# Core Java Interview Questions

# 4 major principles of Object-Oriented Programming

**Encapsulation**

What is encapsulation? Well, in a nutshell, encapsulation is the  
hiding of data implementation by restricting access to accessors and  
mutators.

<http://www.tutorialspoint.com/java/java_encapsulation.htm>

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as as single unit. In encapsulation the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class, therefore it is also known as data hiding.

To achieve encapsulation in Java

1. Declare the variables of a class as private.
2. Provide public setter and getter methods to modify and view the variables values.

**Abstraction**

An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual  
boundaries, relative to the perspective of the viewer.

Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words user will have the information on what the object does instead of how it does it.

Abstraction in Java is achieved by using abstract keyword or interface in Java. An interface or abstract class is something which is not concrete , something which is incomplete.

Note: The class contain abstract method should be abstract class.

**Inheritance**

Now lets discuss inheritance.  Objects can relate to each other  
with either a “has a”, “uses a” or an “is a” relationship.  “Is a”  
is the inheritance way of object relationship.

<http://www.tutorialspoint.com/java/java_inheritance.htm>

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. A class that is inherited is called a superclass. The class that does the inheriting is called a subclass. Inheritance is done by using the keyword extends. The two most common reasons to use inheritance are: To promote code reuse, To use polymorphism

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class). And Java doesn’t support multiple inheritance.

Extends/implements Keyword 🡪 IS-A relationship

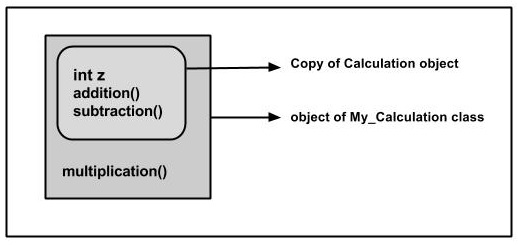
extends is the keyword used to inherit the properties of a class. Below given is the syntax of extends keyword. Generally, the implements keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class

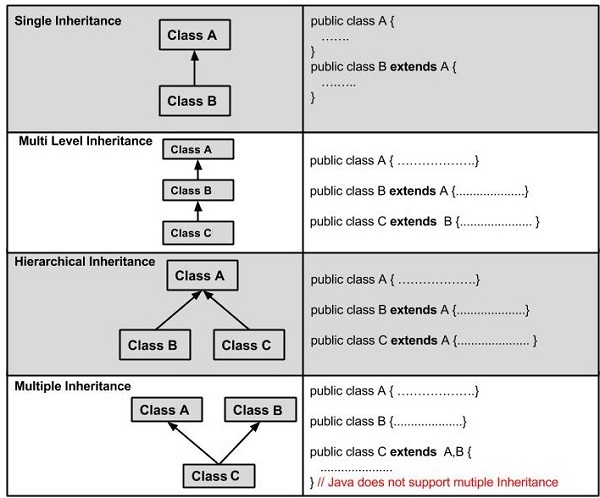
E.g

In the given program when an object to My\_Calculation class is created, a copy of the contents of the super class is made with in it. That is why, using the object of the subclass you can access the members of a super class.

class Calculation{ }

public class My\_Calculation extends Calculation{}





instanceOf keyword 🡪 IS-A relationship

E.g Let us use the instanceof operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal

interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal{

public static void main(String args[]){

Mammal m = new Mammal();

Dog d = new Dog();

System.out.println(m instanceof Animal);

System.out.println(d instanceof Mammal);

System.out.println(d instanceof Animal);

}

}

This would produce the following result: true / true / true

super keyword

The super keyword is similar to this keyword following are the scenarios where the super keyword is used.

(a)It is used to differentiate the members of superclass from the members of subclass, if they have same names.

(b)It is used to invoke the superclass constructor from subclass.

E.g If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the super class. But if you want to call a parametrized constructor of the super class, you need to use the super keyword as shown below

super(values)

class Superclass{

int age;

Superclass(int age){

this.age = age;

}

public void getAge(){

System.out.println("The value of the variable named age in super class is: " +age);

}

}

public class Subclass extends Superclass {

Subclass(int age){

super(age);

}

public static void main(String argd[]){

Subclass s = new Subclass(24);

s.getAge();

}

}

On executing the program you will get the following result −

The value of the variable named age in super class is: 24

HAS-A relationship

These relationships are mainly based on the usage. This determines whether a certain class HAS-A certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets us look into an example −

public class Vehicle{}

public class Speed{}

public class Van extends Vehicle{

private Speed sp;

}

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.

In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

**Polymorphism**

Polymorphism in Java has two types: Compile time polymorphism (static binding) and Runtime polymorphism (dynamic binding). Method overloading is an example of static polymorphism, while method overriding is an example of dynamic polymorphism.

An important example of polymorphism is how a parent class refers to a child class object. In fact, any object that satisfies more than one IS-A relationship is polymorphic in nature.

For instance, let’s consider a class Animal and let Cat be a subclass of Animal. So, any cat IS animal. Here, Cat satisfies the IS-A relationship for its own type as well as its super class Animal.

Note: It’s also legal to say every object in Java is polymorphic in nature, as each one passes an IS-A test for itself and also for Object class.

Static Polymorphism:

In Java, static polymorphism is achieved through method overloading. Method overloading means there are several methods present in a class having the same name but different types/order/number of parameters.

At compile time, Java knows which method to invoke by checking the method signatures. So, this is called compile time polymorphism or static binding. The concept will be clear from the following example:

Dynamic Polymorphism:

Suppose a sub class overrides a particular method of the super class. Let’s say, in the program we create an object of the subclass and assign it to the super class reference. Now, if we call the overridden method on the super class reference then the sub class version of the method will be called.

class Vehicle{

public void move(){

System.out.println(“Vehicles can move!!”);

}

}

class MotorBike extends Vehicle{

public void move(){

System.out.println(“MotorBike can move and accelerate too!!”);

}

}

class Test{

public static void main(String[] args){

Vehicle vh=new MotorBike();

vh.move(); // prints MotorBike can move and accelerate too!!

vh=new Vehicle();

vh.move(); // prints Vehicles can move!!

}

}

It should be noted that in the first call to move(), the reference type is Vehicle and the object being referenced is MotorBike(Parent class reference point to child class object). So, when a call to move() is made, Java waits until runtime to determine which object is actually being pointed to by the reference. In this case, the object is of the class MotorBike. So, the move() method of MotorBike class will be called. In the second call to move(), the object is of the class Vehicle. So, the move() method of Vehicle will be called.

As the method to call is determined at runtime, this is called dynamic binding or late binding.

**What are the differences between method overloading and method overriding?**

|  |  |  |
| --- | --- | --- |
|  | **Overloaded Method** | **Overridden Method** |
| **Arguments** | Must change | Must not change |
| **Return type** | Can change | Can’t change except for covariant returns |
| **Exceptions** | Can change | Can reduce or eliminate. Must not throw new or broader checked exceptions |
| **Access** | Can change | Must not make more restrictive (can be less restrictive) |
| **Invocation** | Reference type determines which overloaded version is selected. Happens at compile time. | Object type determines which method is selected. Happens at runtime. |

1. **Upcasting and Downcasting(Polymorphism)**

**public** **class** UpCastingTest{

**class** Apple{

String type = "apple";

**void** setType(){

System.*out*.println("inside apple class");

**this**.type = "apple";

}

}

**class** RedApple **extends** Apple{

String type = "redapple";

**void** setType(){

System.*out*.println("inside redapple class");

**this**.type = "redapple";

}

}

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

UpCastingTest.Apple apple = **new** UpCastingTest().**new** Apple();

apple.setType();

UpCastingTest.RedApple redApple = **new** UpCastingTest().**new** RedApple();

redApple.setType();

Apple newRedApple = (Apple)redApple;

newRedApple.setType();

}

}

**The result is:**

inside apple class

inside redapple class

inside redapple class

**That's because that's how polymorphism works in Java**: it always uses the most-derived version of the method, which *overrides* other versions(even if it's cast to a super class, the child-most method will be called.). The only way to get the base-class version is to use super.setType within the most-derived override.

Like this:

**class** RedApple **extends** Apple{

String type = "redapple";

**void** setType(){

// System.out.println("inside redapple class");

// this.type = "redapple";

**super**.setType();

}

}

The result is:

inside apple class

inside apple class

inside apple class

**However, an example of where the upcasting DOES make a difference is here:**

class MyClass {

static void doSomething(Apple apple) { System.out.println("Apple"); }

static void doSomething(RedApple apple) { System.out.println("RedApple"); }

}

...

RedApple apple = new RedApple();

MyClass.doSomething(apple);

MyClass.doSomething((Apple)apple);

Output:

RedApple

Apple

Since we upcast it to an Apple the best matched method is the one with the Apple parameter.

<http://stackoverflow.com/questions/6729039/problem-in-upcasting-in-java>

But downcasting must always be done manually:

Cat c1 = new Cat();

Animal a = c1; //automatic upcasting to Animal

Cat c2 = (Cat) a; //manual downcasting back to a Cat

Why is that so, that upcasting is automatical, but downcasting must be manual? Well, you see, upcasting can never fail. But if you have a group of different Animals and want to downcast them all to a Cat, then there's a chance, that some of these Animals are actually Dogs, and process fails, by throwing ClassCastException.

<http://forum.codecall.net/topic/50451-upcasting-downcasting/>

1. What is difference between JDK and JRE ?

<http://stackoverflow.com/questions/1906445/what-is-the-difference-between-jdk-and-jre>

JRE: Java Runtime Environment. It is basically the Java Virtual Machine where your Java programs run on. It also includes browser plugins for Applet execution.

JDK: It's the full featured Software Development Kit for Java, including JRE, and the compilers and tools (like JavaDoc, and Java Debugger) to create and compile programs.

Usually, when you only care about running Java programs on your browser or computer you will only install JRE. It's all you need. On the other hand, if you are planning to do some Java programming, you will also need JDK.

Sometimes, even though you are not planning to do any Java Development on a computer, you still need the JDK installed. For example, if you are deploying a WebApp with JSP, you are technically just running Java Programs inside the application server. Why would you need JDK then? Because application server will convert JSP into Servlets and use JDK to compile the servlets. I am sure there might be more examples.

# What is side effect ?

# <http://stackoverflow.com/questions/1073909/side-effect-whats-this>

# A side effect is anything a method does besides computing and returning a value. Any change of instance or class field values is a side effect, as is drawing something on the screen, writing to a file or a network connection.

# Strictly speaking, a "function" is defined as not having side effects - which is why Java uses the word "method" instead. A real function with no return value would be pointless.

# Obviously, a method that does not have a return value must have some sort of side effect that justifies its existence. Set methods are an example - the side effect is changing the object's internal state.

# What is wrapper class ?

# Q1. What are the Wrapper classes available for primitive types ?

# boolean - java.lang.Boolean

# byte - java.lang.Byte

# char - java.lang.Character

# double - java.lang.Double

# float - java.lang.Float

# int - java.lang.Integer

# long - java.lang.Long

# short - java.lang.Short

# void - java.lang.Void

# Q2. What are wrapper classes ?

# They are wrappers to primitive data types. They allow us to access primitives as objects.

# Java is an object-oriented language and as said everything in java is an object. But what about the primitives? They are sort of left out in the world of objects, that is, they cannot participate in the object activities, such as being returned from a method as an object, and being added to a Collection of objects, etc. . As a solution to this problem, Java allows you to include the primitives in the family of objects by using what are called - wrapper classes.

# Q3. Difference between boolean and Boolean ?

# boolean is a primitive type whereas Boolean is a class.

# Q4. Explain Autoboxing ?

# Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes

# Boxing/Unboxing: Java 5 (and hence AspectJ 1.5) supports automatic conversion of primitive types (int, float, double etc.) to their object equivalents (Integer, Float, Double,…) in assignments and method and constructor invocations. This conversion is know as autoboxing

# Q5. What are Wrapper Classes ? What are Primitive Wrapper Classes ?

# A wrapper class is any class which "wraps" or "encapsulates" the functionality of another class or component. A Wrapper Class that wraps or encapsulates the primitive data type is called Primitive Wrapper Class.

# Q6. What Design pattern Wrapper Classes implement ?

# Adapter.

# Java String Interview Question

# <http://www.journaldev.com/1321/java-string-interview-questions-and-answers>

1. What are 2 ways to create String ?

We can create String object using new operator like any normal java class or we can use double quotes to create a String object. There are several constructors available in String class to get String from char array, byte array, StringBuffer and StringBuilder.

|  |  |
| --- | --- |
| (1)  (2) | String str = new String("abc");  String str1 = "abc"; |

When we create a String using double quotes, JVM looks in the String pool to find if any other String is stored with same value. If found, it just returns the reference to that String object else it creates a new String object with given value and stores it in the String pool.  
When we use new operator, JVM creates the String object but don’t store it into the String Pool, instead it will store on heap. We can use intern() method to store the String object into String pool or return the reference if there is already a String with equal value present in the pool.

1. Where is String pool exist ?

<http://stackoverflow.com/questions/4918399/where-does-javas-string-constant-pool-live-the-heap-or-the-stack>

Runtime constant pool in method area, belong to class data.

1. What the method to check if input String is Palidrome ?

A String is said to be Palindrome if it’s value is same when reversed. For example “aba” is a Palindrome String.  
(1)StringBuffer and StringBuilder class has reverse method that we can use to check if String is palindrome or not.

private static boolean isPalindrome(String str) {

if (str == null)

return false;

StringBuilder strBuilder = new StringBuilder(str);

strBuilder.reverse();

return strBuilder.toString().equals(str);

}

(2) Sometimes interviewer asks not to use any other class to check this, in that case we can compare characters in the String from both ends to find out if it’s palindrome or not.

private static boolean isPalindromeString(String str) {

if (str == null)

return false;

int length = str.length();

System.out.println(length / 2);

for (int i = 0; i < length / 2; i++) {

if (str.charAt(i) != str.charAt(length - i - 1))

return false;

}

return true;

}

1. Write a method to remove given character in String ?

We can use replaceAll method to replace all the occurance of a String with another String. The important point to note is that it accepts String as argument, so we will use Character class to create String and use it to replace all the characters with empty String.

private static String removeChar(String str, char c) {

if (str == null)

return null;

return str.replaceAll(Character.toString(c), "");

}

1. How to compare two Strings in Java ?

Method 1: ==

Method 2: equals()

Method 3: compareTo() / compareToIngenoreCase()

Java String implements Comparable interface and it has two variants of compareTo() methods.

compareTo(String anotherString) method compares the String object with the String argument passed lexicographically. If String object precedes the argument passed, it returns negative integer and if String object follows the argument String passed, it returns positive integer. It returns zero when both the String have same value, in this case equals(String str) method will also return true.

compareToIgnoreCase(String str): This method is similar to the first one, except that it ignores the case. It uses String CASE\_INSENSITIVE\_ORDER Comparator for case insensitive comparison. If the value is zero then equalsIgnoreCase(String str) will also return true

<http://www.journaldev.com/810/java-string-compareto-examples>

E.g As “ABC” is lexicographically less than the “DEF”. Output is -3 because it compares the character values. You can also confirm this with below test program.

public class Test {

public static void main(String[] args) {

char a = 'A';

char d = 'D';

System.out.println(a-d); //prints -3

}

}

1. How to convert String to Char and vice versa ?

This is a tricky question because String is a sequence of characters, so we can’t convert it to a single character. We can use use charAt method to get the character at given index or we can use toCharArray() method to convert String to character array.

<http://www.journaldev.com/794/how-to-convert-string-to-char-to-string-in-java>

1. How to convert String to byte array and vice versa?

We can use String getBytes() method to convert String to byte array and we can use String constructor new String(byte[] arr) to convert byte array to String.

<http://www.journaldev.com/770/convert-string-to-byte-array-and-byte-array-to-string-in-java>

public static void main(String[] args) {

String str = "www.journaldev.com";

//convert String to byte array

byte[] byteArr = str.getBytes();

System.out.println("String to byte array : "+Arrays.toString(byteArr));

//convert byte array to String

String str1 = new String(byteArr);

System.out.println("byte array to String : "+str1);

//let's see if str and str1 are equals or not

System.out.println("str == str1? " + (str == str1)); 🡪 Return false

System.out.println("str.equals(str1)? " + (str.equals(str1))); 🡪 Return true

}

<http://stackoverflow.com/questions/767372/java-string-equals-versus>

The string.equals(Object other) function checks the actual contents of the string, the == operator checks whether the references to the objects are equal.

1. Write a program to print all permutations in String ?

<http://www.journaldev.com/526/java-program-to-find-all-permutations-of-a-string>

Algorithm

To get all the permutations, we will first take out the first char from String and permute the remaining chars.

If String = “ABC”

First char = A and remaining chars permutations are BC and CB.

Now we can insert first char in the available positions in the permutations.

BC -> ABC, BAC, BCA

CB -> ACB, CAB, CBA

So we can write a recursive function call to return the permutations and then another function call to insert the first characters to get the complete list of permutations.

public class StringHelper {

public static Set<String> permutationFinder(String str) {

Set<String> perm = new HashSet<String>(); 🡪 Note that I have used SET to remove duplicates, so that it works for those strings also having same chars.

//Handling error scenarios

if (str == null) {

return null;

} else if (str.length() == 0) {

perm.add(""); 🡪 This is for avoid recursive to the end return NULL and break the process.

return perm;

}

char initial = str.charAt(0); // first character

String rem = str.substring(1); // Full string without first character

Set<String> words = permutationFinder(rem); 🡪 Recursively substring and find until last character.

for (String strNew : words) {

for (int i = 0;i<=strNew.length();i++){

perm.add(charInsert(strNew, initial, i));

}

}

return perm;

}

public static String charInsert(String str, char c, int j) {

String begin = str.substring(0, j);

String end = str.substring(j);

return begin + c + end; 🡪 Can be improve by introducing StringBuffer/StringBuilder, since the “+” operator creates a new string rather than appending to the existing string.

}

public static void main(String[] args) {

String s = "AAC";

String s1 = "ABC";

System.out.println("\nPermutations for " + s + " are: \n" + permutationFinder(s));

System.out.println("\nPermutations for " + s1 + " are: \n" + permutationFinder(s1));

}

}

1. How to find out longest palindrome in String ?

<http://www.journaldev.com/530/java-program-to-find-out-longest-palindrome-in-a-string>

<http://articles.leetcode.com/longest-palindromic-substring-part-i/>

1. Difference between String, StringBuffer and StringBuilder ?

Since String is immutable in java, whenever we do String manipulation like concat, substring etc, it generates a new String and discard the older String for garbage collection. These are heavy operations and generate a lot of garbage in heap. String manipulations are resource consuming, so java provides two utility classes for String manipulations – StringBuffer and StringBuilder. String concat + operator internally uses StringBuffer or StringBuilder class.

StringBuffer and StringBuilder are mutable classes. StringBuffer operations are thread-safe and synchronized where StringBuilder operations are not thread-safe. So when multiple threads are working on same String, we should use StringBuffer but in single threaded environment we should use StringBuilder.

StringBuilder performance is fast than StringBuffer because of no overhead of synchronization.

1. Why String is immutable or final in Java ?

<http://www.journaldev.com/802/why-string-is-immutable-or-final-in-java>

1. 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(Runtime constant 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. Since String is immutable, it is safe for multithreading 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.
3. Strings are used in 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.
4. 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. (Why String is popular HashMap key in Java?)
5. Why Char array is more preferred than String on storing password ?

String is immutable in java and stored in String pool. Once it’s created it stays in the pool until unless garbage collected, so even though we are done with password it’s available in memory for longer duration and there is no way to avoid it. It’s a security risk because anyone having access to memory dump can find the password as clear text.

If we use char array to store password, we can set it to blank once we are done with it. So we can control for how long it’s available in memory that avoids the security threat with String.

1. How is intern() method in String works ?

String s1 = "abc";

String s2 = new String("abc");

s2.intern();

System.out.println(s1 ==s2);

It’s a tricky question and output will be false. We know that intern() method will return the String object reference from the string pool, but since we didn’t assigned it back to s2, there is no change in s2 and hence both s1 and s2 are having different reference. If we change the code in line 3 to s2 = s2.intern(); then output will be true.

# How do you avoid NullPointerException, while comparing two Strings in Java?

<http://javarevisited.blogspot.com/2013/08/10-equals-and-hashcode-interview.html>

Since when compared to null, equals return false and doesn't throw NullPointerException, you can use this property to avoid NPE while using comparing String. Suppose you have a known String "abc" and you are comparing with an unknown String variable str, then you should call equals as "abc".equals(str), this will not throw Exception in thread Main: java.lang.NullPointerException, even if str is null. On the other hand, if you call str.equals("abc"), it will throw NPE.

# What’s the difference between an interface and an abstract class in Java?

# A class must be declared *abstract* when it has one or more abstract *methods*. A method is declared abstract when it has a method heading, but no body – which means that an abstract method has no implementation code inside curly braces like normal methods do.

# You should also know that any non-abstract class is called a concrete class. Knowing your terminology definitely pays off in an interview.

# An interface differs from an abstract class because an interface is not a class. An interface is essentially a type that can be satisfied by any class that implements the interface.

# Any class that implements an interface must satisfy 2 conditions: It must have the phrase "implements Interface Name" at the beginning of the class definition. It must implement all of the method headings listed in the interface definition.

# Abstract classes are meant to be inherited from, and when one class inherits from another it means that there is a strong relationship between the 2 classes. Because a Townhouse is a type of House, that relationship is very strong, and would be more appropriately defined through inheritance instead of interfaces. With an interface on the other hand, the relationship between the interface itself and the class implementing the interface is not necessarily strong. For example, if we have a class called "House", that class could also implement an interface called "Air-conditioning".

# Java does not allow multiple inheritances – see the discussion on Java Multiple Inheritance if you need a refresher on this. In Java, a class can only derive from one class, whether it’s abstract or not. However, a class can implement multiple interfaces – which could be considered as an alternative to for multiple inheritances. So, one major difference is that a Java class can inherit from only one abstract class, but can implement multiple interfaces.

# An abstract class may provide some methods with definitions – so an abstract class can have non-abstract methods with actual implementation details. An abstract class can also have constructors and instance variables as well. An interface, however, cannot provide any method definitions – it can only provide method headings. Any class that implements the interface is responsible for providing the method definition/implementation.

# When to use abstract class and interface in Java

# An abstract class is good if you think you will plan on using inheritance since it provides a common base class implementation to derived classes.

# An abstract class is also good if you want to be able to declare non-public members. In an interface, all methods must be public.

# If you think you will need to add methods in the future, then an abstract class is a better choice. Because if you add new method headings to an interface, then all of the classes that already implement that interface will have to be changed to implement the new methods. That can be quite a hassle.

# Interfaces are a good choice when you think that the API will not change for a while.

# Interfaces are also good when you want to have something similar to multiple inheritances, since you can implement multiple interfaces.

# <http://www.programmerinterview.com/index.php/java-questions/interface-vs-abstract-class/>

In this post we will discuss **difference between Abstract Class and Interface in Java with examples.**

**(记忆顺序1/2,4,3/6,7,8)**

|  |  |  |
| --- | --- | --- |
|  | **abstract Classes** | **Interfaces** |
| 1 | abstract class can extend only one class or one abstract class at a time | **Interface is not a class** and can extend any number of interfaces at a time |
| 2 | abstract  class  can extend from a class or from an abstract class | interface can extend only from an interface |
| 3 | abstract  class  can  have  both  abstract and concrete methods | interface can  have only abstract methods |
| 4 | A class can extend only one abstract class(**Because a Townhouse is a type of House, that relationship is very strong**) | A class can implement any number of interfaces(**the relationship between the interface itself and the class implementing the interface is not necessarily strong. Java does not allow multiple inheritances, In Java, a class can only derive from one class, whether it’s abstract or not. However, a class can implement multiple interfaces – which could be considered as an alternative to for multiple inheritances.**) |
| 5 | In abstract class keyword ‘abstract’ is mandatory to declare a method as an abstract | In an interface keyword ‘abstract’ is optional to declare a method as an abstract |
| 6 | abstract  class can have  protected , public and public abstract methods | Interface can have only public abstract methods i.e. by default |
| 7 | abstract class can have  static, final  or static final  variable with any access specifier | interface  can  have only static final (constant) variable i.e. by default |

8. If you think you will need to add methods in the future, then an abstract class is a better choice. Because if you add new method headings to an interface, then all of the classes that already implement that interface will have to be changed to implement the new methods.

Each of the above difference between **Abstract class vs Interface** is explained with an example below – <http://beginnersbook.com/2013/05/abstract-class-vs-interface-in-java/>

3. What does the ‘static’ keyword mean?

## Static Class

A Class can be made **static** only if it is a nested Class. The nested static class can be accessed without having an object of outer class. [Static inner class](http://beginnersbook.com/2013/05/inner-class/) cannot access instance data of outer class.

**Example 1:**

class Example1{

//Static class

static class X{

static String str="Inside Class X";

}

public static void main(String args[])

{

X.str="Inside Class Example1";

System.out.println("String stored in str is- "+ X.str);

}

}

Output:

String stored in str is- Inside Class Example1

**Example 2: Compile time Error!!**

class Example2{

int num;

//Static class

static class X{

static String str="Inside Class X";

num=99;

}

public static void main(String args[])

{

Example2.X obj = new Example2.X();

System.out.println("Value of num="+obj.str);

}

}

Output: Compile time error.

## Static Methods

Static Methods can access class variables without using object of the class. Static methods can be accessed directly in static and non-static methods.

**Static methods are always invoked without reference to a particular instance of a class.**

**Note:The use of a static method suffers from the following restrictions:**

**• A static method can only call other static methods.**

**• A static method must only access static data.**

It can access non-static methods and non-static variables by using objects.

**• A static method cannot reference to the current object using keywords super or this.**

**Example 1: public static void main itself is a static method(The main() method in C++, C# and Java are static because they can then be invoked by the runtime engine without having to instantiate an instance of the parent class**.**)**

class Example5{

static int i;

static String s;

public static void main(String args[]) //Its a Static Method

{

Example5 obj=new Example5();

//Non Static variables accessed using object obj

System.out.println("i:"+obj.i);

System.out.println("s:"+obj.s);

}

}

**Output:**

i:0

s:null

**Example 2: Static method display()**

class Example6{

static int i;

static String s;

//Static method

static void display()

{

//Its a Static method

Example6 obj1=new Example6();

System.out.println("i:"+obj1.i);

System.out.println("i:"+obj1.i);

}

void funcn()

{

//Static method called in non-static method

display();

}

public static void main(String args[]) //Its a Static Method

{

//Static method called in another static method

display();

}

}

Output:

i:0

i:0

## Static Variables

Variables that have only one copy per class are known as static variables. They are **not attached to a particular instance** **of a class but rather belong to a class as a whole.**

* Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods.

A static variable is associated with the class as a whole rather than with specific instances of a class. **Non-static variables** take on unique values with each object instance.

* Data stored in static variables is common for all the instances of that Class.
* Memory allocation for such variables only happens once when the class is loaded in the memory.
* These variables can be accessed in any other class using class name.

**Example 1: Static variables can be accessed without reference in Static method**

class Example7{

static int var1;

static String var2;

//Its a Static Method

public static void main(String args[])

{

System.out.println("Var1 is:"+Var1);

System.out.println("Var2 is:"+Var2);

}

}

**Output:**

Var1 is:0

Var2 is:null

As you can see in the above example that both the variables are accessed in void main method without any object(reference).

**Example 2: Static variables are common for all instances**

package beginnersbook.com;

class Example8{

static int Var1=77; //Static integer variable

String Var2;//non-static string variable

public static void main(String args[])

{

Example8 ob1 = new Example8();

Example8 ob2 = new Example8();

ob1.Var1=88;

ob1.Var2="I'm Object1";

ob2.Var2="I'm Object2";

System.out.println("ob1 integer:"+ob1.Var1);

System.out.println("ob1 String:"+ob1.Var2);

System.out.println("ob2 integer:"+ob2.Var1);

System.out.println("ob2 STring:"+ob2.Var2);

}

}

Output:

ob1 integer:88

ob1 String:I'm Object1

ob2 integer:88

ob2 String:I'm Object2

In above example String variable is non-static and integer variable is Static. So you can see that String variable value is different for both objects but integer variable value is common for both the instances as all the objects share the same copy of a static variable.

<http://beginnersbook.com/2013/04/java-static-class-block-methods-variables/>

1. **Autoboxing:** The Java compiler brings about an automatic transformation of primitive type (int, float, double etc.) into their object equivalents or wrapper type (Integer, Float, Double,etc) for the ease of compilation.、

# What is Object ?

# Definition: An object is a software bundle of variables and related methods. In the object-oriented programming paradigm, "object" refers to a particular instance of a class where the object can be a combination of variables, functions, and data structures.

1. **What is difference between iterator access and index access?**

**Index based access allow access of the element directly on the basis of index. The cursor of the data structure can directly go to the 'n' location and get the element. It does not traverse through n-1 elements.**

**In Iterator based access, the cursor has to traverse through each element to get the desired element. So to reach the 'n' th element it need to traverse through n-1 elements.**

**Insertion, updation or deletion will be faster for iterator based access if the operations are performed on elements present in between the data structure.**

**Insertion, updation or deletion will be faster for index based access if the operations are performed on elements present at last of the data structure.**

**Traversal or search in index based data structure is faster.**

**ArrayList is index access and LinkedList is iterator access.**

[**http://java-questions.com/collections-interview-questions\_1.html**](http://java-questions.com/collections-interview-questions_1.html)

**Q9) When to use ArrayList or LinkedList ?**

Ans)

1. Adding new elements is pretty fast for either type of list. Inserting element to n location in arraylist and to first location in linkedlist takes O(1).
2. For the ArrayList, doing random lookup using "get" is fast, but for LinkedList O(n), it's slow. It's slow because there's no efficient way to index into the middle of a linked list. Linkedlist lookup always start from 1st location.
3. When removing elements, using ArrayList is slow. This is because all remaining elements in the underlying array of Object instances must be shifted down for each remove operation. But LinkedList is fast, because deletion can be done simply by changing a couple of links.

So an ArrayList works best for cases where you're doing random access on the list and a LinkedList works better if you're doing a lot of editing in the middle of the list.

[**http://java-questions.com/collections-interview-questions.html**](http://java-questions.com/collections-interview-questions.html)

1. **How to sort list in reverse order?**

[**http://stackoverflow.com/questions/3962766/java-reverse-list**](http://stackoverflow.com/questions/3962766/java-reverse-list)

To sort the elements of the List in the reverse natural order of the strings, **get a reverse Comparator from the Collections class with reverseOrder().** Then, pass the reverse Comparator to the sort() method.

List list = new ArrayList();

Comparator comp = Collections.reverseOrder();

Collections.sort(list, comp)

If we use for loop to reverse the string should be look like

<http://stackoverflow.com/questions/2612976/printing-reverse-of-any-string-without-using-any-predefined-function>

public String reverse(String s) {

StringBuffer sb = new StringBuffer();

for(int i = string.length()-1; i = 0; i--) {

sb.append(s.charAt(i));

}

return sb.toString();

}

1. Difference Between ClassNotFoundException Vs NoClassDefFoundError In Java <http://javaconceptoftheday.com/classnotfoundexception-vs-noclassdeffounderror-in-java/>

In Java, both ClassNotFoundException and NoClassDefFoundError occur when a particular class is not found at run time. But, they occur at different scenarios.

ClassNotFoundException is an exception which occurs when you try to load a class at run time using Class.forName() or loadClass() methods and mentioned classes are not found in the classpath. On the other hand, NoClassDefFoundError is an error which occurs when a particular class is present at compile time but it was missing at run time

ClassNotFoundException In Java :

ClassNotFoundException is a run time exception which is thrown when an application tries to load a class at run time using Class.forName() or loadClass() or findSystemClass() methods and the class with specified name are not found in the classpath. For example, you may have come across this exception when you try to connect to MySQL or Oracle databases and you have not updated the classpath with required JAR files. In most of time, this exception occurs when you try to run an application without updating the classpath with required JAR files.

For example, below program will throw ClassNotFoundException if the mentioned class “oracle.jdbc.driver.OracleDriver” is not found in the classpath.

public class MainClass

{

public static void main(String[] args)

{

try

{

Class.forName("oracle.jdbc.driver.OracleDriver");

}

catch (ClassNotFoundException e)

{

e.printStackTrace();

}

}

}

If you run the above program without updating the classpath with required JAR files, you will get the exception like below,

java.lang.ClassNotFoundException: oracle.jdbc.driver.OracleDriver

at java.net.URLClassLoader.findClass(Unknown Source)

NoClassDefFoundError is an error which is thrown when Java Runtime System tries to load the definition of a class and class definition is no longer available. The required class definition was present at compile time but it was missing at run time. For example, compile the below program.

class A{ }

public class B

{

public static void main(String[] args)

{

A a = new A();

}

}

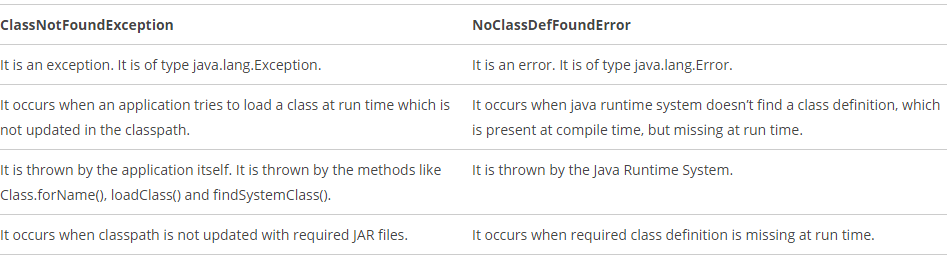
When you compile the above program, two .class files will be generated. One is A.class and another one is B.class. If you remove the A.class file and run the B.class file, Java Runtime System will throw NoClassDefFoundError like below,

Exception in thread "main" java.lang.NoClassDefFoundError: A

at MainClass.main(MainClass.java:10)

Caused by: java.lang.ClassNotFoundException: A

Below is the quick recap of above findings.



# Exception Handling

# Q1) What is an Exception?

# Ans) The exception is said to be thrown whenever an exceptional event occurs in java which signals that something is not correct with the code written and may give unexpected result. An exceptional event is a occurrence of condition which alters the normal program flow. Exceptional handler is the code that does something about the exception

* Checked Exception - exception that occur at compile time.
* UnChecked Exception - exception that occur at runtime.

<http://javarevisited.blogspot.com/2013/06/10-java-exception-and-error-interview-questions-answers-programming.html>

Q7) What is difference between Error and Exception?

Ans) An error is an irrecoverable condition occurring at runtime. Such as OutOfMemory error. These JVM errors and you can not repair them at runtime. Though error can be caught in catch block but the execution of application will come to a halt and is not recoverable.

While exceptions are conditions that occur because of bad input etc. e.g. FileNotFoundException will be thrown if the specified file does not exist. Or a NullPointerException will take place if you try using a null reference. In most of the cases it is possible to recover from an exception (probably by giving user a feedback for entering proper values etc.)

<http://javarevisited.blogspot.com/2013/06/10-java-exception-and-error-interview-questions-answers-programming.html>

- What is Difference between Error and Exception?

Both Error and Exception extends Throwable.

Exception - They can be checked or unchecked, cause due to programmer or user fault and must handle at developer level.

Error - They are always unchecked, usually occur during system error. Like OutofMemoryError, StackOverflowError.

<http://www.tkhts.com/interview-questions/exception-handling-basics.jsp>

5 - What is checked and unchecked exception?

<http://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/>

(1)What are checked exceptions? – SQLException / IOException(FileNotFoundException) / ClassNotFoundException

Checked exceptions are checked at compile-time. Handle checked exception in 2 ways: using try-catch block or it should declare the exception using throws keyword.

import java.io.\*;

class Example {

public static void main(String args[])

{

FileInputStream fis = null;

/\*This constructor FileInputStream(File filename)

\* throws FileNotFoundException which is a checked

\* exception\*/

fis = new FileInputStream("B:/myfile.txt");

int k;

/\*Method read() of FileInputStream class also throws

\* a checked exception: IOException\*/

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

/\*The method close() closes the file input stream

\* It throws IOException\*/

fis.close();

}

}

Method 1: try catch block 🡪 The catch block should catch explicit kind of exception.

import java.io.\*;

class Example {

public static void main(String args[])

{

FileInputStream fis = null;

try{

fis = new FileInputStream("B:/myfile.txt");

}catch(FileNotFoundException fnfe){

System.out.println("The specified file is not " +

"present at the given path");

}

int k;

try{

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

fis.close();

}catch(IOException ioe){

System.out.println("I/O error occurred: "+ioe);

}

}

Method 2: Declare the exception using throws keyword.

import java.io.\*;

class Example {

public static void main(String args[]) throws IOException

{

FileInputStream fis = null;

fis = new FileInputStream("B:/myfile.txt");

int k;

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

fis.close();

}

}

Th reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare that too like this public static void main(String args[]) throws IOException, FileNotFoundException.

(2)What are unchecked exceptions ? – NullPointerException / ArrayIndexOutOfBoundsException / ArithmeticException / IllegalArgumentException

Unchecked exception also known as runtime exception, the compile of code will be success, but when it run, will fail. If you want to catch a Runtime Exception, it also can be handle by try catch block, no difference than normal exceptions. NOTE: In catch block, the unchecked exception top level parent is RuntimeException. <http://stackoverflow.com/questions/2028719/handling-runtimeexceptions-in-java>

try {

someMethodThatThrowsRuntimeException();

} catch (RuntimeException ex) {

// do something with the runtime exception

}

ArithmeticException/ ArrayIndexOutOfBoundsException examples

class Example {

public static void main(String args[])

{

int num1=10;

int num2=0;

/\*Since I'm dividing an integer with 0

\* it should throw ArithmeticException\*/

int res=num1/num2;

System.out.println(res);

}

}

class Example {

public static void main(String args[])

{

int arr[] ={1,2,3,4,5};

/\*My array has only 5 elements but

\* I'm trying to display the value of

\* 8th element. It should throw

\* ArrayIndexOutOfBoundsException\*/

System.out.println(arr[7]);

}

}

6 - What is throw keyword?

Throw keyword is used to throw the exception manually. The throw keyword is used to indicate that an exception occurred. Using “throw keyword” we can throw checked, unchecked and user -defined exceptions

Steps: 1.Define throw with explicit exception instance always create by new

2. Put that method in try block 3. Catch with that explicit exception in catch block

<http://beginnersbook.com/2013/04/throw-in-java/>

Points to Note: Method call should be in try block as it is throwing an exception.

class MyOwnException extends Exception {

public MyOwnException(String msg){

super(msg);

}

}

class EmployeeTest {

static void employeeAge(int age) throws MyOwnException{

if(age < 0)

throw new MyOwnException("Age can't be less than zero");

else

System.out.println("Input is valid!!");

}

public static void main(String[] args) {

try {

employeeAge(-2); 🡪 Method contain exception called in try block

}

catch (MyOwnException e) {

e.printStackTrace();

}

}

} Output: beginnersbook.com.MyOwnException: Age can't be less than zero

7 - What is use of throws keyword?

Any exception that comes along throws keyword is a checked exception. When the method is used, if the exception is not handled, the program does not compile. Throws keyword throws exception to next class until main class handles it by surrounding it by try and catch block.

When using “throws” keyword, should prepare to handle the delegated exceptions in outer method. As below example, the “throws” exception create by sleep() method finally handled by try catch block in main() method. <http://stackoverflow.com/questions/4889711/what-happens-if-a-method-throws-an-exception-that-was-not-specified-in-the-metho>

**Declaration:**

public static void sleep(long millis) throws InterruptedException

So if you wish to call it for instance in your main method you must either catch it:

public static void main(String args[]) {

try {

Thread.sleep(1000);

} catch(InterruptedException ie) {

System.out.println("Opps!");

}

}

8 – Difference between throw and throws ?

Note: throw can also be used to throw user defined exception.

|  |  |  |
| --- | --- | --- |
| **No.** | **Throw** | **throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException,SQLException. |

9 - What is StackOverflowError?

StackOverFlowError is an Error object thrown when it encounters that your program has ran out of the memory.

10 - Is it valid to have a try block without catch or finally?

No, try block should have one or both blocks(catch or finally) immediately followed to handle the occurred exception.

11 - Can you catch more than one exceptions in a single catch block?

Yes, you can catch multiple exception in a single catch block by using Exception as a argument by following multiple if conditions.

<http://www.tkhts.com/interview-questions/exception-handling-basics.jsp>

**12) What is difference in final, finalize and finally keyword in Java?**

Final and finally are keyword, while finalize is method. final keyword is very useful for creating ad Immutable class in Java By making a class final, we prevent it from being extended, similarly by making a method final, we prevent it from being overridden,. On the other hand, finalize() method is called by garbage collector, before that object is collected, but this is not guaranteed by Java specification. finally keyword is the only one which is related to error and exception handling and you should always have finally block in production code for closing connection and resources.

13) What is wrong with following code :

public static void start() throws IOException, RuntimeException{

throw new RuntimeException("Not able to Start");

}

public static void main(String args[]) {

try {

start();

} catch (Exception ex) {

ex.printStackTrace();

} catch (RuntimeException re) {

re.printStackTrace();

}

}

This code will throw compiler error on line where RuntimeException variable “re” is written on catch block. since Exception is super class of RuntimeException, all RuntimeException thrown by start() method will be captured by first catch block and code will never reach second catch block and that's the reason compiler will flag error as “exception java.lang.RuntimeException has already been caught".

try

{

int x = 0;

int y = 5 / x;

}

catch (Exception e)

{

System.out.println("Exception");

}

catch (ArithmeticException ae)

{

System.out.println(" Arithmetic Exception");

}

System.out.println("finished");

Compilation fails because ArithmeticException has already been caught.ArithmeticException is a subclass of java.lang.Exception, by time theArithmeticException has been specified it has already been caught by the Exceptionclass.

If ArithmeticException appears before Exception, then the file will compile. When catching exceptions the more specific exceptions must be listed before the more general (the subclasses must be caught before the superclasses).

15) What is wrong with following Java Exception code:

public static void start(){

System.out.println("Java Exception interivew question Answers for Programmers");

}

public static void main(String args[]) {

try{

start();

}catch(IOException ioe){

ioe.printStackTrace();

}

}

**In this Java Exception example code, compiler will complain on line where we are handling IOException, since IOException is a checked Exception and start() method doesn't throw IOException, so compiler will flag error as "exception java.io.IOException is never thrown in body of corresponding try statement", but if you change IOException to Exception compiler error will disappear because Exception can be used to catch all RuntimeException which doesn't require declaration in throws clause.**

<http://javarevisited.blogspot.com/2013/06/10-java-exception-and-error-interview-questions-answers-programming.html>

Q14)What is StackOverflowError?

Ans) The StackOverFlowError is an Error Object thorwn by the Runtime System when it Encounters that your application/code has ran out of the memory. It may occur in case of recursive methods or a large amount of data is fetched from the server and stored in some object. This error is generated by JVM.

e.g. void swap(){

swap();

}

<http://java-questions.com/Exceptions-interview-questions.html>

**Q15) How assertion works?**

Assertion enables developers to test assumptions in their programs as a way to defect and fix bugs.

**Assertion example 1**

**The following simple program illustrates the short version of assert statement:**

**public class AssertionExample {**

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

**// get a number in the first argument**

**int number = Integer.parseInt(args[0]);**

**assert number <= 10; // stops if number > 10**

**System.out.println("Pass");**

**}**

**}**

**When running the program above with this command:**

**java -ea AssertionExample 15**

**A java.lang.AssertionError error will be thrown:**

**Exception in thread "main" java.lang.AssertionError at AssertionExample.main(AssertionExample.java: line 6)**

**But the program will continue and print out “Pass” if we pass a number less than 10, in this command:**

**java -ea AssertionExample 8**

**Assertion example 2**

**public class AssertionExample2 {**

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

**int argCount = args.length;**

**assert argCount == 5 : "The number of arguments must be 5";**

**System.out.println("OK");**

**}**

**}**

**When running the program above with this command:**

**java -ea AssertionExample2 100 101 102 103**

**it will throw this error:**

**Exception in thread "main" java.lang.AssertionError: The number of arguments must be 5**

**at AssertionExample2.main(AssertionExample2.java: line 6)**

**Generally, assertion is enabled during development time to defect and fix bugs, and is disabled at deployment or production to increase performance.**

[**http://www.codejava.net/java-core/the-java-language/java-keyword-assert**](http://www.codejava.net/java-core/the-java-language/java-keyword-assert)

**Q16)**What causes compilation to fail?

public class Test

{

public void foo()

{

assert false; /\* Line 5 \*/

assert false; /\* Line 6 \*/

}

public void bar()

{

while(true)

{

assert false; /\* Line 12 \*/

// break;

}

assert false; /\* Line 14 \*/

}

}

Option D is correct. Compilation fails because of an unreachable statement at line 14. It is a compile-time error if a statement cannot be executed because it is unreachable. Examine the following:

A while statement can complete normally if and only if at least one of the following is true:

- The while statement is reachable and the condition expression is not a constant expression with value true.

-There is a reachable break statement that exits the while statement.

The while statement at line 11 is infinite and there is no break statement therefore line 14 is unreachable.

<http://www.indiabix.com/online-test/java-programming-test/61>

Q17) What will be the output of the program?

**public** **class** Test

{

**public** **static** **int** *y*;

**public** **static** **void** foo(**int** x)

{

System.*out*.print("foo ");

*y* = x;

}

**public** **static** **int** bar(**int** z)

{

System.*out*.print("bar ");

**return** *y* = z;

}

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

{

**int** t = 0;

**assert** t > 0 : *bar*(7);

**assert** t > 1 : *foo*(8); /\* Line 18 \*/

System.*out*.println("done ");

}

}

 Syntax of an assert statement is as follow (short version):

**assert***expression1*;

or (full version):

**assert** *expression1* : *expression2*;

Where:

* *expression1* must be a**boolean**expression.
* *expression2* must return a value (must not return void).

The foo() method returns void. It is a perfectly acceptable method, but because it returns void it cannot be used in an assert statement, so line 18 will not compile. **Compilation fails**

<http://www.indiabix.com/online-test/java-programming-test/61>

Q18)

**(A)It is appropriate to handle assertion failures using a catch clause?**

Incorrect because assertions throw errors and not exceptions, and assertion errors do cause program termination and should not be handled.

**(B)A try statement must have at least one corresponding catch block?**

A try statement can exist without catch, but it must have a finally statement.

**(C)Multiple catch statements can catch the same class of exception more than once?**

A try statement executes a block. If a value is thrown and the try statement has one or more catch clauses that can catch it, then control will be transferred to the first such catch clause. If that catch block completes normally, then the try statement completes normally.

(D) An Error that might be thrown in a method must be declared as thrown by that method, or be handled within that method.

Exceptions of type Error and RuntimeException do not have to be caught, only checked exceptions (java.lang.Exception) have to be caught. However, speaking ofExceptions, Exceptions do not have to be handled in the same method as the throw statement. They can be passed to another method.

Q19) What will be the output of the program? (Catch also catch the runtimeexception)

public class RTExcept

{

public static void throwit ()

{

System.out.print("throwit ");

throw new RuntimeException();

}

public static void main(String [] args)

{

try

{

System.out.print("hello ");

throwit();

}

catch (Exception re )

{

System.out.print("caught ");

}

finally

{

System.out.print("finally ");

}

System.out.println("after ");

}

}

The main() method **properly catches and handles the RuntimeException** in the catch block, finally runs (as it always does), and then the code returns to normal. Result is

hello throwit caught finally after

Q20) What will be the output of the program?

And the command line invocation: > java Test red green blue

public class Test

{

public static void main (String[] args)

{

String foo = args[1];

String bar = args[2];

String baz = args[3];

System.out.println("baz = " + baz); /\* Line 8 \*/

}

}

When running the program you entered 3 arguments "red", "green" and "blue". When dealing with arrays in java you must remember ALL ARRAYS IN JAVA ARE ZERO BASED therefore args[0] becomes "red", args[1] becomes "green" and args[2] becomes "blue".

When the program entcounters line 8 above at runtime it looks for args[3] which has never been created therefore you get an

ArrayIndexOutOfBoundsException at runtime.

[**http://www.indiabix.com/online-test/java-programming-test/65**](http://www.indiabix.com/online-test/java-programming-test/65)

# Final Variable, Method, Class

# Final is different than finally keyword which is used on Exception handling in Java.

# Final should not be confused with finalize() method which is declared in object class and called before an object is garbage collected by JVM.

# (1)What is final variable in Java?

# A final [variable](http://en.wikipedia.org/wiki/Variable_(programming)) can only be initialized once, either via an initializer or an assignment statement. It does not need to be initialized at the point of declaration: this is called a "blank final" variable. A blank final instance variable of a class must be definitely assigned in every constructor of the class in which it is declared; similarly, a blank final static variable must be definitely assigned in a static initializer of the class in which it is declared; otherwise, a compile-time error occurs in both cases.[[5]](http://en.wikipedia.org/wiki/Final_(Java)#cite_note-5) (Note: If the variable is a reference, this means that the variable cannot be re-bound to reference another object. But the object that it references is still [mutable](http://en.wikipedia.org/wiki/Mutable_object), if it was originally mutable.)

# <http://en.wikipedia.org/wiki/Final_(Java)>

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

# public static final String LOAN = "loan";

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

# (2)What is final method in Java

# Final methods are bonded during compile time also called static binding.

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

# class PersonalLoan{

# public final String getName(){

# return "personal loan";

# }

# }

# class CheapPersonalLoan extends PersonalLoan{

# @Override

# public final String getName(){

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

# }

# }

# (3)What is final Class in Java

# Java class with final modifier is called final class in Java. Final class is complete in nature and can not be sub-classed or inherited. Several classes in Java are final e.g. String, Integer and o上ther wrapper classes. Here is an example of final class in java

# final class PersonalLoan{

# }

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

# }

# (4)Final and Immutable Class in Java

# Final keyword helps to write immutable class. Immutable classes are the one which can not be modified once it gets created and String is primary example of immutable and final class which I have discussed in detail on Why String is final or immutable in Java. Immutable classes offer several benefits one of them is that they are effectively read-only and can be safely shared in between multiple threads without any synchronization overhead.

# <http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html>

# Finally Block

# The finally block, if used, is placed after a try block and the catch blocks that follow it. The finally block contains code that will be run whether or not an exception is thrown in a try block. (remember that the code in the finally block will still be executed even if there is a return statement somewhere in the try block.)

# <http://www.programmerinterview.com/index.php/java-questions/finally-block/>

# Difference between Runnable Vs. Thread, which is better?

# 1) Java doesn't support multiple inheritance, which means you can only extend one class in Java so once you extended Thread class you lost your chance and can not extend or inherit another class in Java.

# 2) In Object oriented programming extending a class generally means adding new functionality, modifying or improving behaviors. If we are not making any modification on Thread than use Runnable interface instead.

# 3) Separating task as Runnable means we can reuse the task and also has liberty to execute it from different means. since you can not restart a Thread once it completes. again Runnable vs Thread for task, Runnable is winner.( Using Runnable Interface, you can run the class several times whereas Thread have the start() method that can be called only once.)

# 4) By extending Thread, each of your threads has a unique object associated with it, whereas implementing Runnable, many threads can share the same object instance.

# <http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html>

# When to use Runnable vs Thread in Java?

# This is follow-up of previous multi-threading interview question. As we know we can implement thread either by extending Thread class or implementing Runnable interface, question arise, which one is better and when to use one? This question will be easy to answer, if you know that Java programming language doesn't support multiple inheritance of class, but it allows you to implement multiple interface. Which means, its better to implement Runnable

# <http://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-answers.html>

# What will be the output of the program?

# class MyThread extends Thread {

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

# MyThread t = new MyThread();

# t.start();

# System.out.print("one. ");

# t.start();

# System.out.print("two. ");

# }

# public void run() {

# System.out.print("Thread ");

# }

# }

# A. Compilation fails

# B. An exception occurs at runtime.

# C. It prints "Thread one. Thread two."

# D. The output cannot be determined.

# When the start() method is attempted a second time on a single(same) Thread object, the method will throw an IllegalThreadStateException (you will not need to know this exception name for the exam). Even if the thread has finished running, it is still illegal to call start() again.

# <http://www.indiabix.com/online-test/java-programming-test/61>

# Garbage Collection, how to invoke garbage?

1) **objects are created on heap in Java**

2) Garbage collection is a mechanism provided by Java Virtual Machine to **reclaim heap space** from objects which are**eligible for Garbage collection**.  
3) **Garbage collection**relieves java programmer from**memory management** which is essential part of C++ programming and gives more time to focus on business logic.  
4) **Garbage Collection in Java** is carried by a daemon thread called ***Garbage Collector***.  
5) Before removing an object from memory **Garbage collection thread invokes finalize () method**of that object and gives an opportunity to perform any sort of cleanup required.  
6) You as Java programmer **can not force Garbage collection in Java**; it will only **trigger** if JVM thinks it needs a garbage collection **based on**[**Java heap size**](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html).  
7) There are methods like **System.gc ()** and **Runtime.gc** () which **is used to send request of Garbage collection to JVM** but it’s *not guaranteed that garbage collection will happen*.  
8) If there is no memory space for creating new object in Heap **Java Virtual Machine** throws **OutOfMemoryError**or [**java.lang.OutOfMemoryError heap space**](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html)

# Q1) Which part of the memory is involved in Garbage Collection? Stack or Heap?

# Ans) Heap

# Q2)What is responsiblity of Garbage Collector?

# Ans) Garbage collector frees the memory occupied by the unreachable objects during the java program by deleting these unreachable objects.

# It ensures that the available memory will be used efficiently, but does not guarantee that there will be sufficient memory for the program to run.

# Q3) Is garbage collector a dameon thread?

# Ans) Yes GC is a dameon thread. A dameon thread runs behind the application. It is started by JVM. The thread stops when all non-dameon threads stop.

# Q4)Garbage Collector is controlled by whom?

# Ans) The JVM controls the Garbage Collector; it decides when to run the Garbage Collector. JVM runs the Garbage Collector when it realizes that the memory is running low, but this behavior of jvm can not be guaranteed.

# One can request the Garbage Collection to happen from within the java program but there is no guarantee that this request will be taken care of by jvm.

# Q5) When does an object become eligible for garbage collection?

# Ans) An object becomes eligible for Garbage Collection when no live thread can access it.

# Q6) What are the different ways to make an object eligible for Garbage Collection when it is no longer needed?

# Ans)

# 1. Set all available object references to null once the purpose of creating the object is served :

# public class GarbageCollnTest1 {

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

# String str = "Set the object ref to null";

# //String object referenced by variable str is not eligible for GC yet

# str = null;

# /\*String object referenced by variable str becomes eligible for GC \*/

# }

# }

# 2. Make the reference variable to refer to another object : Decouple the reference variable from the object and set it refer to another object, so the object which it was referring to before reassigning is eligible for Garbage Collection.

# publc class GarbageCollnTest2 {

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

# String str1 = "Garbage collected after use";

# String str2 = "Another String";

# System.out.println(str1);

# //String object referred by str1 is not eligible for GC yet

# str1 = str2;

# /\* Now the str1 variable referes to the String object "Another String" and the object "Garbage collected after use" is not referred by any variable and hence is eligible for GC \*/

# }

# }

# 3) Creating Islands of Isolation : If you have two instance reference variables which are referring to the instances of the same class, and these two reference variables refer to each other and the objects referred by these reference variables do not have any other valid reference then these two objects are said to form an Island of Isolation and are eligible for Garbage Collection.

# public class GCTest3 {

# GCTest3 g;

# public static void main(String [] str){

# GCTest3 gc1 = new GCTest3();

# GCTest3 gc2 = new GCTest3();

# gc1.g = gc2; //gc1 refers to gc2

# gc2.g = gc1; //gc2 refers to gc1

# gc1 = null;

# gc2 = null;

# //gc1 and gc2 refer to each other and have no other valid //references

# //gc1 and gc2 form Island of Isolation

# //gc1 and gc2 are eligible for Garbage collection here

# }

# }

# Q7) Can the Garbage Collection be forced by any means?

# Ans) No. The Garbage Collection can not be forced, though there are few ways by which it can be requested there is no guarantee that these requests will be taken care of by JVM.

# Q8) How can the Garbage Collection be requested?

# Ans) There are two ways in which we can request the jvm to execute the Garbage Collection.

# 1) The methods to perform the garbage collections are present in the Runtime class provided by java. The Runtime class is a Singleton for each java main program.

# The method getRuntime() returns a singleton instance of the Runtime class. The method gc() can be invoked using this instance of Runtime to request the garbage collection.

# 2) Call the System class System.gc() method which will request the jvm to perform GC.

# Q9) What is the purpose of overriding finalize() method?

# Ans) The finalize() method should be overridden for an object to include the clean up code or to dispose of the system resources that should to be done before the object is garbage collected.

protected void finalize() throws Throwable {}

* every class inherits the finalize() method from java.lang.Object
* **the method is called by the garbage collector when it determines no more references to the object exist**
* the Object finalize method performs no actions but it may be overridden by any class
* normally it should be overridden to clean-up non-Java resources ie closing a file
* if overridding finalize() it is good programming practice to use a try-catch-finally statement and to always call super.finalize(). This is a safety measure to ensure you do not inadvertently miss closing a resource used by the objects calling class
* protected void finalize() throws Throwable {
* try {
* close(); // close open files
* } finally {
* super.finalize();
* }

}

* any exception thrown by finalize() during garbage collection halts the finalization but is otherwise ignored
* finalize() is never run more than once on any object

# Q10) If an object becomes eligible for Garbage Collection and its finalize() method has been called and inside this method the object becomes accessible by a live thread of execution and is not garbage collected. Later at some point the same object becomes eligible for Garbage collection, will the finalize() method be called again?

# Ans) No

# Q11) How many times does the garbage collector calls the finalize() method for an object?

# Ans) Only once.

# Q12) What happens if an uncaught exception is thrown from during the execution of the finalize() method of an object?

# Ans) The exception will be ignored and the garbage collection (finalization) of that object terminates.

# Q13) What are different ways to call garbage collector?

# Ans) Garbage collection can be invoked using System.gc() or Runtime.getRuntime().gc().

# Q14) How to enable/disable call of finalize() method of exit of the application

# Ans) Runtime.getRuntime().runFinalizersOnExit(boolean value) . Passing the boolean value will either disable or enable the finalize() call.

# <http://java-questions.com/garbagecollection-interview-questions.html>

# Q15) At what point is the Bar object, created on line 6, eligible for garbage collection?

class Bar { }

class Test

{

Bar doBar()

{

Bar b = new Bar(); /\* Line 6 \*/

return b; /\* Line 7 \*/

}

public static void main (String args[])

{

Test t = new Test(); /\* Line 11 \*/

Bar newBar = t.doBar(); /\* Line 12 \*/

System.out.println("newBar");

newBar = new Bar(); /\* Line 14 \*/

System.out.println("finishing"); /\* Line 15 \*/

}

}

# All references to the Bar object created on line 6 are destroyed when a new reference to a new Bar object is assigned to the variable newBar on line 14. Therefore the Bar object, created on line 6, is eligible for garbage collection after line 14.

# 16) When is the Float object, created in line 3, eligible for garbage collection?

public Object m()

{

Object o = new Float(3.14F);

Object [] oa = new Object[l];

oa[0] = o; /\* Line 5 \*/

o = null; /\* Line 6 \*/

oa[0] = null; /\* Line 7 \*/

return o; /\* Line 8 \*/

}

# A. just after line 5 B. just after line 6

# C. just after line 7 D. just after line 8

Option B is wrong. The reference o is set to null, **but, oa[0] still maintains the reference to the Float object.** Option C is correct. The thread of execution will then not have access to the object.

<http://www.indiabix.com/online-test/java-programming-test/63>

# Collection Interview Questions

# What is Collection in Java?

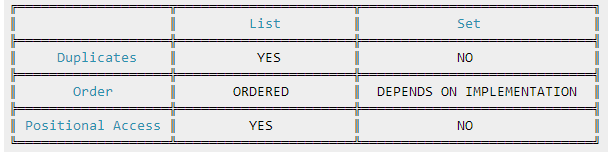
# public interface Collection<E> extends Iterable<E>

# The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

# <http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html>

# Difference between List and Set

List is an ordered sequence of elements whereas Set is a distinct list of elements which is unordered



List allows null elements and you can have many null objects in a List, because it also allowed duplicates. Set just allow one nullelement, and always use iterator.next() method to get the element in Set as it doesn’t provide index accessibility.

<http://stackoverflow.com/questions/1035008/what-is-the-difference-between-set-and-list>

List<E>: An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Set<E>: A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction.

1. What is Big-O annotation, give example ?

The Big-O notation describes the performance of an algorithm in terms of number of elements in a data structure. Since Collection classes are actually data structures, we usually tend to use Big-O notation to choose the collection implementation to use based on time, memory and performance.

Example 1: ArrayList get(index i) is a constant-time operation and doesn’t depend on the number of elements in the list. So it’s performance in Big-O notation is O(1).

Example 2: A linear search on array or list performance is O(n) because we need to search through entire list of elements to find the element.

1. What is an iterator ?

<http://www.journaldev.com/1330/java-collections-interview-questions-and-answers#hashcode-equals>

Java Collection iterator provides a generic way for traversal through the elements of a collection

1. What is different ways to traverse a list?

<http://www.journaldev.com/1330/java-collections-interview-questions-and-answers>

We can iterate over a list in two different ways – using iterator and using for-each loop.

List<String> strList = new ArrayList<>();

//using for-each loop

for(String obj : strList){

System.out.println(obj);

}

//using iterator

Iterator<String> it = strList.iterator();

while(it.hasNext()){

String obj = it.next();

System.out.println(obj);

}

Using iterator is more thread-safe because it makes sure that if underlying list elements are modified, it will throw ConcurrentModificationException.

1. What is iterator fail-fast ?

<http://www.journaldev.com/1330/java-collections-interview-questions-and-answers>

Iterator fail-fast property checks for any modification in the structure of the underlying collection everytime we try to get the next element. If there are any modifications found, it throws ConcurrentModificationException. All the implementations of Iterator in Collection classes are fail-fast by design except the concurrent collection classes like ConcurrentHashMap and CopyOnWriteArrayList.

1. Why can't Java ArrayList be accessed by unique key?

The elements in the collection are accessed using a unique key: That would be describing a Map instead of a List. I would have asked myself the same question, and indeed the array index can be thought of as a "key". However since such a key would be implicitly unique, the last statement seems more descriptive of a map structure.

A key is usually associated with a map or table, or any collection of pairs. A list is not really an associative array so they should be different concepts. However nothing prevents you from thinking of an index as a "key". It depends on the context of the question.

<http://stackoverflow.com/questions/25472891/why-cant-java-arraylist-be-accessed-by-unique-key>

1. Sortedset ?

This means that when you iterate the elements of a SortedSet the elements are returned in the sorted order.

The order of the sorting is either the natural sorting order of the elements (if they implement java.lang.Comparable), or the order determined by a Comparator that you can give to the SortedSet.

By default the elements are iterated in ascending order, starting with the "smallest" and moving towards the "largest". But it is also possible to iterate the elements in descending order using the method TreeSet.descendingIterator().

The Java Collections API only has one implementation of the SortedSet interface - the java.util.TreeSet class. The java.util.concurrent package also has an implementation of this interface, but I will leave the concurrency utilities out of this trail.

Here are two examples of how to create a SortedSet:

SortedSet setA = new TreeSet();

Comparator comparator = new MyComparator();

SortedSet setB = new TreeSet(comparator);

1. Difference between Comparator and Comparable ?

Comparable: A comparable object is capable of comparing itself with another object. The class itself must implements the java.lang.Comparable interface in order to be able to compare its instances.

Comparator: A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class’s instances. This comparator class must implement the java.util.Comparator interface.

(1) Sorting logic: Comparable must be in same class whose objects are being sorted. Hence this is called natural ordering of objects / Comparator is in separate class. Hence we can write different sorting based on different attributes of objects to be sorted

(2) Sorting method: Comparable int compareTo(Object o1)This method compares this object with o1 object and returns a integer. Its value has following meaning (positive – this object is greater than o1, zero – this object equals to o1, negative – this object is less than o1) / Comparator int compare(Object o1,Object o2)  This method compares o1 and o2 objects. and returns a integer. Its value has following meaning.( positive – o1 is greater than o2, zero – o1 equals to o2 , negative – o1 is less than o1)

(3) Calling method: Comparable is Collections.sort(List) / Comparator is Collections.sort(List, comparator)

1. What is difference between List/Set/Map ?

<http://java67.blogspot.com/2013/01/difference-between-set-list-and-map-in-java.html>

Duplicate Objects

Main difference between List and Set interface in Java is that List allows duplicates while Set doesn't allow duplicates. All implementation of Set honor this contract. Map holds two object per Entry e.g. key and value and It may contain duplicate values but keys are always unique. See here for more difference between List and Set data structure in Java.

Order

Another key difference between List and Set is that List is an ordered collection, List's contract maintains insertion order or element. Set is an unordered collection, you get no guarantee on which order element will be stored. Though some of the Set implementation e.g. LinkedHashSet maintains order. Also SortedSet and SortedMap e.g. TreeSet and TreeMap maintains a sorting order, imposed by using Comparator or Comparable.

Index(Position) Accessiable

List implementation as Arraylist is index accessiable (Linkedlist is not), but Set is not index accessiable. We need to use iterator to go through Set to find certain element.

Null elements

List allows null elements and you can have many null objects in a List, because it also allowed duplicates. Set just allow one null element as there is no duplicate permitted while in Map you can have null values and at most one null key. worth noting is that Hashtable doesn't allow null key or values but HashMap allows null values and one null keys. This is also the main difference between these two popular implementation of Map interface, aka HashMap vs Hashtable.

When to use List / Set / Map ?

1) If you need to access elements frequently by using index, than List is a way to go. Its implementation e.g. ArrayList provides faster access if you know index.

2) If you want to store elements and want them to maintain an order on which they are inserted into collection then go for List again, as List is an ordered collection and maintain insertion order.

3) If you want to create collection of unique elements and don't want any duplicate than choose any Set implementation e.g. HashSet, LinkedHashSet or TreeSet. All Set implementation follow there general contract e.g. uniqueness but also add addition feature e.g. TreeSet is a SortedSet and elements stored on TreeSet can be sorted by using Comparator or Comparable in Java. LinkedHashSet also maintains insertion order.

4) If you store data in form of key and value than Map is the way to go.

1. When to use set when to use hashmap ?

<http://stackoverflow.com/questions/5689517/java-hashset-vs-hashmap>

The map holds unique keys. When you invoke put with a key that exists in the map, the object under that key is replaced with the new object. Hence the size 1.

The difference between the two should be obvious:

in a Map you store key-value pairs

in a Set you store only the keys

In fact, a HashSet has a HashMap field, and whenever add(obj) is invoked, the put method is invoked on the underlying map map.put(obj, DUMMY) - where the dummy object is a private static final Object DUMMY = new Object(). So the map is populated with your object as key, and a value that is of no interest.

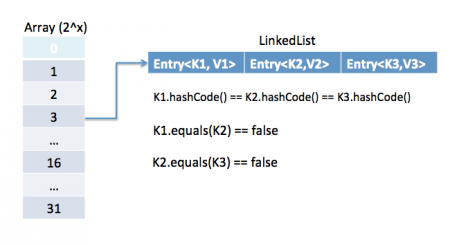
1. How hashmap works in Java ?

<http://www.journaldev.com/1330/java-collections-interview-questions-and-answers#hashmap-working>

HashMap stores key-value pair in Map.Entry static nested class implementation. HashMap works on hashing algorithm and uses hashCode() and equals() method in put and get methods.

When we call put method by passing key-value pair, HashMap uses Key hashCode() with hashing to find out the index to store the key-value pair. The Entry is stored in the LinkedList, so if there are already existing entry, it uses equals() method to check if the passed key already exists, if yes it overwrites the value else it creates a new entry and store this key-value Entry.

When we call get method by passing Key, again it uses the hashCode() to find the index in the array and then use equals() method to find the correct Entry and return it’s value. Below image will explain these detail clearly.



The other important things to know about HashMap are capacity, load factor, threshold resizing. HashMap initial default capacity is 16 and load factor is 0.75. Threshold is capacity multiplied by load factor and whenever we try to add an entry, if map size is greater than threshold, HashMap rehashes the contents of map into a new array with a larger capacity. The capacity is always power of 2, so if you know that you need to store a large number of key-value pairs, for example in caching data from database, it’s good idea to initialize the HashMap with correct capacity and load factor.

More explicit example of HashMap

<https://www.javacodegeeks.com/2014/03/how-hashmap-works-in-java.html>

In brief: In HashMap, hashcode() method is used for set/find the position of linkedlist(which store entries), equals() method is used for finally match the right entry stored on linkedlist by checking key value of that entry. (which means key object’s equals() method is used for ensure uniqueness of key object)

I have a Country class, we are going to use Country class object as key and its capital name(string) as value. Below example will help you to understand, how these key value pair will be stored in hashmap.

Country.java

package org.arpit.javapostsforlearning;

public class Country {

String name;

long population;

public Country(String name, long population) {

super();

this.name = name;

this.population = population;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public long getPopulation() {

return population;

}

public void setPopulation(long population) {

this.population = population;

}

// If length of name in country object is even then return 31(any random number) and if odd then return 95(any random number).

// This is not a good practice to generate hashcode as below method but I am doing so to give better and easy understanding of hashmap.

@Override

public int hashCode() {

if(this.name.length()%2==0)

return 31;

else

return 95;

}

@Override

public boolean equals(Object obj) {

Country other = (Country) obj;

if (name.equalsIgnoreCase((other.name)))

return true;

return false;

}

}

HashMapStructure.java(main class)

import java.util.HashMap;

import java.util.Iterator;

public class HashMapStructure {

/\*\*

\* @author Arpit Mandliya

\*/

public static void main(String[] args) {

Country india=new Country("India",1000);

Country japan=new Country("Japan",10000);

Country france=new Country("France",2000);

Country russia=new Country("Russia",20000);

HashMap<country,string> countryCapitalMap=new HashMap<country,string>();

countryCapitalMap.put(india,"Delhi");

countryCapitalMap.put(japan,"Tokyo");

countryCapitalMap.put(france,"Paris");

countryCapitalMap.put(russia,"Moscow");

Iterator<country> countryCapitalIter=countryCapitalMap.keySet().iterator();

//put debug point at this line

while(countryCapitalIter.hasNext())

{

Country countryObj=countryCapitalIter.next();

String capital=countryCapitalMap.get(countryObj);

System.out.println(countryObj.getName()+"----"+capital);

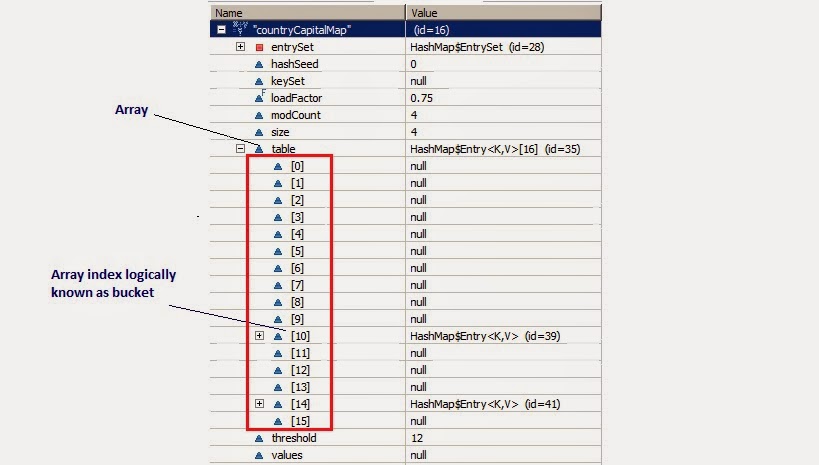
}

}

}

</country></country,string></country,string>

Now put debug point at line 23 and right click on project->debug as-> java application. Program will stop execution at line 23 then right click on countryCapitalMap then select watch.You will be able to see structure as below.



Now From above diagram, you can observe following points

1. There is an Entry[] array called table which has size 16.

2. This table stores Entry class’s object. HashMap class has a inner class called Entry.This Entry have key value as instance variable. Lets see structure of entry class Entry Structure.

static class Entry implements Map.Entry

{

final K key;

V value;

Entry next;

final int hash;

...//More code goes here

}

3. Whenever we try to put any key value pair in hashmap, Entry class object is instantiated for key value and that object will be stored in above mentioned Entry[](table). Now you must be wondering, where will above created Enrty object get stored(exact position in table). The answer is, hash code is calculated for a key by calling Hascode() method. This hashcode is used to calculate index for above Entry[] table.

4. Now, If you see at array index 10 in above diagram, It has an Entry object named HashMap$Entry.

5. We have put 4 key-values in hashmap but it seems to have only 2!!!!This is because if two objects have same hashcode, they will be stored at same index. Now question arises how? It stores objects in a form of LinkedList(logically).

So how hashcode of above country key-value pairs are calculated.

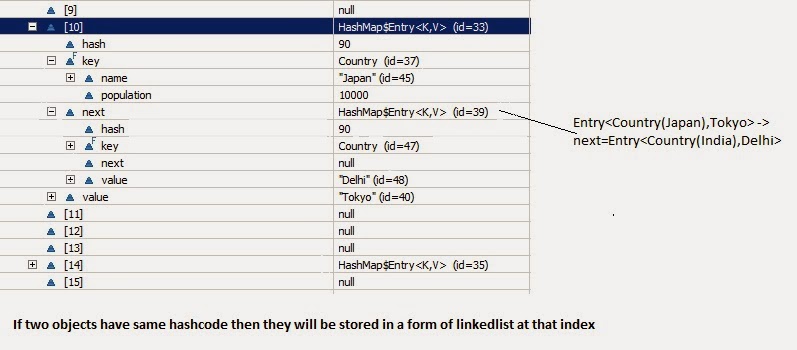
Hashcode for Japan = 95 as its length is odd.

Hashcode for India =95 as its length is odd

HashCode for Russia=31 as its length is even.

HashCode for France=31 as its length is even.

Below diagram will explain LinkedList concept clearly.



So now if you have good understanding of hashmap structure,Lets go through put and get method.

Put :

Lets see implementation of put method:

/\*\*

\* Associates the specified value with the specified key in this map. If the

\* map previously contained a mapping for the key, the old value is

\* replaced.

\*

\* @param key

\* key with which the specified value is to be associated

\* @param value

\* value to be associated with the specified key

\* @return the previous value associated with <tt>key</tt>, or <tt>null</tt>

\* if there was no mapping for <tt>key</tt>. (A <tt>null</tt> return

\* can also indicate that the map previously associated

\* <tt>null</tt> with <tt>key</tt>.)

\*/

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;

}

Now lets understand above code step by step

(1) Key object is checked for null. If key is null then it will be stored at table[0] because hashcode for null is always 0.

(2) Key object’s hashcode() method is called and hash code is calculated. This hashcode is used to find index of array for storing Entry object. It may happen sometimes that, this hashcode function is poorly written so JDK designer has put another function called hash() which takes above calculated hash value as argument.If you want to learn more about hash() function, you can refer hash and indexFor method in hashmap.

(3) indexFor(hash,table.length) is used to calculate exact index in table array for storing the Entry object.

(4) As we have seen in our example, if two key objects have same hashcode(which is known as collision) then it will be stored in form of linkedlist.So here, we will iterate through our linkedlist.

(5) If there is no element present at that index which we have just calculated then it will directly put our Entry object at that index.

(6) If There is element present at that index then it will iterate until it gets Entry->next as null.Then current Entry object become next node in that linkedlist

(5)/(6) are implement in addEntry(hash, key, value, i);

(7) What if we are putting same key again, logically it should replace old value. Yes, it will do that.While iterating it will check key equality by calling equals() method(key.equals(k)), if this method returns true then it replaces value object with current Entry’s value object.

Get:

Lets see implementation of get now:

/\*\*

\* Returns the value to which the specified key is mapped, or {@code null}

\* if this map contains no mapping for the key.

\*

\* <p>

\* More formally, if this map contains a mapping from a key {@code k} to a

\* value {@code v} such that {@code (key==null ? k==null :

\* key.equals(k))}, then this method returns {@code v}; otherwise it returns

\* {@code null}. (There can be at most one such mapping.)

\*

\* </p><p>

\* A return value of {@code null} does not <i>necessarily</i> indicate that

\* the map contains no mapping for the key; it's also possible that the map

\* explicitly maps the key to {@code null}. The {@link #containsKey

\* containsKey} operation may be used to distinguish these two cases.

\*

\* @see #put(Object, Object)

\*/

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) As you got the understanding on put functionality of hashmap. So to understand get functionality is quite simple. If you pass any key to get value object from hashmap.

(2) Key object is checked for null. If key is null then value of Object resides at table[0] will be returned.

(3) Key object’s hashcode() method is called and hash code is calculated.

(4) indexFor(hash,table.length) is used to calculate exact index in table array using generated hashcode for getting the Entry object.

(5) After getting index in table array, it will iterate through linkedlist and check for key equality by calling equals() method and if it returns true then it returns the value of Entry object else returns null.

Key points to Remeber:

(1) HashMap has a inner class called Entry which stores key-value pairs.

(2) Above Entry object is stored in Entry[ ](Array) called table

(3) An index of table is logically known as bucket and it stores first element of linkedlist

(4) Key object’s hashcode() is used to find bucket of that Entry object.

(5) If two key object ‘s have same hashcode , they will go in same bucket of table array.

(6) Key object ‘s equals() method is used to ensure uniqueness of key object.

(7) Value object ‘s equals() and hashcode() method is not used at all

1. How to design a good key for hashmap ?

<http://stackoverflow.com/questions/20212440/immutable-objects-and-hashmap-keys>

String, Integer and other wrapper classes are natural candidates of HashMap key, and String is most frequently used key as well because String is immutable and final,and overrides equals and hashcode() method. Other wrapper class also shares similar property. Immutabiility is required, in order to prevent changes on fields used to calculate hashCode() because if key object return different hashCode during insertion and retrieval than it won't be possible to get object from HashMap. Immutability is best as it offers other advantages as well like thread-safety, If you can keep your hashCode same by only making certain fields final, then you go for that as well. Since equals() and hashCode() method is used during reterival of value object from HashMap, its important that key object correctly override these methods and follow contact. If unequal object return different hashcode than chances of collision will be less which subsequently improve performance of HashMap.

1. What is the use of adding a null key or value into hashmap ?

<http://stackoverflow.com/questions/2945309/what-is-the-use-of-adding-a-null-key-or-value-to-a-hashmap-in-java>

I'm not positive what you're asking, but if you're looking for an example of when one would want to use a null key, I use them often in maps to represent the default case (i.e. the value that should be used if a given key isn't present):

Map<A, B> foo;

A search;

B val = foo.containsKey(search) ? foo.get(search) : foo.get(null);

HashMap handles null keys specially (since it can't call .hashCode() on a null object), but null values aren't anything special, they're stored in the map like anything else

1. How the HashSet implement internally?

<http://java67.blogspot.com/2014/01/how-hashset-is-implemented-or-works-internally-java.html>

HashSet is internally implemented using HashMap 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<>();

}

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;

}

How Object is retrieved from HashSet(Major difference between HashMap)

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.

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();

}

1. What is different between set and hashset ?

<http://stackoverflow.com/questions/5139724/whats-the-difference-between-hashset-and-set>

A Set represents a generic "set of values". A TreeSet is a set where the elements are sorted (and thus ordered), a HashSet is a set where the elements are not sorted or ordered.

A HashSet is typically a lot faster than a TreeSet. A TreeSet is typically implemented as a red-black tree (See http://en.wikipedia.org/wiki/Red-black\_tree - I've not validated the actual implementation of sun/oracle's TreeSet), whereas a HashSet uses Object.hashCode() to create an index in an array. Access time for a red-black tree is O(log(n)) whereas access time for a HashSet ranges from constant-time to the worst case (every item has the same hashCode) where you can have a linear search time O(n).

1. What is difference between ConcurrentHashMap and Collections.synchronizedMap(Map)?

<http://stackoverflow.com/questions/510632/whats-the-difference-between-concurrenthashmap-and-collections-synchronizedmap>

<http://crunchify.com/hashmap-vs-concurrenthashmap-vs-synchronizedmap-how-a-hashmap-can-be-synchronized-in-java/>

a. The locking for ConcurrentHashMap is at a much finer granularity at a hashmap bucket level, there is no locking at object level, it is thread safe without synchronizing the whole map, reads can happen very fast while write is done with a lock.

b. The locking for SynchronizedHashMap is at object level, the whole map is under synchronization, this essentially gives access to only one thread to the entire map and blocks all the other threads, every read and writes operation needs to acquire lock.

c. You should use ConcurrentHashMap when you need very high concurrency in your project, and performance is critical, you should use SynchronizedHashMap if need to ensure data consistency

d. ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it. SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification.

1. Difference between Arraylist vs Linkedlist ?

<http://beginnersbook.com/2013/12/difference-between-arraylist-and-linkedlist-in-java/>

a.Search: ArrayList search operation is pretty fast than LinkedList. get(int index) in ArrayList = O(1), but LinkedList = O(n). Reason: ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements doubly linked list which requires the traversal through all the elements for searching an element.

b. Deletion / Insert: LinkedList = O(1), but ArrayList = O(n) in worst case (while removing or adding first element) and O(1) in best case (While removing or adding last element). Reason: LinkedList’s each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

c. Memory Overhead: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

d. When to use Arraylist or Linkedlist ?

A requirement of frequent addition and deletion in application then LinkedList is a best choice, less add and remove operations and more search operations requirement, ArrayList would be your best bet.

e. Similarity:

(1)Both ArrayList and LinkedList are implementation of List interface.

(2)They both maintain the elements insertion order which means while displaying ArrayList and LinkedList elements the result set would be having the same order in which the elements got inserted into the List.

(3) Both these classes are non-synchronized and can be made synchronized explicitly by using Collections.synchronizedList method.

(4)The iterator and listIterator returned by these classes are fail-fast (if list is structurally modified at any time after the iterator is created, in any way except through the iterator’s own remove or add methods, the iterator will throw a ConcurrentModificationException).

1. What is difference between Array vs ArrayList ?

<http://java67.blogspot.com/2012/12/difference-between-array-vs-arraylist-java.html>

a. Size Issue: First and Major difference between Array and ArrayList in Java is that Array is a fixed length data structure while ArrayList is a variable length Collection class. You can not change length of Array once created in Java but ArrayList re-size itself when gets full depending upon capacity and load factor. Since ArrayList is internally backed by Array in Java, any resize operation in ArrayList will slow down performance as it involves creating new Array and copying content from old array to new array.

b. Generic Issue: Another difference between Array and ArrayList in Java is that you can not use Generics along with Array, as Array instance knows about what kind of type it can hold and throws ArrayStoreException, if you try to store type which is not convertible into type of Array. ArrayList allows you to use Generics to ensure type-safety.

c. Caculate Length Issue: You can also compare Array vs ArrayList on How to calculate length of Array or size of ArrayList. All kinds of Array provides length variable which denotes length of Array while ArrayList provides size() method to calculate size of ArrayList in Java

d. Primitive or Objects Issue: One more major difference between ArrayList and Array is that, you can not store primitives in ArrayList, it can only contain Objects. While Array can contain both primitives and Objects in Java. Though Autoboxing of Java 5 may give you an impression of storing primitives in ArrayList, it actually automatically converts primitives to Object.

e. Mandatory or Not on initial size Issue: One more difference on Array vs ArrayList is that you can create instance of ArrayList without specifying size, Java will create Array List with default size but its mandatory to provide size of Array while creating either directly or indirectly by initializing Array while creating it. By the way you can also initialize ArrayList while creating it

1. What is difference between fail-fast and fail-safe ?

<http://java67.blogspot.com/2015/06/what-is-fail-safe-and-fail-fast-iterator-in-java.html>

(1)Fail-fast iterator traverse over original collection class while fail-safe iterator traverse over a copy or view of original collection. That's why they don't detect any change on original collection classes and this also means that you could operate with stale value.

(2) Fail-fast Iterator throws ConcurrentModfiicationException as soon as they detect any structural change in collection during iteration, basically which changes the modCount variable hold by Iterator. While fail-fast iterator doesn't throw.

(3)Iterator returned by synchronized Collection are fail-fast while iterator returned by concurrent collections are fail-safe in Java.

(4)Fail fast iterator works in live data but become invalid when data is modified while fail-safe iterator are weekly consistent.

1. You need to store elements in a collection that guarantees that no duplicates are stored and all elements can be accessed in natural order. Which interface provides that capability?

A. java.util.Map

B. java.util.Set

C. java.util.List

D. java.util.Collection

Option A is wrong. A map is an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order (ascending key order); others, like the HashMap class, do not (does not guarantee that the order will remain constant over time).

Option B is correct. A set is a collection that contains no duplicate elements. The iterator returns the elements in no particular order (unless this set is an instance of some class that provides a guarantee). A map cannot contain duplicate keys but it may contain duplicate values. List and Collection allow duplicate elements.

Option C is wrong. A list is an ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Unlike sets, lists typically allow duplicate elements.

Option D is wrong. A collection is also known as a sequence. The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Unlike sets, lists typically allow duplicate elements.

<http://www.indiabix.com/online-test/java-programming-test/63>

1. What will be the output of the program?

TreeSet map = new TreeSet();

map.add("one");

map.add("two");

map.add("three");

map.add("four");

map.add("one");

Iterator it = map.iterator();

while (it.hasNext() )

{

System.out.print( it.next() + " " );

}

TreeSet assures no duplicate entries; also, **when it is accessed it will return elements in natural order, which typically means alphabetical.**

four one three two

<http://www.indiabix.com/online-test/java-programming-test/64>

1. Difference between HashMap and HashTable ?

|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| HashMap lets you have null values as well as one null key. | HashTable does not allows null values as key and value. |
| The iterator in the HashMap is fail-safe (If you change the map while iterating, you’ll know). | The enumerator for the HashTable is not fail-safe. |
| HashMap is unsynchronized. | HashTable is synchronized. |

# More detail explaination:

<http://stackoverflow.com/questions/40471/differences-between-hashmap-and-hashtable>

1. [Hashtable](http://java.sun.com/javase/7/docs/api/java/util/Hashtable.html) is synchronized, whereas [HashMap](http://java.sun.com/javase/7/docs/api/java/util/HashMap.html) is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
2. Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
3. One of HashMap's subclasses is [LinkedHashMap](http://java.sun.com/javase/7/docs/api/java/util/LinkedHashMap.html), so in the event that you'd want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.

Hashtable maintain similar mechanism as Enumerator.

Since synchronization is not an issue for you, I'd recommend HashMap. If synchronization becomes an issue, you may also look at [ConcurrentHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html).

# Note: Only one NULL is allowed as a key in HashMap. HashMap does not allow multiple keys to be NULL. Nevertheless, it can have multiple NULL

# (7) ArrayList Vs. Vector?

1) **Synchronization** - ArrayList is not thread-safe whereas Vector is thread-safe. In Vector class each method like add(), get(int i) is surrounded with a synchronized block and thus making Vector class thread-safe.

2) **Data growth** - Internally, both the ArrayList and Vector hold onto their contents using an Array. When an element is inserted into an ArrayList or a Vector, the object will need to expand its internal array if it runs out of room. A Vector defaults to doubling the size of its array, while the ArrayList increases its array size by 50 percent.

3) **Performance**: ArrayList gives better performance as it is non-synchronized. Vector operations gives poor performance as they are thread-safe, the thread which works on Vector gets a lock on it which makes other thread wait till the lock is released.

When to use ArrayList and when to use vector?

It totally depends on the requirement. If there is a need to perform “thread-safe” operation the vector is your best bet as it ensures that only one thread access the collection at a time.**Performance:** Synchronized operations consumes more time compared to non-synchronized ones so if there is no need for thread safe operation, ArrayList is a better choice as performance will be improved because of the concurrent processes.

(8) **If an Employee class is present and its objects are added in an arrayList. Now I want the list to be sorted on the basis of the employeeID of Employee class. What are the steps?**

Ans) 1) Implement Comparable interface for the Employee class and override the compareTo(Object obj) method in which compare the employeeID

2) Now call Collections.sort() method and pass list as an argument.

How to implement custom compareTo method

**First you should type the Comparable interface you're implementing.** Here's what it should look like:

public class Student implements Comparable<Student> {

private int age;

private String name;

@Override

public int compareTo(Student s) {

if (name.equals(s.name))

return age - s.age;

return name.compareTo(s.name));

}

}

Notice how with the typed interface Comparable<Student>, instead of the raw type Comparable, there's no need to cast.

***Now consider that Employee class is a jar file.***

1) **Since Comparable interface cannot be implemented**, create Comparator and override the compare(Object obj, Object obj1) method .

2) Call Collections.sort() on the list and pass comparator as an argument.

<http://stackoverflow.com/questions/18757805/implementing-custom-compareto>

# (9) What is the importance of hashCode() and equals() methods? How they are used in Java?

# <http://www.fromdev.com/2008/05/java-collections-questions.html>

The java.lang.Object has two methods defined in it. They are

public boolean equals(Object obj)

public int hashCode().

These two methods are used heavily when objects are stored in collections. There is a contract between these two methods which should be kept in mind while overriding any of these methods. The Java API documentation describes it in detail.

# The hashCode Method

The hashCode() method returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable or java.util.HashMap.

The general contract of hashCode is: Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. The equals(Object obj) method indicates whether some other object is "equal to" this one.

# The equals Method

The equals method implements an equivalence relation on non-null object references:

It is reflexive: for any non-null reference value x, x.equals(x) should return true.

It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.

It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

For any non-null reference value x, x.equals(null) should return false. The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true). Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

# A practical Example of hashcode() and equals():

This can be applied to classes that need to be stored in Set collections. Sets use equals() to enforce non-duplicates, and HashSet uses hashCode() as a first-cut test for equality. Technically hashCode() isn't necessary then since equals() will always be used in the end, but providing a meaningful hashCode() will improve performance for very large sets or objects that take a long time to compare using equals().

# <http://www.journaldev.com/1330/java-collections-interview-questions-and-answers>

* If o1.equals(o2), then o1.hashCode() == o2.hashCode()should always be true.
* If o1.hashCode() == o2.hashCode is true, it doesn’t mean that o1.equals(o2) will be true.

# <http://www.java2blog.com/2014/02/hashcode-and-equals-method-in-java.html>

(1)If you are overriding equals method then you should override hashcode() also.

(2)If two objects are equal then they must have same hashcode.

(3)If two objects have same hashcode then they may or may not be equal

(4)Always use same attributes to generate equals and hashcode as in our case we have used name

# (10) What happens if equals() is not consistent with compareTo() method?

<http://javarevisited.blogspot.com/2013/08/10-equals-and-hashcode-interview.html>

This is an interesting questions, which asked along with equals() and hashCode() contract. Some java.util.Set implementation e.g. SortedSet or it's concrete implementation TreeSet uses compareTo() method for comparing objects. If compareTo() is not consistent means doesn't return zero, if equals() method returns true, it may break Set contract, which is not to avoid any duplicates.

More explicit explain on this problem is

<http://stackoverflow.com/questions/12123960/what-does-comparison-being-consistent-with-equals-mean-what-can-possibly-happe>

Say we have this simple Student class implementing Comparable<Student> but not overriding equals()/hashCode(). Of course equals() is not consistent with compareTo() - two different students with the same age aren't equal:

class Student implements Comparable<Student> {

private final int age;

Student(int age) {

this.age = age;

}

@Override

public int compareTo(Student o) {

return this.age - o.age;

}

@Override

public String toString() {

return "Student(" + age + ")";

}

}

We can safely use it in TreeMap<Student, String>:

Map<Student, String> students = new TreeMap<Student, String>();

students.put(new Student(25), "twenty five");

students.put(new Student(22), "twenty two");

students.put(new Student(26), "twenty six");

for (Map.Entry<Student, String> entry : students.entrySet()) {

System.out.println(entry);

}

System.out.println(students.get(new Student(22)));

The results are easy to predict: students are nicely sorted according to their age (despite being inserted in different order) and fetching student using new Student(22) key works as well and returns"twenty two". This means we can safely use Student class in TreeMap.

However change students to HashMap and things go bad:

Map<Student, String> students = new HashMap<Student, String>();

Obviously the enumeration of items returns "random" order due to hashing - that's fine, it doesn't violate any Map contract. But the last statement is completely broken. Because HashMap uses equals()/hashCode() to compare instances, fetching value by new Student(22) key fails and returns null!

This is what the JavaDoc tries to explain: such classes will work with TreeMap but might fail to work with other Map implementations. Note that Map operations are documented and defined in terms of equals()/hashCode(), e.g. [containsKey()](http://docs.oracle.com/javase/6/docs/api/java/util/Map.html#containsKey%28java.lang.Object%29):

[...] returns true if and only if this map contains a mapping for a key k such that (key==null ? k==null : key.equals(k))

Thus I don't believe there are any standard JDK classes that implemente Comparable but fail to implement equals()/hashCode() pair.

(11) What happens if you compare an object to null using equals()?

When a null object is passed as an argument to equals() method as equals(null), it should return false, it must not throw NullPointerException, but if you call equals method on reference, which is null as null.equals(), it will throw NullPointerException. That’s why it’s better to use == operator for comparing null e.g. if(object != null) object.equals(anohterObject). By the way, if you comparing String literal with another String object then you better call equals() method on the String literal rather than known object to avoid NPE

(12)

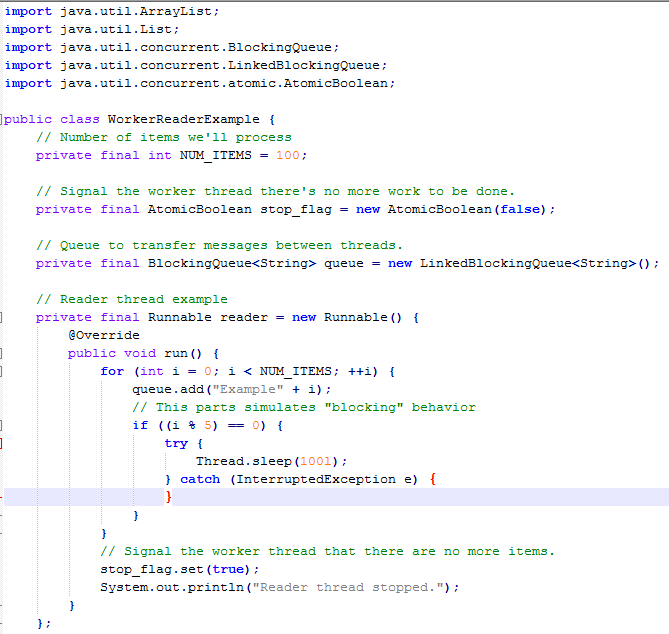
**Java Multi-Thread Questions**

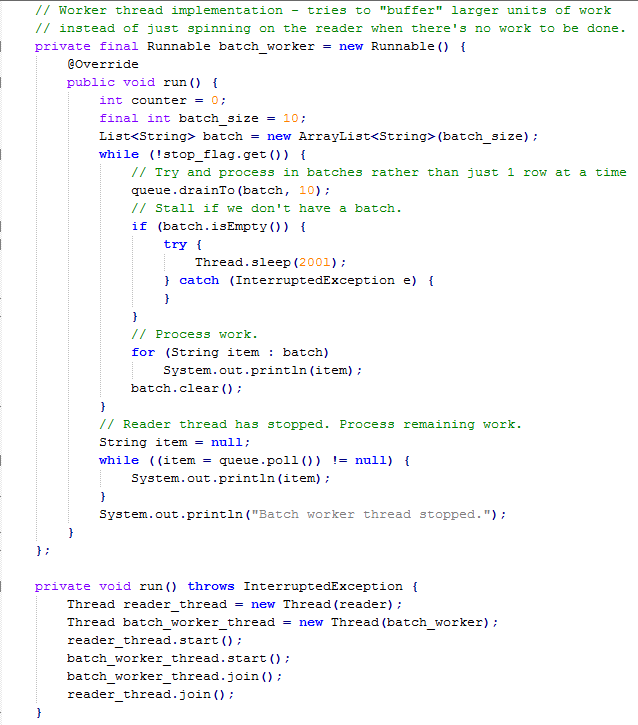
**Why we need multiple thread ?**

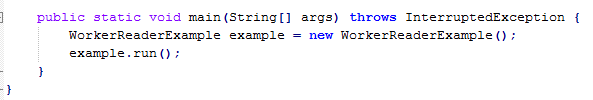
[**https://www.quora.com/What-are-some-real-life-examples-of-multi-threading-as-we-study-in-Java**](https://www.quora.com/What-are-some-real-life-examples-of-multi-threading-as-we-study-in-Java)

A common multithreading example is that I often have a reader thread where it could be blocking (e.g., while reading from a database) and a worker thread where it’s more performance-efficient to process work in batches instead of polling one item at a time.

I’m not a Java expert, but here’s Java6 complaint code that might demonstrate this:







(1) Java Synchronization and concurrency

The words synchronization and concurrency are overlapping and sometimes synonymous terms. The word synchronization generally means sharing data between multiple processors or threads, while concurrency refers to a measure of– or the art of improving– how effectively an application allows multiple jobs required by that application (e.g. serving web page requests from a web server) to run simultaneously.

<http://www.javamex.com/tutorials/synchronization_concurrency_1.shtml>

(2) Two ways to create a thread

<http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html>

Thread Subclass

The first way to specify what code a thread is to run, is to create a subclass of Thread and override the run() method. The run() method is what is executed by the thread after you call start(). Here is an example of creating a Java Thread subclass:

public class MyThread extends Thread {

public void run(){

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

}

}

To create and start the above thread you can do like this:

MyThread myThread = new MyThread();

myTread.start();

The start() call will return as soon as the thread is started. It will not wait until the run() method is done. The run() method will execute as if executed by a different CPU. When the run() method executes it will print out the text "MyThread running".

You can also create an anonymous subclass of Thread like this:

Thread thread = new Thread(){

public void run(){

System.out.println("Thread Running");

}

}

thread.start();

This example will print out the text "Thread running" once the run() method is executed by the new thread.

Runnable Interface Implementation

The second way to specify what code a thread should run is by creating a class that implements java.lang.Runnable. The Runnable object can be executed by a Thread.

Here is a Java Runnable example:

public class MyRunnable implements Runnable {

public void run(){

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

}

}

To have the run() method executed by a thread, pass an instance of MyRunnable to a Thread in its constructor. Here is how that is done:

Thread thread = new Thread(new MyRunnable());

thread.start();

When the thread is started it will call the run() method of the MyRunnable instance instead of executing it's own run() method. The above example would print out the text "MyRunnable running".

You can also create an anonymous implementation of Runnable, like this:

Runnable myRunnable = new Runnable(){

public void run(){

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

}

}

Thread thread = new Thread(myRunnable);

thread.start();

Subclass or Runnable?

There are no rules about which of the two methods that is the best. Both methods works. Personally though, I prefer implementing Runnable, and handing an instance of the implementation to a Thread instance. When having the Runnable's executed by a thread pool it is easy to queue up the Runnable instances until a thread from the pool is idle. This is a little harder to do with Thread subclasses.

Sometimes you may have to implement Runnable as well as subclass Thread. For instance, if creating a subclass of Thread that can execute more than one Runnable. This is typically the case when implementing a thread pool.

(3) class X implements Runnable {

public static void main(String args[]) {

/\* Missing code? \*/}

public void run() {}

}

Which of the following line of code is suitable to start a thread ?

A. Thread t = new Thread(X);

B. Thread t = new Thread(X); t.start();

C. X run = new X(); Thread t = new Thread(run); t.start();

D. Thread t = new Thread(); x.run();

# X run = new X(); //Creates objects that implements run() as per Runnable interface Thread t = new Thread(run); // Creates thread t.start(); // Starts the thread. Note: The above three statements can be written as simply (new Thread(new X()).start(); Hope this help you. Have a nice day!

(4) **Runnable Vs. Thread**

Well so many good Answers, i want to add more on this, This will help to understand Extending v/s Implementing Thread\_  
Extends binds two class files very closely and can cause some pretty hard to deal with code.

Both approaches do the same job but there have been some differences.  
*The most common difference is*

1. When you extends Thread class, after that you can’t extend any other class which you required. (As you know, Java does not allow inheriting more than one class).
2. When you implements Runnable, you can save a space for your class to extend any other class in future or now.

**However, one significant difference between implementing Runnable and extending Thread is that  
by extending Thread, each of your threads has a unique object associated with it, whereas implementing Runnable, many threads can share the same object instance.**

//Implement Runnable Interface...

class ImplementsRunnable implements Runnable {

private int counter = 0;

public void run() {

counter++;

System.out.println("ImplementsRunnable : Counter : " + counter);

}

}

//Extend Thread class...

class ExtendsThread extends Thread {

private int counter = 0;

public void run() {

counter++;

System.out.println("ExtendsThread : Counter : " + counter);

}

}

//Use above classes here in main to understand the differences more clearly...

public class ThreadVsRunnable {

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

// Multiple threads share the same object.

ImplementsRunnable rc = new ImplementsRunnable();

Thread t1 = new Thread(rc);

t1.start();

Thread.sleep(1000); // Waiting for 1 second before starting next thread

Thread t2 = new Thread(rc);

t2.start();

Thread.sleep(1000); // Waiting for 1 second before starting next thread

Thread t3 = new Thread(rc);

t3.start();

// Creating new instance for every thread access.

ExtendsThread tc1 = new ExtendsThread();

tc1.start();

Thread.sleep(1000); // Waiting for 1 second before starting next thread

ExtendsThread tc2 = new ExtendsThread();

tc2.start();

Thread.sleep(1000); // Waiting for 1 second before starting next thread

ExtendsThread tc3 = new ExtendsThread();

tc3.start();

}

}

**Output of the above program.**

ImplementsRunnable : Counter : 1

ImplementsRunnable : Counter : 2

ImplementsRunnable : Counter : 3

ExtendsThread : Counter : 1

ExtendsThread : Counter : 1

ExtendsThread : Counter : 1

In the Runnable interface approach, only one instance of a class is being created and it has been shared by different threads. So the value of counter is incremented for each and every thread access.

Whereas, Thread class approach, you must have to create separate instance for every thread access. Hence different memory is allocated for every class instances and each has separate counter, the value remains same, which means no increment will happen because none of the object reference is same.

<http://stackoverflow.com/questions/541487/implements-runnable-vs-extends-thread>

(5) Sychronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can overrite data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.

Multithreading example without or with Synchronization:

**class** PrintDemo{

**public** **void** printCount(){

**try**{

**for**(**int** i = 5; i > 0; i--){

System.*out*.println("Counter --- " + i);

}

} **catch** (Exception e) {

System.*out*.println("Thread interrupted.");

}

}

}

**class** ThreadDemo **extends** Thread{

Thread t;

PrintDemo PD;

String threadName;

ThreadDemo(String name, PrintDemo pd){

PD = pd;

threadName = name;

}

**public** **void** run(){

//synchronized(PD){

PD.printCount();

//}

System.*out*.println("Thread " + threadName + " exiting.");

}

**public** **void** start(){

System.*out*.println("Starting " + threadName );

**if** (t == **null**){

t = **new** Thread(**this**, threadName);

t.start();

}

}

}

**public** **class** SychronizedTest {

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

PrintDemo PD = **new** PrintDemo();

ThreadDemo T1 = **new** ThreadDemo("Thread-1", PD);

ThreadDemo T2 = **new** ThreadDemo("Thread-2", PD);

// We can use the default start() method

T1.start();

T2.start();

}

}

Here is a simple example which may or may not print counter value in sequence and every time we run it, it produces different result based on CPU availability to a thread.

The output is:

Starting Thread-1

Starting Thread-2

Counter --- 5

Counter --- 4

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 2

Counter --- 1

Counter --- 3

Thread Thread-2 exiting.

Counter --- 2

Counter --- 1

Thread Thread-1 exiting.

But with synchronization:

Starting Thread-1

Starting Thread-2

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 2

Counter --- 1

Thread Thread-1 exiting.

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 2

Counter --- 1

Thread Thread-2 exiting.

Here is the same example which prints counter value in sequence and every time we run it, it produces same result.

<http://www.tutorialspoint.com/java/java_thread_synchronization.htm>

(6) Which of the following class level (nonlocal) variable declarations will not compile?

A. protected int a;

B. transient int b = 3;

C. private synchronized int e;

D. volatile int d;

The synchronized modifier applies only to methods.

(7) What is static synchronization ?

<http://stackoverflow.com/questions/437620/java-synchronized-static-methods-lock-on-object-or-class?lq=1>

A synchronized method acquires a monitor (§17.1) before it executes. For a class (static) method, the monitor associated with the Class object for the method's class is used. For an instance method, the monitor associated with this (the object for which the method was invoked) is used.

<http://stackoverflow.com/questions/578904/how-do-synchronized-static-methods-work-in-java>

By using synchronized on a static method lock you will synchronize the class methods and attributes ( as opposed to instance methods and attributes )

So your assumption is correct.

I am wondering if making the method synchronized is the right approach to ensure thread-safety.

Not really. You should let that work do your RDBMS instead. They are good at this kind of stuff.

The only thing you will get by synchronizing the access to the database is to make your application terribly slow. Further more, in the code you posted you're building a Session Factory each time, that way, your application will spend more time accessing the DB than performing the actual job.

Imagine the following scenario:

Client A and B attempt to insert different information into record X of table T.

With your approach the only thing you're getting is to make sure one is called after the other, when this would happen anyway in the DB, because the RDBMS will prevent them from inserting half information from A and half from B at the same time. The result will be the same but only 5 times ( or more ) slower.

Probably it could be better to take a look at the "Transactions and Concurrency" chapter in the Hibernate documentation. Most of the times the problems you're trying to solve, have been solved already and a much better way.

(8) Race condition and prevention

<http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html>

Race Conditions in Critical Sections

More formally, the situation where 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. Thread synchronization can also be achieved using other synchronization constructs like locks or atomic variables like java.util.concurrent.atomic.AtomicInteger.

(9) Thread safety and shared resource

<http://tutorials.jenkov.com/java-concurrency/thread-safety.html>

Local Variables

Local variables are stored in each thread's own stack. That means that local variables are never shared between threads. That also means that all local primitive variables are thread safe. Here is an example of a thread safe local primitive variable:

public void someMethod(){

long threadSafeInt = 0;

threadSafeInt++;

}

Local Object References

Local references to objects are a bit different. The reference itself is not shared. The object referenced however, is not stored in each threads's local stack. All objects are stored in the shared heap.

If an object created locally never escapes the method it was created in, it is thread safe. In fact you can also pass it on to other methods and objects as long as none of these methods or objects make the passed object available to other threads.

Here is an example of a thread safe local object:

public void someMethod(){

LocalObject localObject = new LocalObject();

localObject.callMethod();

method2(localObject);

}

public void method2(LocalObject localObject){

localObject.setValue("value");

}

The LocalObject instance in this example is not returned from the method, nor is it passed to any other objects that are accessible from outside the someMethod() method. Each thread executing the someMethod() method will create its own LocalObject instance and assign it to the localObject reference. Therefore the use of the LocalObject here is thread safe.

In fact, the whole method someMethod() is thread safe. Even if the LocalObject instance is passed as parameter to other methods in the same class, or in other classes, the use of it is thread safe.

The only exception is of course, if one of the methods called with the LocalObject as parameter, stores the LocalObject instance in a way that allows access to it from other threads.

Object Member Variables

Object member variables (fields) are stored on the heap along with the object. Therefore, if two threads call a method on the same object instance and this method updates object member variables, the method is not thread safe. Here is an example of a method that is not thread safe:

public class NotThreadSafe{

StringBuilder builder = new StringBuilder();

public add(String text){

this.builder.append(text);

}

}

If two threads call the add() method simultaneously on the same NotThreadSafe instance then it leads to race conditions. For instance:

NotThreadSafe sharedInstance = new NotThreadSafe();

new Thread(new MyRunnable(sharedInstance)).start();

new Thread(new MyRunnable(sharedInstance)).start();

public class MyRunnable implements Runnable{

NotThreadSafe instance = null;

public MyRunnable(NotThreadSafe instance){

this.instance = instance;

}

public void run(){

this.instance.add("some text");

}

}

Notice how the two MyRunnable instances share the same NotThreadSafe instance. Therefore, when they call the add() method on the NotThreadSafe instance it leads to race condition.

However, if two threads call the add() method simultaneously on different instances then it does not lead to race condition. Here is the example from before, but slightly modified:

new Thread(new MyRunnable(new NotThreadSafe())).start();

new Thread(new MyRunnable(new NotThreadSafe())).start();

Now the two threads have each their own instance of NotThreadSafe so their calls to the add method doesn't interfere with each other. The code does not have race condition anymore. So, even if an object is not thread safe it can still be used in a way that doesn't lead to race condition.

(10) Thread safety and immutability

<http://tutorials.jenkov.com/java-concurrency/thread-safety-and-immutability.html>

Race conditions occur only if multiple threads are accessing the same resource, and one or more of the threads write to the resource. If multiple threads read the same resource race conditions do not occur.

We can make sure that objects shared between threads are never updated by any of the threads by making the shared objects immutable, and thereby thread safe. Here is an example:

public class ImmutableValue{

private int value = 0;

public ImmutableValue(int value){

this.value = value;

}

public int getValue(){

return this.value;

}

}

Notice how the value for the ImmutableValue instance is passed in the constructor. Notice also how there is no setter method. Once an ImmutableValue instance is created you cannot change its value. It is immutable. You can read it however, using the getValue() method.

The Reference is not Thread Safe!

It is important to remember, that even if an object is immutable and thereby thread safe, the reference to this object may not be thread safe. Look at this example:

public class Calculator{

private ImmutableValue currentValue = null;

public ImmutableValue getValue(){

return currentValue;

}

public void setValue(ImmutableValue newValue){

this.currentValue = newValue;

}

public void add(int newValue){

this.currentValue = this.currentValue.add(newValue);

}

}

The Calculator class holds a reference to an ImmutableValue instance. Notice how it is possible to change that reference through both the setValue() and add() methods. Therefore, even if the Calculator class uses an immutable object internally, it is not itself immutable, and therefore not thread safe. In other words: The ImmutableValue class is thread safe, but the use of it is not. This is something to keep in mind when trying to achieve thread safety through immutability.

To make the Calculator class thread safe you could have declared the getValue(), setValue(), and add() methods synchronized.

(11) Blocking Queue implementation with wait() and notifyAll()

<http://tutorials.jenkov.com/java-concurrency/blocking-queues.html>

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

Here is a diagram showing two threads cooperating via a blocking queue:



public class BlockingQueue {

private List queue = new LinkedList();

private int limit = 10;

public BlockingQueue(int limit){

this.limit = limit;

}

public synchronized void enqueue(Object item)

throws InterruptedException {

while(this.queue.size() == this.limit) {

wait();

}

if(this.queue.size() == 0) {

notifyAll();

}

this.queue.add(item);

}

public synchronized Object dequeue()

throws InterruptedException{

while(this.queue.size() == 0){

wait();

}

if(this.queue.size() == this.limit){

notifyAll();

}

return this.queue.remove(0);

}

}

Notice how notifyAll() is only called from enqueue() and dequeue() if the queue size is equal to the size bounds (0 or limit). If the queue size is not equal to either bound when enqueue() or dequeue() is called, there can be no threads waiting to either enqueue or dequeue items

(12) Where should use BlockingQueue?

<http://stackoverflow.com/questions/358457/where-should-you-use-blockingqueue-implementations-instead-of-simple-queue-imple>

(13) Wait() and notify()/notifyAll() example

<http://stackoverflow.com/questions/2536692/a-simple-scenario-using-wait-and-notify-in-java>

The wait() and notify() methods are designed to provide a mechanism to allow a thread to block until a specific condition is met. For this I assume you're wanting to write a blocking queue implementation, where you have some fixed size backing-store of elements.

The first thing you have to do is to identify the conditions that you want the methods to wait for. In this case, you will want the put() method to block until there is free space in the store, and you will want the take() method to block until there is some element to return.

public class BlockingQueue<T> {

private Queue<T> queue = new LinkedList<T>();

private int capacity;

public BlockingQueue(int capacity) {

this.capacity = capacity;

}

public synchronized void put(T element) throws InterruptedException {

while(queue.size() == capacity) {

wait();

}

queue.add(element);

notify(); // notifyAll() for multiple producer/consumer threads

}

public synchronized T take() throws InterruptedException {

while(queue.isEmpty()) {

wait();

}

T item = queue.remove();

notify(); // notifyAll() for multiple producer/consumer threads

return item;

}

}

There are a few things to note about the way in which you must use the wait and notify mechanisms.

(a)You need to ensure that any calls to wait() or notify() are within a synchronized region of code (with the wait() and notify() calls being synchronized on the same object = “queue” in this example). The reason for this (other than the standard thread safety concerns) is due to something known as a missed signal.

An example of this, is that a thread may call put() when the queue happens to be full, it then checks the condition, sees that the queue is full, however before it can block another thread is scheduled. This second thread then take()'s an element from the queue, and notifies the waiting threads that the queue is no longer full. Because the first thread has already checked the condition however, it will simply call wait() after being re-scheduled, even though it could make progress.

By synchronizing on a shared object (), you can ensure that this problem does not occur, as the second thread's take() call will not be able to make progress until the first thread has actually blocked.

(b) You need to put the condition you are checking in a while loop, rather than an if statement, due to a problem known as spurious wake-ups. This is where a waiting thread can sometimes be re-activated without notify() being called. Putting this check in a while loop will ensure that if a spurious wake-up occurs, the condition will be re-checked, and the thread will call wait() again.

(14) Notify() vs notifyAll() ?

<http://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again>

Clearly, notify wakes (any) one thread in the wait set, notifyAll wakes all threads in the waiting set. The following discussion should clear up any doubts. notifyAll should be used most of the time. If you are not sure which to use, then use notifyAll. Please see explanation that follows.

Read very carefully and understand. Please send me an email if you have any questions.

Look at producer/consumer (assumption is a ProducerConsumer class with two methods). IT IS BROKEN (because it uses notify) - yes it MAY work - even most of the time, but it may also cause deadlock - we will see why:

public synchronized void put(Object o) {

while (buf.size()==MAX\_SIZE) {

wait(); // called if the buffer is full (try/catch removed for brevity)

}

buf.add(o);

notify(); // called in case there are any getters or putters waiting

}

public synchronized Object get() {

// Y: this is where C2 tries to acquire the lock (i.e. at the beginning of the method)

while (buf.size()==0) {

wait(); // called if the buffer is empty (try/catch removed for brevity)

// X: this is where C1 tries to re-acquire the lock (see below)

}

Object o = buf.remove(0);

notify(); // called if there are any getters or putters waiting

return o;

}

FIRSTLY,

**Why do we need a while loop surrounding the wait?**

We need a while loop in case we get this situation:

Consumer 1 (C1) enter the synchronized block and the buffer is empty, so C1 is put in the wait set (via the wait call). Consumer 2 (C2) is about to enter the synchronized method (at point Y above), but Producer P1 puts an object in the buffer, and subsequently calls notify. The only waiting thread is C1, so it is woken and now attempts to re-acquire the object lock at point X (above).

Now C1 and C2 are attempting to acquire the synchronization lock. One of them (nondeterministically) is chosen and enters the method, the other is blocked (not waiting - but blocked, trying to acquire the lock on the method). Let's say C2 gets the lock first. C1 is still blocking (trying to acquire the lock at X). C2 completes the method and releases the lock. Now, C1 acquires the lock. Guess what, lucky we have a while loop, because, C1 performs the loop check (guard) and is prevented from removing a non-existent element from the buffer (C2 already got it!). If we didn't have a while, we would get anIndexArrayOutOfBoundsException as C1 tries to remove the first element from the buffer!

NOW,

**Ok, now why do we need notifyAll?**

In the producer/consumer example above it looks like we can get away with notify. It seems this way, because we can prove that the guards on the *wait* loops for producer and consumer are mutually exclusive. That is, it looks like we cannot have a thread waiting in the put method as well as the get method, because, for that to be true, then the following would have to be true:

buf.size() == 0 AND buf.size() == MAX\_SIZE (assume MAX\_SIZE is not 0)

HOWEVER, this is not good enough, we NEED to use notifyAll. Let's see why ...

Assume we have a buffer of size 1 (to make the example easy to follow). The following steps lead us to deadlock. Note that ANYTIME a thread is woken with notify, it can be non-deterministically selected by the JVM - that is any waiting thread can be woken. Also note that when multiple threads are blocking on entry to a method (i.e. trying to acquire a lock), the order of acquisition can be non-deterministic. Remember also that a thread can only be in one of the methods at any one time - the synchronized methods allow only one thread to be executing (i.e. holding the lock of) any (synchronized) methods in the class. If the following sequence of events occurs - deadlock results:

**STEP 1:**  
- P1 puts 1 char into the buffer

**STEP 2:**  
- P2 attempts put - checks wait loop - already a char - waits

**STEP 3:**  
- P3 attempts put - checks wait loop - already a char - waits

**STEP 4:**  
- C1 attempts to get 1 char   
- C2 attempts to get 1 char - blocks on entry to the get method  
- C3 attempts to get 1 char - blocks on entry to the get method

**STEP 5:**  
- C1 is executing the get method - gets the char, calls notify, exits method  
- The notify wakes up P2  
- BUT, C2 enters method before P2 can (P2 must reacquire the lock), so P2 blocks on entry to the put method  
- C2 checks wait loop, no more chars in buffer, so waits  
- C3 enters method after C2, but before P2, checks wait loop, no more chars in buffer, so waits

**STEP 6:**  
- NOW: there is P3, C2, and C3 waiting!  
- Finally P2 acquires the lock, puts a char in the buffer, calls notify, exits method

**STEP 7:**  
- P2's notification wakes P3 (remember any thread can be woken)  
- P3 checks the wait loop condition, there is already a char in the buffer, so waits.  
- NO MORE THREADS TO CALL NOTIFY and THREE THREADS PERMANENTLY SUSPENDED!

SOLUTION: Replace notify with notifyAll in the producer/consumer code (above).

(15) Solve Consumer Producer pattern by using wait() and notify() methods in multithreading in Java

<http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-pattern-by.html>

<http://javarevisited.blogspot.com/2013/12/inter-thread-communication-in-java-wait-notify-example.html>

(16) What is difference between volatile and synchronization ?

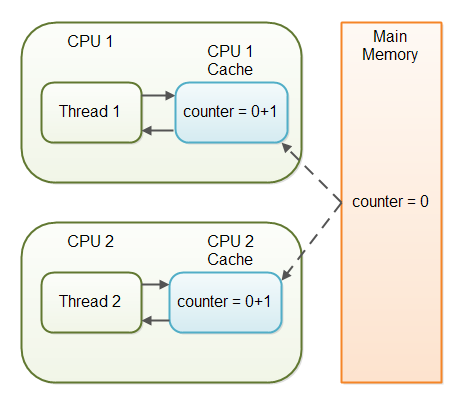
<http://tutorials.jenkov.com/java-concurrency/volatile.html>

The Java volatile keyword guarantees visibility of changes to variables across threads. More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

If two threads are both reading and writing to a shared variable, then using the volatile keyword for that is not enough. You need to use a synchronized in that case to guarantee that the reading and writing of the variable is atomic. Reading or writing a volatile variable does not block threads reading or writing. For this to happen you must use the synchronized keyword around critical sections.

For example:

Imagine if Thread 1 reads a shared counter variable with the value 0 into its CPU cache, increment it to 1 and not write the changed value back into main memory. Thread 2 could then read the same counter variable from main memory where the value of the variable is still 0, into its own CPU cache. Thread 2 could then also increment the counter to 1, and also not write it back to main memory. This situation is illustrated in the diagram below:



In case only one thread reads and writes the value of a volatile variable and other threads only read the variable, then the reading threads are guaranteed to see the latest value written to the volatile variable. Without making the variable volatile, this would not be guaranteed.

(17) What is context-switching in multi-threading?

Context Switching is the process of storing and restoring of CPU state so that Thread execution can be resumed from the same point at a later point of time. Context Switching is the essential feature for multitasking operating system and support for multi-threaded environment.

(18) What is thread scheduler and time slicing ?

Thread Scheduler is the Operating System service that allocates the CPU time to the available runnable threads. Once we create and start a thread, it’s execution depends on the implementation of Thread Scheduler. Time Slicing is the process to divide the available CPU time to the available runnable threads. Allocation of CPU time to threads can be based on thread priority or the thread waiting for longer time will get more priority in getting CPU time. Thread scheduling can’t be controlled by java, so it’s always better to control it from application itself.

(19) How does Thread Communicate with each other ?

When threads share resources, communication between Threads is important to coordinate their efforts. Object class wait(), notify() and notifyAll() methods allows threads to communicate about the lock status of a resource. In Java every Object has a monitor and wait, notify methods are used to wait for the Object monitor or to notify other threads that Object monitor is free now. There is no monitor on threads in java and synchronization can be used with any Object.

(20) Different ways to achieve Thread Safety ?

synchronization, atomic concurrent classes, implementing concurrent Lock interface, using volatile keyword, using immutable classes and Thread safe classes.

(21) Which is more preferred – Synchronized method or Synchronized block?

Synchronized block is more preferred way because it doesn’t lock the Object, synchronized methods lock the Object and if there are multiple synchronization blocks in the class, even though they are not related, it will stop them from execution and put them in wait state to get the lock on Object.

(22) What is Deadlock and how to avoid it ?

When two or more threads are waiting for each other to release lock and get stuck for infinite time, situation is called deadlock .

<http://tutorials.jenkov.com/java-concurrency/deadlock.html>

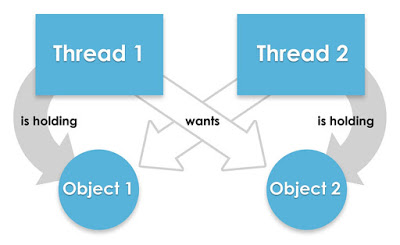
Deadlock is a programming situation where two or more threads are blocked forever

For instance, if thread 1 locks A, and tries to lock B, and thread 2 has already locked B, and tries to lock A, a deadlock arises. Thread 1 can never get B, and thread 2 can never get A. In addition, neither of them will ever know. They will remain blocked on each their object, A and B, forever. This situation is a deadlock.

The situation is illustrated below:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for A



Write a program which will result in deadlock

<http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html>

/\*\*

\* Java program to create a deadlock by imposing circular wait.

\*

\* @author WINDOWS 8

\*

\*/

public class DeadLockDemo {

/\*

\* This method request two locks, first String and then Integer

\*/

public void method1() {

synchronized (String.class) {

System.out.println("Aquired lock on String.class object");

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

}

}

}

/\*

\* This method also requests same two lock but in exactly

\* Opposite order i.e. first Integer and then String.

\* This creates potential deadlock, if one thread holds String lock

\* and other holds Integer lock and they wait for each other, forever.

\*/

public void method2() {

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

synchronized (String.class) {

System.out.println("Aquired lock on String.class object");

}

}

}

}

If method1() and method2() both will be called by two or many threads , there is a good chance of deadlock because if thread 1 acquires lock on Sting object while executing method1() and thread 2 acquires lock on Integer object while executing method2() both will be waiting for each other to release lock on Integer and String to proceed further which will never happen.

Write code prevent deadlock

public class DeadLockFixed {

/\*\*

\* Both method are now requesting lock in same order, first Integer and then String.

\* You could have also done reverse e.g. first String and then Integer,

\* both will solve the problem, as long as both method are requesting lock

\* in consistent order.

\*/

public void method1() {

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

synchronized (String.class) {

System.out.println("Aquired lock on String.class object");

}

}

}

public void method2() {

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

synchronized (String.class) {

System.out.println("Aquired lock on String.class object");

}

}

}

}

Now there would not be any deadlock because both methods are accessing lock on Integer and String class literal in same order. So, if thread A acquires lock on Integer object , thread B will not proceed until thread A releases Integer lock, same way thread A will not be blocked even if thread B holds String lock because now thread B will not expect thread A to release Integer lock to proceed further.

1. What are the states of a thread ?

<https://beginnersbook.com/2013/03/multithreading-in-java/>

public static enum **Thread.State**

extends [Enum](https://docs.oracle.com/javase/7/docs/api/java/lang/Enum.html)<[Thread.State](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html)>

A thread state. A thread can be in one of the following states:

* [NEW](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#NEW)  
  A thread that has not yet started is in this state.
* [RUNNABLE](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#RUNNABLE)  
  A thread executing in the Java virtual machine is in this state.
* [BLOCKED](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#BLOCKED)  
  A thread that is blocked waiting for a monitor lock is in this state.
* [WAITING](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#WAITING)  
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
* [TIMED\_WAITING](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#TIMED_WAITING)  
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
* [TERMINATED](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#TERMINATED)  
  A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.

1. What is thread pool and how to create it ?

<http://programmers.stackexchange.com/questions/173575/what-is-a-thread-pool>

A thread pool is a group of pre-instantiated, idle threads which stand ready to be given work. These are preferred over instantiating new threads for each task when there is a large number of short tasks to be done rather than a small number of long ones. This prevents having to incur the overhead of creating a thread a large number of times.

Implementation will vary by environment, but in simplified terms, you need the following(also check the manually implementation as below example):

* A way to create threads and hold them in an idle state. This can be accomplished by having each thread wait at a barrier until the pool hands it work. (This could be done with mutexes as well.)
* A container to store the created threads, such as a queue or any other structure that has a way to add a thread to the pool and pull one out.
* A standard interface or abstract class for the threads to use in doing work. This might be an abstract class called Task with an execute() method that does the work and then returns.

When the thread pool is created, it will either instantiate a certain number of threads to make available or create new ones as needed depending on the needs of the implementation.

When the pool is handed a Task, it takes a thread from the container (or waits for one to become available if the container is empty), hands it a Task, and meets the barrier. This causes the idle thread to resume execution, invoking the execute() method of the Task it was given. Once execution is complete, the thread hands itself back to the pool to be put into the container for re-use and then meets its barrier, putting itself to sleep until the cycle repeats.

Manually create thread pool

<http://tutorials.jenkov.com/java-concurrency/thread-pools.html>

Thread pool contain 2 major parts: One is Task Queue which implement as Blocking Queue, used for store tasks, another is a container to store a group of Pool Thread which extends from Thread, used for impelementing consume(dequeue) task.

ThreadPool.java

public class ThreadPool {

private BlockingQueue taskQueue = null; 🡪 Part 1

private List<PoolThread> threads = new ArrayList<PoolThread>(); 🡪 Part 2

private boolean isStopped = false;

Note: Both TaskQueue and PoolThread(s) should contained in constructor of ThreadPool which wiring together as an integration of enqueue & dequeue.

public ThreadPool(int noOfThreads, int maxNoOfTasks){

taskQueue = new BlockingQueue(maxNoOfTasks);

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

threads.add(new PoolThread(taskQueue)); 🡪 BlockingQueue will be created as a shared source which defined as member variable

}

for(PoolThread thread : threads){

thread.start();

}

}

public synchronized void execute(Runnable task) throws Exception{

if(this.isStopped) throw

new IllegalStateException("ThreadPool is stopped");

this.taskQueue.enqueue(task);

} 🡪 Note: PoolThread will used for insert tasks onto Task Queue by execute()

public synchronized void stop(){

this.isStopped = true;

for(PoolThread thread : threads){

thread.doStop();

}

}

}

PoolThread.java

public class PoolThread extends Thread {

private BlockingQueue taskQueue = null;

private boolean isStopped = false;

public PoolThread(BlockingQueue queue){

taskQueue = queue;

}

public void run(){

while(!isStopped()){

try{

Runnable runnable = (Runnable) taskQueue.dequeue();

🡪Note:PoolThread will implement dequeue() method in run() method, when any PoolThread running, it will consume a task from Task Queue.

runnable.run();

} catch(Exception e){

//log or otherwise report exception,

//but keep pool thread alive.

}

}

}

public synchronized void doStop(){

isStopped = true;

this.interrupt(); //break pool thread out of dequeue() call.

}

public synchronized boolean isStopped(){

return isStopped;

}

}

Create with ExecutorService

<http://tutorials.jenkov.com/java-util-concurrent/executorservice.html>

An ExecutorService is thus very similar to a thread pool. In fact, the implementation of ExecutorService present in the java.util.concurrent package is a thread pool implementation.

First create an ExecutorService instance,

ExecutorService executorService = Executors.newFixedThreadPool(10);

Then as it contains different methods,

(1)execute(Runnable)

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously.

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use a Callable for that (explained in the following sections).

(2) submit(Runnable)

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Future object can be used to check if the Runnable as finished executing.

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

(3) submit(Callable)

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.

The Callable's result can be obtained via the Future object returned by the submit(Callable) method.

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

The above code example will output this:

Asynchronous Callable

future.get() = Callable Result

(4) invokeAll()

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.

Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a Future to tell the difference.

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

1. How many different kinds of thread pool ?

<http://stackoverflow.com/questions/17186206/types-of-thread-pools-in-java>

a. Single Thread Executor : A thread pool with only one thread. So all the submitted tasks will be executed sequentially. Method : Executors.newSingleThreadExecutor()

b. Cached Thread Pool : A thread pool that creates as many threads it needs to execute the task in parrallel. The old available threads will be reused for the new tasks. If a thread is not used during 60 seconds, it will be terminated and removed from the pool. Method : Executors.newCachedThreadPool()

c.Fixed Thread Pool : A thread pool with a fixed number of threads. If a thread is not available for the task, the task is put in queue waiting for an other task to ends. Method : Executors.newFixedThreadPool()

d.Scheduled Thread Pool : A thread pool made to schedule future task. Method : Executors.newScheduledThreadPool()

e.Single Thread Scheduled Pool : A thread pool with only one thread to schedule future task. Method : Executors.newSingleThreadScheduledExecutor()

1. What are concurrent collection class ?

Java Collection classes are fail-fast which means that if the Collection will be changed while some thread is traversing over it using iterator, the iterator.next() will throw ConcurrentModificationException.

Concurrent Collection classes support full concurrency of retrievals and adjustable expected concurrency for updates.

1. At least how many threads and stack exist when Java application start ?

Besides GC daemon thread, there is one thread for main method. And one stack corresponding to this main method.

# Inner Class Interview Questions

<https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html>

# (1) Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

**Note:** A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static nested classes are accessed using the enclosing class name:

OuterClass.StaticNestedClass

For example, to create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject =new OuterClass.StaticNestedClass();

(2) Inner Classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

class OuterClass {

...

class InnerClass {

...

}

}

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.

<http://www.programmerinterview.com/index.php/java-questions/java-inner-class-example/>

Two situations to instantiate an inner class,

a. If you want to create an inner class code outside of the outer class code, or inside a static method that is still a member of the outer class, must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

OuterClass.InnerClass innerObject = new OuterClass().new InnerClass();

b. If you want to create an inner class instance from inside the outer class code, then you can instantiate the class using the normal syntax

InnerClass inClass = new InnerClass();

# (3) Why we are using inner class ?

# Compelling reasons for using nested classes include the following:

# • It is a way of logically grouping classes that are only used in one place: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

# • It increases encapsulation: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

# • It can lead to more readable and maintainable code: Nesting small classes within top-level classes places the code closer to where it is used.

1. What is reflection used for access private constructor ?

<http://stackoverflow.com/questions/2599440/how-can-i-access-a-private-constructor-of-a-class>

you would need to get the class, find the constructor(getDeclaredConstructor) which takes a single argument with the lower bound of T (in this case Object), force the constructor to be accessible (using the setAccessible method), and finally invoke it with the desired argument.

Class fooClazz = Class.forName("path.to.package.Foo");

Constructor<Foo> constructor = fooClazz.getDeclaredConstructor(Object.class);

constructor.setAccessible(true);

Foo obj = constructor.newInstance("foo");

System.out.println(obj);

For class : getDeclaredConstructor -> setAccessible -> newInstance

1. What is reflection used for access private method ?

<http://tutorials.jenkov.com/java-reflection/private-fields-and-methods.html>

<http://stackoverflow.com/questions/880365/any-way-to-invoke-a-private-method>

To access a private method you will need to call the Class.getDeclaredMethod(String name, Class[] parameterTypes) or Class.getDeclaredMethods() method. The methods Class.getMethod(String name, Class[] parameterTypes) and Class.getMethods() methods only return public methods, so they won't work. Here is a simple example of a class with a private method, and below that the code to access that method via Java Reflection

PrivateObject privateObject = new PrivateObject("The Private Value");

Method privateStringMethod = PrivateObject.class.

getDeclaredMethod("getPrivateString", null);

privateStringMethod.setAccessible(true);

String returnValue = (String)privateStringMethod.invoke(privateObject, null);

System.out.println("returnValue = " + returnValue);

For method: getDeclaredMethod -> setAccessible -> invoke

# [Class.forName() vs ClassLoader.loadClass() - which to use for dynamic loading?](http://stackoverflow.com/questions/8100376/class-forname-vs-classloader-loadclass-which-to-use-for-dynamic-loading)

# <http://stackoverflow.com/questions/8100376/class-forname-vs-classloader-loadclass-which-to-use-for-dynamic-loading>

As stated in the documentation for [Class.forName(String)](http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html#forName%28java.lang.String%29),

Returns the Class object associated with the class or interface with the given string name. Invoking this method is equivalent to: Class.forName(className, true, currentLoader)

(true here refers to do you want to initialize the class?)

On the other hand, [ClassLoader.loadClass(String)](http://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html#loadClass%28java.lang.String%29):

Invoking this method is equivalent to invoking loadClass(name, false).

(here, the boolean has nothing to do with initialization; but if you check loadClass(String, boolean) documentation, you will see that all it does is load the class, not initialize it).

The first one (Class.forName("SomeClass");) will:

* use the class loader that loaded the class which calls this code
* initialize the class (that is, all static initializers will be run)

The other (ClassLoader.getSystemClassLoader().loadClass("SomeClass");) will:

* use the "system" class loader ([which is overridable](http://stackoverflow.com/q/5380275/632951))
* not initialize the class (say, if you use it to load a JDBC driver, it won't get registered, and you won't be able to use JDBC!)

Suppose you are coding a web application that will be executed on a container such as Tomcat. What Tomcat does is create a class loader for each web application (so that it can unload the webapps later and release memory -- you need a dedicated class loader for this to work!). In this situation, you can see that both calls will yield quite different results!

For more detailed (and authoritative) information on class loading and initialization, check sections[12.2](http://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.2) and [12.4](http://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.4) of the latest (3rd) edition of the Java Language Specification.

# When Class is loaded and when class is initialized ?

# <http://stackoverflow.com/questions/17693828/difference-between-loading-a-class-and-instantiating-it>

When Class is loaded in Java

Class loading is done by ClassLoaders in Java which can be implemented to eagerly load a class as soon as another class references it or lazy load the class until a need of class initialization occurs. If Class is loaded before its actually being used it can sit inside before being initialized. I believe this may vary from JVM to JVM. While its guaranteed by JLS that a class will be loaded when there is a need of static initialization

When a Class is initialized in Java

When a Class is initialized in Java After class loading, initialization of class takes place which means initializing all static members of class. A Class is initialized in Java when :

1) an Instance of class is created using either new() keyword or using reflection using class.forName(), which may throw ClassNotFoundException in Java.

2) an static method of Class is invoked.

3) an static field of Class is assigned.

4) an static field of class is used which is not a constant variable.

5) if Class is a top level class and an assert statement lexically nested within class is executed.

**Java Design Patterns Interview Question**

# What is Singleton?

# <http://www.javaworld.com/article/2073352/core-java/simply-singleton.html>

# Sometimes it's appropriate to have exactly one instance of a class: window managers, print spoolers, and filesystems are prototypical examples. Typically, those types of objects—known as singletons—are accessed by disparate objects throughout a software system, and therefore require a global point of access. Of course, just when you're certain you will never need more than one instance, it's a good bet you'll change your mind.

# The Singleton design pattern addresses all of these concerns. With the Singleton design pattern you can:

# Ensure that only one instance of a class is created

# Provide a global point of access to the object

# Allow multiple instances in the future without affecting a singleton class's clients

# Two implementation ways:

# <http://stackoverflow.com/questions/70689/what-is-an-efficient-way-to-implement-a-singleton-pattern-in-java>

# (1)The normal way

public final class Foo {

private static final Foo INSTANCE = new Foo(); 🡪Hold only instance

private Foo() { 🡪 Private constructor to prevent user create their own instance, the private constructor will only be called when class is first used

if (INSTANCE != null) {

throw new IllegalStateException("Already instantiated");

}

}

public static Foo getInstance() {

return INSTANCE; 🡪 Return the only instance

}

}

Let's go over the code. First, you want the class to be final. In this case, I've used the final keyword to let the users know it is final. Then you need to make the constructor private to prevent users to create their own Foo. Throwing an exception from the constructor prevents users to use reflection to create a second Foo. Then you create a private static final Foo field to hold the only instance, and a public static Foo getInstance() method to return it. The Java specification makes sure that the constructor is only called when the class is first used

# (2)The enum(enumerate) way

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

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

INSTANCE;

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

*// Perform operation here*

}

}

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

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

# <http://en.wikipedia.org/wiki/Singleton_pattern>

Singleton classes represent objects for which only one single instance should exist. (e.g. An enum type to handle all requests to and from server. Used "Singleton pattern" for this enum class.)

1. When to use Singleton ?

<http://stackoverflow.com/questions/228164/on-design-patterns-when-to-use-the-singleton>

A Singleton candidate must satisfy three requirements:

(1)controls concurrent access to a shared resource.

(2)access to the resource will be requested from multiple, disparate parts of the system.

(3)there can be only one object.

If your proposed Singleton has only one or two of these requirements, a redesign is almost always the correct option.

For example, a printer spooler is unlikely to be called from more than one place (the Print menu), so you can use mutexes to solve the concurrent access problem.

You use a singleton when you need to manage a shared resource. A simple logger is the most obvious example of a possibly-valid Singleton, but this can change with more complex logging schemes. Another examples as a printer spooler or a database connection or a file manager, your application should only have a single instance of the spooler in order to avoid conflicting request for the same resource

1. What is Factory Pattern and when to use?

<http://stackoverflow.com/questions/69849/factory-pattern-when-to-use-factory-methods>

I like thinking about design pattens in terms of my classes being 'people,' and the patterns are the ways that the people talk to each other.

So, to me the factory pattern is like a hiring agency. You've got someone that will need a variable number of workers. This person may know some info they need in the people they hire, but that's it. So, when they need a new employee, they call the hiring agency and tell them what they need. Now, to actually *hire* someone, you need to know a lot of stuff - benefits, eligibility verification, etc. But the person hiring doesn't need to know any of this - the hiring agency handles all of that.

In the same way, using a Factory allows the consumer to create new objects without having to know the details of how they're created, or what their dependencies are - they only have to give the information they actually want.

public interface IThingFactory

{

Thing GetThing(string theString); 🡪 String will come from consumer of ThingFactory

}

public class ThingFactory : IThingFactory

{

public Thing GetThing(string theString)

{

return new Thing(theString, firstDependency, secondDependency);

}

}

So, now the consumer of the ThingFactory can get a Thing, without having to know about the dependencies of the Thing, except for the string data that comes from the consumer.

# What is SDLC ?

# <http://www.tutorialspoint.com/sdlc/sdlc_overview.htm>

# Stages of SDLC

# Stage 1: Planning and Requirement Analysis

# Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational, and technical areas.

# Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

# Stage 2: Defining Requirements

# Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through .SRS. . Software Requirement Specification document which consists of all the product requirements to be designed and developed during the project life cycle.

# Stage 3: Designing the product architecture

# SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

# This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity , budget and time constraints , the best design approach is selected for the product.

# A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

# Stage 4: Building or Developing the Product

# In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

# Developers have to follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers etc are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java, and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

# Stage 5: Testing the Product

# This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However this stage refers to the testing only stage of the product where products defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

# Stage 6: Deployment in the Market and Maintenance

# Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometime product deployment happens in stages as per the organizations. business strategy. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

# Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

# SDLC Models

# There are various software development life cycle models defined and designed which are followed during software development process. These models are also referred as "Software Development Process Models". Each process model follows a Series of steps unique to its type, in order to ensure success in process of software development.

# Following are the most important and popular SDLC models followed in the industry:

# Waterfall Model Iterative Model Spiral Model V-Model Big Bang Model

# The other related methodologies are Agile Model, RAD Model, Rapid Application Development and Prototyping Models.

# What’s new in JDK 1.8 ?

# <https://www.javacodegeeks.com/2014/05/java-8-features-tutorial.html#nashorn_engine: jjs>

# <https://www.javacodegeeks.com/2014/03/8-new-features-for-java-8.html>

# What is Java Annotation ?

# <http://tutorials.jenkov.com/java/annotations.html>

# <http://www.buggybread.com/2014/04/java-interview-questions-and-answers-on.html>

# Java annotations are used to provide meta data for your Java code. Being meta data, Java annotations do not directly affect the execution of your code, although some types of annotations can actually be used for that purpose.

# Which is the Parent Class of Annotation class? 🡪 Object

# What is the package name for Annotation class? 🡪 java.text

# Java Annotation Purposes

# Java annotations are typically used for the following purposes (Explain annotation)?

# Annotations a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate. Annotations have a number of uses, among them:

# • Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.

# • Compile-time and deployment-time processing — Software tools can process annotation information to generate code, XML files, and so forth.

# • Runtime processing — Some annotations are available to be examined at runtime.Java has 3 built-in annotations that you can use to give instructions to the Java compiler.

# Java annotations can be be used at build-time, when you build your software project. The build process includes generating source code, compiling the source, generating XML files (e.g. deployment descriptors), packaging the compiled code and files into a JAR file etc. Building the software is typically done by an automatic build tool like Apache Ant or Apache Maven. Build tools may scan your Java code for specific annotations and generate source code or other files based on these annotations.

# Normally, Java annotations are not present in your Java code after compilation. It is possible, however, to define your own annotations that are available at runtime. These annotations can then be accessed via Java Reflection, and used to give instructions to your program, or some third party API.

# Annotation Elements

# A Java annotation can have elements for which you can set values. An element is like an attribute or parameter. Here is an example of a Java annotation with an element:

# @Entity(tableName = "vehicles")

# The annotation in this example contains a single element named tableName, with the value set to vehicles. Elements are enclosed inside the parentheses after the annotation name. Annotations without elements do not need the parentheses.

# An annotation can contain multiple elements. Here is a multiple element Java annotation example:

# @Entity(tableName = "vehicles", primaryKey = "id")

# In case an annotation contains just a single element, it is convention to name that element value, like this:

# @InsertNew(value = "yes")

# When an annotation just contains a single element named value, you can leave out the element name, and just provide the value. Here is an example of an annotation element with only the value provided:

# @InsertNew("yes")

# Built-in Java Annotations ( What are few of the Annotations pre-defined by Java ?)

# Java comes with three built-in annotations which are used to give the Java compiler instructions. These annotations are:

# @Deprecated

# The @Deprecated annotation is used to mark a class, method or field as deprecated, meaning it should no longer be used. If your code uses deprecated classes, methods or fields, the compiler will give you a warning. Here is @Deprecated Java annotation example:

# @Deprecated

# public class MyComponent {

# }

# The use of the @Deprecated Java annotation above the class declaration marks the class as deprecated.

# You can also use the @Deprecated annotation above method and field declarations, to mark the method or field as deprecated.

# @Override

# The @Override Java annotation is used above methods that override methods in a superclass. If the method does not match a method in the superclass, the compiler will give you an error.

# The @Override annotation is not necessary in order to override a method in a superclass. It is a good idea to use it still, though. In case someone changed the name of the overridden method in the superclass, your subclass method would no longer override it. Without the @Override annotation you would not find out. With the @Override annotation the compiler would tell you that the method in the subclass is not overriding any method in the superclass.

# Here is an @Override Java annotation example:

# public class MySuperClass {

# public void doTheThing() {

# System.out.println("Do the thing");

# }

# }

# public class MySubClass extends MySuperClass{

# @Override

# public void doTheThing() {

# System.out.println("Do it differently");

# }

# }

# @SuppressWarnings

# The @SuppressWarnings annotation makes the compiler suppress warnings for a given method. For instance, if a method calls a deprecated method, or makes an insecure type cast, the compiler may generate a warning. You can suppress these warnings by annotating the method containing the code with the @SuppressWarnings annotation.

# Here is a @SuppressWarnings Java annotation example:

# @SuppressWarnings

# public void methodWithWarning() {

# }

# @FunctionalInterface

# Introduced in Java SE 8, indicates that the type declaration is intended to be a functional interface, as defined by the Java Language Specification.

# A Function Interface has only one abstract method and many default methods. Compiler generates an error if the interface specified with the annotation doesn't abide by the specifications for functional interface.

# Creating Your Own Annotations

# It is possible to create your own (custom) Java annotations. Annotations are defined in their own file, just like a Java class or interface. Here is custom Java annotation example:

# @interface MyAnnotation {

# String value();

# String name();

# int age();

# String[] newNames();

# }

# This example defines an annotation called MyAnnotation which has four elements. Notice the @interface keyword. This signals to the Java compiler that this is a Java annotation definition.

# Notice that each element is defined similarly to a method definition in an interface. It has a data type and a name. You can use all primitive data types as element data types. You can also use arrays as data type. You cannot use complex objects as data type.

# To use the above annotation, you could use code like this:

# @MyAnnotation(

# value="123",

# name="Jakob",

# age=37,

# newNames={"Jenkov", "Peterson"}

# )

# public class MyClass {

# }

# As you can see, I have to specify values for all elements of the MyAnnotation annotation.

# Element Default Values

# You can specify default values for an element. That way the element becomes optional and can be left out. Here is an example of how the annotation definition looks with a default value for an element:

# @interface MyAnnotation {

# String value() default "";

# String name();

# int age();

# String[] newNames();

# }

# The value element can now be left out when using the annotation. If you leave it out, it will be considered as if you had used the default value for the value element. Here is an example of an annotation with an element value left out, so that element is set to the default value:

# @MyAnnotation(

# name="Jakob",

# age=37,

# newNames={"Jenkov", "Peterson"}

# )

# public class MyClass {

# }

# Notice that the value element is no longer present.

# @Retention

# You can specify for your custom annotation if it should be available at runtime, for inspection via reflection. You do so by annotating your annotation definition with the @Retention annotation. Here is how that is done:

# import java.lang.annotation.Retention;

# import java.lang.annotation.RetentionPolicy;

# @Retention(RetentionPolicy.RUNTIME)

# @interface MyAnnotation {

# String value() default "";

# }

# Notice the @Retention annotation added above the MyAnnotation definition:

# @Retention(RetentionPolicy.RUNTIME)

# This is what signals to the Java compiler and JVM that the annotation should be available via reflection at runtime

# <http://tutorials.jenkov.com/java-reflection/annotations.html>

# Annotations are a kind of comment or meta data you can insert in your Java code. These annotations can then be processed at compile time by pre-compiler tools, or at runtime via Java Reflection. Here is an example of class annotation:

# @MyAnnotation(name="someName", value = "Hello World")

# public class TheClass {

# }

# The class TheClass has the annotation @MyAnnotation written ontop. Annotations are defined like interfaces. Here is the MyAnnotation definition:

# @Retention(RetentionPolicy.RUNTIME)

# @Target(ElementType.TYPE)

# public @interface MyAnnotation {

# public String name();

# public String value();

# }

# The @ in front of the interface marks it as an annotation. Once you have defined the annotation you can use it in your code, as shown in the earlier examples.

# The two directives in the annotation definition, @Retention(RetentionPolicy.RUNTIME) and @Target(ElementType.TYPE), specifies how the annotation is to be used.

# @Retention(RetentionPolicy.RUNTIME) means that the annotation can be accessed via reflection at runtime. If you do not set this directive, the annotation will not be preserved at runtime, and thus not available via reflection.

# @Target(ElementType.TYPE) means that the annotation can only be used ontop of types (classes and interfaces typically). You can also specify METHOD or FIELD, or you can leave the target out alltogether so the annotation can be used for both classes, methods and fields.

# (a)Class Annotations 🡪 class.getAnnottions()

# You can access the annotations of a class, method or field at runtime. Here is an example that accesses the class annotations:

# Class aClass = TheClass.class;

# Annotation[] annotations = aClass.getAnnotations();

# for(Annotation annotation : annotations){

# if(annotation instanceof MyAnnotation){

# MyAnnotation myAnnotation = (MyAnnotation) annotation;

# System.out.println("name: " + myAnnotation.name());

# System.out.println("value: " + myAnnotation.value()); }

# }

# (b)Method Annotations 🡪 method.getDeclaredAnnotations()

# Here is an example of a method with annotations:

# public class TheClass {

# @MyAnnotation(name="someName", value = "Hello World")

# public void doSomething(){}

# }

# You can access method annotations like this:

# Method method = ... //obtain method object

# Annotation[] annotations = method.getDeclaredAnnotations();

# for(Annotation annotation : annotations){

# if(annotation instanceof MyAnnotation){

# MyAnnotation myAnnotation = (MyAnnotation) annotation;

# System.out.println("name: " + myAnnotation.name());

# System.out.println("value: " + myAnnotation.value()); }

# }

# (c)Parameter Annotations

# It is possible to add annotations to method parameter declarations too. Here is how that looks:

# public class TheClass {

# public static void doSomethingElse(

# @MyAnnotation(name="aName", value="aValue") String parameter){

# }

# }

# You can access parameter annotations from the Method object like this:

# Method method = ... //obtain method object

# Annotation[][] parameterAnnotations = method.getParameterAnnotations();

# Class[] parameterTypes = method.getParameterTypes();

# int i=0;

# for(Annotation[] annotations : parameterAnnotations){

# Class parameterType = parameterTypes[i++];

# for(Annotation annotation : annotations){

# if(annotation instanceof MyAnnotation){

# MyAnnotation myAnnotation = (MyAnnotation) annotation;

# System.out.println("param: " + parameterType.getName());

# System.out.println("name : " + myAnnotation.name());

# System.out.println("value: " + myAnnotation.value());

# }

# }

# }

# Notice how the Method.getParameterAnnotations() method returns a two-dimensional Annotation array, containing an array of annotations for each method parameter.

# (d)Field Annotations 🡪 Field.getDeclaredAnnotations()

# Here is an example of a field with annotations:

# public class TheClass {

# @MyAnnotation(name="someName", value = "Hello World")

# public String myField = null;

# }

# You can access field annotations like this:

# Field field = ... //obtain field object

# Annotation[] annotations = field.getDeclaredAnnotations();

# for(Annotation annotation : annotations){

# if(annotation instanceof MyAnnotation){

# MyAnnotation myAnnotation = (MyAnnotation) annotation;

# System.out.println("name: " + myAnnotation.name());

# System.out.println("value: " + myAnnotation.value());

# }

# }

# What are meta Annotations ?

# @Retention specifies how the marked annotation is stored:

# @Documented indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool. (By default, annotations are not included in Javadoc.)

# @Target marks another annotation to restrict what kind of Java elements the annotation can be applied to.

# @Inherited indicates that the annotation type can be inherited from the super class. (This is not true by default.) When the user queries the annotation type and the class has no annotation for this type, the class' superclass is queried for the annotation type. This annotation applies only to class declarations.

# @Repeatable introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use. For more information, see Repeating Annotations.

# What is Java NIO ?

# <http://tutorials.jenkov.com/java-nio/index.html>

# <http://www.ibm.com/developerworks/java/tutorials/j-nio/j-nio.html>

# Java NIO: Channels and Buffers

# Channel read data into buffer = Reading from a channel is simple: we simply create a buffer and then ask a channel to read data into it = channel.read(buffer)

# Buffer write data into channel = Writing is also fairly simply: we create a buffer, fill it with data, and then ask a channel to write from it = channel.write(buffer)

# (a)Reading from a file

# For our first exercise, we'll read some data from a file. If we were using original I/O, we would simply create a FileInputStream and read from that. In NIO, however, things work a little differently: we first get a Channel object from the FileInputStream, and then use that channel to read the data.

# Any time you perform a read operation in an NIO system, you are reading from a channel, but you don't read directly from a channel. Since all data ultimately resides in the buffer, you read from a channel into a buffer.

# So reading from a file involves three steps: (1) getting the Channel from FileInputStream; (2) creating the Buffer; and (3) reading from the Channel into the Buffer.

# Our first step is to get a channel. We get the channel from the FileInputStream:

# FileInputStream fin = new FileInputStream( "readandshow.txt" );

# FileChannel fc = fin.getChannel();

# The next step is to create a buffer:

# ByteBuffer buffer = ByteBuffer.allocate( 1024 );

# And, finally, we need to read from the channel into the buffer, as shown here:

# fc.read( buffer );

# (b)Writing to a file

# We start by getting a channel from a FileOutputStream:

# FileOutputStream fout = new FileOutputStream( "writesomebytes.txt" );

# FileChannel fc = fout.getChannel();

# Our next step is to create a buffer and put some data in it -- in this case, the data will be taken from an array called message which contains the ASCII bytes for the string "Some bytes." (The buffer.flip() and buffer.put() calls will be explained later in the tutorial.)

# ByteBuffer buffer = ByteBuffer.allocate( 1024 );

# for (int i=0; i<message.length; ++i) {

# buffer.put( message[i] );

# }

# buffer.flip();

# Our final step is to write to the buffer:

# fc.write( buffer );

# Checking the status

# Our next step is to check to see when we're done copying. We're done when there's no more data, and we can tell this when the read() method returns -1, as shown below:

# int r = fcin.read( buffer );

# if (r==-1) {

# break;

# }

# Resetting the buffer

# And, finally, we call the clear() method before we read into a buffer from the input channel. Likewise, we call the flip() method before we write a buffer to the output channel, as shown below:

# buffer.clear();

# int r = fcin.read( buffer );

# if (r==-1) {

# break;

# }

# buffer.flip();

# fcout.write( buffer );

# The clear() method resets the buffer, making it ready to have data read into it. The flip() method prepares the buffer to have the newly-read data written to another channel.

# Buffer Write = Channel Read 🡪 Channel read data into buffer

# Buffer Read = Channel Write 🡪 Buffer write data into channel

# Java NIO: Buffer capacity, position and limit in write and read mode.

# Buffer internals overview

# In this section, we'll look at two important components of buffers in NIO: state variables and accessor methods.

# State variables are key to the "internal accounting system" mentioned in the previous section. With each read/write operation, the buffer's state changes. By recording and tracking those changes, a buffer is able to internally manage its own resources.

# When you read data from a channel, the data is placed in a buffer. In some cases, you can write this buffer directly to another channel, but often, you'll want to look at the data itself. This is accomplished using the accessor method get(). Likewise, when you want to put raw data in a buffer, you use the accessor method put().

# In this section, you'll learn about state variables and accessor methods in NIO. Each component will be described, and then you'll have the opportunity to see it in action. While NIO's internal accounting system might seem complicated at first, you'll quickly see that most of the real work is done for you. The bookkeeping you're probably accustomed to coding by hand -- using byte arrays and index variables -- is handled internally in NIO.

# State variables

# Three values can be used to specify the state of a buffer at any given moment in time: Position/ limit/ capacity

# Together, these three variables track the state of the buffer and the data it contains. We'll examine each one in detail, and also see how they fit into a typical read/write (input/output) process. For the sake of the example, we'll assume that we are copying data from an input channel to an output channel.

# Position

# You will recall that a buffer is really just a glorified array. When you read from a channel, you put the data that you read into an underlying array. The position variable keeps track of how much data you have written. More precisely, it specifies into which array element the next byte will go. Thus, if you've read 3 bytes from a channel into a buffer, that buffer's position will be set to 3, referring to the fourth element of the array.

# Likewise, when you are writing to a channel, you get the data from a buffer. The position value keeps track of how much you have gotten from the buffer. More precisely, it specifies from which array element the next byte will come. Thus, if you've written 5 bytes to a channel from a buffer, that buffer's position will be set to 5, referring to the sixth element of the array.

# Limit

# The limit variable specifies how much data there is left to get (in the case of writing from a buffer into a channel), or how much room there is left to put data into (in the case of reading from a channel into a buffer).

# The position is always less than, or equal to, the limit.

# Capacity

# The capacity of a buffer specifies the maximum amount of data that can be stored therein. In effect, it specifies the size of the underlying array -- or, at least, the amount of the underlying array that we are permitted to use.

# The limit can never be larger than the capacity.

# Observing the variables

# We'll start with a newly created buffer. For the sake of the example, let's assume that our buffer has a total capacity of eight bytes. The Buffer 's state is shown here:

# React operations

# Buffer's state

# Buffer's state

# Recall that the limit can never be larger than the capacity, and in this case both values are set to 8. We show this by pointing them off the end of the array (which is where slot 8 would be if there were a slot 8):

# React operations

# Buffer limit

# Buffer limit

# The position is set to 0. If we read some data into the buffer, the next byte read will go into slot 0. If we write from the buffer, the next byte taken from the buffer will be taken from slot 0. The position setting is shown here:

# React operations

# Position setting

# Position setting

# Because the capacity is not going to change, we can omit it from the discussion that follows.

# The first read

# Now we are ready to begin read/write operations on our newly created buffer. We start by reading some data from our input channel into the buffer. The first read gets three bytes. These are put into the array starting at the position, which was set to 0. After this read, the position is increased to 3, as shown here:

# React operations

# Position increased to 3

# Position increased to 3

# The limit is unchanged.

# The second read

# For our second read, we read two more bytes from the input channel into our buffer. The two bytes are stored at the location pointed to by position; position is thus increased by two:

# React operations

# Position increased by 2

# Position increased by 2

# The limit is unchanged.

# The flip

# Now we are ready to write our data to an output channel. Before we can do this, we must call the flip() method. This method does two crucial things:

# (1)It sets the limit to the current position.

# (2)It sets the position to 0.

# The figure on the section shows our buffer before the flip. Here is the buffer after the flip:

# React operations

# Buffer after the flip

# Buffer after the flip

# We are now ready to begin writing data to a channel from the buffer. The position has been set to 0, which means the next byte we get will be the first one. And the limit has been set to the old position, which means that it just includes all the bytes we read before, and no more.

# The first write

# In our first write, we take four bytes from the buffer and write them to our output channel. This advances the position to 4, and leaves the limit unchanged, as shown here:

# React operations

# Code limit unchanged

# Code limit unchanged

# The second write

# We only have one byte left to write. The limit was set to 5 when we did our flip(), and the position cannot go past the limit. So the last write takes one byte from our buffer and writes it to the output channel. This advances the position to 5, and leaves the limit unchanged, as shown here:

# React operations

# Code limit unchanged

# Code limit unchanged

# The clear

# Our final step is to call the buffer's clear() method. This method resets the buffer in preparation for receiving more bytes. Clear does two crucial things:

# It sets the limit to match the capacity.

# It sets the position to 0.

# This figure shows the state of the buffer after clear() has been called:

# React operations

# Buffer after clear has been called

# Buffer after clear has been called

# The buffer is now ready to receive fresh data.

# Java file IO vs NIO ?

# <http://www.ibm.com/developerworks/java/tutorials/j-nio/j-nio.html>

# NIO was created to allow Java programmers to implement high-speed I/O without having to write custom native code. NIO moves the most time-consuming I/O activities (namely, filling and draining buffers) back into the operating system, thus allowing for a great increase in speed.

# Streams vs blocks

# The most important distinction between the original I/O library (found in java.io.\* ) and NIO has to do with how data is packaged and transmitted. As previously mentioned, original I/O deals with data in streams, whereas NIO deals with data in blocks.

# A stream-oriented I/O system deals with data one byte at a time. An input stream produces one byte of data, and an output stream consumes one byte of data. It is very easy to create filters for streamed data. It is also relatively simply to chain several filters together so that each one does its part in what amounts to a single, sophisticated processing mechanism. On the flip side, stream-oriented I/O is often rather slow.

# A block-oriented I/O system deals with data in blocks. Each operation produces or consumes a block of data in one step. Processing data by the block can be much faster than processing it by the (streamed) byte. But block-oriented I/O lacks some of the elegance and simplicity of stream-oriented I/O.

# What is a buffer?

# A Buffer is an object, which holds some data, that is to be written to or that has just been read from. The addition of the Buffer object in NIO marks one of the most significant differences between the new library and original I/O. In stream-oriented I/O, you wrote data directly to, and read data directly from, Stream objects.

# In the NIO library, all data is handled with buffers. When data is read, it is read directly into a buffer. When data is written, it is written into a buffer. Anytime you access data in NIO, you are pulling it out of the buffer.

# A buffer is essentially an array. Generally, it is an array of bytes, but other kinds of arrays can be used. But a buffer is more than just an array. A buffer provides structured access to data and also keeps track of the system's read/write processes.

# <http://javapapers.com/java/java-nio-file-read-write-with-channels/>

# Main difference between NIO and IO is that NIO provides asynchronous, non blocking IO, which is critical to write faster and scalable networking systems. While most of utility from IO classes are blocking and slow. NIO take advantage of asynchronous system calls in UNIX systems such as select() system call for network sockets. Using select(), an application can monitor several resources at the same time and can also poll for network activity without blocking. The select() system call identifies if data is pending or not, then read() or write() may be used knowing that they will complete immediately.

# Buffered File Read Write

# We create an instance of BufferedReader and read line by line till end of file

# Read File

# Charset charset = Charset.forName("US-ASCII");

# Path filePath = FileSystems.getDefault().getPath(".", "temp.txt");

# BufferedReader bufferedReader = Files.newBufferedReader(filePath, charset);

# String line = null;

# while ((line = bufferedReader.readLine()) != null) {

# System.out.println(line);

# }

# Write File

# Charset charset = Charset.forName("US-ASCII");

# Path filePath = FileSystems.getDefault().getPath(".", "tempCopy1.txt");

# BufferedWriter bufferedWriter = Files.newBufferedWriter(filePath, charset);

# System.out.println(string);

# bufferedWriter.write(string, 0, string.length());

# Stream File Read Write

# Read File

# Path filePath = FileSystems.getDefault().getPath(".", "temp.txt");

# InputStream inputStream = Files.newInputStream(filePath);

# BufferedReader bufferedReader = new BufferedReader(

# new InputStreamReader(inputStream));

# String line = null;

# while ((line = bufferedReader.readLine()) != null) {

# System.out.println(line);

# }

# Write File

# Path filePath = FileSystems.getDefault().getPath(".", "tempCopy2.txt");

# OutputStream outputStream = new BufferedOutputStream(

# Files.newOutputStream(filePath, StandardOpenOption.CREATE,

# StandardOpenOption.APPEND));

# outputStream.write(byteArray, 0, byteArray.length);

# File Read with ByteChannel

# We create an instance for byte channel using Files NIO utility class. Then we allocate a ByteBuffer using which the data will be read. rewind is used to read again the data it already contains. rewind sets the buffer position to zero. flip is used to prepare a buffer for get operation and makes it ready.

# public static void byteChannelRead() throws IOException {

# Path filePath = FileSystems.getDefault().getPath(".", "temp.txt");

# SeekableByteChannel byteChannel = Files.newByteChannel(filePath);

# ByteBuffer byteBuffer = ByteBuffer.allocate(10);

# Charset charset = Charset.forName("US-ASCII");

# while (byteChannel.read(byteBuffer) > 0) {

# byteBuffer.rewind();

# System.out.print(charset.decode(byteBuffer));

# byteBuffer.flip();

# }

# }

# File Write with ByteChannel

# We set the options mode like create/append using StandardOpenOption and then write the buffer using the channel.

# public static void byteChannelWrite(ByteBuffer byteBuffer)

# throws IOException {

# Set options = new HashSet();

# options.add(StandardOpenOption.CREATE);

# options.add(StandardOpenOption.APPEND);

# Path file = Paths.get("./byByteChannel.txt");

# SeekableByteChannel byteChannel = Files.newByteChannel(file, options);

# byteChannel.write(byteBuffer);

# }

# File Read with FileChannel

# ByteChannel and FileChannel usage are almost similar, the way we create the instances are little bit different

# public static void fileChannelRead() throws IOException {

# RandomAccessFile randomAccessFile = new RandomAccessFile("./temp.txt","rw");

# FileChannel fileChannel = randomAccessFile.getChannel();

# ByteBuffer byteBuffer = ByteBuffer.allocate(512);

# Charset charset = Charset.forName("US-ASCII");

# while (fileChannel.read(byteBuffer) > 0) {

# byteBuffer.rewind();

# System.out.print(charset.decode(byteBuffer));

# byteBuffer.flip();

# }

# fileChannel.close();

# randomAccessFile.close();

# }

# File Write with FileChannel

# public static void fileChannelWrite(ByteBuffer byteBuffer) throws IOException {

# Set options = new HashSet();

# options.add(StandardOpenOption.CREATE);

# options.add(StandardOpenOption.APPEND);

# Path path = Paths.get("./byFileChannel.txt");

# FileChannel fileChannel = FileChannel.open(path, options);

# fileChannel.write(byteBuffer);

# fileChannel.close();

# }

# What is JSON and why should I use that ?

[**http://stackoverflow.com/questions/383692/what-is-json-and-why-would-i-use-it**](http://stackoverflow.com/questions/383692/what-is-json-and-why-would-i-use-it)

An example of where this is used is web services responses. In the 'old' days, web services used XML as their primary data format for transmitting back data, but since JSON appeared (The JSON format is specified in RFC 4627 by Douglas Crockford), it has been the preferred format because it is much more lightweight

JSON is built on two structures:

(a)A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.

(b)An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence

Here is an example of JSON data:

{

"firstName": "John",

"lastName": "Smith",

"address": {

"streetAddress": "21 2nd Street",

"city": "New York",

"state": "NY",

"postalCode": 10021

},

"phoneNumbers": [

"212 555-1234",

"646 555-4567"

]

}

<http://stackoverflow.com/questions/1938428/marshall-unmarshall-a-json-to-a-java-class-using-jaxb>

Marshalling to XML is easy, but it took me a while to figure out how to marshall to JSON. Pretty simple after you find the solution though. JSONJAXBContext is from

<https://jersey.java.net/apidocs/1.1.1-ea/jersey/com/sun/jersey/api/json/JSONJAXBContext.html>

public static String marshalToXml( Object o ) throws JAXBException {

StringWriter writer = new StringWriter();

Marshaller marshaller = JAXBContext.newInstance( o.getClass() ).createMarshaller();

marshaller.setProperty( Marshaller.JAXB\_FORMATTED\_OUTPUT, true );

marshaller.marshal( o, writer );

return writer.toString();

}

public static String marshalToJson( Object o ) throws JAXBException {

StringWriter writer = new StringWriter();

JAXBContext context = JSONJAXBContext.newInstance( o.getClass() );

Marshaller m = context.createMarshaller();

JSONMarshaller marshaller = JSONJAXBContext.getJSONMarshaller( m );

marshaller.setProperty( Marshaller.JAXB\_FORMATTED\_OUTPUT, true );

marshaller.marshallToJSON( o, writer );

return writer.toString();

}

In short - JSON is a way of serializing in such a way, that it becomes JavaScript code. When executed (with eval or otherwise), this code creates and returns a JavaScript object which contains the data you serialized. This is available because JavaScript allows the following syntax:

var MyArray = [ 1, 2, 3, 4]; // MyArray is now an array with 4 elements

var MyObject = {

'StringProperty' : 'Value',

'IntProperty' : 12,

'ArrayProperty' : [ 1, 2, 3],

'ObjectProperty' : { 'SubObjectProperty': 'SomeValue' }

}; // MyObject is now an object with property values set.

You can use this for several purposes. For one, it's a comfortable way to pass data from your server backend to your JavaScript code. Thus, this is often used in AJAX.

You can also use it as a standalone serialization mechanism, which is simpler and takes up less space than XML. Many libraries exists that allow you to serialize and deserialize objects in JSON for various programming languages

# Java generic interview questions

# <http://javarevisited.blogspot.com/2012/06/10-interview-questions-on-java-generics.html>

# (1) What is Generics in Java ? What are advantages of using Generics?

# This is one of the first interview questions asked on generics in any Java interview, mostly at beginners and intermediate level. Those who are coming from prior to Java 5 background knows that how inconvenient it was to store object in Collection and then cast it back to correct Type before using it. Generics prevents from those. it provides compile time type-safety and ensures that you only insert correct Type in collection and avoids ClassCastException in runtime.

# (2) How Generics works in Java ? What is type erasure ?

# This is one of better interview question in Generics. Generics is implemented using Type erasure, compiler erases all type related information during compile time and no type related information is available during runtime. For example List<String> is represented by only List at runtime. This was done to ensure binary compatibility with the libraries which were developed prior to Java 5. You don't have access to Type argument at runtime and Generic type is translated to Raw type by compiler during runtime.

# (3) What is Bounded and Unbounded wildcards in Generics ?

# There are two kinds of Bounded wildcards <? extends T> which impose an upper bound by ensuring that type must be sub class of T and <? super T> where its imposing lower bound by ensuring Type must be super class of T. This Generic Type must be instantiated with Type within bound otherwise it will result in compilation error. On the other hand <?> represent and unbounded type because <?> can be replace with any Type

# (4) Can you pass List<String> to a method which accepts List<Object> ?

# This generic interview question in Java may look confusing to any one who is not very familiar with Generics as in fist glance it looks like String is object so List<String> can be used where List<Object> is required but this is not true. It will result in compilation error. It does make sense if you go one step further because List<Object> can store any any thing including String, Integer etc but List<String> can only store Strings.

# (5) Can we use Generics with Array?

# This was probably most simple generics interview question in Java, if you know the fact that Array doesn't support Generics and that's why Joshua Bloch suggested in Effective Java to prefer List over Array because List can provide compile time type-safety over Array. <http://stackoverflow.com/questions/2992786/why-doesnt-java-allow-for-the-creation-of-generic-arrays>

# Arrays are reified - they retain type information at runtime.

# Generics are a compile-time construct - the type information is lost at runtime. This was a deliberate decision to allow backward compatibility with pre-generics Java bytecode. The consequence is that you cannot create an array of generic type, because by the time the VM wants to create the array, it won't know what type to use.

# (6) Difference between List<Object> and List raw type ?

# a.Main difference between raw type and parametrized type List<Object> is that, compiler will not check type-safety of raw type at compile time but it will do that for parametrized type and by using Object as Type it inform compiler that it can hold any Type of Object e.g. String or Integer. This Java Generics interview question is based on correct understanding of raw type in Generics.

# b.Any way second difference between them is that you can pass any parametrized type to raw type List but you can not pass List<String> to any method which accept List<Object> it will result in compilation error.

# (7) Difference between List<Object> and List<?>

# This generics interview question may look related to previous interview questions but completely different. List<?> is List of unknown type while List<Object> is essentially List of any Type. You can assign List<String>, List<Integer> to List<?> but you can not assign List<String> to List<Object>.

# (8) Difference between List<? extends Base> vs List<Base> <http://stackoverflow.com/questions/9488445/list-extends-base-vs-listbase>

List<Base> can contain a mixture of different things that all derive from Base. List<? extend Base>contains homogeneous items (in the sense that they must all derive from some specific, unknown type that in turn derives from Base).

Put another way, List<? extends Base> is the base class for List<T extends Base>. So you can pass a List<T extends Base> to any method that takes a List<? extends Base>. The same is not true for methods that take a List<Base>.

# Java Serialization Interview questions

# <http://www.fromdev.com/2012/06/15-java-serialization-interview.html>

# Java Serialization Interview Questions and Answers

# What’s the difference between serialization and marshaling ?

# <http://stackoverflow.com/questions/770474/what-is-the-difference-between-serialization-and-marshaling>

# The term "marshal" is considered to be synonymous with "serialize" in the Python standard library[1], but the terms are not synonymous in the Java-related RFC 2713:

# To "marshal" an object means to record its state and codebase(s) in such a way that when the marshalled object is "unmarshalled", a copy of the original object is obtained, possibly by automatically loading the class definitions of the object. You can marshal any object that is serializable or remote. Marshalling is like serialization, except marshalling also records codebases. Marshalling is different from serialization in that marshaling treats remote objects specially. (RFC 2713)

# To "serialize" an object means to convert its state into a byte stream in such a way that the byte stream can be converted back into a copy of the object.

# <https://en.wikipedia.org/wiki/Marshalling_(computer_science)>

# Note: "Codebase" here is used in its Java-specific meaning, to refer to a list of URLs where the object code can be loaded from, rather than in the more general meaning of codebase which refers to source code.

# What do you mean by Serialization in Java?

# Serialization is a mechanism by which you can save or transfer the state of an object by converting it to a byte stream. This can be done in java by implementing Serialiazable interface. Serializable is defined as a marker interface which needs to be implemented for transferring an object over a network or persistence of its state to a file. Since its a marker interface, it does not contain any methods. Implementation of this interface enables the conversion of object into byte stream and thus can be transferred. The object conversion is done by the JVM using its default serialization mechanism.

# What is purpose of Serialization? Why is serialization required ? What is need to serialization ?

Serialization is required for a variety of reasons. It is required to send across the state of an object over a network by means of a socket. One can also store an object’s state in a file. Additionally, manipulation of the state of an object as streams of bytes is required. The core of Java Serialization is the Serializable interface. When Serializable interface is implemented by your class it provides an indication to the compiler that java Serialization mechanism needs to be used to serialize the object.

Some more explicit explaination

<http://stackoverflow.com/questions/2232759/what-is-the-purpose-of-serialization-in-java>

Let's define serialization first, then we can talk about why it's so useful.

Serialization is simply turning an existing object into a byte array. This byte array represents the class of the object, the version of the object, and the internal state (metadata) of the object. This byte array can then be used between JVM's running the same code to transmit/read the object.

Why would we want to do this?

There are several reasons:

1. Communication: If you have two machines that are running the same code, and they need to communicate, an easy way is for one machine to build an object with information that it would like to transmit, and then serialize that object to the other machine. It's not the best method for communication, but it gets the job done.
2. Persistence: If you want to store the state of a particular operation in a database, it can be easily serialized to a byte array, and stored in the database for later retrieval.

🡪 Save the explicit version of object you want into database with serialization

1. Deep Copy: If you need an exact replica of an Object, and don't want to go to the trouble of writing your own specialized clone() class, simply serializing the object to a byte array, and then de-serializing it to another object achieves this goal.
2. Caching: Really just an application of the above, but sometimes an object takes 10 minutes to build, but would only take 10 seconds to de-serialize. So, rather than hold onto the giant object in memory, just cache it out to a file via serialization, and read it in later when it's needed. 🡪 Giant object can be serialized and store in file instead of store in memory
3. Cross JVM Synchronization: Serialization works across different JVMs that may be running on different architectures.

# What is difference between externalizable and serializable interface ?

# Serializable is a marker interface therefore you are not forced to implement any methods, however Externalizable contains two methods readExternal() and writeExternal() which must be implemented.

# Serializable interface provides a inbuilt serialization mechanism to you which can be in-efficient at times. However Externilizable interface is designed to give you greater control over the serialization mechanism. The two methods provide you immense opportunity to enhance the performance of specific object serialization based on application needs.

# Serializable interface provides a default serialization mechanism, on the other hand, Externalizable interface instead of relying on default Java Serialization provides flexibility to control this mechanism.

# When will you use Serializable or Externalizable interface ? and why ?

# Most of the times when you want to do a selective attribute serialization you can use Serializable interface with transient modifier for variables not to be serialized. However, use of Externalizable interface can be really effective in cases when you have to serialize only some dynamically selected attributes of a large object.

# Lets take an example, Some times when you have a big Java object with hundreds of attributes and you want to serialize only a dozen dynamically selected attributes to keep the state of the object you should use Externalizable interface writeExternal method to selectively serialize the chosen attributes.

# In case you have small objects and you know that most or all attributes are required to be serialized then you should be fine with using Serializable interface and use of transient variable as appropriate.

# What the ways to speed up Object serialization ? How to improve Serialization performance ?

# The default Java Serialization mechanism is really useful, however it can have a really bad performance based on your application and business requirements. The serialization process performance heavily depends on the number and size of attributes you are going to serialize for an object. Below are some tips you can use for speeding up the marshaling and un-marshaling of objects during Java serialization process.

# (a) Mark the unwanted or non Serializable attributes as transient. This is a straight forward benefit since your attributes for serialization are clearly marked and can be easily achieved using Serialzable interface itself.

# (b) Save only the state of the object, not the derived attributes. Some times we keep the derived attributes as part of the object however serializing them can be costly. Therefore consider calcualting them during de-serialization process.

# (c) Serialize attributes only with NON-default values. For examples, serializing a int variable with value zero is just going to take extra space however, choosing not to serialize it would save you a lot of performance. This approach can avoid some types of attributes taking unwanted space. This will require use of Externalizable interface since attribute serialization is determined at runtime based on the value of each attribute.

# (d) Use Externalizable interface and implement the readExternal and writeExternal methods to dynamically identify the attributes to be serialized. Some times there can be a custom logic used for serialization of various attributes.

# What is serialVersionUID ? and why we should use that ? How to generate one ?

# The serialVersionUID represents your class version, and you should change it if the current version of your class is not backwards compatible with its earlier versions. Most of the times, we probably do not use serialization directly. In such cases, I would suggest to generate a default serializable uid by clicking the quick fix option in eclipse.

# What would happen if the serialVersionUID of an object is not defined ?

# If you don't define serialVersionUID in your serilizable class, Java compiler will make one by creating a hash code using most of your class attributes and features. When an object gets serialized, this hash code is stamped on the object which is known as the SerialVersionUID of that object. This ID is required for the version control of an object. SerialVersionUID can be specified in the class file also. In case, this ID is not specified by you, then Java compiler will regenerate a SerialVersionUID based on updated class and it will not be possible for the already serialized class to recover when a class field is added or modified. Its recommended that you always declare a serialVersionUID in your Serializable classes (Then this manually defined serialVersionUID can be used for auto check on whether serializable classes been changed or not).

# Does setting the serialVersionUID class field improve Java serialization performance ?

# Declaring an explicit serialVersionUID field in your classes saves some CPU time only the first time the JVM process serializes a given Class. However the gain is not significant, In case when you have not declared the serialVersionUID its value is computed by JVM once and subsequently kept in a soft cache for future use

# What are alternatives to serialization ? If serialization is not used, is it possible to persist or transfer an object using other approach ?

# (a)Saving object state to database, this is most common technique used by most applications. You can use ORM tools (e.g. hibernate) to save the objects in a database and read them from the database.

# (b)Xml based data transfer is another popular mechanism, and a lot of XML based web services use this mechanism to transfer data over network. Also a lot of tools save XML files to persist data/configurations.

# (c)JSON Data Transfer - is recently popular data transfer format. A lot of web services are being developed in JSON due to its small footprint and inherent integration with web browser due to JavaScript format.

# What are transient variables? What role do they play in Serialization process ?

# Also check <https://www.cs.uic.edu/~troy/fall04/cs441/drake/serialization.html>

# <http://javarevisited.blogspot.com/2011/09/transient-keyword-variable-in-java.html>

# The transient keyword in Java is used to indicate that a field should not be serialized. Once the process of de-serialization is carried out, the transient variables do not undergo a change and retain their default value. Marking unwanted fields as transient can help you boost the serialization performance. Below is a simple example where you can see the use of transient keyword.

# class MyVideo implements Serializable

# {

# private Video video;

# private transient Image thumbnailVideo;

# private void generateThumbnail()

# {

# // Generate thumbnail.

# }

# private void readObject(ObjectInputStream inputStream)

# throws IOException, ClassNotFoundException

# {

# inputStream.defaultReadObject();

# generateThumbnail();

# }

# }

# Why does serialization NOT save the value of static class attributes? Why static variables are not serialized?

# The Java variables declared as static are not considered part of the state of an object since they are shared by all instances of that class. Saving static variables with each serialized object would have following problems

# (a)It will make redundant copy of same variable in multiple objects which makes it in-efficient.

# (b)The static variable can be modified by any object and a serialized copy would be stale or not in sync with current value.

# How to Serialize a collection in java? How to serialize a ArrayList, Hashmap or Hashset object in Java?

# All standard implementations of collections List, Set and Map interface already implement java.io.Serializable. All the commonly used collection classes like java.util.ArrayList, java.util.Vector, java.util.Hashmap, java.util.Hashtable, java.util.HashSet, java.util.TreeSet do implement Serializable. This means you do not really need to write anything specific to serialize collection objects.

# However you should keep following things in mind before you serialize a collection object – a.Make sure all the objects added in collection are Serializable. – b.Serializing the collection can be costly therefore make sure you serialize only required data isntead of serializing the whole collection. - In case you are using a custom implementation of Collection interface then you may need to implement serialization for it.

# Is it possible to customize the serialization process? How can we customize the Serialization process?

# Yes, the serialization process can be customized. When an object is serialized, objectOutputStream.writeObject (to save this object) is invoked and when an object is read, ObjectInputStream.readObject () is invoked. What most people do not know is that Java Virtual Machine provides you with an option to define these methods as per your needs. Once this is done, these two methods will be invoked by the JVM instead of the application of the default serialization process. Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

# private void writeObject(java.io.ObjectOutputStream out)

# throws IOException

# private void readObject(java.io.ObjectInputStream in)

# throws IOException, ClassNotFoundException;

# private void readObjectNoData()

# throws ObjectStreamException;

# How can a sub-class of Serializable super class avoid serialization? If serializable interface is implemented by the super class of a class, how can the serialization of the class be avoided?

# In Java, if the super class of a class is implementing Serializable interface, it means that it is already serializable. Since, an interface cannot be unimplemented, it is not possible to make a class non-serializable. However, the serialization of a new class can be avoided. For this, writeObject () and readObject() methods should be implemented in your class so that a Not Serializable Exception can be thrown by these methods. And, this can be done by customizing the Java Serialization process. Below the code that demonstrates it

# class MySubClass extends SomeSerializableSuperClass {

# private void writeObject(java.io.ObjectOutputStream out)

# throws IOException {

# throw new NotSerializableException(“Can not serialize this class”);

# }

# private void readObject(java.io.ObjectInputStream in)

# throws IOException, ClassNotFoundException {

# throw new NotSerializableException(“Can not serialize this class”);

# }

# private void readObjectNoData()

# throws ObjectStreamException; {

# throw new NotSerializableException(“Can not serialize this class”);

# }

# }

# What changes are compatible and incompatible to the mechanism of java Serialization?

# This is one of a difficult and tricky questions and answering this correctly would mean you are an expert in Java Serialization concept. In an already serialized object, the most challenging task is to change the structure of a class when a new field is added or removed. As per the specifications of Java Serialization, addition of any method or field is considered to be a compatible change whereas changing of class hierarchy or non-implementation of Serializable interface is considered to be a non-compatible change. You can go through the Java serialization specification for the extensive list of compatible and non-compatible changes. If a serialized object need to be compatible with an older version, it is necessary that the newer version follows some rules for compatible and incompatible changes. A compatible change to the implementing class is one that can be applied to a new version of the class, which still keeps the object stream compatible with older version of same class.

# Some Simple Examples of compatible changes are:

# (a)Addition of a new field or class will not affect serialization, since any new data in the stream is simply ignored by older versions. the newly added field will be set to its default values when the object of an older version of the class is un marshaled.

# (b)The access modifiers change (like private, public, protected or default) is compatible since they are not reflected in the serialized object stream.

# (c)Changing a transient field to a non-transient field is compatible change since it is similar to adding a field.

# (d)Changing a static field to a non-static field is compatible change since it is also similar to adding a field.

# Some Simple Examples of incompatible changes are:

# (a)Changing implementation from Serializable to Externalizable interface can not be done since this will result in the creation of an incompatible object stream.

# (b)Deleting a existing Serializable fields will cause a problem.

# (c)Changing a non-transient field to a transient field is incompatible change since it is similar to deleting a field.

# (d)Changing a non-static field to a static field is incompatible change since it is also similar to deleting a field.

# (e)Changing the type of a attribute within a class would be incompatible, since this would cause a failure when attempting to read and convert the original field into the new field.

# (f)Changing the package of class is incompatible. Since the fully-qualified class name is written as part of the object byte stream.

# Explain marshall and unmarshall in java

# <http://javaquestionsinterview.blogspot.com/2012/03/java-interview-questions-can-you_19.html>

# Marshalling:

# Marshalling creates an XML document from a content tree

# To marshal a content tree

# Create a JAXBContext object

# Create a Marshaller object (Marshaller marshaller = jaxbContext.createMarshaller();)

# Set required properties using setProperty method of Marshaller marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, new Boolean(true));

# Call the marshal method marshaller.marshal(collection, new FileOutputStream("jaxbOutput.xml"));

# Unmarshalling:

# Unmarshalling an XML document means creating a tree of content objects that represents the content and organization of the document

# To unmarshal an XML document,

# Create a JAXBContext object(

# JAXBContext jaxbContext = JAXBContext.newInstance("package name ");

# Create an Unmarshaller object

# Unmarshaller unmarshaller = jc.createUnmarshaller();

# Call the unmarshal method unmarshaller.unmarshal(new File( "xml name"));

# Use the get methods in the schema-derived classes to access the XML data

# JSON interview question

# <http://www.withoutbook.com/Technology.php?tech=50&subject=JSON%20Interview%20Questions%20and%20Answers>

# (1)What is JSON?

# JSON full form is JavaScript Object Notation. JSON is a lightweight text-based open standard designed for human-readable data interchange. It is derived from the JavaScript programming language for representing simple data structures and associative arrays, called objects. And JSON is language-independent, with parsers available for virtually every programming language. Uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python,php

# The JSON format is often used for serializing and transmitting structured data over a network connection. When third party data interchane(REST Services) then JSON may used there LIKE SHOP .It is primarily used to transmit data between a server and web application, serving as an alternative to XML.

# (2)What is the file extension of JSON?

# The JSON filename extension is .json.

# (3) Why use JSON over XML?

# • Lighter and faster than XML as on-the-wire data format

# • JSON objects are typed while XML data is typeless

# > JSON types: string, number, array, boolean,

# > XML data are all string

# • Native data form for JavaScript code

# > Data is readily accessible as JSON objects in your JavaScript

# code vs. XML data needed to be parsed and assigned to variables through tedious DOM APIs

# > Retrieving values is as easy as reading from an object property in your JavaScript code

# (4)Explain JSON Structures.

# • A collection of name/value pairs

# > In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array

# • An ordered list of values

# > In most languages, this is realized as an array, vector, list, or sequence

# • These are universal data structures supported

# • A JSON object is an unordered set of name/value pairs

# • A JSON object begins with { (left brace) and ends with } (right brace)

# • Each name is followed by : (colon) and the name/value pairs are separated by , (comma)

# (5)How to Generate or Send JSON Data at the Server Side?

# • Create JSON Java object

# • Add name and value pairs using put method

# • Convert it to String type using toString method and send it to the client with content-type as \"text/xml\" or \"text/plain\"

# myString = new JSONObject().put(\"JSON\", \"Hello, World!\").toString();

# // myString is {\"JSON\": \"Hello, World\"}

# (6) How to receive JSON Data at the Client Side?

# • JSON data is received as a string

# • Calling eval() will generate JSON object in JavaScript code

# var JSONdata = eval(req.responseText);

# • Once you have JSON object, you can use . notation to access its properties

# var name = JSONdata.name;

# var address = JSONdata.addresses[3];

# var streetname = JSONdata.addresses[3].street;

# (7) How to receive JSON Data at the Server Side?

# • Read the JSON data as a String type

# • Create JSONObject Java object from the string String json = readJSONStringFromRequestBody(request);

# //Use the JSON-Java binding library to create a JSON object in Java JSONObject jsonObject = null;

# try {

# jsonObject = new JSONObject(json);

# }

# catch(ParseException pe) {

# }

# Other Java Questions

1. Use iterate and recursive way to conculate factorial ?

<http://www.programmerinterview.com/index.php/general-miscellaneous/java-method-calculate-factorial/>

Solution 1: Recursive

Boundary conditions in recursion prevent infinite function calls

One thing is obviously missing from the code above. If we were to pass in a number, then the function simply will not stop executing! We need to add a boundary condition, but what should it be? Because factorials are only defined for non negative integers, it makes sense to use 0 as our boundary case. So, if ‘n’ (the number passed in to the function) ever equals 0 then we will return a 1, because the factorial of 0 is 1.

public int Factorial(int n) {

if(n == 0) {

return 1;

} else {

return n \* Factorial(n - 1);

}

}

Solution 2: Iterate for loop

Short Coming of recursive method

Although we presented the recursive answer to the question above, using recursion is not the best solution to the problem – especially when you are dealing with very large numbers. For instance, if you are trying to calculate the factorial of a large number like 1,000,000 then you could very well end up with a stack overflow issue – because of the large amount of memory required by too many recursive calls

public int Factorial(int n) {

int X, Fact = 1;

for(int x = n; x > 1; x--) {

fact \* x = fact;

}

return fact;

}

1. Sorting Array in O(n) time

<http://stackoverflow.com/questions/12240997/sorting-array-in-on-run-time>

3.

(1)What will be the output of the program?

public class Test138

{

public static void stringReplace (String text)

{

text = text.replace ('j' , 'c'); /\* Line 5 \*/

}

public static void bufferReplace (StringBuffer text)

{

text = text.append ("c"); /\* Line 9 \*/

}

public static void main (String args[])

{

String textString = new String ("java");

StringBuffer textBuffer = new StringBuffer ("java"); /\* Line 14 \*/

stringReplace(textString);

bufferReplace(textBuffer);

System.out.println (textString + textBuffer);

}

}

string is immutable, it cannot be changed, that's the reason for the StringBuffer class. The stringReplace method does not change the string declared on line 14, so this remains set to "java".

Method parameters are always passed by value - a copy is passed into the method - if the copy changes, the original remains intact, line 5 changes the reference i.e. text points to a new String object, however this is lost when the method completes. The textBuffer is a StringBuffer so it can be changed.

This change is carried out on line 9, so "java" becomes "javac", the text reference on line 9 remains unchanged. This gives us the output of "javajavac"

# <http://www.indiabix.com/online-test/java-programming-test/61>

# What will be the output of the program?

String a = "ABCD";

String b = a.toLowerCase();

b.replace('a','d');

b.replace('b','c');

System.out.println(b);

String objects are immutable, they cannot be changed, in this case we are talking about the replace method which returns a new String object resulting from replacing all occurrences of oldChar in this string with newChar.

b.replace(char oldChar, char newChar);

But since this is only a temporary String it must either be put to use straight away i.e.

System.out.println(b.replace('a','d'));

Or a new variable must be assigned its value i.e.

String c = b.replace('a','d');

# <http://www.indiabix.com/online-test/java-programming-test/63>

# (2)

/\* Missing statements ? \*/

public class NewTreeSet extends java.util.TreeSet

{

public static void main(String [] args)

{

java.util.TreeSet t = new java.util.TreeSet();

t.clear();

}

public void clear()

{

TreeMap m = new TreeMap();

m.clear();

}

}

which two statements, added independently at beginning of the program, allow the code to compile?

1. No statement is required
2. import java.util.\*;
3. import.java.util.Tree\*;
4. import java.util.TreeSet;
5. import java.util.TreeMap;

(2) and (5). TreeMap is the only class that must be imported. TreeSet does not need an import statement because it is described with a fully qualified name.

<http://www.indiabix.com/online-test/java-programming-test/62>

# (3) Difference between Null and “” & Difference between == and equals()

A. String a1 = **null**;

String b1 = **null**;

# System.*out*.println("a1.equals(b1) is " + a1.equals(b1));

Return: Exception in thread "main" java.lang.NullPointerException

# at Test.main(Test.java:7)

B. String a1 = **null**;

String b1 = **null**;

# System.*out*.println("a1 == b1 is " + (a1 == b1));

Return: a1 == b1 is true

C. String a2 = "";

String b2 = **null**;

System.*out*.println("a2.equals(b2) is " + a2.equals(b2));

Return: a2.equals(b2) is false

D. String a2 = "";

String b2 = **null**;

System.*out*.println("a2 == b2 is " + (a2 == b2));

Return: a2 == b2 is false

E. String a3 = "";

String b3 = "";

System.*out*.println("a3.equals(b3) is " + a3.equals(b3));

Return: a3.equals(b3) is true

F. String a3 = "";

String b3 = "";

System.*out*.println("a3 == b3 is " + (a3 == b3));

Return: a3 == b3 is true

G. String a4 = "abc";

String b4 = "";

System.*out*.println("a4.equals(b4) is " + a4.equals(b4));

Return: a4.equals(b4) is false

H. String a4 = "abc";

String b4 = "";

System.*out*.println("a4 == b4 is " + (a4 == b4));

Return: a4 == b4 is false

I. String a5 = "abc";

String b5 = "abc";

System.*out*.println("a5.equals(b5) is " + a5.equals(b5));

Return: a5.equals(b5) is true

J. String a5 = "abc";

String b5 = "abc";

System.*out*.println("a5 == b5 is " + (a5 == b5));

Return: a5 == b5 is true

**"" is an actual string, albeit an empty one.**

**Null, however, means that the String variable points to nothing.**

For Case D.

a==b returns false because "" and null do not occupy the same space in memory--in other words, their variables don't point to the same objects.

For Case C.

a.equals(b) returns false because "" does not equal null, obviously.

The difference is though that since "" is an actual string, you can still invoke methods or functions on it like a.length(), a.substring(0, 1) and so on.

If the String equals null, like b, Java would throw a NullPointerException if you tried invoking, say: b.length()

If the difference you are wondering about is == versus equals, it's this:

== compares references, like if I went

String a = new String("");

String b = new String("");

System.out.println(a==b);

That would output false because I allocated two different objects, and a and b point to different objects. **Here we use String a = new String(“”) method, not use String a = “”, their results are different, because, in special cas, String a = “” and String b = “”, when compare use == , it is true like F. Because when we don’t use “new String()”, we are using string pool mechanism, this will first check whether you have new an object like “” here in our string pool, if it is like String a = ””, then the next object which also be “” create by String b = “” will still use the same object in string pool, so when we use == to check a and b, they are exactly the same object, and return true.**

However, a.equals(b) in this case would return true, because equals for Strings will return true [if and only if the argument String is not null and represents the same sequence of characters.](http://download.oracle.com/javase/1.4.2/docs/api/java/lang/String.html#equals%28java.lang.Object%29)

For Case F, J

Be warned, though, that Java does have a special case for Strings.

String a = "abc";

String b = "abc";

System.out.println(a==b);

You would think that the output would be false, since it should allocate two different Strings. Actually, Java will [intern](http://download.oracle.com/javase/1.4.2/docs/api/java/lang/String.html#intern%28%29) literal Strings (ones that are initialized like a and b in our example). So be careful, because that can give some false positives on how == works.

For Case B.

Both a1 and b1 are null, compare them with == means compare their reference values, but they haven’t initialized yet, so they have same reference as 0.

<http://stackoverflow.com/questions/4802015/difference-between-null-and-java-string>

**Example of handling different java exceptions**

Java exception handling tutorial: In this tutorial we will learn how to handle exception with the help of suitable examples. **Exceptions are errors which occur when the program is executing.** Consider the Java program below which divides two integers.

**import** java.util.Scanner;

**class** Division {

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

**int** a, b, result;

Scanner input = **new** Scanner(System.in);

System.out.println("Input two integers");

a = input.nextInt();

b = input.nextInt();

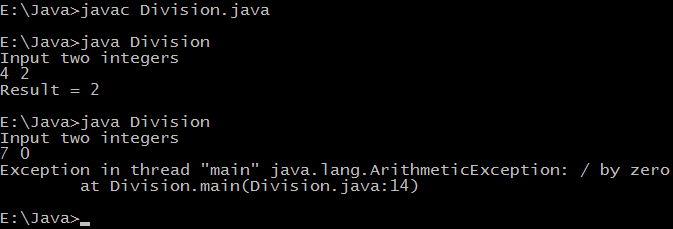
result = a / b;

System.out.println("Result = " + result);

}

}

Now we compile and execute the above code two times, see the output of program in two cases:



In the second case we are dividing a by zero which is not allowed in mathematics, so a run time error will occur i.e. an exception will occur. If we write programs in this way then they will be terminated abnormally and user who is executing our program or application will not be happy. This occurs because input of user is not valid so we have to take a preventive action and the best thing will be to notify the user that it is not allowed or any other meaningful message which is relevant according to context. You can see the information displayed when exception occurs it includes name of thread, file name, line of code (14 in this case) at which exception occurred, name of exception (ArithmeticException) and it's description('/ by zero'). Note that exceptions don't occur only because of invalid input only there are other reasons which are beyond of programmer control such as stack overflow exception, out of memory exception when an application requires memory larger than what is available.

Java provides a powerful way to handle such exceptions which is known as exception handling. In it we write vulnerable code i.e. code which can throw exception in a separate block called as **try** block and exception handling code in another block called **catch** block. Following modified code handles the exception.

## Java exception handling example

**class** Division {

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

**int** a, b, result;

Scanner input = **new** Scanner(System.in);

System.out.println("Input two integers");

a = input.nextInt();

b = input.nextInt();

*// try block*

**try** {

result = a / b;

System.out.println("Result = " + result);

}

*// catch block*

**catch** (ArithmeticException e) {

System.out.println("Exception caught: Division by zero.");

}

}

}

Whenever an exception is caught corresponding catch block is executed, For example above code catches ArithmeticException only. If some other kind of exception is thrown it will not be caught so it's the programmer work to take care of all exceptions as in our try block we are performing arithmetic so we are capturing only arithmetic exceptions. A simple way to capture any exception is to use an object of Exception class as other classes inherit Exception class, see another example below:

**class** Exceptions {

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

String languages[] = { "C", "C++", "Java", "Perl", "Python" };

**try** {

**for** (**int** c = 1; c <= 5; c++) {

System.out.println(languages[c]);

}

}

**catch** (Exception e) {

System.out.println(e);

}

}

}

Output of program:

C++

Java

Perl

Python

java.lang.ArrayIndexOutOfBoundsException: 5

Here our catch block capture an exception which occurs because we are trying to access an array element which does not exists (languages[5] in this case). Once an exception is thrown control comes out of try block and remaining instructions of try block will not be executed. At compilation time syntax and semantics checking is done and code is not executed on machine so exceptions can only be detected at run time.

## Finally block in Java

Finally block is always executed whether an exception is thrown or not.

**Why we need to use finally ?**

**Because you need that code to execute regardless of any exceptions that may be thrown. For example, you may need to clean up some unmanaged resource, like in my android project, I need to clean up business card image stored in local place, and start camera to retake the photo again. (the 'using' construct compiles to a try/finally block).**

**class** Allocate {

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

**try** {

**long** data[] = **new** **long**[1000000000];

}

**catch** (Exception e) {

System.out.println(e);

}

**finally** {

System.out.println("finally block will execute always.");

}

}

}

Output of program:

**finally** block will execute always.

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

at Allocate.main(Allocate.java:5)

Exception occurred because we try to allocate a large amount of memory which is not available.

<http://www.programmingsimplified.com/java/tutorial/java-exception-handling-tutorial>

**Why NullPointerException occur in your code**

**NullPointerException is a situation in code where you try to access/ modify an object which has not been initialized yet.** It essentially means that object reference variable is not pointing anywhere and refers to nothing or ‘null’. A simple example can be:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | package com.howtodoinjava.demo.npe;  public class SampleNPE {      public static void main(String[] args) {          String s = null;          System.out.println(s.toString()); // s is un-initialized and is null      }  } |

**Common places where NullPointerException usually occur**

Well, NullPointerException can occur anywhere in your code for various reasons but i have prepared list of most frequent places based on my experience:

1) Invoking methods on an object which is not initialized  
2) Parameters passed in a method are null  
3) Calling toString() method on object which is null  
4) Comparing object properties in if block without checking null equality  
5) Incorrect configuration for frameworks like spring which works on dependency injection  
6) Using synchronized on an object which is null  
7) Chained statements i.e. multiple method calls in a single statement

This is not an exhaustive list. There are several other places and reasons also. If you can recall any such other, please leave a comment. it will help others (beginners) also.

**Best ways to avoid NullPointerException**

**1) Ternary Operator**

This [operator](http://en.wikipedia.org/wiki/%3F:#Java) results to the value on the left hand side if not null else right hand side is evaluated. It has syntax like :

boolean expression ? value1 : value2;

If expression is evaluated as true then entire expression returns value1 otherwise value2. Its more like if-else construct but it is more effective and expressive. To prevent NullPointerException (NPE) , use this operator like below code:

String str = (param == null) ? "NA" : param;

**2) Use apache commons StringUtils for String operations**

[Apache commons lang](http://commons.apache.org/proper/commons-lang/) is a collection of several utility classes for various king of operation. One of them is [StringUtils.java](http://commons.apache.org/proper/commons-lang/javadocs/api-2.6/org/apache/commons/lang/StringUtils.html). Use***StringUtils.isNotEmpty()*** for verifying if string passed as parameter is null or empty string. If it is not null or  empty; then use it further.

Other similar methods are ***StringUtils. IsEmpty(), and StringUtils.equals().***They claim in their javadocs that if StringUtils.isNotBlank() throws an NPE, then there is a bug in the API.

|  |  |
| --- | --- |
| 1  2  3  4 | if (StringUtils.isNotEmpty(obj.getvalue())){      String s = obj.getvalue();      ....  } |

**3) Check Method Arguments for null very early**

You should always put input validation at the beginning of your method so that the rest of your code does not have to deal with the possibility of incorrect input. So if someone passes in a null, things will break  early in the stack rather than in some deeper location where the root problem will be rather difficult to identify.

*Aiming for fail fast behavior is a good choice in most situations.*

**4) Consider Primitives Rather than Objects**

Null problem occurs where object references points to nothing. So it is always safe to use primitives as much as possible because they does not suffer with null references. All primitives must have some default values also attached so beware of it.

**5) Carefully Consider Chained Method Calls**

While chained statements are nice to look at in the code, they are not NPE friendly. A single statement spread over several lines will give you the line number of the first line in the stack trace regardless of where it occurs.

ref.method1().method2().method3().methods4();

These kind of chained statement will print only “*NullPointerException occurred in line number xyz”.*It really is hard to debug such code. Avoid such calls.

**6) Use String.valueOf() Rather than toString()**

If you have to print the string representation of any object, the don’t use object.toString(). This is a very soft target for NPE. Instead use String.valueOf(object).  
Even if object is null in second method, it will not give exception and will prints ‘null’ to output stream.

**7) Avoid returning null from your methods**

An awesome tip to avoid NPE is to return empty strings or empty collections rather than a null. Do this consistently across your application. You will note that a bucket load of null checks become unneeded if you do so.

An example could be:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<string> data = null;    @SuppressWarnings("unchecked")  public List getDataDemo()  {      if(data == null)          return Collections.EMPTY\_LIST; //Returns unmodifiable list      return data;  } |

Users of above method, even if they missed the null check, will not see ugly NPE.

**8) Discourage Passing of Null Parameters**

I have seen some method declarations where method expects two or more parameters. If one of parameter is passed as null, then also method works if some different manner. Avoid this.

In stead you should define two methods; one with single parameter and second with two parameters. Make parameters passing mandatory. This helps a lot when writing application logic inside methods because you are sure that method parameters will not be null; so you don’t put unnecessary assumptions and assertions.

**9) Call String.equals(String) on ‘Safe’ Non-Null String**

In stead of writing below code for string comparison

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class SampleNPE {      public void demoEqualData(String param) {          if (param.equals("check me")) {              // some code          }      }  } |

write above code like this. This will not cause in NPE even if param is passed as null.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class SampleNPE {      public void demoEqualData(String param) {          if ("check me".equals(param)) // Do like this          {              // some code          }      }  } |

**Available NullPointerException safe operations**

**instanceof operator**

The instanceof operator is NPE safe. So, instanceof null always returns false. It does not cause a NullPointerException. You can eliminate messy conditional code if you remember this fact.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // Unnecessary code  if (data != null &amp;&amp; data instanceof InterestingData) {  }    // Less code. Better!!  if (data instanceof InterestingData) {  } |

**Accessing static members of a class**

If you are dealing with static variables or static method than you won’t get null pointer exception even if you have your reference variable pointing to null because static variables and method call are bonded during compile time based on class name and not associated with object

|  |  |
| --- | --- |
| 1  2 | MyObject obj = null;  String attrib = obj.staticAttribute; //no NullPointerException because staticAttribute is static variable defined in class MyObject |

Please let me know if you know some more such language constructs which does not fail when null is encountered.

**What if you must allow NullPointerException in some places**

***Joshua bloch*** in effective java says that “Arguably, all erroneous method invocations boil down to an illegal argument or illegal state, but other exceptions are standardly used for certain kinds of illegal arguments and states. If a caller passes null in some parameter for which null values are prohibited, convention dictates that NullPointerException be thrown rather than IllegalArgumentException.”

So if you must allow NullPointerException in some places in you code then **make sure you make them more informative then they usually are**. Take a look at below example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | package com.howtodoinjava.demo.npe;    public class SampleNPE {      public static void main(String[] args) {          // call one method at a time          doSomething(null);          doSomethingElse(null);      }        private static String doSomething(final String param) {          System.out.println(param.toString());          return "I am done !!";      }        private static String doSomethingElse(final String param) {          if (param == null) {              throw new NullPointerException(                      " :: Parameter 'param' was null inside method 'doSomething'.");          }          System.out.println(param.toString());          return "I am done !!";      }  } |

Output of both method calls is this:

Exception in thread "main" java.lang.NullPointerException

at com.howtodoinjava.demo.npe.SampleNPE.doSomething(SampleNPE.java:14)

at com.howtodoinjava.demo.npe.SampleNPE.main(SampleNPE.java:8)

Exception in thread "main" java.lang.NullPointerException:  :: **Parameter 'param' was null inside method 'doSomething'.**

at com.howtodoinjava.demo.npe.SampleNPE.doSomethingElse(SampleNPE.java:21)

at com.howtodoinjava.demo.npe.SampleNPE.main(SampleNPE.java:8)

Clearly second stack trace is more informative and makes debugging easy. Use this in future.

I am done with my experience around NullPointerException till date [As much i can recall]. If you know other points around the topic, please share with all of us !!

# <http://howtodoinjava.com/2013/04/05/how-to-effectively-handle-nullpointerexception-in-java/>

# (4) Null Safe way to Check if String is Empty or Not in Java 1) Using equals Method As discussed in my tips to deal with NullPointerException, I have mentioned a technique to call equals() method on known String object instead of calling on unknown object. This technique can be used to check *emptiness of String* as well. All you need to do is to call equals() method on [empty String literal](http://java67.blogspot.sg/2014/08/difference-between-string-literal-and-new-String-object-Java.html) and pass the object you are testing as shown below :

String nullString = null;

String empty = new String();

boolean test = "".equals(empty); // true

System.out.println(test);

boolean check = "".equals(nullString); // false

System.out.println(check);  
There will be no null pointers in this case.  Both of these way confirms that *String is not null and empty*.

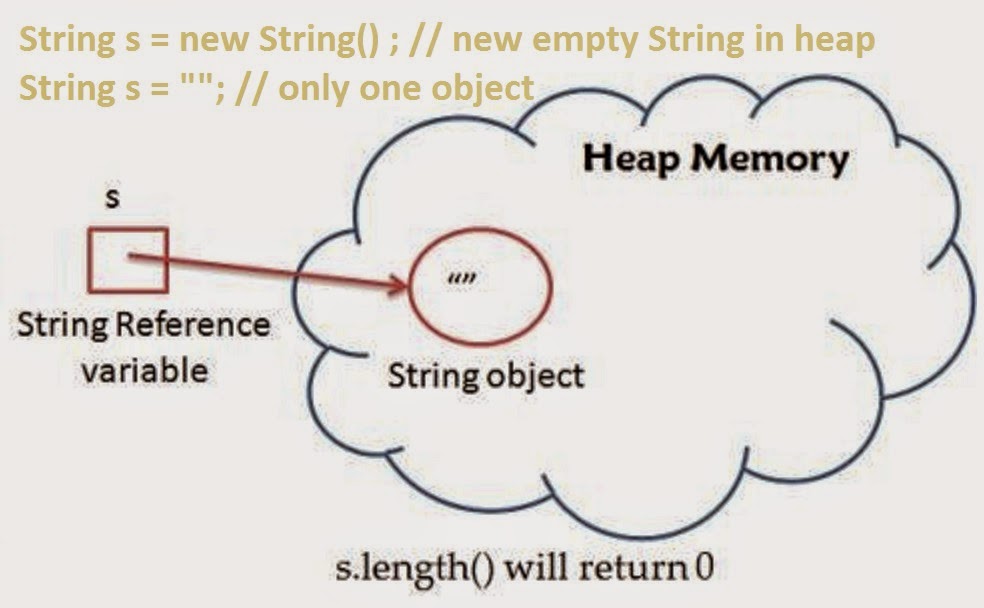
**2) Using Apache Commons StringUtils class**  
If you are already using Apache commons then you can also use isEmpty() method to check if String is empty or not. Only caveat here is that this method return true in case of null input as well, which may not be correct depending upon your application's definition of empty String. If you treat null as empty String then you can use this wonderful [null-safe method](http://java67.blogspot.sg/2012/09/top-10-tricky-java-interview-questions-answers.html)for quick test, as shown in following example :

boolean nullCheck = StringUtils.isEmpty(nullString);

boolean emptyCheck = StringUtils.isEmpty("");

System.out.println(nullCheck); // true

System.out.println(emptyCheck); // true  
You an see that check against null String is opposite to how equals method behave. So use this method with caution. Since they return true even for null input, they cannot be trusted for tests like not null and empty. By the way, if you have to create empty String, always use String literal "", it's more memory efficient. Since [String object is Immutable](http://java67.blogspot.sg/2014/01/why-string-class-has-made-immutable-or-final-java.html) and can safely share between threads, there is no need to create multiple separate String object, as String s = new String() will do, as shown in following diagram.

[](http://1.bp.blogspot.com/-0LN0RdV8Hvw/VBBaEA9i3QI/AAAAAAAAB38/sijM8W44BLg/s1600/Empty+String+check+in+Java.jpg)

### Use equals() is better

There are many ways to check if String is empty in Java, but what is the right way of doing it? right in the sense of robustness, performance and readability. **If robustness is your priority then using** [**equals() method**](http://java67.blogspot.sg/2012/11/difference-between-operator-and-equals-method-in.html)or Apache commons **StringUtils** is the right way to do this check.

<http://java67.blogspot.com/2014/09/right-way-to-check-if-string-is-empty.html>

**Another 2 examples between == and equals()**

public class ObjComp

{

public static void main(String [] args )

{

int result = 0;

ObjComp oc = new ObjComp();

Object o = oc;

if (o == oc)

result = 1;

if (o != oc)

result = result + 10;

if (o.equals(oc) )

result = result + 100;

if (oc.equals(o) )

result = result + 1000;

System.out.println("result = " + result);

}

}

* it's .equals(...) though (first letter is not capitalized).
* equals will only compare what it is written to compare, no more, no less.
* **if equals is not overridden(Like overridden by String class in Java)**, it defaults to the Object#equals(Object o) method. Per the Object API this is the same as ==; that is, it returns true *if and only if* both variables refer to the same object, if their references are one and the same.
* **Always remember to override hashCode if you override equals so as not to "break the contract".**

<http://stackoverflow.com/questions/7520432/java-vs-equals-confusion>s

**So, here** even though o and oc are reference variables of different types, they are both referring to the same object. This means that == will resolve to true and that the default equals()method will also resolve to true.

public class Test178

{

public static void main(String[] args)

{

String s = "foo";

Object o = (Object)s;

if (s.equals(o))

{

System.out.print("AAA");

}

else

{

System.out.print("BBB");

}

if (o.equals(s))

{

System.out.print("CCC");

}

else

{

System.out.print("DDD");

}

}

}

The output will be AAACCC

<http://www.indiabix.com/online-test/java-programming-test/65>

Also, we can overridden equals() method by ourselves.

In Java, the equals() method that is inherited from Object is:

public boolean equals(Object other);

**In other words, the parameter must be of type Object.**

The ArrayList uses the correct equals method, where you were always calling the one that didn't properly override Object's equals.

Not overriding the method correctly can cause problems.

I override equals the following everytime:

@Override

public boolean equals(Object other){

if (other == null) return false;

if (other == this) return true;

if (!(other instanceof MyClass))return false;

MyClass otherMyClass = (MyClass)other;

...test other properties here...

}

<http://stackoverflow.com/questions/185937/overriding-the-java-equals-method-quirk>

(5)Compilation fails because the argument of the while loop, the condition, must be of primitive type boolean. In Java, 1 does not represent the true state of a boolean, rather it is seen as an integer.

**public** **class** Test

{

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

{

**int** i = 0;

**while**(1)

{

**if**(i == 4)

{

**break**;

}

++i;

}

System.*out*.println("i = " + i);

}

}

(6) The code will not compile because a continue statement can only occur in a looping construct. If this syntax were legal, the combination of the continue and the if statements would create a kludgey kind of loop, but the compiler will force you to write cleaner code than this.

int I = 0;

label:

if (I < 2) {

System.out.print("I is " + I);

I++;

continue label;

}

(7) What will be the output of the program?

public class A

{

void A() /\* Line 3 \*/

{

System.out.println("Class A");

}

public static void main(String[] args)

{

new A();

}

}

# The specification at line 3 is for a method and not a constructor and this method is never called therefore there is no output. The constructor that is called is the default constructor. The code executes with no output.

# (8) What will be the output of the program?

class Test

{

public static void main(String [] args)

{

int x= 0;

int y= 0;

for (int z = 0; z < 5; z++)

{

if (( ++x > 2 ) || (++y > 2))

{

x++;

}

}

System.out.println(x + " " + y);

}

}

# The first two iterations of the for loop both x and y are incremented. On the third iteration x is incremented, and for the first time becomes greater than 2. The short circuit or operator|| keeps y from ever being incremented again and x is incremented twice on each of the last three iterations. x = 8, y = 2.

# (9) Which is true about an anonymous inner class?

# A. It can extend exactly one class and implement exactly one interface.

# B. It can extend exactly one class and can implement multiple interfaces.

# C. It can extend exactly one class or implement exactly one interface.

# D. It can implement multiple interfaces regardless of whether it also extends a class.

# Option C is correct because the syntax of an anonymous inner class allows for only one named type after the new, and that type must be either a single interface (in which case the anonymous class implements that one interface) or a single class (in which case the anonymous class extends that one class).

# <http://www.indiabix.com/online-test/java-programming-test/63>

# (10)

public class MyOuter

{

public static class MyInner

{

public static void foo() { }

}

}

# which statement, if placed in a class other than MyOuter or MyInner, instantiates an instance of the nested class?

# MyInner is a static nested class, so it must be instantiated using the fully-scoped name ofMyOuter.MyInner.

# <http://www.indiabix.com/online-test/java-programming-test/63>

# (11) which two code fragments will compile?

interface Base

{

boolean m1 ();

byte m2(short s);

}

1. interface Base2 implements Base {}
2. abstract class Class2 extends Base   
   { public boolean m1(){ return true; }}
3. abstract class Class2 implements Base {}
4. abstract class Class2 implements Base   
   { public boolean m1(){ return (7 > 4); }}
5. abstract class Class2 implements Base   
   { protected boolean m1(){ return (5 > 7) }}

# (3) is correct because an abstract class doesn't have to implement any or all of its interface's methods. (4) is correct because the method is correctly implemented ((7 > 4) is a boolean).

# (1) is incorrect because interfaces don't implement anything. (2) is incorrect because classes don't extend interfaces. (5) is incorrect because interface methods are implicitly public, so the methods being implemented must be public.

# <http://www.indiabix.com/online-test/java-programming-test/64>

(12) which two of the following statements, inserted independently, could legally be inserted into missing section of this code?

import java.awt.\*;

class Ticker extends Component

{

public static void main (String [] args)

{

Ticker t = new Ticker();

/\* Missing Statements ? \*/

}

}

1. boolean test = (Component instanceof t);
2. boolean test = (t instanceof Ticker);
3. boolean test = t.instanceof(Ticker);
4. boolean test = (t instanceof Component);

# (2) is correct because class type Ticker is part of the class hierarchy of t; therefore it is a legal use of the instanceof operator. (4) is also correct because Component is part of the hierarchy of t, because Ticker extends Component.

# (1) is incorrect because the syntax is wrong. A variable (or null) always appears before the instanceof operator, and a type appears after it. (3) is incorrect because the statement is used as a method (t.instanceof(Ticker);), which is illegal.

# <http://www.indiabix.com/online-test/java-programming-test/64>

# (13) What will be the output of the program?

int i = 1, j = 10;

do

{

if(i > j)

{

break;

}

j--;

} while (++i < 5);

System.out.println("i = " + i + " and j = " + j);

The order is, test i against j, if bigger, it breaks from the loop, decrements j by one, and then tests the loop condition, where a pre-incremented by one i is tested for being lower than 5. The test is at the end of the loop, so i can reach the value of 5 before it fails. So it goes, start:

1, 10

2, 9

3, 8

4, 7

5, 6 loop condition fails.

<http://www.indiabix.com/online-test/java-programming-test/64>

# (14) What will be the output of the program?

public class Test

{

public static void main(String args[])

{

int i = 1, j = 0;

switch(i)

{

case 2: j += 6;

case 4: j += 1;

default: j += 2;

case 0: j += 4;

}

System.out.println("j = " + j);

}

}

Because there are no break statements, the program gets to the default case and adds 2 toj, then goes to case 0 and adds 4 to the new j. The result is j = 6.

<http://www.indiabix.com/online-test/java-programming-test/64>

(15) which two code fragments inserted at end of the program, will allow to compile?

interface DoMath

{

double getArea(int rad);

}

interface MathPlus

{

double getVol(int b, int h);

}

/\* Missing Statements ? \*/

1. class AllMath extends DoMath { double getArea(int r); }
2. interface AllMath implements MathPlus { double getVol(int x, int y); }
3. interface AllMath extends DoMath { float getAvg(int h, int l); }
4. class AllMath implements MathPlus { double getArea(int rad); }
5. abstract class AllMath implements DoMath, MathPlus { public double getArea(int rad) { return rad \* rad \* 3.14; } }

(3) are (5) are correct because interfaces and abstract classes do not need to fully implement the interfaces they extend or implement (respectively).

(16) What will be the output of the program?

public class ExamQuestion7

{

static int j;

static void methodA(int i)

{

boolean b;

do

{

b = i<10 | methodB(4); /\* Line 9 \*/

b = i<10 || methodB(8); /\* Line 10 \*/

}while (!b);

}

static boolean methodB(int i)

{

j += i;

return true;

}

public static void main(String[] args)

{

methodA(0);

System.out.println( "j = " + j );

}

}

The lines to watch here are lines 9 & 10. Line 9 features the non-shortcut version of the ORoperator so both of its operands will be evaluated and therefore methodB(4) is executed.

However line 10 has the shortcut version of the OR operator and if the 1st of its operands evaluates to true (which in this case is true), then the 2nd operand isn't evaluated, somethodB(8) never gets called.

The loop is only executed once, b is initialized to false and is assigned true on line 9. Thus j = 4.

# <http://www.indiabix.com/online-test/java-programming-test/65>

# Creating a jar File in Eclipse

# In eclipse

# <https://www.cs.utexas.edu/~scottm/cs307/handouts/Eclipse%20Help/jarInEclipse.htm>

# In terminal

The basic format of the command for creating a JAR file is:

jar cf *jar-file input-file(s)*

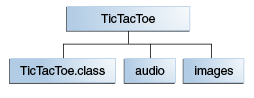
The options and arguments used in this command are:

* The c option indicates that you want to create a JAR file.
* The f option indicates that you want the output to go to a file rather than to stdout.
* jar-file is the name that you want the resulting JAR file to have. You can use any filename for a JAR file. By convention, JAR filenames are given a .jar extension, though this is not required.
* The input-file(s) argument is a space-separated list of one or more files that you want to include in your JAR file. The input-file(s) argument can contain the wildcard \* symbol. If any of the "input-files" are directories, the contents of those directories are added to the JAR archive recursively.

The c and f options can appear in either order, but there must not be any space between them.

This command will generate a compressed JAR file and place it in the current directory. The command will also generate a [default manifest file](http://docs.oracle.com/javase/tutorial/deployment/jar/defman.html) for the JAR archive.

Let us look at an example. A simple TicTacToe applet. This demo contains class files, audio files, and images having this structure:



**TicTacToe folder Hierarchy**

The audio and images subdirectories contain sound files and GIF images used by the applet. You can obtain all these files from *jar/examples* directory when you download the entire Tutorial online. To package this demo into a single JAR file named TicTacToe.jar, you would run this command from inside the TicTacToe directory:

jar cvf TicTacToe.jar TicTacToe.class audio images

# <http://docs.oracle.com/javase/tutorial/deployment/jar/build.html>

# Another Example:

1) Ensure that all necessary files are within the directory, you have opened a terminal/Command Prompt and have navigated to that directory.

2) Compile the .java class using javac, for example HelloWorld.java with

javac HelloWorld.java

3) This will produce a .class file needed for the JAR file.

4) Next create a manifest file (saved using the extension .txt) using the text editor and input the following

Main-Class: HelloWorld

or whatever your file's name is.

5) Next create the JAR file using this code by using jar cf jar-file input-file(s):

jar cfm HelloWorld.jar Manifest.txt HelloWorld.class

6) Run the file by using java –jar :

java -jar HelloWorld.jar

# So finally: javac🡪 jar –cf🡪 java -jar

# <http://stackoverflow.com/questions/10125639/how-to-create-a-jar-file-using-the-terminal>

**平常经常使用JAVA对文件进行读写等操作，这里汇总一下常用的文件操作。**

创建文件

**[java]** [view plaincopy](http://blog.csdn.net/brushli/article/details/12356695)

1. **public** **static** **boolean** createFile(String filePath){
2. **boolean** result = **false**;
3. File file = **new** File(filePath);
4. **if**(!file.exists()){
5. **try** {
6. result = file.createNewFile();
7. } **catch** (IOException e) {
8. e.printStackTrace();
9. }
10. }
12. **return** result;
13. }

创建文件夹

**[java]** [view plaincopy](http://blog.csdn.net/brushli/article/details/12356695)

1. **public** **static** **boolean** createDirectory(String directory){
2. **boolean** result = **false**;
3. File file = **new** File(directory);
4. **if**(!file.exists()){
5. result = file.mkdirs();
6. }
8. **return** result;
9. }

删除文件

**[java]** [view plaincopy](http://blog.csdn.net/brushli/article/details/12356695)

1. **public** **static** **boolean** deleteFile(String filePath){
2. **boolean** result = **false**;
3. File file = **new** File(filePath);
4. **if**(file.exists() && file.isFile()){
5. result = file.delete();
6. }
8. **return** result;
9. }

删除文件夹

递归删除文件夹下面的子文件和文件夹

**[java]** [view plaincopy](http://blog.csdn.net/brushli/article/details/12356695)

1. **public** **static** **void** deleteDirectory(String filePath){
2. File file = **new** File(filePath);
3. **if**(!file.exists()){
4. **return**;
5. }
7. **if**(file.isFile()){
8. file.delete();
9. }**else** **if**(file.isDirectory()){
10. File[] files = file.listFiles();
11. **for** (File myfile : files) {
12. deleteDirectory(filePath + "/" + myfile.getName());
13. }
15. file.delete();
16. }
17. }

读文件

（1）以字节为单位读取文件，常用于读二进制文件，如图片、声音、影像等文件

**[java]** [view plaincopy](http://blog.csdn.net/brushli/article/details/12356695)

1. **public** **static** String readFileByBytes(String filePath){
2. File file = **new** File(filePath);
3. **if**(!file.exists() || !file.isFile()){
4. **return** **null**;
5. }
7. StringBuffer content = **new** StringBuffer();
9. **try** {
10. **byte**[] temp = **new** **byte**[1024];
11. FileInputStream fileInputStream = **new** FileInputStream(file);
12. **while**(fileInputStream.read(temp) != -1){
13. content.append(**new** String(temp));
14. temp = **new** **byte**[1024];
15. }
17. fileInputStream.close();
18. } **catch** (FileNotFoundException e) {
19. e.printStackTrace();
20. } **catch** (IOException e) {
21. e.printStackTrace();
22. }
24. **return** content.toString();
25. }

 （2）以字符为单位读取文件，常用于读文本，数字等类型的文件，支持读取中文

**[java]** [view plaincopy](http://blog.csdn.net/brushli/article/details/12356695)

1. **public** **static** String readFileByChars(String filePath){
2. File file = **new** File(filePath);
3. **if**(!file.exists() || !file.isFile()){
4. **return** **null**;
5. }
7. StringBuffer content = **new** StringBuffer();
8. **try** {
9. **char**[] temp = **new** **char**[1024];
10. FileInputStream fileInputStream = **new** FileInputStream(file);
11. InputStreamReader inputStreamReader = **new** InputStreamReader(fileInputStream, "GBK");
12. **while**(inputStreamReader.read(temp) != -1){
13. content.append(**new** String(temp));
14. temp = **new** **char**[1024];
15. }
17. fileInputStream.close();
18. inputStreamReader.close();
19. } **catch** (FileNotFoundException e) {
20. e.printStackTrace();
21. } **catch** (IOException e) {
22. e.printStackTrace();
23. }
25. **return** content.toString();
26. }

（3）以行为单位读取文件，常用于读面向行的格式化文件

**[java]** [view plaincopy](http://blog.csdn.net/brushli/article/details/12356695)

1. **public** **static** List<String> readFileByLines(String filePath){
2. File file = **new** File(filePath);
3. **if**(!file.exists() || !file.isFile()){
4. **return** **null**;
5. }
7. List<String> content = **new** ArrayList<String>();
8. **try** {
9. FileInputStream fileInputStream = **new** FileInputStream(file);
10. InputStreamReader inputStreamReader = **new** InputStreamReader(fileInputStream, "GBK");
11. BufferedReader reader = **new** BufferedReader(inputStreamReader);
12. String lineContent = "";
13. **while** ((lineContent = reader.readLine()) != **null**) {
14. content.add(lineContent);
15. System.out.println(lineContent);
16. }
18. fileInputStream.close();
19. inputStreamReader.close();
20. reader.close();
21. } **catch** (FileNotFoundException e) {
22. e.printStackTrace();
23. } **catch** (IOException e) {
24. e.printStackTrace();
25. }
27. **return** content;
28. }

写文件

字符串写入文件的几个类中，FileWriter效率最高，BufferedOutputStream次之，FileOutputStream最差。

（1）通过FileOutputStream写入文件

**[java]** [view plaincopy](http://blog.csdn.net/brushli/article/details/12356695)

1. **public** **static** **void** writeFileByFileOutputStream(String filePath, String content) **throws** IOException{
2. File file = **new** File(filePath);
3. **synchronized** (file) {
4. FileOutputStream fos = **new** FileOutputStream(filePath);
5. fos.write(content.getBytes("GBK"));
6. fos.close();
7. }
8. }

（2）通过BufferedOutputStream写入文件

**[java]** [view plaincopy](http://blog.csdn.net/brushli/article/details/12356695)

1. **public** **static** **void** writeFileByBufferedOutputStream(String filePath, String content) **throws** IOException{
2. File file = **new** File(filePath);
3. **synchronized** (file) {
4. BufferedOutputStream fos = **new** BufferedOutputStream(**new** FileOutputStream(filePath));
5. fos.write(content.getBytes("GBK"));
6. fos.flush();
7. fos.close();
8. }
9. }

（3）通过FileWriter将字符串写入文件

**[java]** [view plaincopy](http://blog.csdn.net/brushli/article/details/12356695)

1. **public** **static** **void** writeFileByFileWriter(String filePath, String content) **throws** IOException{
2. File file = **new** File(filePath);
3. **synchronized** (file) {

有一个 class：

class myClass {

int i;

Map<String, String> myMap;

}

修改它，把它变成 immutable ?

# How to Create an Immutable Class in Java

<https://dzone.com/articles/how-to-create-an-immutable-class-in-java>

An object is immutable if its state cannot change after construction. Immutable objects don’t expose any way for other objects to modify their state; the object’s fields are initialized only once inside the constructor and never change again.

In this article, we'll define the typical steps for creating an immutable class in Java and also shed light on the common mistakes which are made by developers while creating immutable classes.

## 1. Usage of Immutable Classes

Nowadays, the “must-have” specification for every software application is to be distributed and multi-threaded—multi-threaded applications always cause headaches for developers since developers are required to protect the state of their objects from concurrent modifications of several threads at the same time, for this purpose, developers normally use the Synchronized blocks whenever they modify the state of an object.

With immutable classes, states are never modified; every modification of a state results in a new instance, hence each thread would use a different instance and developers wouldn’t worry about concurrent modifications.

## 2. Some Popular Immutable Classes

**String**is the most popular immutable class in Java. Once initialized its value cannot be modified. Operations like **trim(), substring(), replace()** always return a new instance and don’t affect the current instance, that’s why we usually call **trim()**as the following:

String alex = "Alex";

alex = alex.trim();

Another example from JDK is the wrapper classes like: **Integer, Float, Boolean** … these classes don’t modify their state, however they create a new instance each time you try to modify them.

Integer a =3;

a += 3;

After calling **a += 3,** a new instance is created holding the value: 6 and the first instance is lost.

## 3. How Do We Create an Immutable Class

In order to create an immutable class, you should follow the below steps:

1. Make your class **final,**so that no other classes can extend it.
2. Make all your fields **final,**so that they’re initialized only once inside the constructor and never modified afterward.
3. Don’t expose setter methods.
4. When exposing methods which modify the state of the class, you must always return a new instance of the class.
5. If the class holds a mutable object:
   * Inside the constructor, make sure to use a clone copy of the passed argument and never set your mutable field to the real instance passed through constructor, this is to prevent the clients who pass the object from modifying it afterwards.
   * Make sure to always return a clone copy of the field and never return the real object instance.

### 3.1. Simple Immutable Class

Let’s follow the above steps and create our own immutable class (**ImmutableStudent.java**).

package com.programmer.gate.beans;

public final class ImmutableStudent {

private final int id;

private final String name;

public ImmutableStudent(int id, String name) {

this.name = name;

this.id = id;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

}

The above class is a very simple immutable class which doesn’t hold any mutable object and never expose its fields in any way; these type of classes are normally used for caching purposes.

### 3.2. Passing Mutable Objects to Immutable Class

Now, let’s complicate our example a bit, we create a mutable class called **Age**and add it as a field to **ImmutableStudent:**

package com.programmer.gate.beans;

public class Age {

private int day;

private int month;

private int year;

public int getDay() {

return day;

}

public void setDay(int day) {

this.day = day;

}

public int getMonth() {

return month;

}

public void setMonth(int month) {

this.month = month;

}

public int getYear() {

return year;

}

public void setYear(int year) {

this.year = year;

}

}

package com.programmer.gate.beans;

public final class ImmutableStudent {

private final int id;

private final String name;

private final Age age;

public ImmutableStudent(int id, String name, Age age) {

this.name = name;

this.id = id;

this.age = age;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

public Age getAge() {

return age;

}

}

So, we added a new mutable field of type **Age** to our immutable class and assign it as normal inside the constructor.

Let’s create a simple test class and verify that **ImmutableStudent** is no more immutable:

public static void main(String[] args) {

Age age = new Age();

age.setDay(1);

age.setMonth(1);

age.setYear(1992);

ImmutableStudent student = new ImmutableStudent(1, "Alex", age);

System.out.println("Alex age year before modification = " + student.getAge().getYear());

age.setYear(1993);

System.out.println("Alex age year after modification = " + student.getAge().getYear());

}

After running the above test, we get the following output:

Alex age year before modification = 1992

Alex age year after modification = 1993

We claim that **ImmutableStudent** is an immutable class whose state is never modified after construction, however in the above example we are able to modify the age of **Alex**even after constructing **Alex** object. If we go back to the implementation of **ImmutableStudent** constructor, we find that age field is being assigned to the instance of the **Age** argument, so whenever the referenced **Age** is modified outside the class, the change is reflected directly on the state of **Alex.**Check out my [Pass by value OR pass by reference article](http://programmergate.com/java-pass-reference-pass-value/) to more deeply understand this concept.

In order to fix this and make our class again immutable, we follow step **#5** from the steps that we mention above for creating an immutable class. So we modify the constructor in order to clone the passed argument of **Age** and use a clone instance of it.

public ImmutableStudent(int id, String name, Age age) {

this.name = name;

this.id = id;

Age cloneAge = new Age();

cloneAge.setDay(age.getDay());

cloneAge.setMonth(age.getMonth());

cloneAge.setYear(age.getYear());

this.age = cloneAge;

}

Now, if we run our test, we get the following output:

Alex age year before modification = 1992

Alex age year after modification = 1992

As you see now, the age of **Alex** is never affected after construction and our class is back to immutable.

### 3.3. Returning Mutable Objects From Immutable Class

However, our class still has a leak and is not fully immutable, let’s take the following test scenario:

public static void main(String[] args) {

Age age = new Age();

age.setDay(1);

age.setMonth(1);

age.setYear(1992);

ImmutableStudent student = new ImmutableStudent(1, "Alex", age);

System.out.println("Alex age year before modification = " + student.getAge().getYear());

student.getAge().setYear(1993);

System.out.println("Alex age year after modification = " + student.getAge().getYear());

}

Output:

Alex age year before modification = 1992

Alex age year after modification = 1993

Again according to step **#4**, when returning mutable fields from immutable object, you should return a clone instance of them and not the real instance of the field.

So we modify **getAge()**in order to return a clone of the object’s age:

public Age getAge() {

Age cloneAge = new Age();

cloneAge.setDay(this.age.getDay());

cloneAge.setMonth(this.age.getMonth());

cloneAge.setYear(this.age.getYear());

return cloneAge;

}

Now the class becomes fully immutable and provides no way or method for other objects to modify its state.

Alex age year before modification = 1992

Alex age year after modification = 1992

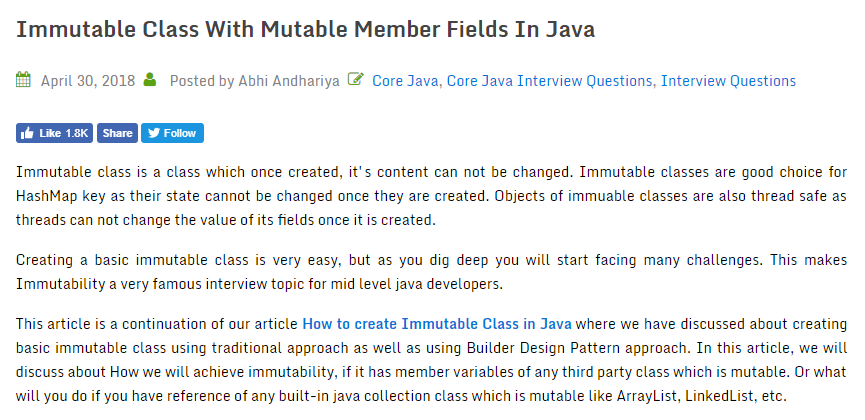
## 4. Conclusion

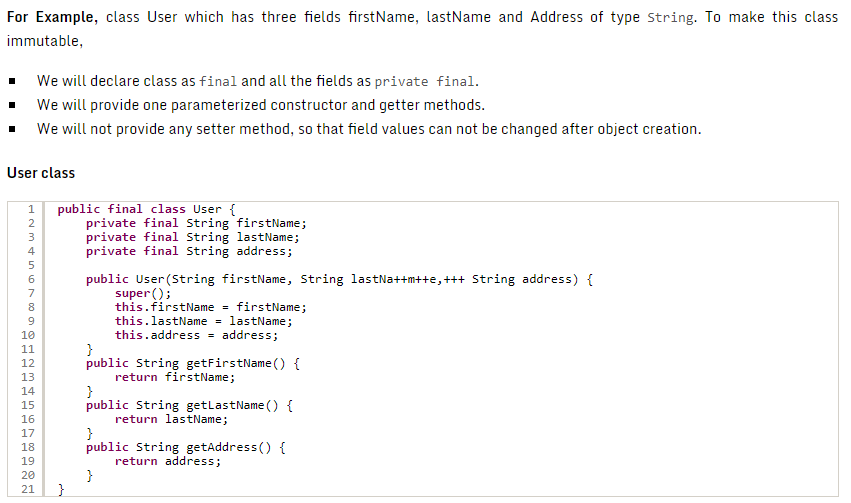
Immutable classes provide a lot of advantages especially when used correctly in a multi-threaded environment. The only disadvantage is that they consume more memory than the traditional class since upon each modification of them a new object is created in the memory... but, a developer should not overestimate the memory consumption as its negligible compared to the advantages provided by these type of classes.

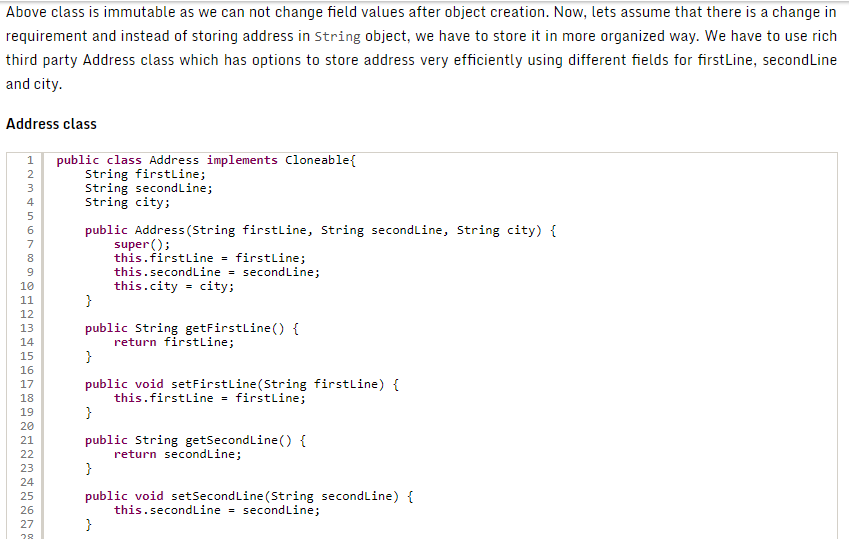
Finally, an object is immutable if it can present only one state to the other objects, no matter how and when they call its methods. If so it’s thread safe by any definition of thread-safe.

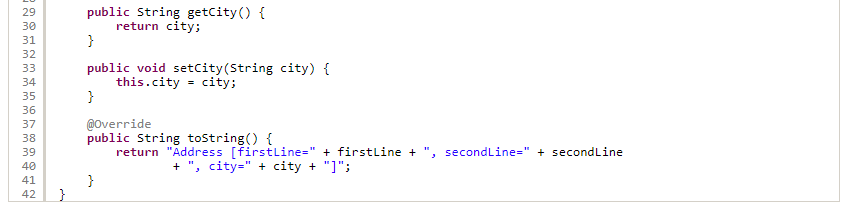
# Immutable class with mutable member fields in java

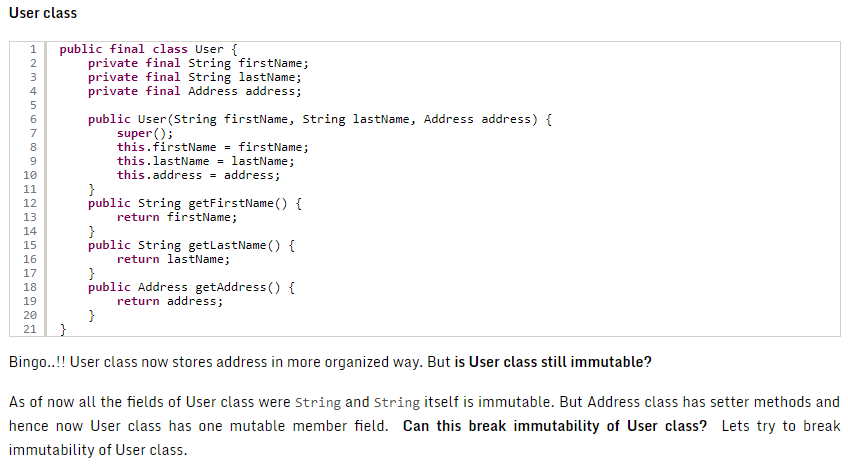
<https://codepumpkin.com/immutable-class-with-mutable-member-fields-in-java/>

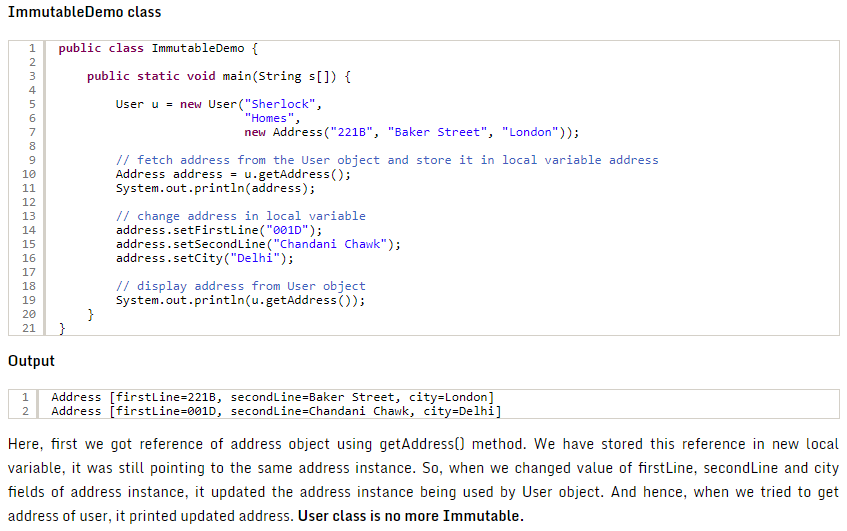


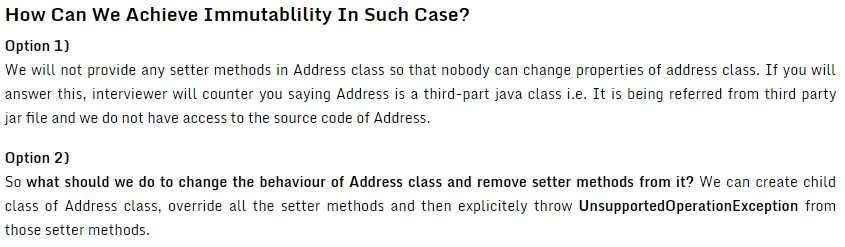


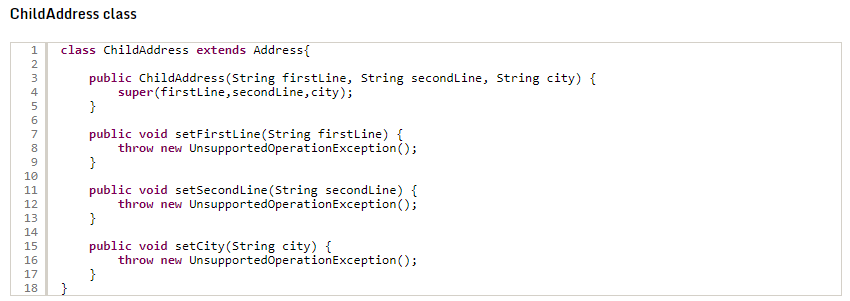


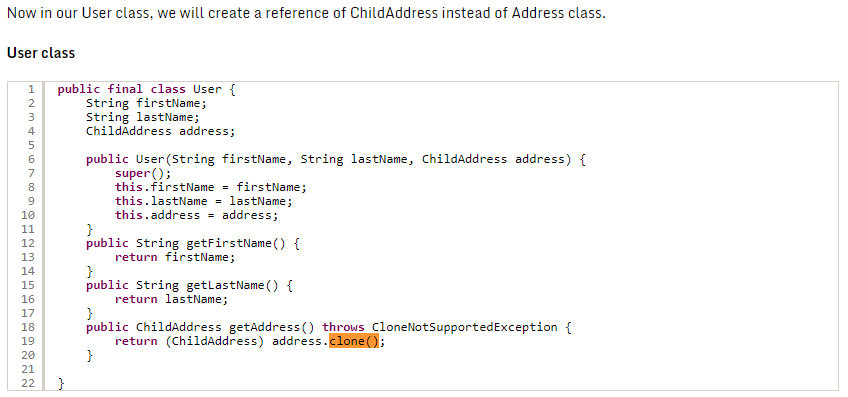




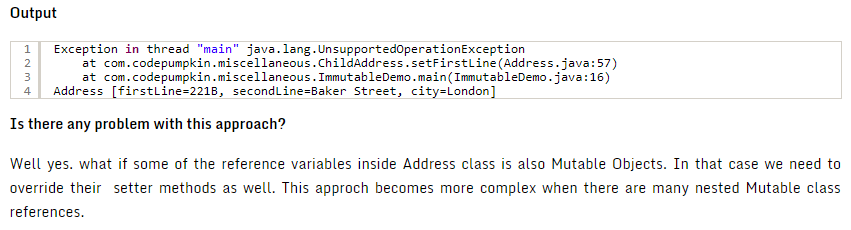


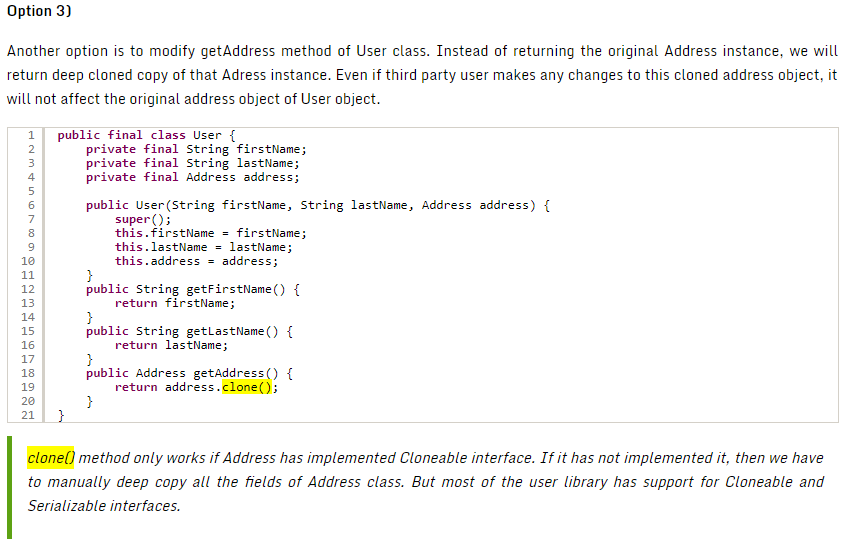


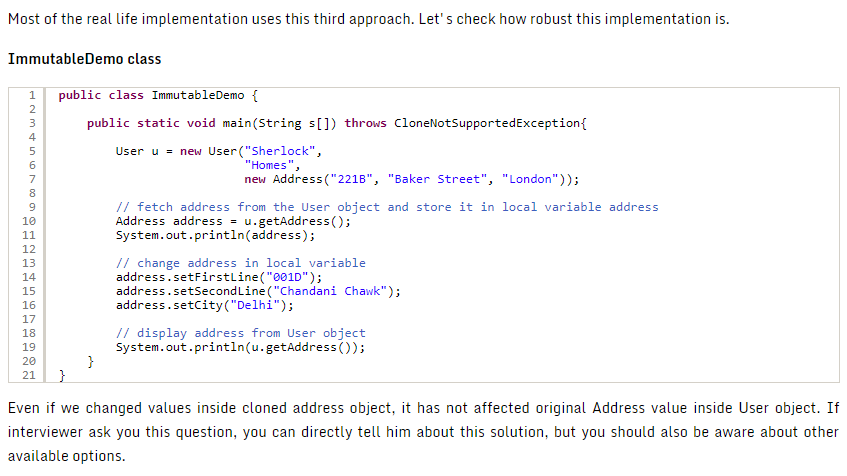


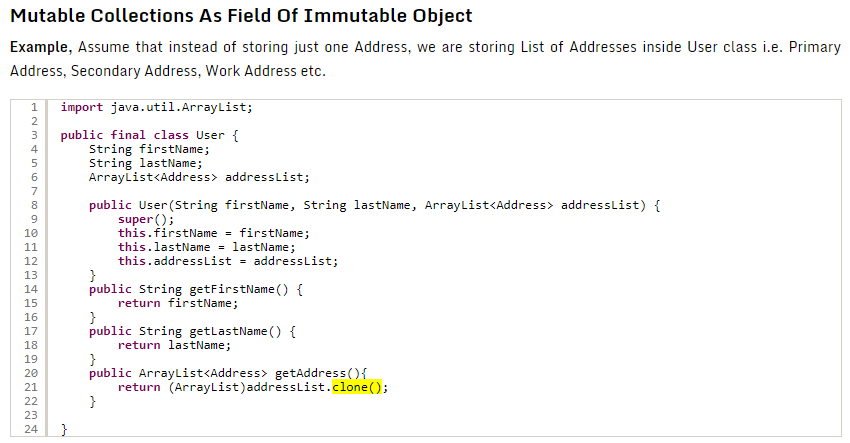


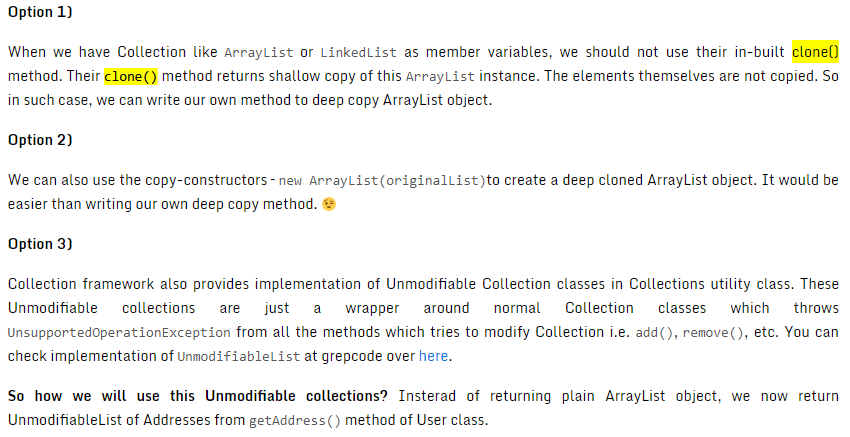


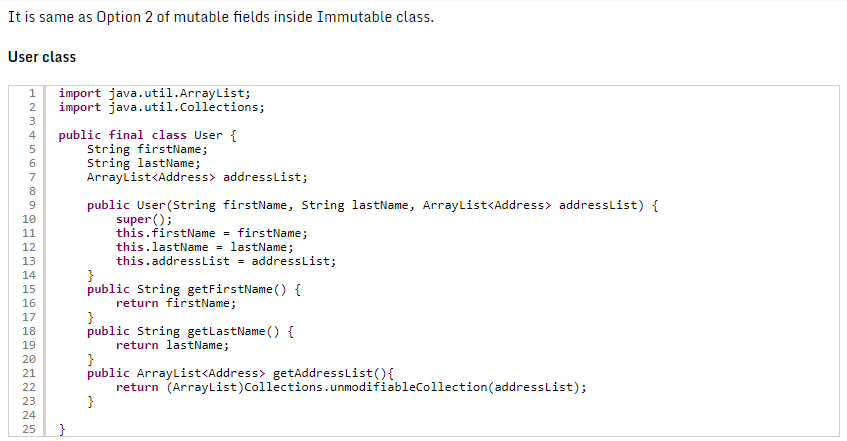


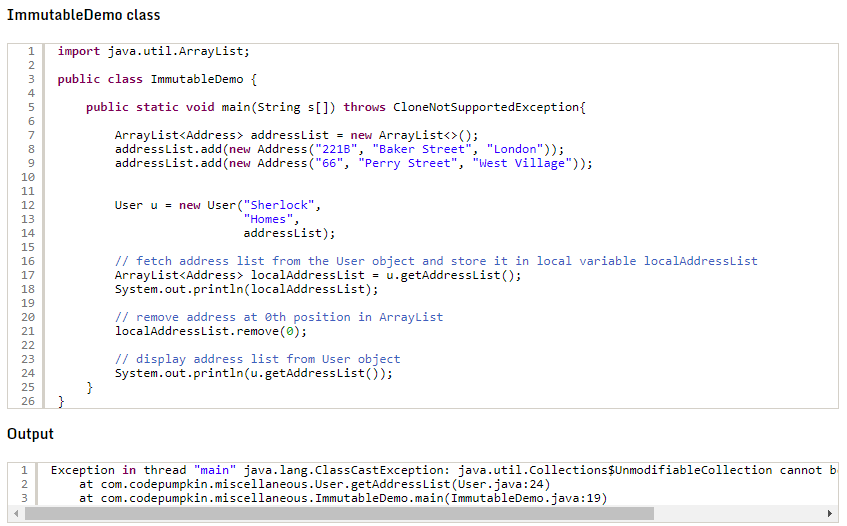












### Deep copy vs. Shallow copy?

<https://dzone.com/articles/java-copy-shallow-vs-deep-in-which-you-will-swim>

#### ****Shallow Copy****

First let’s talk about the shallow copy. A shallow copy of an object copies the ‘main’ object, but doesn’t copy the inner objects. The ‘inner objects’ are shared between the original object and its copy. For example, in our Person object, we would create a second Person, but both objects would share the same Name and Address objects.

public class Person {

private Name name;

private Address address;

public Person(Person originalPerson) {

this.name = originalPerson.name;

this.address = originalPerson.address;

}

[…]

}

The problem with the shallow copy is that the two objects are not independent. If you modify the Name object of one Person, the change will be reflected in the other Person object.

#### ****Deep Copy****

Unlike the shallow copy, a deep copy is a**fully independent copy of an object**. If we copied our Person object, we would copy the entire object structure.

public class Person {

private Name name;

private Address address;

public Person(Person otherPerson) {

this.name = new Name(otherPerson.name);

this.address = new Address(otherPerson.address);

}

[…]

}

### How to make a deep copy of Java ArrayList?

<https://stackoverflow.com/questions/7042182/how-to-make-a-deep-copy-of-java-arraylist>

<https://stackoverflow.com/questions/715650/how-to-clone-arraylist-and-also-clone-its-contents>