%{ConfusedCoders}

THE JAVA INTERVIEW β

YASH SHARMA | DEBARGHO CHATTERJEE

Please keep a paper and pencil handy, and have your Eclipse IDE ready.
We'd be digging Cool Concepts and Nasty Code bits here.

A Note from the Authors

This is a primer book to help you brush up your java concepts before taking up interviews. This book is targeted only on JAVA Interview and contains a smaller subset of the thousand interesting Java interview questions.
This book is not intended to teach you java; this is for giving you a quick walk through the old java concepts which has probably gone hazy with time.

Please do not try to memorize the answers but understand the points. We have tried to provide ample code snippets; even where the concept may be implicit, just that you can try those on your machines immediately.

Here you'll have lots of code to type and practice. We have not given output of many code blocks just because we need you to start hitting them on your keyboards. You cannot understand how things work until you see the outputs on the console. Let's stop being Lazy and start coding.

> Finally, feel free to share the book with friends.
> The book is absolutely free, so if some website is asking you money for the book,
> Say LoL on their faces, and download the book directly from www.ConfusedCoders.com.

Happy Learning, Cheers.

P.S. The book is made keeping in mind coders between 0 to 2.x years of java experience. Beyond that the book may not teach you anything new. You're already champs.

How is the book organized?

The book is organized into 3 sections.

Breaking the surface – Very fundamental Java questions to get you started.

Get set go – A collection of lot of Java questions, mostly objective type. Most probably you'll get all the average difficulty level questions from this section, and you might already be knowing answers to most of the questions in this section. Just skim through the known questions.

Only for the Caffeine Blooded – Cool conceptual questions dealing with Java Internals, Design Patterns, Data Structures etc. These questions are occasionally asked by the super techie interviewers and they may not be expecting perfect answers from you, since the answers varies from programmer to programmer. But if you answer these questions, you definitely have a cutting edge over the other candidates.

The best way of understanding concepts is to discuss the question with fellow coders, and get their point of view on it. It's the other way to keep stuffs glued to the brain.

Be sure to cover the 3rd section properly before the major interviews, and at least try out the code on your IDE's. The interviewers are smart enough to tweak the questions to trick you. Practicing the code is your only way out.

There are certain notations we have used in the book:

These kinds of Grey boxes are used to highlight some tricky points, or some points you must pay special attention.

<code />

A Blue box like this one would be used for providing the code snippets.
This is the part you should immediately type in your IDE.

About the Authors

Yash Sharma
Java Enthusiast and Big Data
Engineer at Impetus Inc.

Debargho Chatterjee
Java Senior Web Developer
at Make My Trip Pvt. Ltd.

Special Contributors

Ranjit Kumar Gupta
Java Developer at Samsung pvt. ltd

Abhi Verma
Java Developer at Infosys Technologies Ltd

Aditya Sharma
Java Developer at TCS

Jayant
Java & Mobile App
Developer at OnMobole

Contents

Breaking the Ice

This section of the book is to get you started with the first few questions that would come to the mind of any Java beginner.
No, I don't think you'd be asked these kind of questions unless its really your day. These are the very basic questions which you must definitely be aware of being a java programmer.

Just get the feel of the first few questions.

What is Java?

What better question to start a Java book with?

Java is a cool programming language. Its Free and Open Source; developed in Sun Microsystems (now Oracle Inc.). Its Object Oriented, and most of its syntax is primarily derived from C++, but little more simplified by better API's, extensive Exception Handling, better Threading support, easy connectivity with outer world, cool memory management etc.
It's a general purpose programming language suited for both Desktop Applications (by AWT & Swing), on Browsers(Applets), on Mobile Phones(Android) as well as for the Server Side programming (JSP, Servlets, Web Frameworks like Struts/Spring/JSF etc.).

It's Open Source and has a huge community of contributors and developers, and that also means that next time we're stuck with some issue we have large number of hotshots waiting to resolve our issues over the internet.
Java is platform independent and follows the Write Once, Run Anywhere vision, and that means that next time you've written your code on the windows machine, you just need to have the class file, and can execute the file from any machine in the world having the JRE, even on your cute mobile phone, and you're good to go.

Below are few more java features, which we would slightly touch in few questions,but the details are out of the scope of this puny book.
but of-course which you should explore in your free time :

- *Object Oriented technology*
- *Threads*
- *Input/output*
- *Data Structures*
- *System Properties, Date/Time*
- *Applets*
- *2D and 3D graphics*
- *Animation*
- *Mails, Electronic Signatures*
- *Cryptography, Key management*

Why Java?

This is a very loaded question, and lot of other programmers can get agitated defending their favorite language. Let's just say that java is another language in the programming jungle, but few points that really help it dominate the jungle is listed below:

Java is Open Source:
Java language is Free and Open Source, you are free to use it, develop your applications in it and distribute the apps for your own business absolutely free. You can even play with java internal codes for fun, and millions of community people would help you with your programming issues.

Cool first language:
Java is one of the best languages to start programming with. Java teaches us strong OOP Concepts, avoids lots of intuitive mistakes by great exception handling, Avoids pointer misuse as in traditional languages like C/C++, Implicit garbage collection, where the programmer does not have to bother about internal issues like memory leaks etc. The programmer just focused on the logic and he's good to go. More ever you don't you think that allocating/de-allocating memory for every silly thing before using memory is just boring, Java is all about focusing on code logic not memory issues.
NOM: to the cool memory level coders.

Java is object oriented:
Java has quite an extraordinary strength in code structure and design. Object Oriented code is one of the most demanded skills that any programmer must master, and java clearly helps us achieve that. Here java has a clear cut advantage over languages like C, C++(partially object oriented).

Java runs everywhere:
Java is a compiled as well as an interpreted language. First java code is compiled to machine understandable format, like any other compiled language, but it doesn't convert it to Zero's n One's like traditional compiled language, rather it converts it to a byte code which is a JVM understandable language. This byte code is machine independent, where any OS/Machine/Device having a JVM can run this byte code flawlessly. So this byte code is interpreted by the JVM to give you your desired output.

P.S. JVM itself is platform dependent, i.e. there are different JVM installation kits for Windows, Linux or Mac.

Java is simple and safe:
Java definitely help us avoid lot of complex stuffs that we used in languages like C/C++, java removes pointers for you, java removes operator overloading for you, no friend functions, no virtual functions, no explicit memory free/flush etc. Java's extensive Exception Handling mechanism enforces the programmer to be more careful while writing code and avoids the most intuitive programming errors.

Java is Growing (Internet and Mobiles):
Java is no longer a language which is limited to traditional desktop programming etc. in fact java is now a language that is more widely used for web applications and mobile platforms than the desktop apps. Java supports wide variety for web frameworks like Struts/ Spring/ JSF which are gaining huge popularity. The Android mobile platform is based on java which shows us the power of the language. Java itself has its J2ME platform for mobile applications too.

Java is as fast as C++:
In old days people were concerned about java's speed over the low level programming languages like C/C++, but then the JIT compiler was first made available as a performance update in the Java Development Kit (JDK) 1.1.6 release which made java performance as good as the performance of C++.

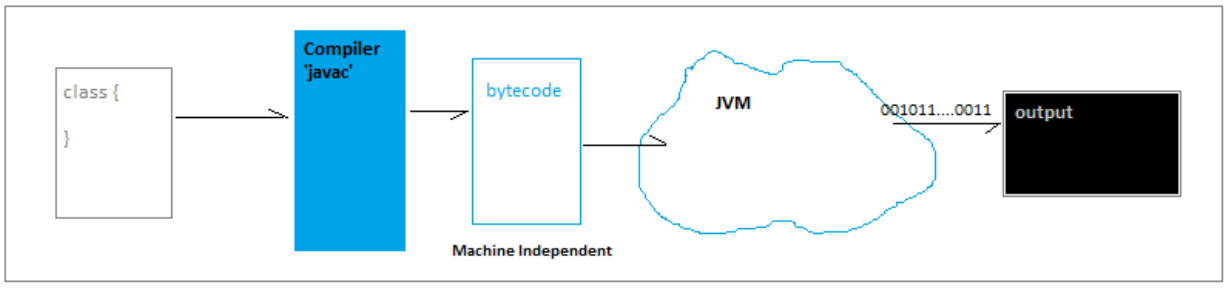
Some other Facts about Java:

- *1.1 billion desktops run Java*
- *930 million Java Runtime Environment downloads each year*
- *3 billion mobile phones run Java*
- *31 times more Java phones ship every year than Apple and Android combined*
- *100% of all Blu-ray players run Java*
- *1.4 billion Java Cards are manufactured each year*
- *Java powers set-top boxes, printers, Web cams, games, car navigation systems, lottery terminals, medical devices, parking payment stations, and more.*
  (Source: http://www.java.com/en/about/)

JRE, JVM and JDK – A basic explanation

These are the 3 terms that you probably came across when you tried your hands on Java for the first time, Let me just put it down in as fewest lines as possible:
JRE = JVM + Other Runtime Libraries (like java.util, java.lang, java.math and many many more…)
JDK = JRE + Tools to code a java program (e.g. Java Compiler 'javac', tools.jar, Java DBetc.)

Java is both a compiled as well as an interpreted language,
The Compiler inside the JDK (javac compiler) actually compiles the JAVA source code into Byte Code which the JVM finally interprets and gives us the output.



Note: *The standard JDK's you download comes pre shipped with the JRE and libraries, but you can also download and install the Standalone JRE which doesn't include the tools like javac, tools.jar, Java DB etc(only if you have the java .class file or the bytecode and just want to execute the code).*

Object Oriented Programming (OOPS) Concepts

Object Oriented Programming Approach has become a standard in the programming world, and every code written today uses OOP. OOP gives a structure to your code, makes it more modulated, easy to reuse and most importantly it keeps your code simple. Here are the chief OOP concepts we would use in our everyday coding:

Classes and Objects: The entry point into OOP world is class and objects. OOP states that every piece of code must fall inside a class, and to use any piece of code an object of the class has to be created, and the code can only be invoked via the object (except for 1 exception of static code, which is invoked by the class itself)

```
class Wolverine {
	private String[] superPowers = { "Regeneration", "Strength",
			"AdamantiumClaws" };

	public void showPowers() {
		for (String power : superPowers)
			System.out.println(power);
	}
}


class Game {
	public static void main(String[] args) {
		// Use Wolverine class's code here
		Wolverine wolverine = new Wolverine();
		wolverine.showPowers();
	}
}
```

Inheritance –

Inheritance is the OOP principle where any class can extend one class as its parent. Inheritance enables code reuse in our programs, where any child class gets all the extendable code from its parent, exactly similar how the word inheritance itself works.

> ***Note:*** *in java we can extend only one class, that means we cannot have multiple parents of a class; hence Multiple Inheritance is not supported in Java.*

```
class SuperGeek {
	public final int iqLevel = 100;
	public String[] degrees = { "M.S.", "Ph.D." };
	private String something = "some other property"; // private field is not
                                                      // extended
	public void talkGeekyCrap() {
	}
}


class SheldonCooper extends SuperGeek {
	// Sheldon automatically gets the iqLevel 100 and all the degrees
	// and he can talk all geeky crap too

	/**
	 * Sheldon's private friends, won't be shared with any class extending
	 * SheldonCooper
	 * */
	private String[] friends = { "Leonard", "Rajesh", "Howard", "Penny" };

	private void buildRocket() {
	}
}
```

Polymorphism –

The concept of polymorphism tells about multiple forms a particular code can take. Its like one thing taking multiple forms, capable of doing multiple tasks. There are two kinds of Polymorphism supported by java, Static Polymorphism (Compile time polymorphism) and Dynamic Polymorphism (Runtime polymorphism). Method Overloading is an example of compile time polymorphism; whereas Method Overriding is an example of runtime polymorphism in java.

```
/**  Class showing Method Overloading
 *   Static/Compile time Polymorphism
 **/

class SuperHero {
	public void fly() {
	}

	Public void fly(long height) {
	}
}
```

```
/** Class showing Method Overriding
 *  Dynamic/Runtime Polymorphism
 **/

class SuperMan extends SuperHero {
      public void fly(long height) {
            // fly faster with superspeed
      }
}
```

Abstraction, Encapsulation, Data Hiding

Friends this question seems very innocent and we all have read this question in the first few classes of our first year engineering perhaps, but trust me if an interviewer decides to confuse you he can scare the hell out of you on this puny little question. The topics are so closely related that any term can be confused with the other two, so let's be confident on our fundamentals before we deep dive into other questions.

Data Hiding –

Data Hiding is a proposal, asking us to hide as much vital things as possible, and to expose/reveal only the most necessary things to the outer world. Data hiding does justice to its name, and only talks about hiding the data, not on how you're going to implement it.

Abstraction –

Now let's talk about Abstraction; Abstraction is an implementation model over Data Hiding. It's a technique for achieving Information Hiding. Abstraction says about extracting out the complex details and exposing out the necessary details (i.e. what Data Hiding proposes). Abstraction therefore gives a blue print of an idea, and concrete implementations would later be built over it. Example of Abstraction is Java's Abstract class, where we only give the blue print of what is in our mind, without the actual code. The code is later build on top of it.

Below is an example of an Abstract class providing only the most general definitions without talking anything about how are we ever going to achieve it via code:

```
public abstract class FlyingCar {

      public abstract void useAsCar();
      public abstract void useAsHoverCraft();

      public abstract void fly();
      public abstract void land();
      public abstract void drive();

}
```

Encapsulation –

Encapsulation is simple, as simple as its name is. Encapsulation simply says about putting all things in a capsule, exactly what we do while creating Java classes. People also tend to confuse Encapsulation with Data Hiding. Well everything that is encapsulated may not be hidden. But yes encapsulation can be used to achieve Data Hiding also, by adding private methods or fields etc., but that is an additional step taken by the programmer, Encapsulation does not demand data hiding from you. Encapsulation only proposes to keep similar things together in a capsule. Below is a simple example of encapsulation:

```
class FacebookWall{

	String userName;
	Image profilePic;
	List<String> friends;

	public void postStatusUpdate(String message){
		// code for posting status falls here.
	}

	public void poke(String friendUserId){
		// code for poking friend here.
	}

}
```

Relationships –

IS-A Relationship:

IS-A is denotes an Inheritance relation between two entities. IS-A is used to specify a more specific type of the entity. Eg. Dog IS-A Animal, or SuperMan IS-A SuperHero. Where the parent classes Animal and SuperHero are more general, we have more specific subclasses Dog and Superman. The concept of IS-A is all about Inheritance.

HAS-A Relationship:

The HAS-A Relationship tells about containing some entity inside another entity. Its like Zoo HAS Animal. The Animal can be considered an instance variable inside the Zoo class. The HAS-A Relationship is also known as the Aggregation/Composition relation.

USES-A Relationship:
USES-A is a simple relationship where one entity is not related to the other entity but just wants to use it for some purpose. Its more like you using a second class inside your first class's code. The USES-A Relationship is also known as the Association relation.

```
class Users{

}

class FaceBookWall{

	public void postOnWall(String message){

	}
}

class SocialNetwork{

}

class FaceBook extends SocialNetwork{

	List<Users> millionUsers;

	public void updateStatus(){
		FaceBookWall wall = new FaceBookWall();
		wall.postOnWall("I Hate Weekdays");
	}
}
```

Let's sync our understanding with the above code :
FaceBook IS-A SocialNetwork website.(Inheritance).
FaceBook HAS Users(actually Million users).(Aggregation/Composition).
FaceBook USES-A FaceBookWall to post any status updates.(Association).

On Your Code, Get Set Go

This Section of the book contains all the random frequently asked questions, and probable 60% of your questions would fall from this section unless you are going for a company which actually works on java internals etc.

The questions in this section are of intermediate level and you must be very thorough with almost all of these questions. Also not to mention there can be many more questions so emphasize on the concepts instead of trying to memorize the code.

Do not be scared with the quantity of questions in the section, you might actually be knowing the answers to most of the questions in the section already, so you just have to skim through the questions in that case. But, if the questions appear new or different please try the Code Snippet given with the explanations. Typing the code might take a few minutes but you would really be able to understand how the code behaves.

We cannot learn Coding until we Code. Let's take this lil pain..

What are the differences b/w Interface and Abstract class?

A Point blank answer can be -: An Interface is a 100% Abstract class.
Now let's dive in to the individual properties of interfaces and abstract classes.

Abstract class:
An abstract class is a class which may have some unimplemented methods along with some implemented methods. Yes it may or may not have any implemented/un-implemented methods.
An abstract class can have non-final instance variables, and can be extended by any concrete or abstract class. The extending class must implement all the un-implemented methods or has to be an abstract class itself. Abstract class can have constructors, but they cannot be directly instantiated.

Interfaces:
An interface cannot have any implemented methods, only method signature falls in interface. Any method in interface is by default *Public* + *Abstract*. Interface cannot have a private method. Any instance variable in interface has to be final. Interfaces can neither be instantiated, nor can they have constructors.

Explain Synchronization and Multithreading.

Traditionally all our programs run only on a single thread (main thread) and the thread completes when the sequential execution of code is over. Java permits Multi-Threading, where we can have more than one threads, each executing an independent module of its own to gain parallelization in the code, making it efficient.

Then comes the problem of shared resource. When we have multiple threads in our program, and we have a particular variable or piece of code which works on a sensitive data. And different threads may access the code at a same time, therefore interfering with each other's operation, and giving dirty values to the program. Java has a *synchronized keyword* and a *synchronized block* to avoid this issue.

Any method marked synchronized becomes thread safe and can be accessed only by one thread at a time; similarly any block of code inside the synchronized block becomes thread safe and only one thread can be inside that block at a time.

```
/** A Synchronized Method **/
public synchronized void updateBankBalance(){
            // some code here for bank balance updation

      }

/** A Method with Synchronized Block **/
public void updateBalance(){

      /** A Synchronized Block **/
      synchronized(this) {
                  // code falls here for bank balance updation

                  }
      }
```

Have you worked on Threads? How do you create Threads?

Threads are integral part of many applications which seek better performance and are extensively used in the IT industry. People would definitely need you to know the In-Outs of threads, if not at least the basics in place.
These are the steps you need to follow to make your class Threaded and run your thread :

1. Making Your Class Threaded

This can be done by two ways, either by extending the Thread class or by implementing the Runnable Interface. Below is a short note on both the approaches:

Extending the thread class:

One way of making you class a threaded class is extending the Thread class.

Extending Runnable Interface :

This is another way of making your class threaded. This approach gains more popularity over the above because in java we do not have multiple inheritance, and therefore if we extend Thread class we won't be able to extend any other parent class from our class. Implementing Runnable interface on the other hand does not stop us from extending an parent class, hence gives a programming advantage over Thread Class.

2. Implement the run method

As the rule of un-implemented methods (from Abstract class & Interfaces) we must implement the run method of the Thread class/ Runnable interface. This is the primary method which

would contain the code which you want your thread to execute. Anything that comes into this method is run as a new thread by the JVM.

3. **Create a thread object and start the new thread**

Now since we have our thread class ready, we can create an object of the class from anywhere, within the class or outside in another class, and just have to trigger the start() method to start our thread.

> ***Note:***
> *It is important to use the threadObj.start() method rather than using the threadObj.run() method. The run method is internally called by the JVM to create a thread and execute the code you provided, directly calling the run method executes you code sequentially as any ordinary method would do. It's the start method that is the lead actor of the play, taking the responsibility of creating a thread and calling run method from that new thread independently.*

> ***Related question:***
>
> *Q: Which is better, extending the Thread class or implementing the Runnable Interface?*
> *Q: Why do we need to call the start() method of the thread, why don't we call the run() method directly ?*

Code:

I am making a Game, not something like CounterStrike, but a very very naïve one. I have my player shooting Terrorists, and two other threads run in parallel in the Background, one which shows you the map on the screen, and other that shows you player's health. Here is a skeleton showing Threads in Action.

```
class GameMapThread implements Runnable{

            public void run(){
            // code for showing game map on screen

            }
      }


class PlayerHealthThread implements Runnable{
```

```
            public void run(){
            //code for showing player's health on screen

            }
      }


class MyGameArena{

      public void playGame(){

            /** Create Threads for the parallel operations MAP & Health **/
            Thread mapThread = new Thread(new GameMapThread());
            Thread healthThread = new Thread(new PlayerHealthThread());

            /** This thread will show you the
                map and player's location on the screen */
            mapThread.start();

            /** This thread will continuously show your
                player's health on screen */
            healthThread.start();


            // actual code for playing game falls here

            }
      }
```

Differentiate between pass by reference and pass by value?

This is generally a C/C++ question but few interviewers still seem attached to this question and tend to ask the question to Java guys as well. Pass by value and Pass by reference is an age old programming concept, where Pass by value passes only the value of a variable in a method call, where pass by reference passes the address of the variable (via pointers) to the method, hence any changes to the variable in the called function actually reflects the change in the called function.

Fortunately Java only supports Pass by Value since java doesn't deal with Addresses and pointer stuffs.

Next Question:

Q: Is Java Pass by Value or Pass by Reference?
Ans: Java is Pass by Value, and does not support Pass by Reference.

Related Questions:

Q: Primitive data types are passed by reference or pass by value?
Ans: Everything in java is passed by Value.

Q: Objects are passed by value or by reference?
Ans: Pass By Value, friends.

Differentiate between HashMap and Map?

Map is the parent Interface of any class implementing the Map Interface (like HashMap /LinkedHashMap /TreeMap etc). HashMap is a particular implementation of Map (HashMap IS-A Map), and stores everything in a Key-Value pair. HashMap itself is an unordered collection, and does not follow the order of insertion. HashMap internally uses the hashing algorithm and buckets to put objects in memory, and gives an O(1) time complexity for insertion and retrieval of values, hence has an clear advantage over the other Data Structures.

```
public void someMethod()
{

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

	// Adding parameters to map
	myMap.put("os_name", "MS Windows");
	myMap.put("os_version", "Windows 7");

	// Getting parameters from the map
	String osName = myMap.get("os_name");
	System.out.println("OS Name:"+osName);


}
```

Differences between HashMap and HashTable?

HashMap and HashTable are both very similar Data structures that save the data in <Key, Value>, with these few differences:

HashMap	HashTable
Non Synchronized, thread unsafe. Preferred when we are not dealing with threaded scenarios. An equivalent thread safe map can be created by :Collections.synchronizeMap(myMap);	Thread Safe and Synchronized. Performance trade-off due to synchronization overhead.
NULL values are allowed in HashMap, 1 time as Key and any number of times as Value.	No NULL values are allowed in HashTable.
No guarantee over the order of insertion in HashMap.	All the data is kept in the order of insertion in HashTables.
Suggested by Java.	Deprecated.

Differentiate between Vector and ArrayList?

The Vector class was deprecated by Java when they introduced the ArrayList class. The differences between Vector and ArrayList are:

Vector	ArrayList
Deprecated Class.	New implementation provided by Java.
Thread safe and Synchronized.	Un-Synchronized, thread unsafe.
There is a little performance trade-off with vectors due to the synchronization overhead.	Better performance than Vectors.

Differentiate between Map and Set?

The main difference between Map and Set is that Map contains data in <Key, Value> pair whereas the Set is a unique collection of values (Objects). Set holds unique values, whereas the Map holds unique Keys and can have any amount of duplicate Values assigned to the unique keys. Maps are part of java collection framework but do not extend the Collection Interface, Set is a part of collection framework and also extends Collection Interface.

Differentiate between AWT and Swing?

AWT and Swing are both used to develop GUI for Java Applications, while AWT components were heavy; Java introduced Swing as lighter and better performing APIs. Advantages of Swing over AWT were that it was lighter and faster, and more developed than AWT.

What is the difference between a constructor and any ordinary method?

Constructor is a special method that is used to initialize objects before we start using them. A constructor does not have a return type and has the same name as the class name itself. Java provides a default constructor to every class unless the programmer has defined any constructor himself. The default constructor enables us to create objects of our class by the line of code:

```
      // new MyClass() is the call to the default constructor.
      MyClass myObj = new MyClass();
```

If the coder has provided a parameterized constructor without providing a default constructor, we won't be able to use the above line of code for creating objects, because in this case Java won't be providing us the default constructor.

```
class MyClass {

      /** Parameterized constructor */
      public MyClass(int parameter){

      }
}

class TestClass{
      public static void main(String[] a){

            /** This line would give compilation error **/
            MyClass obj1 = new MyClass();

            /** This works just fine **/
            MyClass obj2 = new MyClass(10);

            }
      }
```

What is an Iterator?

Iterator is just another java object that lets you iterate over an java collection. Consider it as another way to scan over any java collection like Array/ArrayList/LinkedList etc. Yes you can always use our favorite foreach loop to scan over the collection, but sometimes we may need to modify the collection while in iteration, here the iterator comes to our rescue. We can remove the element from the iterator while being inside the iteration (iterators are fail fast), the foreach loop would blow off in this situation with a ConcurrentModificationException.

```
public void someMethod()
{
	ArrayList<String> myCollection = new ArrayList<String>();
	myCollection.add("Some Value");
	myCollection.add("Another Value");
	myCollection.add("Third Value");


	for(String item : myCollection){

		if (item.equals("Third Value")) {
		/** we'll get ConcurrentModificationException here  **/
		myCollection.remove(item);
		}
	}
	System.out.println(myCollection);

	Iterator iterator = myCollection.iterator();
	while (iterator.hasNext()) {
		String item = (String) iterator.next();
		if (item.equals("Third Value")) {

		/** note: its iterator.remove() and
		    not myCollection.remove() **/

		iterator.remove();
		}
	}
	System.out.println(myCollection);
}
```

Explain public, private, protected, default modifiers.

These are the 4 access modifiers/specifiers provided by java varying with the relaxation in their respective accessibility. Again we must be clear that there are two obvious ways for accessing

any instance variable/method in java, either by creating an object of the class, or by extending the class where you inherit all the properties/methods of the class directly. Again there is a third way with static methods and static variables that can be accessed directly by the class name.
Now discussing about the access modifiers:

Public : any instance variable/method marked public would be visible to any other program in your application. You just create an object of the class or inherit it, it's accessible for your use. This is the most relaxed access modifier.

Private: this is the most restricted access modifier and any instance variable/method marked private becomes a sole property of that class only. It is not available to any class extending it, nor can be accessed by objects of the class.

Protected : protected fields are accessible via objects/inheritance inside the same package. Outside the package protected access specified properties are only accessible via Inheritance.

Default : This is another restrictive access modifier. any property marked default is only accessible inside the package. There is no way to access a default property outside the package.

We insist you to type down the code below, and play with it to get your concepts in place. Typing just two Java classes should not be a very difficult task.

```
package somepackage;

public class AccessSpecifiers {

	// Accessible everywhere outside/inside class
	public String publicString = "publicString";

	// UnAccessible everywhere outside class
	private String privateString = "privateString";

	// Accessible via objects/inheritance inside same package,
	// Accessible outside only via Inheritance
	protected String protectedString = "protectedString";

	// Accessible only inside same package
	String defaultString = "defaultString";


}

class TestClassInsideSamePackage{

	public void readAll(){
		/** Accessed via object of the class */
		AccessSpecifiers access = new AccessSpecifiers();
```

```
            System.out.println(access.privateString); // Un-Accessible
            System.out.println(access.defaultString);
            System.out.println(access.protectedString);
            System.out.println(access.publicString);
      }
}
```

```
package anotherPackage;
import somepackage.AccessSpecifiers;


/** This class uses normal object creation to test access specifiers */
class TestClass{

      public void readAll()
      {
            /** Accessed via object of the class */
            AccessSpecifiers access = new AccessSpecifiers();
            System.out.println(access.privateString);   // Unaccessible
            System.out.println(access.defaultString);   // Unaccessible
            System.out.println(access.protectedString); // Unaccessible
            System.out.println(access.publicString);
      }
}

/** This class uses inheritance to test access specifiers */
class TestClassExtended extends AccessSpecifiers{

      public void readAll()
      {
            /** Accessed via object of the class */
            AccessSpecifiers access = new AccessSpecifiers();
            System.out.println(access.privateString);   // Unaccessible
            System.out.println(access.defaultString);   // Unaccessible
            System.out.println(access.protectedString); // Unaccessible
            System.out.println(access.publicString);

            /** Accessed via 'this' - class's own reference */
            System.out.println(this.privateString);     // Unaccessible
            System.out.println(this.defaultString);     // Unaccessible
            System.out.println(this.protectedString);
            System.out.println(this.publicString);
      }
}
```

What is the significance of a static modifier?

On a direct note: static means **one per class**.
Let's understand the statement now:
What happens anytime we create an object of the class, the JVM would create an object in the Heap memory, pack it up with the properties/variables defined in the class, provide it the power of the functions you declared(loaded on time of use). Everything is packed inside that little object in the Heap, and if we have 10 such objects created, each of these little objects would have their own properties and methods working on those properties, such that on modifying any property of the first object would have no effect on the other 9 objects. So basically we have 10 people with their own data, and their own powers.
Now say we need to have a data that needs to be shared across these 10 people, what do we do?
We make that variable static; which would ensure that the property will not be packed inside the Objects on Heap, rather it would be a class's data and any changes on the data by any place in your program will be on that common data element.

> *In case you might be wondering where the static data goes, I'll say the JVM actually takes care of it and it does it pretty well. Just for the information, the static data lands up in the Heap only – a special heap area known as perm heap (or permanent generation), but not inside an object or the Heap where objects are present. This area contains all static data of any class along with other metadata information. We would be talking a little over this in the third section of the book in the Java Memory Allocation question.*
> ***Greedy for learning more?** : Go on, do some surfing over it, you might learn something totally awesome.*

Different usage of static key word:

Static variable:
One per class, shared variable. Can be accessed without creating class's object by 'ClassName.variableName' directly.

Static Method:
Belongs to class, similar to static variable. It can access only static data and static methods directly. Else we need to create object of the class and call the methods from the object (exactly what we do in our main method most of the times).A static method cannot use the *this* and *super* keyword inside itself. It can also be accessed via the ClassName.methodName() directly.

Static Block:

Your program can also have an interesting thing known as the static block. A static block is a piece of code which you want to execute before any part of your class is used. A static blocks is the 1st thing that would be executed by the JVM, even before the constructor. The static block would be executed only once. It's generally used for initializing variables before the objects can be initialized.

Open your IDE and try this piece of code instantly and observe the order of execution:

```
class StaticTest {

	/** A static variable */
	private static int num;

	/** A static block */
	static
	{
	num = 20;
	System.out.println("Inside static block");
	}

	/** Constructor */
	public StaticTest(){
		System.out.println("Inside Constructor");
	}


	public static void myStaticMethod(){
		System.out.println("Inside static func");
	}

	public void myNonStaticMethod(){
		System.out.println("Inside non static func");
	}


	public static void main(String[] args){
		StaticTest.myStaticMethod();
		StaticTest test = new StaticTest();
		test.myNonStaticMethod();
	}
}
```

What is a final modifier?

Final is exactly what its name says. Anything that is marked final in java becomes final i.e. committed. A variable marked final cannot be changed by re-assigning any value to it again. Any

method marked as final cannot be overridden by any class extending the class. And any class marked final cannot be extended by any other class. Final may be used by programmers who believe in distributing their class files to people but won't like people messing up with their code, so they just mark it as final and people can use the libraries flawlessly without altering his code.

Can main method be declared as private?

Why not, there is no compilation error for sure if my main method is private; but then I am not able to run my code. Wondering why?

Let's just understand how our program gets called. Who calls our program.. Etc etc..
The **JVM** is the person responsible for calling your code, and it does not know what all methods you've declared. So it requests you – please provide me a main method which I can call. I would be treating this method as the entry point to your program. JVM picks up all the command line parameters, constructs a String array out of the parameters, and calls the static method 'main' passing all the arguments to that method as a string array.

Now, How does the JVM make the method call?
Simple, it just uses the ClassName.main(String [] arguments) method call, since main is a static method, the JVM doesn't need to create any object to call the method, it just has to call it by the ClassName.methodName syntax.

So understanding this, we can now think that if the main is not public, how would it be visible to the JVM. So the non-public main method would be the class's personal property, JVM would never be able to see the method, and being unable to find the method it may just fail with NoSuchMethodError politely.

Related Questions :

Q: What if the static modifier is removed from the main method signature ?
Ans: Again JVM won't be able to call the method as ClassName.main(String[] arguments) for a non static method.

Q: What if I do not provide the String array as the argument to the method ?
Ans: VM would again Fail, not being able to find the correct method.

Q: What if I write static public void main, instead of public static void main ?
Ans: Its just fine. No issues here.

Q: What is the first argument of the String array in main method ? Is it the name of the program itself (as in C/C++) ?
Ans: No unlike C/C++ java does not hold the program name in the 'args' array. It just holds the parameters provided to the program.

Q: If I do not provide any arguments on the command line, then the String array of Main method will be empty or null ?
Ans: It would be an empty array, NOT NULL.

Q: How can we check if the argument array is empty or NULL ?
Ans : We can check it programmatically:

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

	/** check by preventing null array usage  */
	if(null == args){
		System.out.println("Null array");
	}


	/** Check by handling the null array usage */
	try{
		System.out.println(args[0]);
	}
	catch(NullPointerException ex){
		System.out.println("Null array");
	}
}
```

Can an application have more than one classes having main method ?

Why not, just while executing you must specify the entry point class name, where it should look for the main method. Again, we can also have multiple main methods in a single class as well, as long as their method signatures are different as per the method overloading standards, but JVM still would call the 'public static main(String[] args) method'.
Generally it makes more sense to have only one main, since there should logically one entry point only in the applications. So in case of multiple main application, while bundling the application the entry point has to be specified which the JVM would call to use the application.

Generally you would distribute your application as a JAR, and there you have to mention which class is going to be the entry point for the app. So even if you have multiple classes with multiple mains, The JVM knows which class to call.

Related Question :

Q: Can I have multiple main methods in the same class ?
Ans: Yes, of course, all you are trying to do is overload the main method, so its all valid as long as they stand upto method overloading rules.

Do we have to import java.lang package in our code?

No, java.lang can be considered pre-imported into every java class. You can assume JVM does that for us.

Can we import same package/class twice?

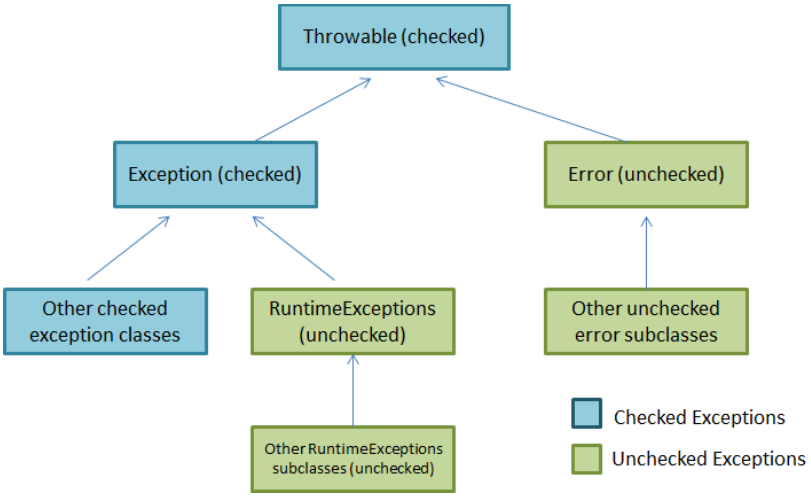
Yes, a package can be imported N- number of times without any issue. Unlike C/C++, JVM never loads any package into your code while compiling the code. Instead package import is just a qualified name of the class, i.e. you're using 'import java.lang.Math' just to use the Math.random() function directly (instead of using full name : java.lang.Math.randon()). If you can write the absolute package qualified name everywhere in your code you'll never need to import java.lang.Math class.
On runtime the java compiler would simply replace your 'Math.random()' call to the fully qualified name 'java.lang.Math.random()'. That is also why it's totally fine to have an import any number of time.

What are Checked and Un-Checked Exception?

To have the programmer a little more cautious and to avoid some most intuitive programming errors, Java has got one of the perfect Exception Handling API's and a compile time code check to ensure that the programmer has handled at least the most common programming exceptions. The exceptions are broadly classified into two types:

Checked Exceptions:
All classes which are subclass of the class Exception are checked exceptions, except for a set of classes (Runtime exceptions & Subsets). In general these are the exceptions which the programmer has to handle in his code because they are the ones very liable to occur. Examples are SQLException, IOException etc.

UnChecked Exceptions:
The classes RuntimeException & Error itself , and any class that extends RuntimeException or Error class fall under the Unchecked Exceptions category. This is a relaxation from Java, since these arise only because of the programming logic issue or system errors these are not mandatory to be handled or are un-recoverable (and therefore unhand able). Some examples of unchecked exceptions are ArrayIndexOutOfBounds, DivisionByZero, NullPointerException, OutOfMemoryError etc.
The diagram below must buy you some more clarity:



Related Questions :

Q: What are the differences between Error and Exceptions ?
Ans: Errors unlike exceptions are generally unrecoverable. Exceptions are both Checked and Unchecked, whereas all the Errors are Unchecked.

Q: Name an unrecoverable Error ?
Ans: OutOfMemoryError.

Q: Does it mean that we cannot catch Errors ?
Ans: Yes we can catch them, but still there would be no guarantee of the code being stable, since Errors are mostly Unrecoverable.

What is Dynamic Method Dispatching?

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at Run Time, rather than at Compile Time. Dynamic method dispatch is important because this is how Java implements Runtime Polymorphism. Method Overriding is a Runtime Polymorphism mechanism in java, where you can override the definition of any method in the parent class and provide your own new definition for the method.
Overriding is also known as Dynamic Method Invocation or Runtime Polymorphism, because the JVM itself does not know which method would be called until its executing the code at runtime.

```
class ParentClass {

      public void someMethod(){
            //  This method is not cool,
            //  needs to be overridden with a cooler version
            }
      }

class ChildClass extends ParentClass{

      public void someMethod(){
            // put your cool code here,
            // now this code would be called when
            // ChildClass's object would be used.
            }
      }
```

Are the imports checked for validity at compile time?

Yes the imports are validated at the Compile time, and any wrong import in your code won't let it compile. It would give you the unresolved symbol compilation error.

Can a class be declared private or protected?

No a class cannot be private or protected. Only access modifiers allowed with a class are public, default, abstract or final.

What is serialization?

Certain times we might just have to save the state of an object in our program. It can be for that cool game we're developing, where we simply want to save the level/stage where the user leaves the game, so that next time he comes to play the game we can simply load the object back in the memory and start the game from the exact point he left. Or sometimes we might have to send objects over the network.

This process of saving the state of the object on files/database/anywhere is known as Serialization. Serialization has a pretty simple syntax and all you need to do is to make your class Serializable.

We can make our class serializable by simply implementing the Serializable interface. *Serializable is a Marker Interface* i.e. it does not have anything inside it, and so you do not have to override any method etc.

Once your class is made serializable it can be flattened/written to files etc.

Here is the code for serializing/deserializing an Object for our Game class :

```
class GamePlayer implements Serializable {

	int gameLevel;
	int health;
	ArrayList<String> powers = new ArrayList<String>();


	public static void serialize(GamePlayer player) throws IOException{
		FileOutputStream fos = new FileOutputStream("serial");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(player);
		oos.flush();
		oos.close();
	}


	public static GamePlayer deserialize()
		throws IOException, ClassNotFoundException{
		GamePlayer player;
		FileInputStream fis = new FileInputStream("serial");
		ObjectInputStream ois = new ObjectInputStream(fis);
		player = (GamePlayer)ois.readObject();
		ois.close();
```

```
            return player;
      }


      /** toString() is overridden just to see the object
       *  in a readable format when its put in a Sysout statement,
       *  rather than the default hashcode,
       **/
      public String toString(){
            return"PlayerLevel:"+this.gameLevel+", PlayerHealth:"+this.health;
      }
}


class Game{

      publicstaticvoid main(String[] args)
            throws IOException, ClassNotFoundException{

            GamePlayer player = newGamePlayer();
            player.gameLevel = 7;
            player.health = 50;
            player.powers.add("MediPack");
            player.powers.add("LaserGun");
            player.powers.add("FireBall");

            /**  Write Object to file **/
            GamePlayer.serialize(player);

            /** Get the Object back from file **/
            GamePlayer newPlayer = GamePlayer.deserialize();
            System.out.println(newPlayer);

      }
}
```

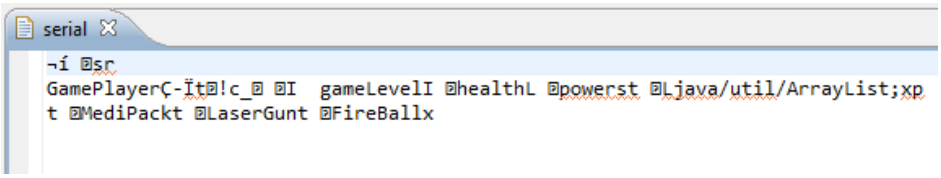
Note : In the above code we have provided two utility methods serialize/deserialize inside the GamePlayer class. It doesn't need to be in the class, its just to keep the code compact. It is just a piece of code to read/write into files and can be present in any class you want. Typically we use a third utility class for read/write so that it can be used by other parts of your app too.

Some imp points about serialization:

- *Only objects whose classes are made serializable can be serialized.*
- *Static fields are not serializable*

- *Volatile fields are not serialized*
- *Transient fields are not serialized*
- *All internal objects in your Serializable object are also serialized recursively with the main object iff the inner objects are themselves serializable(else you get NotSerializable exception). The inner objects must also implement the Serializable interface for being serialized along with the main object.*

A sample serialized file looks like this :



What are inner/nested classes ?

Personally I myself am not very keen in using the Java Inner classes frequently, but since we are talking about the interviews here, we simply cannot ignore this cute little concept. So let's have a sweet little talk over it.

Inner classes, also known as the Nested Classes, are nothing but classes enclosed inside some other parents class. Now the class can be present in two places in a parent class :

- In the class itself as instances are declared.
- Inside some method of the parent class.

Based on their presence location and access, there are 4 types of inner classes:

- Inner classes (Normal)
- Static Inner classes
- Local Inner classes
- Anonymous Inner classes.

While the first two fall in the 1st category (declared directly inside class) , the other two fall in the 2nd type (declared inside some method).

Now let's read what makes Inner classes so special :

The inner class can access all the attributes of its parent class, even the private methods and variables. And similar is true for Parent class. The parent class can also access private fields of the inner class.

Syntax for Parent class accessing Inner class's private fields :

```
InnerClass inner = newInnerClass();
inner.privateMethod();
```

Syntax for Inner class accessing Parent class's private fields :

```
ParentClass.this.privateMethod();
```

With this basic understanding let's move back to our IDE's to get a Hands-On on Inner classes:

The below code shows a Normal Inner class – InnerProgrammer, defined in the Parent class Programmer. We can see how both the classes can access the private fields and methods of each other.

```
/** Programmer : The Outer Class **/
class Programmer{

	private String name;

	privatevoid someOuterFunction(){
		System.out.println("Inside private method of Outer class");
		InnerProgrammer inner = new InnerProgrammer();
		inner.language = "New Language";
		inner.someInnerFunction();
	}


	/** InnerProgrammer : The Inner Class **/
	class InnerProgrammer{

		private String language;
		private int age;

		private void someInnerFunction(){
			System.out.println("Inside private method of Inner class");
			Programmer.this.name = "New Name";
			Programmer.this.someOuterFunction();
		}
	}
	/** End of Inner Class **/
```

```
}


/** USAGE **/
class Test{
      public static void main(String[] args){
            Programmer.InnerProgrammer inner
                              = (new Programmer()).new InnerProgrammer();
    }
}
```

Note: For static inner class we just have to add the static keyword in front of the inner class and then the inner class would be able to access all static variables/method of the Parent class. But for the non static ones we would have to instantiate the Parent class and use the variables/methods via that object. Just try out the static class.

```
ParentClass.this.privateMethod();
```

Would be changed to :

```
ParentClass parent = new ParentClass();
Parent.privateMethod();
```

Below is the code for Local Inner classes and Anonymous Inner classes. Both of the classes are declared inside a method someMethod().

Anonymous Inner classes are special type of Local Inner class where we don't even create a class definition, but instead write the code of class on the go. Anonymous classes are generally used for single-use scenarios, so we do not bother creating the class formally.

But, creating anonymous inner class requires an interface or super class structure before we can use it.

Below is the complete code, drill it out :

```
interface Programmer{
      public void writeCode();
      public void drinkCoffee();
}
```

```
class ParentClass{
	private String name="Bond";

	public void someMethod(){


		class LocalInnerClass{
			private void doSomething(){
			System.out.println("Lol, I can see ur private fields : "
									+ParentClass.this.name);
			}

		}

		/** Anonymous Class **/
		new Programmer(){

			@Override
			public void writeCode() {
			System.out.println("Lol, I can see ur private fields : "
									+ParentClass.this.name);
				System.out.println(ParentClass.this.name
									+" is coding hard");
			}

			@Override
			public void drinkCoffee() {
			System.out.println("Lol, I can see ur private fields : "
									+ParentClass.this.name);
				System.out.println(ParentClass.this.name
									+" is drinking coffee");
			}};
		/** End of Anonymous Class **/

	}
}
```

How to find a method's total execution time?

Java has got a method that returns the system time in milliseconds. We can use the method to get the time before and after the execution of the method in milliseconds. The difference of the two times can be used to calculate the time taken by the method.

Here is the code:

```
class TimeTestClass {

	public static void someMethod(){
		for(int count=0; count<999999999; count++);
	}
```

```
	public static void main(String[] args)
	{
		long starttime = System.currentTimeMillis();
		someMethod();
		long endtime = System.currentTimeMillis();

		System.out.println ("Execution Time : "
					+ (endtime - starttime)+" millisecs");
	}
}
```

What are wrapper classes?

Since Java deals with both the primitives as well as objects, Java has provided special class for each primitive. Like the Integer class for int, Float class for float etc. Java does this so that the coder has a choice to select any one over them based on his requirement.

More ever, sometimes it makes more sense to deal with objects rather than primitives, like while using an ArrayList you need to add Integer objects to the list, you cannot add primitives to the list. So Java lets you add numbers to list, where it internally converts your int primitive value to an Integer object and adds it to the collection.

> *This process by which java itself handles the conversion from primitives to Objects & vice versa is known a Auto-boxing/Un-boxing. This was a new feature introduced in Java 5.*

How to create a Custom Exception class?

Simply extend the Exception class or any other sub class of the class Exception, and now you are ready to throw any exception of your own class from your code. For passing message along with your Exception class, you can simply add a constructor that takes in a parameter and passes it to its super class's constructor by the super() call.

Below I the code for Custom Exception class:

```
class MyException extends Exception  {

	public MyException(){
		super();
```

```
	}

	public MyException(String message){
		super(message);
	}

}

public class TestClass {
	public static void main(String[] a) throws MyException{
		throw new MyException("I dont like ur name");
	}
}
```

> ***Note :*** *If your class is already extending some class, and since java does not support multiple inheritance, you cannot extend the Exception class in your class, hence it cannot be used as a custom exception class.*

What are the different ways to handle exceptions?

An exception can be handled by 2 ways :

- **Try-Catch-Finally block :** Where the coder handles the exception himself.
- **Throws clause :**Where the coder postpones the exception handling, leaving the exception handling over the next person who uses this function/code.

Is it necessary that each try block must be followed by a catch block ?

No, in a try-catch-block structure any one among the catch and finally block is required. We can miss either one of them but cannot miss both of them. And that makes sense also, if we don't want any exception handling (neither catched nor any finalizing code) then what was the need of the try block in the first place.

Will the finally block still execute if we write return at the end of the try block ?

Yes the finally block would still execute even if there is a return statement in the try block. Only a System.exit(0) can stop the execution of the finally block.

Below is another interesting code snippet with 2 return statements, 1 in try block, and one in finally block. And we're sure that the finally block would still execute even if there is a return statement in the try block.

So now, what value do you think would be finally returned?

```
public class TryCatchFinallyTest {

	public static int testMethod()
	{
		try
		{
			return 1;
		}
		finally
		{
			return 2;
		}
	}

	public static void main(String[] args){
		System.out.println(testMethod());
	}
}
```

How to read configuration/Properties files.

First you might ask, who needs a property file. Well we all do, and once you're into regular development you'll understand that it really gets very difficult to hard code text values into you code, and when one text message has to be changed you've gotto hunt for all the locations where you've put the code, and next few minutes you busy replacing all the text with the new text.

So we put all our stuffs into a property file, like constants, file paths, configurations, text messages for UI etc, and we just pick them up from the properties file whenever needed, and so everything is at on location and we're not bothered next time when some configuration or message has changed.

Get your IDE out and hit this small piece of code :

> *Be sure to put a flie 'PropertyFile.property' in your folder, and change the path of the File in the program according to the location where you've put your property file.*

A property file is a very simple 'key=value' pair file and looks like this :

```
name=Yash
group=Confused
languages=English,Java,Crap

```

Here is your code, Drill it:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Properties;

public class LoadResourceBundle {

	public static void main(String[] args) {
		// This print statement is just to see the directory
		// where the program is executing,
		// You can remove this.
		System.out.println("Present working dir: "
				+ new File(".").getAbsolutePath());



		try {

			// Put appropriate file path here
			File file
			= new File("..\\DemoCodes\\src\\PropertyFile.property");

			FileInputStream fileInput = new FileInputStream(file);

			Properties properties = new Properties();
			properties.load(fileInput);
			fileInput.close();



			/** Iterating over all the properties */

			Enumeration enuKeys = properties.keys();

			while (enuKeys.hasMoreElements()) {
				String key = (String) enuKeys.nextElement();
```

```
                        String value = properties.getProperty(key);
                        System.out.println(key + ": " + value);
                  }

                  /** Fetching values directly when keys are known */
                  System.out.println(properties.getProperty("name"));
                  System.out.println(properties.getProperty("languages"));

            }
            catch (FileNotFoundException e) {
                  e.printStackTrace();
            }
            catch (IOException e) {
                  e.printStackTrace();

            }

      }

}
```

Difference between equals() and == operator

The equals() method compares the value of the two objects, whereas the == operator compares the references of the two objects, i.e even if two objects are having the same value, the == operator would return a false.
Don't hesitate, get your IDE out and hit the code right away, and tell me the output, I'll wait ..

```
public void someMethod() {

            String timon = "Hakuna Matata";
            String pumbaa = "Hakuna Matata";
            String newString = new String("Hakuna Matata");

            System.out.println(timon.equals(pumbaa));
            System.out.println(timon.equals(newString));
            System.out.println(timon == pumbaa);
            System.out.println(timon == newString);
      }
```

If you tried the code, you'd had figured another interesting property of String literals, the String Pooling. The str1 and str2 both have same reference, hence there is only one object on the heap for both str1 and str2. str3 on the other hand explicitly creates a new String object, hence it has a different reference and hence fails the str1==str3 check.

Write a sweet little JDBC code?

Here is a sample JDBC code:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JDBCTest {

	Connection con;
	PreparedStatement stmt;

	String DB_PATH = "jdbc:odbc:emp";
	String USERID = "";
	String PWD = "";

	public void getEmployees() {
		try {
			Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
			con = DriverManager.getConnection(DB_PATH, USERID, PWD);

			String query
				= "SELECT * FROM EMP WHERE EMP_ID = ? AND EMP_SAL < ?";
			stmt = con.prepareStatement(query);
			stmt.setString(0, "10001");
			stmt.setInt(1, 20000);

			ResultSet rs = stmt.executeQuery();

			while (rs.next()) {
				System.out.println("----------------------");
				System.out.println(rs.getString(1));
				System.out.println(rs.getString(2));
				System.out.println("----------------------");
			}

			rs.close();
			con.close();

		}
		catch (SQLException sqlEx) {} // Handle Appropriately
		catch (ClassNotFoundException classEx) {} // Handle Appropriately
		catch (Exception ex) {} // Handle Appropriately

	}

}
```

Related Questions :

Q: What are Prepared Statements ?
Ans : Prepared statements are Special statement provided by Java by which we do not have to form a long SQL query by appending the parameters to the query. Rather we can use question marks (?) as placeholders for the parameters, and we can set the parameters in the SQL Query at a go.

Q: What are Callable Statements ?
Ans : Callable staments are other types of statements provided by Java which can be used to make calls to database Stored Procedures rather than traditional SQL Queries.

Final vs Finally vs Finalize

The key words final, finally and finalize may sound familiar, but are very different in their operations. Let's discuss few points over them:

final : final is a keyword, saying that my class/method/variable is final, and I don't want anyone to change it, not even myself.

finally : finally is a block in the Exception handling try-catch-finally mechanism, where any code in finally block always executes unless stopped by an System.exit(0). Finally block is generally used to close streams or open connections or any other vital code that is to be executed even if the program has to terminate because of any exception.

finalize : finalize is a method that lands in your class, just to have some code that needs to be executed before your object is garbage collected. You just add the definition of the finalize() method in your class and the java runtime calls your method before the object is Garbage Collected.

Sometimes finalize method can also be exploited to reassign/reference the Object (to be garbage collected) back in our program to stop it from being garbage collected.

Hence, you're making an immortal object that would never be garbage collected. This is also one of the common interview questions.

Collections Framework Hierarchy

The Java collection framework makes our life quite easy by providing lots of utility methods as well as some Classes like lists, maps and sets that give us few implementations that the programmer would had to work his hours on. There are many util methods in collections like sort, length, remove, removeAll etc. which do most our job y themselves. There are few pre implemented data structures like ArrayList, LinkedLists, Trees, Sets, Maps etc. which we would use in our daily life activities.

These are some of the most common collections:

List: The most common collection, basically an increasing array, with all the powers of collection framework. A best replacement for traditional array with powerful pre implemented methods. Implementations of lists are ArrayList, LinkedList etc.

Set: The Set collection ensures that it would only contain unique elements. So if the interviewer asks to maintain a unique array etc, you know you've got a Set collection for your backup. An example of Set implementation is TreeSet which ensures both uniqueness of elements and sorted order of elements.

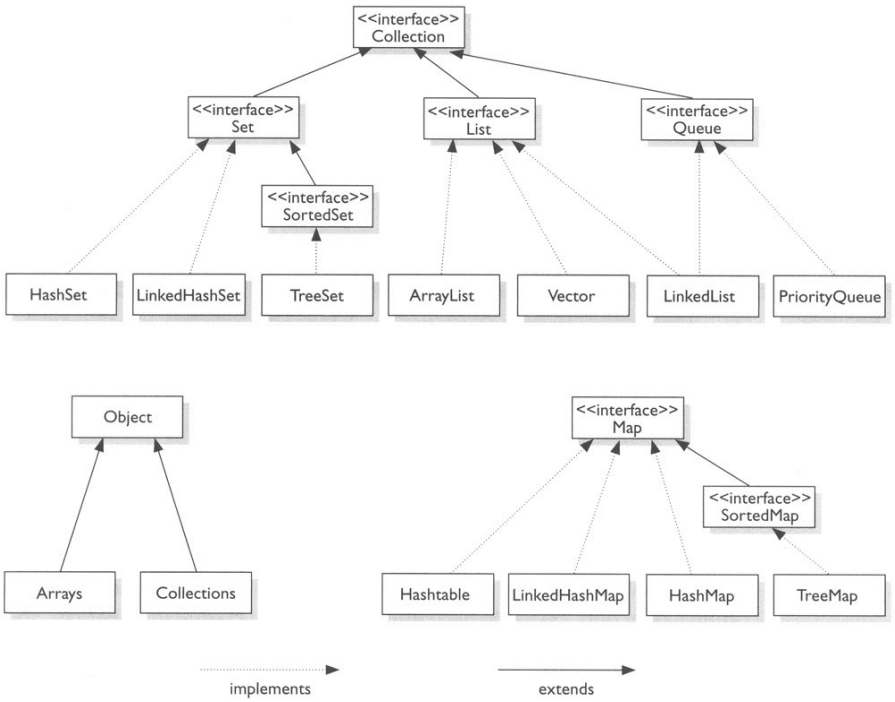
Maps: When you've got a requirement to have key-value pairs, the Map class is here to help you out. Map has values in key-value pairs and guarantees a insertion/deletion in O(1) time complexity. The examples of Map implementation are HashMap, TreeMap etc.

> ***Note:*** *Maps does not implement the Collection Interface, but is a vital part of the Collection Framework.*

Tree: Ensure sorted order.

LinkedList: As the name suggests implements the linked list data structure implementation. It also ensures the insertion order (i.e. ordered list).

Below is the Diagram to show you the Collection Framework Hierarchy:



Here is the table to show you the implementations:

Interface	Implementation				Historical
Set	HashSet		TreeSet		
List		ArrayList		LinkedList	Vector, Stack
Map	HashMap		TreeMap		Hashtable, Properties

Linked List vs Array List

This question is basically to test your data structure awareness. The LinkedList datastructure takes an O(n) time to reach to the element, whereas the ArrayList takes an O(1) time to reach to the element (since ArrayList implements the RandomAccess Interface). Hence you can see the clear advantage of ArrayList over the LinkedList here.

Before we rush, there is another point to consider. The delete/insert operation in an ArrayList requires shifting of all the elements towards right which involves an O(n) time, whereas in LinkedList its just an pointer update, hence can be achhieved in O(1) time.

Now the choice of ArrayList/LinkedList is just depends on the requirement you're into. Discuss the problem with your interviewer by mentioning all the points of both the collection and find out which one is best for his requirement.

String vs StringBuffer vs StringBuilder

While talking of these three string classes and making a choice among them, there are two things that must Immutability and Synchronized (Thread Safe).

String : Strings are immutable, and that means every time we make any changes to the String Object we would be internally making new Objects for each modification. Due to the immutable nature Strings are also Thread Safe. Strings due to this nature are not used much unless we need Read-only constant strings.

StringBuffer : StringBuffer was java's mutable version of String, where any update on the StringBuffer takes place by the 'append()' method and there are no Objects created unnecessarily. StringBuffers are Synchronized and come with an overhead of Synchronization, and hence must not be used unless we are dealing with a Threaded environment.

StringBuilders : StringBuilder is java's mutable and un-synchronized version of String, which is neither immutable nor is Synchronized. StringBuilder is the most suggested claas to be used for Strings for common un-threaded scenarios.

wait(),notify() & notifyall()

You would have come across these three methods while writing your Thread programs. Here is a sweet difference between them and you can definitely expect such questions on threads and related methods.

wait() : If one of your thread has completed a part of its execution, and needs to allow some other thread to perform its work it uses a wait method to pause its execution, and any other thread would get a chance to be active and complete its execution. A wait() call ensures that all the resources occupied by the thread are released. A waiting thread can only be made alive by a notify() call by other thread. wait() method throws an InterruptedException and must be handled accordingly.

notify() : Any thread that has completed its execution can signal other waiting thread to come out of its waiting state and continue with the operation. The notify() method call wakes up the 1st thread that would had called the wait() method in the program.

notifyAll() : Call to notifyAll() is similar to notify(), with a difference that it wakes up all the waiting threads in the application, and the thread with the highest priority starts its execution.

> ***Note:***
> *notify() must be used when we want the threads to come live in order of their issue of wait() call, or if there is only one other thread in your application. notifyAll() on the other hand can be used when you need the highest priority thread to come live. If all the threads are of equal priority (or if you're not sure of the priorities) you must be sure that your application works fine with any thread that wakes up.*
> *Also, all the three methods (wait, notify and notifyAll) must be present inside the synchronized block.*

sleep() vs. wait()

Another common interview question is the difference between sleep() and wait(). Sleep and Wait behave quite similarly where you might be confused between them, but the fact is that there are hell lot of differences between them :

sleep() : Sleeps the current thread, takes in the time interval (in microseconds) as a parameter for which the thread sleeps. Sleep is not related to multi thread implementation. Sleep does not send any signal to any other thread to come live etc. Most importantly Sleep does not release any resources occupied by it, it loves sleeping with the resources it has and starts its execution with the resources as soon as it wakes up. A sleeping thread wakes up itself (after the specified time interval).

wait() : Wait method is actually an important part of the threaded apps, where a wait() call would bring the current thread to waiting state, releasing all its resources. A waiting thread cannot wake up itself, unless issued a notify() signal by some other thread. Again, the wait() method call throws an InterruptedException and must be handled accordingly.

What does the join() method do ?

The join method links one thread of your program with any other thread such that the first thread would now wait for the second thread to complete its execution.

How to execute system commands in java ?

Java has provided a class 'java.lang.Runtime' which lets you execute the system commands directly from your java program. So now you can create directories or call some other program directly from your java program. Here is the crisp code :

```
String command = "cd";
Runtime.getRuntime().exec(command);
```

What is the Reflection API?

Generally we create classes and hence we have all the information about the class and its objects, but certain times we just have a class object, but do not have any information about the class and we need to know about the class properties etc at runtime. The Java Reflection API is just for these purposes where we can get information about classes and their behavior at runtime.
The code below takes a class name as input and shows the list of methods available in the class. There are lots other functions in the Reflection API which you sure should explore :

```
import java.lang.reflect.Method;

class ReflectionDemo {


public static void main(String args[])
   {
      try{
         System.out.println("Pleez gimme the Class Name :");
         Scanner sc = new Scanner(System.in);
         String className = sc.next();
         Class c = Class.forName(className);
         Method m[] = c.getDeclaredMethods();
         for(int i = 0; i<m.length; i++)
            System.out.println(m[i].toString());
         }
       catch(Throwable e) {
         System.err.println(e);
       }
   }
}
```

What is java cloning ? What are the different types of cloning?

Certain times we need to create a copy of any object because we just don't want to modify the original object. By default you can call the object.clone() method to obtain a copy of your object.Cloning is of two kinds :

Shallow Cloning :
This is the Java's default cloning mechanism. Here a new object is created with all the properties of the old object copied into the new object, but if the object itself contains another Object, both the new and old objects contain a reference to the same object internally. So if you modify the inner object of the new object it also modifies the old object.

Deep Cloning :
Deep Cloning is the mechanism by which the Object and all the Objects inside the Object and all the objects inside those objects – all are copied into new objects recursively. Deep cloning has to be handled by the programmer by overriding the clone() method handling the copying by recursion.

What are annotations, how are they useful, name few common annotations?

Annotations was a feature introduced by Java 5 where we were allowed to mark our classes, methods and members. Usage of annotations have rapidly grown and are now used extensively in ordinary java codes .. in web service codes .. in ORM's like hibernate and JPA .. and so on ..
Annotations are used to describe a class/method/instance , how they are going to be used, and all this information is used for the construction of the object.

Annotation is a very powerful mechanism and can be used in a lot of different ways (source: stackoverflow) :

- To describe constraints or usage of an element: e.g. @Deprecated, @Override, or @NotNull
- To describe the "nature" of an element, e.g. @Entity, @TestCase, @WebService
- To describe the behavior of an element: @Statefull, @Transaction
- To describe how to process the element: @Column, @XmlElement

Feel free to explore the annotations, and use them in your daily life codes.

Only for the Caffeine Blooded

This section of the book covers some of the most interesting and conceptual questions asked in Java interviews. The section has few uncommon interview questions which can give you a very clear understanding how java internals work.

Answering these kinds of questions gives you a special edge over other candidates.

Please be thorough with these questions. While some questions may be familiar to you, there may be few which would be totally new to you. Practice the code provided to you and understand the concepts how the code is behaving. Customize the provided code, play with it to see its different behaviors.

How does garbage collection work in Java ?

Garbage collection is the mechanism by which java internally cleans its heap and removes all un-referenced objects from the heap. Java garbage collection is an implicit process unlike traditional C/C++, where the coder has to write code for freeing/flushing the memory after his work is done. The programmer himself can also make a request to the Garbage Collector to be invoked by the below line of code :

```
System.gc();
```

Which ofcourse is not a guaranteed operation, and the JVM might just ignore the request.

Java internally uses the **Mark & Sweep** algorithm for garbage collection, where it performs its operation in three steps.
In the first step it starts with the top level references – also known as the root references, and then travels through the Object Graph. It finds out all the reachable objects by traversal through the Object Graph and Marks all the reachable Objects.
In the second step the GC walks through every Object present in the Java heap and discards all the Objects that are not marked. These are the Objects that are now eligible for Garbage Collection.
The final step of GC is about reclaiming heap from the GC eligible Objects, and cleaning those un-referenced Objects.

The un-referenced objects are all the objects which have no references to them from the program, for example below is a code showing when two Objects obj1 and obj2 go un-referenced:

```
public void someMethod(){

            /** one object created in memory  **/
            MyClass obj = new MyClass();

            /** This line of code makes the obj's object un-referenced **/
            obj = null;

            /** another object created in memory  **/
            MyClass obj2 = new MyClass();

            /**  the next object obj2 becomes un-referenced **/
            obj2 = obj;
      }
```

Again, we must be aware that Mark & Sweep is not the only GC Algo Java uses, its just the most popular one. There are also many different GC algorithms present such as : The Serial Collector, The Parallel/Throughput Collector, The Parallel Old Generation Collector etc.

Feel free to explore them in free time.

How is HashMap actually implemented in java ?

This one of the java internals questions which has become common these days, While some people say why do we need to bother about it, haven't we got a java API for that; there are few who have really dug up API's getting insights over the internals.
We're gonna be the second type now :

Before we dig up, **Lets understand little Hashing from basics**.
I have 10 numbers : 12, 32, 56, 34, 78, 10, 4, 21, 16, 62.
(consider these numbers as hashcodes returned from your hashCode method).
And I have a hashing function mod 10, to distribute my numbers in hashmap.

The hashing function uses the numbers to give us the bucket number where my object must fall. And the hashing function has a pre defined number of buckets.

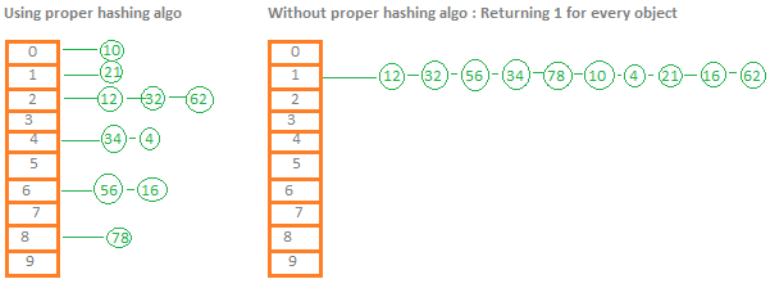
So since my hashing function is mod 10, so the number of buckets which I want to use is also 10. And we put values in the bucket using our hashing algo 'mod 10'.

The point where two values give the same bucket number is known as a Hash Collision.
e.g. 12%10 and 32%10, both result is equal to 2. This is a **hash collision**.

There are number of ways to resolve hash collision, a common solution is by using LinkedList of elements for each bucket. Every time a new element comes in, its hashcode would be used to calculate a bucket number, and if the bucket is initially empty the object is directly put in the bucket, or else a hash collision occurs and the element is added to the end of the linked list.

The below diagram shows the calculated bucket number for each item, and puts them in buckets. Now you can see the advantage of the Hashing where we have distributed the values somewhat evenly, and for searching an element we just have to go to the bucket number and search the value among the values in that particular bucket.
Also, every node in the linked list keeps both the key and value inside each node. The key inside node would be helpful for retrieving values from the HashMap.

Hashing Formula	Bucket Num
12 % 10	2
32 % 10	2
56 % 10	6
34 % 10	4
78 % 10	8
10 % 10	0
4 % 10	4
21 % 10	1
16 % 10	6
62 % 10	2



The above diagram also shows an poor hashcode implementation, where I would always return a constant value from my hashcode method, and when hashing algo is applied on the hashcode a collision occurs for every element, hence all elements fall in a single bucket in a very long linked list.

Now, How java implements Hashing ?

For putting an element into the map via the '*put(key, value)*' method, java gets the hashCode() of the key, and applies its hash algo over the hashcode to figure out the bucket number the *value* object has to fall into. Now java knows where the value has to fall.

Now it needs to find if the key is already present in the Map, whether it has to insert new element into map or update/replace with old value. So it goes to the bucket to check if it has any element in it.

If there is no element present in the bucket, the element is directly added to the bucket, else if there is an element present, java would now have to traverse through the LinkedList to find if the key is alersdy present in it. It uses the equals() method on *key* of the elemts to find that out. If it gets a matched key it replaces the node of the LinkedList, else it would make another enty in the end of the LinkedList.

For getting an element from the map, using the *get(key)* on the object, it again gets the hashcode and applies the hash algo to get the bucket number, there it uses the equals() method to find the node with the matching key, and return the corresponding value.

Related Question :

Q: What order does java use equals and hashCode method?

A: Java calls the hashCode() before the equals() method. That is because hashCode() is used to get the bucket number, and equals() can only be used after we have reached the bucket, since equals() would be used on the nodes of the LinkedList contained in the bucket.

Q: What if I always return a constant value in the hashCode method (say return 1). Does that satisfy the Contract ?

Yes the contract is satisfied, since we are always returning a constant value from the hashCode() method, any two equal objects would also have the same hashCode() value, hence the contract is satisfied.

But, as we have seen in above diagram, every object is going to have the same hashcode, so when Java uses the hashcode in the hashing algo, every objects tends to return a same value, hence every object is put in the same bucket in a very long LinkedList. It is equivalent to having a long array of elements. So where are we taking advantage of the performance of Hashing ?

Q: Can you write a better HashMap than java ?
A: Well we can definitely give some suggestion for a better HashMap implementation. As we know java keeps LinkedList for all inner buckets, so finding any element on that LinkedList is an O(n) time operation, since in linked lists we need to traverse through every element to reach to the N-th element.
A suggestion can be to maintain a Binary Search tree instead of the LinkedList, so we can reduce the insert/retrieval operation's time to O(logN), which is a significant improvement.

Relationship of equals and hashCode method.

Here's the story :
Java has two pretty methods 'public boolean equals()' and 'public int hashCode()' . While the equals method checks the equality of two objects by returning a Boolean flag, the hashCode method gives a unique code for the objects, where two equal objects must return a same int value.

> ***Note :*** *This does NOT mean that two unequal objects cannot have the same hashcode. Two different Objects may/maynot have a same hashCode, but two equal objects are sure to have a same hashCode according to the default hashCode implementation by Java. And that is what java expects from you if you override the hashCode() method.*

The equals method works by comparing the hashCodes of objects, that is a way Java can know if two objects are equal. The hashCode method calculates a code for every object based on some formula – any formula. We'll talk about the formula later and see that why certain formulas can give the same hashcode for different objects.
Now we can talk about the Contract … **Contract, what contract ?**

> *Java says that you can feel free to override the equals() method but please override the hashCode() method also accordingly.*

Why ?
Because while java implements the hashed collections/maps it would need to maintain a unique key condition. Now the key itself can be an object. So it would need to know if the key(object) is already present in the HashMap. And how would it know it that ? – by the help of equals and hashCode method. Any two equal objects would pass both the tests i.e. would return true from equals method and both would have a same hashcode returned. By these two methods it identifies that the incoming key is already present in the map, hence the value has to be replaced by the new value, and if the key is a new key we can directly put it into the map without any issues.

Explanation by code:
Let's take a sample logic. Say I have a class 'Coder', which has a String value 'language', and so I want any object having same value for the language to be considered equal. So two objects with "java" as the language would be equal to each other, hence the equals method would compare the language field for both the objects and return a true value for equal language value and false for a non-equal language value. Below is the code for your understanding :

```
class Coder {
	private String language;

	public boolean equals(Object object) {
		Coder coderObj = (Coder) object;

		if (coderObj.language.equals(language))
			return true;
		else
			return false;

	}
}
```

So the contract says,

Since you've overrided the equals method based on your logic, the equal method is going to return a true for two similar objects (with same language value for both objects). So now please override the hashCode() method such that those two similar objects return the same int value for both the objects. This return value from hashCode() method would be used by java to calculate the hash value of every object to implement its hashing algo.
If its clear, we can move on to discuss on some algorithm that we can use for hashCode method.

Requirement for hashCode() method :
Two equal strings must return a same int value.

A simple approach can be :
Assign a value to all the alphabets (a to z) : a=1, b=2, c=3, d=4 .. and so on .. y=25,z=26.
also we can have an index for each alphabets in a string, and so for any word we calculate the hashcode in a weighted average manner. So for the word 'java' the hashcode is calculated as :
Sum(value of alphabet * index of alphabet) for all alphabets in string
(10*1) + (1*2) + (22*3) + (1*4) = 1 + 2 + 66 + 4 = 73.
We can use this for starting with hashcode overriding.

> *For extra read, Tha java implementation of hashcode for Strings is :*
> *s[0]*31^(n-1) + s[1]*31^(n-2) + … + s[n-1]*
> *where, s[0]..s[n-1] are the characters in string at locations 0 .. n, and, n is the length of the String. ^ here refers to exponentiation (to the power of).*

Well we can always come up with our own algo for hashcode, or you can always use hava's own implementation. Below is the code using java's own implementation which you can use for most cases. You can replace the code inside hashCode() method to add your own algo.
Type in the code to check the output :

```
public class Coder {

	private String language;

	public Coder(String language) {
		this.language = language;
	}

	public boolean equals(Object obj) {
		Coder test = (Coder) obj;

		if (test.language.equals(language))
			returntrue;
		else
			returnfalse;
	}

	public int hashCode() {
		returnlanguage.hashCode();
	}

	public static void main(String[] a) {

		Coder c1 = new Coder("Java");
		Coder c2 = new Coder("Java");
		Coder c3 = new Coder("Python");

		System.out.println("HashCodes(c1,c2,c3)" + c1.hashCode() + ","
				+ c2.hashCode() + "," + c3.hashCode());
		System.out.println("c1 equals c2 : " + c1.equals(c2));
		System.out.println("c1 equals c3 : " + c1.equals(c3));

	}

}
```

What happens if we override equals() method without overriding hashCode() method?

If we override the equals() method of in a class but do not override the hashCode() method, java hashing algo would not respect the equals() method while comparing equality of two objects; even if the equals() method is present in the class and two objects are equal.

> *Java would use the hashCode() and equals() method for implementing its hashing mechanism. The hashCode() method is called before the equals() method. Two objects are equal if and only if they pass both the hashcode and equals test.*

Type out the below code for clarity, please play with this code, its important:

```
import java.util.HashMap;
import java.util.Map;

publicclass Coder {

	private String language;

	public Coder(String language) {
		this.language = language;
	}

	public boolean equals(Object obj) {

		Coder test = (Coder) obj;

		if (test.language.equals(language))
			return true;
		else
			return false;
	}

	/*
	 * public int hashCode() { return language.hashCode(); }
	 */

	public static void main(String[] a) {

		Coder c1 = new Coder("Java");
		Coder c2 = new Coder("Java");
		Coder c3 = new Coder("Python");

		Map<Coder, String> myMap = new HashMap<Coder, String>();
		myMap.put(c1, c1.language);
		myMap.put(c3, c3.language);
```

```
        System.out.println("Map Size:"+myMap.size());
        System.out.println(myMap.get(c2));

        }

}
```

Try running the program and observe the output. You'll see that in spite of the implementation of the equals() method the System.out.println(myMap.get(c2)) returns a NULL value, even though we've told the equals method to compare only the 'language' value. So basically java is ignoring the equals() method because it looks for the hashCode() method to check if an object is present in the HashMap.
Now, un-comment the hashCode() code and run the code again, now you'd see that the value 'Java' is returned, hence java is able to find the object in the HashMap.

What is immutability? Write an Immutable Class.

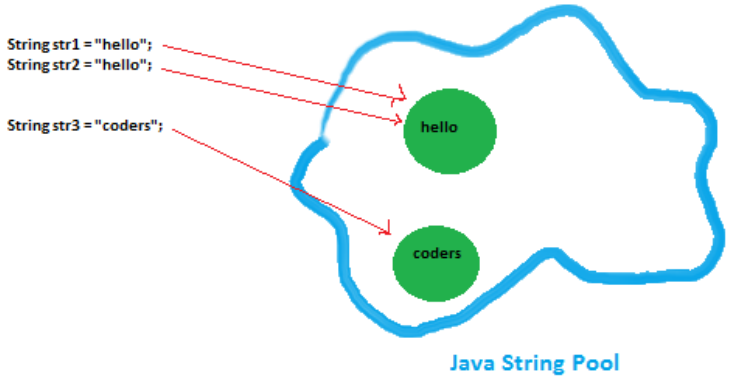
Immutability is another cool OOP concept where we are not allowed to edit any instance of any Object once its created. So what's cool in that ?
Well all i can say is it is a great model where you restrict the other coder from changing any object you've made. So basically you've made a class such that you can use its instance but cannot modify the instance. You can always use it as a read-only object but any changes on the object can be made on a copy of the object or the values separately, but none of the changes would be made on the original instance of the class. Some example os immutable class can be some class with some configuration values which you don't want to change by any other stupid coder.

```
import java.util.ArrayList;
import java.util.List;

public class ImmutableClass {

	String strVal;
	int intVal;
	List listVal;

	public ImmutableClass(String strVal, int intVal, List listVal) {
		this.strVal = strVal;
		this.intVal = intVal;
		this.listVal = listVal;
	}

	/**
	 * Getters for the fields.
	 */
	public String getStrVal() {
	/**
	 * String can be returned directly here, since this string cannot be
	 * modified by the returned value.
	 */

		return strVal;
	}

	public int getIntVal() {
	/**
	 * primitives can be returned directly here,
	 * since this primitive cannot
	 * be modified by the returned value.
	 */
		return intVal;
	}
```

```
	public List getListVal() {
	/**
	 * Any external code can change the list from the reference, by the
	 * .add(), .remove() methods over the reference. So, return a copy of
	 * the list instead of direct reference.
	 */

		return new ArrayList(listVal);
	}

	/**
	 * Do not provide any setters for any instance variables.
	 */
}
```

What is string pooling, why are strings immutable ?

String Pooling is the mechanism by which any String literal with a same value is referenced to the same Object, that is present on the String Pool. That means that when you're creating two strings str1 and str2 both with a same value 'ConfusedWithJava', only one String Object is created on the String pool and any other String literal would point to the same Object in the String pool. Hence Java is saving lots of Heap Memory by implementing the String pool where all the same valued String literals do not have to create new Objects but internally use the same Heap Memory Object.
The diagram below throws some light over the same :



Java String Pool

Java maintains the String Pool to optimize String Object creation and hence it takes special care so as to ensure that the String pool works flawlessly. Java ensures that by making the String Objects IMMUTABLE.
Yes, Java makes String objects immutable so that you are restricted to change the "hello" value mentioned above, because there might be N-number of other references referring to the same Object in the Heap, and since the Object is shared across those variables, so modifying the original Object would lead to a disaster.

So java makes the String Objects Immutable so that any modification on a String object would not modify the Object on the Heap. For example :
str2 = str2+" coders";
would not modify the same object on the Heap, but rather :

– > create a new String Object
–> copy the value of the original String to this new String Object

–> apply the modification on this new String Object (str2+" coders")
–>update the reference to the old Object to this new Object.

So, now you can understand why Java has to put in this amount of extra effort just to ensure that its String Pooling is working flawlessly ?

Note : String pool is only for String literals, hence writing :
String str2 = new String("hello");
Doesn't work at all, since you're explicitly asking Java to create a new Object for you, hence it is incapable of using the String Pool.

Where do string objects fall? Do new String() objects also fall in string pool?

Any new String literal would fall in the String literal pool. Any String Object created by the 'new String()' would fall in the Java Heap. And yes, the String literal pool is itself the part of the Java Heap.

What are Design Patterns ? Could you write code for some most common Design Patterns ?

Design patterns are the programming best practices that make our code more organized, more re-usable and more efficient by implementing standards for some most common programming scenarios. Below are the few most common Design Patterns :

Singleton Design Pattern :
Singleton Design Pattern is the design pattern that ensures that only one instance of your class would be created by any program/programmer. Singleton would probably the 1st design pattern that any interviewer would ask you, so better understand the code properly and keep it on tip of your tongue.

```
public final class Singleton {

	private static Singleton singleton;

	private Singleton() {
	}

	public synchronized Singleton getInstance() {
		if (null == singleton)
			singleton = new Singleton();

		return singleton;
	}

}
```

Code explanation :
Instance of the class is inside itself, and made private static. We have made the constructor private so that it cannot be instantiated by the new operator directly. We have provided a public 'getInstance()' method which is the only way the class's object can be received since we have blocked all other means of object creation. In the getInstance() method we check if the singleton object is initialized or not. We only create an object of singleton if its initially null, every other time it would return us the same object that was created the first time.
Also note that the instance itself is static, so as to keep only one copy of it for the class.

Factory Design pattern :

Factory is another very frequently used design pattern where we create a master class whose work would be to return us objects as per our requests. The class would have a public method that would return us object of different classes based on some input parameter we pass. The returned object can be type-casted and used accordingly.

```
class Mutant {}

class Mystique extends Mutant {}

class Wolverine extends Mutant {}

class Magneto extends Mutant {}

public class MutantFactory {

	public Mutant getInstance(String power)
	{
		if ("ADAMANTIUM_CLAWS".equalsIgnoreCase(power))
		{
			return new Wolverine();
		}
		else if ("SHAPE_SHIFTING".equalsIgnoreCase(power))
		{
			return new Mystique();
		}
		else if ("MAGNETIC_MANIPULATION".equalsIgnoreCase(power))
		{
			return new Magneto();
		}
		else
			return new Mutant();
	}

}
```

Code Explanation :

We have a MutantFactory which is capable of returning different Mutants based on the power required, so you pass the power to your MutantFactory and it decides which mutant is capable of the power and returns the corresponding Mutant.

There are two points to note in the get instance method, one is that it needs to have the comparison logic inside it to decide which object has to be returned. The second thing is the return type of the method, which needs to be the Super class's type, since it has to return object of different classes. So we keep the return type as the parent class's type, and the used can Type cast it back to the child type before use.

Abstract Factory Design Pattern:

Since we've now got a feel of Factory design pattern, its time to talk about the Abstract factory design pattern. Abstract design pattern is nothing but a collection of multiple Factories. The factory design pattern was returning us the Superhero instance, similarly the AbstractFactory would return us a whole Factory Object which we can again use for getting our Superheroes out.

```
class Superhero {}


class Superman extends Superhero {}
class Batman extends Superhero {}
class GreenLantern extends Superhero {}
class Thor extends Superhero {}
class Ironman extends Superhero {}
class Hulk extends Superhero {}



interface SuperheroFactory {
      public Superhero getSuperhero(String power);
}



/** One SuperHero Factory Made by the DC Comics **/
class JusticeLeagueFactory implements SuperheroFactory {

      public Superhero getSuperhero(String power) {

            if ("SUPER_STRENGTH".equalsIgnoreCase(power))
                  return new Superman();

            else if ("BAT_MOBILE".equalsIgnoreCase(power))
                  return new Batman();

            else if ("POWER_RING".equalsIgnoreCase(power))
                  return new GreenLantern();

            else
                  return new Superhero();
      }

}
```

```
/** Another SuperHero Factory made by the Marvel Comics **/
class AvengersFactory implements SuperheroFactory {

	public Superhero getSuperhero(String power) {

		if ("COOL_WEAPONS".equalsIgnoreCase(power))
			return new Ironman();
		elseif ("THUNDER_GOD".equalsIgnoreCase(power))
			return new Thor();
		elseif ("HULK_SMASH".equalsIgnoreCase(power))
			return new Hulk();

		else
			return new Superhero();
	}
}


/** My Abstract Factory can give you Factories of SuperHeroes **/
public class SuperHeroAbstractFactory {

	public SuperheroFactory getSuperheroFactory(String teamName) {

		if ("AVENGERS".equalsIgnoreCase(teamName))
			return new AvengersFactory();
		else
			return new JusticeLeagueFactory();
	}

}
```

Adapter Design Pattern :

As the name says, Adapter must be some kind of bridge between any two incompatible classes/interfaces etc. Its like you've got something to do with your parameters, but the class that performs the actual action does not take those input values. So you write a adapter class once – and every time you have to use that class you create object of your adapter class which internally uses the class and returns the appropriate result back.

```
class Original {
	public String secretFormula(String param) {
		return"secret output :" + param;
	}
}
```

```
/** Adapter Extends the Original Class **/
class Adapter extends Original {

	public String secretFormula(float param) {
		/** Conversion **/
		return super.secretFormula(String.valueOf(param));
	}
}


public class UserClass {
	public static void main(String[] args) {
		Adapter adapter = new Adapter();
		System.out.println(adapter.secretFormula(9.99F));
	}
}
```

MVC Design Pattern :

MVC is the model-view-controller design pattern where we try to keep the code logic separate from the view(UI) code. Its a very appreciated design pattern and very simple one indeed. You'd see lots of MVC web frameworks nowadays like Struts, Spring JSF etc. The idea is very simple, we'll have all our logic in a different class and all our UI related code in a separate class. And there is a controller class which plays with the whole flow through code.

```
import java.util.ArrayList;
import java.util.List;



class Movie {
	String name;
	String type;
	int rating;

	public Movie(String name, String type, int rating) {
		this.name = name;
		this.type = type;
		this.rating = rating;
	}

}
```

```
/** Model Class - Handles Main App Logic/Data Handling etc **/
class Model {

	public List<Movie> getMovieData() {
		List<Movie> movies = new ArrayList<Movie>();

		movies.add(new Movie("The Godfather", "crime|drama", 9));
		movies.add(new Movie("The Hangover", "comedy", 7));
		movies.add(new Movie("The Grudge", "horror|mystery", 5));
		movies.add(new Movie("Step Up", "drama|music|romance", 6));

		return movies;
	}
}



/** View CLass - Handles User Output/Display **/
class View {

	public void display(List<Movie> movies) {


	System.out.println("*************************************************");
	System.out.println(" Movies to Watch ");

	System.out.println("*************************************************");
	System.out.println("| Movie Name | Movie Type | Rating |");

	System.out.println("*************************************************");
		for (Movie movie : movies) {
			System.out.println("| " + movie.name + "\t"
					+ movie.type + "\t\t"
					+ movie.rating);
		}


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


/** Controller Class - Manages App Flow **/
class Controller {
	public void displayMovies() {
		Model model = new Model();
		View view = new View();

		List<Movie> movies = model.getMovieData();
		view.display(movies);
	}
}
```

```
public class MVCDesignPattern {
	public static void main(String[] args) {
		Controller application = new Controller();
		application.displayMovies();
	}

}
```

Write code for threaded Producer-Consumer problem

The idea is to create two threads – A Producer thread and one Consumer thread. A producer thread would only produce when the resource has been consumed, and so the Consumer thread can only consume a resource if it has been produced.
Hence the Producer and Consumer work alternatively to produce & consume the resource. Another variation of this question can be printing all the numbers upto 100, such that one thread would be printing odd number while other would be printing even numbers. It is also another variation of the Producer-Consumer problem.
Below is the code :

```
/** The Resource Class - The root cause of all fight **/
class Resource {
	private Boolean isProduced = false;
	private String data = "EMPTY";


	/** Put method : puts only if is not already produced **/
	public synchronized void put(String data) throws InterruptedException {
		if (isProduced) {
			wait();		// Not Consumed Yet,Wait for consumer's signal.
		} else {
			this.data = data;
			isProduced = true;
			notify();	// Tell the Consumer that i'm done producing.
			System.out.println("Producer >> Data:" + this.data);
		}
	}


	/** Get method : gets only if it has been produced **/
	public synchronized String get() throws InterruptedException {
		if (!isProduced) {
			wait();		// Not produced yet,wait for producer's signal.
		} else {
			String data = this.data;
			this.data = "EMPTY";
			isProduced = false;
			notify();	// Tell the Producer that i'm done consuming.
			System.out.println("Consumer >> Data:" + this.data);
			return data;
		}

		return data;
	}
}
```

```
/** The Produver Thread  **/
class ProducerThread implements Runnable {

	Resource resource;

	public void run() {
		for (int i = 0; i < 50; i++) {
			try {
				resource.put("DATA ADDED");
			} catch (InterruptedException e) {
			}
		}
	}

	public ProducerThread(Resource resource) {
		this.resource = resource;
		new Thread(this).start();
	}

}
```

```
/** The Consumer Thread **/
class ConsumerThread implements Runnable {

	Resource resource;

	public void run() {
		for (int i = 0; i < 50; i++) {
			try {
				resource.get();
			} catch (InterruptedException ex) {
			}
		}
	}

	public ConsumerThread(Resource resource) {
		this.resource = resource;
		new Thread(this).start();
	}

}
```

```
/** Test Class **/
public class ProducerConsumerDemo {

	public static void main(String[] args) {
		Resource resource = new Resource();
		new ProducerThread(resource);
		new ConsumerThread(resource);
	}

}
```

Explanation :

We have a Resource class, a class that would be accessed by both the Producer Thread and Consumer Thread. The resource is a shared object hence is passed to both the Threads by our main method. The resource object is a smart object and keeps a track of the last accessing thread by the boolean flag 'isProduced' and helps us decide whether a thread must go on with its execution process or must wait till the other thread completes its operation.

> ***Some points to note :*** *Wait must be inside a synchronized block, and it throws InterruptedException and must be handled in the code. Again, for the Odd-Even problem just change the String 'data' to a int value, and keep on incrementing and printing it after operation completes (just before notify() call).*

What is Connection Pooling, Write a piece of code for implementing a Connection Pool.

Connections are one of the most expensive thing in your application. And opening/closing/re-opening connections is one of the most frequent operation in the app and also the most expensive one. So smart people have come up with an approach where we create a pool of connections and instead of opening and closing a connection, we just pick up the connection from the pool and once our work is done we just return it back to the pool.
This ensures that we are avoiding the cost of opening/closing/re-opening connections.
Generally Connection pooling is internally handled by most of the Application Servers, but people can always ask you the write some code on how would you achieve it manually. Here is a sample implementation of connection pool, not perfect one but cool enough to get you started. Feel free to explore better approaches for the same :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;



/** A Connection Pool with 5 Available Connections **/
class ConnectionPool {

	private List<Connection>availableConnections =
                                                  new ArrayList<Connection>();
	private List<Connection>usedConnections = new ArrayList<Connection>();
	private final int MAX_CONNECTIONS = 5;

	private String URL;
	private String USERID;
	private String PASSWORD;


	/** Initialize all 5 Connections and put them in the Pool **/
	public ConnectionPool(String Url, String UserId, String password)
			throws SQLException {
		this.URL = Url;
		this.USERID = UserId;
		this.PASSWORD = password;

		for (int count = 0; count <MAX_CONNECTIONS; count++) {
			availableConnections.add(this.createConnection());
		}

	}
```

```
	/** Private function,
	used by the Pool to create new connection internally **/

	private Connection createConnection() throws SQLException {
		return DriverManager
				.getConnection(this.URL, this.USERID, this.PASSWORD);
	}



	/** Public function, used by us to get connection from Pool **/
	public Connection getConnection() {
		if (availableConnections.size() == 0) {
			System.out.println("All connections are Used !!");
			return null;
		} else {
			Connection con =
			availableConnections.remove(
				availableConnections.size() - 1);
			usedConnections.add(con);
			return con;
		}
	}



	/** Public function, to return connection back to the Pool **/
	public boolean releaseConnection(Connection con) {
		if (null != con) {
			usedConnections.remove(con);
			availableConnections.add(con);
			return true;
		}
		return false;
	}




	/** Utility function to check the number of Available Connections **/
	public int getFreeConnectionCount() {
		return availableConnections.size();
	}
}
```

```
/** Test Class for testing Connection Pool **/
public class ConnectionPoolTest {

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

		ConnectionPool pool
			= new ConnectionPool("jdbc:mysql://localhost/mydb",
				"testusr", "somepwd");
		Connection con1 = pool.getConnection();
		Connection con2 = pool.getConnection();
		System.out.println(pool.getFreeConnectionCount());
		Connection con3 = pool.getConnection();
		Connection con4 = pool.getConnection();
		Connection con5 = pool.getConnection();
		Connection con6 = pool.getConnection();
		System.out.println(pool.getFreeConnectionCount());
		pool.releaseConnection(con1);
		pool.releaseConnection(con2);
		pool.releaseConnection(con4);
		System.out.println(pool.getFreeConnectionCount());
	}

}
```

Can you implement few common Data Structures?

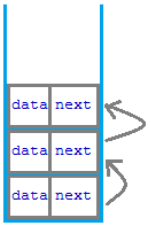
Data Structure is a high probability topic, and any one should at least be able to write code for the most basic data structures like Stacks, Queues and Linked Lists. They form the construction blocks for other more complex data structures.
Below we have given code for Stack, Queue and Linked List implementation in java. Before we jump to the code let's discuss the approach we must take for writing any data structure code.

Draw diagrams –
Try to understand what is required. The toughest data structure problems are solved by simple diagrams. Draw diagram to put down your understanding on paper.

Figuring out the smallest component –
Any data structure that we're trying to implement would definitely have a smallest unit/element. For example the smallest element of Linked List/Tree would be a Node, and for Stack/Queue it'd be a simple item. The choice of the smallest element is required to write complex code over it.
This is a small diagram showing the choice of basic units for Stack/Queue/LinkedList :

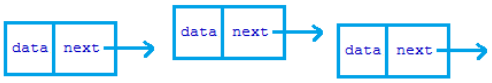
Stack/Queue

```
class Item {

      int data;
      Item next;

      public Item(int data) {
            this.data = data;
      }
}
```



Linked List

```
class Node {
      int data;
      Node next;

      public Node(int data) {
            this.data = data;
      }
}
```



Figuring out how to connect the smaller elements to form complete DS –
Now we've decided a fundamental building block, now we need to connect the smallest elements to form a collection to make our data structure ready. We generally connect the basic blocks by keeping reference of the 'Item' within itself. The inner reference of Item would

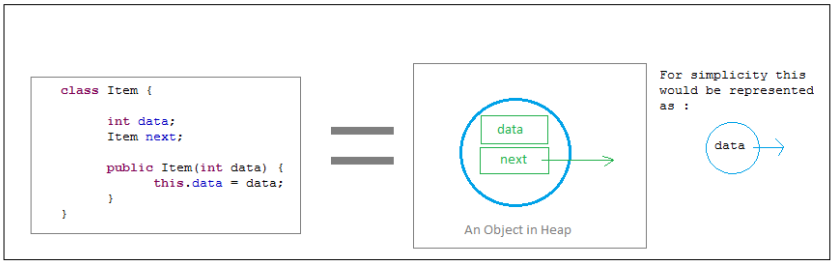
enable it to point to a similar Item; hence we can have a chain of Items without using any external collection like Array or ArrayList.
Now sometimes its simpler to identify that we need a reference inside the same Item, like in Linked List's Node where its implicit that one node would be pointing to the other node, so keeping the 'next' pointer in Node itself makes sense. But with scenarios like stacks or queues it may not be implicit, but we can still have that internal reference to achieve the chain of Objects without having to use additional collections etc.

Figuring out the common operations on the DS –
The common operations for stack is push() and pop(), for queue its enqueue() and dequeue(). For linked list we can have lot of common operations like insert(), insertAtLast(), insertAtFirst(), remove() etc etc.

Dry Run the code with 2 and 3 elements –
Write your code and test it with 2 objects and then 3 objects. If it works fine with lesser elements then go for larger data sets.

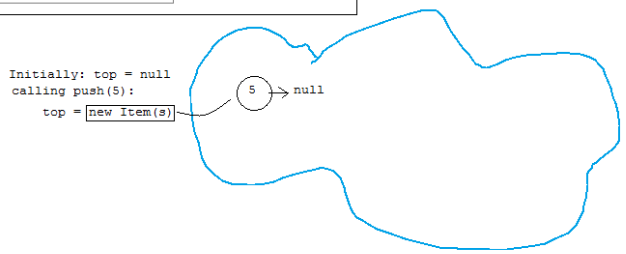
> ***Note**:*
> *I have given implementation for Stack/Queue/Linked List for basic understanding over most common Data Structures. The code for simplicity only has the core logic implemented. There is no Boundary value check implemented. Add boundary value checks on each code after you've got the basics in place. It's a homework ☺*

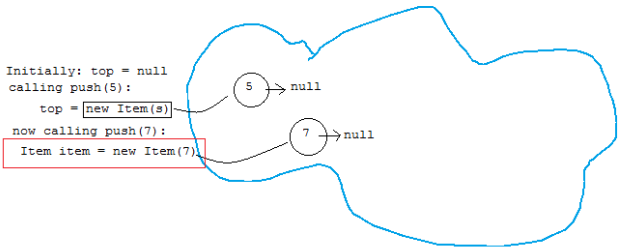
Stack:
Stack is the LIFO Data Structure, where insertion and deletion both occur at the same end. I have selected 'Item' as the fundamental block, where one Item contains data and another Item reference which would be used to point to another Item Object, hence building a chain of Item Objects.
As you can notice we are using the Linked List's idea to emulate a Stack. Below is the plan for push and pop operations:

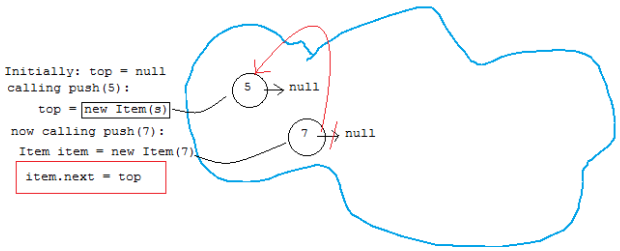


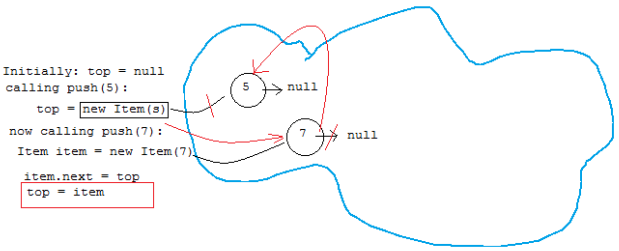
Stack PUSH Operation

```
public class Stack {

    private Item top;

    public void push(int data) {
        if (null == top) {
            top = new Item(data);
        } else {
            Item item = new Item(data);
            item.next = top;
            top = item;
        }
    }

    public int pop() {
        int data = top.data;
        top = top.next;
        return data;
    }
}
```

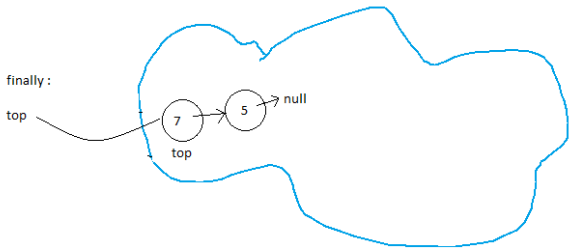


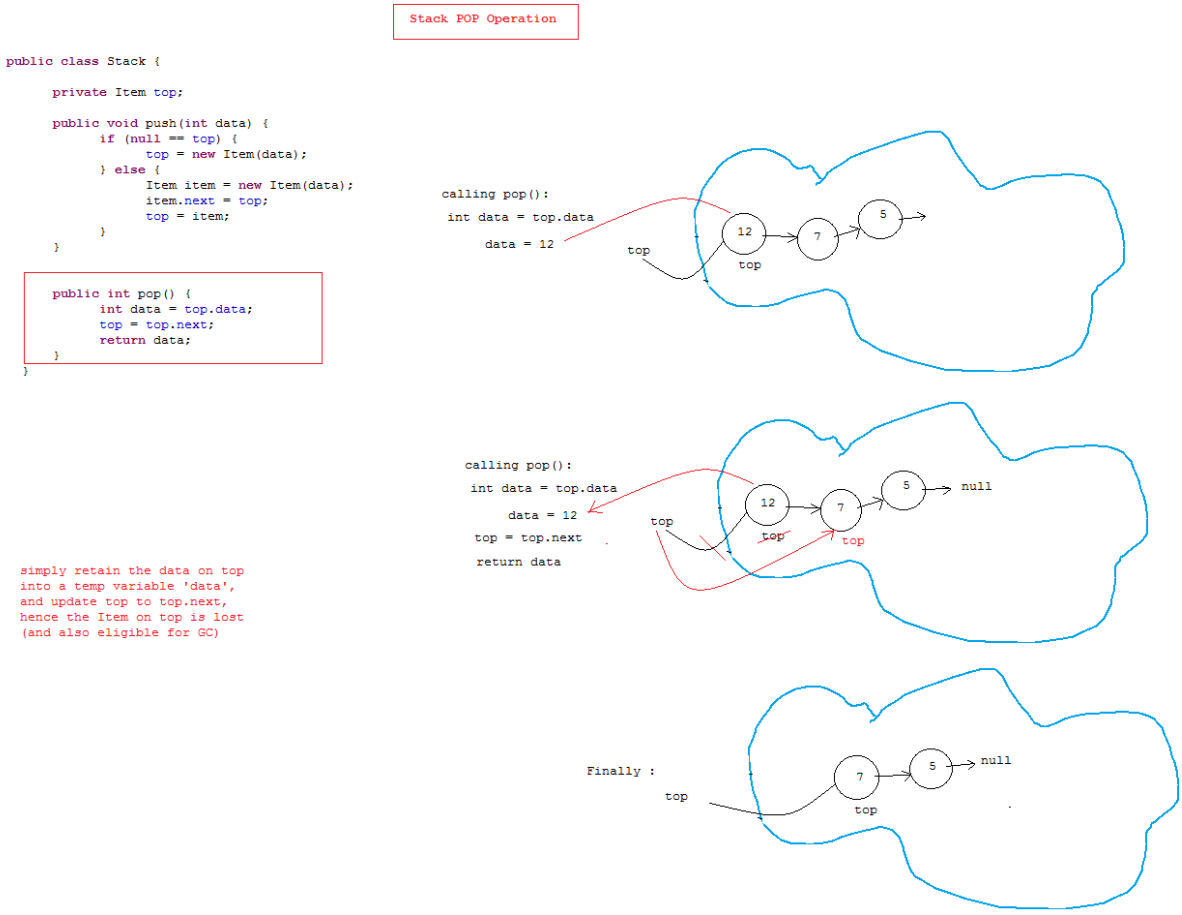






The whole idea is to create new object every time we get a push(data) request, amke it point to the old stack so that the new object becomes the new top,
finally update the top of the stack to that object becomes part of the whole stack.





So basically the push and pop both are happening at the same end, from the top side, hence we have our stack in place with LIFO implementation.

Below is the code for entire operation:

```
class Item {
	intdata;
	Item next;

	public Item(int data) {
		this.data = data;
	}
}


public class Stack {

	private Item top;
```

```
	public void push(int data) {
		if (null == top) {
			top = new Item(data);
		} else {
			Item item = new Item(data);
			item.next = top;
			top = item;
		}
	}


	public int pop() {
		int data = top.data;
		top = top.next;
		return data;
	}


	public void printStack() {
		Item tmp = top;

		if (top == null) {
			System.out.print("Stack is empty !!");
		}
		System.out.println();
		while (tmp != null) {
			System.out.print(" > " + tmp.data);
			tmp = tmp.next;
		}
	}


	public static void main(String[] args) {
		Stack stack = new Stack();
		stack.printStack();
		stack.push(5);
		stack.push(14);
		stack.push(3);
		stack.push(23);
		stack.printStack();
		System.out.print("\npoped: " + stack.pop());
		System.out.print("\npoped: " + stack.pop());
		stack.printStack();

	}

}
```

Queue:

The Queue is a FIFO Data Structure, and so it means that it would have Insertion at one end of the Object chain, and removal at the other end. This also gives us an idea that we are going to need two inner references for implementing our Queue, one tracking the top of the object chain (called front) and other tracking the bottom of the object chain(called rear).

We have two operations enqueue(data) and dequeue() in Queues. Below is the code for the Queue operation and is very similar to that of Stack.

Bring out your paper and pencil and draw the heap, and understand how objects are goint to be created for the Queue.

> ***Note**: If you can understand how to play with both the references in the Queue, Linked List would just be a cake walk for you. Go on play with the code, and even try to write it yourself.*

```
class Item {

	int data;
	Item next;

	public Item(int data) {
		this.data = data;
	}
}


public class Queue {

	private Item front;
	private Item rear;

	public void enqueue(int data) {
		if (rear == null) {
			front = rear = new Item(data);
		} else {
			Item item = new Item(data);
			front.next = item;
			front = item;

		}
	}


	public int dequeue() {
		int data = rear.data;
		rear = rear.next;
		return data;
	}
```

```
	public void printQueue() {
		if (rear == null) {
			System.out.println("Queue is Empty");
			return;
		}
		Item tmp = rear;
		System.out.print("REAR");
		while (tmp != null) {
			System.out.print("<" + tmp.data);
			tmp = tmp.next;
		}
		System.out.println("<FRONT");
	}


	public static void main(String[] args) {

		Queue q = new Queue();
		q.printQueue();
		q.enqueue(1);
		q.enqueue(2);
		q.enqueue(3);
		q.enqueue(4);
		q.printQueue();
		System.out.println("Dequeue :" + q.dequeue());
		System.out.println("Dequeue :" + q.dequeue());
		q.printQueue();

	}

}
```

Linked List:

Well if you've finished above two code bits, and have tried to write the code yourself you must not be scared at all because all this time we were playing with a Linked List's code only. Till now we were using the Linked List's concept to emulate a Stack and a Queue. Now its time for the Linked List itself.

A Linked List can have plenty of utility functions over it, and I've tried to put in some 5-6 methods here in code. That is also why it looks so lengthy, so no need to worry if you're not giving all the implementations to the interviewer.

But linked lists are quite favorite to guys out there, and its really fun trying to solve Linked List problems, so I suggest DO NOT look the code below before you've tried it out yourself.

You have already implemented Queue, so Linked List is much easier that it. Please try out few methods like insertAtFront(data), insertAtIndex(data, index), insertAtEnd(data), removeFromIndex(index), removeFromFront() etc yourself, then compare with the code

provided. Start with the simpler ones (front/end operations)then move to the complex ones (index based).

> ***Note** : One thing to keep in mind, you are using a linked list and you have to traverse through all N-1 elements to reach to the N-th element.*

Here is our implementation for the same:

```
class Node {
	int data;
	Node next;

	public Node(int data) {
		this.data = data;
	}
}



public class LinkedList {

	private Node root;


	public void addAtEnd(int data) {
		if (root == null) {
			root = new Node(data);
			return;
		}

		Node tmp = root;
		while (tmp.next != null) {
			tmp = tmp.next;
		}

		tmp.next = new Node(data);
	}


	public void addAtFront(int data) {
		Node newRoot = new Node(data);
		newRoot.next = root;
		root = newRoot;
	}


	public void addAtIndex(int data, int index) {
		if (root == null)
			return;
```

```
            Node tmp = root;
            while (index > 1) {
                  if (tmp == null)
                        return;

                  tmp = tmp.next;
                  index--;
            }

            Node nextNode = tmp.next;
            tmp.next = new Node(data);
            tmp.next.next = nextNode;
      }


      public int removeFromFront() {
            int data = root.data;
            root = root.next;
            return data;
      }


      public int removeFromEnd() {
            Node tmp = root;
            while (tmp.next.next != null) {
                  tmp = tmp.next;
            }
            int data = tmp.next.data;
            tmp.next = null;
            return data;
      }


      public int removeFromIndex(int index) {
            int data = 0;
            Node tmp = root;

            if (index < 0)
                  return -1;
            if (index == 0) {
                  data = root.data;
                  root = root.next;
                  return data;
            }

            while (index > 0) {
                  if (tmp == null) {
                        return -1;
                  }
                  tmp = tmp.next;
                  index--;
            }
            data = tmp.data;
            tmp.data = tmp.next.data;
            tmp.next = tmp.next.next;
```

```
            return data;
      }



      public void printList() {
            if (root == null) {
                  System.out.println("List is Empty !!");
                  return;
            }
            Node tmp = root;
            System.out.println();
            while (tmp != null) {
                  System.out.print(">" + tmp.data);
                  tmp = tmp.next;
            }
      }



      public staticvoid main(String[] args) {
            LinkedList list = new LinkedList();
            list.printList();
            list.addAtEnd(1);
            list.printList();
            list.addAtFront(2);
            list.printList();
            list.addAtEnd(3);
            list.printList();
            list.addAtIndex(4, 2);
            list.printList();
            list.addAtIndex(5, 4);
            list.printList();
            list.addAtEnd(6);
            list.printList();
            list.addAtFront(7);
            list.printList();
            list.addAtEnd(8);
            list.printList();

            list.removeFromEnd();
            list.printList();
            list.removeFromEnd();
            list.printList();
            list.removeFromFront();
            list.printList();
            list.removeFromIndex(0);
            list.printList();
            list.removeFromIndex(2);
            list.printList();

      }
}
```

Homework: I'm expecting that you've tried out all the above utility methods, so here is a new scenario I'm giving to test you:

> *I need a new method 'removeAll(data)' where if I call the method on the Linked List, it should remove all the occurrences of the data from the Linked List.*
> *E.g.*
> *List: 2>1>5>7>4>6>5>8>2*
> *list.removeAll(5)*
> *List : 2> 1> 7> 4> 6> 8> 2*
>
> *C'mon give it a shot.*
> *If you're done, try implementing a binary tree.*

Comparator vs Comparable. When to choose which.

Comparator and Comparable are two java interfaces which are used to compare any two objects. The most common utility of Comparator and Comparable would be to sort your objects.
Let's take an example of an Employee Object, where the salary of the Employee would decide who is superior over other. So if we want to sort the Employee objects they would be sorted by their salary. Here is the Employee Object which we would be comparing in the code blocks below:

```
class Employee
	{
		public String name;
		public int salary;
	}
```

Below is the individual implementation via Comparator and Comparable :
Comparator :
Comparator is the java interface which contains the 'int compare(Object obj1, Object obj2)' method which takes in two Objects as parameter and decides the equality of the two objects based on your logic. The compare() method returns a int value, where Zero represents two equal objects, -ve value represents that the 1st object was smaller than the second object, where a +ve value tells us that the 2nd object is smaller than the 1st object.

```
class SalaryComparator implements Comparator{

	public int compare(Object e1, Object e2) {
		return ((Employee)e1).salary- ((Employee)e2).salary;
		}
	}
```

Comparable :
Comparable is another Interface provided by java which works very similar to the Comaparator Interface, with few differences. The Comparator Interface has the 'int compareTo(Object obj)' method and it is present inside your class. As the name tells – Comparable makes your class comparable, hence its objects are comparable and can be now compared. compareTo() also returns a int value similar to the compare() method of the Comparator interface.

```
class Employee implements Comparable {
      public String name;
      public int salary;

      public int compareTo(Object emp) {
            return this.salary - ((Employee)emp).salary;
            }

      }
```

***Note**: Comparable needs the Employee Class itself to be modified, hence there is more coupling. Whereas the Comparator is a pluggable component which does not nood to modify your Employee class to be modified.*

Discussion :

Like you can see Comparator was a different class which you can use in any code to sort Employee Objects, whereas the Comparable makes your own class comparable i.e. comparator's code comes inside your own class, and here in our example both are serving the same purpose of sorting the Employee objects by their salary.

Well they are definitely very close to each other but there are certain situations where one might clearly seem to be a better option over the other.

Below are few common situations using Comparable & Comparator :

- *Comparable can be used when you're comparing the instances of the same class, whereas comparable would be of use when you are comparing two different class's objects like of class Employee and class Politician. Who do you think would have more salary ?*
- *If the sorting/comparison is a very obvious and common in your application and is going to be used very frequently - use Comparable.*
- *If the comparison is not very clear and it might not be used very frequently, on a safe side go for a Comparator. Its cool to have the comparator outside your class. Its less coupled and and more simpler to maintain by some other person.*
- *If there are more than one Comparison policies (i.e by salary, by age, by hmm.. yrs. of experience) then its better to have a comparator, where you can have all your code in a separate external class.*

Usage :

```
public void someMethod(){
	/** Create a Emp List and add dummy values to Sort **/
	List<Employee> myEmpList = new ArrayList<Employee>();
	Employee emp = new Employee();
	emp.salary = 5000;
	emp.name = "Arnold";
	myEmpList.add(emp);
	emp = new Employee();
	emp.salary = 15000;
	emp.name = "Clooney";
	myEmpList.add(emp);
	emp = new Employee();
	emp.salary = 500;
	emp.name = "Brad";
	myEmpList.add(emp);

	/** Comparable **/
	Collections.sort(myEmpList);

	/** Comparator **/
	Collections.sort(myEmpList, new SalaryComparator());

}
```

What are the best practices you follow while coding ?

Yes I know, you might be thinking I probably messed up and put the question in the wrong section, but trust me this innocent little question is gonna make your life miserable in most of the interviews, and people out there are really serious about this silly question. So lets get straight to few of the best practices without wasting much time :

Naming Conventions & Indentations: Please strictly follow the naming conventions for the Classes, methods and variables. It makes your code more readable. Also don't forget your indentations.

Avoid deprecated classes : Do not use the deprecated classes unless its utmostly unavoidable. They are depricated because of a reason. Use HashMap instead of HashTable, use ArrayList instead of Vectors. They have got better performance over the deprecated classes.

Strings/StringBuffers/StringBuilders: You must carefully decide which one you're going to use. Do not use Strings for frequently changing Strings. Do not use StringBuffer for un-threaded environments. Choose carefully.

Close Open Streams : Please close the Taps you've opened, we're not here to waste resources. PLease close File, InputStreams, OutputStreams, Connections, ResultSets, Statements etc etc. You've opened it keep in mind to close them when u're sone.

Proper exception handling : Exception handling is going to hide the ugly face of your application. When your app fails your exception handling mechanisms are going to talk to the end user, so better keep it neat and sweet. Also, please please avoid Exceptions thrown from the 'finally' block.

Use primitives instead of Wrapper classes : Primitives are just values but the Wrapper classes are classes and hence come with certain amount of overhead. Morever we make really silly coding errors while using the Wrapper classes sometimes, eg. using == operator on objects of two Integer objects. Two Integer objects are Objects not int values, and hence are not equal to each other. Try out printing :

```
System.out.println(new Integer(1) == new Integer(1));
```

Return not-null values from methods : Whenever you're going to return certain Objects/Lists/Maps etc from your value, and there is no value to return, do not return a null from the method. Instead return an empty String, blank Object or an empty lists. This avoids un-planned null pointer exceptions in the calling code.

Never throw/catch Exception unless you have a more specific type of Exception. Eg. use SQLException rather than Exception. It makes the code more specific, and of-course makes the life of debugger much more easier.

Prefer Interfaces over the Abstract classes : When you are planning to make a blue print of any plan, better keep it as an interface and then extend it by your class, rather than making abstract classes for everything. Makes your code clear.

Make instance variables private : Always make the class's instance variables private, and provide public getters and setter for getting/setting the values. Note, this also enables to to put some extra check in your setters where you can restrict anyone fron setting any invalid value to your Object.

That's pretty much I can think of now, but this definitely is not the final list, there ought to be many more. Just Google It if you're hungry ..

What do you know about java concurrency model?

Well this is not a question you would expect in every interview, but there are certain interviewers who really love digging inside the java internals. Just for them here is a short note on Java Concurrency.
Concurrency is a programming ability to achieve parallelism in your code, where you can run several sub-parts of your code in parallel to gain performance. Why concurrency models are becoming popular is because of the increase in the multi-core CPU's (Dual core, Quad core etc etc). The software therefore must fully utilize all the h/w advantages to get out best performance.
Now, there are two major concurrency models:

- Shared Nothing model
- Share State model

The **Share Nothing** model is a light weight model where all the sub-processes work absolutely isolated with each others, each being a self-contained process. Any data passing or information request takes place via asynchronous message passing. Due to the non-shared nature of the model, the programmer is restricted in terms of data access but gains advantage of the light concurrency model. This model is implemented by languages like Scala (known as Actor Model).

The **Share State model** on the other hand puts importance on sharing the important data. The access to these data is ensured by synchronization and context locks. Also its a point to note that sharing states comes with its trade-off, so the model becomes little heavy compared to Share Nothing model because it has to deal with maintaining stacks for tracking the locks and context switches which it would use for achieving synchronization. Java uses the Share State concurrency model.

Many programmers think that the Share nothing model is better due to its light nature, whereas many other think that Share State model is better since it would be really difficult to make/pass messages for every tine operation. The choice of the concurrency model comes coupled along with the choice of programming language.

How does java allocate Stack and Heap Memory?

Java Memory Allocation is a very frequent java interview question, specially for those interviewers who are expecting you to have some knowledge of the Java internals.

> *Sometimes the questions are not as straight as above, but are more like :*
>
> *Q: Where do the Objects land?*
> *Q: Where do the literals land?*
> *Q: Is String str = "abc"; treated as a primitive ?*
> *Q: Does the above string land in the same place where the objects land?*
> *Q: State the difference b/w Stack and Heap Memory…etc etc.*

Here is a short note on Java Memory Allocation:

Java has got two primary types of memory – The **Stack Memory** and the **Heap Memory**. Apart from them java has another special memory known as the **Non-Heap Memory**, which again has two parts – Code *memory* and *the Static memory*. Let's read little through their features and differences. Then we'll read how Java uses them.

Stack Memory: Stack memory is a temporary memory, and that means that whatever you're saving into it would be totally deleted once your program has finished its execution. Any local variable (variable, not Object) in some method in your program would also be on the stack memory and would be deleted as soon as the method finishes its execution.

Heap Memory: The Heap Memory is the permanent memory and is allocated to every program via the OS. When the JVM starts it gets some memory from the OS which it uses for all of its programs. The heap memory is not cleared when the program finishes its execution, so the objects created by you still stay in the heap memory after your program has finished execution. The JVM runs its Garbage Collector periodically to clean all the Objects on the heap and frees memory by removing all un-referenced Objects. In traditional programming languages this memory flush step was done by the programmer himself. The Heap memory is also used primarily for Dynamic Memory Allocation.

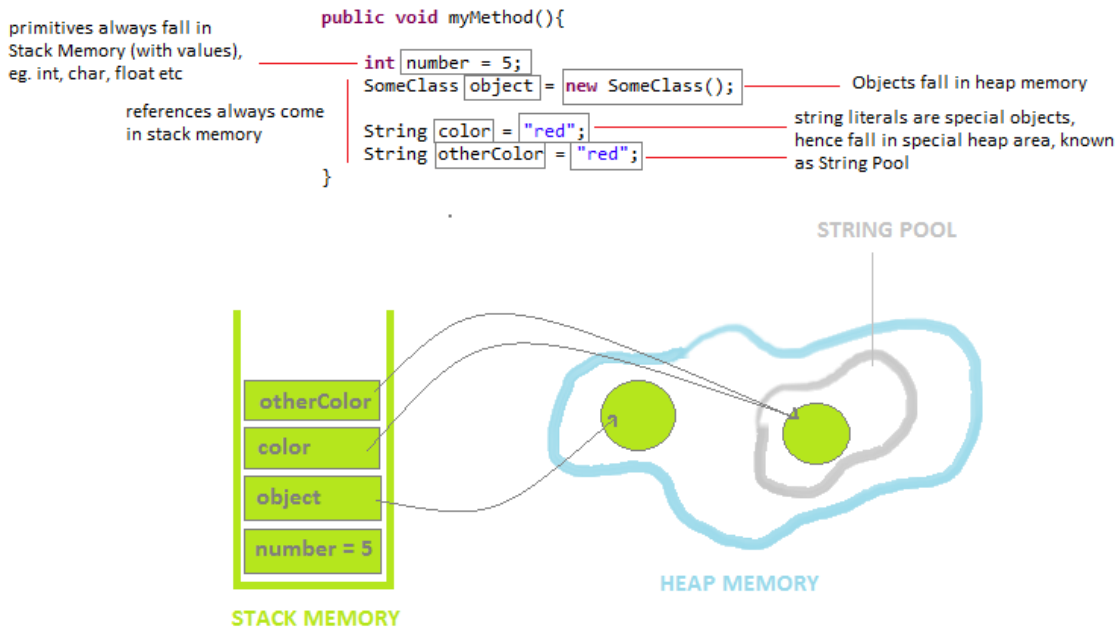
Static memory: Its a part of the Non-heap Memory. Java would use the memory to store the static variables and the static methods. Java would also use it to save some metadata related to class's info etc. This area is also referred to as the Perm Generation area.

Code memory: Its the second part of the Non-Heap memory. The code memory would be used by JVM to save your program's byte code, since JVM needs to have the code in memory to execute it line by line.

These are the rules java uses to allocate memory :

1. *All the primitives would land in the stack memory, along with their values.*
   E.g. int num = 5; entirely lands in stack, both num and 5.
2. *All the Objects land in the Heap Memory.*
3. *Any string literal is itself a Object so lands in the Heap Memory. But since java uses string pooling so String literal land in a special heap area known as the String Pool.*
4. *All the references land in the Stack Memory.*
5. *All static data goes in the Static memory.*

Below is a diagram showing the usage of Stack and heap memory for the given block of code :



What is a Memory Leak? Can you create one?

Memory Leak is nothing but a bad state a program has landed in where it is consuming a huge amount of energy but is not able to release it back. Memory leaks are horrible and I don't know why would someone ask to write a code to create a memory leak, but yes they do ask this question and you would not like to question his question.
So lets see a few cases which land up with memory leak scenarios, Note the key is more and more memory consumption without thinking of releasing it. For the first we're getting points for our Bad Coding Practices.
Cool .. so we just have to write code to consume all our memory so that the memory usage process is faster than the memory cleaning process :

Most common thing that'll come to your mind: Keep creating objects in long long loop. So we would see huge memory consumption.

Use static class variables, holding values of huge objects. It would help us explode the PermGen memory.

```
class TempClass {

      static String[] hugeArray = new String[99999999];

}
```

Connections are the most expensive resources, so lets waste them. Never close Connections. No need of connection.close(); in the finally block.

Never close any **Streams** like InputReader, InputWriter, OutputReader, Statements, ResultSets, Nothing. Ultimate bad coding practice, but it would fetch us cool points here.

If its a web application, keep on adding huge values to **session** till it explodes. We can also add huge values to **Application context** of the application till it explodes and your app dies in no time due to memory crunch.

Use String.intern() on huge strings. This would put the string in the String Pool and would not be cleared. Wow, now we learned to blow the String pool too. Cool.

```
String myString = "blah.. blah.. blah.. blah.. blah.. blah.. " +
                  "blah.. blah.. blah.. blah.. blah.. blah.. blah.. " +
                  "blah.. blah.. blah.. blah.. blah.. blah.. blah.. " +
```

```
                "blah.. blah.. blah.. blah.. ";

    myString.intern();
```

Another memory overhead can be using **new String() every time instead of String literals**, therefore forcing JVM to create new objects forcibly instead of taking advantage of string pooling.

Create huge number of **threads** in your app, but don't start them.

Another interesting method is java's File.deleteOnExit(). This method is generally used for deleting temporary files used/created in your app, but the interesting thing is that it guarantees a memory leak. Every time a file needs to be deleted via deleteOnExit it takes up about 1kb of memory, which is only recovered on system exit. So you can see how we can exploit it for our leak requirement.

```
    File tempFile = File.createTempFile("MyFile", ".txt");
    tempFile.deleteOnExit();
```

What are Weak/Soft/Phantom References?

This is another question which is quite uncommonly answered by many candidates but there are quite a number of places where we have got to use them. I agree this answer can be little lengthy read, but it's gonna be worth it. It's a rarely found cool concept.

Let's start with the basic Object/reference creation style:

Object obj = new Object();

By this normal Object creation we have a reference 'obj'. This is what we call a Strong reference.

Now you may ask the question....
Why bother about this Reference thing?
Being a java programmer... This is the most common statement/style we usefor Object creation. Isn't it?

I agree... But there is anobvious catch most of us generally don't bother of while normal coding. The reference 'obj' refers to an object stored in the heap. As long as the obj reference exists, the garbage collector will never free the heap space. The object is available for GC only when there are no references to it, or it goes out of scope, or it is explicitly assigned to null.

However, an important detail to note is that just because an object is available for collection does not mean it will be reclaimed by a given run of the garbage collector. Because garbage collection algorithms vary, some algorithms analyze older, longer-lived objects less frequently than short-lived objects. Therefore, an object that is available for collection may never be reclaimed. This can occur if the program ends prior to the garbage collector freeing the object. Hence, the bottom line is that you can never guarantee that an available object will ever be collected by the garbage collector.

What can we do then?

Java allows us few more styles to create references apart from the traditional 'Strong reference' way. So, let's play a little with references.

Java provides 4 types of references:

- ***Strongly reachable:*** *An object that can be accessed by a strong reference. Like the one 'obj' we created above.*

- ***Softly reachable:*** *An object that is not strongly reachable and can be accessed through a soft reference.*
- ***Weakly reachable:*** *An object that is not strongly or softly reachable and can be accessed through a weak reference.*
- ***Phantomly reachable:*** *An object that is not strongly, softly, or weakly reachable, has been finalized, and can be accessed through a phantom reference.*

The above definitions must be very vague, but that's how it is documented. Now let's actually understand what they are with little more details:

The ReferenceQueue -
Let's talk little about the ReferenceQueue before moving to the References. A ReferenceQueue is a special queue which is used to track the garbage collected references. The speciality of ReferenceQueue is that if it is used along with soft/phantom/weak references, it can be used to track when the object has been garbage collected. As soon as the object is garbage collected it would be queued to the ReferenceQueue.

The SoftReference class -
The idea of a SoftReference is that you hold a referenceto an object with one guarantee - that all of your soft references will be cleared before the JVM reports an out-of-memory condition. The key point is that when the garbage collector runs, it may or may not free an object that is softly reachable. Whether the object is freed depends on the algorithm of the garbage collector as well as the amount of memory available while the collector is running.But if there is an Out-Of-Memory all the SoftReferences would definitely be cleared.
A typical use of the SoftReference class is for a memory-sensitive cache.

The WeakReference class -
In addition, weak references are useful for objects that would otherwise live for a long time and are also inexpensive to re-create. The key point is that when the garbage collector runs, if it encounters a weakly reachable object, it will free the object the WeakReference refers to. Note, however, that it may take multiple runs of the garbage collector before it finds and frees a weakly reachable object.
A typical use of the WeakReference class is for canonicalized mappings.Canonicalized map is an in-memory map which is referred by different parts of your code.

The PhantomReference class -
APhantomReference must be used with the ReferenceQueue class. The ReferenceQueue is required because itserves as the mechanism of notification. When the garbage collector determines an object is phantomly reachable, the PhantomReference object is placed on its ReferenceQueue.The placing of the PhantomReference object on the ReferenceQueue is your notification that the object the PhantomReference object referred to has been finalized and is

ready to be collected. This allows you to take action just prior to the objectmemory being reclaimed.
The PhantomReference class is useful only to track the impending collection of the referring object. As such, it can be used to perform pre-mortem cleanup operations.

Code for creating Soft/Weak/Phantom References:

```
ReferenceQueue refQueue = new ReferenceQueue();
WeakReference weakRef = new WeakReference(new SomeClass(), refQueue);

ReferenceQueue refQueue = new ReferenceQueue();
SoftReference softRef = new SoftReference(new SomeClass(), refQueue);

ReferenceQueue refQueue = new ReferenceQueue();
PhantomReference phantomRef
                  = new PhantomReference(new SomeClass(), refQueue);
```

And the value of the object can be accessed by the methods:

```
   // used before GC. Same syntax for softRef & phantomRef.
   // get() method returns null if the object has been GC'd

   Object obj = weakRef.get();

   // can be used after being GC'd.
   WeakReference wkRef = (WeakReference) refQueue.poll();
```

Here is a code which shows you how JVM handles all the references. Please try out the code in you IDE and try changing the code and observe the O/P in each case:

```
import java.lang.ref.PhantomReference;
import java.lang.ref.ReferenceQueue;
import java.lang.ref.SoftReference;
import java.lang.ref.WeakReference;



class SomeClass{


      /** toString() is called every time we print the object. **/
      public String toString(){
            return "I am an Object, Still in Heap";
      }
```

```
      /** finalize() would be called just before the object dies **/
      Public void finalize(){
            System.out.println("The GC just shot me..I am dying ..");
      }
}

Public class References {

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

      ReferenceQueue refQueue = null;


      System.out.println("\n--- Strong Ref -------------------");
      SomeClass strongRef = new SomeClass();
      System.gc();Thread.sleep(2000);
      System.out.println("Post GC: "+strongRef);


      System.out.println("\n--- Weak Ref ---------------------");
      refQueue = new ReferenceQueue();
      WeakReference weakRef
                  = new WeakReference(new SomeClass(),refQueue);
      System.out.println("Pre GC, weakRef.get(): "+weakRef.get());
      System.out.println("Pre GC, refQueue.poll(): "+refQueue.poll());
      System.gc(); Thread.sleep(2000);
      System.out.println("Post GC, weakRef.get(): "+weakRef.get());
      System.out.println("Post GC, refQueue.poll(): "+refQueue.poll());


      System.out.println("\n--- Soft Ref ---------------------");
      refQueue = new ReferenceQueue();
      SoftReference softRef
                  = new SoftReference(new SomeClass(), refQueue);
      System.out.println("Pre GC, softRef.get():"+softRef.get());
      System.out.println("Pre GC, refQueue.poll():"+refQueue.poll());
      System.gc();Thread.sleep(2000);
      System.out.println("Post GC, softRef.get():"+softRef.get());
      System.out.println("Post GC, refQueue.poll():"+refQueue.poll());


      System.out.println("\n--- Phantom Ref ------------------");
      refQueue = new ReferenceQueue();
      PhantomReference phantomRef
                  = new PhantomReference(new SomeClass(), refQueue);
      System.out.println("Pre GC, phantomRef.get():"+phantomRef.get());
      System.out.println("Pre GC, refQueue.poll():"+refQueue.poll());
      System.gc();Thread.sleep(2000);
      System.out.println("Post GC, phantomRef.get():"+phantomRef.get());
      System.out.println("Post GC, refQueue.poll():"+refQueue.poll());

      }
}
```

Working with java internals

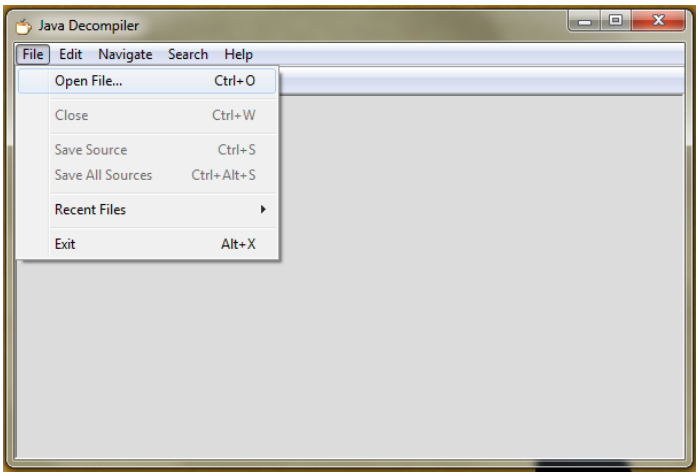
Another interview question to test your geeky half. While all coders are busy writing pretty code in Java, there are few elite coders that find it fascinating tweaking Java's own code and playing around with it. Java's implementation is already very pretty, but there may be situations where your weird requirement may force you to tweak the Java's libraries a little to get your work done.

In this answer we are not going to tweak any Java libraries, but yes we'll see how we can do it, so that next time someone asks if you've worked with Java internals, you can confidently say – yes sir have played around with it some while back. Or next time when you have a tricky requirement you can reach to Java implementation and just make it suit your requirement.

Now we have two ways to play with java internals, either download the Source Code and start reading how the classes are written and try understanding the code. While this is much easier way of doing it, but most places in your office/labs you would only be having the binary distribution and jars. So what we're gonna do here is, we'll download a Java Decompiler. So that we can simply open our jars and class files to see the corresponding code.

I personally use this little decompiler JD-GUI, its small around 700kbs, it does not need installation- just double click and run, and best of en' all – Its absolutely free. Download it from this URL so that we can move ahead towards the fun part : http://java.decompiler.free.fr/?q=jdgui

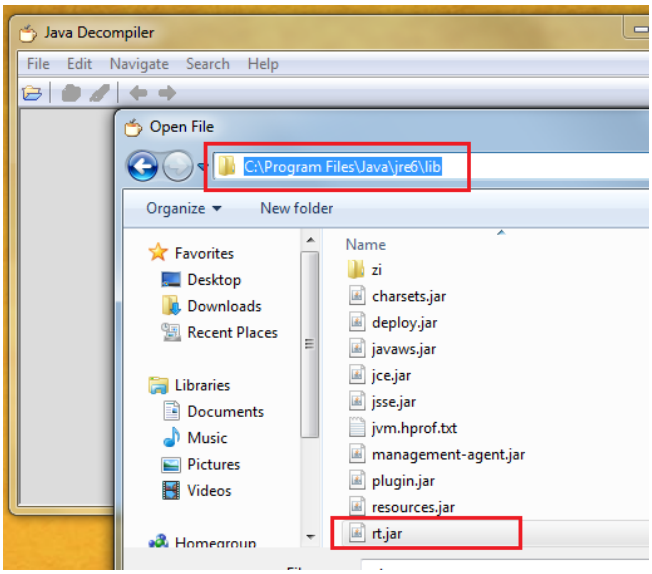
The JD-GUI looks like this :



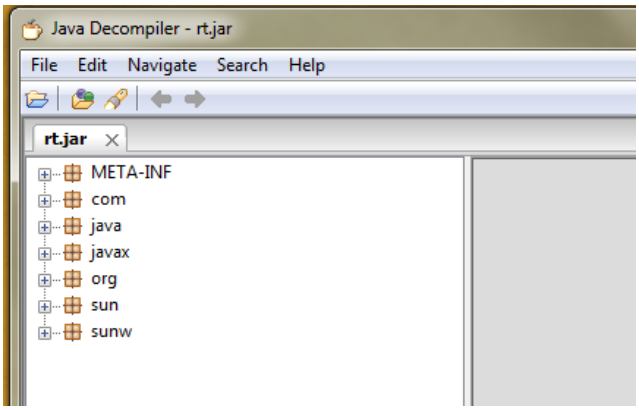
Let's just start digging Java :

Click: File> Open File >

Browse to the java installation directory. In the **lib** folder you'd find a jar **rt.jar**. Open the file in the decompiler.

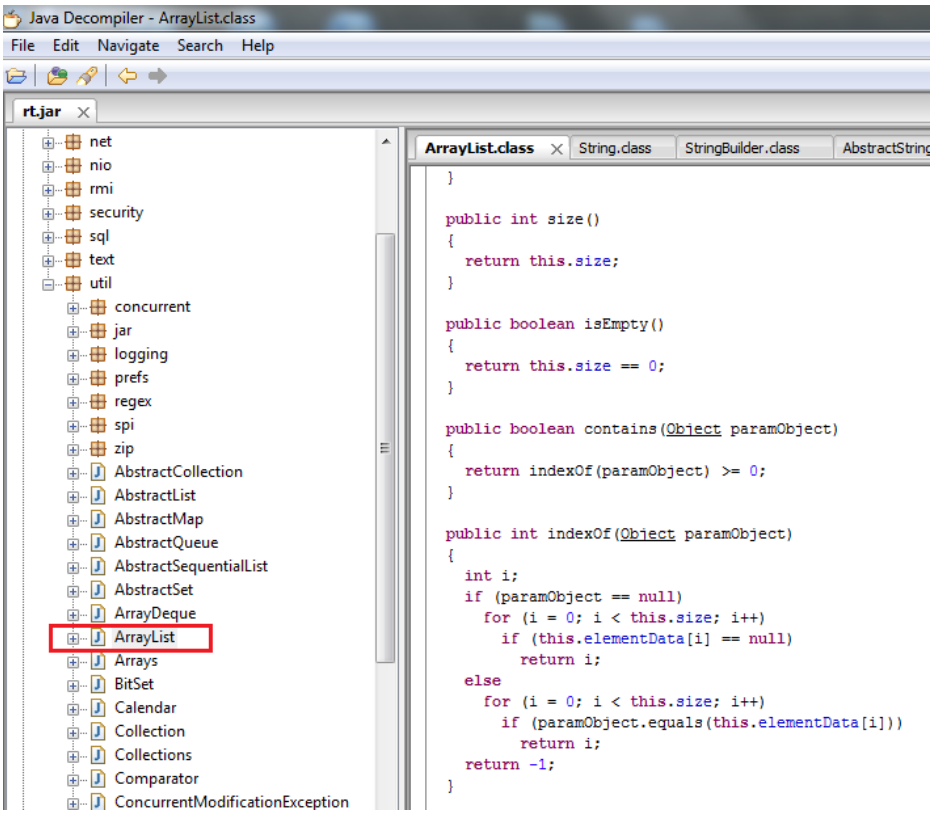


This is how rt.jar looks like :



Now let the fun begin, let's see some of Java's code.
What is the first java class that comes to your mind?
For me its ArrayList, probably because I was just stuck with a issue with it 10 minutes back. So let me hunt down ArrayList. I know it is in the java.util package, so let me expand the packages on the left and hunt for it.

See I found it here, and here is its internal code :



See.. it is so simple and so noob that I wonder why did I even put it in the book. But yes it's a very important thing we should all know, because you never know when you might think like improving Java's API and write down better ones for us.

Don't stop here, just hunt down other popular java classes like, String, StringBuffers, HashMaps etc.

Go look for java's implementation of LinkedList and check if they have written a better code than us .. Go have fun with the internals & don't be content with what Java has provided you. Break their code and write better ones. Ending the book with this note.

Best of luck for your interviews.


the world is Open Source ..