name=" + name + ", id=" + id  + '}';     }  }  /\*\* \*/  **public** **class** ComparableUsage {  **public** **static** **void** main(String[] args) {         Employee emp1=**new** Employee("sam","4");         Employee emp2=**new** Employee("amy","2");         ArrayList<Employee> list=**new** ArrayList<Employee>();         list.add(emp1);         list.add(emp2);         Collections.*sort*(list);         System.*out*.println(list); |

### Answer.

### /\*OUTPUT

### [{name=amy, id=2}, {name=sam, id=4}]

### \*/

### compareTo method of Comparable has been implemented properly and will sort Employee class on basis of name in ascending order.

### Read : [Comparable vs Comparator - differences](http://www.javamadesoeasy.com/2015/04/comparable-vs-comparator-differences.html) for more.

### 

### Collection interview Question 83.

### Collection Output interview question 33.

|  |
| --- |
| import **java.util.ArrayList;**  **import** java.util.Collections;  **import** java.util.Comparator;  **class** Employee **implements** Comparator<Employee>{     String name;     String id;    **public** Employee() {}    **public** Employee(String name, String id) {  **this**.name = name;  **this**.id = id;     }       @Override  **public** **int** compare(Employee obj1, Employee obj2) {  **return** obj2.name.compareTo(obj1.name);     }     @Override  **public** String toString() {  **return** "{" + "name=" + name + ", id=" + id  + '}';     }  }  /\*\* \*/  **public** **class** ComparatorUsage {  **public** **static** **void** main(String[] args) {         Employee emp1=**new** Employee("sam","4");         Employee emp2=**new** Employee("amy","2");        ArrayList<Employee> list=**new** ArrayList<Employee>();         list.add(emp1);         list.add(emp2);         Collections.*sort*(list,**new** Employee());         System.*out*.println(list);     }  } |

### Answer.

### /\*OUTPUT

### [{name=sam, id=4}, {name=amy, id=2}]

### \*/

### compare method of Comparator has been implemented properly and will sort Employee class on basis of name in descending order.

### 

### Collection interview Question 84.

### Collection Output interview question 34.

|  |
| --- |
| import **java.util.ArrayList;**  **import** java.util.Collections;  **import** java.util.Comparator;  **class** Employee{     String name;  **public** Employee() {}  **public** Employee(String name) {  **this**.name = name;     }  **public** String toString() {  **return** "name=" + name;     }  **static** **class** ComparatorName **implements** Comparator<Employee>{  **public** **int** compare(Employee obj1, Employee obj2) {  **return** obj1.name.compareTo(obj2.name);         }     }  }  /\*\* \*/  **public** **class** ComparatorUsage {  **public** **static** **void** main(String[] args) {         Employee emp1=**new** Employee("ankit");         Employee emp2=**new** Employee("brad");          ArrayList<Employee> list=**new** ArrayList<Employee>();         list.add(emp1);         list.add(emp2);         Collections.*sort*(list,**new** Employee.ComparatorName());         System.*out*.println(list);     }  } |

### Answer.

### /\*OUTPUT

### [name=ankit, name=brad]

### \*/

### compare method of Comparator has been implemented properly by static class ComparatorName and will sort Employee class on basis of name in ascending order.

### 

### Collection interview Question 85.

### Collection Output interview question 35.

|  |
| --- |
| import **java.util.Arrays;**  **import** java.util.Comparator;  **class** Sort **implements** Comparator<Integer> {  **public** **int** compare(Integer o1, Integer o2) {  **return** o2.compareTo(o1);     }  }  /\*\* \*/  **public** **class** MyClass {  **public** **static** **void** main(String...a){         Integer intArray[]={2,3,1};         Arrays.*sort*(intArray, **new** Sort());  **for**(**int** i: intArray){            System.*out*.print(i+" ");         }     }  } |

### Answer.

### /\*OUTPUT

### 3 2 1

### \*/

### In program, we sort Integer array by using Arrays.sort (we will define Comparator to sort elements in descending order)

### *Read :* [*Arrays.sort to sort arrays by implementing Comparator and how Comparator of superclass can be used by subclasses*](http://www.javamadesoeasy.com/2015/04/arrayssort-to-sort-arrays-by.html)

### 

### Read:

### [COLLECTION - Top 100 interview questions and answers in java for fresher and experienced in detail - Set-1 > Q1- Q50](http://www.javamadesoeasy.com/2015/05/collection-top-50-interview-questions.html)

### [COLLECTION - Top 100 important interview OUTPUT questions and answers in java, Set-2 > Q51- Q75](http://www.javamadesoeasy.com/2015/07/collection-top-100-important-interview.html)

### 

### Collection interview Question 86.

### Collection Output interview question 36.

|  |
| --- |
| import **java.util.Comparator;**  **import** java.util.Set;  **import** java.util.TreeSet;  /\*\* \*/  **public** **class** SortSet {  **public** **static** **void** main(String...a){         Set<Integer> treeSet = **new** TreeSet<Integer>(**new** Comparator<Integer>() {  **public** **int** compareTo(Integer o1, Integer o2) {  **return** o2.compareTo(o1);                   }            });         treeSet.add(3);         treeSet.add(1);         treeSet.add(2);         System.*out*.println(treeSet);     }  } |

### Answer.

### /\*OUTPUT

### compile time exception

### \*/

### We haven’t implemented compare method of Comparator. If compare would have been there in place of compareTo program would have compiled and executed properly, hence would have sorted elements of treeSet in reverse order..

### Read : [Sort Set by using TreeSet and by implementing Comparator and Comparable interface](http://www.javamadesoeasy.com/2015/04/sort-set-by-using-treeset-and-by.html)

### 

### Collection interview Question 87.

### Collection Output interview question 37.

|  |
| --- |
| import **java.util.Collection;**  **import** java.util.HashSet;  **import** java.util.Set;  **import** java.util.TreeSet;  /\*\* \*/  **public** **class** SortSet {  **public** **static** **void** main(String...a){         Collection<Integer> collection = **new** HashSet<Integer>();         collection.add(3);         collection.add(1);         collection.add(2);         Set<Integer> treeSet = **new** TreeSet<Integer>(collection);         System.*out*.println(treeSet);     }  } |

### Answer.

### /\*OUTPUT

### [1, 2, 3]

### \*/

### *If elements are stored in stored in* [*HashSet*](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)*/*[*ArrayList*](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html) *or any other class that implements* [*Collection*](http://www.javamadesoeasy.com/2015/04/collection-in-java.html)*, then we can use TreeSet’s addAll method or constructor for sorting.*

### *Read :* [*Sort Set by using TreeSet and by implementing Comparator and Comparable interface in java*](http://www.javamadesoeasy.com/2015/04/sort-set-by-using-treeset-and-by.html)

### 

### Collection interview Question 88.

### Collection Output interview question 38.

|  |
| --- |
| import **java.util.Collection;**  **import** java.util.HashSet;  **import** java.util.Set;  **import** java.util.TreeSet;  /\*\* \*/  **public** **class** SortSet {  **public** **static** **void** main(String...a){         Collection<Integer> collection = **new** HashSet<Integer>();         collection.add(3);         collection.add(1);         collection.add(2);         collection.add(**null**);         Set<Integer> treeSet = **new** TreeSet<Integer>();         treeSet.addAll(collection);         System.*out*.println(treeSet);     }  } |

### Answer.

### /\*OUTPUT

### Runtime Exception - NullPointerException

### \*/

### *If elements are stored in stored in* [*HashSet*](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)*, then we can use TreeSet’s addAll method for sorting, but* [*TreeSet*](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html) *does not allow null.*

### Collection interview Question 89.

### Collection Output interview question 39.

|  |
| --- |
| import **java.util.Comparator;**  **import** java.util.Map;  **import** java.util.TreeMap;  /\*\* \*/  **public** **class** SortMap {  **public** **static** **void** main(String...a){         Map<Integer, Integer> treeMap = **new** TreeMap<Integer, Integer>(**new** Comparator<Integer>(){  **public** **int** compare(Integer o1, Integer o2) {  **return** o2.compareTo(o1);                   }            });         treeMap.put(4, 1);         treeMap.put(2, 1);         treeMap.put(3, 1);           System.*out*.println(treeMap);     }  } |

### Answer.

### /\*OUTPUT

### {4=1, 3=1, 2=1}

### \*/

### *TreeMap* is sorted by natural order of keys, but we will implement Comparator interface to change the behaviour to sort TreeMap in descending order of keys.

### Here, [Comparator interface has been implemented in form of anonymous inner class.](http://www.javamadesoeasy.com/2015/04/sort-map-by-key-in-ascending-and.html)

### 

### Read:

### [COLLECTION - Top 100 interview questions and answers in java for fresher and experienced in detail - Set-1 > Q1- Q50](http://www.javamadesoeasy.com/2015/05/collection-top-50-interview-questions.html)

### [COLLECTION - Top 100 important interview OUTPUT questions and answers in java, Set-2 > Q51- Q75](http://www.javamadesoeasy.com/2015/07/collection-top-100-important-interview.html)

### 

### Collection interview Question 90.

### Collection Output interview question 40.

|  |
| --- |
| import **java.util.ArrayList;**  **import** java.util.Collections;  **import** java.util.Comparator;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map;  **import** java.util.Map.Entry;  **import** java.util.Set;  **class** Sort **implements** Comparator<Map.Entry<Integer, Integer>>{     @Override  **public** **int** compare( Map.Entry<Integer, Integer> entry1, Map.Entry<Integer, Integer> entry2 ){  **return** (entry2.getValue()).compareTo( entry1.getValue() );     }  }  /\*\* \*/  **public** **class** MyClass {  **public** **static** **void** main(String...a){         Map<Integer, Integer> map = **new** HashMap<Integer, Integer>();         map.put(1, 2);         map.put(2, 1);         map.put(4, 8);           Set<Entry<Integer, Integer>> set = map.entrySet();         List<Entry<Integer, Integer>> list = **new** ArrayList<Entry<Integer, Integer>>(set);         Collections.*sort*(list, **new** Sort());  **for**(Map.Entry<Integer, Integer> entry:list)          System.*out*.print(entry.getKey());     }  } |

### Answer.

### /\*OUTPUT

### 412

### \*/

### Read : [Sort Map by value in Ascending and descending order by implementing Comparator interface and overriding its compare method](http://www.javamadesoeasy.com/2015/04/sort-map-by-value-in-ascending-and.html)

### 

### Collection interview Question 91.

### Collection Output interview question 41.

|  |
| --- |
| import **java.util.ArrayList;**  **import** java.util.Collections;  **import** java.util.Comparator;  **import** java.util.LinkedHashMap;  **import** java.util.List;  **import** java.util.Map;  **import** java.util.Map.Entry;  **import** java.util.Set;  **class** Sort **implements** Comparator<Map.Entry<Integer, Integer>>{     @Override  **public** **int** compare( Map.Entry<Integer, Integer> entry1, Map.Entry<Integer, Integer> entry2 ){  **return** (entry2.getKey()).compareTo( entry1.getKey() );     }  }  /\*\* \*/  **public** **class** SortMap {  **public** **static** **void** main(String...a){         Map<Integer, Integer> map = **new** LinkedHashMap<Integer, Integer>();         map.put(4, 1);         map.put(2, 6);         map.put(5, 1);           Set<Entry<Integer, Integer>> entrySet = map.entrySet();         List<Entry<Integer, Integer>> listOfentrySet = **new** ArrayList<Entry<Integer, Integer>>(entrySet);           Collections.*sort*(listOfentrySet, **new** Sort());    **for**(Map.Entry<Integer, Integer> entry:listOfentrySet)          System.*out*.print(entry.getKey());     }  } |

### Answer.

### /\*OUTPUT

### 542

### \*/

### Read : [Sort Map by value in Ascending and descending order by implementing Comparator interface and overriding its compare method](http://www.javamadesoeasy.com/2015/04/sort-map-by-value-in-ascending-and.html)

### 

### Collection interview Question 92.

### Collection Output interview question 42.

|  |
| --- |
| import **java.util.ArrayList;**  **import** java.util.List;  **public** **class** MyClass {  **public** **static** **void** main(String[] args) {            List<Number> numberList = **new** ArrayList<Number>();            numberList.add(2);            numberList.add(3);  *m*(numberList);     }  **static** **void** m(List<? **super** Double> l) {            System.*out*.print(l.get(0));            System.*out*.print(l.get(1));     }  } |

### Answer.

### /\*

### 23

### \*/

### 

### Collection interview Question 93.

### Collection Output interview question 43.

|  |
| --- |
| import **java.util.ArrayList;**  **import** java.util.List;  **public** **class** MyClass {  **public** **static** **void** main(String[] args) {            List<Integer> l = **new** ArrayList<Integer>();            l.add(2);  *m*(l);       }  **static** **void** m(List<? **super** Double> l) {            System.*out*.println(l.get(0));            System.*out*.println(l.get(1));     }  } |

### Answer.  Program won’t compile.

### List<? super Double> can not accept List<Integer>, it can accept list of anySuperClassOfDouble i.e. List<Number> or List<Object>

### 

### Collection interview Question 94.

### Collection Output interview question 44.

|  |
| --- |
| class **Abc {**     <t> **void** display(t obj[]) {  **for** (t i : obj) {                   System.*out*.print(i + "  ");            }     }  }  **class** MyClass {  **public** **static** **void** main(String... args) {            Abc o = **new** Abc();              Integer i[] = { 1, 2 };            o.display(i);            Double d[] = { 1.1, 2.2 };            o.display(d);     }  } |

### Answer.

### /\*

### 1  2  1.1  2.2

### \*/

### because t can of any type may be Integer or double

### 

### Collection interview Question 95.

### Collection Output interview question 45.

|  |
| --- |
| import **java.util.ArrayList;**  **import** java.util.List;  **public** **class** MyClass {  **public** **static** **void** main(String[] args) {            List<Integer> list = **new** ArrayList<Integer>();            list.add(2);            list.add(3);            System.*out*.println(*sum*(list));     }  **public** **static** **double** sum(List<? **extends** Number> list) {  **double** sum = 0;  **for** (Number num : list) {                   sum += num.doubleValue();            }  **return** sum;     }  } |

### Answer.

### /\*

### 5.0

### \*/

### List<? super Number> can accept List<Integer>, it can accept list of anySubClassOfNumber i.e. List<Double>, List<Float>, etc.

### 

### Collection interview Question 96.

### Collection Output interview question 46.

|  |
| --- |
| import **java.util.ArrayList;**  **import** java.util.List;  **public** **class** MyClass {  **public** **static** **void** main(String[] args) {            List<Integer> list = **new** ArrayList<Integer>();            list.add(2);            list.add(3);  *m*(list);     }  **public** **static** **void** m(List<Number> list) {            System.*out*.println(list);     }  } |

### Answer.  Program won’t compile.

### List<Number> cannot accept List<Integer>, to avoid compilation error we must use List<? extends Number>

### 

### Collection interview Question 97.

### Collection Output interview question 47.

|  |
| --- |
| import **java.util.ArrayList;**  **import** java.util.List;  **public** **class** MyClass {  **public** **static** **void** main(String[] args) {            List<Integer> list = **new** ArrayList<Integer>();            list.add(1);            list.add(2);            System.*out*.println(*sum*(list));     }  **public** **static** **double** sum(List<? **extends** Number> list) {            list.add(4);  **double** sum = 0;  **for** (Number num : list) {                   sum += num.doubleValue();            }  **return** sum;     }  } |

### Answer.  Program won’t compile.

### List<? extends Number> cannot add or remove elements from list. So, list.add(4) will cause compilation error.

### 

### Collection interview Question 98.  Output

### Collection Output interview question question 48.

|  |
| --- |
| import **java.util.PriorityQueue;**  **public** **class** MyClass {  **public** **static** **void** main(String[] args) {            PriorityQueue<Integer> q = **new** PriorityQueue<Integer>();            q.add(1);            q.add(2);            q.add(3);            System.*out*.println(q.poll());            System.*out*.println(q.offer(4));            q.add(1);            q.remove(2);            System.*out*.println(q.peek());            System.*out*.println(q);     }  } |

### Answer.

### /\* OUTPUT

### 1

### true

### 1

### [1, 3, 4]

### \*/

### 

### [ArrayList custom implementation - add, get, remove Employee object in java](http://www.javamadesoeasy.com/2015/02/arraylist-custom-implementation-add-get.html)

*Contents of page :*

* [***1) Methods used in custom ArrayList in java****-*](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation-add-get.html#1)
* [***2) Full Program/SourceCode to add, get, remove Employee object***](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation-add-get.html#2) ***in*** [***custom ArrayList in java.***](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html)

In this post i will be explaining how to **add, get, remove Employee object in** [**custom ArrayList.**](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html)

**1) Methods used in custom ArrayList in java**-

|  |  |
| --- | --- |
| public void **add**(E value) | Add objects in **ArrayListCustom** |
| public E **get**(int index) | Method returns element on specific index. |
| public Object **remove**(int index) | Method returns removedElement on specific index, else it throws IndexOutOfBoundException if index is negative or greater than size of size. |
| public void **display**() | -Method displays all objects in **ArrayListCustom**.  **-Insertion order is guaranteed**. |
| private void **ensureCapacity**() | Method increases capacity of list by making it double. |

**2) Full Program/SourceCode to add, get, remove Employee object in** [**custom ArrayList in java.**](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html)

|  |
| --- |
| **package** com.ankit;  **import** java.util.Arrays;  **class** **Employee** {  **private** String id;  **private** String name;       /\*\*     \* Employee constructor     \*/  **public** Employee(String id, String name) { // constructor  **this**.id = id;  **this**.name = name;     }     @Override  **public** String toString() {  **return** "Employee[id=" + id + ", name=" + name + "] ";     }  }  /\*\*   \*/  /\*\*  \* **@author** AnkitMittal  \* . All Contents are copyrighted and must not be reproduced in any form.  \* This class provides custom implementation of ArrayList(without using java api's)  \* Insertion order of objects is maintained.  \* Implementation allows you to store null as well.  \* **@param** <E>  \*/  **class** **ArrayListCustom**<E> {    **private** **static** **final** **int** *INITIAL\_CAPACITY* = 10;  **private** Object elementData[]={};  **private** **int** size = 0;   /\*\*   \* constructor.   \*/  **public** **ArrayListCustom**() {     elementData = **new** Object[*INITIAL\_CAPACITY*];   }   /\*\*    \* method adds elements in ArrayListCustom.    \*/  **public** **void** **add**(E e) {  **if** (size == elementData.length) {       ensureCapacity(); //increase current capacity of list, make it double.     }     elementData[size++] = e;   }   /\*\*    \* method returns element on specific index.    \*/   @SuppressWarnings("unchecked")  **public** E **get**(**int** index) {  **if** ( index <0 || index>= size) { //if index is negative or greater than size of size, we throw Exception.  **throw** **new** IndexOutOfBoundsException("Index: " + index + ", Size " + index);     }  **return** (E) elementData[index]; //return value on index.   }   /\*\*    \* method returns removedElement on specific index.    \* else it throws IndexOutOfBoundException if index is negative or greater than size of size.    \*/  **public** Object **remove**(**int** index) {  **if** ( index <0 || index>= size) { //if index is negative or greater than size of size, we throw Exception.  **throw** **new** IndexOutOfBoundsException("Index: " + index + ", Size " + index);     }       Object removedElement=elementData[index];  **for**(**int** i=index;i<size - 1;i++){        elementData[i]=elementData[i+1];     }     size--;   //reduce size of ArrayListCustom after removal of element.    **return** removedElement;   }   /\*\*    \* method increases capacity of list by making it double.    \*/  **private** **void** **ensureCapacity**() {  **int** newIncreasedCapacity = elementData.length \* 2;     elementData = Arrays.*copyOf*(elementData, newIncreasedCapacity);   }   /\*\*    \* method displays all the elements in list.    \*/  **public** **void** **display**() {      System.*out*.print("Displaying list : ");  **for**(**int** i=0;i<size;i++){             System.*out*.print(elementData[i]+" ");      }   }  }  /\*\*   \*/  /\*\*  \* Main class to test ArrayListCustom functionality.  \*/  **public** **class** **ArrayListEmployee** {    **public** **static** **void** main(String...a) {            ArrayListCustom<Employee> list = **new** ArrayListCustom<Employee>();     list.add(**new** Employee("1", "sam"));     list.add(**new** Employee("2", "amy"));     list.add(**new** Employee("3", "wil"));     list.add(**new** Employee("4", "cat"));     list.add(**new** Employee("1", "sam"));     list.add(**new** Employee("2", "amy"));     list.add(**null**);       list.display();     System.*out*.println("\nelement at index "+1+" = "+list.get(1));     System.*out*.println("element removed from index "+1+" = "+list.remove(1));       System.*out*.println("\nlet's display list again after removal at index 1");       list.display();       //list.remove(11); //will throw IndexOutOfBoundsException, because there is no element to remove on index 11.     //list.get(11);   //will throw IndexOutOfBoundsException, because there is no element to get on index 11.        }    }  /\***Output**  Displaying list : Employee[id=1, name=sam]  Employee[id=2, name=amy]  Employee[id=3, name=wil]  Employee[id=4, name=cat]  Employee[id=1, name=sam]  Employee[id=2, name=amy]  null  element at index 1 = Employee[id=2, name=amy]  element removed from index 1 = Employee[id=2, name=amy]  let's display list again after removal at index 1  Displaying list : Employee[id=1, name=sam]  Employee[id=3, name=wil]  Employee[id=4, name=cat]  Employee[id=1, name=sam]  Employee[id=2, name=amy]  null  \*/ |

**Collection interview Question 1. What is Collection framework in java?**

**Answer**. It’s the basic Collection framework interview question. Freshers must know about this.  [***java.util.Collection***](http://www.javamadesoeasy.com/2015/04/collection-in-java.html)*is the* root interface in the *hierarchy of Java Collection framework in java*.

The JDK does not provide any classes which directly implements this interface, but it provides classes which are implementations of more specific subinterfaces like [Set and List](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) in java.

java.util.[**Set**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) extends java.util.Collection interface in java.

[**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)***,*** [**CopyOnWriteArraySet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html)***,*** [**LinkedHashSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html)***,*** [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [**ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html), [**EnumSet**](http://www.javamadesoeasy.com/2015/04/enumset-in-java-with-program.html) classes implements [**Set**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface.

java.util.[**List**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html) extends java.util.Collection interface in java.

[**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)***,*** [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html)***,*** [**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)***,*** [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html) classes implements [**List**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface.

***Also read >***

[COLLECTION - Top 100 important interview OUTPUT questions and answers in java, Set-2 > Q51- Q75](http://www.javamadesoeasy.com/2015/07/collection-top-100-important-interview.html)

### [COLLECTION - Top 100 important interview OUTPUT questions and answers in java, Set-3 > Q75- Q100](http://www.javamadesoeasy.com/2015/07/collection-top-100-important-interview45.html)

**Collection interview Question 2. Which interfaces and classes are most frequently used in Collection framework in java?**

**Answer**. This collection framework interview question will test your practical knowledge. Freshers may get away by answering few interface and classes but experienced developers must answer this question in detail.

**Most frequently used interface in Collection framework are >**

[**List**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html), [**Set**](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html) and [**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html).

**Most frequently used classes in Collection framework are >**

[**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)***,*** [**LinkedHashSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html)***,*** [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [**ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html) classes implements [**Set**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface.

[**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)***,*** [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html)***,*** [**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)***,*** [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html) classes implements [**List**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface.

[**HashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-in-java.html)***,*** [**Hashtable**](http://www.javamadesoeasy.com/2015/04/hashmap-and-hashtable-similarity-and.html)***,*** [**ConcurrentHashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html)***,*** [**LinkedHashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html), [**TreeMap**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html), [**ConcurrentSkipListMap**](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html) classes implements **Map** interface.

**Collection interview Question 3. What are subinterfaces of Collection interface in java? Is Map interface also a subinterface of Collection interface in java?**

**Answer**. [**List**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html) and [**Set**](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html) are subinterfaces of java.util.[**Collection**](http://www.javamadesoeasy.com/2015/04/collection-in-java.html) in java.

*It’s important to note* [***Map***](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) *interface is a member of the Java Collections Framework, but it does not implement Collection interface in java.*

**Collection interview Question 4. What are differences between** [**ArrayList and LinkedList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-linkedlist-similarity-and.html) **in java?**

**Answer**. This is very important collection framework interview question in java.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.ArrayList*** | **java.util.LinkedList** |
| 1 | Structure | java.util.ArrayList is index based structure in java.  [https://lh3.googleusercontent.com/wCfo_q1uxCzZZCgGpetqEQYMeVj9YMJokT9-WJ7QY4jxCF11u5-WVIjVheBCfKlPJtQ9Bp5zzxTJcPgYLMr0N3n6PvjXPzd-7O-FJr2KoW7qrUjERB-yXK2YxFkH6qrLAX6hvdg5](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html) | A java.util.**LinkedList** is a data structure consisting of a group of **nodes** which together represent a sequence.  node is composed of a data and a reference (in other words, a **link**) to the next node in the sequence in java.  [https://lh3.googleusercontent.com/ykSE04usYkDTj50vuGVTWKtVGJootTOKa07Eub-E6D5KkOCNAb399G4agtbSKOyeaPAUvAngY6JjDMs-SBNmblDOXLv62eHNVIwEuGD5-GNXTP45Ubtyp0BYg0seOxGSpXHatWJP](http://www.javamadesoeasy.com/2015/01/doublylinkedlist-insert-and-delete-at.html) |
| 2 | **Resizable** | **ArrayList is Resizable-array in java.** | New node is created for storing new element in LinkedList in java. |
| 3 | **Initial capacity** | java.util.ArrayList is created with initial capacity of 10 in java. | For storing every element node is created in LinkedList, so linkedList’s initial capacity is 0 in java. |
| 4 | Ensuring **Capacity**/ resizing. | ArrayList is created with initial capacity of 10.  ArrayList’s size is **increased by 50%** i.e. after resizing it’s size become 15 in java. | For storing every element node is created, so linkedList’s initial capacity is 0, it’s size grow with addition of each and every element in java. |
| 5 | RandomAccess interface | ArrayList implements RandomAccess(Marker interface) to indicate that they support fast random access (i.e. index based access) in java. | LinkedList does not implement RandomAccess interface in java. |
| 6 | AbstractList and AbstractSequentialList | ArrayList extends AbstractList (abstract class) which provides implementation to  List interface to minimize the effort required to implement this interface backed by RandomAccess interface. | LinkedList extends AbstractSequentialList (abstract class), AbstractSequentialList extends AbstractList.  In LinkedList, data is accessed sequentially, so for obtaining data at specific index, iteration is done on nodes sequentially in java. |
| 7 | How **get(index)** method works?  (Though difference has been discussed briefly in above 2 points but in this in point we will figure difference in detail.) | Get method of ArrayList directly gets element on specified index. Hence, offering O(1) complexity in java. | Get method of LinkedList iterates on nodes sequentially to get element on specified index. Hence, offering O(n) complexity in java. |
| **8** | **When to use** | **Use ArrayList when get operations is more frequent than add and remove operations in java.** | **Use LinkedList when add and remove operations are more frequent than get operations in java.** |

For more detail like complexity comparison of method please read : [**ArrayList vs LinkedList in java**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-linkedlist-similarity-and.html)

**Collection interview Question 5. What are differences between** [**ArrayList and Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html) **in java?**

**Answer**. Another very important collection framework interview question to differentiate between ArrayList and Vector in java.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.ArrayList*** | ***java.util.Vector*** |
| 1 | synchronization | java.util.ArrayList is **not synchronized**  (because 2 threads on same ArrayList object can access it at same time).  I have created [**program**](http://www.javamadesoeasy.com/2015/05/consequence-of-using-arraylist-in.html)to show consequence of using ArrayList in multithreading environment.  In the program we will implement our own arrayList in java. | java.util.Vector is **synchronized** (because 2 threads on same Vector object cannot  access it at same time).  I have created [**program**](http://www.javamadesoeasy.com/2015/05/advantage-of-using-vector-in.html)to show advantage of using Vector in multithreading environment.  In the program we will implement our own vector in java. |
| 2 | Performance | ArrayList is not synchronized, hence its operations are **faster** as compared to Vector in java. | Vector is synchronized, hence its operations are **slower** as compared to ArrayList in java.  If we are working not working in multithreading environment jdk recommends us to use ArrayList. |
| 3 | Enumeration | **Enumeration** is [**fail-fast**](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html), means any modification made to ArrayList during iteration using Enumeration will throw ConcurrentModificationException in java. | **Enumeration** is **fail-safe**, means any modification made to Vector during iteration using Enumeration don’t throw any exception in java. |
| 4 | Introduced  in which java version | ArrayList was introduced in second version of java i.e. **JDK 2.0** | Vector was introduced in first version of java i.e. **JDK 1.0**  But it was refactored in java 2 i.e. JDK 1.2 to implement the List interface, hence making it a member of member of the [Java Collections Framework](http://www.javamadesoeasy.com/2015/04/collection-in-java.html). |
| 5 | Ensuring Capacity/ resizing. | ArrayList is created with initial capacity of 10.  When its full size is **increased by 50%** i.e. after resizing it’s size become 15 in java. | Vector is created with initial capacity of 10.  Vector’s size is **increased by 100%** i.e. after resizing it’s size become 20 in java. |
| 6 | Custom implementation | [https://lh3.googleusercontent.com/2yHNtovknpsdxOKpK4Sd3oFiHP3fKhhrMsZDH3DJaRNsvWB7RnEqtXjyS5yrk6175OwELqF6-viscZQxK8uMK58-gmsz1tN0sHmVSJBEwKJ1UZwle61DItNZeF8MDwyFx-NUXHDU](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html)Read : [ArrayList custom implementation](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html) | [https://lh5.googleusercontent.com/Gm_SkaJR6TXlRpDt3ipjw739Gfcg4b2V-pMT4WWrw0cMVTsUhDvApjrvMATYF4XiBotrk8O0Sbc7kwi_v0V7SrrY_cfTUU5dR0_rEogBBG34UB1IviiWwIHcpc5XuH_k5KmOsnmP](http://javamadesoeasy.com/2015/02/vector-custom-implementation.html)Read :  [Vector custom implementation](http://javamadesoeasy.com/2015/02/vector-custom-implementation.html) |

For more detail like complexity comparison of method please read: [**ArrayList vs Vector- Similarity and Differences in java**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)

**Collection interview Question 6. What are differences between** [**List and Set**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) **interface in java?**

**Answer**. Another very very important collection framework interview question to differentiate between **List and Set** in java.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.List*** | ***java.util.Set*** |
| 1 | Insertion order | java.util.List is ordered collection it **maintain insertion order** in java. | *Most of the java.util.Set implementation* does not **maintain insertion order**.  HashSet does not maintains insertion order in java.  Thought LinkedHashSet maintains insertion order in java.    TreeSet is sorted by natural order in java. |
| 2 | Duplicate elements | List **allows to store duplicate elements** in java. | *Set does* ***not allow to store duplicate elements*** in java*.* |
| 3 | Null keys | List allows to store **many null keys** in java. | Most of the Set implementations allow to add only **one null** in java**.**  TreeSet does not allow to add null in java. |
| 4 | Getting element on specific **index** | List implementations provide get method to get element on specific index in java.  ArrayList, Vector, copyOnWriteArrayList and LinkedList provides -  *get(int index)*  Method returns element on specified *index*.  **Get method directly gets element on specified index. Hence, offering O(1) complexity.** | Set implementations does not provide any such get method to get element on specified index in java. |
| 5 | Implementing classes | [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)***,*** [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html)***,*** [**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)***,*** [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html) classes implements [**List**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface in java. | [**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)***,*** [**CopyOnWriteArraySet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html)***,*** [**LinkedHashSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html)***,*** [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [**ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html), [**EnumSet**](http://www.javamadesoeasy.com/2015/04/enumset-in-java-with-program.html) classes implements [**Set**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface in java. |
| 6 | listIterator | **listIterator** method returns listIterator to iterate over elements in List in java.  **listIterator provides** additional methods as compared to iterator like  **hasPrevious(), previous(), nextIndex(), previousIndex(), add(E element), set(E element)** | Set does not provide anything like listIterator. It simply return Iterator in java. |
| 7 | Structure and resizable | **List** are Resizable-array implementation of the java.util.**List** interface in java. | Set uses [**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html)for their implementation.  Hence, structure is map based and resizing depends on Map implementation.  *Example >* [***HashSet***](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html) *internally uses* [*HashMap*](http://javamadesoeasy.com/2015/02/hashmap-custom-implementation.html)*.* |
| 8 | Index based structure /RandomAccess | As **ArrayList** uses array for implementation it is index based structure, hence provides random access to elements.  But [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html) is not indexed based structure in java. | Set is not index based structure at all in java. |

For more detail read : [**List vs Set - Similarity and Differences in java**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html)

**Collection interview Question 7. What are differences between** [**Iterator and ListIterator**](http://www.javamadesoeasy.com/2015/04/iterator-vs-listiterator-similarity-and.html)**? in java**

**Answer**. This collection framework interview question is tests your knowledge of iterating over different collection framework classes in java.

|  |  |  |
| --- | --- | --- |
|  | ***java.util.ListIterator*** | ***java.util.Iterator*** |
| 1 | **hasPrevious()**  method returns true if this listIterator has more elements when traversing the list in the reverse direction. | **No such method** in java.util.Iterator. |
| 2 | **previous()**  returns previous element in iteration (traversing in backward direction).  if the iteration has no previous elements than NoSuchElementException is thrown. | **No such method** in java.util.Iterator. |
| 3 | **nextIndex()**  method returns the index of the element that would be returned by a subsequent call to next() method. If listIterator is at the end of the list than method returns size of list. | **No such method** in java.util.Iterator. |
| 4 | **previousIndex()**  method returns the index of the element that would be returned by a subsequent call to previous() method. If listIterator is at the start of the list than method returns -1. | **No such method** in java.util.Iterator. |
| 5 | **add(E element)**  Method inserts the specified **element** into the list.  The element is inserted immediately before the element that would be returned by next (So, subsequent call to next would be unaffected), if any, and after the element that would be returned by previous (So,subsequent call to previous would return the new **element**), if any.  If the list does not contain any element than new **element** will be the sole element in the list. | **No such method** in java.util.Iterator. |
| 6 | **set(E element)**  Method replaces the last element returned by next() or previous() method with the specified **element**. This call can be made only if neither remove nor add have been called after the last call to next or previous.  If call to set() method is followed up by any call made to remove() or add() method after next() or previous() than UnsupportedOperationException is thrown. | **No such method** in java.util.Iterator. |
| 7 | All the implementations of [**List**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html) interface like [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)***,*** [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html), [**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)***,*** [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html) classes returns listIterator. | All Implementation classes of [**Collection**](http://www.javamadesoeasy.com/2015/04/collection-in-java.html) interface’s subinterfaces like [Set and List](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) return iterator. |

For more detail read : [**Iterator vs ListIterator - Similarity and Differences in java**](http://www.javamadesoeasy.com/2015/04/iterator-vs-listiterator-similarity-and.html)

**Collection interview Question 8. What are differences between** [**Collection and Collections**](http://www.javamadesoeasy.com/2015/04/collection-vs-collections-differences.html) **in java?**

**Answer**.  This is another very important collection framework interview question.In real projects you must have used both Collection and Collections but what is the difference between two of them in java?

java.util.[***Collection***](http://www.javamadesoeasy.com/2015/04/collection-in-java.html) ***​*** *is the* root **interface** in the ​*hierarchy of Java Collection framework​*.

The JDK does not provide any classes which directly implements java.util.Collection interface, but it  provides classes such as [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html), [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html), [**vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html), [**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html), [**EnumSet**](http://www.javamadesoeasy.com/2015/04/enumset-in-java-with-program.html), [**LinkedHashSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), [CopyOnWriteArrayList](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html), [CopyOnWriteArraySet](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html), [ConcurrentSkipListSet](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html)  which implements more specific subinterfaces like ​[Set and List​](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) in java.

java.util.**Collections** is a utility **class** which **consists** of **static methods** that **operate on** or return **Collection** in java.

**java.util.Collections provides method like >**

* **reverse** method for reversing [**List**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html) in java.
* **shuffle** method for shuffling elements of **List** in java.
* **unmodifiableCollection**, [**unmodifiableSet**](http://www.javamadesoeasy.com/2015/04/hashset-making-set-unmodifiable-using.html), [**unmodifiableList**](http://www.javamadesoeasy.com/2015/04/arraylist-making-list-unmodifiable.html), [**unmodifiableMap**](http://www.javamadesoeasy.com/2015/04/hashmap-making-map-unmodifiable-using.html) methods for making **List**, [**Set**](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html) and [**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) unmodifiable in java.
* **min** method to return smallest element in **Collection** in java.
* **max** method to return smallest element in **Collection**.
* **sort** method for sorting **List**.
* **synchronizedCollection**, [**synchronizedSet**](http://www.javamadesoeasy.com/2015/04/hashset-synchronizing-using.html), [**synchronizedList**](http://www.javamadesoeasy.com/2015/04/arraylist-synchronizing-using.html), [**synchronizedMap**](http://www.javamadesoeasy.com/2015/04/hashmap-synchronizing-map-using.html)methods for synchronizing **List**, **Set** and **Map** respectively in java**.**

Additionally you must know that *java.util.Collection and java.util.Collections both were introduced in* ***second version of java i.e. in JDK 2.0.***

**Collection interview Question 9. What are core classes and interfaces in java.util.List hierarchy in java?**

**Answer**. Freshers must know core classes in List hierarchy but experienced developers must be able to explain this java.util.List hierarchy in detail.

[](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html)

java.util.[**List**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface extends java.util.Collection interface.

java.util.[**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)***, java.util.***[**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html)***, java.util.***[**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html)***, java.util.concurrent.***[**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html)classes implements java.util.[**List**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface.

Also some abstract classes like java.util.**AbstractCollection**, java.util.**AbstractList** and java.util.**AbstractSequentialList** have been mentioned in hierarchy.

**Collection interview Question 10. What are core classes and interfaces in java.util.Set hierarchy?**

**Answer**. Freshers must know core classes in Set hierarchy but experienced developers must be able to explain this java.util.Set hierarchy in detail.

[](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html)

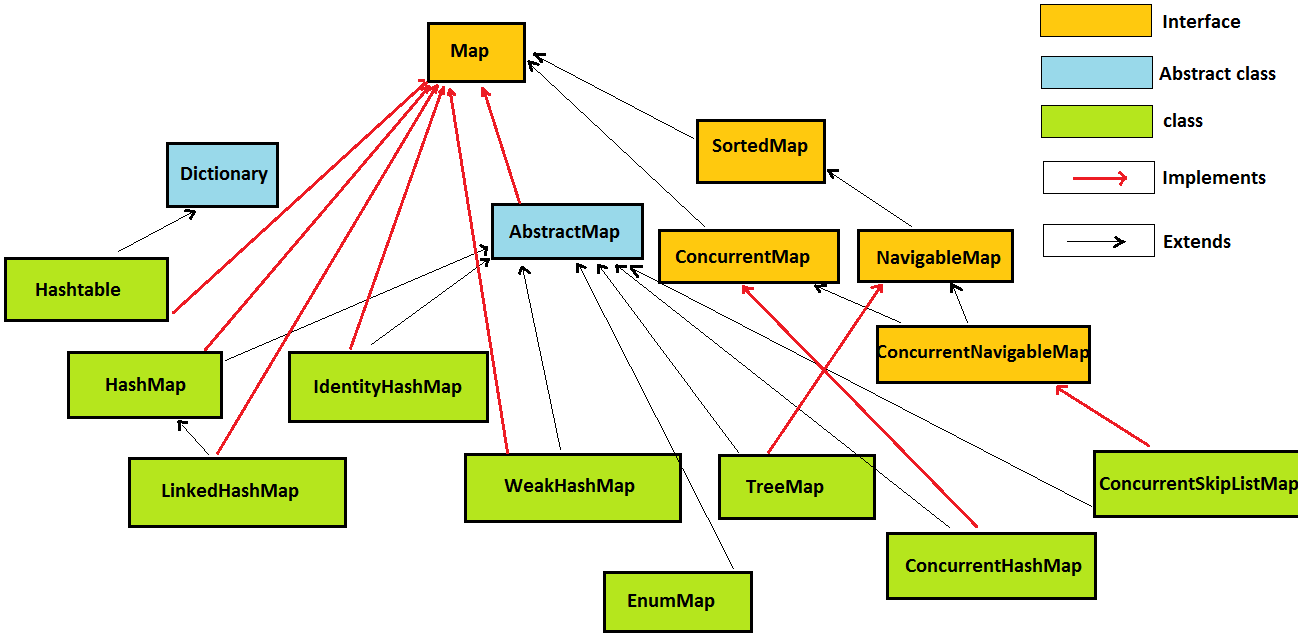
java.util.[**Set**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface extends java.util.Collection interface.

java.util.[**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)***, java.util.concurrent.***[**CopyOnWriteArraySet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html)***, java.util.***[**LinkedHashSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html)***, java.util.***[**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html), java.util.concurrent.[**ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html), java.util.[**EnumSet**](http://www.javamadesoeasy.com/2015/04/enumset-in-java-with-program.html) classes implements java.util.[**Set**](http://www.javamadesoeasy.com/2015/04/list-vs-set-similarity-and-differences.html) interface.

Also some abstract classes like java.util.**Dictionary** and java.util.**AbstractSet** and java.util.**AbstractCollection** have been mentioned in hierarchy.

**Collection interview Question 11. What are core classes and interfaces in java.util.Map hierarchy?**

**Answer**. Freshers must know core classes in Map hierarchy but experienced developers must be able to explain this java.util.Map hierarchy in detail.

[](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html)

**java.util.Map** interface extends java.util.Collection interface.

java.util.[**HashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-in-java.html)***, java.util.***[**Hashtable**](http://www.javamadesoeasy.com/2015/04/hashmap-and-hashtable-similarity-and.html)***, java.util.concurrent.***[**ConcurrentHashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html)***, java.util.***[**LinkedHashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html), java.util.[**TreeMap**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html), java.util.concurrent.[**ConcurrentSkipListMap**](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html), java.util.[**IdentityHashMap**](http://www.javamadesoeasy.com/2015/04/identityhashmap-in-java.html), java.util.[**WeakHashMap**](http://www.javamadesoeasy.com/2015/04/weakhashmap-in-java.html), java.util.[**EnumMap**](http://www.javamadesoeasy.com/2015/04/enummap-in-java-with-program.html) classes implements java.util.**Map** interface.

Also some abstract classes like java.util.**Dictionary** and java.util.**AbstractMap** have been mentioned in hierarchy.

**Collection interview Question 12.  What are differences between** [**Iterator and Enumeration**](http://www.javamadesoeasy.com/2015/04/iterator-vs-enumeration-differences-and.html) **in java?**

**Answer**. Experienced developers must be well versed to answer this collection framework interview question in java.

***Differences*** *between java.util.****Iterator*** *and java.util.****Enumeration*** *in java**>*

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***Property*** | ***java.util.Enumeration*** | ***java.util.Iterator*** |
| 1 | Remove elements during iteration | java.util.Enumeration **doesn’t allows** to remove elements from collection during iteration in java. | java.util.Iterator **allows** to remove elements from collection during iteration by using **remove()** method in java. |
| 2 | Improved naming conventions in Iterator | **nextElement()**  Method Returns the next element of this enumeration if this enumeration object has at least one more element to provide.  **hasMoreElements()**  returns true if enumeration contains more elements. | **nextElement()** has been changed to **next()** in Iterator.  And  **hasMoreElements()** has been changed to **hasNext()** in Iterator. |
| 3 | Introduced in  which java  version | Enumeration was introduced in first version  of java i.e. ​**JDK 1.0** | Iterator was introduced in second version  of java i.e. ​**JDK 2.0**  Iterator was introduced to replace Enumeration in the Java Collections Framework. |
| 4 | Recommendation | **Java docs** recommends iterator over enumeration**.** | **Java docs** recommends iterator over enumeration**.** |
| 5 | Enumeration and Iterator over [**Vector**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html) | **Enumeration** returned by Vector is [**fail-safe**](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html), means any modification made to Vector during iteration using Enumeration don’t throw any exception in java. | **Iterator** returned by Vector are [**fail-fast**](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)**,** means any structural modification made to ArrayList during iteration will throw ConcurrentModificationException  in java. |

### For more detail read : [Iterator vs Enumeration - Differences and similarities in java](http://www.javamadesoeasy.com/2015/04/iterator-vs-enumeration-differences-and.html)

**Collection interview Question 13. How do we override equals and hashcode method in java, write a code to use Employee as key in HashMap in java? (Important)**

**Answer**.  This is one of the most important collection framework interview question in java. Prepare for this question properly. Freshers must know the concept how to override equals and hashcode method but experienced developers must be able to write the java code to override equals and hashcode neatly. We will override equals() and hashCode() like this -

By overriding equals() and hashCode() method we could use custom object as key in HashMap.

1)  Check whether obj is null or not.

**if(obj==null) //If obj is null, return without comparing obj & Employee class.**

2)  check whether  obj is instance of Employee class or not.

**if(this.getClass()!=obj.getClass()) //identifies whether obj is instance of Employee class or not.**

3) Then, type cast obj into employee instance.

**Employee emp=(Employee)obj;  //type cast obj into employee instance.**

|  |
| --- |
| **@Override**  **public boolean equals(Object obj){**    **if(obj==null)**  **return false;**    **if(this.getClass()!=obj.getClass())**  **return false;**    **Employee emp=(Employee)obj;**  **return (emp.id==this.id || emp.id.equals(this.id))**  **&& (emp.name==this.name || emp.name.equals(this.name));**  **}**    **@Override**  **public int hashCode(){**  **int hash=(this.id==null ? 0: this.id.hashCode() ) +**  **(this.name==null ? 0: this.name.hashCode() );**  **return hash;**  **}** |

Let’s say in an organisation there exists a employee with **id=1 and name=’sam’**     and **some data** is stored corresponding to him, but if modifications have to be made in data, **previous data must be overridden**.

[DETAILED DESCRIPTION : Override equals() and hashCode() method](http://www.javamadesoeasy.com/2015/02/override-equals-and-hashcode-method.html).

**Must read :** [**Overriding equals and hashcode method - Top 18 Interview questions in java**](http://www.javamadesoeasy.com/2015/02/overriding-equals-and-hashcode-method.html)

**Collection interview Question 14. What classes should i prefer to use a key in HashMap in java? (Important)**

**Answer**. This collection framework interview question will check your in depth knowledge of Java’s Collection Api’s. we should prefer **String, Integer, Long, Double, Float, Short and any other wrapper class.** Reason behind using them as a key is that they override equals() and hashCode() method, we need not to write any explicit code for overriding equals() and hashCode() method in java.

Let’s use Integer class as key in HashMap(Example) -

|  |
| --- |
| **import** java.util.HashMap;  **import** java.util.Map;  **public** **class** StringInMapExample {  **public** **static** **void** main(String...a){             //HashMap's key=Integer class  (Integer’s api has already overridden hashCode() and equals() method for us )            Map<Integer, String> hm=**new** HashMap<Integer, String>();            hm.put(1, "data");            hm.put(1, "data OVERRIDDEN");              System.*out*.println(hm.get(1));       }  }  /\*OUTPUT  data OVERRIDDEN  \*/ |

If, we note above program, what we will see is we didn’t override equals() and hashCode() method, but still we were able to store data in HashMap, override data and retrieve data using get method.

>Let’s check in **Integer’s API**, how Integer class has overridden equals() and hashCode() method :

|  |
| --- |
| **public** **int** **hashCode**() {  **return** value;  }  **public** **boolean** **equals**(Object obj) {  **if** (obj **instanceof** Integer) {  **return** value == ((Integer)obj).intValue();         }  **return** **false**;  } |

**Collection interview Question 15. What are differences between** [**HashMap and Hashtable**](http://www.javamadesoeasy.com/2015/04/hashmap-and-hashtable-similarity-and.html) **in java?**

**Answer**. Fresher and Experienced developers must answer this important collection framework interview question in detail in java.

***Differences*** *between java.util.*[***HashMap***](http://www.javamadesoeasy.com/2015/04/hashmap-in-java.html) *and java.util.****Hashtable*** *in java >*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.HashMap*** | ***java.util.Hashtable*** |
| 1 | synchronization | java.util.HashMap is **not synchronized**  (because 2 threads on same HashMap object can access it at same time) in java. | java.util.Hashtable is **synchronized** (because 2 threads on same Hashtable object cannot access it at same time) in java. |
| 2 | Performance | HashMap is not synchronized, hence its operations are **faster** as compared to Hashtable in java. | Hashtable is synchronized, hence its operations are **slower** as compared to HashMap in java.  If we are working not working in multithreading environment jdk recommends us to use HashMap. |
| 3 | Null keys and values | HashMap allows to store **one null key** and **many null values** i.e. many keys can have null value in java. | Hashtable does **not allow to store null key or null value**.  Any attempt to store null key or value throws runtimeException (NullPointerException) in java. |
| 4 | Introduced  in which java version | HashMap was introduced in second version of java i.e. **JDK 2.0** | Hashtable was introduced in first version of java i.e. **JDK 1.0**  But it was refactored in java 2 i.e. JDK 1.2 to implement the Map interface, hence making it a member of member of the [Java Collections Framework](http://download.oracle.com/javase/7/docs/technotes/guides/collections/index.html). |
| 5 | Recommendation | In non-multithreading environment it is recommended to use HashMap than using Hashtable in java. | I**n java 5 i.e. JDK 1.5**, it is **recommended** to use [ConcurrentHashMap](http://www.javamadesoeasy.com/2015/04/concurrenthashmap-in-java.html) than using Hashtable. |
| 6 | Extends Dictionary (Abstract class, which is obsolete) | HashMap does not extends Dictionary in java. | Hashtable extends Dictionary (which maps non-null keys to values. In a given Dictionary we can look up value corresponding to key) in java. |

For more detail read : [**HashMap and Hashtable - Similarity and Differences in java**](http://www.javamadesoeasy.com/2015/04/hashmap-and-hashtable-similarity-and.html)

**Collection interview Question 16. when to use** [**HashSet vs LinkedHashSet vs TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html) **in java?**

**Answer**. Another very important collection framework interview question to differentiate between **following Set implementations** in java.

***Differences*** *between java.util.*[***HashSet***](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html) *vs java.util.****LinkedHashSet*** *vs java.util.****TreeSet*** *in java>*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Property | *java.util.HashSet* | *java.util.LinkedHashSet* | *java.util.TreeSet* |
| 1 | Insertion order | java.util.HashSet does not maintains insertion order in java.  Example in java >  **set.add("b");**  **set.add("c");**  **set.add("a");**  Output >  **No specific order** | java.util.LinkedHashSet maintains insertion order in java.  Example in java >  **set.add("b");**  **set.add("c");**  **set.add("a");**  Output >  **b**  **c**  **a** | java.util.TreeSet is sorted by natural order in java.  Example in java >  **set.add("b");**  **set.add("c");**  **set.add("a");**  Output >  **a**  **b**  **c** |
| 2 | Null elements | HashSet allows to store **one null** in java**.** | LinkedHashSet allows to store **one null** in java. | TreeSet does **not** allows to store **any null** in java.  Any attempt to add null throws runtimeException (NullPointerException). |
| 3 | Data structure internally used for storing data | For storing elements HashSet internally uses HashMap. | For storing elements LinkedHashSet internally uses  LinkedHashMap. | For storing elements TreeSet internally uses TreeMap. |
| 4 | Introduced  in which java version | java.util.HashSet was introduced in second version of java (1.2) i.e. **JDK 2.0** | java.util.LinkedHashSet was introduced in second version of java (1.4) i.e. **JDK 4.0** | java.util.TreeSet was introduced in second version of java (1.2) i.e. **JDK 2.0** |
| 5 | Implements which interface | HashSet implements **java.util.**[**Set**](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html)interface. | LinkedHashSet implements **java.util.Set** interface. | TreeSet implements **java.util.Set**  **java.util.SortedSet**  **java.util.NavigableSet** interface. |

For more detail read : [**HashSet vs LinkedHashSet vs TreeSet in java**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html)

# LinkedHashSet class in Java with Examples

[**2.2**](https://www.geeksforgeeks.org/easy/)

A LinkedHashSet is an ordered version of [HashSet](http://quiz.geeksforgeeks.org/hashset-in-java/) that maintains a doubly-linked List across all elements. When the iteration order is needed to be maintained this class in used. When iterating through a [HashSet](http://quiz.geeksforgeeks.org/hashset-in-java/) the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted.when cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

**Syntax:**

LinkedHashSet<String> hs = new LinkedHashSet<String>();

* Contains unique elements only like [HashSet](http://quiz.geeksforgeeks.org/hashset-in-java/). It extends [HashSet](http://quiz.geeksforgeeks.org/hashset-in-java/) class and implements Set interface.
* Maintains insertion order.

Basic **Operations** of LinkedHashSet:

|  |
| --- |
| import java.util.LinkedHashSet;  public class Demo  {      public static void main(String[] args)      {          LinkedHashSet<String> linkedset =                             new LinkedHashSet<String>();            // Adding element to LinkedHashSet          linkedset.add("A");          linkedset.add("B");          linkedset.add("C");          linkedset.add("D");            //This will not add new element as A already exists          linkedset.add("A");          linkedset.add("E");            System.out.println("Size of LinkedHashSet = " +                                      linkedset.size());          System.out.println("Original LinkedHashSet:" + linkedset);          System.out.println("Removing D from LinkedHashSet: " +                              linkedset.remove("D"));          System.out.println("Trying to Remove Z which is not "+                              "present: " + linkedset.remove("Z"));          System.out.println("Checking if A is present=" +                              linkedset.contains("A"));          System.out.println("Updated LinkedHashSet: " + linkedset);      }  } |

Run on IDE

**Output:**

Size of LinkedHashSet=5

Original LinkedHashSet:[A, B, C, D, E]

Removing D from LinkedHashSet: true

Trying to Remove Z which is not present: false

Checking if A is present=true

Updated LinkedHashSet: [A, B, C, E]

[LinkedHashmap](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/) vs LinkedHashset

* [LinkedHashMap](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/) does a mapping of keys to values whereas a LinkedHashSet simply stores a collection of things with no duplicates.
* LinkedHashMap extends HashMap and LinkedHashSet extends HashSet.

**Important :** Keeping the insertion order in both LinkedHashmap and LinkedHashset have additional associated costs, both in terms of spending additional CPU cycles and needing more memory. If you do not need the insertion order maintained, it is recommended to use the lighter-weight [HashSet](http://quiz.geeksforgeeks.org/hashset-in-java/) and [HashMap](https://www.geeksforgeeks.org/hashmap-treemap-java/) instead.

**Collection interview Question 17. What are differences between** [**HashMap and ConcurrentHashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html) **in java?**

**Answer**. Take my words java developers won’t be able to get away from this very important collection framework interview question.

***Differences*** *between java.util.*[***HashMap***](http://www.javamadesoeasy.com/2015/04/hashmap-in-java.html) *and java.util.concurrent.*[*ConcurrentHashMap*](http://www.javamadesoeasy.com/2015/04/concurrenthashmap-in-java.html) *in java >*

|  |  |  |
| --- | --- | --- |
| Property | *java.util.****HashMap*** | *java.util.concurrent.* ***ConcurrentHashMap*** |
| synchronization | HashMap is **not synchronized.** | ConcurrentHashMap is **synchronized**. |
| 2 threads on same Map object can access it at concurrently? | Yes, because HashMap is not synchronized**.** | Yes.  But how despite of being synchronized, 2 threads on same *ConcurrentHashMap* object can access it at same time?  *ConcurrentHashMap* is divided into different **segments** based on concurrency level. So different threads can access different **segments** concurrently. |
| Performance | We will **synchronize HashMap and then compare its performance with ConcurrentHashMap**.  *We can synchronize hashMap by using Collections’s class* ***synchronizedMap*** *method.*   |  | | --- | | *Map synchronizedMap = Collections.****synchronizedMap****(hashMap);* |   *Now, no 2 threads can access same instance of map concurrently.*  **Hence synchronized HashMap’s performance is slower as compared to ConcurrentHashMap.**  But why we didn’t compared HashMap (unSynchronized) with ConcurrentHashMap?  Because performance of unSynchronized collection is always better than some synchronized collection. As, default (unSynchronized) hashMap didn’t cause any locking. | **ConcurrentHashMap’s performance is faster as compared to HashMap (**because it is divided into segments, as discussed in above point**).**  [*Read this post for performance comparison between HashMap and ConcurrentHashMap.*](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html) |
| Null keys and values | HashMap allows to store **one null key** and **many null values** i.e. any key can have null value. | ConcurrentHashMap does **not allow to store null key or null value**.  Any attempt to store null key or value throws runtimeException (NullPointerException). |
| iterators | The iterators returned by the iterator() method of HashMap are [***fail-fast***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html) *>*  *hashMap.keySet().iterator()*  *hashMap.values().iterator()*  *hashMap.entrySet().iterator()*  all three iterators are ***fail-fast*** | iterators are [***fail-safe***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)*.*  *concurrentHashMap.keySet().iterator()*  *concurrentHashMap.values().iterator()*  *concurrentHashMap.entrySet().iterator()*  all three iterators are ***fail-safe.*** |
| **putIfAbsent** | HashMap does not contain putIfAbsent method.  ***putIfAbsent*** *method is equivalent to writing following code >*   |  | | --- | | **synchronized** (map){  **if** (!*map*.containsKey(key))  **return** *map*.put(key, value);  **else**  **return** *map*.get(key);  } |   [**Program to create method that provides functionality similar to putIfAbsent method of ConcurrentHashMap and to be used with HashMap**](http://www.javamadesoeasy.com/2015/04/program-to-create-method-that-provides.html) | If map does not contain specified **key**, put specified **key-value** pair in map and return null.  If map already contains specified **key**, return value corresponding to specified **key**.    [**Program to use ConcurrentHashMap’s putIfAbsent method**](http://www.javamadesoeasy.com/2015/04/program-to-use-concurrenthashmaps.html) |
| Introduced  in which java version | HashMap was introduced in **java 2 i.e. JDK 1.2**, | ConcurrentHashMap was introduced in **java 5** i.e. **JDK 1.5**, since its introduction Hashtable has become obsolete, because of concurrency level its performance is better than Hashtable. |
| Implements which interface | HashMap implements **java.util.**[**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) | ConcurrentHashMap implements  **java.util.Map** and  **java.util.concurrent.ConcurrentMap** |
| Package | HashMap is in **java.util** package | ConcurrentHashMap is in **java.util.concurrent** package. |

For more detail read : [**HashMap and ConcurrentHashMap in java**](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html)

**Collection interview Question 18. When to use** [**HashMap vs Hashtable vs LinkedHashMap vs TreeMap**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html) **in java?**

**Answer**. Another important collection framework interview question

to differentiate between **following Map implementations** in java.

***Differences*** *between java.util.*[***HashMap***](http://www.javamadesoeasy.com/2015/04/hashmap-in-java.html) *vs java.util.*[***Hashtable***](http://www.javamadesoeasy.com/2015/04/hashmap-and-hashtable-similarity-and.html)*vs java.util.****LinkedHashMap*** *vs java.util.*[***TreeMap***](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html) ***>***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Property** | ***HashMap*** | ***Hashtable*** | ***LinkedHashMap*** | ***TreeMap*** |
| 1 | Insertion order | HashMap does not maintains insertion order in java. | Hashtable does not maintains insertion order in java. | LinkedHashMap  maintains insertion order in java. | TreeMap is sorted by natural order of keys in java. |
| 2 | Performance | HashMap is not synchronized, hence its operations are **faster** as compared to Hashtable. | Hashtable is synchronized, hence its operations are **slower** as compared HashMap.  If we are working not working in multithreading environment jdk recommends us to use HashMap. | LinkedHashMap must be used only when we want to maintain insertion order. **Time and space overhead** is there because for maintaining order it internally uses **Doubly Linked list**. | TreeMap must be used only when we want sorting based on natural order. Otherwise sorting operations cost performance. (Comparator is called for sorting purpose) |
| 3 | Null keys and values | HashMap allows to store **one null key** and **many null values** i.e. many keys can have null value in java. | Hashtable does **not allow to store null key or null value**.  Any attempt to store null key or value throws runtimeException (NullPointerException) in java. | LinkedHashMap allows to store **one null key** and **many null values** i.e. any key can have null value in java. | TreeMap does **not allow to store null key but allow many null values**.  Any attempt to store null key throws runtimeException (NullPointerException) in java. |
| 4 | Implements which interface | HashMap implements **java.util.**[**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) | Hashtable implements **java.util.Map** | LinkedHashMap implements **java.util.Map** | TreeMap implements  **java.util.Map**  **java.util.SortedMap**  **java.util.NavigableMap** |
| 5 | Implementation uses? | HashMap use [**buckets**](http://javamadesoeasy.com/2015/02/hashmap-custom-implementation.html) | Hashtable use **buckets** | LinkedHashMap uses [**doubly linked lists**](http://www.javamadesoeasy.com/2015/02/linkedhashmap-custom-implementation.html) | TreeMap uses **Red black tree** |
| 6 | Complexity of put, get and remove methods | O(1) | O(1) | O(1)  **overhead** of updating **Doubly Linked list** for maintaining order it internally uses. | O(log(n)) |
| 7 | Extends java.util.**Dictionary** (Abstract class, which is obsolete) | HashMap **doesn’t** extends Dictionary. | Hashtable **extends** Dictionary (which maps non-null keys to values. In a given Dictionary we can look up value corresponding to key) | LinkedHashMap **doesn’t** extends Dictionary. | TreeMap **doesn’t** extends Dictionary. |
| 8 | Introduced in which java version? | HashMap was introduced in second version of java i.e. **JDK 2.0** | Hashtable was introduced in first version of java i.e. **JDK 1.0**  But it was refactored in java 2 i.e. JDK 1.2 to implement the Map interface, hence making it a member of member of the [Java Collections Framework](http://download.oracle.com/javase/7/docs/technotes/guides/collections/index.html). | LinkedHashMap was introduced in fourth version of java i.e. **JDK 4.0** | TreeMap was introduced in second version of java i.e. **JDK 2.0** |

For more detail read : [**HashMap vs Hashtable vs LinkedHashMap vs TreeMap in java**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html)

**Collection interview Question 19. What are differences between** [**HashMap vs IdentityHashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-identityhashmap-similarity.html) **in java?**

**Answer**. This is tricky and complex collection framework interview question for experienced developers in java.

***Differences*** *between java.util.*[***HashMap***](http://www.javamadesoeasy.com/2015/04/hashmap-in-java.html)*and java.util.*[***IdentityHashMap***](http://www.javamadesoeasy.com/2015/04/identityhashmap-in-java.html) *in java**>*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.HashMap*** | ***java.util.IdentityHashMap*** |
| 1 | **Keys comparison***object-equality  vs reference-equality* | **HashMap** when comparing keys (and values) performs object-equality not reference-equality. In an HashMap, two keys k1 and k2 are equal if and only if (k1==null ? k2==null : k1.equals(k2)) | **IdentityHashMap** when comparing keys (and values) performs reference-equality in place of object-equality. In an IdentityHashMap, two keys k1 and k2 are equal if and only if (k1==k2) |
| 2 | Initial size | Constructs a new HashMap, Its initial capacity is 16 in java.   |  | | --- | | **new** HashMap(); | | Constructs a new IdentityHashMap, with maximum size of 21 in java.   |  | | --- | | **new** IdentityHashMap(); | |
| 3 | Introduced in which java version | HashMap was introduced in second version of java i.e. **JDK 2.0** | IdentityHashMap was introduced in fourth version of java i.e. **JDK 4.0** |
| 4 | *Program* | Program 1 shows > *comparing keys (and values) performs object-equality in place of reference-equality . In an HashMap, two keys k1 and k2 are equal if and only if* **(k1==null ? k2==null : k1.equals(k2)).** | Program 2 shows >  *comparing keys (and values) performs reference-equality in place of object-equality. In an IdentityHashMap, two keys k1 and k2 are equal if and only if* **(k1==k2).** |
| 5 | overridden equals() and hashCode() method call? | [*overridden equals() and hashCode() method*](http://www.javamadesoeasy.com/2015/02/override-equals-and-hashcode-method.html)are called when put, get methods are called in ***HashMap***.  As shown in Program 3. | *overridden equals() and hashCode() method* are not called when put, get methods are called in ***IdentityHashMap***.  *Because IdentityHashMap implements equals() and hashCode() method by itself and checks for reference-equality of keys.*  As shown in Program 4. |
| 6 | Application - can maintain *proxy object* | HashMap cannot be used to maintain *proxy object.* | IdentityHashMap can be used to maintain *proxy objects*. For example, we might need to maintain proxy object for each object debugged in the program. |

For more detail read : [**HashMap vs IdentityHashMap - Similarity and Differences with program in java**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-identityhashmap-similarity.html)

**Collection interview Question 32. What is difference between** [**Comparable and Comparator**](http://www.javamadesoeasy.com/2015/04/comparable-vs-comparator-differences.html)**? How can you sort List?**

**Answer**.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***Comparable*** | ***Comparator*** |
| 1 | Comparing instances of class | Comparable is used to compare instances of same class | Comparator can be used to compare instances of same or different classes. |
| **2** | **sorting order** | Comparable can be implemented by class which need to define a **natural ordering for its objects.**  **Example** - String, Integer, Long , [Date](http://www.javamadesoeasy.com/2015/07/creating-date-in-java-using-calendar.html) and all other wrapper classes implements Comparable. | Comparator is implemented when one wants a **different sorting order** and define custom way of comparing two instances. |
| 3 | Changes to class | For using Comparable, original Class must implement it.    **Example-**  **class** Employee **implements Comparable<Employee>**    For using Comparable, Employee Class must implement it, no other class can implement it.  As used in **Program 1** | Class itself can implement Comparator  or  any other class can implement Comparator. Hence avoiding modification to original class.  **Example-**  **class ComparatorName implements Comparator<Employee>**  **class ComparatorId implements Comparator<Employee>**  In above example modifications were made to **ComparatorName** and **ComparatorId.** Hence avoiding modification to Employee class.  As used in **Program 4** |
| 4 | Sorting on basis on one or many criteria | Provides sorting only on **one** criteria, **because** Comparable can be implemented by original class only. | We can use Comparator to sort class on **many** criterias **because** class itself or any other class can implement Comparator. |
| 5 | Method | compareTo method  **@Override**  **public int compareTo(Employee obj) {**  **//sort Employee on basis of name(ascending order)**  **return this.name.compareTo(obj.name);**  **}**  Method compares **this** with **obj** object and returns a integer.   * positive – **this** is **greater** than **obj** * zero – **this** is **equal** to **obj** * negative – **this** is **less** than **obj**     As used in **Program 1** | compare method  **@Override**  **public int compare(Employee obj1, Employee obj2) {**  **//sort Employee on basis of name(ascending order)**  **return obj1.name.compareTo(obj2.name);**  **}**    Method compares **obj1** with **obj2** object and returns a integer.   * positive – **obj1** is **greater** than **obj2** * zero – **obj1** is **equal** to **obj2** * negative – **obj1** is **less** than **obj2**     As used in **Program 3** |
| 6 | Package | **java.lang**  **java.lang** package is automatically imported by every program in java.  Hence, we need to write explicit statement for importing java.lang.Comparable. | **java.util**  We need to write explicit import statement -  **import** java.util.Comparator |
| 7 | Using **Collections.sort** | Let's say we wanna sort list of Employee,  **Collections.sort(**list**)** uses Comparable interface for sorting class.  As used in Program 1 | Let's say we wanna sort list of Employee,  **Collections.*sort*(list,new ComparatorName());**  uses Comparator interface for sorting class.  As used in Program 5 |

## 1. Employee – Model Class

Our Employee class is plain old class with four fields: id, firstName, lastName and age. I have chosen these fields purposefully.

|  |
| --- |
| Employee.java |
| package corejava.compare;    public class Employee {      private int id = -1;      private String firstName = null;      private String lastName = null;      private int age = -1;        public Employee(int id, String fName, String lName, int age) {          this.id = id;          this.firstName = fName;          this.lastName = lName;          this.age = age;      }        // Setters and Getters        @Override      public String toString() {          return "Employee : " + id + " - " + firstName + " - " + lastName          + " - " + age + "n";      }  } |

## 2. Java sort arraylist of objects – Comparable Example

Comparable interface provides one method [compareTo(T o)](https://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html#compareTo%28T%29) to implement in any class so that two instances of that class can be compared.

Method syntax is:

|  |
| --- |
| public int compareTo(T o); |

Here, out of two instances to compare, one is instance itself on which method will be invoked, and other is passed as parameter *o*.

Lets see how our Employee class will look after implementing Comparable interface.

|  |
| --- |
| Employee.java |
| package corejava.compare;    public class Employee implements Comparable<Employee> {        private int id = -1;      private String firstName = null;      private String lastName = null;      private int age = -1;        public Employee(int id, String fName, String lName, int age) {          this.id = id;          this.firstName = fName;          this.lastName = lName;          this.age = age;      }        @Override      public int compareTo(Employee o) {          return this.id - o.id;      }        // Setters and Getters        @Override      public String toString() {          return "Employee : " + id + " - " + firstName + " - " + lastName + " - " + age + "\n";      }  } |

Default way to sort list of employees, in our case, is by their id. Whatever, your default sorting order is, use in compare() method.

In implemented compare() method, we have simply returned the difference in employee ids of two instances. Two equal employee ids will return zero, indicating the same object.

#### 2.1. Collections.sort()

Lets test our compare() method implemented in above Employee class.

|  |
| --- |
| package corejava.compare;    import java.util.ArrayList;  import java.util.Collections;  import java.util.List;    public class TestSorting {      public static void main(String[] args) {          Employee e1 = new Employee(1, "aTestName", "dLastName", 34);          Employee e2 = new Employee(2, "nTestName", "pLastName", 30);          Employee e3 = new Employee(3, "kTestName", "sLastName", 31);          Employee e4 = new Employee(4, "dTestName", "zLastName", 25);            List<Employee> employees = new ArrayList<Employee>();          employees.add(e2);          employees.add(e3);          employees.add(e1);          employees.add(e4);            // UnSorted List          System.out.println(employees);            Collections.sort(employees);            // Default Sorting by employee id          System.out.println(employees);      }  } |

In the above program, first print statement prints an unsorted list of employees and in the second print statement, employees are sorted by their employee id.

|  |
| --- |
| Console |
| //Unsorted    [Employee : 2 - nTestName - pLastName - 30  , Employee : 3 - kTestName - sLastName - 31  , Employee : 1 - aTestName - dLastName - 34  , Employee : 4 - dTestName - zLastName - 25]    //Sorted    [Employee : 1 - aTestName - dLastName - 34  , Employee : 2 - nTestName - pLastName - 30  , Employee : 3 - kTestName - sLastName - 31  , Employee : 4 - dTestName - zLastName - 25] |

## 3. Java sort arraylist of objects – Comparator Example

So, now we can sort a list of employees by their id. Now let’s consider a case where we want to sort employees list based on some user input which is essentially sorting field i.e. sometimes he wants to sort by first name, sometimes sort by age also.

This can be achieved by [jquery](https://jquery.com/) plugins easily, but what if browser has disables the javascript. You will have to sort the list on the server side only to not break the application functionality.

Here comes the Comparator interface to rescue you. A Comparator can be used to sort a collection of instances on some particular basis. **To sort of different fields, we need multiple Comparator implementations.**

#### 3.1. Comparator interface example

We have determined multiple implementations for different sorting cases. Let’s write them.

##### 3.1.1. First name sorter

|  |
| --- |
| FirstNameSorter.java |
| package corejava.compare;    import java.util.Comparator;    public class FirstNameSorter implements Comparator<Employee>{    @Override  public int compare(Employee o1, Employee o2) {  return o1.getFirstName().compareTo(o2.getFirstName());  }  } |

##### 3.1.2. Last name sorter

|  |
| --- |
| LastNameSorter.java |
| package corejava.compare;    import java.util.Comparator;    public class LastNameSorter implements Comparator<Employee> {        @Override      public int compare(Employee o1, Employee o2) {          return o1.getLastName().compareTo(o2.getLastName());      }    } |

##### 3.1.3. Age sorter

|  |
| --- |
| AgeSorter.java |
| package corejava.compare;    import java.util.Comparator;    public class AgeSorter implements Comparator<Employee> {      @Override      public int compare(Employee o1, Employee o2) {          return o1.getAge() - o2.getAge();      }  } |

#### 3.2. How to compare with Comparator

So theoretically, we should be able to sort of any field at our wish with minimum code. Let’s see if we really are:

|  |
| --- |
| package corejava.compare;    import java.util.ArrayList;  import java.util.Collections;  import java.util.List;    public class TestSorting  {      public static void main(String[] args)      {          Employee e1 = new Employee(1, "aTestName", "dLastName", 34);          Employee e2 = new Employee(2, "nTestName", "pLastName", 30);          Employee e3 = new Employee(3, "kTestName", "sLastName", 31);          Employee e4 = new Employee(4, "dTestName", "zLastName", 25);            List<Employee> employees = new ArrayList<Employee>();          employees.add(e2);          employees.add(e3);          employees.add(e1);          employees.add(e4);            // UnSorted List          System.out.println(employees);            Collections.sort(employees);            // Default Sorting by employee id          System.out.println(employees);            Collections.sort(employees, new FirstNameSorter());            // Sorted by firstName          System.out.println(employees);            Collections.sort(employees, new LastNameSorter());            // Sorted by lastName          System.out.println(employees);            Collections.sort(employees, new AgeSorter());            // Sorted by age          System.out.println(employees);      }  }    Output:  //Unsorted    [Employee : 2 - nTestName - pLastName - 30  , Employee : 3 - kTestName - sLastName - 31  , Employee : 1 - aTestName - dLastName - 34  , Employee : 4 - dTestName - zLastName - 25]    //Default sorting based on employee id    [Employee : 1 - aTestName - dLastName - 34  , Employee : 2 - nTestName - pLastName - 30  , Employee : 3 - kTestName - sLastName - 31  , Employee : 4 - dTestName - zLastName - 25]    //Sorted by first name    [Employee : 1 - aTestName - dLastName - 34  , Employee : 4 - dTestName - zLastName - 25  , Employee : 3 - kTestName - sLastName - 31  , Employee : 2 - nTestName - pLastName - 30]    //Sorted by last name    [Employee : 1 - aTestName - dLastName - 34  , Employee : 2 - nTestName - pLastName - 30  , Employee : 3 - kTestName - sLastName - 31  , Employee : 4 - dTestName - zLastName - 25]    //Sorted by age    [Employee : 4 - dTestName - zLastName - 25  , Employee : 2 - nTestName - pLastName - 30  , Employee : 3 - kTestName - sLastName - 31  , Employee : 1 - aTestName - dLastName - 34] |

Above class’s output shows clearly, now we are able to sort the list of employees on any field using appropriate Comparator implementation.

## 4. Comparator in Java 8

Latest Lambda changes have made using Comparator a lot easier than ever before.

|  |
| --- |
| Comparator with Lambda |
| List<Employee> employees  = getEmployeesFromDB();    //Sort all employees by first name  employees.sort(Comparator.comparing(e -> e.getFirstName()));    //OR you can use below  employees.sort(Comparator.comparing(Employee::getFirstName));    //Sort all employees by first name in reverse order  Comparator<Employee> comparator = Comparator.comparing(e -> e.getFirstName());  employees.sort(comparator.reversed());    //Sorting on multiple fields; Group by.  Comparator<Employee> groupByComparator = Comparator.comparing(Employee::getFirstName).thenComparing(Employee::getLastName);  employees.sort(groupByComparator); |

Read More: [Java 8 Comparator Example](https://howtodoinjava.com/java8/using-comparator-becomes-easier-with-lambda-expressions-java-8/)

## 5. Java sort arraylist of objects – SortedSet and SortedMap

Its important to understand the importance of both interfaces in code collection APIs which provide implicit behavior **sort list of objects by property**. Such APIs are for example [SortedMap](https://docs.oracle.com/javase/6/docs/api/java/util/SortedMap.html) or [SortedSet](https://docs.oracle.com/javase/6/docs/api/java/util/SortedSet.html).

These collections store elements in sorted order. To determining the sorting, they also use compare()method. Always remember if you Employee class does not implement *Comparable* interface and you are not using *Comparator* also, then adding elements in sorted collections will give you error.

|  |
| --- |
| Adding non-comparable items in SortedSet |
| Exception in thread "main" java.lang.ClassCastException: corejava.compare.Employee cannot be cast to java.lang.Comparable  at java.util.TreeMap.put(Unknown Source)  at java.util.TreeSet.add(Unknown Source)  at corejava.compare.SortedSetTest.main(SortedSetTest.java:17) |

So, it is necessary to use either of both interfaces to able to store instances in sorted collections.

#### 5.1. SortedSet with Comparable

If you are not using Comparator implementation, then Employee class’s compare() method will be used for sorting.

|  |
| --- |
| SortedSetTest.java |
| package corejava.compare;    import java.util.SortedSet;  import java.util.TreeSet;    public class SortedSetTest  {      public static void main(String[] args)      {          SortedSet<Employee> set = new TreeSet<Employee>();            Employee e1 = new Employee(1, "aTestName", "dLastName", 34);          Employee e2 = new Employee(2, "nTestName", "pLastName", 30);          Employee e3 = new Employee(3, "kTestName", "sLastName", 31);          Employee e4 = new Employee(4, "dTestName", "zLastName", 25);            set.add(e2);          set.add(e3);          set.add(e1);          set.add(e4);            System.out.println(set);      }  } |

Program output:

|  |
| --- |
| Console |
| [Employee : 1 - aTestName - dLastName - 34  , Employee : 2 - nTestName - pLastName - 30  , Employee : 3 - kTestName - sLastName - 31  , Employee : 4 - dTestName - zLastName - 25] |

#### 5.1. SortedSet with Comparator

If you want to add some other sorting behavior using Comparator implementation, you can pass the Comparator to constructor of sorted collection. e.g.

|  |
| --- |
| SortedSetTest.java |
| package corejava.compare;    import java.util.SortedSet;  import java.util.TreeSet;    public class SortedSetTest {      public static void main(String[] args)      {          SortedSet<Employee> set = new TreeSet<Employee>(new FirstNameSorter());            Employee e1 = new Employee(1, "aTestName", "dLastName", 34);          Employee e2 = new Employee(2, "nTestName", "pLastName", 30);          Employee e3 = new Employee(3, "kTestName", "sLastName", 31);          Employee e4 = new Employee(4, "dTestName", "zLastName", 25);            set.add(e2);          set.add(e3);          set.add(e1);          set.add(e4);            System.out.println(set);      }  } |

Program output:

|  |
| --- |
| Console |
| [Employee : 1 - aTestName - dLastName - 34  , Employee : 2 - nTestName - pLastName - 30  , Employee : 3 - kTestName - sLastName - 31  , Employee : 4 - dTestName - zLastName - 25] |

**Collection interview Question 38. What are differences between** [**ArrayList vs CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html)**?**

**Answer**.

***Differences*** *between java.util.*[***ArrayList***](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html)*and java.util.concurrent.****CopyOnWriteArrayList****in java >*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.ArrayList*** | **java.util.concurrent. CopyOnWriteArrayList** |
| 1 | synchronization | ArrayList is not **synchronized**  (because 2 threads on same ArrayList object can access it at same time).  I have created **program** to show see consequence of using ArrayList in multithreading environment.  In the program i will implement our own arrayList. | **CopyOnWriteArrayList**is **synchronized**  (because 2 threads on same CopyOnWriteArrayList object cannot access it at same time). |
| 2 | Iterator and listIterator | Iterator and listIterator returned by ArrayList are [**Fail-fast**](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)**,** means any structural modification made to ArrayList during iteration using Iterator or listIterator will throw ConcurrentModificationException in java.  As shown in Program 1 below. | Iterator and listIterator returned by CopyOnWriteArrayList are **Fail-safe** in java.  As shown in Program 2 below. |
| 3 | Enumeration is fail-fast | **Enumeration** returned by ArrayList is **fail-fast**, means any structural modification made to ArrayList during iteration using Enumeration will throw ConcurrentModificationException.  As shown in Program 1 below. | **Enumeration** returned by CopyOnWriteArrayList is **fail-safe.**    As shown in Program 2 below. |
| 4 | Iterate using **enhanced for loop** | Iteration done on ArrayList using **enhanced for loop** is **Fail-fast,** means any structural modification made to ArrayList during iteration using **enhanced for loop** will throw ConcurrentModificationException.  As shown in Program 1 below. | Iteration done on CopyOnWriteArrayList using **enhanced for loop** is **Fail-safe.**  As shown in Program 2 below. |
| 5 | Performance | ArrayList is not synchronized, hence its operations are **faster** as compared to CopyOnWriteArrayList. | CopyOnWriteArrayList is synchronized, hence its operations are **slower** as compared to ArrayList. |
| 6 | AbstractList | ArrayList extends AbstractList (abstract class) which provides implementation to  List interface to minimize the effort required to implement this interface backed by RandomAccess interface. | CopyOnWriteArrayList does not extends AbstractList, though CopyOnWriteArrayList also implements RandomAccess interface. |
| 7 | Introduced in which java version | ArrayList was introduced in second version of java (1.2) i.e. **JDK 2.0** | CopyOnWriteArrayList was introduced in fifth version of java (1.5) i.e. **JDK 5.0** |
| 8 | Package | java.util | java.util.**concurrent** |

**For more detail read :**[ArrayList vs CopyOnWriteArrayList - Similarity and Differences with program](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html)

**Collection interview Question 39. What are differences between** [**HashSet vs CopyOnWriteArraySet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html)**?**

**Answer**.

***Differences*** *between java.util.*[***HashSet***](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)*and java.util.concurrent.****CopyOnWriteArraySet*** *in java >*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.HashSet*** | **java.util.concurrent. CopyOnWriteArraySet** |
| 1 | synchronization | HashSet is not **synchronized**  (because 2 threads on same HashSet object can access it at same time) in java. | **CopyOnWriteArraySet**is **synchronized**  (because 2 threads on same CopyOnWriteArraySet object cannot access it at same time) in java. |
| 2 | Iterator | Iterator returned by HashSet is [**Fail-fast**](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)**,** means any structural modification made to HashSet during iteration using Iterator will throw ConcurrentModificationException in java.  As shown in Program 1 below. | Iterator returned by **CopyOnWriteArraySet** is **Fail-safe** in java.  As shown in Program 2 below. |
| 3 | Enumeration is fail-fast | **Enumeration** returned by HashSet is **fail-fast**, means any structural modification made to HashSet during iteration using Enumeration will throw ConcurrentModificationException.  As shown in Program 1 below. | **Enumeration** returned by CopyOnWriteArraySet is **fail-safe.**  As shown in Program 2 below. |
| 4 | Iterate using **enhanced for loop** | Iteration done on HashSet using **enhanced for loop** is **Fail-fast,** means any structural modification made to HashSet during iteration using **enhanced for loop** will throw ConcurrentModificationException.  As shown in Program 1 below. | Iteration done on CopyOnWriteArraySet using **enhanced for loop** is **Fail-safe.**    As shown in Program 2 below. |
| 5 | Performance | HashSet is not synchronized, hence its operations are **faster** as compared to CopyOnWriteArraySet. | CopyOnWriteArraySet is synchronized, hence its operations are **slower** as compared to HashSet. |
| 6 | Introduced in which java version | HashSet was introduced in second version of java (1.2) i.e. **JDK 2.0** | CopyOnWriteArraySet  was introduced in fifth version of java (1.5) i.e. **JDK 5.0** |
| 7 | Package | java.util | java.util.**concurrent** |

### *For more detail read :* [HashSet vs CopyOnWriteArraySet - Similarity and Differences with program](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html)

**Collection interview Question 40. What are differences between** [**TreeSet vs ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html)**?**

**Answer**.

***Differences*** *between java.util.*[***TreeSet***](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html) *and java.util.concurrent.****ConcurrentSkipListSet*** *in java**>*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.TreeSet*** | **java.util.concurrent. ConcurrentSkipListSet** |
| 1 | synchronization | TreeSet is not **synchronized**  (because 2 threads on same TreeSet object can access it at same time) in java. | **ConcurrentSkipListSet**is **synchronized**  (because 2 threads on same ConcurrentSkipListSet object cannot access it at same time) in java. |
| 2 | Iterator | Iterator returned by TreeSet is [**Fail-fast**](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)**,** means any structural modification made to TreeSet during iteration using Iterator will throw ConcurrentModificationException in java.  As shown in Program 1 below. | Iterator returned by **ConcurrentSkipListSet** is **Fail-safe** in java.    As shown in Program 2 below. |
| 3 | Enumeration is fail-fast | **Enumeration** returned by TreeSet is **fail-fast**, means any structural modification made to TreeSet during iteration using Enumeration will throw ConcurrentModificationException.  As shown in Program 1 below. | **Enumeration** returned by ConcurrentSkipListSet is **fail-safe.**    As shown in Program 2 below. |
| 4 | Iterate using **enhanced for loop** | Iteration done on TreeSet using **enhanced for loop** is **Fail-fast,** means any structural modification made to TreeSet during iteration using **enhanced for loop** will throw ConcurrentModificationException.  As shown in Program 1 below. | Iteration done on ConcurrentSkipListSet using **enhanced for loop** is **Fail-safe.**  As shown in Program 2 below. |
| 5 | Performance | TreeSet is not synchronized, hence its operations are **faster** as compared to ConcurrentSkipListSet. | ConcurrentSkipListSet is synchronized, hence its operations are **slower** as compared to TreeSet. |
| 6 | Introduced in which java version | TreeSet was introduced in second version of java (1.2) i.e. **JDK 2.0** | ConcurrentSkipListSet was introduced in sixth version of java (1.6) i.e. **JDK 6.0** |
| 7 | Package | java.util | java.util.**concurrent** |

### *For more detail read :* [TreeSet vs ConcurrentSkipListSet - Similarity and Differences with program](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html)

**Collection interview Question 41. What are differences between** [**TreeMap vs ConcurrentSkipListMap**](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html)**?**

**Answer**.

***Differences*** *between java.util.*[***TreeMap***](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html)*and java.util.concurrent.****ConcurrentSkipListMap*** *in java >*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.util.TreeMap*** | ***java.util.concurrent. ConcurrentSkipListMap*** |
| 1 | synchronization | TreeMap is **not synchronized**  (because 2 threads on same TreeMap object can access it at same time) in java. | ConcurrentSkipListMap is **synchronized** (because 2 threads on same ConcurrentSkipListMap object cannot access it at same time) in java. |
| 2 | Iterator | The iterators returned by the iterator() method of Map's “collection view methods" are [***fail-fast***](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html)*>*   * *map.keySet().iterator()* * *map.values().iterator()* * *map.entrySet().iterator()*   all three iterators are ***fail-fast*,** means any structural modification made to TreeMap during iteration using any of 3 Iterator will throw ConcurrentModificationException.  As shown in Program 1 below. | The iterators returned by the iterator() method of Map's “collection view methods" are *fail-safe >*   * *map.keySet().iterator()* * *map.values().iterator()* * *map.entrySet().iterator()*   all three iterators are ***fail-safe.***    As shown in Program 2 below. |
| 3 | Performance | TreeMap is not synchronized, hence its operations are **faster** as compared to ConcurrentSkipListMap. | ConcurrentSkipListMap is synchronized, hence its operations are **slower** as compared to TreeMap. |
| 4 | Introduced inin which java version | TreeMap was introduced in second version of java i.e. **JDK 2.0** | ConcurrentSkipListMap was introduced in sixth version of java i.e. **JDK 6.0** |
| 5 | Package | java.util | java.util.**concurrent** |
| 6 | Implements which interface | Map  SortedMap  NavigableMap | Map  SortedMap  NavigableMap  ConcurrentNavigableMap |

### *For more detail read :* [TreeMap vs ConcurrentSkipListMap - Similarity and Differences with program](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html)

**Collection interview Question 43. Can we use null element in TreeSet? Give reason?**

**Answer**. No, [TreeSet](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html) does **not** allows to store **any null keys**.

Any attempt to add null throws runtimeException (NullPointerException).

TreeSet internally compares elements for sorting elements by natural order ([comparator may be used for sorting](http://www.javamadesoeasy.com/2015/04/program-to-sort-set-in-ascending-order_24.html), if defined at creation time) and null is not comparable, Any attempt to compare null with other object will throw NullPointerException.

**Collection interview Question 44. Can we use null key in TreeMap? Give reason?**

**Answer**. No, [TreeMap](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html) **not allow to store null key.**

Any attempt to store null key throws runtimeException (NullPointerException).

TreeMap internally compares keys for sorting keys by natural order ([comparator may be used for sorting](http://www.javamadesoeasy.com/2015/04/program-to-sort-set-in-ascending-order_24.html), if defined at creation time)  and null is not comparable, Any attempt to compare null with other object will throw NullPointerException.

**Collection interview Question 45.  How ConcurrentHashMap works? Can 2 threads on same ConcurrentHashMap object access it concurrently?**

**Answer**. [*ConcurrentHashMap*](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html)is divided into different **segments** based on concurrency level. So different threads can access different **segments** concurrently.

**Can threads read the segment locked by some other thread?**

Yes. When thread locks one segment for updation it does not block it for retrieval (done by get method) hence some other thread can read the segment (by get method), but it will be able to read the data before locking.

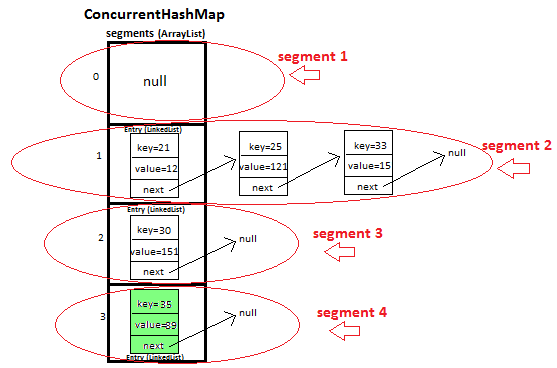
For operations such as putAll concurrent retrievals may reflect removal of only some entries.

For operations such as clear concurrent retrievals may reflect removal of only some entries.

***Segments*** *in ConcurrentHashMap with* ***diagram*** *>*

we have ConcurrentHashMap with **4 segments -**

(Diagram shows how **segments** are formed in ConcurrentHashMap)

**[](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html)**

**Collection interview Question 46. Write a program to show consequence of using ArrayList in multithreading environment?**

**Answer.** Program to show [consequence of using ArrayList in multithreading environment in java](http://www.javamadesoeasy.com/2015/05/consequence-of-using-arraylist-in.html)

[dri](https://docs.google.com/document/d/10tUFPo2Xcw5JiwNR_bqIGEw52V6eUGP0KvE2nTHkb-U/edit) [blog](https://www.blogger.com/blogger.g?blogID=5056459490283781613#editor/target=post;postID=4105800325043770563;onPublishedMenu=allposts;onClosedMenu=allposts;postNum=0;src=link)

**Collection interview Question 47. Write a program to show advantage of using Vector in multithreading environment?**

**Answer.**  Program to show [advantage of using Vector in multithreading environment in java](http://www.javamadesoeasy.com/2015/05/advantage-of-using-vector-in.html)

**Collection interview Question 48. Mention properties of most frequently used Collection classes and Interfaces? Mention as many properties as much you can.**

**Answer**. This question is real test for experienced developers, this will test your in depth awareness of Collection classes and Interfaces. Answering this question in detail will really ensure your selection.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| [**List**](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html) | **Duplicate elements** | **insertion order** | **Sorted by natural order** | **synchronized** | **null elements** | **Iterator** |
| [**ArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html) | Yes | Yes |  |  | Yes | Iterator & listIterator  are  [Fail-fast](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html) |
| [**LinkedList**](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html) | Yes | Yes |  |  | Yes | Iterator & listIterator  are  Fail-fast |
| [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html) | Yes | Yes |  | Yes | Yes | Iterator & listIterator  are  [**Fail-safe**](http://www.javamadesoeasy.com/2015/04/concurrentmodificationexception-fail.html) |
|  |  |  |  |  |  |  |
| [**Set**](http://www.javamadesoeasy.com/2015/04/set-hierarchy-in-java-detailed-hashset.html) | **Duplicate elements** | **insertion order** | **Sorted by natural order** | **synchronized** | **null elements** | **Iterator** |
| [**HashSet**](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html) |  |  |  |  | Yes | Fail-fast |
| [**LinkedHashSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html) |  | Yes |  |  | Yes | Fail-fast |
| [**TreeSet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-linkedhashset-vs-treeset.html) |  |  | Yes |  | No | Fail-fast |
| [**ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html) |  |  | Yes | Yes | No | **Fail-safe** |
|  |  |  |  |  |  |  |
| [**Map**](http://www.javamadesoeasy.com/2015/04/map-hierarchy-in-java-detailed-hashmap.html) | **Duplicate Keys** | **insertion order of keys** | **Sorted by natural order of keys** | **synchronized** | **null keys or null values** | **Iterator**  **Map implementations returns 3 iterators >**  *map.keySet().iterator()*  *map.values().iterator()*  *map.entrySet().iterator()* |
| [**HashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-in-java.html) |  |  |  |  | one null key and many null values | All are Fail-fast |
| [**Hashtable**](http://www.javamadesoeasy.com/2015/04/hashmap-and-hashtable-similarity-and.html) |  |  |  | Yes | No | All are Fail-fast |
| [**ConcurrentHashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html) |  |  |  | Yes | No | All are **Fail-safe** |
| [**TreeMap**](http://www.javamadesoeasy.com/2015/04/hashmap-vs-hashtable-vs-linkedhashmap.html) |  |  | Yes |  | Null key not allowed,  Allow many null values | All are Fail-fast |
| [**ConcurrentSkipListMap**](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html) |  |  | Yes | Yes | No | All are **Fail-safe** |

[**Collection - List, Set and Map all properties in tabular form**](http://www.javamadesoeasy.com/2015/04/collection-list-set-and-map-all.html)

**Collection interview Question. 49 Which list class must be preferred in multithreading environment, considering performance constraint?**

**Answer**. [**CopyOnWriteArrayList**](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html)

**Collection interview Question 50. Which Set class must be preferred in multithreading environment, considering performance constraint?**

**Answer**. [**CopyOnWriteArraySet**](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html)(allows null and elements aren't sorted in natural order) **or** [**ConcurrentSkipListSet**](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html)(doesn’t allows null and elements are sorted in natural order)

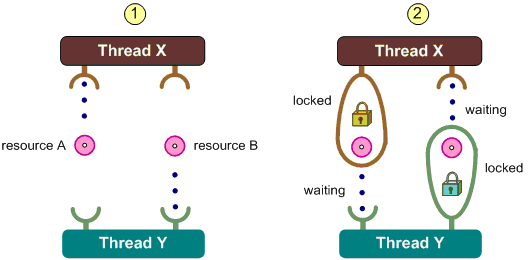
**Select one depending on your requirement.**

**Collection interview Question 51. Which Map class must be preferred in multithreading environment, considering performance constraint?**

**Answer**. [**ConcurrentHashMap**](http://www.javamadesoeasy.com/2015/04/hashmap-and-concurrenthashmap.html)(keys aren't sorted in natural order) **or** [**ConcurrentSkipListMap**](http://www.javamadesoeasy.com/2015/04/treemap-vs-concurrentskiplistmap.html)(keys are sorted in natural order)

**Select one depending on your requirement.**

Deadlock in Java Multithreading

[**synchronized**](http://quiz.geeksforgeeks.org/synchronized-in-java/) keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock.  
It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called [**Deadlock**](http://quiz.geeksforgeeks.org/operating-system-process-management-deadlock-introduction/). Below is a simple example of Deadlock condition.  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/threads_deadlock.gif)  
Image source: https://software.intel.com/en-us/articles/multi-threading-in-the-net-environment

|  |
| --- |
| // Java program to illustrate Deadlock  // in multithreading.  class Util  {      // Util class to sleep a thread      static void sleep(long millis)      {          try          {              Thread.sleep(millis);          }          catch (InterruptedException e)          {              e.printStackTrace();          }      }  }    // This class is shared by both threads  class Shared  {      // first synchronized method      synchronized void test1(Shared s2)      {          System.out.println("test1-begin");          Util.sleep(1000);            // taking object lock of s2 enters          // into test2 method          s2.test2(this);          System.out.println("test1-end");      }        // second synchronized method      synchronized void test2(Shared s1)      {          System.out.println("test2-begin");          Util.sleep(1000);            // taking object lock of s1 enters          // into test1 method          s1.test1(this);          System.out.println("test2-end");      }  }      class Thread1 extends Thread  {      private Shared s1;      private Shared s2;        // constructor to initialize fields      public Thread1(Shared s1, Shared s2)      {          this.s1 = s1;          this.s2 = s2;      }        // run method to start a thread      @Override      public void run()      {          // taking object lock of s1 enters          // into test1 method          s1.test1(s2);      }  }      class Thread2 extends Thread  {      private Shared s1;      private Shared s2;        // constructor to initialize fields      public Thread2(Shared s1, Shared s2)      {          this.s1 = s1;          this.s2 = s2;      }        // run method to start a thread      @Override      public void run()      {          // taking object lock of s2          // enters into test2 method          s2.test2(s1);      }  }      public class GFG  {      public static void main(String[] args)      {          // creating one object          Shared s1 = new Shared();            // creating second object          Shared s2 = new Shared();            // creating first thread and starting it          Thread1 t1 = new Thread1(s1, s2);          t1.start();            // creating second thread and starting it          Thread2 t2 = new Thread2(s1, s2);          t2.start();            // sleeping main thread          Util.sleep(2000);      }  } |

Run on IDE

Output : test1-begin

test2-begin

It is not recommended to run the above program with online IDE. We can copy the source code and run it on our local machine. We can see that it runs for indefinite time, because threads are in deadlock condition and doesn’t let code to execute. Now let’s see step by step what is happening there.

1. Thread t1 starts and calls test1 method by taking the object lock of s1.
2. Thread t2 starts and calls test2 method by taking the object lock of s2.
3. t1 prints test1-begin and t2 prints test-2 begin and both waits for 1 second, so that both threads can be started if any of them is not.
4. t1 tries to take object lock of s2 and call method test2 but as it is already acquired by t2 so it waits till it become free. It will not release lock of s1 until it gets lock of s2.
5. Same happens with t2. It tries to take object lock of s1 and call method test1 but it is already acquired by t1, so it has to wait till t1 release the lock. t2 will also not release lock of s2 until it gets lock of s1.
6. Now, both threads are in wait state, waiting for each other to release locks. Now there is a race around condition that who will release the lock first.
7. As none of them is ready to release lock, so this is the Dead Lock condition.
8. When you will run this program, it will be look like execution is paused.

**Detect Dead Lock condition**

We can also detect deadlock by running this program on cmd. We have to collect Thread Dump. Command to collect depends on OS type. If we are using Windows and Java 8, command is jcmd $PID Thread.print  
We can get PID by running jps command. Thread dump for above program is below:

5524:

2017-04-21 09:57:39

Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.25-b02 mixed mode):

"DestroyJavaVM" #12 prio=5 os\_prio=0 tid=0x0000000002690800 nid=0xba8 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Thread-1" #11 prio=5 os\_prio=0 tid=0x0000000018bbf800 nid=0x12bc waiting for monitor entry [0x000000001937f000]

java.lang.Thread.State: BLOCKED (on object monitor)

at Shared.test1(GFG.java:15)

- waiting to lock (a Shared)

at Shared.test2(GFG.java:29)

- locked (a Shared)

at Thread2.run(GFG.java:68)

"Thread-0" #10 prio=5 os\_prio=0 tid=0x0000000018bbc000 nid=0x1d8 waiting for monitor entry [0x000000001927f000]

java.lang.Thread.State: BLOCKED (on object monitor)

at Shared.test2(GFG.java:25)

- waiting to lock (a Shared)

at Shared.test1(GFG.java:19)

- locked (a Shared)

at Thread1.run(GFG.java:49)

"Service Thread" #9 daemon prio=9 os\_prio=0 tid=0x000000001737d800 nid=0x1680 runnable [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"C1 CompilerThread2" #8 daemon prio=9 os\_prio=2 tid=0x000000001732b800 nid=0x17b0 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #7 daemon prio=9 os\_prio=2 tid=0x0000000017320800 nid=0x7b4 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #6 daemon prio=9 os\_prio=2 tid=0x000000001731b000 nid=0x21b0 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Attach Listener" #5 daemon prio=5 os\_prio=2 tid=0x0000000017319800 nid=0x1294 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os\_prio=2 tid=0x0000000017318000 nid=0x1efc runnable [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Finalizer" #3 daemon prio=8 os\_prio=1 tid=0x0000000002781800 nid=0x5a0 in Object.wait() [0x000000001867f000]

java.lang.Thread.State: WAITING (on object monitor)

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

- waiting on (a java.lang.ref.ReferenceQueue$Lock)

at java.lang.ref.ReferenceQueue.remove(Unknown Source)

- locked (a java.lang.ref.ReferenceQueue$Lock)

at java.lang.ref.ReferenceQueue.remove(Unknown Source)

at java.lang.ref.Finalizer$FinalizerThread.run(Unknown Source)

"Reference Handler" #2 daemon prio=10 os\_prio=2 tid=0x000000000277a800 nid=0x15b4 in Object.wait() [0x000000001857f000]

java.lang.Thread.State: WAITING (on object monitor)

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

- waiting on (a java.lang.ref.Reference$Lock)

at java.lang.Object.wait(Unknown Source)

at java.lang.ref.Reference$ReferenceHandler.run(Unknown Source)

- locked (a java.lang.ref.Reference$Lock)

"VM Thread" os\_prio=2 tid=0x00000000172e6000 nid=0x1fec runnable

"GC task thread#0 (ParallelGC)" os\_prio=0 tid=0x00000000026a6000 nid=0x21fc runnable

"GC task thread#1 (ParallelGC)" os\_prio=0 tid=0x00000000026a7800 nid=0x2110 runnable

"GC task thread#2 (ParallelGC)" os\_prio=0 tid=0x00000000026a9000 nid=0xc54 runnable

"GC task thread#3 (ParallelGC)" os\_prio=0 tid=0x00000000026ab800 nid=0x704 runnable

"VM Periodic Task Thread" os\_prio=2 tid=0x0000000018ba0800 nid=0x610 waiting on condition

JNI global references: 6

Found one Java-level deadlock:

=============================

"Thread-1":

waiting to lock monitor 0x0000000018bc1e88 (object 0x00000000d5d645a0, a Shared),

which is held by "Thread-0"

"Thread-0":

waiting to lock monitor 0x0000000002780e88 (object 0x00000000d5d645b0, a Shared),

which is held by "Thread-1"

Java stack information for the threads listed above:

===================================================

"Thread-1":

at Shared.test1(GFG.java:15)

- waiting to lock (a Shared)

at Shared.test2(GFG.java:29)

- locked (a Shared)

at Thread2.run(GFG.java:68)

"Thread-0":

at Shared.test2(GFG.java:25)

- waiting to lock (a Shared)

at Shared.test1(GFG.java:19)

- locked (a Shared)

at Thread1.run(GFG.java:49)

Found 1 deadlock.

As we can see there is clearly mentioned that found 1 deadlock. It is possible that the same message appears when you try on your machine.

**Avoid Dead Lock condition**

We can avoid dead lock condition by knowing its possibilities. It’s a very complex process and not easy to catch. But still if we try, we can avoid this. There are some methods by which we can avoid this condition. We can’t completely remove its possibility but we can reduce.

* **Avoid Nested Locks :**This is the main reason for dead lock. Dead Lock mainly happens when we give locks to multiple threads. Avoid giving lock to multiple threads if we already have given to one.
* **Avoid Unnecessary Locks :**We should have lock only those members which are required. Having lock on unnecessarily can lead to dead lock.
* **Using thread join :**Dead lock condition appears when one thread is waiting other to finish. If this condition occurs we can use Thread.join with maximum time you think the execution will take.

**Important Points :**

* If threads are waiting for each other to finish, then the condition is known as Deadlock.
* Deadlock condition is a complex condition which occurs only in case of multiple threads.
* Deadlock condition can break our code at run time and can destroy business logic.
* We should avoid this condition as much as we can.

Inter-thread Communication in Java

Prerequisite : [Multithreading in Java](http://geeksquiz.com/multithreading-in-java/), [Synchronized in Java](http://quiz.geeksforgeeks.org/synchronized-in-java/)

[**Multithreading**](https://howtodoinjava.com/category/core-java/multi-threading/) in java is pretty complex topic and requires a lot of attention while writing application code dealing with multiple threads accessing one/more shared resources at any given time. Java 5, introduced some classes like [**BlockingQueue**](https://howtodoinjava.com/java-5/how-to-use-blockingqueue-and-threadpoolexecutor-in-java/) and **Executors** which take away some of the complexity by providing easy to use APIs. Programmers using these classes will feel a lot more confident than programmers directly handling synchronization stuff using **wait() and notify()**method calls. I will also recommend to use these newer APIs over synchronization yourself, BUT many times we are required to do so for various reasons e.g. maintaining legacy code. A good knowledge around these methods will help you in such situation when arrived. In this tutorial, I am discussing some **concepts around methods wait(), notify() and notifyAll()**.

## What are wait(), notify() and notifyAll() methods?

Before moving into concepts, lets note down few very basic definitions involved for these methods.

The Object class in Java has three final methods that allow threads to communicate about the locked status of a resource. These are :

1. **wait()** : It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify(). The wait() method releases the lock prior to waiting and reacquires the lock prior to returning from the wait() method. The wait() method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism. In other words, it is not possible for us to implement the wait() method purely in Java: **it is a native method**.

General syntax for calling wait() method is like this:

|  |
| --- |
| synchronized( lockObject )  {      while( ! condition )      {          lockObject.wait();      }        //take the action here;  } |

1. **notify()** : It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource. It tells a waiting thread that that thread can wake up. However, the lock is not actually given up until the notifier’s synchronized block has completed. So, if a notifier calls notify() on a resource but the notifier still needs to perform 10 seconds of actions on the resource within its synchronized block, the thread that had been waiting will need to wait at least another additional 10 seconds for the notifier to release the lock on the object, even though notify() had been called.

General syntax for calling notify() method is like this:

|  |
| --- |
| synchronized(lockObject)  {      //establish\_the\_condition;        lockObject.notify();        //any additional code if needed  } |

1. **notifyAll()** : It wakes up all the threads that called wait() on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as notify()method above.

General syntax for calling notify() method is like this:

|  |
| --- |
| synchronized(lockObject)  {      establish\_the\_condition;        lockObject.notifyAll();  } |

In general, a thread that uses the wait() method confirms that a condition does not exist (typically by checking a variable) and then calls the wait() method. When another thread establishes the condition (typically by setting the same variable), it calls the notify() method. The wait-and-notify mechanism does not specify what the specific condition/ variable value is. It is on developer’s hand to specify the condition to be checked before calling wait() or notify().

So far, we learned few basic things which you probably already knew. Let’s write a small program to understand how these methods should be used to get desired results.

# Working With hashcode() and equals()

### Need to implement your own custom equality-checking mechanism? Here are some tips for when you need to override hashcode() and equals().

By default, the Java super class **java.lang.Object**provides two important methods for comparing objects: **equals()** and **hashcode()**. These methods become very useful when implementing interactions between several classes in large projects. In this article, we will talk about the relationship between these methods, their default implementations, and the circumstances that force developers to provide a custom implementation for each of them.

## Method Definition and Default Implementation

* **equals(Object obj):** a method provided by **java.lang.Object** that indicates whether some other object passed as an argument is **"equal to"** the current instance. The default implementation provided by the JDK is based on memory location — two objects are equal if and only if they are stored in the same memory address.
* **hashcode():**a method provided by **java.lang.Object** that returns an integer representation of the object memory address. By default, this method returns a random integer that is unique for each instance. This integer might change between several executions of the application and won't stay the same.

## The Contract Between equals() and hashcode()

The default implementation is not enough to satisfy business needs, especially if we're talking about a huge application that considers two objects as equal when some business fact happens. In some business scenarios, developers provide their own implementation in order to force their own equality mechanism regardless the memory addresses.

As per the Java documentation, developers should override both methods in order to achieve a fully working equality mechanism — it's not enough to just implement the **equals()** method.

**If two objects are equal according to the equals(Object) method, then calling the hashcode() method on each of the two objects must produce the same integer result.**

In the following sections, we provide several examples that show the importance of overriding both methods and the drawbacks of overriding **equals()** without **hashcode()**.

## Practical Example

We define a class called **Student** as the following:

package com.programmer.gate.beans;

public class Student {

private int id;

private String name;

public Student(int id, String name) {

this.name = name;

this.id = id;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

For testing purposes, we define a main class **HashcodeEquals**that checks whether two instances of **Student** (who have the exact same attributes) are considered as equal.

public class HashcodeEquals {

public static void main(String[] args) {

Student alex1 = new Student(1, "Alex");

Student alex2 = new Student(1, "Alex");

System.out.println("alex1 hashcode = " + alex1.hashCode());

System.out.println("alex2 hashcode = " + alex2.hashCode());

System.out.println("Checking equality between alex1 and alex2 = " + alex1.equals(alex2));

}

}

Output:

alex1 hashcode = 1852704110

alex2 hashcode = 2032578917

Checking equality between alex1 and alex2 = false

Although the two instances have exactly the same attribute values, they are stored in different memory locations. Hence, they are not considered equal as per the default implementation of **equals()**. The same applies for **hashcode()** — a random unique code is generated for each instance.

## Overriding equals()

For business purposes, we consider that two students are equal if they have the same ID, so we override the **equals()** method and provide our own implementation as the following:

@Override

public boolean equals(Object obj) {

if (obj == null) return false;

if (!(obj instanceof Student))

return false;

if (obj == this)

return true;

return this.getId() == ((Student) obj).getId();

}

In the above implementation, we are saying that two students are equal if and only if they are stored in the same memory address **OR**they have the same ID. Now if we try to run **HashcodeEquals**,we will get the following output:

alex1 hashcode = 2032578917

alex2 hashcode = 1531485190

Checking equality between alex1 and alex2 = true

As you noticed, overriding **equals()**with our custom business forces Java to consider the ID attribute when comparing two **Student** objects.

### equals() With ArrayList

A very popular usage of **equals()**is defining an array list of **Student** and searching for a particular student inside it. So we modified our testing class in order the achieve this.

public class HashcodeEquals {

public static void main(String[] args) {

Student alex = new Student(1, "Alex");

List < Student > studentsLst = new ArrayList < Student > ();

studentsLst.add(alex);

System.out.println("Arraylist size = " + studentsLst.size());

System.out.println("Arraylist contains Alex = " + studentsLst.contains(new Student(1, "Alex")));

}

}

After running the above test, we get the following output:

Arraylist size = 1

Arraylist contains Alex = true

## Overriding hashcode()

Okay, so we override **equals()**and we get the expected behavior — even though the hash code of the two objects are different. So, what's the purpose of overriding **hashcode()**?

### equals() With HashSet

Let's consider a new test scenario. We want to store all the students in a **HashSet**, so we update **HashcodeEquals**as the following:

public class HashcodeEquals {

public static void main(String[] args) {

Student alex1 = new Student(1, "Alex");

Student alex2 = new Student(1, "Alex");

HashSet < Student > students = new HashSet < Student > ();

students.add(alex1);

students.add(alex2);

System.out.println("HashSet size = " + students.size());

System.out.println("HashSet contains Alex = " + students.contains(new Student(1, "Alex")));

}

}

If we run the above test, we get the following output:

HashSet size = 2

HashSet contains Alex = false

WAIT!! We already override **equals()**and verify that alex1 and alex2 are equal, and we all know that **HashSet** stores unique objects, so why did it consider them as different objects ?

**HashSet** stores its elements in memory buckets. Each bucket is linked to a particular hash code. When calling students.add(alex1), Java stores alex1 inside a bucket and links it to the value of alex1.hashcode(). Now any time an element with the same hash code is inserted into the set, it will just replace alex1.However, since alex2 has a different hash code, it will be stored in a separate bucket and will be considered a totally different object.

Now when **HashSet** searches for an element inside it, it first generates the element's hash code and looks for a bucket which corresponds to this hash code.

Here comes the importance of overriding **hashcode()**, so let's override it in **Student** and set it to be equal to the ID so that students who have the same ID are stored in the same bucket:

@Override

public int hashCode() {

return id;

}

Now if we try to run the same test, we get the following output:

HashSet size = 1

HashSet contains Alex = true

See the magic of **hashcode()**! The two elements are now considered as equal and stored in the same memory bucket, so any time you call contains() and pass a student object holding the same hash code, the set will be able to find the element.

The same is applied for **HashMap, HashTable**, or any data structure that uses a hashing mechanism for storing elements.

## Conclusion

In order to achieve a fully working custom equality mechanism, it is mandatory to override **hashcode()** each time you override **equals().** Follow the tips below and you'll never have leaks in your custom equality mechanism:

* If two objects are equal, they MUST have the same hash code.
* If two objects have the same hash code, it doesn't mean that they are equal.
* Overriding **equals()**alone will make your business fail with hashing data structures like: **HashSet, HashMap, HashTable** ... etc.
* Overriding **hashcode()**alone doesn't force Java to ignore memory addresses when comparing two objects.

**Facts about CountDownLatch:**

1. Creating an object of CountDownLatch by passing an int to its constructor (the count), is actually number of invited parties (threads) for an event.
2. The thread, which is dependent on other threads to start processing, waits on until every other thread has called count down. All threads, which are waiting on await() proceed together once count down reaches to zero.
3. countDown() method decrements the count and await() method blocks until count == 0

Java.util.concurrent.CyclicBarrier in Java

[**4.5**](https://www.geeksforgeeks.org/hard/)

CyclicBarrier is used to make threads wait for each other. It is used when different threads process a part of computation and when all threads have completed the execution, the result needs to be combined in the parent thread. In other words, a CyclicBarrier is used when multiple thread carry out different sub tasks and the output of these sub tasks need to be combined to form the final output. After completing its execution, threads call await() method and wait for other threads to reach the barrier. Once all the threads have reached, the barriers then give the way for threads to proceed.

**Working of CyclicBarrier**

CyclicBarriers are defined in java.util.concurrent package. First a new instance of a CyclicBarriers is created specifying the number of threads that the barriers should wait upon.

**CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads);**

Each and every thread does some computation and after completing it’s execution, calls await() methods as shown:

public void run()

{

// thread does the computation

newBarrier.await();

}

**Working of CyclicBarrier:**

https://contribute.geeksforgeeks.org/wp-content/uploads/cyclicbarrier.png  
Once the number of threads that called await() equals **numberOfThreads**, the barrier then gives a way for the waiting threads. The CyclicBarrier can also be initialized with some action that is performed once all the threads have reached the barrier. This action can combine/utilize the result of computation of individual thread waiting in the barrier.

Runnable action = ...

//action to be performed when all threads reach the barrier;

CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads, action);

**Important Methods of CyclicBarrier:**

1. **getParties:** Returns the number of parties required to trip this barrier.  
   **Syntax:**

public int getParties()

**Returns:**  
the number of parties required to trip this barrier

1. **reset:** Resets the barrier to its initial state.  
   **Syntax:**

public void reset()

**Returns:**  
void but resets the barrier to its initial state. If any parties are currently waiting at the barrier, they will return with a BrokenBarrierException.

1. **isBroken:** Queries if this barrier is in a broken state.  
   **Syntax:**

public boolean isBroken()

**Returns:**  
true if one or more parties broke out of this barrier due to interruption or timeout since construction or the last reset, or a barrier action failed due to an exception; false otherwise.

1. **getNumberWaiting:** Returns the number of parties currently waiting at the barrier.  
   **Syntax:**

public int getNumberWaiting()

**Returns:**  
the number of parties currently blocked in await()

1. **await:** Waits until all parties have invoked await on this barrier.  
   **Syntax:**

public int await() throws InterruptedException, BrokenBarrierException

**Returns:**  
the arrival index of the current thread, where index getParties() – 1 indicates the first to arrive and zero indicates the last to arrive.

1. **await:** Waits until all parties have invoked await on this barrier, or the specified waiting time elapses.  
   **Syntax:**
2. public int await(long timeout, TimeUnit unit)
3. throws InterruptedException,

BrokenBarrierException, TimeoutException

**Returns:**  
the arrival index of the current thread, where index getParties() – 1 indicates the first to arrive and zero indicates the last to arrive

|  |
| --- |
| //JAVA program to demonstrate execution on Cyclic Barrier    import java.util.concurrent.TimeUnit;  import java.util.concurrent.TimeoutException;  import java.util.concurrent.BrokenBarrierException;  import java.util.concurrent.CyclicBarrier;    class Computation1 implements Runnable  {      public static int product = 0;      public void run()      {          product = 2 \* 3;          try          {              Tester.newBarrier.await();          }          catch (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }      }  }    class Computation2 implements Runnable  {      public static int sum = 0;      public void run()      {          // check if newBarrier is broken or not          System.out.println("Is the barrier broken? - " + Tester.newBarrier.isBroken());          sum = 10 + 20;          try          {              Tester.newBarrier.await(3000, TimeUnit.MILLISECONDS);                // number of parties waiting at the barrier              System.out.println("Number of parties waiting at the barrier "+              "at this point = " + Tester.newBarrier.getNumberWaiting());          }          catch (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }          catch (TimeoutException e)          {              e.printStackTrace();          }      }  }      public class Tester implements Runnable  {      public static CyclicBarrier newBarrier = new CyclicBarrier(3);        public static void main(String[] args)      {          // parent thread          Tester test = new Tester();            Thread t1 = new Thread(test);          t1.start();      }      public void run()      {          System.out.println("Number of parties required to trip the barrier = "+          newBarrier.getParties());          System.out.println("Sum of product and sum = " + (Computation1.product +          Computation2.sum));            // objects on which the child thread has to run          Computation1 comp1 = new Computation1();          Computation2 comp2 = new Computation2();            // creation of child thread          Thread t1 = new Thread(comp1);          Thread t2 = new Thread(comp2);            // moving child thread to runnable state          t1.start();          t2.start();            try          {              Tester.newBarrier.await();          }          catch (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }            // barrier breaks as the number of thread waiting for the barrier          // at this point = 3          System.out.println("Sum of product and sum = " + (Computation1.product +          Computation2.sum));            // Resetting the newBarrier          newBarrier.reset();          System.out.println("Barrier reset successful");      }  } |

Run on IDE

**Output:**

<Number of parties required to trip the barrier = 3

Sum of product and sum = 0

Is the barrier broken? - false

Number of parties waiting at the barrier at this point = 0

Sum of product and sum = 36

Barrier reset successful

**Explanation:** The value of (sum + product) = 0 is printed on the console because the child thread has’t yet ran to set the values of sum and product variable. Following this, (sum + product) = 36 is printed on the console because the child threads ran setting the values of sum and product. Furthermore, the number of waiting thread on the barrier reached 3, due to which the barrier then allowed all thread to pass and finally 36 was printed. The value of “Number of parties waiting at the barrier at this point” = 0 because all the three threads had already called await() method and hence, the barrier is no longer active. In the end, newBarrier is reset and can be used again.

**BrokenBarrierException**

A barrier breaks when any of the waiting thread leaves the barrier. This happens when one or more waiting thread is interrupted or when the waiting time is completed because the thread called the await() methods with a timeout as follows:

newBarrier.await(1000, TimeUnit.MILLISECONDS);

// thread calling this await()

// methods waits for only 1000 milliseconds.p

When the barrier breaks due to one of more participating threads, the await() methods of all the other threads throws a BrokenThreadException. Whereas, the threads that are already waiting in the barriers have their await() call terminated.

**Difference between a CyclicBarrier and a CountDownLatch**

* A CountDownLatch can be used only once in a program(until it’s count reaches 0).
* A CyclicBarrier can be used again and again once all the threads in a barriers is released.

[CountDownLatch](http://codepumpkin.com/?p=895) Vs [CyclicBarrier](http://codepumpkin.com/?p=935): Though both are used as a synchronization aid that allows one or more threads to wait but there are certain differences between them that you should know in order to know when one of these utilities will serve you better.

As per the [java.util.concurrent API](http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html),

* CountDownLatch: A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads complete.
* CyclicBarrier: A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.

Here are few basic differences.

**1.**In CountDownLatch**,**onlymain thread waits for other threads to complete their execution, where as in CyclicBarrier, Each worker threads wait for each other to complete their execution. Let's understand this by following Example :

* **CountDownLatch:**

Consider a IT world scenario where manager divided modules between development teams (A and B) and he wants to assign it to QA team for testing only when both the teams completes their task.

* Here manager thread works as main thread and development team works as worker thread. Manager thread waits for development teams thread to complete their task. Once developer teams complete their tasks, they will inform manager thread and then manager thread assign modules to QA team.
* **CyclicBarrier:**

Consider the same scenario where manager divided modules between development teams (A and B). He goes on leave. He asked both teams to wait for each other to complete their respective taskand  once both teams are done, assign it to QA team for testing.

* Here manager thread works as main thread and development team works as worker thread. Development team threads wait for other development team threads after completing their task.

In Other words, a CountDownLatch initialized to N can be used to make one thread wait until N threads have completed some action. Where as if CyclicBarrier has been initialized to 3 then you should have at least 3 threads to call await().

**2.** **Reusability :**We can not **reuse**same CountDownLatch instance once count reaches to zero and latch is open. CyclicBarrier can be reused after all the waiting threads are released.

**3. barrierAction**:  A  CyclicBarrier  supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released.

So with CyclicBarrier you have an option to have an **Action class specified** in the CyclicBarrier constructor that will be run after the last thread has called await(). This barrier action is useful for updating shared-state before any of the parties continue.

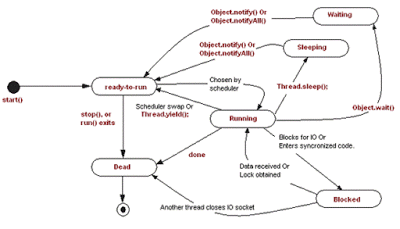
|  |  |
| --- | --- |
| 1 | public CyclicBarrier(int parties, Runnable barrierAction) |

CountDownLatch doesn't provide any such constructor to specify a runnable action.

**Summary**

|  | **COUNT DOWN LATCH** | **CYCLIC BARRIER** |
| --- | --- | --- |
| 1 | Main threads waits for other threads to complete their execution. | Each worker threads wait for each other to complete their execution |
| 2 | We can not reuse same CountDownLatch instance once count reaches to zero and latch is open. | CyclicBarrier can be reused by resetting Barrier, Once barrier is broken. |
| 3 | No Such Action can be provided with CountDownLatch | We can provide barrierAction that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. |

**6) What is the difference between wait and sleep in Java?**([method](http://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html))  
One more *classic Java multithreading question* from the telephonic round of interviews. The key point to mention while answering this question is to mention that wait will release the lock and must be called from the synchronized context, while sleep will only pause the thread for some time and keep the lock.  
  
By the way, both methods throw IntrupptedException and can be interrupted, which can lead to some follow-up questions like, can we awake a sleeping or waiting for a thread in Java? You can also read a detailed answer on my post of same title [here](http://java67.blogspot.com/2012/08/what-are-difference-between-wait-and.html).

[](https://2.bp.blogspot.com/-kJOX8eUPO80/V10_YpHWQYI/AAAAAAAAGNg/6x_PZfACG4wwiacTyn25tkE3B7TO85P0QCLcB/s1600/Sleep+vs+wait+vs+yield+method+java.gif)

**10 Tricky Java interview question - Answered**

Here is my list of 10 tricky Java interview questions, Though I have prepared and shared lot of difficult core Java interview question and answers, But I have chosen them as Top 10 tricky questions because you can not guess answers of this tricky Java questions easily, you need some subtle details of Java programming language to answer these questions.

**Question: What does the following Java program print?**

public class Test {

public static void main(String[] args) {

System.out.println(Math.min(Double.MIN\_VALUE, 0.0d));

}

}

Answer: This question is tricky because unlike the [Integer](http://java67.blogspot.sg/2013/03/how-to-convert-java-string-to-int-or.html), where MIN\_VALUE is negative, both the MAX\_VALUE and MIN\_VALUE of the Double class are positive numbers. The Double.MIN\_VALUE is 2^(-1074), a double constant whose magnitude is the least among all double values. So unlike the obvious answer, this program will print 0.0 because Double.MIN\_VALUE is greater than 0. I have asked this question to Java developer having experience up to 3 to 5 years and surprisingly almost 70% candidate got it wrong.

**What will happen if you put return statement or System.exit () on try or catch block? Will finally block execute?**  
This is a very popular tricky Java question and it's tricky because many programmers think that no matter what, but the [finally block](http://java67.blogspot.com/2016/06/difference-between-final-vs-finally-vs-finalize-in-java.html) will always execute. This question challenge that concept by putting a return statement in the try or catch block or calling System.exit() from try or catch block. Answer of this tricky question in Java is that finally block will execute even if you put a return statement in the try block or catch block but finally block won't run if you call System.exit() from try or catch block.

**Question: Can you override a private or static method in Java?**  
Another popular Java tricky question, As I said method overriding is a good topic to ask trick questions in Java. Anyway, [you can not override a private or static method in Java](http://java67.blogspot.sg/2012/08/can-we-override-static-method-in-java.html), if you create a similar method with same return type and same method arguments in child class then it will hide the superclass method, this is known as method hiding.

Similarly, you cannot override a private method in sub class because it's not accessible there, what you do is create another private method with the same name in the child class. See [Can you override a private method in Java](http://java67.blogspot.sg/2012/08/can-we-override-private-method-in-java.html) or more details.

**Question: What do the expression 1.0 / 0.0 will return? will it throw Exception? any compile time error?**  
Answer: This is another tricky question from Double class. Though Java developer knows about the double primitive type and Double class, while doing floating point arithmetic they don't pay enough attention to Double.INFINITY, NaN, and -0.0 and other rules that govern the arithmetic calculations involving them. The simple answer to this question is that it will not throw ArithmeticExcpetion and return Double.INFINITY.  
  
Also, note that the comparison x == Double.NaN always evaluates to false, even if x itself is a NaN. To test if x is a NaN, one should use the method call Double.isNaN(x) to check if given number is NaN or not. If you know SQL, this is very close to NULL there.   
  
Btw, If you are running out of time for your interview preparation, you can also check out [Java Programming Interviews exposed](http://www.amazon.com/Java-Programming-Interviews-Exposed-Markham/dp/1118722868?tag=javamysqlanta-20) for more of such popular questions,

**Does Java support multiple inheritances?**  
This is the trickiest question in Java if C++ can support direct multiple inheritances than why not Java is the argument Interviewer often give. Answer of this question is much more subtle then it looks like, because Java does support multiple inheritances of Type by allowing an interface to extend other interfaces, what Java doesn't support is multiple inheritances of implementation. This distinction also gets blur because of default method of Java 8, which now provides Java, multiple inheritances of behavior as well. See [why multiple inheritances are not supported in Java](http://javarevisited.blogspot.sg/2011/07/why-multiple-inheritances-are-not.html) to answer this tricky Java question.

**What will happen if we put a key object in a HashMap which is already there?**  
This tricky Java question is part of another frequently asked question, How HashMap works in Java. HashMap is also a popular topic to create confusing and tricky question in Java. Answer of this question is if you put the same key again then it will replace the old mapping because HashMap doesn't allow duplicate keys. The Same key will result in the same hashcode and will end up at the same position in the bucket.

 Each bucket contains a linked list of Map.Entry object, which contains both Key and Value. Now Java will take the Key object from each entry and compare with this new key using equals() method, if that return true then value object in that entry will be replaced by new value. See [How HashMap works in Java](http://java67.blogspot.sg/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) for more tricky Java questions from HashMap.

**Question: What does the following Java program print?**

public class Test {

public static void main(String[] args) throws Exception {

char[] chars = new char[] {'\u0097'};

String str = new String(chars);

byte[] bytes = str.getBytes();

System.out.println(Arrays.toString(bytes));

}

}

Answer: The trickiness of this question lies on character encoding and how String to byte array conversion works. In this program, we are first creating a String from a character array, which just has one character '\u0097', after that we are getting the byte array from that String and printing that byte. Since \u0097 is within the 8-bit range of byte primitive type, it is reasonable to guess that the str.getBytes() call will return a byte array that contains one element with a value of -105 ((byte) 0x97).  
  
However, that's not what the program prints and that's why this question is tricky. As a matter of fact, the output of the program is operating system and locale dependent. On a Windows XP with the US locale, the above program prints [63], if you run this program on Linux or Solaris, you will get different values.  
  
To answer this question correctly, you need to know about how Unicode characters are represented in Java char values and in Java strings, and what role character encoding plays in String.getBytes().  
  
In simple word, t[o convert a string to a byte array](http://javarevisited.blogspot.sg/2014/08/2-examples-to-convert-byte-array-to-String-in-Java.html), Java iterate through all the characters that the string represents and turn each one into a number of bytes and finally put the bytes together. The rule that maps each Unicode character into a byte array is called a character encoding. So It's possible that if same character encoding is not used during both encoding and decoding then retrieved value may not be correct. When we call str.getBytes() without specifying a character encoding scheme, the JVM uses the default character encoding of the platform to do the job.  
  
The default encoding scheme is operating system and locale dependent. On Linux, it is UTF-8 and on Windows with a US locale, the default encoding is Cp1252. This explains the output we get from running this program on Windows machines with a US locale. No matter which character encoding scheme is used, Java will always translate Unicode characters not recognized by the encoding to 63, which represents the character U+003F (the question mark, ?) in all encodings.

**If a method throws NullPointerException in the superclass, can we override it with a method which throws RuntimeException?**  
One more tricky Java questions from the overloading and overriding concept. The answer is you can very well throw superclass of RuntimeException in overridden method, but you can not do same if its checked Exception. See [Rules of method overriding in Java](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html) for more details.

**What is the issue with following implementation of compareTo() method in Java**

public int compareTo(Object o){

Employee emp = (Employee) o;

return this.id - e.id;

}

**where an id is an integer number.**

Well, three is nothing wrong in this Java question until you guarantee that id is always positive. This Java question becomes tricky when you can't guarantee that id is positive or negative. the tricky part is, If id becomes negative than **subtraction may overflow** and produce an incorrect result. See [How to override compareTo method in Java](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) for the complete answer of this Java tricky question for an experienced programmer.

**How do you ensure that N thread can access N resources without deadlock?**  
If you are not well versed in writing multi-threading code then this is a real tricky question for you. This Java question can be tricky even for the experienced and senior programmer, who are not really exposed to deadlock and race conditions. The key point here is ordering, if you acquire resources in a particular order and release resources in the reverse order you can prevent deadlock. See [how to avoid deadlock in Java](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html) for a sample code example.

**Question: Consider the following Java code snippet, which is initializing two variables and both are not volatile, and two threads T1 and T2 are modifying these values as following, both are not synchronized**

int x = 0;

boolean bExit = false;

Thread 1 (not synchronized)

x = 1;

bExit = true;

Thread 2 (not synchronized)

if (bExit == true)

System.out.println("x=" + x);

**Now tell us, is it possible for Thread 2 to print “x=0”?**  
  
Answer: It's impossible for a list of tricky Java questions to not contain anything from multi-threading. This is the simplest one I can get. Answer of this question is Yes, It's possible that thread T2 may print x=0.Why? because without any instruction to compiler e.g. synchronized or volatile, bExit=true might come before x=1 in compiler reordering. Also, x=1 might not become visible in Thread 2, so Thread 2 will load x=0. Now, how do you fix it?  
  
 When I asked this question to a couple of programmers they answer differently, one suggests to make both threads synchronized on a common mutex, another one said make both variable volatile. Both are correct, as it will prevent reordering and guarantee visibility.  
  
But the best answer is you just need to make bExit as volatile, then Thread 2 can only print “x=1”. x does not need to be volatile because x cannot be reordered to come after bExit=true when bExit is volatile.

**What is difference between CyclicBarrier and CountDownLatch in Java**  
Relatively newer Java tricky question, only been introduced from Java 5. The main difference between both of them is that you can reuse CyclicBarrier even if Barrier is broken, but you can not reuse CountDownLatch in Java. See [CyclicBarrier vs CountDownLatch in Java](http://java67.blogspot.sg/2012/08/difference-between-countdownlatch-and-cyclicbarrier-java.html) for more differences.

**What is the difference between StringBuffer and StringBuilder in Java?**  
Classic Java questions which some people think tricky and some consider very easy. StringBuilder in Java was introduced in JDK 1.5 and the only difference between both of them is that StringBuffer methods e.g. length(), capacity() or append() are [synchronized](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) while corresponding methods in StringBuilder are not synchronized.  
  
Because of this fundamental difference, concatenation of String using StringBuilder is faster than StringBuffer. Actually, it's considered the bad practice to use StringBuffer anymore, because, in almost 99% scenario, you perform string concatenation on the same thread. See [StringBuilder vs StringBuffer](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) for more differences.

**Can you access a non-static variable in the static context?**  
Another tricky Java question from Java fundamentals. No, you can not access a non-static variable from the static context in Java. If you try, it will give compile time error. This is actually a common problem beginner in Java face when they try to access instance variable inside the main method. Because main is static in Java, and instance variables are non-static, you can not access instance variable inside main. See, [why you can not access a non-static variable from static method](http://javarevisited.blogspot.sg/2012/02/why-non-static-variable-cannot-be.html) to learn more about this tricky Java questions.

# Why String is immutable or final in Java

Why String is [immutable](https://www.journaldev.com/129/how-to-create-immutable-class-in-java) in java is one of the popular interview question. String is one of the most used classes in any programming language. We know that String is immutable and final in java and java runtime maintains a [String pool](https://www.journaldev.com/797/what-is-java-string-pool) that makes it a special class.

## Why String is immutable in Java?

Let’s look at some benefits of String immutability, that will help in understanding why String is immutable in java.

1. [String pool](https://www.journaldev.com/797/what-is-java-string-pool) is possible only because String is immutable in java, this way Java Runtime saves a lot of java heap space because different String variables can refer to same String variable in the pool. If String would not have been immutable, then String interning would not have been possible because if any variable would have changed the value, it would have been reflected to other variables also.
2. If String is not immutable then it would cause severe security threat to the application. For example, database username, password are passed as String to get database connection and in [socket programming](https://www.journaldev.com/741/java-socket-programming-server-client) host and port details passed as String. Since String is immutable it’s value can’t be changed otherwise any hacker could change the referenced value to cause security issues in the application.
3. Since String is immutable, it is safe for [multithreading](https://www.journaldev.com/1079/multithreading-in-java) and a single String instance can be shared across different threads. This avoid the usage of synchronization for thread safety, Strings are implicitly thread safe.
4. Strings are used in [java classloader](https://www.journaldev.com/349/java-classloader) and immutability provides security that correct class is getting loaded by Classloader. For example, think of an instance where you are trying to load java.sql.Connection class but the referenced value is changed to myhacked.Connection class that can do unwanted things to your database.
5. Since String is immutable, its **hashcode** is cached at the time of creation and it doesn’t need to be calculated again. This makes it a great candidate for key in a Map and it’s processing is fast than other HashMap key objects. This is why String is mostly used Object as HashMap keys.

Above are some of the reasons I could think of that shows benefits of String immutability. It’s a great feature of [Java String](https://www.journaldev.com/16928/java-string) class and makes it special.

# 

# [Can we Override Private Method in Java? Inner Class?](http://www.java67.com/2013/08/can-we-override-private-method-in-java-inner-class.html)

No, you cannot override private methods in Java, private methods are non-virtual in Java and access differently than non-private one. Since [method overriding](http://java67.blogspot.com/2012/08/what-is-method-overriding-in-java-example-tutorial.html) can only be done on derived class and private methods are not accessible in a subclass, you just can not override them. By the way, one more possibility of overriding private methods in an inner class, since private methods are accessible in an inner class, and that’s why it is one of the [tricky java interview questions](http://java67.blogspot.com/2012/09/top-10-tricky-java-interview-questions-answers.html). Anyway, this will also not work because private methods are bonded during compile time and only Type (or Class) is used to locate a private method. For Example in below code where it looks like that [nested class](http://java67.blogspot.com/2012/10/nested-class-java-static-vs-non-static-inner.html) is an overriding private method, but if you call privateMethod() with a type of super class but the object of the subclass, it will only execute privateMethod() declared in the parent class, which is not exactly method overriding.

[Read more »](http://www.java67.com/2013/08/can-we-override-private-method-in-java-inner-class.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[7 comments:](http://www.java67.com/2013/08/can-we-override-private-method-in-java-inner-class.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=4366972295390608191)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=4366972295390608191)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4366972295390608191&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4366972295390608191&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4366972295390608191&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4366972295390608191&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4366972295390608191&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [programming](http://www.java67.com/search/label/programming?&max-results=3) Location: [United States](https://maps.google.com/maps?q=United+States@37.09024,-95.71289100000001&z=10)

# [Best Way to Iterate Over Each Entry of HashMap in Java](http://www.java67.com/2013/08/best-way-to-iterate-over-each-entry-in.html)

What is the best way to Iterate over HashMap in Java? and not just HashMap, but any Map implementation including old Hashtable, TreeMap, LinkedHashMap and relatively newer ConcurrentHashMap, is a frequently asked query from Java Programmers, with some experience under his belt. Well, when it comes to choosing between [different ways to iterate over Map in Java](http://javarevisited.blogspot.com/2011/12/how-to-traverse-or-loop-hashmap-in-java.html), it's you need, which plays an important role. For example, if you just want to iterate over each entry of HashMap, without modifying Map, then iterating over entry set using Java 1.5 foreach loop seems the most elegant solution to me. The reason, it just two lines of code using a foreach loop and Generics, and by getting the set of entries, we get key and value together, without further searching in HashMap. This makes it also the fastest way to loop over HashMap in Java.

[Read more »](http://www.java67.com/2013/08/best-way-to-iterate-over-each-entry-in.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[7 comments:](http://www.java67.com/2013/08/best-way-to-iterate-over-each-entry-in.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=8776434022551483645)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=8776434022551483645)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8776434022551483645&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8776434022551483645&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8776434022551483645&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8776434022551483645&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8776434022551483645&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [Java collection tutorial](http://www.java67.com/search/label/Java%20collection%20tutorial?&max-results=3) Location: [United States](https://maps.google.com/maps?q=United+States@37.09024,-95.71289100000001&z=10)

# [Difference between Comparator and Comparable in Java - Interview Question](http://www.java67.com/2013/08/difference-between-comparator-and-comparable-in-java-interface-sorting.html)

Comparator and Comparable are two interfaces in Java API, which is used to compare two objects in Java. Though both are used for comparison there are some differences between them, a major difference between Comparable and Comparator is that former is used to define the natural ordering of object e.g. lexicographic order for java.lang.String, while later is used to define any alternative ordering for an object.  The main usage of java.lang.Comparable and java.util.Comparator interface is for [sorting a list of objects in Java](http://java67.blogspot.com/2012/10/how-to-sort-object-in-java-comparator-comparable-example.html). For example to sort a list of Employee by their Id, we can use Comparable interface and provide additional sorting capability, we can define multiple comparators e.g. AgeComparator to compare the age of the employee, SalaryComparator to compare the salary of employees etc.  This brings another i*mportant difference between Comparator and Comparable interface in Java*, you can have only one ordering via Comparable e.g. natural ordering while you can define multiple Comparator for alternative ordering as discussed above.

# [15 Java Enum Interview Questions for Developers with Answers](http://www.java67.com/2013/07/15-java-enum-interview-questions-amswers-for-experienced-programmers.html)

Enum was introduced in Java 5 and since then it's been very popular among Java developers and widely used in different Java applications. Since Enum in Java is much more versatile than Enum in C or C++, it also presents lots of interesting use cases, couple of them, we have seen in my article [10 ways to use Enum in Java](http://javarevisited.blogspot.sg/2011/08/enum-in-java-example-tutorial.html). But, despite being so popular, many Java programmer are still not aware of functionality provided by Enum and subtle details of using Enum in Java code. I realized this fact, when couple of my readers asked me some of the questions e.g. **Can Enum implement an interface in Java** or **Why we can not create Enum instances outside of Enum**, stating that these has been asked to them in there Java Interviews. This motivates me to put together a list of frequently asked question in Java Enum, which not only helps to do well in Interviews, but also open new path for learning. As I had said before, lot of time a question in Interviews, makes you to take a topic more seriously than otherwise, which is not a bad thing, and given the power and facilities offered by Java Enum, I think it's high time to get master of it.

[Read more »](http://www.java67.com/2013/07/15-java-enum-interview-questions-amswers-for-experienced-programmers.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[5 comments:](http://www.java67.com/2013/07/15-java-enum-interview-questions-amswers-for-experienced-programmers.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=4543208265068388382)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=4543208265068388382)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4543208265068388382&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4543208265068388382&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4543208265068388382&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4543208265068388382&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4543208265068388382&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [enum](http://www.java67.com/search/label/enum?&max-results=3)

# [Difference between this and super keywords in Java](http://www.java67.com/2013/06/difference-between-this-and-super-keyword-java.html)

this and super are two special keywords in Java, which is used to represent current instance of a class and it's super class. Java Programmers often confused between them and not very familiar with there special properties, which is asked at various [core Java interviews](http://java67.blogspot.com/2012/09/top-10-tough-core-java-interview-questions-answers.html). A couple of questions, which I remember about this and super keyword is  that, **Can we reassign this in Java?**  and the ***difference between this and super keyword in Java***. Do you want to try that? Ok, I am not giving the answer now, rather I will let you know the answer at the end of this post. As I said in the first line, the main difference between this and super in Java is that this represents current instance of a class, while super represent current instance of the parent class. Now where does this and super variables used, well you might have seen examples of calling one constructor from other i.e. [constructor chaining](http://java67.blogspot.com/2012/12/how-constructor-chaining-works-in-java.html), that's achieved by using this and super keyword?

Read more: <http://www.java67.com/search/label/core%20java%20interview%20question%20answer?updated-max=2014-01-22T06:49:00-08:00&max-results=6#ixzz4i1XxK3Qz>

[How get method of HashMap or Hashtable works internally in Java](http://www.java67.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html)

In this article, I am revisiting a couple of interesting question related to the internal working of HashMap in Java, mostly asked senior Java developers, ranging from 4 to 6 and up to 8 years of experience. I did cover lot of these questions from HashMap, ranging from thread-safety to race conditions, in my post about [internal working of Java HashMap](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html), but I thought to revisit two of those questions, *How get method of HashMap or Hashtable works internally in Java* and *What happens if two different keys return the same hashCode*, how do you return value from HashMap in that case. These are the question, which is highly popular in investment banking domain, and preferred choice of interviewer, while interviewing experienced Java professional.

[Read more »](http://www.java67.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[8 comments:](http://www.java67.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=4545729862003998075)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=4545729862003998075)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4545729862003998075&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4545729862003998075&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4545729862003998075&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4545729862003998075&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4545729862003998075&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [Java collection tutorial](http://www.java67.com/search/label/Java%20collection%20tutorial?&max-results=3), [Sorting Algorithm](http://www.java67.com/search/label/Sorting%20Algorithm?&max-results=3)

# Difference between Deep and Shallow Copy in Java Object Cloning

Shallow copy and deep copy is related with cloning process so before go into the deep of shallow and deep copy we need to understand what is clone in java. Clone is nothing but the process of copying one object to produce the exact object, which is not guaranteed. We all know in Java object is referred by reference we can not copy one object directly to another object. So we have cloning process to achieve this objective. Now one question arises in mind why we need this process so the answer is whenever we need a local copy of the object to modify the object in some method but not in method caller.  So we can define Cloning as “**create a copy of object** “  .I think now we are somehow clear about the cloning but there  is more to it depending on how we are doing this copy, we can divide cloning into two types.

* Shallow Copy
* Deep Copy

Before going into the deep of shallow and deep copy we need to understand how we achieve cloning in java.

### How to Clone in java?

[Difference between deep cloning vs shallow cloning in Java](http://3.bp.blogspot.com/-1lzFJzIgaHk/UF2Ci6kY5pI/AAAAAAAAAes/OYiM7r-DHzc/s1600/17.jpg)In Java, everything is achieved through class, object and interface .By default no Java class support cloning but Java provide one interface called Cloneable, which is a [marker interface](http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html) andbyimplementingthis interface we can make the duplicate copy of our object by calling clone() method of  java.lang.Object class.   
  
  
  
This Method is protected inside the object class and Cloneable interface is  a marker interface and this method also throw  **CloneNotSupportedException**if we have not implemented thisinterface and try to call clone() method of Object class.  By default any clone() method gives the **shallow copy** of the object i.e. if we invoke **super.clone()** then it’s a shallow copy but if we want to **deep copy** we have to override the clone() method and make it public and give own definition of making copy of object. Now we let’s see  what is shallow and deep copy of object in Java programming language.

### Shallow Copy

Whenever we use default implementation of clone method we get shallow copy of object means it create new instance and copy all the field of object to that new instance and return it as **object type** we need to explicitly cast it back to our original object. This is shallow copy of the object. clone() method of the object class support shallow copy of the object. If the object contains primitive as well as non primitive or reference type variable In  shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves. That's why the name shallow copy or shallow cloning in Java. If only primitive type fields or [Immutable objects](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html) are there then there is no difference between shallow and deep copy in Java.

### Deep Copy

Whenever we need own meaning of copy not to use default implementation we call it as deep copy, whenever we need deep copy of the object we need to implement according to our need. So for deep copy we need to ensure all the member class also implement the Cloneable interface and override the clone() method of the object class. After that we override the clone() method in all those classes even in the classes where we have only primitive type members otherwise we would not be able to call the protected clone() method of Object class on the instances of those classes inside some other class. It’s typical restriction of the protected access.

### Difference between Shallow and Deep Copy in Java

I think now we know what is deep and shallow copy of object in Java, let see some difference between them so that we can get some more clarity on them.

* When we call Object.clone(), this method performs a shallow copy of object, by copying data field by field, and if we override this method and by convention first call super.clone(), and then modify some fields to "deep" copy, then we get deep copy of object. This modification is done to ensure that original and cloned object are independent to each other.
* In shallow copy main or parent object is copied, but they share same fields or children if fields are modified in one parent object other parent fields have automatic same changes occur,but in deep copy this is not the case.
* If our parent object contains only primitive value then shallow copy is good for making clone of any object because in new object value is copied but if parent object contains any other object then only reference value is copied in new parent object and both will point to same object so in that case according to our need we can go for deep copy.
* Deep copy is expensive as compare to shallow copy in terms of object creation, because it involves recursive copying of data from other mutable objects, which is part of original object.

This is all about deep copy and shallow copy of objects in Java. Now the question comes when we use shallow copy and when go for deep copy , so answer would be  simple that if the object has only primitive fields or Immutable objects, then obviously we will go for shallow copy, but if the object has references to other mutable objects, then based on the requirement, shallow copy or deep copy can be chosen. Means  if the references are not modified anytime, then there is no point in going for deep copy, We can go for shallow copy. But if the references are modified often, then you need to go for deep copy. Again there is no hard and fast rule, it all depends on the requirement.

Hope this article will help to make clear about deep and shallow copy of cloning process.

Read more: <http://www.java67.com/2013/05/difference-between-deep-copy-vs-shallow-cloning-java.html#ixzz5UpdMvnE8>

# Java transient keyword example

By Lokesh Gupta | Filed Under: [Java Basics](https://howtodoinjava.com/java/basics/)

The **Java transient keyword** is used on class attributes/variables to indicate that serialization process of such class should ignore such variables while creating a persistent byte stream for any instance of that class.

**A transient variable is a variable that can not be serialized. According to Java Language Specification [**[**jls-8.3.1.3**](https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.3.1.3)**] – “Variables may be marked transient to indicate that they are not part of the persistent state of an object.”**

In this post, I will discussing various concepts involved around usage of **transient** keyword in context of [**serialization**](https://howtodoinjava.com/java/serialization/a-mini-guide-for-implementing-serializable-interface-in-java/).

Table of Contents

1. [What is transient keyword in Java?](https://howtodoinjava.com/java/basics/transient-keyword-in-java-with-real-time-example/#what_is_transient)

2. [When should we use transient keyword in java?](https://howtodoinjava.com/java/basics/transient-keyword-in-java-with-real-time-example/#when_to_use_transient)

3. [Usage of transient with final keyword](https://howtodoinjava.com/java/basics/transient-keyword-in-java-with-real-time-example/#transient_with_final)

4. [Case study: How does a HashMap use transient?](https://howtodoinjava.com/java/basics/transient-keyword-in-java-with-real-time-example/#transient_case_study)

5. [Summary Notes](https://howtodoinjava.com/java/basics/transient-keyword-in-java-with-real-time-example/#summary)

## 1. What is Java transient keyword

The modifier **transient** in java can be applied to field members of a class to turn off serialization on these field members. Every field marked as **transient** will not be serialized. You use the **transient**keyword to indicate to the java virtual machine that the **transient** variable is not part of the persistent state of an object.

Let’s write an very basic example to understand what exactly above analogy means. I will create an Employee class and will define 3 attributes i.e. firstName, lastName and confidentialInfo. We do not want to store/save “**confidentialInfo**” for some purpose so we will mark the field as “**transient**“.

class Employee implements Serializable

{

private String firstName;

private String lastName;

private transient String confidentialInfo;

//Setters and Getters

}

Now let’s **serialize an instance of Employee** class.

try

{

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("empInfo.ser"));

Employee emp = new Employee();

emp.setFirstName("Lokesh");

emp.setLastName("Gupta");

emp.setConfidentialInfo("password");

//Serialize the object

oos.writeObject(emp);

oos.close();

} catch (Exception e)

{

System.out.println(e);

}

Now let’s **de-serialize back into java object**, and verify if “confidentialInfo” was saved or not?

try

{

ObjectInputStream ooi = new ObjectInputStream(new FileInputStream("empInfo.ser"));

//Read the object back

Employee readEmpInfo = (Employee) ooi.readObject();

System.out.println(readEmpInfo.getFirstName());

System.out.println(readEmpInfo.getLastName());

System.out.println(readEmpInfo.getConfidentialInfo());

ooi.close();

} catch (Exception e)

{

System.out.println(e);

}

Program Output.

Lokesh

Gupta

null

Clearly, “confidentialInfo” was not saved to persistent state while serialization and that’s exactly why we use “**transient**” keyword in java.

## 2. When should we use transient keyword in java?

Now we have a very knowledge of “**transient**” keyword. Let’s expand out understanding by identifying the situations where you will need the **use of transient keyword**.

1. First and very logical case would be where you may have **fields that are derived/calculated from other fields** within instance of class. They should be calculated programmatically everytime rather than having the state be persisted via serialization. An example could be time-stamp based value; such as age of a person OR duration between a timestamp and current timestamp. In both cases, you would be calculating value of variable based on current system time rather than when the instance was serialized.
2. Second logical example can be**any secure information** which should not leak outside the JVM in any form (either in database OR byte stream).
3. Another example could be fields which are **not marked as “Serializable”** inside JDK or application code. Classes which do not implement Serializable interface and are referenced within any serializable class, cannot be serialized; and will throw “java.io.NotSerializableException” exception. These non-serializable references should be marked “transient” before serializing the main class.
4. And lastly, there are times when it **simply doesn’t make sense to serialize some fields**. Period. For example, In any class if you have added a logger reference, then what’s use of serializing that logger instance. Absolutely no use. You serialize the information which represent the state of instance, logically. Loggers never share the state of an instance. They are just utilities for programming/debugging purpose. Similar example can be reference of a Thread class. Threads represent a state of a process at any given point of time, and there is no use to store thread state with your instance; simply because they do not constitute the state of your class’s instance.

Above four usecases are when you should use the keyword “**transient**” with reference variables. If you have some more logical cases where “**transient**” can be used; please share with me and I will update that here in list so that everybody can benefit from your knowledge.

Read More : [A min guide for implementing serializable interface](https://howtodoinjava.com/java/serialization/a-mini-guide-for-implementing-serializable-interface-in-java/)

## 3. Transient with final

I am talking about use of **transient** with **final** keyword specifically because it behaves differently in different situations which is not generally the case with other keywords in java.

For making this concept practical, I have modified the Employee class as below:

private String firstName;

private String lastName;

//final field 1

public final transient String confidentialInfo = "password";

//final field 2

public final transient Logger logger = Logger.getLogger("demo");

Now when I again run the serialization (write/read) again, below is the output:

Program output.

Lokesh

Gupta

password

null

This is strange. We have marked the “confidentialInfo” to transient; and still the field was serialized. For similar declaration, logger was not serialized. Why?

Reason is that whenever any final field/reference is evaluated as “[**constant expression**](https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.28)“, it is serialized by JVM ignoring the presence of transient keyword.

In above example, value “password” is a constant expression and instance of logger “demo” is reference. So by rule, confidentialInfo was persisted where as logger was not.

Are you thinking, what if I remove “transient” from both fields? Well, then fields implementing Serializable references will persist otherwise not. So, if you remove transient in above code, String (which implements Serializable) will be persisted; where as Logger (which does NOT implements Serializable) will not be persisted AND “java.io.NotSerializableException” will be thrown.

If you want to persist the state of non-serializable fields then use readObject() and writeObject() methods. writeObject()/readObject() are usually chained into serialization/deserialization mechanisms internally and thus called automatically.

Read More : [SerialVersionUID in java and related fast facts](https://howtodoinjava.com/java/serialization/serialversionuid/)

## 4. Case study : How does a HashMap use transient keyword?

So far, we have been talking about **concepts related to “transient” keyword** which are mostly theoretical in nature. Let’s understand the proper use of “**transient**” which is **used inside HashMap class**very logically. It will give you better idea about **real life usage of transient keyword in java**.

Before understanding the solution which has been created using transient, let’s **first identify the problem** itself.

HashMap is used to store key-value pairs, we all know that. And we also know that location of keys inside HashMap is calculated based on hash code obtained for instance of key. Now when we serialize a HashMap that means all keys inside HashMap and all values respective to key’s will also be serialized. After serialization, when we de-serialize the HashMap instance then all key instances will also be deserialized. We know that during this serialization/deserialization process, there may be loss of information (used to calculate hashcode) as well as most important is it is a NEW INSTANCE itself.

In java, **any two instances (even of same class) can not have same hashcode**. This is a big problem because the location where keys should be placed according to new hashcodes, are not in there correct positions. When retrieving the value of a key, you will be referring to the wrong index in this new HashMap.

Read More : [Working with hashCode and equals methods in java](https://howtodoinjava.com/java/basics/java-hashcode-equals-methods/)

So, when a hash map is serialized, it means that the hash index, and hence the ordering of the table is no longer valid and should not be preserved. This is the problem statement.

Now look at how it is solved inside HashMap class. If go through the sourcecode of [**HashMap.java**](http://www.docjar.com/html/api/java/util/HashMap.java.html), you will find below declarations:

transient Entry table[];

transient int size;

transient int modCount;

transient int hashSeed;

private transient Set entrySet;

All important fields have been marked at “**transient**” (all of them are actually calculated/change at runtime), so they are not part of serialized HashMap instance. To populate again this important information back, HashMap class uses **writeObject()** and **readObject()** methods as below:

private void writeObject(ObjectOutputStream objectoutputstream) throws IOException

{

objectoutputstream.defaultWriteObject();

if (table == EMPTY\_TABLE)

objectoutputstream.writeInt(roundUpToPowerOf2(threshold));

else

objectoutputstream.writeInt(table.length);

objectoutputstream.writeInt(size);

if (size > 0)

{

Map.Entry entry;

for (Iterator iterator = entrySet0().iterator(); iterator.hasNext(); objectoutputstream.writeObject(entry.getValue()))

{

entry = (Map.Entry) iterator.next();

objectoutputstream.writeObject(entry.getKey());

}

}

}

private void readObject(ObjectInputStream objectinputstream) throws IOException, ClassNotFoundException

{

objectinputstream.defaultReadObject();

if (loadFactor <= 0.0F || Float.isNaN(loadFactor))

throw new InvalidObjectException((new StringBuilder())

.append("Illegal load factor: ").append(loadFactor).toString());

table = (Entry[]) EMPTY\_TABLE;

objectinputstream.readInt();

int i = objectinputstream.readInt();

if (i < 0)

throw new InvalidObjectException((new StringBuilder()).append("Illegal mappings count: ").append(i).toString());

int j = (int) Math.min((float) i \* Math.min(1.0F / loadFactor, 4F), 1.073742E+009F);

if (i > 0)

inflateTable(j);

else

threshold = j;

init();

for (int k = 0; k < i; k++)

{

Object obj = objectinputstream.readObject();

Object obj1 = objectinputstream.readObject();

putForCreate(obj, obj1);

}

}

With above code, HashMap still let the non-transient fields to be treated as they would normally do, but they wrote the stored key-value pairs at the end of the byte array one after the other. While de-serializing, it let the non-transient variables to be handled by default de-serialization process and then read the key-value pairs one by one. **For each key the hash and the index is calculated again and is inserted to the correct position in the table** so that it can be retrieved again without any error.

Above use of transient keyword was a very good example of proper usecase. You should keep remember it and mention it whenever it is asked in your next [java interview question](https://howtodoinjava.com/java-interview-questions/).

Related Post: [How HashMap works in Java?](https://howtodoinjava.com/java/collections/how-hashmap-works-in-java/)

## 5. Summary Notes

1. The modifier transient can be applied to field members of a class to turn off serialization on these field members.
2. You can use transient keyword in classes with fields which needs to be secured or calculated on existing state fields. And use it when it simply doesn’t make sense to serialize those fields such as loggers and threads.
3. Serialization does not care about access modifiers such as private; all non-transient fields are considered part of an object’s persistent state and are eligible for persistence.
4. Whenever any final field/reference is evaluated as “constant expression”, it is serialized by JVM ignoring the presence of transient keyword.
5. A good usecase of transient keyword in java is HashMap class.

[How to Override Equals, HashCode and CompareTo method in Java](http://www.java67.com/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html)

Though modern IDE like Eclipse, IntelliJ or Netbeans allows you to generate equals, hashCode and compareTo methods for your value classes, it's equally important, you know how to do that by hand. By overriding equals and hashcode method by yourself, you know how they work, what kind of errors you can get, and most importantly, it's expected form you, as a Java programmer in any [core Java interview](http://java67.blogspot.com/2012/09/top-10-tough-core-java-interview-questions-answers.html). More often, you would see a coding question in Java, which ask you to override equals(), hashcode(), compare() and compareTo() methods for a value class. Since I have already shared some tips on How to override compareTo method in Java, and couple of example of writing your own Comparator in Java, here I am sharing another simple example of overriding equals and hashCode methods. If you know rules of overriding equals and hashCode, you might know that, whenever you override equals, you must have to override hashCode, otherwise your object will not behave properly on various collection classes e.g. [Map or Set](http://java67.blogspot.sg/2013/01/difference-between-set-list-and-map-in-java.html), which uses equals, compareTo, hashCode to implement there invariants e.g. Set implementations should not allow any duplicates.

[Read more »](http://www.java67.com/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[5 comments:](http://www.java67.com/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=6623235597521393353)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=6623235597521393353)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=6623235597521393353&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=6623235597521393353&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=6623235597521393353&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=6623235597521393353&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=6623235597521393353&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [Java programming Tutorial](http://www.java67.com/search/label/Java%20programming%20Tutorial?&max-results=3)

[Difference between notify and notifyAll in Java](http://www.java67.com/2013/03/difference-between-wait-vs-notify-vs-notifyAll-java-thread.html)

wait, notify, and notifyAll methods are used for inter-thread communication in Java. wait() allows a thread to check for a condition, and wait if the condition doesn't meet, while notifying() and notifyAll() method informs waiting for a thread for rechecking condition, after changing the state of a shared variable. One good example of how to wait and notify method works is [Producer consumer problem](http://java67.blogspot.sg/2012/12/producer-consumer-problem-with-wait-and-notify-example.html), where one thread produces and wait if the bucket is full; and another thread consumes and waits if the bucket is empty. Both Producer and Consumer thread, notify each other as well. Producer thread notifies consumer thread after inserting an item in the shared queue, while consumer thread notifies producer, after consuming item from the queue. Though Both notify() and notifyAll()  are used to notify waiting for threads, waiting on shared queue object, but there are some subtle differences between notify and notifyAll in Java.

[Read more »](http://www.java67.com/2013/03/difference-between-wait-vs-notify-vs-notifyAll-java-thread.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[1 comment:](http://www.java67.com/2013/03/difference-between-wait-vs-notify-vs-notifyAll-java-thread.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=8518867980212190279)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=8518867980212190279)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8518867980212190279&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8518867980212190279&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8518867980212190279&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8518867980212190279&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8518867980212190279&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [thread](http://www.java67.com/search/label/thread?&max-results=3)

[Difference between JDK and JRE in Java Platform](http://www.java67.com/2013/02/difference-between-jdk-and-jre-in-java.html)

Java Platform offers JRE and JDK to run Java programs. JRE stands for Java runtime environment and JDK stands for Java development kit. JRE is meant for normal users, who wants to run Java program in their computer. JRE is normally used to run Java programs downloadedover internet e.g. Java Applets and Java Desktop application built using [AWT and Swing](http://java67.blogspot.com/2013/01/10-awt-swing-interview-questions-answers-java.html). The main difference between JRE and JDK, comes from the fact that they are different tools. JDK is created for Java programmers and contains tools required for Java programming, e.g. javacfor compiling Java source files to [.class files](http://javarevisited.blogspot.com/2012/05/10-points-about-class-file-in-java.html). Without JDK, you can not create Java applications and programs. By the way, JDK comes with its own JRE, but when you run Java program using java command, the JRE which comes first in System PATH is used for execution.

[Read more »](http://www.java67.com/2013/02/difference-between-jdk-and-jre-in-java.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[2 comments:](http://www.java67.com/2013/02/difference-between-jdk-and-jre-in-java.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=8569148139596326967)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=8569148139596326967)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8569148139596326967&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8569148139596326967&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8569148139596326967&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8569148139596326967&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=8569148139596326967&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [programming](http://www.java67.com/search/label/programming?&max-results=3)

[Difference between JIT and JVM in Java - Interview Question](http://www.java67.com/2013/02/difference-between-jit-and-jvm-in-java.html)

The main difference between JIT and JVM is that JIT is part of JVM itself and its main function is to improve the performance of JVM by directly compiling some hot code (code which executes above a certain threshold) into native instruction. JIT stands for Just In time compilation and JVM stands for Java Virtual Machine. JVM is a virtual machine used in Java programming platform to execute or run Java programs. The main advantage of JVM is that [JVM  makes Java platform-independent](http://java67.blogspot.com/2012/08/how-java-achieves-platform-independence.html) by executing bytecodes. Java source code is compiled into class files, which contains bytecode. These byte codes are then executed by JVM. Now here comes JIT. Since the execution of bytecode is slower than the execution of machine language code, because JVM first needs to translate bytecode into machine language code.

# [Can abstract class have Constructor in Java - Interview Question](http://www.java67.com/2013/02/can-abstract-class-have-constructor-in-java.html)

Yes, an abstract class can have a constructor in Java. You can either explicitly provide a constructor to abstract class or if you don't, the compiler will add [default constructor](http://java67.blogspot.com/2012/12/how-constructor-chaining-works-in-java.html) of no argument in abstract class. This is true for all classes and it also applies to an abstract class. For those who want to recall what is an abstract class in Java, it's a class which can not be instantiated with new() operator or any other ways. In order to use an abstract class in Java,  You need to extend it and provide a concrete class. Abstract class is commonly used to define a base class for a type hierarchy with default implementation, which is applicable to all child classes. By the way, [difference between interface and abstract class in Java](http://java67.blogspot.sg/2012/09/what-is-difference-between-interface-abstract-class-java.html) is also one of the popular and tricky Java questions and should be prepared well for Java interviews.

[Read more »](http://www.java67.com/2013/02/can-abstract-class-have-constructor-in-java.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[3 comments:](http://www.java67.com/2013/02/can-abstract-class-have-constructor-in-java.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=3495565766918344761)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=3495565766918344761)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3495565766918344761&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3495565766918344761&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3495565766918344761&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3495565766918344761&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3495565766918344761&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [Java Interview Question](http://www.java67.com/search/label/Java%20Interview%20Question?&max-results=3)

# [Difference between Set, List and Map in Java - Interview question](http://www.java67.com/2013/01/difference-between-set-list-and-map-in-java.html)

Set, List and Map are three important interface of Java collection framework and Difference between Set, List and Map in Java is one of the most frequently asked [Java Collection interview question](http://java67.blogspot.com/2012/09/java-collection-interview-questions.html). Some time this question is asked as When to use List, Set and Map in Java. Clearly, interviewer is looking to know that whether you are familiar with fundamentals of Java collection framework or not. In order to decide when to use List, Set or Map , you need to know what are these interfaces and what functionality they provide. [List in Java](http://java67.blogspot.com/2012/07/sort-list-ascending-descending-order-set-arraylist.html) provides ordered and indexed collection which may contain duplicates. Set provides an un-ordered collection of unique objects, i.e. Set doesn't allow duplicates, while Map provides a data structure based on key value pair and hashing. All three List, Set and Map are interfaces in Java and there are many concrete implementation of them are available in Collection API. ArrayList and LinkedList are two most popular used List implementation while [LinkedHashSet, TreeSet and HashSet](http://javarevisited.blogspot.com/2012/11/difference-between-treeset-hashset-vs-linkedhashset-java.html) are frequently used Set implementation. In this Java article we will see *difference between Map, Set and List in Java* and learn when to use List, Set or Map.

[Read more »](http://www.java67.com/2013/01/difference-between-set-list-and-map-in-java.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[26 comments:](http://www.java67.com/2013/01/difference-between-set-list-and-map-in-java.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=3814741245718182578)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=3814741245718182578)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3814741245718182578&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3814741245718182578&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3814741245718182578&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3814741245718182578&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3814741245718182578&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [Java collection tutorial](http://www.java67.com/search/label/Java%20collection%20tutorial?&max-results=3)

# [Difference between synchronized block and method in Java](http://www.java67.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html)

Synchronized block and synchronized methods are two ways to use [synchronized keyword in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) and implement mutual exclusion on critical section of code. Since Java is mainly used to write multi-threading programs,  which present various kinds of thread related issues like [thread-safety](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html), [deadlock](http://www.blogger.com/javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html) and [race conditions](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html), which plagues into code mainly because of poor understanding of synchronization mechanism provided by Java programming language. Java provides inbuilt synchronized and [volatile keyword](http://java67.blogspot.sg/2012/08/what-is-volatile-variable-in-java-when.html) to achieve synchronization in Java. Main *difference between synchronized method and synchronized block* is selection of lock on which critical section is locked. Synchronized method depending upon whether its a [static method](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) or non static locks on either class level lock or object lock. Class level lock is one for each class and represented by class literal e.g. Stirng.class. Object level lock is provided by current object e.g. this instance, You should [never mix static and non static synchronized method in Java](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html).. On the other hand synchronized block locks on monitor evaluated by expression provided as parameter to synchronized block. In next section we will see an example of both synchronized method and synchronized block to understand this difference better.

**How to use Volatile keyword in Java**  
What is volatile variable in Java and when to use  the volatile variable in Java is a famous [multi-threading interview question](http://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-answers.html) in Java interviews. Though many programmers know what is a volatile variable but they fail on second part i.e. where to use volatile variable in Java as it's not common to have a clear understanding and hands-on on volatile in Java. In this tutorial, we will address this gap by providing a simple example of the volatile variable in Java and discussing some when to use the volatile variable in Java. Anyway,  the volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from [main memory](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html). So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in an int or a boolean variable then  you can declare them as volatile variable.  
  
From Java 5 along with major changes like Autoboxing, Enum, Generics and Variable arguments , Java introduces some change in Java Memory Model (JMM), Which guarantees visibility of changes made from one thread to another also as "happens-before" which solves the problem of memory writes that happen in one thread can "leak through" and be seen by another thread.  
  
The Java volatile keyword cannot be used with method or class and it can only be used with a variable. Java volatile keyword also guarantees visibility and ordering, after Java 5 write to any volatile variable happens before any read into the volatile variable. By the way use of volatile keyword also prevents compiler or JVM from the reordering of code or moving away them from synchronization barrier.

## The Volatile variable Example in Java

To Understand example of volatile keyword in java let’s go back to [Singleton pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) and see [double checked locking in Singleton](http://javarevisited.blogspot.com/2014/05/double-checked-locking-on-singleton-in-java.html) with Volatile and without the volatile keyword in java.

/\*\*

\* Java program to demonstrate where to use Volatile keyword in Java.

\* In this example Singleton Instance is declared as volatile variable to ensure

\* every thread see updated value for \_instance.

\*

\* @author Javin Paul

\*/

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

**private** **static** **volatile** Singleton \_instance; //volatile variable

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

**if**(\_instance == **null**){

**synchronized**(Singleton.class){

**if**(\_instance == **null**)

\_instance = **new** Singleton();

}

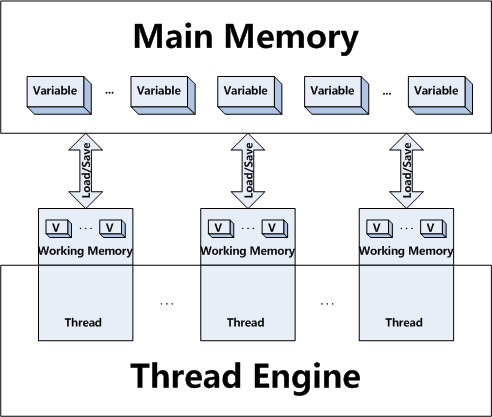
}

**return** \_instance;

}

If you look at the code carefully you will be able to figure out:  
1) We are only creating instance one time  
2) We are creating instance lazily at the time of the first request comes.

If we do not make the \_instance variable volatile than the Thread which is creating instance of Singleton is not able to communicate other thread, that instance has been created until it comes out of the Singleton block, so if Thread A is creating Singleton instance and just after creation lost the CPU, all other thread will not be able to see value of \_instance as not null and they will believe its still [null](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html).

[](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https://www.udemy.com/multithreading-and-parallel-computing-in-java/)

Why? because reader threads are not doing any locking and until writer thread comes out of synchronized block, memory will not be synchronized and value of \_instance will not be updated in main memory. With Volatile keyword in Java, this is handled by Java himself and such updates will be visible by all reader threads.  
  
So in Summary apart from [synchronized keyword in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), volatile keyword is also used to communicate the content of memory between threads.

### Let’s see another example of volatile keyword in Java

most of the time while writing game we use a variable bExit to check whether user has pressed exit button or not, value of this variable is updated in [event thread](http://javarevisited.blogspot.sg/2011/09/invokeandwait-invokelater-swing-example.html) and checked in game thread, So if we don't use volatile keyword with this variable, Game Thread might miss update from event handler thread if it's not synchronized in Java already. volatile keyword in java guarantees that value of the volatile variable will always be read from main memory and "*happens-before"* relationship in Java Memory model will ensure that content of memory will be communicated to different threads.

**private** **boolean** bExit**;**

**while(!**bExit**)** **{**

checkUserPosition**();**

updateUserPosition**();**

**}**

In this code example, One Thread (Game Thread) can cache the value of "bExit" instead of getting it from [main memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html) every time and if in between any other thread (Event handler Thread) changes the value; it would not be visible to this thread. Making boolean variable "bExit" as volatile in java ensures this will not happen.  
  
Also, If you have not read already then I also suggest you read the topic about volatile variable from [Java Concurrency in Practice](http://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20) book by Brian Goetz, one of the must read to truly understand this complex concept.

## When to use Volatile variable in Java

One of the most important thing in learning of volatile keyword is understanding when to use volatile variable in Java. Many [programmer](http://javarevisited.blogspot.sg/2011/06/top-programming-interview-questions.html) knows what is volatile variable and how does it work but they never really used volatile for any practical purpose. Here are couple of example to demonstrate when to use Volatile keyword in Java:

1) You can use Volatile variable if you want to read and write long and [double](http://javarevisited.blogspot.sg/2011/10/convert-double-to-string-example.html) variable atomically. long and double both are [64 bit](http://javarevisited.blogspot.sg/2012/01/find-jvm-is-32-or-64-bit-java-program.html) data type and by default writing of long and double is not atomic and platform dependence. Many platform perform write in long and double variable 2 step, writing 32 bit in each step, due to this its possible for a Thread to see 32 bit from two different write. You can avoid this issue by making long and double variable volatile in Java.  
  
  
2) A volatile variable can be used as an alternative way of achieving [synchronization in Java](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) in some cases, like Visibility. with volatile variable, it's guaranteed that all reader thread will see updated value of the volatile variable once write operation completed, without volatile keyword different reader thread may see different values.  
  
  
3) volatile variable can be used to inform the compiler that a particular field is subject to be accessed by multiple threads, which will prevent the compiler from doing any reordering or any kind of optimization which is not desirable in a multi-threaded environment. Without volatile variable compiler can re-order the code, free to cache value of volatile variable instead of always reading from main memory. like following example without volatile variable may result in an [infinite loop](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html)

**private** **boolean** isActive **=** thread**;**

**public** **void** printMessage**(){**

**while(**isActive**){**

System**.**out**.**println**(**"Thread is Active"**);**

**}**

**}**

without the *volatile modifier*, it's not guaranteed that one [Thread](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html) sees the updated value of isActive from other thread. The compiler is also free to cache value of isActive instead of reading it from main memory in every iteration. By making isActive a volatile variable you avoid these issue.  
  
  
4) Another place where a volatile variable can be used is to fixing double checked locking in Singleton pattern. As we discussed in [Why should you use Enum as Singleton](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) that double checked locking was broken in Java 1.4 environment.

### Important points on Volatile keyword in Java

1. The volatile keyword in Java is only application to a variable and using volatile keyword with class and method is illegal.  
  
2. volatile keyword in Java guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.  
  
3. In Java reads and writes are [atomic](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) for all variables declared using Java volatile keyword (including long and double variables).  
  
4. Using the volatile keyword in Java on variables reduces the risk of memory consistency errors because any write to a volatile variable in Java establishes a happens-before relationship with subsequent reads of that same variable.  
  
5. From Java 5 changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable in Java, it sees not just the [latest change to the volatile variable](http://java67.blogspot.sg/2012/08/what-is-volatile-variable-in-java-when.html) but also the side effects of the code that led up the change.  
  
6. Reads and writes are atomic for reference variables are for most primitive variables (all types except long and double) even without the use of volatile keyword in Java.  
  
7. An access to a volatile variable in Java never has a chance to block, since we are only doing a simple read or write, so unlike a synchronized block we will never hold on to any lock or wait for any [lock](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html).  
  
8. Java volatile variable that is an object reference may be null.  
  
9. Java volatile keyword doesn't mean atomic, its common misconception that after declaring volatile ++ will be atomic, to make the operation atomic you still need to ensure exclusive access using synchronized method or block in Java.  
  
10. If a variable is not shared between multiple threads, you don't need to use volatile keyword with that variable.

## Difference between synchronized and volatile keyword in Java

What is the difference between volatile and synchronized is another popular [core Java question](http://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html) asked on multi-threading and concurrency interviews. Remember volatile is not a replacement of synchronized keyword but can be used as an alternative in certain cases. Here are few differences between volatile and synchronized keyword in Java.  
  
1. The volatile keyword in Java is a field modifier while synchronized modifies code blocks and methods.  
  
2. Synchronized obtains and releases the lock on monitor’s Java volatile keyword doesn't require that.  
  
3. Threads in Java can be blocked for waiting for any monitor in case of synchronized, that is not the case with the [volatile keyword](http://java67.blogspot.com/2012/11/difference-between-transient-vs-volatile-modifier-variable-java.html) in Java.  
  
4. Synchronized method affects performance more than a volatile keyword in Java.  
  
5. Since volatile keyword in Java only synchronizes the value of one variable between Thread memory and "main" memory while synchronized synchronizes the value of all variable between thread memory and "main" memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile.  
  
6. You can not synchronize on the null object but your volatile variable in Java could be null.  
  
7. From Java 5 writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire

# Race Conditions and Critical Sections

* [Critical Sections](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#critical-sections)
* [Race Conditions in Critical Sections](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#race-conditions-in-critical-sections)
* [Preventing Race Conditions](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html)
* [Critical Section Throughput](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html#critical-section-throughput)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2016-09-11 |

A *race condition* is a special condition that may occur inside a critical section. A *critical section* is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition. The term race condition stems from the metaphor that the threads are racing through the critical section, and that the result of that race impacts the result of executing the critical section.

This may all sound a bit complicated, so I will elaborate more on race conditions and critical sections in the following sections.

## Critical Sections

Running more than one thread inside the same application does not by itself cause problems. The problems arise when multiple threads access the same resources. For instance the same memory (variables, arrays, or objects), systems (databases, web services etc.) or files.

In fact, problems only arise if one or more of the threads write to these resources. It is safe to let multiple threads read the same resources, as long as the resources do not change.

Here is a critical section Java code example that may fail if executed by multiple threads simultaneously:

public class Counter {

protected long count = 0;

public void add(long value){

this.count = this.count + value;

}

}

Imagine if two threads, A and B, are executing the add method on the same instance of the Counterclass. There is no way to know when the operating system switches between the two threads. The code in the add() method is not executed as a single atomic instruction by the Java virtual machine. Rather it is executed as a set of smaller instructions, similar to this:

1. Read this.count from memory into register.
2. Add value to register.
3. Write register to memory.

Observe what happens with the following mixed execution of threads A and B:

this.count = 0;

A: Reads this.count into a register (0)

B: Reads this.count into a register (0)

B: Adds value 2 to register

B: Writes register value (2) back to memory. this.count now equals 2

A: Adds value 3 to register

A: Writes register value (3) back to memory. this.count now equals 3

The two threads wanted to add the values 2 and 3 to the counter. Thus the value should have been 5 after the two threads complete execution. However, since the execution of the two threads is interleaved, the result ends up being different.

In the execution sequence example listed above, both threads read the value 0 from memory. Then they add their i ndividual values, 2 and 3, to the value, and write the result back to memory. Instead of 5, the value left in this.count will be the value written by the last thread to write its value. In the above case it is thread A, but it could as well have been thread B.

## Race Conditions in Critical Sections

The code in the add() method in the example earlier contains a critical section. When multiple threads execute this critical section, race conditions occur.

More formally, the situation where two threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions. A code section that leads to race conditions is called a critical section.

## Preventing Race Conditions

To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

Race conditions can be avoided by proper thread synchronization in critical sections. Thread synchronization can be achieved using a [**synchronized block of Java code**](http://tutorials.jenkov.com/java-concurrency/synchronized.html). Thread synchronization can also be achieved using other synchronization constructs like [**locks**](http://tutorials.jenkov.com/java-concurrency/locks.html) or atomic variables like [**java.util.concurrent.atomic.AtomicInteger**](http://tutorials.jenkov.com/java-concurrent-util/atomicinteger.html).

## Critical Section Throughput

For smaller critical sections making the whole critical section a synchronized block may work. But, for larger critical sections it may be beneficial to break the critical section into smaller critical sections, to allow multiple threads to execute each a smaller critical section. This may decrease contention on the shared resource, and thus increase throughput of the total critical section.

Here is a very simplified Java code example to show what I mean:

public class TwoSums {

private int sum1 = 0;

private int sum2 = 0;

public void add(int val1, int val2){

synchronized(this){

this.sum1 += val1;

this.sum2 += val2;

}

}

}

Notice how the add() method adds values to two different sum member variables. To prevent race conditions the summing is executed inside a Java synchronized block. With this implementation only a single thread can ever execute the summing at the same time.

However, since the two sum variables are independent of each other, you could split their summing up into two separate synchronized blocks, like this:

public class TwoSums {

private int sum1 = 0;

private int sum2 = 0;

private Integer sum1Lock = new Integer(1);

private Integer sum2Lock = new Integer(2);

public void add(int val1, int val2){

synchronized(this.sum1Lock){

this.sum1 += val1;

}

synchronized(this.sum2Lock){

this.sum2 += val2;

}

}

}

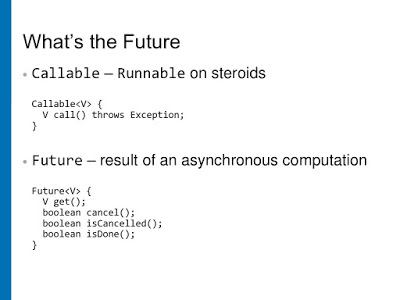
# [Difference between Callable and Runnable in Java - Interview question](http://www.java67.com/2013/01/difference-between-callable-and-runnable-java.html)

Difference between Callable and Runnable interface in Java is one of the interesting questions from my list of [Top 15 Java multi-threading questions](http://javarevisited.blogspot.it/2011/07/java-multi-threading-interview.html), and it’s also very popular in various Java Interviews. The Callable interface is newer than Runnable interface and added on Java 5 release along with other major changes e.g. [Generics](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html), [Enum](http://java67.blogspot.sg/2012/11/java-enum-example-with-constructor.html), [Static imports](http://javarevisited.blogspot.sg/2012/10/what-is-static-import-in-java-5-example-tutorial.html) and [variable argument method](http://javarevisited.blogspot.sg/2011/09/variable-argument-in-java5-varargs.html). Though both Callable and Runnable interface are designed to represent a task, which can be executed by any thread, there is some significant difference between them. In my opinion, the major difference between Callable and Runnable interface is that Callable can return the result of an operation performed inside call() method, which was one of the limitations with Runnable interface.

Callable interface was added in Java 5 to complement existing Runnable interface, which is used to wrap a task and pass it to a Thread or thread pool for asynchronous execution. Callable actually represent an asynchronous computation, whose value is available via Future object. All the code which needs to be executed asynchronously goes into call() method. Callable is also a single abstract method type (SAM type), so it can be used along with [lambda expression on Java 8](http://javarevisited.blogspot.sg/2014/02/10-example-of-lambda-expressions-in-java8.html). Both Callable and Future are parametric type and can be used to wrap classes like Integer, String or anything else. When you pass a Callable to thread pool, it choose one thread and execute the Callable. It immediately return a Future object which promises to hold result of computation once done. You can then call get() method of Future, which will return result of computation or block if Computation is not complete. If you don't like indefinite blocking then you can also use overloaded get() method with timeout. Future also allows you to cancel the task if its not started or interrupt if its started. We will see, how we can calculate factorial of large number using Callable and Future in Java. BTW, if you are serious about mastering concurrency API of Java, I suggest you to also take a look at one of the best book on the subject, [Java Concurrency in Practice](http://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20) by Brian Goetz. It is one of the book I keep refer whenever I have a doubt or want to refresh my knowledge.

### Callable vs Runnable

Many of you would be familiar with Runnable interface, one of the most popular way to use thread in Java, but you would be happy to know that Runnable is not the only way to create a task which can be executed by parallel threads. You can also use Callable interface to do the same. Main *difference between Runnable and Callable* is that Runnable cannot return any value back to caller but Callable can return value. Another difference is that call() method from Callable can also throw checked exception which was not possible by run() method of Runnable interface. See [here](http://java67.blogspot.sg/2013/01/difference-between-callable-and-runnable-java.html) to learn more about difference between Runnable and Callable in Java.

[](https://4.bp.blogspot.com/-EGpKPBfDJug/VYGQJR5jmiI/AAAAAAAADFs/oq5q2BgVtms/s1600/Callable+and+Future+Java.jpg)

## Callable and Future Example in Java

Here is our complete Java program to demonstrate *how you can use Callable and Future together* to implement asynchronous processing  in Java. Once you started using thread pool, Callable and Future, you don't need to wait for task to be completed, you can move on with other task and comeback to check whether task is completed or not. If task is finished then just get the result by calling get() method, but remember its a [blocking call](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html), so it will block if task is not finished.

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

/\*\*

\* Simple Java program to demonstrate how to use Callable and Future class in

\* Java. You can use FutureTask for asynchronous processing.

\*

\* @author WINDOWS 8

\*

\*/

public class HelloWorldApp {

public static void main(String... args) throws InterruptedException, ExecutionException {

// creating thread pool to execute task which implements Callable

ExecutorService es = Executors.newSingleThreadExecutor();

System.out.println("submitted callable task to calculate factorial of 10");

Future result10 = es.submit(new FactorialCalculator(10));

System.out.println("submitted callable task to calculate factorial of 15");

Future result15 = es.submit(new FactorialCalculator(15));

System.out.println("submitted callable task to calculate factorial of 20");

Future result20 = es.submit(new FactorialCalculator(20));

System.out.println("Calling get method of Future to fetch result of factorial 10");

long factorialof10 = result10.get();

System.out.println("factorial of 10 is : " + factorialof10);

System.out.println("Calling get method of Future to get result of factorial 15");

long factorialof15 = result15.get();

System.out.println("factorial of 15 is : " + factorialof15);

System.out.println("Calling get method of Future to get result of factorial 20");

long factorialof20 = result20.get();

System.out.println("factorial of 20 is : " + factorialof20);

}

}

class FactorialCalculator implements *Callable<Long>* {

private int number;

public FactorialCalculator(int number){

this.number = number;

}

@Override

public Long call() throws Exception {

return factorial(number);

}

private long factorial(int n) throws InterruptedException {

long result = 1;

while (n != 0) {

result = n \* result;

n = n - 1;

Thread.sleep(100);

}

return result;

}

}

Output

submitted callable task to calculate factorial of 10

submitted callable task to calculate factorial of 15

submitted callable task to calculate factorial of 20

Calling get method of Future to fetch result of factorial 10

factorial of 10 is : 3628800

Calling get method of Future to get result of factorial 15

factorial of 15 is : 1307674368000

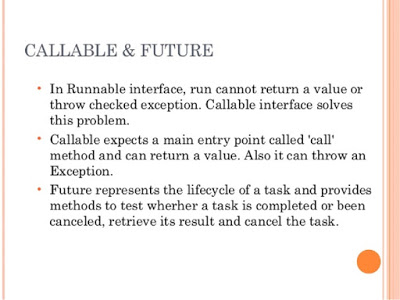
Calling get method of Future to get result of factorial 20

factorial of 20 is : 2432902008176640000

When you run this program, you will see that first 4 lines will be printed immediately because submit() is a non blocking method, it just takes a task and [returns a Future object](http://javarevisited.blogspot.in/2015/01/how-to-use-future-and-futuretask-in-Java.html), it doesn't wait until task is completed. That's why you see that all three tasks to calculate factorials are submitted immediately, but they are not done yet. When our code calls get() on first Future object its blocked until the calculation is done, that's why you will see the fifth line printing after sometime. For next two lines also same story because when you call get() method it will block until result is available. BTW, you don't need to block, you can even use isDone() method to check if calculation is completed or not before calling get method.

### Important points about Callable and Future

1) Callable is a SAM type interface, so it can be used in lambda expression.  
  
2) Callable has just one method call() which holds all the code needs to executed asynchronously.  
  
3) In Runnable interface, there was no way to return the result of computation or throw checked exception but with Callable you can both return a value and can throw [checked exception](http://java67.blogspot.sg/2012/12/difference-between-runtimeexception-and-checked-exception.html).  
  
4) You can use get() method of Future to retrieve result once computation is done. You can check if computation is finished or not by using isDone() method.  
  
5) You can cancel the computation by using Future.cancel() method.  
  
6) get() is a blocking call and it blocks until computation is completed.

[](https://4.bp.blogspot.com/-3iDp1Cf_8vA/VYGQKU_h5JI/AAAAAAAADF0/CjHRI0uNPAY/s1600/Callable+and+Future+Example+in+Java.jpg)

That's all about **how to use Callable and Future object in Java**. You can wrap asynchronous computation inside call() method and pass it to a single thread or thread pool for execution. you don't need to wait until execution complete, your thread can carry on with future object returned by call method. Once computation is done you can query the future object and get the result back.

# [10 AWT Swing Interview Questions Answers in Java](http://www.java67.com/2013/01/10-awt-swing-interview-questions-answers-java.html)

AWT and Swing Interview Questions are part of any Java interviews which involves GUI development work. Since AWT(Abstract Windows Toolkit) and Swing offers most popular Java GUI solutions, Yes, JavaFX  has still lot of path to cover in terms of popularity and usability of Swing. In Investment banking, where Java rules on server side applicationdevelopment, Swing was heavily used in GUI development couple of years back. Now trends is shifting towards C# for GUI development due to couple of reasons e.g. good Swing developers are not easy to be found as compared to C# developers. By the way there are still many development work is going on Swing. One of the reason Swing developers are high in demand is because Swing is quickly becoming a niche technology. This is surprising because  Swing is part of Java, but given steep learning curve of Swing to master different layouts and components e.g. JTable or [JList](http://javarevisited.blogspot.sg/2011/08/java-swing-tutorial-jlist-example.html) , I tend to believe it. In this article we will see some good Swing Interview questions for practice. Questions like [InvokeLater vs InvokeAndWait are](http://javarevisited.blogspot.sg/2011/09/invokeandwait-invokelater-swing-example.html)classic, which is always worth preparing.

[Read more »](http://www.java67.com/2013/01/10-awt-swing-interview-questions-answers-java.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[1 comment:](http://www.java67.com/2013/01/10-awt-swing-interview-questions-answers-java.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=6921595902857291933)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=6921595902857291933)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=6921595902857291933&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=6921595902857291933&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=6921595902857291933&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=6921595902857291933&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=6921595902857291933&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [swing](http://www.java67.com/search/label/swing?&max-results=3)

# [How Constructor Chaining works in Java - Example](http://www.java67.com/2012/12/how-constructor-chaining-works-in-java.html)

How to call one constructor from another constructor in Java or What is Constructor  to call one constructor from another constructor of the same class if you want to call a constructor from based class or super class then you can use super keyword. Calling one constructor from other is called **Constructor chaining in Java**. Constructors can call each other automatically or explicitly using this() and super() keywords. this() denotes a [no-argument constructor](http://javarevisited.blogspot.sg/2012/12/what-is-constructor-in-java-example-chainning-overloading.html) of the same class and super() denotes a no argument or default constructor of parent class. Also having multiple constructors in the same class is known as [constructor overloading in Java](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html).

[Read more »](http://www.java67.com/2012/12/how-constructor-chaining-works-in-java.html#more)

Read more: <http://www.java67.com/search/label/core%20java%20interview%20question%20answer?updated-max=2013-02-20T04:57:00-08:00&max-results=6#ixzz4i1YNsgdj>

# [Producer Consumer Problem with Wait and Notify Example](http://www.java67.com/2012/12/producer-consumer-problem-with-wait-and-notify-example.html)

Producer Consumer Problem is a classical concurrency problem and in fact it is one of the concurrency design pattern. In last article we have seen solving [Producer Consumer problem in Java using blocking Queue](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html) but one of my reader emailed me and requested code example and explanation of solving Producer Consumer problem in Java  with [wait and notify method](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html)as well, Since its often asked as one of the top [coding question in Java](http://java67.blogspot.sg/2012/08/10-java-coding-interview-questions-and.html). In this Java tutorial, I have put the code example of wait notify version of earlier producer consumer concurrency design pattern. You can see this is much longer code with explicit handling blocking conditions like when shared queue is full and when queue is empty. Since we have replaced[BlockingQueue](http://javarevisited.blogspot.sg/2012/12/blocking-queue-in-java-example-ArrayBlockingQueue-LinkedBlockingQueue.html) with Vector we need to implement blocking using [wait and notify](http://javarevisited.blogspot.sg/2012/02/why-wait-notify-and-notifyall-is.html) and that's why we have introduced produce(int i) and consume() method. If you see I have kept consumer thread little slow by allowing it to sleep for 50 Milli second to give an opportunity to producer to fill the queue, which helps to understand that Producer thread is also waiting when Queue is full.

[Read more »](http://www.java67.com/2012/12/producer-consumer-problem-with-wait-and-notify-example.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[21 comments:](http://www.java67.com/2012/12/producer-consumer-problem-with-wait-and-notify-example.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=7312900353389350784)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=7312900353389350784)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=7312900353389350784&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=7312900353389350784&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=7312900353389350784&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=7312900353389350784&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=7312900353389350784&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [java concurrency tutorial](http://www.java67.com/search/label/java%20concurrency%20tutorial?&max-results=3), [thread](http://www.java67.com/search/label/thread?&max-results=3)

# [When to use ArrayList vs LinkedList in Java](http://www.java67.com/2012/12/difference-between-arraylist-vs-LinkedList-java.html)

ArrayList and LinkedList are two popular concrete implementations of List interface from Java's popular Collection framework. Being List implementation both ArrayList and LinkedList are ordered, the index based and allows duplicate. Despite being from same type hierarchy there are a lot of differences between these two classes which makes them popular among Java interviewers. The main difference between ArrayList vs LinkedList is that former is backed by an array while later is based upon linked list data structure, which makes the performance of add(), remove(), contains() and iterator() different for both ArrayList and LinkedList. The difference between ArrayList and LinkedList is also an important  Java collection interview questions, as much popular as [Vector vs ArrayList](http://java67.blogspot.sg/2012/09/arraylist-vs-vector-in-java-interview.html) or [HashMap vs HashSet in Java](http://java67.blogspot.sg/2012/08/difference-between-hashset-and-hashmap.html). Sometimes this is also asked as **When to use LinkedList** and **When to use ArrayList in Java**. In this Java collection tutorial, we will compareLinkedList vs ArrayList on the various parameter which will help us to decide [when to use ArrayList over LinkedList in Java](http://javarevisited.blogspot.sg/2012/02/difference-between-linkedlist-vs.html).

[Read more »](http://www.java67.com/2012/12/difference-between-arraylist-vs-LinkedList-java.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[12 comments:](http://www.java67.com/2012/12/difference-between-arraylist-vs-LinkedList-java.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=4322207026608371273)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=4322207026608371273)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4322207026608371273&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4322207026608371273&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4322207026608371273&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4322207026608371273&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4322207026608371273&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [Java collection tutorial](http://www.java67.com/search/label/Java%20collection%20tutorial?&max-results=3), [programming](http://www.java67.com/search/label/programming?&max-results=3)

# [JDBC Interview questions answers in Java - 2 to 4 years experienced programmer](http://www.java67.com/2012/12/jdbc-interview-questions-answers-in-Java-2-4-years-experienced.html)

JDBC Interview question forms one of the important section in Java Interviews. Similar to [multithreading](http://java67.blogspot.sg/2012/08/5-thread-interview-questions-answers-in.html), [Collection framework](http://javarevisited.blogspot.sg/2011/11/collection-interview-questions-answers.html) and [Garbage collection interview question](http://javarevisited.blogspot.sg/2012/10/10-garbage-collection-interview-question-answer.html), JDBC question must be prepared by any Java programmer. Most of questions from JDBC or Java database connectivity comes from API and basic architecture of JDBC which also involves JDBC drivers. A good understanding of JDBC API along with database basics like [transactions](http://javarevisited.blogspot.sg/2011/11/database-transaction-tutorial-example.html) also help to do well in JDBC interviews. I have collected some of *frequently asked JDBC Interview question* for quick reference. This will help to revise some important JDBC concepts and also give a chance to explore JDBC API to newcomers. If you have any other JDBC interview question, which has been asked to you or friends, and you think it’s good to share among Java community then please share with us. Let's see my 11 questions from JDBC, not so tough but worth preparing.

[Read more »](http://www.java67.com/2012/12/jdbc-interview-questions-answers-in-Java-2-4-years-experienced.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[3 comments:](http://www.java67.com/2012/12/jdbc-interview-questions-answers-in-Java-2-4-years-experienced.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=4776056681638330066)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=4776056681638330066)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4776056681638330066&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4776056681638330066&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4776056681638330066&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4776056681638330066&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=4776056681638330066&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [JDBC](http://www.java67.com/search/label/JDBC?&max-results=3), [sql](http://www.java67.com/search/label/sql?&max-results=3), [SQL interview Question](http://www.java67.com/search/label/SQL%20interview%20Question?&max-results=3)

# [Difference between RuntimeException and checked Exception in Java](http://www.java67.com/2012/12/difference-between-runtimeexception-and-checked-exception.html)

**RuntimeException vs Checked Exception in Java**

Java Exceptions are divided in two categories RuntimeException also known as [unchecked Exception](http://javarevisited.blogspot.com/2011/12/checked-vs-unchecked-exception-in-java.html) and checked Exception. *Main difference between RuntimeException and checked Exception* is that, It is mandatory to provide try catch or [try finally block](http://javarevisited.blogspot.sg/2012/11/difference-between-final-finally-and-finalize-java.html) to handle checked Exception and failure to do so will result in compile time error, while in case ofRuntimeException this is not mandatory. Difference between checked and unchecked exception is one of the most popular question on Java [interview for 2 to years experienced](http://java67.blogspot.sg/2012/10/java-interview-questions-for-2-to-3-4-years-experienced.html) developer especially related to Exception concepts. Answer to this question is rather similar as mentioned in previous lines and they are mostly asked along with other Java Exception interview questions like [difference between throw and throws](http://java67.blogspot.sg/2012/10/difference-between-throw-vs-throws-in.html) and [Error vs Exception](http://java67.blogspot.sg/2012/12/difference-between-error-vs-exception.html). Any Exception which is subclass of RuntimeException are called unchecked and mandatory exception handling is not requirement for them.

[Read more »](http://www.java67.com/2012/12/difference-between-runtimeexception-and-checked-exception.html#more)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[1 comment:](http://www.java67.com/2012/12/difference-between-runtimeexception-and-checked-exception.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=3118299439057674677)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=3118299439057674677)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3118299439057674677&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3118299439057674677&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3118299439057674677&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3118299439057674677&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=3118299439057674677&target=pinterest)

Labels: [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [exception](http://www.java67.com/search/label/exception?&max-results=3)

# [How to reverse String in Java with or without StringBuffer Example](http://www.java67.com/2012/12/how-to-reverse-string-in-java-stringbuffer-stringbuilder.html)

**Reverse String in Java**

There are many ways to reverse String in Java. You can use rich Java API to quickly reverse contents of any String object. Java library provides [StringBuffer and StringBuilder](http://javarevisited.blogspot.ca/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) class with reverse() method which can be used to reverse String in Java. Since converting between String and StringBuffer or StringBuilder is very easy it's the most easy wayavailable to reverse String in Java. At the same time Writing Java program to reverse String in Java without StringBuffer is one of the popular [Java String interview question](http://javarevisited.blogspot.sg/2012/10/10-java-string-interview-question-answers-top.html), which requires you to reverse String by applying logic and by not using API methods. Since reverse is a recursive job, you can use recursion as well as loop to reverse String in Java. In this Java tutorial I have shown **How to reverse String using StringBuffer, StringBuilder** and using **pure loop with logic**. You can also check [How to reverse String with recursion in Java](http://javarevisited.blogspot.ca/2012/01/how-to-reverse-string-in-java-using.html), if you want to see recursive code. let's see complete Java program for this beautiful Java programming exercise.

## Java program to reverse String in Java

[Java program to reverse String in Java without StringBuffer or StringBuilder](http://3.bp.blogspot.com/-1lzFJzIgaHk/UF2Ci6kY5pI/AAAAAAAAAes/OYiM7r-DHzc/s1600/17.jpg)Here is my complete code program to reverse any String in Java. In [main method](http://java67.blogspot.sg/2012/08/what-is-main-method-in-java-why-main-is.html) we have first used StringBuffer and StringBuilder to *reverse contents of String* and then we wrote our own logic to reverse String. This uses toCharArray() method of String class which return [character array of String](http://javarevisited.blogspot.ca/2012/03/why-character-array-is-better-than.html). By looping through character array and appending it into empty String we can get reversed String in Java, as shown in following example.

/\*\*  
 \*  
 \* **Java program to reverse String in Java**. There are multiple ways to reverse  
 \* String in Java, you can either take help of standard Java API StringBuffer  
 \* to reverse String in Java. StringBuffer has a reverse() method which return StringBuffer  
 \* with reversed contents. On the other hand you can also reverse it by applying your  
 \* own logic, if asked to reverse String without using StringBuffer in Java. By the way  
 \* you can also use StringBuilder to reverse String in Java. StringBuilder is non thread-safe  
 \* version of StringBuffer and provides similar API. You can use StringBuilder's reverse()  
 \* method to reverse content and then convert it back to String  
 \*  
 \* @author http://java67.blogspot.com  
 \*/  
**public** **class** StringReverseExample {  
    
    
    **public** **static** **void** main(String args[]) {  
        
        *//quick wasy to reverse String in Java - Use StringBuffer*  
        String word = "HelloWorld";  
        String reverse = **new** StringBuffer(word).reverse().toString();  
        System.out.printf(" original String : %s , reversed String %s  %n", word, reverse);  
        
        *//another quick to reverse String in Java - use StringBuilder*  
        word = "WakeUp";  
        reverse = **new** StringBuilder(word).reverse().toString();  
        System.out.printf(" original String : %s , reversed String %s %n", word, reverse);  
        
        *//one way to reverse String without using StringBuffer or StringBuilder is writing*

*//own utility method*  
        word = "Band";  
        reverse = reverse(word);  
        System.out.printf(" original String : %s , reversed String %s %n", word, reverse);  
    }     
    
    
    **public** **static** String reverse(String source){  
        **if**(source == **null** || source.isEmpty()){  
            **return** source;  
        }         
        String reverse = "";  
        **for**(**int** i = source.length() -1; i>=0; i--){  
            reverse = reverse + source.charAt(i);  
        }  
        
        **return** reverse;  
    }  
      
}  
  
**Output:**  
original String : HelloWorld , reversed String dlroWolleH   
original String : WakeUp , reversed String pUekaW  
original String : Band , reversed String dnaB

That's all on **How to reverse String in Java with and without StringBuffer and StringBuilder**. Though being a Java programmer I prefer to use library and suggest any one to use StringBuffer or StringBuilder to reverse String for any production use. Though its also a [good programming exercise](http://javarevisited.blogspot.ca/2011/06/top-programming-interview-questions.html) and you should practice it before going for any Java programming interview.

Other **Java tutorials** you may like

[String matches examples in Java](http://java67.blogspot.sg/2012/09/java-string-matches-example-regular-expression.html)

[Difference between String and StringBuffer in Java](http://java67.blogspot.sg/2012/08/difference-between-string-and-stringbuffer-in-java.html)

[Best way to convert numbers to String in Java](http://java67.blogspot.sg/2012/10/best-way-to-convert-numbers-to-string-in-java-example.html)

[Difference between == and equals method in Java](http://java67.blogspot.sg/2012/11/difference-between-operator-and-equals-method-in.html)

[How to create Enum from String in Java](http://java67.blogspot.sg/2012/08/how-to-create-enum-from-string-in-java.html)

[How to use contains and indexOf method in String Java](http://java67.blogspot.sg/2012/10/string-contains-and-indexof-example.html)

Posted by [Javin Paul](https://plus.google.com/114528699166048052030)[43 comments:](http://www.java67.com/2012/12/how-to-reverse-string-in-java-stringbuffer-stringbuilder.html#comment-form)[[http://img1.blogblog.com/img/icon18_email.gif](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=5120298873160712092)](https://www.blogger.com/email-post.g?blogID=694855878384792308&postID=5120298873160712092)

[Email This](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=5120298873160712092&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=5120298873160712092&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=5120298873160712092&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=5120298873160712092&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=694855878384792308&postID=5120298873160712092&target=pinterest)

Labels: [coding](http://www.java67.com/search/label/coding?&max-results=3), [Coding Problems](http://www.java67.com/search/label/Coding%20Problems?&max-results=3), [core java](http://www.java67.com/search/label/core%20java?&max-results=3), [core java interview question answer](http://www.java67.com/search/label/core%20java%20interview%20question%20answer?&max-results=3), [programming](http://www.java67.com/search/label/programming?&max-results=3)

# [Difference between Error vs Exception in Java - Interview question](http://www.java67.com/2012/12/difference-between-error-vs-exception.html)

Both Error and Exception are derived from java.lang.Throwable in Java but main difference between Error and Exception is kind of error they represent. java.lang.Error represent errors which are generally can not be handled and usually refer catastrophic failure e.g. running out of System resources, some examples of Error in Java are[java.lang.OutOfMemoryError](http://javarevisited.blogspot.sg/2011/09/javalangoutofmemoryerror-permgen-space.html) or [Java.lang.NoClassDefFoundError](http://java67.blogspot.sg/2012/08/what-is-noclassdeffounderror-in-java.html) and [java.lang.UnSupportedClassVersionError](http://java67.blogspot.sg/2012/10/how-to-fix-javalangunsupportedclassversionerror-major-minor-version-49-50-51.html). On the other hand java.lang.Exception represent errors which can be catch and dealt e.g. IOException which comes while performing I/O operations i.e. [reading files and directories](http://javarevisited.blogspot.ca/2011/12/read-and-write-text-file-java.html). Clear understanding of Error and Exception is must for any serious Java programmer and good programming and debugging skills are required to overcome issues which caused Error and Exception in Java. Apart from its must have knowledge in Java application development, **difference between Error and Exception** is also a popular questions on Java interviews related to Exception handling, similar to [difference between throw and throws in Java](http://java67.blogspot.sg/2012/10/difference-between-throw-vs-throws-in.html). In this Java article we will briefly see major difference between Error and Exception in Java which include both syntactical and logical difference.

Important Questions and Anseres

**Question 1: What’s wrong using HashMap in the multi-threaded environment? When does the get() method go to an infinite loop?** ([answer](http://java67.blogspot.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html))

Well, nothing is wrong, depending on how you use it. For example, if you initialize the HashMap just by one thread and then all threads are only reading from it, then it’s perfectly fine. One example of this is a Map which contains configuration properties.

The real problem starts when at-least one of that thread is updating HashMap i.e. adding, changing or removing any key value pair. Since put() operation can cause re-sizing and which can further lead to infinite loop, that’s why either you should use [Hashtable](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html) or [ConcurrentHashMap](http://javarevisited.blogspot.com/2013/02/concurrenthashmap-in-java-example-tutorial-working.html), later is better.

**Question 2. Does overriding the hashCode() method have any performance implication?** ([answer](http://java67.blogspot.com/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html))

This is a good question and open to all, as per my knowledge a poor hash code function will result in the frequent collision in HashMap which eventually increases the time for adding an object into Hash Map.

From Java 8 onwards though, collision will not impact performance as much as it does in earlier versions, because after a threshold the linked list will be replaced by the binary tree, which will give you O(logN) performance in the worst case, as compared to O(n) of linked list.

**Question 3: Do all properties of an Immutable Object need to be final?** ([answer](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html))

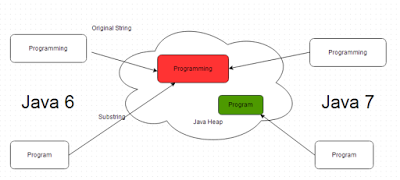
Not necessarily, as stated above you can achieve same functionality by making the member non-final but private and not modifying them except in a constructor. Don’t provide setter methods for them and if it is a mutable object, then don’t ever leak any reference for that member.

Remember making a reference variable final, only ensures that it will not be reassigned to a different value, but you can still change individual properties of object, pointed by that reference variable. This is one of the key point, Interviewer like to hear from candidates.

**Question 4: How does the substring() method inside String works?** ([answer](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html))

Another good Java interview question, I think the answer is not sufficient, but here it is “Substring creates a new object out of source string by taking a portion of original string”.

This question was mainly asked to see if the developer is familiar with the risk of memory leaks, which a sub-string can create. Until Java 1.7, substring holds the reference of the original character array, which means even a sub-string of 5 character long, can prevent 1GB character array from garbage collection, by holding a strong reference.

[](http://www.javacodegeeks.com/wp-content/uploads/2015/11/How-SubString-works-in-Java.png)

This issue is fixed in Java 1.7, where the original character array is not referenced anymore, but that change also made the creation of substring a bit more costly in terms of time. Earlier it was on the range of O(1), which could be O(n) in worst case on Java 7.

**Question**\*\*  5: Can you write a critical section code for the singleton?\*\* ([answer](http://javarevisited.blogspot.sg/2014/05/double-checked-locking-on-singleton-in-java.html))

This core Java question is a followup of the previous question and expecting the candidate to write Java singleton using double checked locking. Remember to use the volatile variable to make Singleton thread-safe.

**Question 6: How do you handle error condition while writing stored procedure or accessing stored procedure from java?** ([answer](http://javarevisited.blogspot.com/2013/04/spring-framework-tutorial-call-stored-procedures-from-java.html))

This is one of the tough Java interview questions and its open for all, my friend didn’t know the answer so he didn’t mind telling me. My take is that stored procedure should return an error code if some operation fails but if stored procedure itself fails than catching SQLException is the only choice.

**Question 7 : What is difference between Executor.submit() and Executer.execute() methods ?** ([answer](http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html))

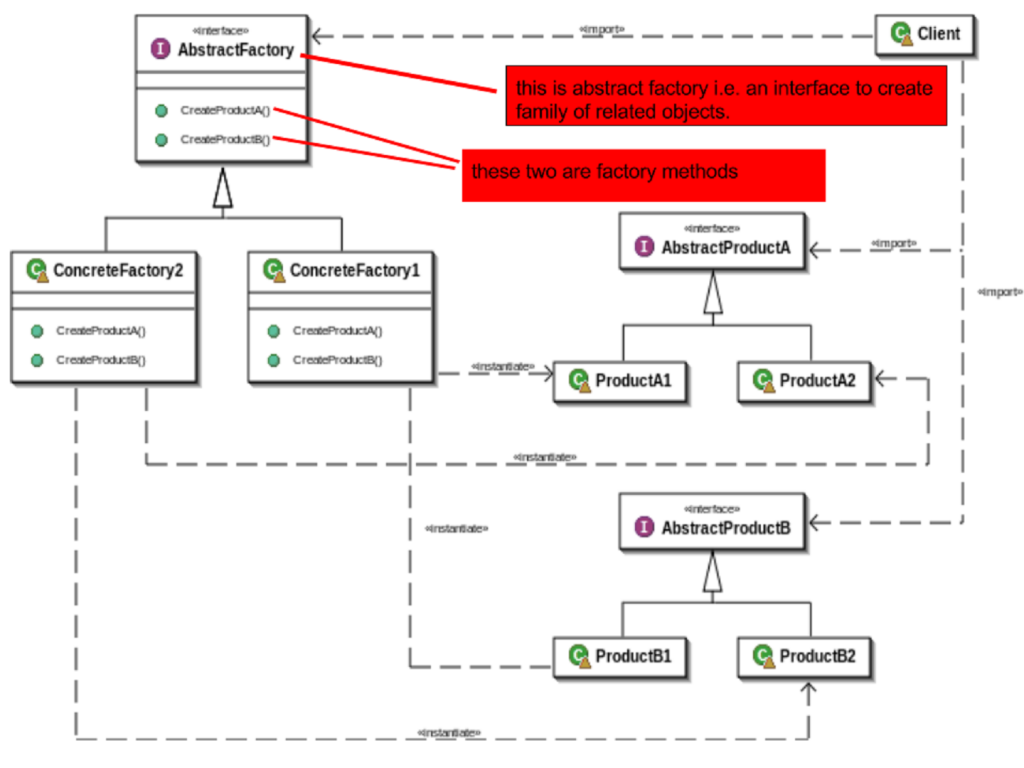
This question is from my list of Top 15 Java multi-threading question answers. It’s getting popular day by day because of huge demand of Java developers with good concurrency skills. The answer is that former returns an object of Future which can be used to find result from worker thread.

There is a difference when looking at exception handling. If your tasks throw an exception and if it was submitted with executing this exception will go to the uncaught exception handler (when you don’t have provided one explicitly, the default one will just print the stack trace to System.err).

If you submitted the task with submit any thrown exception, [checked exception](http://javarevisited.blogspot.sg/2011/12/checked-vs-unchecked-exception-in-java.html) or not, is then part of the task’s return status. For a task that was submitted with submitting and that terminates with an exception, the Future.get() will re-throw this exception, wrapped in an ExecutionException.

**Question 8:  What is the difference between factory and abstract factory pattern?**([answer](http://javarevisited.blogspot.sg/2013/01/difference-between-factory-and-abstract-factory-design-pattern-java.html))

Abstract Factory provides one more level of abstraction. Consider different factories each extended from an Abstract Factory and responsible for the creation of different hierarchies of objects based on the type of factory. E.g. AbstractFactory extended by AutomobileFactory, UserFactory, RoleFactory etc. Each individual factory would be responsible for the creation of objects in that genre. Here is UML diagram of factory and abstract factory pattern:

[](http://www.javacodegeeks.com/wp-content/uploads/2015/11/factory-vs-abstract-factory-pattern.png)

**Question 9: What is a Singleton? Is it better to make the whole method synchronized or only critical section synchronized?** ([answer](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html))

Singleton in Java is a class with just one instance in the whole Java application, for example, java.lang.Runtime is a Singleton class. Creating Singleton was tricky prior Java 4 but once Java 5 introduced Enum its very easy.

**Question 10:  Can you write code for iterating over HashMap in Java 4 and Java 5?**([answer](http://java67.blogspot.com/2014/05/3-examples-to-loop-map-in-java-foreach.html))

Tricky one but he managed to write using while and a for loop. Actually there are four ways to iterate over any Map in Java, one involves using keySet() and iterating over key and then using get() method to retrieve values, which is bit expensive.

Second method involves using entrySet() and iterating over them either by using for each loop or while with Iterator.hasNext() method. This one is a better approach because both key and value objects are available to you during Iteration and you don’t need to call the get() method for retrieving value, which could give O(n) performance in case of huge linked list at one bucket. See my post [4 ways to iterate over Map in Java](http://javarevisited.blogspot.com/2011/12/how-to-traverse-or-loop-hashmap-in-java.html) for detailed explanation and code examples.

**Question 11 : When do you override hashCode() and equals()?** ([answer](http://javarevisited.blogspot.com/2013/08/10-equals-and-hashcode-interview.html))

Whenever necessary, especially if you want to do equality check based upon business logic rather than object equality, e.g. two employee objects are equal if they have the same emp\_id, despite the fact that they are two different objects, created by different part of the code.

Also overriding both these methods are must if you want to use them as key in HashMap. Now as part of the equals-hashcode contract in Java, when you override equals, you must override hashcode as well, otherwise your object will not break invariant of classes e.g. Set, Map which relies on equals() method for functioning properly.

You can also check my post [5 tips on equals in Java](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) to understand subtle issue which can arise while dealing with these two methods.

**Question 12 :. What will be the problem if you don’t override hashCode() method ?**([answer](http://java67.blogspot.sg/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html))

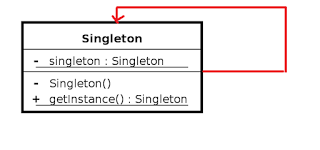
If you don’t override the equals method, then the contract between equals and hashcode will not work, according to which, two objects which are equal by equals() must have the same hashcode. In this case, an other object may return different hashCode and will be stored on that location, which breaks invariant of HashMap class, because they are not supposed to allow duplicate keys.

When you add object using put() method, it iterate through all Map.Entry object present in that bucket location, and update value of previous mapping, if Map already contains that key. This will not work if hashcode is not overridden.

**Question 13 : Is it better to synchronize critical sections of getInstance() method or the whole getInstance() method?** ([answer](http://javarevisited.blogspot.com/2014/05/double-checked-locking-on-singleton-in-java.html))

The answer is only the critical section, because if we lock the whole method, then every time some some one call this method, it will have to wait even though we are not creating any object. In other words, synchronization is only needed, when you create object, which happens only once.

Once object has created, there is no need for any synchronization. In fact, that’s very poor coding in terms of performance, as synchronized method reduce performance up to 10 to 20 times. Here is UML diagram of Singleton pattern:

[](http://www.javacodegeeks.com/wp-content/uploads/2015/11/Singleton-design-Pattern-Java.png)

By the way, there are several ways to create a thread-safe singleton in Java, which you can also mention as part of this question or any follow-up.

**Question 14: Where does equals() and hashCode() method comes in the picture during the get() operation?** ([answer](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html))

This core Java interview question is a follow-up of previous Java question and the candidate should know that once you mention hashCode, people are most likely ask, how they are used in HashMap. When you provide a key object, first it’s hashcode method is called to calculate bucket location. Since a bucket may contain more than one entry as linked list, each of those Map.Entry object is evaluated by using equals() method to see if they contain the actual key object or not.

**Questions 15: How do you avoid a deadlock in Java?** ([answer](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html))

You can avoid deadlock by breaking the circular wait condition. In order to do that, you can make an arrangement in the code to impose the ordering on acquisition and release of locks.

If lock will be acquired in a consistent order and released in just opposite order, there would not be a situation where one thread is holding a lock which is acquired by other and vice-versa. See the detailed answer for the code example and more detailed explanation.

**Question 16:  What is the difference between creating String as new() and literal?**([answer](http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html))

When we create the string with new() Operator, it’s created in heap and not added into string pool while String created using literal are created in String pool itself which exists in PermGen area of heap.

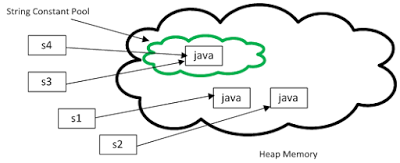
**String** str = **new** **String**("Test")

does not put the object str in String pool, we need to call String.intern() method which is used to put them into String pool explicitly. It’s only when you create String object as a String literal e.g. String s = "Test" that Java automatically puts that into the String pool.

By the way there is a catch here Since we are passing arguments as “Test”, which is a String literal, it will also create another object as “Test” on string pool. This is the one point, which has gone unnoticed until knowledgeable readers of Javarevisited blog suggested it.

To learn more about the difference between String literal and String object, see [this](http://java67.com/2014/08/difference-between-string-literal-and-new-String-object-Java.html)article.

Here is a nice image which shows this difference quite well:

[](http://www.javacodegeeks.com/wp-content/uploads/2015/11/String-literal-vs-String-Object-in-Java.png)

**Question 17: What is an Immutable Object? Can you write an Immutable Class?** ( [answer](http://javarevisited.blogspot.in/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html))

Immutable classes are Java classes whose objects can not be modified once created. Any modification in Immutable object results in the new object. For example, [String is immutable in Java](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html).

Mostly Immutable classes are also final in Java, in order to prevent sub classes from overriding methods, which can compromise Immutability. You can achieve the same functionality by making member as non-final but private and not modifying them except in constructor.

Apart form obvious, you also need to make sure that, you should not expose the internals of Immutable object, especially if it contains a mutable member. Similarly, when you accept the value for the mutable member from client e.g. java.util.Date, use [clone() method](http://javarevisited.blogspot.sg/2013/09/how-clone-method-works-in-java.html) keep a separate copy for yourself, to prevent the risk of malicious client modifying mutable reference after setting it.

The Same precaution needs to be taken while returning value for a mutable member, return another separate copy to the client, never return original reference held by Immutable class. You can see my post How to create an Immutable class in Java for step by step guide and code examples.

**Question 18: Give the simplest way to find out the time a method takes for execution without using any profiling tool?** ([answer](http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html))

Read the system time just before the method is invoked and immediately after thr method returns. Take the time difference, which will give you the time taken by a method for execution.

To put it in code…

long start = System.currentTimeMillis ();

**method** ();

long **end** = System.currentTimeMillis (); System.out.println (“Time taken **for** execution **is** ” + (**end** – start));

Remember that if the time taken for execution is too small, it might show that it is taking zero milliseconds for execution. Try it on a method which is big enough, in the sense the one which is doing considerable amount of processing

**Question 19: Which two methods you need to implement to use an Object as key in HashMap?** ([answer](http://javarevisited.blogspot.com/2013/01/difference-between-identityhashmap-and-hashmap-java.html))

In order to use any object as Key in HashMap or Hashtable, it must implement equals and hash-code methods in Java. Read How HashMap works in Java for a detailed explanation on how equals and hash code method is used to put and get an object from HashMap.

**Question 20: How would you prevent a client from directly instantiating your concrete classes? For example, you have a Cache interface and two implementation classes MemoryCache and DiskCache, How do you ensure there is no object of this two classes is created by client using new() keyword.**

I leave this question for you to practice and think about before I give the answer. I am sure you can figure out the right way to do this, as this is one of the important decision to keep control of classes in your hand, great from a maintenance perspective.

I am also very grateful to my readers who have generously contributed several good questions from Java Interviews for both beginners and experienced developers alike. I have already answered many of these questions in this blog and you can easily find a relevant post by using the search box at the top right corner of this page.

### Multithreading, Concurrency and Thread basics Questions

**1) Can we make array volatile in Java?**  
This is one of the tricky Java multi-threading questions you will see in senior Java developer Interview. Yes, you can make an array volatile in Java but only the reference which is pointing to an array, not the whole array. What I mean, if one thread changes the reference variable to points to another array, that will provide a volatile guarantee, but if multiple threads are changing individual array elements they won't be having happens before guarantee provided by the volatile modifier.  
  
  
**2) Can volatile make a non-atomic operation to atomic?**  
This another good question I love to ask on volatile, mostly as a follow-up of the previous question. This question is also not easy to answer because volatile is not about atomicity, but there are cases where you can use a volatile variable to make the operation atomic.  
  
One example I have seen is having a long field in your class. If you know that a long field is accessed by more than one thread e.g. a counter, a price field or anything, you better make it volatile. Why? because reading to a long variable is not atomic in Java and done in two steps, If one thread is writing or updating long value, it's possible for another thread to see half value (fist 32-bit). While reading/writing a volatile long or double (64 bit) is atomic.  
  
  
  
**3) What are practical uses of volatile modifier?**  
One of the practical use of the volatile variable is to make reading double and long atomic. Both double and long are 64-bit wide and they are read in two parts, first 32-bit first time and next 32-bit second time, which is non-atomic but volatile double and long read is atomic in Java. Another use of the volatile variable is to provide a memory barrier, just like it is used in Disrupter framework. Basically, Java Memory model inserts a write barrier after you write to a volatile variable and a read barrier before you read it. Which means, if you write to volatile field then it's guaranteed that any thread accessing that variable will see the value you wrote and anything you did before doing that right into the thread is guaranteed to have happened and any updated data values will also be visible to all threads, because the memory barrier flushed all other writes to the cache.  
  
  
**4) What guarantee volatile variable provides?**([answer](http://java67.blogspot.sg/2012/08/what-is-volatile-variable-in-java-when.html))  
volatile variables provide the guarantee about ordering and visibility e.g. volatile assignment cannot be re-ordered with other statements but in the absence of any synchronization instruction compiler, JVM or JIT are free to reorder statements for better performance. volatile also provides the happens-before guarantee which ensures changes made in one thread is visible to others. In some cases volatile also provide atomicity e.g. reading 64-bit data types like long and double are not atomic but read of volatile double or long is atomic.  
  
  
  
**5) Which one would be easy to write? synchronization code for 10 threads or 2 threads?**  
In terms of writing code, both will be of same complexity because synchronization code is independent of a number of threads. Choice of synchronization though depends upon a number of threads because the number of thread present more contention, so you go for advanced synchronization technique e.g. lock stripping, which requires more complex code and expertise.  
  
  
**6) How do you call wait() method? using if block or loop? Why?**([answer](http://javarevisited.blogspot.sg/2015/07/how-to-use-wait-notify-and-notifyall-in.html))  
wait() method should always be called in loop because it's possible that until thread gets CPU to start running again the condition might not hold, so it's always better to check condition in loop before proceeding. Here is the standard idiom of using wait and notify method in Java:

// The standard idiom for using the wait method

synchronized (obj) {

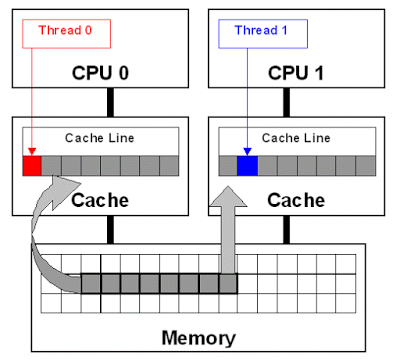
while (condition does not hold)

obj.wait(); // (Releases lock, and reacquires on wakeup)

... // Perform action appropriate to condition

}

See [Effective Java Item 69](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20) to learn more about why wait method should call in the loop.  
  
  
**7)  What is false sharing in the context of multi-threading?**  
false sharing is one of the well-known performance issues on multi-core systems, where each process has its local cache. false sharing occurs when threads on different processor modify variables that reside on same cache line as shown in the following image:

[](https://2.bp.blogspot.com/-Tze9foqpb74/VepwCzXHGCI/AAAAAAAADtM/i4KQDaefqk4/s1600/False+Sharing+in+Multi-threaded+application.gif)

False sharing is very hard to detect because the thread may be accessing completely different global variables that happen to be relatively close together in memory. Like many concurrency issues, the primary way to avoid false sharing is careful code review and aligning your data structure with the size of a cache line.  
  
  
**8) What is busy spin? Why should you use it?**  
Busy spin is one of the technique to wait for events without releasing CPU. It's often done to avoid losing data in CPU cached which is lost if the thread is paused and resumed in some other core. So, if you are working on low latency system where your order processing thread currently doesn't have any order, instead of sleeping or calling wait(), you can just loop and then again check the queue for new messages. It's only beneficial if you need to wait for a very small amount of time e.g. in micro seconds or nano seconds. [LMAX Disrupter](http://lmax-exchange.github.io/disruptor/) framework, a high-performance inter-thread messaging library has a BusySpinWaitStrategy which is based on this concept and uses a busy spin loop for EventProcessors waiting on the barrier.  
  
  
**9) How do you take thread dump in Java?**  
You can take a thread dump of Java application in Linux by using **kill -3 PID**, where PID is the process id of Java process. In Windows, you can press **Ctrl + Break**. This will instruct JVM to print thread dump in standard out or err and it could go to console or log file depending upon your application configuration. If you have used Tomcat then when  
  
  
  
**10) is Swing thread-safe?**([answer](http://javarevisited.blogspot.sg/2013/08/why-swing-is-not-thread-safe-in-java-Swingworker-Event-thread.html))  
No, Swing is not thread-safe. You cannot update Swing components e.g. JTable, JList or JPanel from any thread, in fact, they must be updated from GUI or AWT thread. That's why swings provide invokeAndWait() and invokeLater() method to request GUI update from any other threads. This methods put update request in AWT threads queue and can wait till update or return immediately for an asynchronous update. You can also check the detailed answer to learn more.  
  
  
**11) What is a thread local variable in Java?** ([answer](http://javarevisited.blogspot.sg/2012/05/how-to-use-threadlocal-in-java-benefits.html))  
Thread-local variables are variables confined to a thread, its like thread's own copy which is not shared between multiple threads. Java provides a ThreadLocal class to support thread-local variables. It's one of the many ways to achieve thread-safety. Though be careful while using thread local variable in manged environment e.g. with web servers where worker thread out lives any application variable. Any thread local variable which is not removed once its work is done can potentially cause a memory leak in Java application.  
  
  
**12) Write wait-notify code for producer-consumer problem?** ([answer](http://java67.blogspot.sg/2012/12/producer-consumer-problem-with-wait-and-notify-example.html))  
Please see the answer for a code example. Just remember to call wait() and notify() method from synchronized block and test waiting for condition on the loop instead of if block.  
  
  
**13) Write code for thread-safe Singleton in Java?** ([answer](http://javarevisited.blogspot.in/2012/12/how-to-create-thread-safe-singleton-in-java-example.html))  
Please see the answer for a code example and step by step guide to creating thread-safe singleton class in Java. When we say thread-safe, which means Singleton should remain singleton even if initialization occurs in the case of multiple threads. Using Java enum as Singleton class is one of the easiest ways to create a thread-safe singleton in Java.  
  
  
**14) The difference between sleep and wait in Java?**([answer](http://java67.blogspot.sg/2012/08/what-are-difference-between-wait-and.html))  
Though both are used to pause currently running thread, sleep() is actually meant for short pause because it doesn't release lock, while wait() is meant for conditional wait and that's why it release lock which can then be acquired by another thread to change the condition on which it is waiting.  
  
  
**15) What is an immutable object? How do you create an Immutable object in Java?** ([answer](http://javarevisited.blogspot.sg/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html))  
Immutable objects are those whose state cannot be changed once created. Any modification will result in a new object e.g. String, Integer, and other wrapper class. Please see the answer for step by step guide to creating Immutable class in Java.  
  
  
**16) Can we create an Immutable object, which contains a mutable object?**  
Yes, its possible to create an Immutable object which may contain a mutable object, you just need to be a little bit careful not to share the reference of the mutable component, instead, you should return a copy of it if you have to. Most common example is an Object which contain the reference of java.util.Date object.

## Date types and Basic Java Interview Questions

**17) What is the right data type to represent a price in Java?**([answer](http://javarevisited.blogspot.sg/2012/02/java-mistake-1-using-float-and-double.html))  
BigDecimal if memory is not a concern and Performance is not critical, otherwise double with predefined precision.  
  
  
**18) How do you convert bytes to String?** ([answer](http://javarevisited.blogspot.sg/2014/08/2-examples-to-convert-byte-array-to-String-in-Java.html))  
you can convert bytes to the string using string constructor which accepts byte[], just make sure that right character encoding otherwise platform's default character encoding will be used which may or may not be same.  
  
  
**19) How do you convert bytes to long in Java?** (answer)  
This questions if for you to answer :-)  
  
  
**20) Can we cast an int value into byte variable? what will happen if the value of int is larger than byte?**  
Yes, we can cast but int is 32 bit long in java while byte is 8 bit long in java so when you cast an int to byte higher 24 bits are lost and a byte can only hold a value from -128 to 128.  
  
  
**21) There are two classes B extends A and C extends B, Can we cast B into C e.g. C = (C) B;**([answer](http://javarevisited.blogspot.sg/2012/12/what-is-type-casting-in-java-class-interface-example.html))  
  
  
**22) Which class contains clone method? Cloneable or Object?** ([answer](http://javarevisited.blogspot.sg/2015/01/java-clone-tutorial-part-2-overriding-with-mutable-field-example.html))  
java.lang.Cloneable is marker interface and doesn't contain any method clone method is defined in the object class. It is also knowing that clone() is a native method means it's implemented in C or C++ or any other native language.  
  
  
**23) Is ++ operator is thread-safe in Java?** (answer)  
 No it's not a thread safe operator because its involve multiple instructions like reading a value, incriminating it and storing it back into memory which can be overlapped between multiple threads.  
  
  
**24) Difference between a = a + b and a += b ?** (answer)  
The += operator implicitly cast the result of addition into the type of variable used to hold the result. When you add two integral variable e.g. variable of type byte, short, or int then they are first promoted to int and them addition happens. If result of addition is more than maximum value of a then a + b will give compile time error but a += b will be ok as shown below

byte a = 127;

byte b = 127;

b = a + b; *// error : cannot convert from int to byte*

b += a; *// ok*

**25) Can I store a double value in a long variable without casting?**([answer](http://java67.blogspot.com/2014/11/how-to-convert-double-to-long-in-java-example.html))  
No, you cannot store a double value into a long variable without casting because the range of double is more  that long and you we need to type cast. It's not dificult to answer this question but many develoepr get it wrong due to confusion on which one is bigger between double and long in Java.  
  
  
**26) What will this return 3\*0.1 == 0.3? true or false?**(answer)  
This is one of the really tricky questions. Out of 100, only 5 developers answered this question and only of them have explained the concept correctly. The short answer is false because some floating point numbers can not be represented exactly.  
  
  
**27) Which one will take more memory, an int or Integer?**(answer)  
An Integer object will take more memory an Integer is the an object and it  store meta data overhead about the object and int is primitive type so its takes less space.  
  
  
**28) Why is String Immutable in Java?** ([answer](http://java67.blogspot.sg/2014/01/why-string-class-has-made-immutable-or-final-java.html))  
One of my favorite Java interview question. The String is Immutable in java because java designer thought that string will be heavily used and making it immutable allow some optimization easy sharing same String object between multiple clients. See the link for the more detailed answer. This is a great question for Java programmers with less experience as it gives them food for thought, to think about how things works in Java, what Jave designers might have thought when they created String class etc.  
  
**29) Can we use String in the switch case?** ([answer](http://javarevisited.blogspot.sg/2011/08/string-switch-case-jdk7-example.html))  
Yes from Java 7 onward we can use String in switch case but it is just syntactic sugar. Internally string hash code is used for the switch. See the detaiedl answer for more explanation and discussion.  
  
**30) What is constructor chaining in Java?** ([answer](http://java67.blogspot.sg/2012/12/how-constructor-chaining-works-in-java.html))  
When you call one constructor from other than it's known as constructor chaining in Java. This happens when you have multiple, overloaded constructor in the class.

### JVM Internals and Garbage Collection Interview Questions

In the year 2018 I have seen increased focus on JVM internal and Garbage collection tuning, monitoring Java appliation, dealing with Java performance issues on various Java interviews. This is actually become the prime topic for interviewing any experienced Java developer for senior position e.g. technical lead, VP or team lead. If you feel you are short of experience and knowledge in this area then you should read atleast one book mentioned in my list of [Java Performance books](http://javarevisited.blogspot.com/2014/07/top-5-java-performance-tuning-books.html). I vote goes to Java Performance, The Definitive guide by Scott.

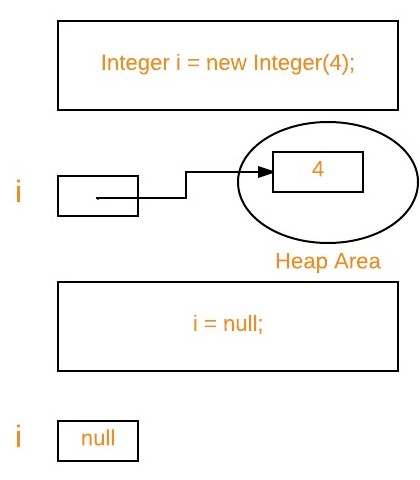
# Garbage Collection in Java

**Introduction**

* In C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects. Due to this negligence, at certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing **OutOfMemoryErrors**.
* But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.
* Garbage collector is best example of [Daemon thread](https://www.geeksforgeeks.org/daemon-thread-java/) as it is always running in background.
* Main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**.

**Important terms :**

1. **Unreachable objects :**An object is said to be unreachable iff it doesn’t contain any reference to it. Also note that objects which are part of [island of isolation](https://www.geeksforgeeks.org/island-of-isolation-in-java/) are also unreachable.
2. Integer i = new Integer(4);
3. // the new Integer object is reachable via the reference in 'i'
4. i = null;
5. // the Integer object is no longer reachable.



1. **Eligibility for garbage collection :**An object is said to be eligible for GC(garbage collection) iff it is unreachable. In above image, after i = null; integer object 4 in heap area is eligible for garbage collection.

**Ways to make an object eligible for GC**

* Even though programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.
* There are generally four different ways to make an object eligible for garbage collection.
  + 1. Nullifying the reference variable
    2. Re-assigning the reference variable
    3. Object created inside method
    4. [Island of Isolation](https://www.geeksforgeeks.org/island-of-isolation-in-java/)

All above ways with examples are discussed in separate article : [How to make object eligible for garbage collection](https://www.geeksforgeeks.org/how-to-make-object-eligible-for-garbage-collection/)

**Ways for requesting**[**JVM**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)**to run Garbage Collector**

* Once we made object eligible for garbage collection, it may not destroy immediately by garbage collector. Whenever JVM runs Garbage Collector program, then only object will be destroyed. But when JVM runs Garbage Collector, we can not expect.
* We can also request JVM to run Garbage Collector. There are two ways to do it :
  + 1. **Using System.gc() method** : System class contain static method gc() for requesting JVM to run Garbage Collector.
    2. **Using Runtime.getRuntime().gc() method** : [Runtime class](https://www.geeksforgeeks.org/java-lang-runtime-class-in-java/) allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate requesting  // JVM to run Garbage Collector  public class Test  {      public static void main(String[] args) throws InterruptedException      {          Test t1 = new Test();          Test t2 = new Test();            // Nullifying the reference variable          t1 = null;            // requesting JVM for running Garbage Collector          System.gc();            // Nullifying the reference variable          t2 = null;            // requesting JVM for running Garbage Collector          Runtime.getRuntime().gc();        }        @Override      // finalize method is called on object once      // before garbage collecting it      protected void finalize() throws Throwable      {          System.out.println("Garbage collector called");          System.out.println("Object garbage collected : " + this);      }  } |

Output:

Garbage collector called

Object garbage collected : Test@46d08f12

Garbage collector called

Object garbage collected : Test@481779b8

**Note :**

* + 1. There is no guarantee that any one of above two methods will definitely run Garbage Collector.
    2. The call System.gc() is effectively equivalent to the call : Runtime.getRuntime().gc()

**Finalization**

* Just before destroying an object, Garbage Collector calls finalize() method on the object to perform cleanup activities. Once finalize() method completes, Garbage Collector destroys that object.
* finalize() method is present in [Object class](https://www.geeksforgeeks.org/object-class-in-java/) with following prototype.
* protected void finalize() throws Throwable

Based on our requirement, we can override finalize() method for perform our cleanup activities like closing connection from database.

**Note :**

* 1. The finalize() method called by Garbage Collector not [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/). Although Garbage Collector is one of the module of JVM.
  2. [Object class](https://www.geeksforgeeks.org/object-class-in-java/) finalize() method has empty implementation, thus it is recommended to override finalize() method to dispose of system resources or to perform other cleanup.
  3. The finalize() method is never invoked more than once for any given object.
  4. If an uncaught exception is thrown by the finalize() method, the exception is ignored and finalization of that object terminates.

For examples on finalize() method, please see [Output of Java programs | Set 10 (Garbage Collection)](https://www.geeksforgeeks.org/output-of-java-programs-set-10-garbage-collection/)

# [Types Of References In Java : Strong, Soft, Weak And Phantom](https://javaconceptoftheday.com/types-of-references-in-java-strong-soft-weak-and-phantom/)

One of the beauty of the Java language is that it doesn’t put burden of memory management on the programmers. Java automatically manages the memory on the behalf of the programmers. Java programmers need not to worry about freeing the memory after the objects are no more required. **Garbage Collector Thread** does this for you. This thread is responsible for sweeping out unwanted objects from the memory. But, you have no control over garbage collector thread. You can’t make it to run whenever you want. It is up to JVM which decides when to run garbage collector thread. But, with the introduction of **java.lang.ref** classes, you can have little control over when your objects will be garbage collected.

Depending upon how objects are garbage collected, references to those objects in java are grouped into 4 types. They are,

1) Strong References

2) Soft References

3) Weak References

4) Phantom References

Let’s discuss these reference types in detail.

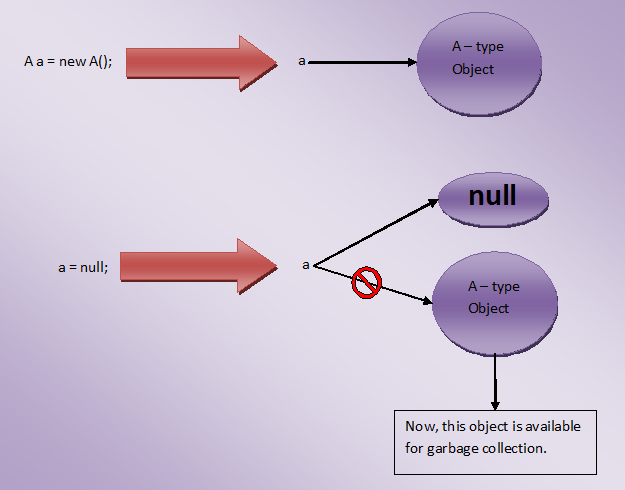
## 1) Strong References

These type of references we use daily while writing the code. Any object in the memory which has active **strong reference** is not eligible for garbage collection. For example, in the below program, reference variable **‘a’** is a strong reference which is pointing to class A-type object. At this point of time, this object can’t be garbage collected as it has strong reference.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class A  {      //Class A  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            a = null;    //Now, object to which 'a' is pointing earlier is eligible for garbage collection.      }  } |

If you make reference **‘a’** to point to null like in Line 12, then, object to which ‘a’ is pointing earlier will become eligible for garbage collection. Because, it will have no active references pointing to it. This object is most likely to be garbage collected when garbage collector decides to run.

Look at the below picture for more precise understanding.



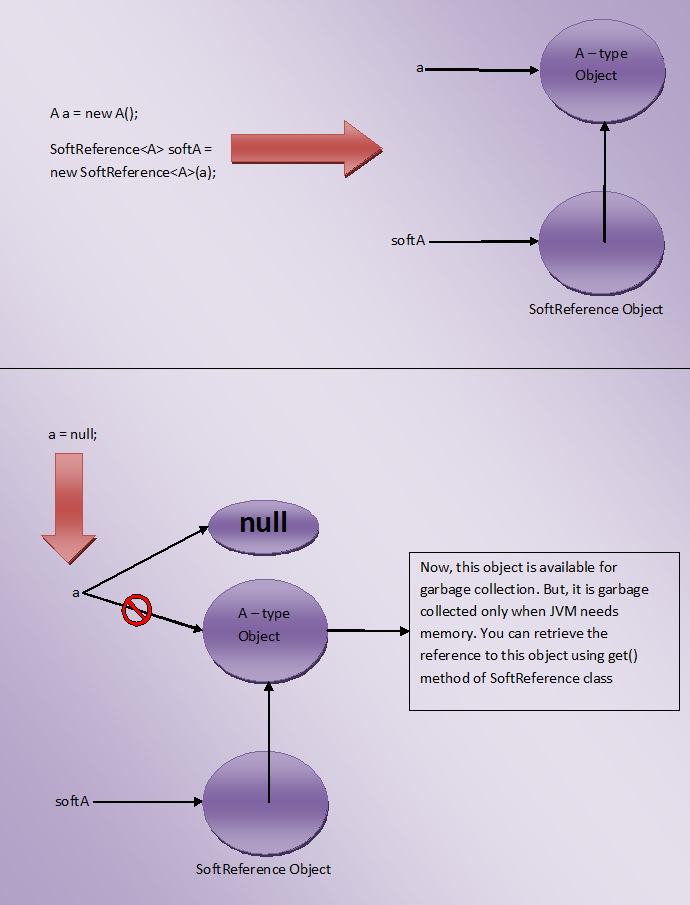
## 2) Soft References

The objects which are softly referenced will not be garbage collected (even though they are available for garbage collection) until JVM badly needs memory. These objects will be cleared from the memory only if JVM runs out of memory. You can create a soft reference to an existing object by using **java.lang.ref.SoftReference** class. Below is the code example on how to create a soft reference.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating Soft Reference to A-type object to which 'a' is also pointing            SoftReference<A> softA = new SoftReference<A>(a);            a = null;    //Now, A-type object to which 'a' is pointing earlier is eligible for garbage collection. But, it will be garbage collected only when JVM needs memory.            a = softA.get();    //You can retrieve back the object which has been softly referenced      }  } |

In the above example, you create two strong references – ‘**a**‘ and ‘**softA**‘. ‘a’ is pointing to A-type object and ‘softA’ is pointing to SoftReference type object. This SoftReference type object is internally referring to A-type object to which ‘a’ is also pointing. When ‘a’ is made to point to null, object to which ‘a’ is pointing earlier becomes eligible for garbage collection. But, it will be garbage collected only when JVM needs memory. Because, it is softly referenced by ‘softA’ object.

Look at the below picture for more clarity.



One more use of SoftReference class is that you can retrieve back the object which has been softly referenced. It will be done by using **get()** method. This method returns reference to the object if object is not cleared from the memory. If object is cleared from the memory, it will return null.

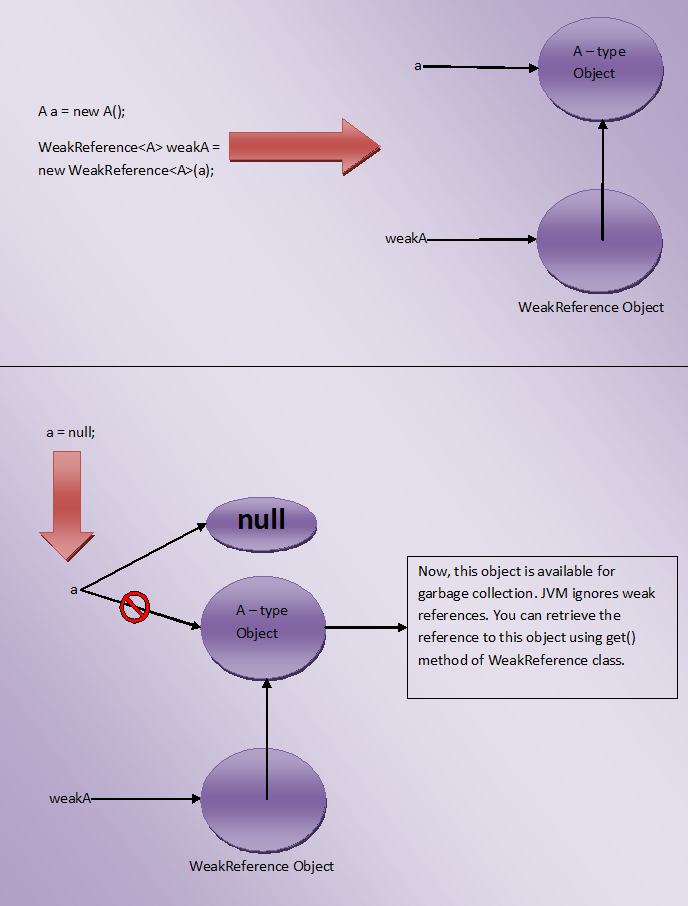
## 3) Weak References

JVM ignores the **weak references**. That means objects which has only week references are eligible for garbage collection. They are likely to be garbage collected when JVM runs garbage collector thread. JVM doesn’t show any regard for weak references.

Below is the code which shows how to create weak references.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating Weak Reference to A-type object to which 'a' is also pointing.            WeakReference<A> weakA = new WeakReference<A>(a);            a = null;    //Now, A-type object to which 'a' is pointing earlier is available for garbage collection.            a = weakA.get();    //You can retrieve back the object which has been weakly referenced.      }  } |

Look at the below picture for more clear understanding.



You may think that what is the use of creating weak references if they are ignored by the JVM, Use of weak reference is that you can retrieve back the weakly referenced object if it is not yet removed from the memory. This is done using get() method of WeakReference class. It will return reference to the object if object is not yet removed from the memory.

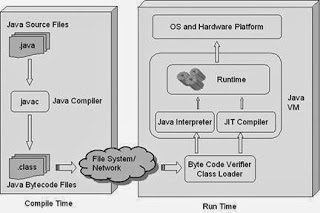
## 4) Phantom References

The objects which are being referenced by **phantom references** are eligible for garbage collection. But, before removing them from the memory, JVM puts them in a queue called **‘reference queue’**. They are put in a reference queue after calling finalize() method on them. You can’t retrieve back the objects which are being phantom referenced. That means calling get() method on phantom reference always returns null.

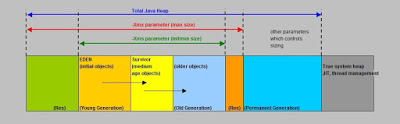
Below example shows how to create Phantom References.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating ReferenceQueue            ReferenceQueue<A> refQueue = new ReferenceQueue<A>();            //Creating Phantom Reference to A-type object to which 'a' is also pointing            PhantomReference<A> phantomA = new PhantomReference<A>(a, refQueue);            a = null;    //Now, A-type object to which 'a' is pointing earlier is available for garbage collection. But, this object is kept in 'refQueue' before removing it from the memory.            a = phantomA.get();    //it always returns null      }  } |

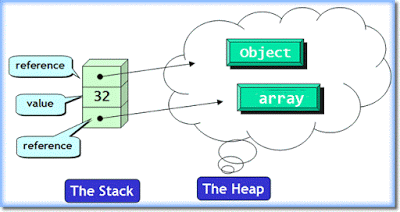
**31) What is the size of int in 64-bit JVM?**  
The size of an int variable is constant in Java, it's always 32-bit irrespective of platform. Which means the size of primitive int is same in both 32-bit and 64-bit Java virtual machine.  
  
**32) The difference between Serial and Parallel Garbage Collector?** ([answer](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html))  
Even though both the serial and parallel collectors cause a stop-the-world pause during Garbage collection. The main difference between them is that a serial collector is a default copying collector which uses only one GC thread for garbage collection while a parallel collector uses multiple GC threads for garbage collection.  
  
**33) What is the size of an int variable in 32-bit and 64-bit JVM?**(answer)  
The size of int is same in both 32-bit and 64-bit JVM, it's always 32 bits or 4 bytes.  
  
**34) A difference between WeakReference and SoftReference in Java?**([answer](http://javarevisited.blogspot.sg/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html))  
Though both WeakReference and SoftReference helps garbage collector and memory efficient, WeakReference becomes eligible for garbage collection as soon as last strong reference is lost but SoftReference even thought it can not prevent GC, it can delay it until JVM absolutely need memory.  
  
**35) How do WeakHashMap works?**(answer)  
WeakHashMap works like a normal HashMap but uses WeakReference for keys, which means if the key object doesn't have any reference then both key/value mapping will become eligible for garbage collection.  
  
**36) What is -XX:+UseCompressedOops JVM option? Why use it?**([answer](http://javarevisited.blogspot.com/2012/06/what-is-xxusecompressedoops-in-64-bit.html))  
When you go migrate your Java application from 32-bit to 64-bit JVM, the heap requirement suddenly increases, almost double, due to increasing size of ordinary object pointer from 32 bit to 64 bit. This also adversely affect how much data you can keep in CPU cache, which is much smaller than memory. Since main motivation for moving to 64-bit JVM is to specify large heap size, you can save some memory by using compressed OOP. By using -XX:+UseCompressedOops, JVM uses 32-bit OOP instead of 64-bit OOP.  
  
  
**37) How do you find if JVM is 32-bit or 64-bit from Java Program?**([answer](http://javarevisited.blogspot.sg/2012/01/find-jvm-is-32-or-64-bit-java-program.html))  
You can find that by checking some system properties like sun.arch.data.model or os.arch  
  
  
**38) What is the maximum heap size of 32-bit and 64-bit JVM?**([answer](http://javarevisited.blogspot.sg/2013/04/what-is-maximum-heap-size-for-32-bit-64-JVM-Java-memory.html))  
Theoretically, the maximum heap memory you can assign to a 32-bit JVM is 2^32 which is 4GB but practically the limit is much smaller. It also varies between operating systems e.g. form 1.5GB in Windows to almost 3GB in Solaris. 64-bit JVM allows you to specify larger heap size, theoretically 2^64 which is quite large but practically you can specify heap space up to 100GBs. There are even JVM e.g. Azul where heap space of 1000 gigs is also possible.  
  
  
**39) What is the difference between JRE, JDK, JVM and JIT?** ([answer](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html))  
JRE stands for Java run-time and it's required to run Java application. JDK stands for Java development kit and provides tools to develop Java program e.g. Java compiler. It also contains JRE. The JVM stands for Java virtual machine and it's the process responsible for running Java application. The JIT stands for Just In Time compilation and helps to boost the performance of Java application by converting Java byte code into native code when the crossed certain threshold i.e. mainly hot code is converted into native code.

[](https://2.bp.blogspot.com/-ls3yC0U7ouo/VhDqX-3OUbI/AAAAAAAAD40/Zcsc5uCaGq0/s1600/JVM+JRE+JDK.jpg)

**40) Explain Java Heap space and Garbage collection?** ([answer](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html))  
When a Java process is started using java command, memory is allocated to it. Part of this memory is used to create heap space, which is used to allocate memory to objects whenever they are created in the program. Garbage collection is the process inside JVM which reclaims memory from dead objects for future allocation.

[](https://3.bp.blogspot.com/-DqV12_uIeZ4/VhDqtPCVIVI/AAAAAAAAD48/uqWZB0BgZUI/s1600/java_heaps_memory.jpg)

**41) Can you guarantee the garbage collection process?**(answer)  
No, you cannot guarantee the garbage collection, though you can make a request using System.gc() or Runtime.gc() method.  
  
  
**42) How do you find memory usage from Java program? How much percent of the heap is used?**  
You can use memory related methods from java.lang.Runtime class to get the free memory, total memory and maximum heap memory in Java.  By using these methods, you can find out how many percents of the heap is used and how much heap space is remaining. Runtime.freeMemory() return amount of free memory in bytes, Runtime.totalMemory() returns total memory in bytes and Runtime.maxMemory() returns maximum memory in bytes.  
  
  
**43) What is the difference between stack and heap in Java?**([answer](http://javarevisited.blogspot.com/2013/01/difference-between-stack-and-heap-java.html))  
Stack and heap are different memory areas in the JVM and they are used for different purposes. The stack is used to hold method frames and local variables while objects are always allocated memory from the heap. The stack is usually much smaller than heap memory and also didn't shared between multiple threads, but heap is shared among all threads in JVM.

[](https://1.bp.blogspot.com/-NZeVo83YJAA/VhDrDO0oWtI/AAAAAAAAD5E/mEek8Ll7NfU/s1600/Difference+between+stack+and+heap+memory+in+Java.gif)

## Basic Java concepts Interview Questions

**44) What's the difference between "a == b" and "a.equals(b)"?** ([answer](http://javarevisited.blogspot.sg/2012/12/difference-between-equals-method-and-equality-operator-java.html))  
The a = b does object reference matching if both a and b are an object and only return true if both are pointing to the same object in the heap space, on the other hand, a.equals(b) is used for logical mapping and its expected from an object to override this method to provide logical equality. For example, String class overrides this equals() method so that you can compare two Strings, which are the different object but contains same letters.  
  
  
**45) What is a.hashCode() used for? How is it related to a.equals(b)?**([answer](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html))  
hashCode() method returns an int hash value corresponding to an object. It's used in hash based collection classes e.g Hashtable, HashMap, LinkedHashMap and so on. It's very tightly related to equals() method. According to Java specification, two objects which are equal to each other using equals() method must have same hash code.  
  
  
**46) Difference between final, finalize and finally?**([answer](http://javarevisited.blogspot.sg/2012/11/difference-between-final-finally-and-finalize-java.html))  
The final is a modifier which you can apply to variable, methods and classes. If you make a variable final it means its value cannot be changed once initialized. finalize is a method, which is called just before an object is a garbage collected, giving it last chance to resurrect itself, but the call to finalize is not guaranteed. finally is a keyword which is used in exception handling along with try and catch. the finally block is always executed irrespective of whether an exception is thrown from try block or not.  
  
  
**47) What is a compile time constant in Java? What is the risk of using it?**  
public static final variables are also known as a compile time constant, the public is optional there. They are replaced with actual values at compile time because compiler know their value up-front and also knows that it cannot be changed during run-time. One of the problem with this is that if you happened to use a public static final variable from some in-house or third party library and their value changed later than your client will still be using old value even after you deploy a new version of JARs. To avoid that, make sure you compile your program when you upgrade dependency JAR files.

## Java Collections Framework Interview Questions

It also contains Data structure and algorithm Interview question in Java, questions on array, linked list, HashMap, ArrayList, Hashtable, Stack, Queue, PriorityQueue, LinkedHashMap and ConcurrentHashMap.  
  
**48) The difference between List, Set, Map, and Queue in Java?** ([answer](http://java67.blogspot.sg/2013/01/difference-between-set-list-and-map-in-java.html))  
The list is an ordered collection which allows duplicate. It also has an implementation which provides constant time index based access, but that is not guaranteed by List interface. Set is unordered collection which  
  
  
**49) Difference between poll() and remove() method?**  
Both poll() and remove() take out the object from the Queue but if poll() fails then it returns null but if remove fails it throws Exception.  
  
  
**50) The difference between LinkedHashMap and PriorityQueue in Java?**([answer](http://javarevisited.blogspot.sg/2013/10/what-is-priorityqueue-data-structure-java-example-tutorial.html))  
PriorityQueue guarantees that lowest or highest priority element always remain at the head of the queue, but LinkedHashMap maintains the order on which elements are inserted. When you iterate over a PriorityQueue, iterator doesn't guarantee any order but iterator of LinkedHashMap does guarantee the order on which elements are inserted.  
  
  
**51) Difference between ArrayList and LinkedList in Java?** ([answer](http://java67.blogspot.sg/2012/12/difference-between-arraylist-vs-LinkedList-java.html))  
The obvious difference between them is that ArrrayList is backed by array data structure, supprots random access and LinkedList is backed by linked list data structure and doesn't supprot random access. Accessing an element with the index is O(1) in ArrayList but its O(n) in LinkedList. See the answer for more detailed discussion.  
  
  
**52) What is a couple of ways that you could sort a collection?** ([answer](http://java67.blogspot.sg/2012/07/sort-list-ascending-descending-order-set-arraylist.html))  
You can either use the Sorted collection like TreeSet or TreeMap or you can sort using the ordered collection like a list and using Collections.sort() method.  
  
  
**53) How do you print Array in Java?** ([answer](http://java67.blogspot.sg/2014/03/how-to-print-array-in-java-example-tutorial.html))  
You can print an array by using the Arrays.toString() and Arrays.deepToString() method. Since array doesn't implement toString() by itself, just passing an array to System.out.println() will not print its contents but Arrays.toString() will print each element.

**54) LinkedList in Java is doubly or singly linked list?** (answer)  
It's a doubly linked list, you can check the code in JDK. In Eclipse, you can use the [shortcut](http://javarevisited.blogspot.com/2010/10/eclipse-tutorial-most-useful-eclipse.html), Ctrl + T to directly open this class in Editor.  
  
**55) Which kind of tree is used to implement TreeMap in Java?** (answer)  
A Red Black tree is used to implement TreeMap in Java.

**56) What is the difference between Hashtable and HashMap?**([answer](http://java67.blogspot.sg/2012/08/5-difference-between-hashtable-hashmap-Java-collection.html))  
There are many differences between these two classes, some of them are following:  
a) Hashtable is a legacy class and present from JDK 1, HashMap was added later.  
b) Hashtable is synchronized and slower but HashMap is not synchronized and faster.  
c) Hashtable doesn't allow null keys but HashMap allows one null key.  
See the answer for more differences between HashMap and Hashtable in Java.  
  
  
**57) How HashSet works internally in Java?** ([answer](http://java67.blogspot.sg/2014/01/how-hashset-is-implemented-or-works-internally-java.html))  
HashSet is internally implemented using an HashMap. Since a Map needs key and value, a default value is used for all keys. Similar to HashMap, HashSet doesn't allow duplicate keys and only one null key, I mean you can only store one null object in HashSet.  
  
  
**58) Write code to remove elements from ArrayList while iterating?** ([answer](http://javarevisited.blogspot.sg/2014/01/ow-to-remove-objects-from-collection-arraylist-java-iterator-traversing.html))  
 Key here is to check whether candidate uses ArrayList's remove() or Iterator's remove(). Here is the [sample code](http://java67.blogspot.com/2015/10/how-to-solve-concurrentmodificationexception-in-java-arraylist.html) which uses right way o remove elements from ArrayList while looping over and avoids ConcurrentModificationException.  
  
  
**59) Can I write my own container class and use it in the for-each loop?**  
Yes, you can write your own container class. You need to implement the Iterable interface if you want to loop over advanced for loop in Java, though. If you implement Collection then you by default get that property.  
  
  
**60) What is default size of ArrayList and HashMap in Java?**([answer](http://javarevisited.blogspot.sg/2014/07/java-optimization-empty-arraylist-and-Hashmap-cost-less-memory-jdk-17040-update.html))  
As of Java 7 now, default size of ArrayList is 10 and default capacity of HashMap is 16, it must be power of 2. Here is code snippet from ArrayList  and HashMap class :

// from ArrayList.java JDK 1.7

private static final int DEFAULT\_CAPACITY = 10;

//from HashMap.java JDK 7

static final int DEFAULT\_INITIAL\_CAPACITY = 1 **<<** 4; // aka 16

**61) Is it possible for two unequal objects to have the same hashcode?**  
Yes, two unequal objects can have same hashcode that's why collision happen in a hashmap.  
the equal hashcode contract only says that two equal objects must have the same hashcode it doesn't say anything about the unequal object.  
  
**62) Can two equal object have the different hash code?**  
No, thats not possible according to hash code contract.  
  
  
**63) Can we use random numbers in the hashcode() method?** ([answer](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html))  
No, because hashcode of an object should be always same. See the answer to learning more about things to remember while overriding hashCode() method in Java.  
  
  
**64) What is the difference between Comparator and Comparable in Java?**([answer](http://java67.blogspot.sg/2013/08/difference-between-comparator-and-comparable-in-java-interface-sorting.html))  
The Comparable interface is used to define the  natural order of object while Comparator is used to define custom order. Comparable can be always one, but we can have multiple comparators to define customized order for objects.  
  
**65) Why you need to override hashcode, when you override equals in Java?** ([answer](http://javarevisited.blogspot.sg/2015/01/why-override-equals-hashcode-or-tostring-java.html))  
 Because equals have code contract mandates to override equals and hashcode together .since many container class like HashMap or HashSet depends on hashcode and equals contract.

### Java IO and NIO Interview questions

IO is very important from Java interview point of view. You should have a good knowledge of old Java IO, NIO, and NIO2 alsong with some operating system and disk IO fundamentals. Here are some frequently asked questions form Java IO.  
  
66) In my Java program, I have three sockets? How many threads I will need to handle that?  
  
67) How do you create ByteBuffer in Java?  
  
68) How do you write and read from ByteBuffer in Java?  
  
69) Is Java BIG endian or LITTLE endian?  
  
70) What is the byte order of ByteBuffer?  
  
71) The difference between direct buffer and non-direct buffer in Java? ([answer](http://javarevisited.blogspot.sg/2015/08/difference-between-direct-non-direct-mapped-bytebuffer-nio-java.html))  
  
72) What is the memory mapped buffer in Java? ([answer](http://javarevisited.blogspot.sg/2012/01/memorymapped-file-and-io-in-java.html))  
  
73) What is TCP NO DELAY socket option?  
  
74) What is the difference between TCP and UDP protocol? ([answer](http://javarevisited.blogspot.com/2014/07/9-difference-between-tcp-and-udp-protocol.html))  
  
75) The difference between ByteBuffer and StringBuffer in Java? (answer)

### Java Best Practices Interview question

Contains best practices from different parts of Java programming e.g. Collections, String, IO, Multi-threading, Error and Exception handling, design patterns etc. This section is mostly for experience Java developer, technical lead,  AVP, team lead and coders who are responsible for products. If you want to create quality products you must know and follow the best practices.  
  
**76) What best practices you follow while writing multi-threaded code in Java?** ([answer](http://javarevisited.blogspot.com/2015/05/top-10-java-multithreading-and.html))  
Here are couple of best practices which I follow while writing concurrent code in Java:  
a) Always name your thread, this will help in debugging.  
b) minimize the scope of synchronization, instead of making whole method synchronized, only critical section should be synchronized.  
c) prefer volatile over synchronized if you can can.  
e) use higher level concurrency utilities instead of waitn() and notify for inter thread communication e.g. BlockingQueue, CountDownLatch and Semeaphore.  
e) Prefer concurrent collection over synchronized collection in Java. They provide better scalability.  
  
  
**77) Tell me few best practices you apply while using Collections in Java?**(answer)  
Here are couple of best practices I follow while using Collectionc classes from Java:  
a) Always use the right collection e.g. if you need non-synchronized list then use ArrayList and not Vector.  
b) Prefer concurrent collection over synchronized collection because they are more scalable.  
c) Always use interface to a represent and access a collection e.g. use List to store ArrayList, Map to store HashMap and so on.  
d) Use iterator to loop over collection.  
e) Always use generics with collection.  
  
  
**78) Can you tell us at least 5 best practice you use while using threads in Java?** ([answer](http://java67.blogspot.com/2014/01/10-points-about-thread-and-javalangthread-in-java.html))  
This is similar to the previous question and you can use the answer given there. Particularly with thread, you should:  
a) name your thread  
b) keep your task and thread separate, use Runnable or Callable with thread pool executor.  
c) use thread pool  
d) use volatile to indicate compiler about ordering, visibility, and atomicity.  
e) avoid thread local variable because incorrect use of ThreadLocal class in Java can create a memory leak.  
Look there are many best practices and I give extra points to the developer which bring something new, something even I don't know. I make sure to ask this question to Java developers of 8 to 10 years of experience just to gauge his hands on experience and knowledge.  
  
  
**79) Name 5 IO best practices?** (answer)  
IO is very important for performance of your Java application. Ideally you should avoid IO in critical path of your application. Here are couple of Java IO best practices you can follow:

a) Use buffered IO classes instead of reading individual bytes and char.

b) Use classes from NIO and NIO2

c) Always close streams in finally block or use try-with-resource statements.

d) use memory mapped file for faster IO.

If a Java candidate doesn't know about IO and NIO, especially if he has at least 2 to 4 years of experience, he needs some reading.  
  
  
**80) Name 5 JDBC best practices your follow?** ([answer](http://javarevisited.blogspot.sg/2012/08/top-10-jdbc-best-practices-for-java.html))  
Another good Java best practices for experienced Java developer of 7 to 8 years experience. Why it's important? because they are the ones which set the trend in the code and educate junior developers. There are many best practices and you can name as per your confort and conviniece. Here are some of the more common ones:  
a) use batch statement for inserting and updating data.  
b) use PreparedStatement to avoid SQL exception and better performance.  
c) use database connection pool  
d) access resultset using column name instead of column indexes.  
e) Don't generate dynamic SQL by concatenating String with user input.  
  
  
**81) Name couple of method overloading best practices in Java?** ([answer](http://javarevisited.blogspot.sg/2013/01/java-best-practices-method-overloading-constructor.html))  
Here are some best practices you can follow while overloading a method in Java to avoid confusion with auto-boxing:  
a) Don't overload method where one accepts int and other accepts Integer.  
b) Don't overload method where number of argument is same and only order of argument is different.  
c) Use varargs after overloaded methods has more than 5 arguments.

### Date, Time and Calendar Interview questions in Java

**82) Does SimpleDateFormat is safe to use in the multi-threaded program?** ([answer](http://javarevisited.blogspot.sg/2012/03/simpledateformat-in-java-is-not-thread.html))  
No, unfortunately, DateFormat and all its implementations including SimpleDateFormat is not thread-safe, hence should not be used in the multi-threaded program until external thread-safety measures are applied e.g. confining SimpleDateFormat object into a ThreadLocal variable. If you don't do that, you will get an incorrect result while parsing or formatting dates in Java. Though, for all practical date time purpose, I highly recommend **joda-time** library.  
  
  
**83) How do you format a date in Java? e.g. in the ddMMyyyy format?** ([answer](http://javarevisited.blogspot.com/2011/09/convert-date-to-string-simpledateformat.html))  
You can either use SimpleDateFormat class or joda-time library to format date in Java. DateFormat class allows you to format date on many popular formats. Please see the answer for code samples to format date into different formats e.g. dd-MM-yyyy or ddMMyyyy.  
  
  
84) How do you show timezone in formatted date in Java? ([answer](http://java67.blogspot.sg/2013/01/how-to-format-date-in-java-simpledateformat-example.html))  
  
85) The difference between java.util.Date and java.sql.Date in Java? ([answer](http://java67.blogspot.sg/2014/02/how-to-convert-javautildate-to-javasqldate-example.html))  
  
86) How to you calculate the difference between two dates in Java? ([program](http://javarevisited.blogspot.sg/2015/07/how-to-find-number-of-days-between-two-dates-in-java.html))  
  
87) How do you convert a String(YYYYMMDD) to date in Java? ([answer](http://java67.blogspot.sg/2014/12/string-to-date-example-in-java-multithreading.html))

### Unit testing JUnit Interview questions

89) How do you test static method? (answer)  
You can use PowerMock library to test static methods in Java.  
  
90) How to do you test a method for an exception using JUnit? ([answer](http://javarevisited.blogspot.sg/2013/04/JUnit-tutorial-example-test-exception-thrown-by-java-method.html))  
  
91) Which unit testing libraries you have used for testing Java programs? (answer)  
  
92) What is the difference between @Before and @BeforeClass annotation? ([answer](http://javarevisited.blogspot.sg/2013/04/JUnit-tutorial-example-test-exception-thrown-by-java-method.html))

### Programming and Coding Questions

93) How to check if a String contains only numeric digits? ([solution](http://java67.blogspot.com/2014/01/java-regular-expression-to-check-numbers-in-String.html))  
  
94) How to write LRU cache in Java using Generics? (answer)  
  
95) Write a Java program to convert bytes to long? (answer)  
  
96) How to reverse a String in Java without using StringBuffer? ([solution](http://java67.blogspot.com/2012/12/how-to-reverse-string-in-java-stringbuffer-stringbuilder.htm))  
  
97) How to find the word with the highest frequency from a file in Java? ([solution](http://java67.blogspot.com/2015/10/java-program-to-find-repeated-words-and-count.html))  
  
98) How do you check if two given String are anagrams? ([solution](http://javarevisited.blogspot.sg/2013/03/Anagram-how-to-check-if-two-string-are-anagrams-example-tutorial.html))  
  
99) How to print all permutation of a String in Java? ([solution](http://javarevisited.blogspot.com/2015/08/how-to-find-all-permutations-of-string-java-example.html))  
  
100) How do you print duplicate elements from an array in Java? ([solution](http://javarevisited.blogspot.com/2015/06/3-ways-to-find-duplicate-elements-in-array-java.html))  
  
101) How to convert String to int in Java? ([solution](http://java67.blogspot.com/2015/08/2-ways-to-parse-string-to-int-in-java.html))  
  
102) How to swap two integers without using temp variable? ([solution](http://java67.blogspot.com/2015/08/how-to-swap-two-integers-without-using.html))

### Java Interview questions from OOP and Design Patterns

It contains Java Interview questions from SOLID design principles, OOP fundamentals e.g. class, object, interface, Inheritance, Polymorphism, Encapsulation, and Abstraction as well as more advanced concepts like Composition, Aggregation, and Association. It also contains questions from GOF design patterns.  
  
**103) What is the interface? Why you use it if you cannot write anything concrete on it?**  
The interface is used to define API. It tells about the contract your classes will follow. It also supports abstraction because a client can use interface method to leverage multiple implementations e.g. by using List interface you can take advantage of [random access of ArrayList](http://javarevisited.blogspot.com/2015/07/java-arraylist-tutorial.html) as well as flexible insertion and deletion of LinkedList. The interface doesn't allow you to write code to keep things abstract but from Java 8 you can declare static and default methods inside interface which are concrete.  
  
  
**104) The difference between abstract class and interface in Java?**([answer](http://javarevisited.blogspot.sg/2013/05/difference-between-abstract-class-vs-interface-java-when-prefer-over-design-oops.html))  
There are multiple differences between abstract class and interface in Java, but the most important one is Java's restriction on allowing a class to extend just one class but allows it to implement multiple interfaces. An abstract class is good to define default behavior for a family of class, but the interface is good to define Type which is later used to leverage Polymorphism. Please check the answer for a more thorough discussion of this question.  
  
  
**105) Which design pattern have you used in your production code? apart from Singleton?**  
This is something you can answer from your experience. You can generally say about dependency injection, factory pattern, decorator pattern or observer pattern, whichever you have used. Though be prepared to answer follow-up question based upon the pattern you choose.  
  
  
**106) Can you explain Liskov Substitution principle?** ([answer](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html))  
This is one of the toughest questions I have asked in Java interviews. Out of 50 candidates, I have almost asked only 5 have managed to answer it. I am not posting an answer to this question as I like you to do some research, practice and spend some time to understand this confusing principle well.  
  
  
**107) What is Law of Demeter violation? Why it matters?** ([answer](http://javarevisited.blogspot.com/2014/05/law-of-demeter-example-in-java.html))  
Believe it or not, Java is all about application programming and structuring code. If  you have good knowledge of common coding best practices, patterns and what not to do than only you can write quality code.  Law of Demeter suggests you "talk to friends and not stranger", hence used to reduce coupling between classes.  
  
  
**108) What is Adapter pattern? When to use it?**  
Another frequently asked Java design pattern questions. It provides interface conversion. If your client is using some interface but you have something else, you can write an Adapter to bridge them together. This is good for Java software engineer having 2 to 3 years experience because the question is neither difficult nor tricky but requires knowledge of OOP design patterns.  
  
  
**109) What is "dependency injection" and "inversion of control"? Why would someone use it?**([answer](http://javarevisited.blogspot.sg/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html))  
  
**110) What is an abstract class? How is it different from an interface? Why would you use it?**([answer](http://java67.blogspot.sg/2014/06/why-abstract-class-is-important-in-java.html))  
One more classic question from Programming Job interviews, it is as old as chuck Norris. An abstract class is a class which can have state, code and implementation, but an interface is a contract which is totally abstract. Since I have answered it many times, I am only giving you the gist here but you should read the article linked to answer to learn this useful concept in much more detail.  
  
  
**111) Which one is better constructor injection or setter dependency injection?**([answer](http://javarevisited.blogspot.sg/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html))  
Each has their own advantage and disadvantage. Constructor injection guaranteed that class will be initialized with all its dependency, but setter injection provides flexibility to set an optional dependency. Setter injection is also more readable if you are using an XML file to describe dependency. Rule of thumb is to use constructor injection for mandatory dependency and use setter injection for optional dependency.  
  
  
**112) What is difference between dependency injection and factory design pattern?** ([answer](http://javarevisited.blogspot.sg/2015/06/difference-between-dependency-injection.html))  
Though both patterns help to take out object creation part from application logic, use of dependency injection results in cleaner code than factory pattern. By using dependency injection, your classes are nothing but POJO which only knows about dependency but doesn't care how they are acquired. In the case of factory pattern, the class also needs to know about factory to acquire dependency. hence, DI results in more testable classes than factory pattern. Please see the answer for a more detailed discussion on this topic.  
  
  
**113) Difference between Adapter and Decorator pattern?**([answer](http://javarevisited.blogspot.sg/2015/01/adapter-vs-decorator-vs-facade-vs-proxy-pattern-java.html))  
Though the structure of Adapter and Decorator pattern is similar, the difference comes on the intent of each pattern. The adapter pattern is used to bridge the gap between two interfaces, but Decorator pattern is used to add new functionality into the class without the modifying existing code.  
  
  
**114) Difference between Adapter and Proxy Pattern?**([answer](http://javarevisited.blogspot.sg/2015/01/adapter-vs-decorator-vs-facade-vs-proxy-pattern-java.html))  
Similar to the previous question, the difference between Adapter and Proxy patterns is in their intent. Since both Adapter and Proxy pattern encapsulate the class which actually does the job, hence result in the same structure, but Adapter pattern is used for interface conversion while the Proxy pattern is used to add an extra level of indirection to support distribute, controlled or intelligent access.  
  
  
**115) What is Template method pattern?** (answer)  
Template pattern provides an outline of an algorithm and lets you configure or customize its steps. For examples, you can view a sorting algorithm as a template to sort object. It defines steps for sorting but let you configure how to compare them using Comparable or something similar in another language. The method which outlines the algorithms is also known as template method.  
  
  
**116) When do you use Visitor design pattern?**(answer)  
The visitor pattern is a solution of problem where you need to add operation on a class hierarchy but without touching them. This pattern uses double dispatch to add another level of indirection.  
  
  
**117) When do you use Composite design pattern?**(answer)  
Composite design pattern arranges objects into tree structures to represent part-whole hierarchies. It allows clients treat individual objects and container of objects uniformly. Use Composite pattern when you want to represent part-whole hierarchies of objects.

**118) The difference between Inheritance and Composition?** ([answer](http://javarevisited.blogspot.sg/2015/06/difference-between-inheritance-and-Composition-in-Java-OOP.html))  
Though both allows code reuse, Composition is more flexible than Inheritance because it allows you to switch to another implementation at run-time. Code written using Composition is also easier to test than code involving inheritance hierarchies.  
  
  
**119) Describe overloading and overriding in Java?** ([answer](http://java67.blogspot.sg/2012/09/difference-between-overloading-vs-overriding-in-java.html))  
Both overloading and overriding allow you to write two methods of different functionality but with the same name, but overloading is compile time activity while overriding is run-time activity. Though you can overload a method in the same class, but you can only override a method in child classes. Inheritance is necessary for overriding.  
  
  
**120) The difference between nested public static class and a top level class in Java?** ([answer](http://javarevisited.blogspot.sg/2012/12/inner-class-and-nested-static-class-in-java-difference.html))  
You can have more than one nested public static class inside one class, but you can only have one top-level public class in a Java source file and its name must be same as the name of Java source file.  
  
  
**121) Difference between Composition, Aggregation and Association in OOP?** ([answer](http://javarevisited.blogspot.sg/2014/02/ifference-between-association-vs-composition-vs-aggregation.html))  
If two objects are related to each other, they are said to be associated with each other. Composition and Aggregation are two forms of association in object-oriented programming. The composition is stronger association than Aggregation. In Composition, one object is OWNER of another object while in Aggregation one object is just USER of another object. If an object A is composed of object B then B doesn't exist if A ceased to exists, but if object A is just an aggregation of object B then B can exists even if A ceased to exist.  
  
  
**122) Give me an example of design pattern which is based upon open closed principle?** ([answer](http://javarevisited.blogspot.sg/2011/11/great-example-of-open-closed-design.html))  
This is one of the practical questions I ask experienced Java programmer. I expect them to know about OOP design principles as well as patterns. Open closed design principle asserts that your code should be open for extension but closed for modification. Which means if you want to add new functionality, you can add it easily using the new code but without touching already tried and tested code.  There are several design patterns which are based upon open closed design principle e.g. [Strategy pattern](http://java67.blogspot.com/2014/12/strategy-pattern-in-java-with-sample.html) if you need a new strategy, just implement the interface and configure, no need to modify core logic. One working example is Collections.sort() method which is based on Strategy pattern and follows the open-closed principle, you don't modify sort() method to sort a new object, what you do is just implement Comparator in your own way.  
  
  
**123) Difference between Abstract factory and Prototype design pattern?** (answer)  
This is the practice question for you, If you are feeling bored just reading and itching to write something, why not write the answer to this question. I would love to see an example the, which should answer where you should use the Abstract factory pattern and where is the Prototype pattern is more suitable.  
  
  
**124) When do you use Flyweight pattern?** (answer)  
This is another popular question from the design pattern. Many Java developers with 4 to 6 years of experience know the definition but failed to give any concrete example. Since many of you might not have used this pattern, it's better to look examples from JDK. You are more likely have used them before and they are easy to remember as well. Now let's see the answer.  
Flyweight pattern allows you to share object to support large numbers without actually creating too many objects. In order to use Flyweight pattern, you need to make your object Immutable so that they can be safely shared. String pool and pool of Integer and Long object in JDK are good examples of Flyweight pattern.

### Miscellaneous Java Interview Questions

It contains XML Processing in Java Interview question, JDBC Interview question, Regular expressions Interview questions, Java Error and Exception Interview Questions, Serialization,  
  
**125) The difference between nested static class and top level class?**([answer](http://java67.blogspot.sg/2012/10/nested-class-java-static-vs-non-static-inner.html))  
One of the fundamental questions from Java basics. I ask this question only to junior Java developers of 1 to 2 years of experience as it's too easy for an experience Java programmers. The answer is simple, a public top level class must have the same name as the name of the source file, there is no such requirement for nested static class. A nested class is always inside a top level class and you need to use the name of the top-level class to refer nested static class e.g. HashMap.Entry is a nested static class, where HashMap is a top level class and Entry is nested static class.  
  
  
**126) Can you write a regular expression to check if String is a number?**([solution](http://javarevisited.blogspot.sg/2012/10/regular-expression-example-in-java-to-check-String-number.html))  
If you are taking Java interviews then you should ask at least one question on the regular expression. This clearly [differentiates an average programmer with a good programmer](http://javarevisited.blogspot.com/2015/05/how-to-differentiate-between-average.html). Since one of the traits of a good developer is to know tools, regex is the best tool for searching something in the log file, preparing reports etc. Anyway, answer to this question is, a numeric String can only contain digits i.e. 0 to 9 and + and - sign that too at start of the String, by using this information you can write following regular expression to check if given String is number or not  
  
  
**127) The difference between checked and unchecked Exception in Java?**([answer](http://java67.blogspot.sg/2012/12/difference-between-runtimeexception-and-checked-exception.html))  
checked exception is checked by the compiler at compile time. It's mandatory for a method to either handle the checked exception or declare them in their throws clause. These are the ones which are a sub class of Exception but doesn't descend from RuntimeException. The unchecked exception is the descendant of RuntimeException and not checked by the compiler at compile time. This question is now becoming less popular and you would only find this with interviews with small companies, both investment banks and startups are moved on from this question.  
  
  
**128) The difference between throw and throws in Java?** ([answer](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html))  
the throw is used to actually throw an instance of java.lang.Throwable class, which means you can throw both Error and Exception using throw keyword e.g.

throw new IllegalArgumentException("size must be multiple of 2")

On the other hand, throws is used as part of method declaration and signals which kind of exceptions are thrown by this method so that its caller can handle them. It's mandatory to declare any unhandled checked exception in **throws** clause in Java. Like the previous question, this is another frequently asked Java interview question from errors and exception topic but too easy to answer.  
  
  
**129) The difference between Serializable and Externalizable in Java?** ([answer](http://javarevisited.blogspot.sg/2012/01/serializable-externalizable-in-java.html))  
This is one of the frequently asked questions from Java Serialization. The interviewer has been asking this question since the day Serialization was introduced in Java, but yet only a few good candidate can answer this question with some confidence and practical knowledge. Serializable interface is used to make Java classes serializable so that they can be transferred over network or their state can be saved on disk, but it leverages default serialization built-in JVM, which is expensive, fragile and not secure. Externalizable allows you to fully control the Serialization process, specify a custom binary format and add more security measure.  
  
  
**130) The difference between DOM and SAX parser in Java?**([answer](http://javarevisited.blogspot.sg/2011/12/difference-between-dom-and-sax-parsers.html))  
Another common Java question but from XML parsing topic. It's rather simple to answer and that's why many interviewers prefers to ask this question on the telephonic round. DOM parser loads the whole XML into memory to create a tree based DOM model which helps it quickly locate nodes and make a change in the structure of XML while SAX parser is an event based parser and doesn't load the whole XML into memory. Due to this reason DOM is faster than SAX but require more memory and not suitable to parse large XML files.  
  
  
**131) Tell me 3 features introduced on JDK 1.7?**([answer](http://javarevisited.blogspot.sg/2014/04/10-jdk-7-features-to-revisit-before-you.html))  
This is one of the good questions I ask to check whether the candidate is aware of recent development in Java technology space or not. Even though JDK 7 was not a big bang release like JDK 5 or JDK 8, it still has a lot of good feature to count on e.g. try-with-resource statements, which free you from closing streams and resources when you are done with that, Java automatically closes that. Fork-Join pool to implement something like the Map-reduce pattern in Java. Allowing String variable and literal into switch statements. Diamond operator for improved type inference, no need to declare generic type on the right-hand side of variable declaration anymore, results in more readable and succinct code. Another worth noting feature introduced was improved exception handling e.g. allowing you to catch multiple exceptions in the same catch block.  
  
  
**132) Tell me 5 features introduced in JDK 1.8?**([answer](http://javarevisited.blogspot.sg/2014/02/10-example-of-lambda-expressions-in-java8.html))  
This is the follow-up question of the previous one. Java 8 is path breaking release in Java's history, here are the top 5 features from JDK 8 release

* **Lambda expression**, which allows you pass an anonymous function as object.
* **Stream API**, take advantage of multiple cores of modern CPU and allows you to write succinct code.
* **Date and Time API**, finally you have a solid and easy to use date and time library right into JDK
* **Extension methods**, now you can have static and default method into your interface
* **Repeated annotation**, allows you apply the same annotation multiple times on a type

**133) What is the difference between Maven and ANT in Java?** ([answer](http://javarevisited.blogspot.sg/2015/01/difference-between-maven-ant-jenkins-and-hudson.html))  
Another great question to check the all round knowledge of Java developers. It's easy to answer questions from core Java but when you ask about setting things up, building Java artifacts, many Java software engineer struggles. Coming back to the answer of this question, Though both are build tool and used to create Java application build, Maven is much more than that. It provides standard structure for Java project based upon "convention over configuration" concept and automatically manage dependencies (JAR files on which your application is dependent) for Java application. Please see the answer for more differences between Maven and ANT tool.  
  
  
That's all guys, **lots of Java Interview questions?** isn't it? I am sure if you can answer this list of Java questions you can easily crack any core Java or advanced Java interview. Though I have not included questions from Java EE or J2EE topics e.g. Servlet, JSP, JSF, JPA, JMS, EJB or any other Java EE technology or from major web frameworks like Spring MVC, Struts 2.0, Hibernate or both SOAP and RESTful web services, it's still useful for Java developers preparing for Java web developer position, because every Java interview starts with questions from fundamentals and JDK API. If you think, I have missed any popular Java question here and you think it should be in this list then feel free to suggest me. My goal is to create the best list of Java Interview Questions with latest and greatest question from recent interviews.  
  
  
Read more: <https://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html#ixzz5WEprlU5W>

**Question 1: What’s wrong with using HashMaps in a multi-threaded environment? When does a get() method go into an infinite loop?**([answer](http://java67.blogspot.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html))

Well, nothing is wrong — it depends on how you use it. For example, if you initialize the HashMap with just one thread and all threads are only reading from it, then it's perfectly fine.

One example of this is a Map that contains configuration properties. The real problem starts when at least one of those threads is updating the HashMap, i.e. adding, changing, or removing any key-value pair.

Since a put() operation can cause re-sizing, which can further lead to infinite loop, that's why either you should use a [Hashtable](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html) or [ConcurrentHashMap](http://javarevisited.blogspot.com/2013/02/concurrenthashmap-in-java-example-tutorial-working.html) (the latter is better).

**Question 2. Does not overriding a hashCode() method have any performance implication?**([answer](http://java67.blogspot.com/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html))

This is a good question and open to all, as per my knowledge, a poor hashCode function will result in frequent [collisions in HashMap](http://javarevisited.blogspot.sg/2016/01/how-does-java-hashmap-or-linkedhahsmap-handles.html), which eventually increases the time for adding an object into said HashMap.

From [Java 8](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fjava-8-core-training-%2F) onwards, though, collisions will not impact performance as much as in earlier versions because, after a threshold, the linked list will be replaced by the binary tree, which will give you **O(log N)**performance in the worst case, as compared to **O(N)** of a linked list.

This is one of the several [tricky Java questions](http://java67.blogspot.com/2012/09/top-10-tricky-java-interview-questions-answers.html) you will face, as many developers only know about equals hashcode contracts and don't think about their performance implications.

**Question 3: Do all properties of immutable objects need to be final?**([answer](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html))

Not necessarily. As stated above, you can achieve the same functionality by making members non-final but [private](http://javarevisited.blogspot.sg/2012/10/difference-between-private-protected-public-package-access-java.html#axzz59Lhz7uVu) — and not modifying them except in constructors.

Don't provide a [setter method](http://javarevisited.blogspot.sg/2012/12/getter-and-setter-method-vs-public-modifier-field-java.html#axzz55oDxm8vv) for them, and if it is a mutable object, then don't ever leak any reference to that member.

Remember, making a reference variable [final](http://javarevisited.blogspot.sg/2016/09/21-java-final-modifier-keyword-interview-questions-answers.html#axzz59Lhz7uVu) only ensures that it will not be reassigned a different value, but you can still change individual properties of an object pointed by that reference variable.

This is one of the key points an interviewer always likes to hear from candidates. By mentioning that, you can score some brownie points.

**Question 4: How does a substring () inside a String work?**([answer](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html))

Many developers know the answer: “A substring creates a new object out of the source string by taking a portion of original string.”

But I think that answer is insufficient.

This question was mainly asked to see if the developer is familiar with the risk of **memory leaks** that substrings can create.

Until Java 1.7, a substring holds the reference of the original character array, which means even a substring 5 characters long can prevent a 1GB character array from garbage collection by holding a [strong reference](http://javarevisited.blogspot.sg/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html#axzz54AAeS1IM).

This issue is fixed in Java 1.7, where the original character array is not referenced anymore, but that change also made the creation of substrings a bit more costly in terms of time. Earlier, it was in the range of O(1), which could be O(N) in the worst case in Java 7.

**Question 5: Can you write critical section code for a singleton?**([answer](http://javarevisited.blogspot.sg/2014/05/double-checked-locking-on-singleton-in-java.html))

This core Java question is a follow-up of the previous question and expects the candidate to write a Java singleton using the [Double-Checked Locking](http://www.java67.com/2016/04/why-double-checked-locking-was-broken-before-java5.html) Pattern.

Remember to use a [volatile variable](http://javarevisited.blogspot.sg/2011/06/volatile-keyword-java-example-tutorial.html) to make Singleton thread-safe.

Here is the code for a critical section of a [thread-safe singleton](http://javarevisited.blogspot.sg/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) class using the Double-Checked Locking idiom:

public class Singleton {

private static volatile Singleton \_instance;

/\*\*

\* Double checked locking code on Singleton

\* @return Singelton instance

\*/

public static Singleton getInstance() {

if (\_instance == null) {

synchronized(Singleton.class) {

if (\_instance == null) {

\_instance = new Singleton();

}

}

}

return \_instance;

}

}

**Question 6: How do you handle error conditions while writing a stored procedure or accessing stored procedures from Java?**([answer](http://javarevisited.blogspot.com/2013/04/spring-framework-tutorial-call-stored-procedures-from-java.html))

This is one of the [tough Java interview questions](http://www.java67.com/2012/09/top-10-tough-core-java-interview-questions-answers.html) and it's open for all. I have a friend who didn't know the answer, and he didn't mind telling me.

My take is that stored procedure should return an error code if some operation fails, but if the stored procedure itself fails, then catching a [SQLException](http://javarevisited.blogspot.sg/2016/09/javasqlsqlexception-no-suitable-driver-mysql-jdbc-localhost.html#axzz59Lhz7uVu) is the only choice.

**Question 7 : What is the difference between the Executor.submit() and Executer.execute() methods?**([answer](http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html))

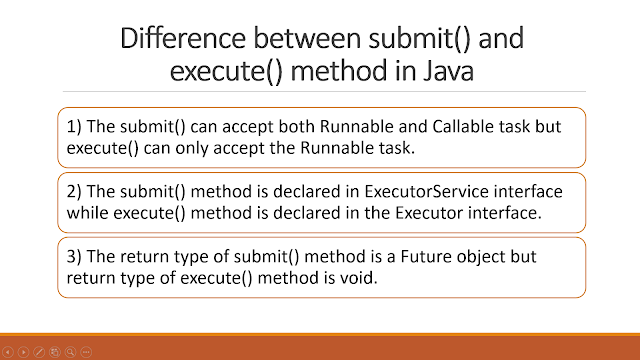
## Executor.execute() vs ExecutorService.submit() method

As I told in the first paragraph that key difference between the execute() and submit() method is that former cannot return the result but later can result of computation. Before seeing more difference, let's see some similarities between the execute() and submit() as well:

1) Both submit() and execute() methods are used to submit a task to [Executor framework](http://javarevisited.blogspot.com/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html) for asynchronous execution.  
  
2) Both submit() and execute() can accept a Runnable task.  
  
3) You can access submit() and execute() from the ExecutorService interface because it also extends the Executor interface which declares the execute() method.  
  
Apart from the fact that submit() method can return output and execute() cannot, following are other notable differences between these two key methods of Executor framework of Java 5.  
  
1) The submit() can accept both [Runnable](http://java67.blogspot.com/2016/01/7-differences-between-extends-thread-vs-implements-Runnable-java.html) and [Callable](http://javarevisited.blogspot.com/2015/06/how-to-use-callable-and-future-in-java.html) task but execute() can only accept the Runnable task.  
  
2) The submit() method is declared in ExecutorService interface while execute() method is declared in the Executor interface.  
  
3) The return type of submit() method is a Future object but return type of execute() method is void.  
  
Btw, Cay S. Horstmann has also covered this essential topic on good detail in his classic book, [Core Java Volume 1 - Fundamentals](http://www.amazon.com/Volume-II-Advanced-Features-Edition-Series/dp/013708160X?tag=javamysqlanta-20), 10th Edition. You can refer that book for further reading on this topic.

## When to use submit() and execute() method in Java

Once you understand the difference between Executor.execute() and ExecutorService.submit() method you have the knowledge to decide w*hen to use submit() and when to use the execute() method*.  
  
In general, if you are doing computational task e.g. calculating some risk stats, [calculating factorial of large numbers](http://java67.blogspot.com/2015/09/how-to-use-biginteger-class-in-java.html) or doing some time-consuming computation e which results in some value then use the submit() method. It immediately returns a Future object, which can be later queried to get the value of computation by calling get() method.  
  
Remember, get() is a [blocking call](http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html) so always call the version which accepts a timeout. While you can use the execute() method if you just want your code to be run in parallel by worker threads of the thread pool.  
  
Here is a nice summary of key differences between submit() vs execute() methods in Java:

[](http://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html)

That's all about **difference between ExecutorService.submit() and Executor.execute() method in Java**. Remember, the key difference between them is that submit() return a Future but execute() return nothing.  
  
The thread pools created by Executors class always return a reference of ExecutorService, which provides access to both submit() and execute() method as it also extend the Executor interface, which is source of execute() method.  
  
Use submit() if your doing computation e.g. calculating value of pie, and use execute() if you just want the code to be run in parallel by worker threads of thread pool.  
  
Read more: <https://javarevisited.blogspot.com/2016/04/difference-between-ExecutorServie-submit-vs-Executor-execute-method-in-Java.html#ixzz5axcGxEz2>

**Question 8: What is the difference between the Factory and Abstract Factory patterns?**([answer](http://javarevisited.blogspot.sg/2013/01/difference-between-factory-and-abstract-factory-design-pattern-java.html))

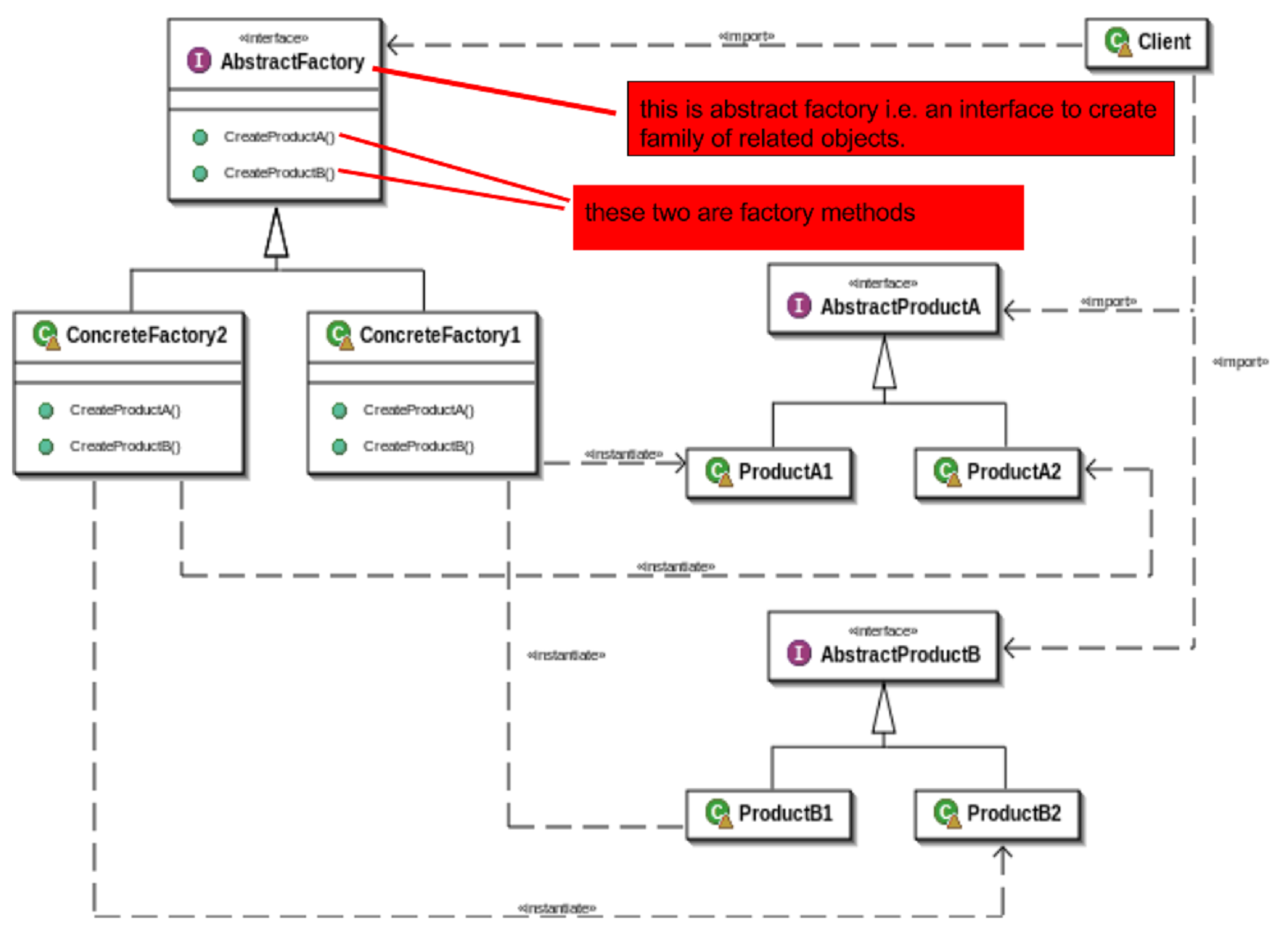
Compared to the Factory pattern, Abstract Factory provides one more level of [Abstraction](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html#axzz59Lhz7uVu).

Consider different factories each extended from an Abstract Factory and that are responsible for the creation of different hierarchies of objects based on the type of factory.

For example, AbstractFactory is extended by AutomobileFactory, UserFactory, RoleFactory, etc.

Each individual factory would be responsible for the creation of objects in that genre.

Here is a UML diagram of the [Factory](http://javarevisited.blogspot.sg/2015/06/difference-between-dependency-injection.html) and [Abstract Factory](http://www.java67.com/2012/09/top-10-java-design-pattern-interview-question-answer.html) patterns:



**Question 9: What is Singleton? Is it better to make the whole method synchronized, or only the critical section?**([answer](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html))

Singleton in Java is a class with just one instance in an entire Java application. For example, java.lang.Runtime is a Singleton class.

Creating a Singleton was tricky prior to Java 5, but once Java 5 introduced [Enum](http://javarevisited.blogspot.sg/2011/08/enum-in-java-example-tutorial.html), it becames very easy.

Please see my article [How to create thread-safe Singletons in Java](http://javarevisited.blogspot.sg/2012/07/why-enum-singleton-are-better-in-java.html#axzz4tzMEHSJw) for more details on writing Singletons using Enums and double-checked locking, which is the purpose of this Java interview question.

**Question 10: Can you write code for iterating over HashMaps in Java 4 and Java 5?**([answer](http://java67.blogspot.com/2014/05/3-examples-to-loop-map-in-java-foreach.html))

This is a tricky one, but as the answerer shows, he managed to write using while and a for loop. Actually, there are[four ways to iterate over any Map in Java](http://javarevisited.blogspot.com/2011/12/how-to-traverse-or-loop-hashmap-in-java.html).

The method involves using [keySet()](http://www.java67.com/2016/05/keyset-vs-entryset-vs-values-in-java-map-example.html) and iterating over the key, then using a [get()](http://www.java67.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) method to retrieve values, which is bit expensive.

The second method involves using entrySet() and iterating over them either by using a for each loop or while with the Iterator.hasNext() method.

This one is a better approach because both key and value objects are available to you during iteration and you don't need to call the [get()](http://javarevisited.blogspot.sg/2011/02/how-hashmap-works-in-java.html) method for retrieving the value, which could give O(N) performance in the case of a huge linked list in one bucket.

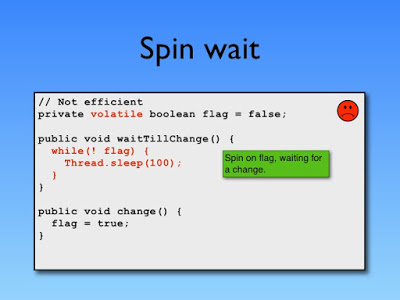
This could be slightly better if you are using Java 8 and there is a tree instead of the linked list.

## Conclusion

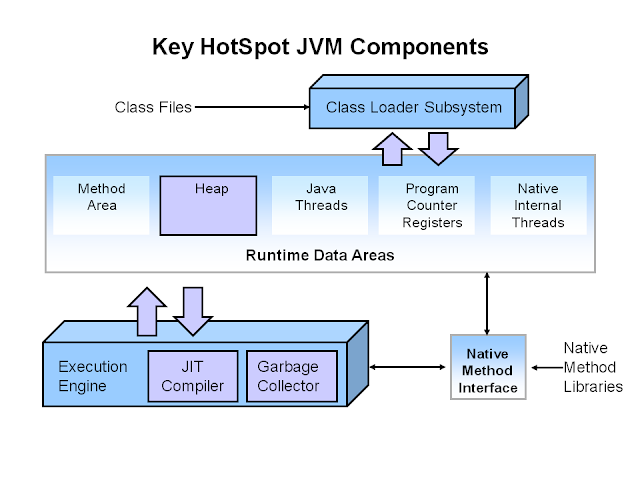
That's all about some of the common Java interview questions from investment banks. If you are going for a Java developer role, then expect a lot of focus on Java concurrency, multithreading, collections, JVM internals, garbage collection, and how to increase the performance of Java applications.

I have not touched all the topics you need to be ready for during Java interviews in this article, but if you are serious about preparation, here are a couple of useful resources to find more structured and topical questions:

**1. What is Busy Spinning? Why Should You Use It in Java?**  
One of the interesting multithreading question to senior Java programmers, busy spinning is a *waiting strategy*, in which a thread just wait in a loop, without releasing the CPU for going to [sleep](http://java67.blogspot.sg/2012/08/what-are-difference-between-wait-and.html). This is a very advanced and specialized waiting strategy used in the high-frequency trading application when wait time between two messages is very minimal.  
  
By not releasing the CPU or [suspending the thread](http://java67.blogspot.com/2015/06/how-to-pause-thread-in-java-using-sleep.html), your thread retains all the cached data and instruction, which may be lost if the thread was suspended and resumed back in a different core of CPU.   
  
This question is quite popular in high-frequency low latency programming domain, where programmers are trying for extremely low latency in the range of micro to milliseconds. See here more [50+ advanced thread interview questions](http://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-answers.html) for experienced programmers. 

[](https://3.bp.blogspot.com/-SN8fjcDcDS0/V6NDiqjeF-I/AAAAAAAAGxk/41lOd51QXCcjB1k4PQUYXNDfJpffg04bQCLcB/s1600/busy+spin+Java+multithreading.jpg)

**2. What is Read-Write Lock? Does ConcurrentHashMap in Java Uses The ReadWrite Lock?**  
ReadWrite Lock is an implementation of *lock stripping* technique, where two separate locks are used for read and write operation. Since read operation doesn't modify the state of the object, it's safe to allow multiple thread access to a shared object for reading without locking, and by splitting one lock into [read and write lock](http://javarevisited.blogspot.com/2014/10/how-to-use-locks-in-multi-threaded-java-program-example.html), you can easily do that.   
  
Java provides an implementation of read-write lock in the form of ReentrantReadWritLock class in the java.util.concurrent.lock package. This is worth looking before you decide to write your own read-write locking implementation.   
  
Also, the current implementation of java.util.ConcurrentHashMap doesn't use the ReadWriteLock, instead, it divides the Map into several segments and locks them separately using different locks. This means any given time, *only a portion of the ConcurrentHashMap is locked*, instead of the whole Map. See [how ConcurrentHashMap internally works in Java](http://javarevisited.blogspot.com/2013/02/concurrenthashmap-in-java-example-tutorial-working.html) for more detail.   
  
  
This core Java question is also very popular on senior and more experienced level Java interviews e.g. 4 to 6 years, where you expect Interviewer to go into more detail, e.g. by asking you to provided an implementation of the read-write lock with different policies. If you are an experienced Java programmer, consider reading [Java Concurrency in Practice](http://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20) to gain more confidence about multithreading and concurrency in Java.   
  
  
  
**3. How to Make an Object Immutable in Java? Why Should You Make an Object Immutable?**  
Well, Immutability offers several advantages including thread-safety, ability to cache and result in more readable multithreading code. See [here](http://javarevisited.blogspot.sg/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html)to learn how to make object Immutable. Once again, this question can also go into more detail and depending on your answer, can bring several other questions e.g. when you mention Spring is Immutable, be ready with some reasons on [Why String is Immutable in Java](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html).  
  
  
**4. Which Design Patterns have You Used in Your Java Project?**  
Always expect some design patterns related question for Core Java Interview of senior developer position. It's a better strategy to mention any GOF design pattern rather than Singleton or MVC, which almost every other Java developer use it.   
  
Your best bet can be [Decorator pattern](http://java67.blogspot.sg/2013/07/decorator-design-pattern-in-java-real-life-example-tutorial.html) or may be [Dependency Injection Pattern](http://javarevisited.blogspot.sg/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html), which is quite popular in Spring Framework. It's also good to mention only the design patterns which you have *actually* used in your project and knows it's tradeoffs.   
  
It's common that once you mention a particular design pattern say Factory or Abstract Factory, Interviewer's next question would be, *have you used this pattern in your project?* So be ready with proper example and why you choose a particular pattern. You can also see [this](http://java67.blogspot.com/2012/09/top-10-java-design-pattern-interview-question-answer.html) article for more advanced design pattern questions from Java interviews.   
  
  
  
**5.  Do you know about Open Closed Design Principle or Liskov Substitution Principle?**  
Design patterns are based on [object-oriented design principles](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html), which I strongly felt every object-oriented developer and the programmer should know, or, at least, have a basic idea of what are these principles and how they help you to write better object oriented code. I  
  
f you don't know the answer to this question, you can politely say No, as it's not expected from you to know the answer to every question, but by answering this question, you can make your claim stronger as many experienced developers fail to answer basic questions like this. See [Clean Code](http://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882?tag=javamysqlanta-20) to learn more about object-oriented and SOLID design principles.  
 **6. Which Design Pattern Will You Use to Shield Your Code From a Third Party library Which Will Likely to be Replaced by Another in Couple of Months?**  
This is just one example of  the scenario-based design pattern interview question. In order to test the practical experience of Java developers with more than 5 years experience, companies ask this kind of questions.  You can expect more real-world design problems in different formats, some with more detail explanation with context, or some with only intent around.   
  
One way to shield your code from third party library is to [code against an interface rather than implementation](http://javarevisited.blogspot.com/2014/11/why-use-interface-in-java-or-object-oriented-programming.html) and then use dependency injection to provide a particular implementation. This kind of questions is also asked quite frequently to experienced and senior *Java developers with 5 to 7 years of experience*.  
 **Question 7)  How  do you prevent SQL Injection in Java Code?**  
This question is more asked to J2EE and Java EE developers than core Java developers, but, it is still a good question to check the JDBC and Security skill of experienced Java programmers.  
  
You can [use PreparedStatement to avoid SQL injection](http://javarevisited.blogspot.com/2012/03/why-use-preparedstatement-in-java-jdbc.html) in Java code. Use of the PreparedStatement for executing SQL queries not only provides better performance but also shield your Java and J2EE application from SQL Injection attack.   
  
On a similar note, If you are working more on Java EE or J2EE side, then you should also be familiar with other security issues including *Session Fixation attack* or *Cross Site Scripting* attack and how to resolve them. These are some fields and questions where a good answer can make a lot of difference on your selection.   
 **Question 8) Tell me about different Reference types available in Java, e.g. WeakReference, SoftReference or PhantomReference? and Why should you use them?**  
Well, they are different reference types coming from java.lang.ref package and provided to assist Java Garbage Collector in a case of low memory issues. If you wrap an object with [WeakReference](http://javarevisited.blogspot.com/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html) than it will be eligible for garbage collected if there are o strong reference. They can later be reclaimed by Garbage collector if JVM is running low on memory.  
  
The java.util.WeakHashMap is a special Map implementation, whose keys are the object of WeakReference, so if only Map contains the reference of any object and no other, those object can be garbage collected if GC needs memory. See [Java Performance The Definitive Guide](http://www.amazon.com/Java-Performance-The-Definitive-Guide/dp/1449358454?tag=javamysqlanta-20) learn more about how to deal with performance issues in Java. 

[](http://java67.blogspot.com/2013/02/difference-between-jit-and-jvm-in-java.html)

**Question 9) How does get method of HashMap works in Java?**  
Yes, this is still one of the most popular core Java questions for senior developer interviews. You can also expect this question on telephonic round, followed by lot's of follow-up questions as discussed on my post [how does HashMap work in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html).   
  
The short answer to this question is that HashMap is based upon hash table data structure and uses hashCode() method to calculate hash code to find the bucket location on underlying array and equals() method to search the object in the same bucket in case of a collision. See [here](http://java67.blogspot.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) to learn a more about how does get() method of HashMap works in Java.   
  
  
  
**Question 10) Which Two Methods HashMap key Object Should Implement?**  
This is one of the follow-up questions I was saying about in previous questions. Since working of HashMap is based upon hash table data structure, any object which you want to use as key for HashMap or any other hash based collection e.g. Hashtable, or ConcurrentHashMap must implement [equals()](http://java67.blogspot.com/2012/11/difference-between-operator-and-equals-method-in.html) and [hashCode()](http://java67.blogspot.com/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html) method.   
  
The hashCode() is used to find the bucket location i.e. index of underlying array and equals() method is used to find the right object in linked list stored in the bucket in case of a collision. By the way, from Java 8, HashMap also started using tree data structure to store the object in case of collision to reduce worst case performance of HashMap from O(n) to O(logN). See the article for learning more about [how does HashMap handless collision in Java](http://javarevisited.blogspot.com/2016/01/how-does-java-hashmap-or-linkedhahsmap-handles.html).   
  
  
  
**Question 11) Why Should an Object Used As the Key should be Immutable?**   
This is another follow-up of previous core Java interview questions. It's good to test the depth of technical knowledge of candidate by asking more and more question on the same topic. If you know about Immutability, you can answer this question by yourself. The short answer to this question is key should be immutable so that hashCode() method  always return the same value.

Since hash code returned by hashCode() method depends on upon the content of object i.e. values of member variables. If an object is [mutable](http://java67.blogspot.com/2014/01/why-string-class-has-made-immutable-or-final-java.html) than those values can change and so is the hash code. If the same object returns different hash code once you inserted the value in HashMap, you will end up searching in different bucket location and will not able to retrieve the object. That's why a key object should be immutable. It's not a rule enforced by the compiler but you should take care of it as an experienced programmer. See the article for more [advanced Java Collection interview questions](http://javarevisited.blogspot.com/2011/11/collection-interview-questions-answers.html).   
  
  
  
**Question 12) How does ConcurrentHashMap achieves its Scalability?**   
Sometimes this multithreading + collection interview question is also asked as, the difference between ConcurrentHashMap and Hashtable in Java. The problem with [synchronized HashMap](http://java67.blogspot.com/2015/02/how-to-synchronize-hashmap-in-java-with.html) or [Hashtable](http://java67.blogspot.com/2012/08/difference-between-hashmap-and-concurrentHashMap-java-collection.html) was that whole Map is locked when a thread performs any operation with Map.   
  
The java.util.ConcurrentHashMap class solves this problem by using *lock stripping* technique, where the whole map is locked at different segments and only a particular segment is locked during the write operation, not the whole map. The ConcurrentHashMap also achieves it's scalability by allowing lock-free reads as read is a thread-safe operation.  See [here](http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html) for more advanced multi-threading and concurrency questions in Java.   
  
  
  
**Question 13) How do you share an object between threads? or How to pass an object from one thread to another?**  
There are multiple ways to do that e.g. Queues, Exchanger etc, but BlockingQueue using [Producer Consumer pattern](http://java67.blogspot.sg/2012/12/producer-consumer-problem-with-wait-and-notify-example.html) is the easiest way to pass an object from thread to another.  
  
  
  
**Question 14) How do find if your program has a deadlock?**  
By taking thread dump using kill -3, using JConsole or VisualVM), I suggest to prepare this core java interview question in more detail, as Interviewer definitely likes to go with more detail e.g. they will press with questions like, have you really done that in your project or not?  
  
  
**Question 15) How do you avoid deadlock while coding?**  
By ensuring locks are acquire and released in an ordered manner, see [here](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html)for detail answer of this question.  
  
Read more: <http://www.java67.com/2013/07/15-advanced-core-java-interview-questions-answers-senior-experienced-5-6-years-programmers-developers.html#ixzz5WEqZjgW0>

## Top Java design pattern questions and answers

Here is my list of t*op 10 design pattern interview question in Java*. I have also provided an answer to those Java design pattern question as a link. no matter which level of Java interview is you going e.g. programmer, software engineer, senior software engineer in Java, you can expect few question from Java design pattern.  
  
**1. When to use Strategy Design Pattern in Java?**  
[Java design pattern interview question and answers for senior and experience programmer](http://3.bp.blogspot.com/-1lzFJzIgaHk/UF2Ci6kY5pI/AAAAAAAAAes/OYiM7r-DHzc/s1600/17.jpg)Strategy pattern in quite useful for implementing set of related algorithms e.g. compression algorithms, filtering strategies etc. Strategy design pattern allows you to create Context classes, which uses Strategy implementation classes for applying business rules. This pattern follows open closed design principle and quite useful in Java.  
  
One of a good example of Strategy pattern from JDK itself is a Collections.sort() method and [Comparator interface](http://java67.blogspot.sg/2012/10/how-to-sort-object-in-java-comparator-comparable-example.html), which is a strategy interface and defines a strategy for comparing objects. Because of this pattern, we don't need to modify sort() method (closed for modification) to compare any object, at the same time we can implement Comparator interface to define new comparing strategy (open for extension).  
  
  
**2. What is Observer design pattern in Java? When do you use Observer pattern in Java?**  
This is one of the most common Java design pattern interview questions. Observer pattern is based upon notification, there are two kinds of object Subject and Observer. Whenever there is change on subject's state observer will receive notification. See [What is Observer design pattern in Java with real life example](http://javarevisited.blogspot.sg/2011/12/observer-design-pattern-java-example.html) for more details.  
  
  
  
**3. Difference between Strategy and State design Pattern in Java?**  
This is an interesting Java design pattern interview questions as both Strategy and State pattern has the same structure. If you look at UML class diagram for both patterns they look exactly same, but their intent is totally different.  
  
State design pattern is used to define and manage the state of an object, while Strategy pattern is used to define a set of an interchangeable algorithm and let's client choose one of them. So [Strategy pattern](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Ffrom-0-to-1-design-patterns%2F) is a client driven pattern while Object can manage their state itself.  
  
  
  
**4. What is decorator pattern in Java? Can you give an example of Decorator pattern?**  
Decorator pattern is another popular Java design pattern question which is common because of its heavy usage in java.io package. BufferedReader and BufferedWriter are a good example of decorator pattern in Java. See [How to use Decorator pattern in Java](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html) for more details.  
  
  
**5. When to use Composite design Pattern in Java? Have you used previously in your project?**  
This design pattern question is asked on Java interview not just to check familiarity with the Composite pattern but also, whether a candidate has the real life experience or not.  
The*Composite pattern* is also a core Java design pattern, which allows you to treat both whole and part object to treat in a similar way. Client code, which deals with a Composite or individual object doesn't differentiate between them, it is possible because Composite class also implement the same interface as their individual part.  
  
One of the good examples of the Composite pattern from JDK is JPanel class, which is both Component and Container.  When the paint() method is called on JPanel, it internally called paint() method of individual components and let them draw themselves.  
  
On the second part of this design pattern interview question, be truthful, if you have used then say yes, otherwise say that you are familiar with the concept and used it by your own. By the way, always remember, giving an example from your project creates a better impression.  
  
  
  
**6. What is Singleton pattern in Java?**  
Singleton pattern in Java is a pattern which allows only one instance of Singleton class available in the whole application. java.lang.Runtime is a good example of Singleton pattern in Java. There are lot's of follow up questions on Singleton pattern see [10 Java singleton interview question answers](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) for those followups  
  
  
**7. Can you write thread-safe Singleton in Java?**  
There are multiple ways to write thread-safe singleton in Java e.g by writing singleton using double checked locking, by using static Singleton instance initialized during [class loading.](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html) By the way using Java enum to create thread-safe singleton is the most simple way. See [Why Enum singleton is better in Java](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) for more details.  
  
  
**8. When to use Template method design Pattern in Java?**  
The Template pattern is another popular core Java design pattern interview question. I have seen it appear many times in real life project itself. Template pattern outlines an algorithm in form of template method and lets subclass implement individual steps.  
  
The key point to mention, while answering this question is that template method should be final, so that subclass can not override and change steps of the algorithm, but same time individual step should be abstract, so that child classes can implement them.  
  
  
**9. What is Factory pattern in Java? What is the advantage of using a static factory method to create an object?**  
Factory pattern in Java is a creation Java design pattern and favorite on many Java interviews.Factory pattern used to create an object by providing static factory methods. There are many advantages of providing factory methods e.g. caching immutable objects, easy to introduce new objects etc. See [What is Factory pattern in Java and benefits](http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html) for more details.  
  
  
**10. What is the difference between Decorator and Proxy pattern in Java?**  
Another tricky Java design pattern question and trick here is that both Decorator and Proxy implements the interface of the object they decorate or encapsulate. As I said, many Java design pattern can have similar or exactly same structure but they differ in their intent.  
  
Decorator pattern is used to implement functionality on an already created object, while a Proxy pattern is used for controlling access to an object.  
  
One more difference between Decorator and the Proxy design pattern is that Decorator doesn't create an object, instead, it get the object in its constructor, while Proxy actually creates objects. You can also read [Head First Analysis and Design](http://www.amazon.com/dp/0596008678/?tag=javamysqlanta-20) to understand the difference between them.  
  
  
**11. When to use Setter and Constructor Injection in Dependency Injection pattern?**  
Use Setter injection to provide optional dependencies of an object, while use Constructor iInjection to provide a mandatory dependency of an object, without which it can not work. This question is related to [Dependency Injection design pattern](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html) and mostly asked in the context of Spring framework, which is now become a standard for developing Java application.  
  
Since Spring provides IOC container, it also gives you a way to specify dependencies either by using setter methods or constructors. You can also take a look my [previous post](http://javarevisited.blogspot.com/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html) on the same topic.  
  
  
**12. What is difference between Factory and Abstract Factory in Java**  
I have already answered this question in detail with my article with the same title. The main difference is that Abstract Factory creates factory while Factory pattern creates objects. So both abstract the creation logic but one abstract is for factory and other for items. You can see [here](http://javarevisited.blogspot.sg/2013/01/difference-between-factory-and-abstract-factory-design-pattern-java.html) to answer this Java design pattern interview question.  
  
  
**13. When to use Adapter pattern in Java? Have you used it before in your project?**  
Use Adapter pattern when you need to make two class work with incompatible interfaces. Adapter pattern can also be used to encapsulate third party code so that your application only depends upon Adapter, which can adapt itself when third party code changes or you moved to a different third party library.  
  
By the way, this Java design pattern question can also be asked by providing an actual scenario. You can further read [Head First Design Pattern](http://www.amazon.com/dp/0596007124/?tag=javamysqlanta-20) to learn more about Adapter pattern and its real world usage. The book is updated for Java 8 as well so you will learn new, Java 8 way to implement these old design patterns.  
  
Read more: <http://www.java67.com/2012/09/top-10-java-design-pattern-interview-question-answer.html#ixzz5WEtONUU9>

# Java Access Modifiers – Public, Private, Protected & Default

BY CHAITANYA SINGH | FILED UNDER: [OOPS CONCEPT](https://beginnersbook.com/category/oops-concept/)

You must have seen public, private and protected keywords while practising java programs, these are called access modifiers. An access modifier restricts the access of a class, constructor, data member and method in another class. In java we have four access modifiers:  
1. default  
2. private  
3. protected  
4. public

## 1. Default access modifier

When we do not mention any access modifier, it is called default access modifier. The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class. Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package. Lets see an example to understand this:

### Default Access Modifier Example in Java

To understand this example, you must have the knowledge of [packages in java](https://beginnersbook.com/2013/03/packages-in-java/).

In this example we have two classes, Test class is trying to access the default method of Addition class, since class Test belongs to a different package, this program would throw compilation error, because the scope of default modifier is limited to the same package in which it is declared.  
**Addition.java**

package abcpackage;

public class Addition {

/\* Since we didn't mention any access modifier here, it would

\* be considered as default.

\*/

int addTwoNumbers(int a, int b){

return a+b;

}

}

**Test.java**

package xyzpackage;

/\* We are importing the abcpackage

\* but still we will get error because the

\* class we are trying to use has default access

\* modifier.

\*/

import abcpackage.\*;

public class Test {

public static void main(String args[]){

Addition obj = new Addition();

/\* It will throw error because we are trying to access

\* the default method in another package

\*/

obj.addTwoNumbers(10, 21);

}

}

**Output:**

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The method addTwoNumbers(int, int) from the type Addition is not visible

at xyzpackage.Test.main(Test.java:12)

## 2. Private access modifier

The scope of private modifier is limited to the class only.

1. Private Data members and methods are only accessible within the class
2. Class and [Interface](https://beginnersbook.com/2013/05/java-interface/) cannot be declared as private
3. If a class has [private constructor](https://beginnersbook.com/2013/12/java-private-constructor-example/) then you cannot create the object of that class from outside of the class.

Let’s see an example to understand this:

### Private access modifier example in java

This example throws compilation error because we are trying to access the private data member and method of class ABC in the class Example. The private data member and method are only accessible within the class.

class ABC{

private double num = 100;

private int square(int a){

return a\*a;

}

}

public class Example{

public static void main(String args[]){

ABC obj = new ABC();

System.out.println(obj.num);

System.out.println(obj.square(10));

}

}

Output:

Compile - time error

## 3. Protected Access Modifier

Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package. You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub classes.  
Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.

### Protected access modifier example in Java

In this example the class Test which is present in another package is able to call the addTwoNumbers() method, which is declared protected. This is because the Test class extends class Addition and the protected modifier allows the access of protected members in subclasses (in any packages).  
**Addition.java**

package abcpackage;

public class Addition {

protected int addTwoNumbers(int a, int b){

return a+b;

}

}

**Test.java**

package xyzpackage;

import abcpackage.\*;

class Test extends Addition{

public static void main(String args[]){

Test obj = new Test();

System.out.println(obj.addTwoNumbers(11, 22));

}

}

Output:

33

## 4. Public access modifier

The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn’t put any restriction on the access.

### public access modifier example in java

Lets take the same example that we have seen above but this time the method addTwoNumbers() has public modifier and class Test is able to access this method without even extending the Addition class. This is because public modifier has visibility everywhere.  
Addition.java

package abcpackage;

public class Addition {

public int addTwoNumbers(int a, int b){

return a+b;

}

}

Test.java

package xyzpackage;

import abcpackage.\*;

class Test{

public static void main(String args[]){

Addition obj = new Addition();

System.out.println(obj.addTwoNumbers(100, 1));

}

}

Output:

101

Lets see the scope of these access modifiers in tabular form:

## The scope of access modifiers in tabular form

------------+-------+---------+--------------+--------------+--------

| Class | Package | Subclass | Subclass |Outside|

| | |(same package)|(diff package)|Class |

————————————+———————+—————————+——————————----+—————————----—+————————

public | Yes | Yes | Yes | Yes | Yes |

————————————+———————+—————————+—————————----—+—————————----—+————————

protected | Yes | Yes | Yes | Yes | No |

————————————+———————+—————————+————————----——+————————----——+————————

default | Yes | Yes | Yes | No | No |

————————————+———————+—————————+————————----——+————————----——+————————

private | Yes | No | No | No | No |

------------+-------+---------+--------------+--------------+--------

# 6 OOP Concepts in Java with examples [2019]

Java is a class-based object-oriented programming (OOP) language that is built around the concept of objects. OOP concepts (OOP) intend to improve code readability and reusability by defining how to structure a Java program efficiently. The main principles of object-oriented programming are:

1. [Abstraction](https://raygun.com/blog/oop-concepts-java/#abstraction)
2. [Encapsulation](https://raygun.com/blog/oop-concepts-java/#encapsulation)
3. [Inheritance](https://raygun.com/blog/oop-concepts-java/#inheritance)
4. [Polymorphism](https://raygun.com/blog/oop-concepts-java/#polymorphism)
5. [Association](https://raygun.com/blog/oop-concepts-java/#association)
6. [Aggregation](https://raygun.com/blog/oop-concepts-java/#aggregation)
7. [Composition](https://raygun.com/blog/oop-concepts-java/#composition)

Java comes with specific code structures for each OOP principle. For example, the extends keyword for inheritance or getter and setter methods for encapsulation.

## What are OOP concepts in Java?

OOP concepts allow us to create specific interactions between Java objects. They make it possible to reuse code without creating security risks or making a Java program less readable.

Here are the four main principles in more detail.

### Abstraction

Abstraction aims to hide complexity from the users and show them only the relevant information. For example, if you want to drive a car, you don’t need to know about its internal workings. The same is true of Java classes. You can hide internal implementation details by using abstract classes or interfaces. On the abstract level, you only need to define the method signatures (name and parameter list) and let each class implement them in their own way.

**Abstraction in Java:**

* Hides the underlying complexity of data
* Helps avoid repetitive code
* Presents only the signature of internal functionality
* Gives flexibility to programmers to change the implementation of the abstract behaviour
* Partial abstraction (0-100%) can be achieved with abstract classes
* Total abstraction (100%) can be achieved with interfaces

### Encapsulation

Encapsulation allows us to protect the data stored in a class from system-wide access. As its name suggests, it safeguards the internal contents of a class like a real-life capsule. You can implement encapsulation in Java by keeping the fields (class variables) private and providing public getter and setter methods to each of them. Java Beans are examples of fully encapsulated classes.

**Encapsulation in Java:**

* Restricts direct access to data members (fields) of a class.
* Fields are set to private
* Each field has a getter and setter method
* Getter methods return the field
* Setter methods let us change the value of the field

### Polymorphism

[Polymorphism](https://en.wikipedia.org/wiki/Polymorphism) refers to the ability to perform a certain action in different ways. In Java, polymorphism can take two forms: method overloading and method overriding. Method overloading happens when various methods with the same name are present in a class. When they are called they are differentiated by the number, order, and types of their parameters. Method overriding occurs when the child class overrides a method of its parent.

**Polymorphism in Java:**

* The same method name is used several times.
* Different methods of the same name can be called from the object.
* All Java objects can be considered polymorphic (at the minimum, they are of their own type and instances of the Object class).
* Example of static polymorphism in Java is method overloading.
* Example of dynamic polymorphism in Java is method overriding.

### Inheritance

[Inheritance](https://raygun.com/blog/oop-concepts-java/www.linkedin.com/pulse/types-relationships-object-oriented-programming-oop-sarah-el-dawody/) makes it possible to create a child class that inherits the fields and methods of the parent class. The child class can override the values and methods of the parent class, however it’s not necessary. It can also add new data and functionality to its parent. Parent classes are also called superclasses or base classes, while child classes are known as subclasses or derived classes as well. Java uses the extends keyword to implement the principle of inheritance in code.

**Inheritance in Java:**

* A class (child class) can extend another class (parent class) by inheriting its features.
* Implements the DRY (Don’t Repeat Yourself) programming principle.
* Improves code reusability.
* Multilevel inheritance is allowed in Java (a child class can have its own child class as well).
* Multiple inheritances are not allowed in Java (a class can’t extend more than one class).

### Association

Besides the four main principles of OOP, Java also works with three further concepts (association, aggregation, composition) you can make use of when designing your programs. Aggregation is a special form of association, while composition is a special form of aggregation.

[Association](https://javapapers.com/oops/association-aggregation-composition-abstraction-generalization-realization-dependency/) simply means the act of establishing a relationship between two unrelated classes. For example, when you declare two fields of different types (e.g. Car and Bicycle) within the same class and make them interact with each other, you have performed association.

**Association in Java:**

* Two separate classes are associated through their objects.
* The two classes are unrelated, each can exist without the other one.
* Can be a one-to-one, one-to-many, many-to-one, or many-to-many relationship.

### Aggregation

[Aggregation](https://beginnersbook.com/2013/05/aggregation/) is a narrower kind of association. It occurs when there’s a one-way (HAS-A) relationship between the two classes you associate through their objects. For example, every Passenger has a Car but a Car doesn’t necessarily have a Passenger. When you declare the Passenger class, you can create a field of the Car type that shows which car the passenger belongs to. Then, when you instantiate a new Passenger object, you can access the data stored in the related Car as well.

**Aggregation in Java:**

* One-directional association.
* Represents a HAS-A relationship between two classes.
* Only one class is dependent on the other.

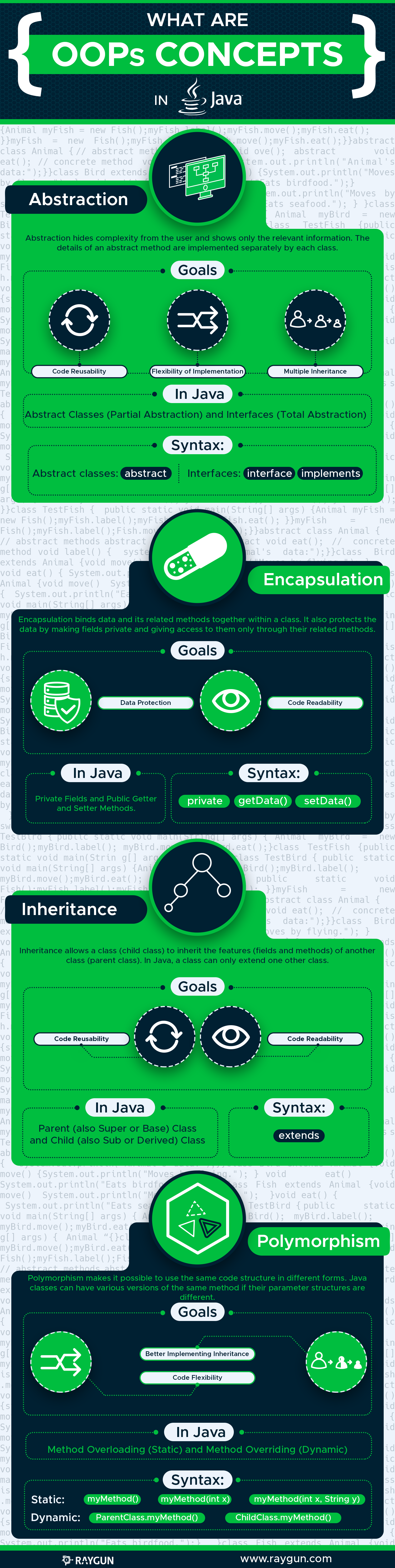
### Composition

[Composition](https://www.infoworld.com/article/3029325/application-development/exploring-association-aggregation-and-composition-in-oop.html)is a stricter form of aggregation. It occurs when the two classes you associate are mutually dependent on each other and can’t exist without each other. For example, take a Car and an Engine class. A Car cannot run without an Engine, while an Engine also can’t function without being built into a Car. This kind of relationship between objects is also called a PART-OF relationship.

Composition in Java:

* A restricted form of aggregation
* Represents a PART-OF relationship between two classes
* Both classes are dependent on each other
* If one class ceases to exist, the other can’t survive alone

### OOP Concepts in Java infographic

[](https://raygun.com/blog/oop-concepts-java/#oop-infographic-modal)

##### Use this infographic on your blog post

×

Embed the "OOP Concepts in Java with examples" infographic on Your Site: Copy and Paste the Code Below:



Close

## What are the characteristics of OOP?

Now, let’s see the real-life characteristics of the four main OOP concepts in Java: abstraction, encapsulation, inheritance, and polymorphism.

### Abstraction

With abstraction, you can hide the internal workings of an object and only show the features the user needs to know about. Java provides two ways to implement abstraction: abstract classes and interfaces. With abstract classes, you can achieve partial abstraction, while interfaces make total (100%) abstraction possible.

#### Abstract classes

An abstract class is a superclass (parent class) that cannot be instantiated. You need to instantiate one of its child classes if you want to create a new object. Abstract classes can have both abstract and concrete methods. Abstract methods contain only the method signature, while concrete methods declare a method body as well. Abstract classes are defined with the abstract keyword.

In the example below, you can see an abstract class called Animal with two abstract and one concrete method.

abstract class Animal {

// abstract methods

abstract void move();

abstract void eat();

// concrete method

void label() {

System.out.println("Animal's data:");

}

}

Extend the Animal abstract class with two child classes: Bird and Fish. Both of them set up their own functionality for the move() and eat() abstract methods.

class Bird extends Animal {

void move() {

System.out.println("Moves by flying.");

}

void eat() {

System.out.println("Eats birdfood.");

}

}

class Fish extends Animal {

void move() {

System.out.println("Moves by swimming.");

}

void eat() {

System.out.println("Eats seafood.");

}

}

Now, test it with the TestBird and TestFish classes. Both call the one concrete (label()) and the two abstract (move() and eat()) methods.

class TestBird {

public static void main(String[] args) {

Animal myBird = new Bird();

myBird.label();

myBird.move();

myBird.eat();

}

}

class TestFish {

public static void main(String[] args) {

Animal myFish = new Fish();

myFish.label();

myFish.move();

myFish.eat();

}

}

In the console, the concrete method has been called from the Animal abstract class, while the two abstract methods have been called from Bird() and Fish(), respectively.

[Console output of TestBird]

Animal's data:

Moves by flying.

Eats birdfood.

[Console output of TestFish]

Animal's data:

Moves by swimming.

Eats seafood.

#### Interfaces

An interface is a 100% abstract class. It can have only static, final, and public fields and abstract methods. It’s frequently referred to as a blueprint of a class as well. Java interfaces allow us to implement multiple inheritance in our code, as a class can implement any number of interfaces. Classes can access an interface using the implements keyword.

In the example, define two interfaces, Animal and Bird. Animal has two abstract methods, while Bird has two static fields and an abstract method.

interface Animal {

public void eat();

public void sound();

}

interface Bird {

int numberOfLegs = 2;

String outerCovering = "feather";

public void fly();

}

The class Eagle implements both interfaces. It defines its own functionality for the three abstract methods. The eat() and sound() methods come from the Animal class, while fly() comes from Bird.

class Eagle implements Animal, Bird {

public void eat() {

System.out.println("Eats reptiles and amphibians.");

}

public void sound() {

System.out.println("Has a high-pitched whistling sound.");

}

public void fly() {

System.out.println("Flies up to 10,000 feet.");

}

}

In the TestEagle test class, instantiate a new Eagle object (called myEagle) and print out all the fields and methods to the console.

As static fields don’t belong to a specific object but to a whole class, you need to access them from the Bird interface instead of the myEagle object.

class TestEagle {

public static void main(String[] args) {

Eagle myEagle = new Eagle();

myEagle.eat();

myEagle.sound();

myEagle.fly();

System.out.println("Number of legs: " + Bird.numberOfLegs);

System.out.println("Outer covering: " + Bird.outerCovering);

}

}

The Java console returns all the information you wanted to access:

[Console output of TestEagle]

Eats reptiles and amphibians.

Has a high-pitched whistling sound.

Flies up to 10,000 feet.

Number of legs: 2

Outer covering: feather

### Encapsulation

With encapsulation, you can protect the fields of a class. To do so, declare the fields as private and providing access to them with getter and setter methods.

The Animal class below is fully encapsulated. It has three private fields and each of them has its own set of getter and setter methods.

class Animal {

private String name;

private double averageWeight;

private int numberOfLegs;

// Getter methods

public String getName() {

return name;

}

public double getAverageWeight() {

return averageWeight;

}

public int getNumberOfLegs() {

return numberOfLegs;

}

// Setter methods

public void setName(String name) {

this.name = name;

}

public void setAverageWeight(double averageWeight) {

this.averageWeight = averageWeight;

}

public void setNumberOfLegs(int numberOfLegs) {

this.numberOfLegs = numberOfLegs;

}

}

The TestAnimal class first sets a value for each field with the setter methods, then prints out the values using the getter methods.

public class TestAnimal {

public static void main(String[] args) {

Animal myAnimal = new Animal();

myAnimal.setName("Eagle");

myAnimal.setAverageWeight(1.5);

myAnimal.setNumberOfLegs(2);

System.out.println("Name: " + myAnimal.getName());

System.out.println("Average weight: " + myAnimal.getAverageWeight() + "kg");

System.out.println("Number of legs: " + myAnimal.getNumberOfLegs());

}

}

As you can see below, the Java console returns properly all the values you set with the setter methods:

[Console output of TestAnimal]

Name: Eagle

Average weight: 1.5kg

Number of legs: 2

### Inheritance

Inheritance allows us to extend a class with child classes that inherit the fields and methods of the parent class. It’s an excellent way to achieve code reusability. In Java, we need to use the extends keyword to create a child class.

In the example, the Eagle class extends the Bird parent class. It inherits all of its fields and methods, plus defines two extra fields that belong only to Eagle.

class Bird {

public String reproduction = "egg";

public String outerCovering = "feather";

public void flyUp() {

System.out.println("Flying up...");

}

public void flyDown() {

System.out.println("Flying down...");

}

}

class Eagle extends Bird {

public String name = "eagle";

public int lifespan = 15;

}

The TestEagle class instantiates a new Eagle object and prints out all the information (both the inherited fields and methods and the two extra fields defined in the Eagle class).

class TestEagle {

public static void main(String[] args) {

Eagle myEagle = new Eagle();

System.out.println("Name: " + myEagle.name); System.out.println("Reproduction: " + myEagle.reproduction);

System.out.println("Outer covering: " + myEagle.outerCovering);

System.out.println("Lifespan: " + myEagle.lifespan);

myEagle.flyUp();

myEagle.flyDown();

}

}

You can see the console output below:

[Console output of TestEagle]

Reproduction: another egg

Outer covering: feather

Lifespan: 15

Flying up...

Flying down...

### Polymorphism

Polymorphism makes it possible to use the same entity in different forms. In Java, this means that you can declare several methods with the same name until they are different in certain characteristics. Java provides us with two ways to implement polymorphism: method overloading and method overriding.

#### Static polymorphism

Method overloading means that you can have several methods with the same name within a class. However, the number, names, or types of their parameters need to be different.

For example, the Bird() class below has three fly() methods. The first one doesn’t have any parameters, the second one has one parameter (height), and the third one has two parameters (name and height).

class Bird {

public void fly() {

System.out.println("The bird is flying.");

}

public void fly(int height) {

System.out.println("The bird is flying " + height + " feet high.");

}

public void fly(String name, int height) {

System.out.println("The " + name + " is flying " + height + " feet high.");

}

}

The test class instantiates a new Bird object and calls the fly() method three times. Firstly, without parameters, secondly, with one integer parameter for height, and thirdly, with two parameters for name and height.

class TestBird {

public static void main(String[] args) {

Bird myBird = new Bird();

myBird.fly();

myBird.fly(10000);

myBird.fly("eagle", 10000);

}

}

In the console, we can see that Java could have differentiated the three polymorphic fly() methods:

[Console output of TestBird]

The bird is flying.

The bird is flying 10000 feet high.

The eagle is flying 10000 feet high.

#### Dynamic polymorphism

By using the method overriding feature of Java, you can override the methods of a parent class from its child class.

The Bird class extends the Animal class in the example below. Both have an eat() method. By default, Bird inherits its parent’s eat() method. However, as it also defines its own eat() method, Java will override the original method and call eat() from the child class.

class Animal {

public void eat() {

System.out.println("This animal eats insects.");

}

}

class Bird extends Animal {

public void eat() {

System.out.println("This bird eats seeds.");

}

}

The TestBird class first instantiates a new Animal object and calls its eat() method. Then, it also creates a Bird object and calls the polymorphic eat() method again.

class TestBird {

public static void main(String[] args) {

Animal myAnimal = new Animal();

myAnimal.eat();

Bird myBird = new Bird();

myBird.eat();

}

}

The console returns the values of the relevant methods properly. Therefore Java could have differentiated the two eat() methods indeed.

[Console output of TestBird]

This animal eats insects.

This bird eats seeds.

## Conclusion

OOP concepts in Java define how to structure a Java problem more efficiently.

1. [Explain the Java Memory Model in detail, what is happens-before,Volatile Variables,Memory Visibility? Also give example.](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point1)
2. [Discuss the differences between the Serializable and Externalizable interfaces in Java. When would you use one over the other for object serialization, and what are the potential performance implications?](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point2)
3. [Could you provide the differences between Heap and Stack Memory in the context of Java, and also provide insights into how these memory areas are used?](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point3)
4. [Explain the differences between abstract classes and interfaces in Java. When should you use one over the other?](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point4)
5. [Explain the concept of method overloading and method overriding in Java. What are the rules for each?](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point5)
6. [Explain the concept of Java generics. What are some benefits of using generics, and what are some common use cases?](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point6)
7. [Explain the difference between final, finally, and finalize in Java.](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point7)
8. [Explain the difference between the Comparable and Comparator interfaces in Java.](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point8)
9. [What are the key differences between ArrayList and LinkedList in Java?](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point9)
10. [What is the purpose of the hashCode() and equals() methods in Java?](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point10)
11. [What are some common uses of Java Reflection API?](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point11)
12. [Define the role of a ClassLoader in Java and its significance within the runtime environment.](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point12)
13. [What is the difference between Checked and Unchecked Exceptions in Java?](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point13)
14. [What is the Singleton design pattern in Java, and how can it be implemented? Include a code example.](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point14)
15. [What is the Java ClassLoader, and what are its primary functions? Include a code example.](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point15)
16. [What is the purpose of the "synchronized" keyword in Java, and when should it be used? Include a code example.](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point16)
17. [What is the difference between a shallow copy and a deep copy in Java, and when should you use each one? Include a code example.](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point17)
18. [Tell me about clone() function, give some example.](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point18)
19. [How has Java evolved over the years, and what are some key features introduced in recent Java versions (Java 8 onwards)?](https://www.codekatha.com/2023/04/Java-interview-questions-for-10-years-experience.html#point19)

**Explain the Java Memory Model in detail, what is happens-before,Volatile Variables,Memory Visibility? Also give example.**

The Java Memory Model (JMM) defines how threads in a multi-threaded Java program interact with memory and each other. Understanding JMM is crucial for ensuring thread safety in high-concurrency scenarios.

**Concepts in the Java Memory Model:** **1. Happens-Before Relationship:**

The "happens-before" relationship is a key concept in JMM. It establishes a partial order among memory operations. If one operation happens before another, the effects of the first operation are visible to the second. For example, if you write to a variable (Operation A) and then read from it (Operation B) and Operation A happens before Operation B, the value written by Operation A is guaranteed to be visible to Operation B.

**2. Volatile Variables:**

A variable declared as volatile ensures a "happens-before" relationship for read and write operations on that variable. When a thread writes to a volatile variable, it flushes its local memory to main memory, ensuring that other threads see the updated value when they read the variable. Volatile variables are useful for simple synchronization scenarios, but they may not be sufficient for complex operations that require atomicity.

**3. Memory Visibility:**

Memory visibility is the property that ensures changes made by one thread to shared variables are visible to other threads. Without proper synchronization, changes made by one thread may not be visible to others, leading to data inconsistencies and bugs. The "happens-before" relationship and synchronization mechanisms like locks and monitors ensure memory visibility.

**Ensuring Thread Safety:**

To ensure thread safety in Java applications, especially in high-concurrency scenarios, you can follow these principles:

**1. Use Synchronization:**

Use synchronized blocks or methods to protect critical sections of code. This ensures that only one thread can access the synchronized block at a time, preventing concurrent modification of shared resources.

package codeKatha;

public class **SynchronizationExample** {

private int sharedVariable = 0;

public synchronized void **increment**() {

sharedVariable++;

}

}

**2. Use Volatile Variables:**

Declare variables as volatile when they are shared among multiple threads and need to be accessed and updated safely.

package codeKatha;

public class **VolatileExample** {

private volatile boolean flag = false;

public void **toggleFlag**() {

flag = !flag;

}

}

**3. Use Thread-Safe Data Structures:**

Prefer using thread-safe data structures from the `java.util.concurrent` package, such as `ConcurrentHashMap` or `ConcurrentLinkedQueue`, when dealing with shared collections or queues.

**4. Leverage Atomic Operations:**

Use atomic classes like `AtomicInteger` or `AtomicLong` to perform compound actions atomically without the need for locks or synchronized blocks.

package codeKatha;

import java.util.concurrent.atomic.AtomicInteger;

public class **AtomicExample** {

private AtomicInteger counter = new **AtomicInteger**(0);

public void **increment**() {

counter.incrementAndGet();

}

}

**Discuss the differences between the Serializable and Externalizable interfaces in Java. When would you use one over the other for object serialization, and what are the potential performance implications?**

In Java, the Serializable and Externalizable interfaces are used for object serialization, but they differ in terms of customization and performance. Let's explore the differences and when to use one over the other.

**Serializable Interface:**

The Serializable interface is a marker interface that indicates that a class can be serialized (converted into a byte stream) and deserialized (reconstructed from a byte stream). When a class implements Serializable, the serialization and deserialization process is handled by the Java runtime, and you have limited control over the process.

package codeKatha;

import java.io.\*;

public class **SerializableExample** implements **Serializable** {

private int data;

**// Constructors, getters, setters, and other methods...**

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

**// Serialization**

try (ObjectOutputStream oos = new **ObjectOutputStream**(new **FileOutputStream**("object.ser"))) {

SerializableExample obj = new **SerializableExample**();

oos.writeObject(obj);

} catch (IOException e) {

e.printStackTrace();

}

**// Deserialization**

try (ObjectInputStream ois = new **ObjectInputStream**(new **FileInputStream**("object.ser"))) {

SerializableExample obj = (SerializableExample) ois.readObject();

**// Use the deserialized object**

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

**Externalizable Interface:**

The Externalizable interface is also used for object serialization, but it provides more control and customization over the serialization and deserialization process. When a class implements Externalizable, you must implement the `writeExternal` and `readExternal` methods, which define how the object is serialized and deserialized.

package codeKatha;

import java.io.\*;

public class **ExternalizableExample** implements **Externalizable** {

private int data;

**// Constructors, getters, setters, and other methods...**

@Override

public void **writeExternal**(ObjectOutput out) throws IOException {

**// Custom serialization logic**

out.writeInt(data);

}

@Override

public void **readExternal**(ObjectInput in) throws IOException, ClassNotFoundException {

**// Custom deserialization logic**

data = in.readInt();

}

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

**// Serialization**

try (ObjectOutputStream oos = new **ObjectOutputStream**(new **FileOutputStream**("object.ser"))) {

ExternalizableExample obj = new **ExternalizableExample**();

oos.writeObject(obj);

} catch (IOException e) {

e.printStackTrace();

}

**// Deserialization**

try (ObjectInputStream ois = new **ObjectInputStream**(new **FileInputStream**("object.ser"))) {

ExternalizableExample obj = (ExternalizableExample) ois.readObject();

**// Use the deserialized object**

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

**When to Use Serializable vs. Externalizable:**

* Use **Serializable** when you want a simple and straightforward way to serialize and deserialize objects with minimal effort. It's suitable for most cases and provides good performance for many scenarios.
* Use **Externalizable** when you need full control over the serialization and deserialization process. This is useful when you want to customize the format of the serialized data, omit certain fields, or perform encryption/decryption during serialization. However, keep in mind that Externalizable requires more code and can be less convenient than Serializable.

**Performance Implications:**

* **Serializable** can have better default performance in some cases because it relies on Java's built-in serialization mechanisms. However, it may serialize more data than you need, leading to larger serialized objects.
* **Externalizable** allows you to optimize serialization by selectively choosing what data to write and read. It can result in smaller and more efficient serialized objects. However, the customization overhead may offset these gains in simple cases.

**Could you provide the differences between Heap and Stack Memory in the context of Java, and also provide insights into how these memory areas are used?**

In Java, **memory management** plays a crucial role in determining **how objects are stored and accessed during program execution**. Heap and Stack Memory are two distinct regions where different types of data are managed. Understanding their differences is essential for efficient memory utilisation and managing object lifecycles.

**Stack Memory**

**Stack Memory is a region used for storing method calls, local variables, and references to objects**. It **operates in a Last-In-First-Out (LIFO) manner**, resembling a stack of items. Each method call creates a new frame in the stack, containing variables specific to that method.

Stack Memory is**relatively fast for allocation and deallocation** because it follows a strict order. However, it has **limited space** and is typically used for small, short-lived data. Primitive data types and references to objects are often stored here.

**Example: Using Stack Memory**

package codeKatha;

public class **StackMemoryExample** {

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

int a = 5;

int b = 10;

int sum = addNumbers(a, b);

System.out.println("Sum: " + sum);

}

public static int **addNumbers**(int x, int y) {

int result = x + y;

return result;

}

}

In this example, the variables 'a', 'b', 'x', 'y', 'result', and 'sum' are stored in the Stack Memory. As the methods are called and return, their corresponding frames are pushed and popped from the stack.

Stack Memory

-----------------

[addNumbers Frame]

result = 15

y = 10

x = 5

[main Frame]

sum = 15

b = 10

a = 5

**Heap Memory**

Heap Memory is a region **used for dynamic memory allocation**, primarily **for objects that have varying lifetimes**. Unlike Stack Memory, Heap Memory **doesn't have a strict order**, and objects can be allocated and deallocated in any order.

Objects stored in the Heap are accessed through references stored in the Stack. Objects in Heap Memory can exist beyond the scope of a single method and **can be shared among multiple methods or even different threads**.

**Example: Using Heap Memory**

package codeKatha;

public class **HeapMemoryExample** {

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

Person person1 = new **Person**("Shanav", 25);

Person person2 = new **Person**("Advait", 30);

person1.sayHello();

person2.sayHello();

}

}

class **Person** {

private String name;

private int age;

public **Person**(String name, int age) {

this.name = name;

this.age = age;

}

public void **sayHello**() {

System.out.println("Hello, my name is " + name + " and I'm " + age + " years old.");

}

}

In this example, the **'Person' objects are created in the Heap Memory using the 'new' keyword**. The **references to these objects ('person1' and 'person2') are stored in the Stack Memory**. The objects can be accessed beyond the scope of the 'main' method, as shown in the 'sayHello' method.

Heap Memory

-----------------

|

[person1 Object] --> Person("Shanav", 25)

|

[person2 Object] --> Person("Advait", 30)

-----------------

In the Stack Memory example, you can see the method call frames and their associated local variables. In the Heap Memory example, the objects are allocated in the Heap, and the references to those objects are stored in the Stack.

**Utilisation and Implications**

Understanding the distinctions between Heap and Stack Memory is vital for efficient memory usage. **Stack Memory** is suited **for small and short-lived data**, while **Heap Memory** is used **for dynamically allocated objects** with varying lifetimes. Effective memory management ensures that objects are released when no longer needed, preventing memory leaks and improving program performance.

By carefully utilising Stack and Heap Memory, developers can optimise their programs for both memory efficiency and object lifecycle management.

**Explain the differences between abstract classes and interfaces in Java. When should you use one over the other?**

Abstract classes and interfaces are two mechanisms in Java for defining contracts and providing abstraction. They have some similarities but also differ in key ways:

* **Abstract Classes:** Can have both abstract and non-abstract methods, can have instance variables, can have constructors, can have a partial implementation, and support inheritance. A class can extend only one abstract class.
* **Interfaces:** Can have only abstract methods (before Java 8), can have default and static methods (Java 8 onwards), cannot have instance variables, cannot have constructors, provide no implementation, and support multiple inheritance. A class can implement multiple interfaces.

**When to use abstract classes:**

* If you want to provide a partial implementation or share common functionality across related classes.
* If you want to use instance variables and constructors in your abstraction.
* If you want to enforce a specific inheritance hierarchy.

**When to use interfaces:**

* If you want to provide a contract that multiple unrelated classes can implement.
* If you need a class to inherit from multiple abstractions.
* If you want to provide a common API for different implementations.

**Explain the concept of method overloading and method overriding in Java. What are the rules for each?**

**Method Overloading:** Method overloading is the concept of having multiple methods with the same name but different parameter lists in the same class. The methods can have different return types and access modifiers. The compiler differentiates these methods based on the number, type, and order of the parameters. Method overloading is a form of compile-time polymorphism.

**Rules for method overloading:**

* Methods must have the same name but different parameter lists.
* Methods can have different return types, access modifiers, and exception lists.
* Constructor overloading is also possible in Java.

**Method Overriding:** Method overriding is the concept of providing a new implementation for an existing method in a subclass. The subclass method must have the same method signature (name, return type, and parameters) as the method in the superclass. Method overriding is a form of runtime polymorphism and is used to provide specialized behavior in subclasses.

**Rules for method overriding:**

* Methods must have the same name, return type, and parameter list as the superclass method.
* The access level of the overriding method cannot be more restrictive than the superclass method.
* If the superclass method throws checked exceptions, the overriding method can throw the same exceptions, subclasses of those exceptions, or no exception, but it cannot throw a new checked exception or a higher-level exception.
* The @Override annotation can be used to indicate that a method is intended to override a superclass method, helping the compiler catch errors.

**Explain the concept of Java generics. What are some benefits of using generics, and what are some common use cases?**

Java generics is a language feature introduced in Java 5 that allows the creation of generic classes, interfaces, and methods that can operate on different types of objects while providing type safety and code reusability. With generics, you can define a single class or method that works with different data types, while maintaining type safety at compile time.

**Benefits of using generics:**

* **Type safety:** Generics help ensure type safety by checking the types of objects at compile time.This helps to catch type-related errors early and reduces the chances of runtime ClassCastException.
* **Code reusability:** Generics enable you to write generic classes, interfaces, and methods that can be reused with different types, reducing code duplication and increasing maintainability.
* **Improved readability:** Generics make the code more expressive and easier to read, as the types of objects being used are explicitly defined.

**Common use cases for generics:**

* **Collections:** One of the most common use cases for generics is in the Java Collections Framework. Generics allow you to create type-safe collections like List<String>, Set<Integer>, and Map<String, Integer>, ensuring that only the specified type of objects can be added or retrieved from the collection.
* **Custom generic classes and interfaces:** You can create your own generic classes and interfaces to handle multiple data types while maintaining type safety. For example, you could create a generic Pair class that can store two objects of different types:

public class **Pair**<K, V> {

private K key;

private V value;

public **Pair**(K key, V value) {

this.key = key;

this.value = value;

}

**// Getters and setters**

}

* **Generic methods:** You can create generic methods that can be used with different types of arguments. For example, you could create a generic method to find the maximum value in a list of comparable elements:

public static <T extends **Comparable**<T>> T **findMax**(List<T> list) {

T max = list.get(0);

for (T item : list) {

if (item.compareTo(max) > 0) {

max = item;

}

}

return max;

}

**Explain the difference between final, finally, and finalize in Java.**

**final:** The final keyword in Java can be applied to variables, methods, and classes. It serves different purposes depending on the context:

* **final variables:** A final variable can be assigned a value only once, either at the time of declaration or within a constructor. Once assigned, its value cannot be changed. Final variables are often used to create constants.
* **final methods:** A final method cannot be overridden by a subclass, ensuring that the method's behavior remains consistent across the class hierarchy.
* **final classes:** A final class cannot be extended, preventing the creation of subclasses. This is useful for creating immutable classes or classes with sensitive behavior that should not be altered.

**finally:** The finally keyword is used in conjunction with a try-catch block in Java's exception handling mechanism. A finally block is used to execute code that must always be executed, regardless of whether an exception is thrown or not. It is typically used to clean up resources, such as closing file handles or database connections.

try {

**// Code that might throw an exception**

} catch (IOException e) {

**// Code to handle the exception**

} finally {

**// Code that will always be executed, regardless of whether an exception was thrown or not**

**// For example: close file handles or database connections**

}

**finalize:** The finalize() method is a protected method of the java.lang.Object class. It is called by the garbage collector before an object is removed from memory. The finalize() method provides an opportunity to perform cleanup operations before the object is garbage collected. However, it is not recommended to rely on the finalize() method for resource management, as there is no guarantee when (or even if) the garbage collector will call it. Instead, use the try-with-resources statement or the finally block for resource cleanup.

@Override

protected void **finalize**() throws Throwable {

**// Cleanup code, for example: release resources or close connections**

**// Note: It is not recommended to rely on finalize() for resource management**

}

**Explain the difference between the Comparable and Comparator interfaces in Java.**

Both Comparable and Comparator interfaces in Java are used for sorting collections of objects. However, they serve different purposes and are used in different scenarios:

**Comparable:** The Comparable interface is used to define a natural ordering for objects of a specific class. It contains a single method, compareTo(), which compares two objects of the same class. To use the Comparable interface:

* A class must implement the Comparable<T> interface and define the compareTo() method.
* The compareTo() method should return a negative, zero, or positive integer, depending on whether the current object is less than, equal to, or greater than the object being compared.

public class **Employee** implements **Comparable**<Employee> {

private int id;

private String name;

**// Constructor, getters, and setters**

@Override

public int **compareTo**(Employee other) {

return Integer.compare(this.id, other.id);

}

}

**Comparator:** The Comparator interface is used to define a custom ordering for objects of a specific class. It contains a single method, compare(), which compares two objects of the same class. To use the Comparator interface:

* Create a separate class that implements the Comparator<T> interface and defines the compare() method.
* The compare() method should return a negative, zero, or positive integer, depending on whether the first object is less than, equal to, or greater than the second object.
* Use the custom Comparator class to sort a collection of objects using the Collections.sort() method or other sorting methods.

public class **EmployeeNameComparator** implements **Comparator**<Employee> {

@Override

public int **compare**(Employee e1, Employee e2) {

return e1.getName().compareTo(e2.getName());

}

}

**// Sorting a list of Employee objects by name**

Collections.sort(employeeList, new **EmployeeNameComparator**());

In summary, use the Comparable interface to define a natural ordering for objects of a class, and use the Comparator interface to define a custom ordering for objects of a class without modifying the original class.

**What are the key differences between ArrayList and LinkedList in Java?**

ArrayList and LinkedList are both implementations of the List interface in Java, but they have different underlying data structures and performance characteristics:

**ArrayList:**

* Underlying data structure: An ArrayList uses a dynamic array to store its elements.
* Access time: ArrayList provides fast random access (O(1)) to its elements due to its array-based structure.
* Insertion and deletion: Inserting or deleting elements in the middle of an ArrayList is slow (O(n)) because it requires shifting elements to maintain a contiguous array.
* Memory overhead: ArrayLists have a lower memory overhead compared to LinkedLists, as they do not require additional memory for pointers.
* Use cases: ArrayList is suitable for scenarios where frequent random access is needed and insertions/deletions are infrequent or mostly occur at the end of the list.

List<String> arrayList = new **ArrayList**<>();

**LinkedList:**

* Underlying data structure: A LinkedList uses a doubly-linked list to store its elements.
* Access time: LinkedList provides slow random access (O(n)) to its elements, as it needs to traverse the list from the beginning or the end.
* Insertion and deletion: Inserting or deleting elements in the middle of a LinkedList is fast (O(1)) if the reference to the node is already known, as it only requires updating the pointers.
* Memory overhead: LinkedLists have a higher memory overhead compared to ArrayLists, as they require additional memory for pointers (next and previous) for each element.
* Use cases: LinkedList is suitable for scenarios where frequent insertions and deletions are needed, and random access is not a primary requirement.

List<String> linkedList = new **LinkedList**<>();

**What is the purpose of the hashCode() and equals() methods in Java?**

The hashCode() and equals() methods are used to compare objects for equality in Java. They play a crucial role when objects are used as keys in hash-based collections, such as HashMap or HashSet.

**hashCode():** The hashCode() method returns an integer hash code representing the object. Objects that are equal according to their equals() method should return the same hash code. However, two objects with the same hash code do not necessarily need to be equal.

@Override

public int **hashCode**() {

**// Calculate and return hash code**

}

**equals():** The equals() method is used to compare two objects for equality. It should return true if the objects are equal and false otherwise. The equals() method should follow these rules:

* Reflexive: x.equals(x) should return true for any non-null reference value x.
* Symmetric: x.equals(y) should return the same value as y.equals(x) for any non-null reference values x and y.
* Transitive: If x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should also return true for any non-null reference values x, y, and z.
* Consistent: Multiple invocations of x.equals(y) should consistently return the same value, provided that neither x nor y are modified between invocations.
* For any non-null reference value x, x.equals(null) should return false.

@Override

public boolean **equals**(Object obj) {

**// Compare and return true if objects are equal, false otherwise**

}

Implementing the hashCode() and equals() methods correctly is important for the proper functioning of hash-based collections. If two objects are considered equal according to their equals() method but have different hash codes, they may end up in different buckets in a hash-based collection, leading to unexpected behavior.

**What are some common uses of Java Reflection API?**

Java Reflection API allows a program to inspect and interact with its own code at runtime. It provides the ability to analyze and manipulate classes, fields, methods, and other elements of the Java code. Some common uses of Java Reflection API include:

* **Dynamic object creation:** Reflection allows you to instantiate objects without knowing their class at compile time. This can be useful for creating objects based on user input or configuration files.
* **Method invocation:** With Reflection, you can invoke methods on objects without knowing the methods at compile time. This is helpful for implementing features like plugins or scripting languages that interact with Java code.
* **Accessing private members:** Reflection can be used to access private fields and methods of an object, which can be useful for testing or debugging purposes. However, using Reflection to break encapsulation in regular code is discouraged.
* **Inspecting class metadata:** Reflection allows you to retrieve information about classes, such as their fields, methods, constructors, and annotations. This can be used for generating documentation, implementing custom serialization, or validating code.

**Example:**

package codeKatha;

import java.lang.reflect.Field;

class **Student** {

private String name;

private int age;

public **Student**(String name, int age) {

this.name = name;

this.age = age;

}

public void **display**() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

public class **ReflectionExample** {

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

**// Create an instance of the Student class**

Student student = new **Student**("Advait", 20);

**// Use Reflection to access and modify private fields**

Class<?> studentClass = student.getClass();

try {

**// Access the 'name' field**

Field nameField = studentClass.getDeclaredField("name");

nameField.setAccessible(true); **// Allow access to private field**

String nameValue = (String) nameField.get(student);

System.out.println("Name Field: " + nameValue);

**// Access the 'age' field**

Field ageField = studentClass.getDeclaredField("age");

ageField.setAccessible(true);

int ageValue = (int) ageField.get(student);

System.out.println("Age Field: " + ageValue);

**// Modify the 'name' field**

nameField.set(student, "Shanav");

System.out.println("Updated Name Field: " + student.display());

} catch (NoSuchFieldException | IllegalAccessException e) {

e.printStackTrace();

}

}

}

In this example:  
We define a Student class with private fields name and age.  
In the ReflectionExample class, we create an instance of Student.  
We use Java Reflection to access and modify the private fields of the Student class.  
We get the class object for Student using getClass().  
We access and modify the private fields by name using getDeclaredField() and setAccessible(true) to bypass access control checks.  
We retrieve and print the values of the fields, both before and after modifying them.  
  
Please note that using Reflection to access or modify private fields should be done with caution, as it can break encapsulation and lead to unexpected behavior. It's typically used in special cases where you need to interact with classes in a way that can't be achieved through regular means.

**Define the role of a ClassLoader in Java and its significance within the runtime environment.**

**ClassLoader Role and Significance:**

A ClassLoader in Java is a crucial component responsible for dynamically loading classes during runtime. It plays a vital role in the Java Virtual Machine (JVM) by locating and loading class files from various sources like the file system, network, or other custom sources. The ClassLoader ensures that classes are available for execution as needed, contributing to Java's flexibility and extensibility.

**ClassLoader Hierarchy:**

Java uses a hierarchical ClassLoader system, comprising the following levels:

* **Bootstrap ClassLoader:** The top-level ClassLoader responsible for loading core Java classes provided by the JVM itself.
* **Extension ClassLoader:** Loads classes from the Java Standard Extension libraries.
* **Application ClassLoader:** Also known as the system ClassLoader, it loads classes from the application's classpath.
* **Custom ClassLoaders:** Developers can create custom ClassLoaders to load classes from non-standard sources.

**ClassLoader Workflow:**

When a class is needed during runtime, the ClassLoader follows a specific sequence to locate and load it:

1. **Bootstrap ClassLoader:** Checks if the class is a core Java class provided by the JVM.
2. **Extension ClassLoader:** Looks for the class in Java's standard extension libraries.
3. **Application ClassLoader:** Searches for the class in the application's classpath.
4. **Custom ClassLoaders:** If the class is not found in the above ClassLoaders, custom ClassLoaders are consulted based on the application's logic.

**ClassLoader Example:**

package codeKatha;

public class **ClassLoaderExample** {

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

**// Get the class loader for a class**

ClassLoader classLoader = String.class.getClassLoader();

**// Print the class loader**

System.out.println("ClassLoader for String: " + classLoader);

}

}

In this example, we obtain the ClassLoader for the String class, which is typically the Bootstrap ClassLoader as it's a core Java class.

**ClassLoader Significance:**

* ClassLoader enables dynamic loading of classes, facilitating features like reflection, plugins, and hot swapping.
* It supports isolation between classes loaded by different ClassLoaders, enhancing security and avoiding conflicts.
* ClassLoader hierarchies help manage classloading efficiently and promote code modularity.

ClassLoader is a fundamental component of the Java runtime environment, contributing to its adaptability, security, and extensibility.

**What is the difference between Checked and Unchecked Exceptions in Java?**

Exceptions in Java are categorized into two types: Checked Exceptions and Unchecked Exceptions.

**Checked Exceptions:** Checked Exceptions are exceptions that are checked by the compiler at compile time. These exceptions must be explicitly handled using a try-catch block or declared in the method's signature using the "throws" keyword. Examples of checked exceptions are IOException, SQLException, and ClassNotFoundException.

public void **readFile**(String fileName) throws IOException {

**// Read file and handle IOException**

}

**Unchecked Exceptions:** Unchecked Exceptions are exceptions that are not checked by the compiler. They usually represent programming errors, such as accessing an element beyond the bounds of an array, or attempting to access a null object. Unchecked Exceptions are subclasses of RuntimeException and do not need to be explicitly handled or declared. Examples of unchecked exceptions are NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.

public int **divide**(int a, int b) {

**// Divide and potentially throw ArithmeticException if b is 0**

}

It is a good practice to handle checked exceptions and to avoid using unchecked exceptions for expected error conditions. Unchecked exceptions should be reserved for programming errors that should be fixed rather than caught and handled at runtime.

**What is the Singleton design pattern in Java, and how can it be implemented? Include a code example.**

The Singleton design pattern ensures that a class has only one instance and provides a global point of access to that instance. This pattern is useful when you want to restrict the creation of multiple objects of a particular class and ensure that there's a single instance shared across the entire application.

**Implementation of Singleton Pattern:**

Here's a simple implementation of the Singleton pattern in Java:

package codeKatha;

public class **Singleton** {

**// Private static instance variable to hold the single instance**

private static Singleton instance;

**// Private constructor to prevent external instantiation**

private **Singleton**() {

}

**// Public method to provide access to the single instance**

public static Singleton **getInstance**() {

**// Lazy initialization: create the instance only when needed**

if (instance == null) {

instance = new **Singleton**();

}

return instance;

}

**// Other methods and properties of the Singleton class**

}

In this example:  
1.We have a private static instance variable instance that holds the single instance of the class.  
2.The constructor Singleton is private, which means that it cannot be accessed from outside the class, preventing external instantiation.  
3.The public static method getInstance is used to provide access to the single instance. It uses lazy initialization, meaning the instance is created only when getInstance is called for the first time.  
**Usage of Singleton Pattern:**

You can use the Singleton pattern as follows:

package codeKatha;

public class **SingletonExample** {

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

**// Get the singleton instance**

Singleton singleton1 = Singleton.getInstance();

Singleton singleton2 = Singleton.getInstance();

**// Both instances are the same**

System.out.println(singleton1 == singleton2); **// Output: true**

}

}

In this usage example, we obtain the Singleton instance twice using getInstance, and we can see that both references (singleton1 and singleton2) point to the same instance. This ensures that there's only one instance of the Singleton class throughout the application.

The Singleton pattern is helpful when you want to control access to a shared resource or configuration, ensure that there's only one instance of a manager or controller, or implement a caching mechanism where you want a single cache instance.

**What is the Java ClassLoader, and what are its primary functions? Include a code example.**

The Java ClassLoader is a part of the Java Runtime Environment (JRE) that dynamically loads Java classes into memory when they are needed.

try {

Class<?> myClass = Class.forName("com.example.MyClass");

Object myInstance = myClass.newInstance();

} catch (ClassNotFoundException | IllegalAccessException | InstantiationException e) {

e.printStackTrace();

}

Its primary functions are:

* **Loading:** Reads the binary data of a class from various sources, such as the file system, a JAR file, or a network location, and converts it into an instance of java.lang.Class.
* **Linking:** Verifies the correctness of the class, resolves symbolic references, and prepares the class for execution by allocating static fields and initializing static variables.
* **Initialization:** Executes the static initializer block of the class, if present, and assigns the initial values to the static fields.

**What is the purpose of the "synchronized" keyword in Java, and when should it be used? Include a code example.**

The "synchronized" keyword in Java is used to ensure that only one thread can access a critical section of code or a shared resource at a time, preventing race conditions and data inconsistency in multithreaded programs.

Consider the following code example:

class **BankAccount** {

private double balance;

public synchronized void **deposit**(double amount) {

balance += amount;

}

public synchronized void **withdraw**(double amount) {

balance -= amount;

}

public synchronized double **getBalance**() {

return balance;

}

}

The synchronized keyword should be used when:

* You have a shared resource or a critical section of code, such as the BankAccount's balance, that can be accessed and modified by multiple threads concurrently.
* You need to ensure that only one thread can access the resource or execute the code block at a time to prevent data inconsistency or race conditions.

**What is the difference between a shallow copy and a deep copy in Java, and when should you use each one? Include a code example.**

**Shallow Copy:**

A shallow copy **creates a new object that is a copy of the original object**, but it does not create new copies of the objects referenced by the original. Instead, **it copies references to the same objects**. As a result, **changes to the objects inside the copy are reflected in the original and vice versa**.

package codeKatha;

class **Student** {

String name;

Course course;

public **Student**(String name, Course course) {

this.name = name;

this.course = course;

}

}

class **Course** {

String name;

public **Course**(String name) {

this.name = name;

}

}

public class **ShallowCopyExample** {

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

Course course = new **Course**("Computer Science");

Student originalStudent = new **Student**("Advait", course);

Student shallowCopyStudent = new **Student**(originalStudent.name, originalStudent.course);

**// Changes in course name affect both original and shallow copy**

shallowCopyStudent.course.name = "Mathematics";

System.out.println(originalStudent.course.name); **// Output: Mathematics**

}

}

**Deep Copy:**

A deep copy **creates a new object and also recursively creates new copies of the objects** referenced by the original. This ensures that **changes in the copied objects do not affect the original or vice versa**.

package codeKatha;

public class **DeepCopyExample** {

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

Course course = new **Course**("Computer Science");

Student originalStudent = new **Student**("Advait", course);

Student deepCopyStudent = new **Student**(originalStudent.name, new **Course**(originalStudent.course.name));

**// Changes in course name of deep copy don't affect the original**

deepCopyStudent.course.name = "Mathematics";

System.out.println(originalStudent.course.name); **// Output: Computer Science**

}

}

**Key Differences:**

* **Shallow Copy:** Copies references, changes affect both copies.
* **Deep Copy:** Creates new objects, changes are isolated.

Use a shallow copy when:

* You want to create a new object with the same values for its fields as the original object, and you don't need to protect the original object from changes made to its referenced objects.

Use a deep copy when:

* You want to create a completely independent copy of the original object, including creating new instances of any referenced objects, to prevent changes made to the referenced objects from affecting the copy.

**Tell me about clone() function, give some example.**

We need to write the number of codes for this deep copy/ Shallow copy. So to reduce this, In java, there is a method called clone(). The clone() function in Java is used to create a copy of an object. It creates a new instance that is a duplicate of the original object. The **clone() method is provided by the Cloneable interface**, and it **performs a shallow copy by default**. However, for deep copying, you need to override the clone() method to create new copies of referenced objects.

**Shallow Copy using clone():**

The default behavior of the clone() method is shallow copying. It creates a new object with copies of the fields and references to the same objects that the original object references.

package codeKatha;

class **Person** implements **Cloneable** {

String name;

Address address;

public **Person**(String name, Address address) {

this.name = name;

this.address = address;

}

@Override

public Object **clone**() throws CloneNotSupportedException {

return super.clone();

}

}

class **Address** {

String city;

public **Address**(String city) {

this.city = city;

}

}

public class **CloneShallowExample** {

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

Address address = new **Address**("Delhi");

Person originalPerson = new **Person**("Advait", address);

Person clonedPerson = (Person) originalPerson.clone();

**// Changing city in the cloned address affects the original**

clonedPerson.address.city = "Muzaffarpur";

System.out.println(originalPerson.address.city); **// Output: Muzaffarpur**

}

}

**Deep Copy using clone():**

To achieve a deep copy using the clone() method, you **need to override the clone() method** and manually clone the referenced objects.

package codeKatha;

public class **CloneDeepExample** {

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

Address address = new **Address**("Delhi");

Person originalPerson = new **Person**("Advait", address);

Person clonedPerson = (Person) originalPerson.clone();

clonedPerson.address = (Address) originalPerson.address.clone();

**// Changing city in the cloned address doesn't affect the original**

clonedPerson.address.city = "Muzaffarpur";

System.out.println(originalPerson.address.city); **// Output: Delhi**

}

}

**How has Java evolved over the years, and what are some key features introduced in recent Java versions (Java 8 onwards)?**

Java has undergone significant changes and improvements since its inception in 1995. With every new version, Java has introduced new features, enhancements, and performance improvements. Some key features introduced in recent Java versions (Java 8 onwards) include:

* **Java 8:** Introduces Lambdas, Functional Interfaces, Streams API, Optional class, and default/static methods in interfaces, along with a new Date and Time API.
* **Java 9:** Brings the Java Platform Module System (JPMS), JShell (REPL), factory methods for collections, Stream API enhancements, and private methods in interfaces.
* **Java 10:** Adds Local variable type inference (var), Application Class-Data Sharing (AppCDS), and garbage collector improvements.
* **Java 11:** Introduces the HTTP Client API, Epsilon garbage collector, new string methods, local variable syntax for lambda parameters, and Nest-based access control.
* **Java 12:** Offers Switch expressions (preview), Shenandoah garbage collector, and JVM Constants API.
* **Java 13:** Presents Text blocks (preview), Switch expressions enhancements, and ZGC garbage collector improvements.
* **Java 14:** Includes Pattern Matching for instanceof (preview), Records (preview), Text blocks enhancements, and makes Switch expressions standard.
* **Java 15:** Text blocks and Pattern Matching for instanceof become standard, and introduces Records (second preview).
* **Java 16:** Enhances Pattern Matching for instanceof, makes Records standard, introduces Foreign Function & Memory API (Incubator), and strengthens encapsulation.
* **Java 17 (LTS):** Features Sealed Classes, Pattern Matching for switch, standardizes the Foreign Function & Memory API, and continues to improve security and maintainability.