**CLASS Loader**

***Question : What is ClassLoader in Java?***

***Answer*** : ClassLoader in Java is a class which is used to load class files in Java. Java code is compiled into class file by javac compiler and JVM executes Java program, by executing byte codes written in class file. ClassLoader is responsible for loading class files from file system, network or any other source. There are three default class loader used in Java, Bootstrap , Extension and System or Application class loader. Every class loader has a predefined location, from where they loads class files. Bootstrap ClassLoader is responsible for loading standard JDK class files from rt.jar and it is parent of all class loaders in Java. Bootstrap class loader don't have any parents, if you call String.class.getClassLoader() it will return null and any code based on that may throw NullPointerException in Java. Bootstrap class loader is also known as Primordial ClassLoader in Java.  Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by  sun.misc.Launcher$ExtClassLoader. Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from CLASSPATH environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$AppClassLoader class. Also, except Bootstrap class loader, which is implemented in native language mostly in C,  all  Java class loaders are implemented using java.lang.ClassLoader.

In short here is the location from which Bootstrap, Extension and Application ClassLoader load Class files.

1) Bootstrap ClassLoader - JRE/lib/rt.jar

2) Extension ClassLoader - JRE/lib/ext or any directory denoted by java.ext.dirs

3) Application ClassLoader - CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside JAR file.

***Question : How ClassLoader works in Java?***

***Answer***: As I explained earlier Java ClassLoader works in three principles : delegation, visibility and uniqueness. In this section we will see those rules in detail and understand working of Java ClassLoader with example. By the way here is a diagram which explains How ClassLoader load class in Java using delegation.

**Delegation principles**

As discussed on when a class is loaded and initialized in Java, a class is loaded in Java, when its needed. Suppose you have an application specific class called Abc.class, first request of loading this class will come to Application ClassLoader which will delegate to its parent Extension ClassLoader which further delegates to Primordial or Bootstrap class loader. Primordial will look for that class in rt.jar and since that class is not there, request comes to Extension class loader which looks on jre/lib/ext directory and tries to locate this class there, if class is found there than Extension class loader will load that class and Application class loader will never load that class but if its not loaded by extension class-loader than Application class loader loads it from Classpath in Java. Remember Classpath is used to load class files while PATH is used to locate executable like javac or java command.

**Visibility Principle**

According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which mean if class Abc is loaded by Application class loader than trying to load class ABC explicitly using extension ClassLoader will throw either java.lang.ClassNotFoundException. as shown in below Example

**Uniqueness Principle**

According to this principle a class loaded by Parent should not be loaded by Child ClassLoader again. Though its completely possible to write class loader which violates Delegation and Uniqueness principles and loads class by itself, its not something which is beneficial. You should follow all  class loader principle while writing your own ClassLoader.

**Serialization  
  
*Question : What is Serialization?***  
***Answer :***Serialization helps us to save and retrieve the state of an object.  
  
Serialization => Convert object state to some internal object representation.  
De-Serialization => The reverse. Convert internal representation to object.  
Two important methods  
  
ObjectOutputStream.writeObject() // serialize and write to file  
ObjectInputStream.readObject() // read from file and deserialize  
How do you serialize an object using Serializable interface?  
  
To serialize an object it should implement Serializable interface. In the example below, Rectangle class implements Serializable interface. Note that Serializable interface does not declare any methods to be implemented.  
  
Below example shows how an instance of an object can be serialized. We are creating a new Rectangle object and serializing it to a file Rectangle.ser.  
  
class Rectangle implements Serializable {  
    public Rectangle(int length, int breadth) {  
        this.length = length;  
        this.breadth = breadth;  
        area = length \* breadth;  
    }  
  
    int length;  
    int breadth;  
    int area;  
}  
  
FileOutputStream fileStream = new FileOutputStream("Rectangle.ser");  
ObjectOutputStream objectStream = new ObjectOutputStream(fileStream);  
objectStream.writeObject(new Rectangle(5, 6));  
objectStream.close();  
How do you de-serialize in Java?  
  
Below example show how a object can be deserialized from a serialized file. A rectangle object is deserialized from the file Rectangle.ser  
  
FileInputStream fileInputStream = new FileInputStream("Rectangle.ser");  
ObjectInputStream objectInputStream = new ObjectInputStream(  
        fileInputStream);  
Rectangle rectangle = (Rectangle) objectInputStream.readObject();  
objectInputStream.close();  
System.out.println(rectangle.length);// 5  
System.out.println(rectangle.breadth);// 6  
System.out.println(rectangle.area);// 30  
What do you do if only parts of the object have to be serialized?  
  
We mark all the properties of the object which should not be serialized as transient. Transient attributes in an object are not serialized. Area in the previous example is a calculated value. It is unnecessary to serialize and deserialize. We can calculate it when needed. In this situation, we can make the variable transient. Transient variables are not serialized. (transient int area;)  
  
//Modified Rectangle class  
  
class Rectangle implements Serializable {  
    public Rectangle(int length, int breadth) {  
        this.length = length;  
        this.breadth = breadth;  
        area = length \* breadth;  
    }  
  
    int length;  
    int breadth;  
    transient int area;  
}  
If you run the program again, you would get following output  
  
System.out.println(rectangle.length);// 5  
System.out.println(rectangle.breadth);// 6  
System.out.println(rectangle.area);// 0  
Note that the value of rectangle.area is set to 0. Variable area is marked transient. So, it is not stored into the serialized file. And when de-serialization happens area value is set to default value i.e. 0.  
 ***Question : How do you serialize a hierarchy of objects?***  
***Answer*** : Objects of one class might contain objects of other classes. When serializing and de-serializing, we might need to serialize and de-serialize entire object chain. All classes that need to be serialized have to implement the Serializable interface. Otherwise, an exception is thrown. Look at the class below. An object of class House contains an object of class Wall.  
  
House implements Serializable. However, Wall doesn't implement Serializable. When we try to serialize an instance of House class, we get the following exception.  
  
class House implements Serializable {  
    public House(int number) {  
        super();  
        this.number = number;  
    }  
  
    Wall wall;  
    int number;  
}  
  
class Wall{  
    int length;  
    int breadth;  
    int color;  
}  
  
  
Output:  
Exception in thread "main" java.io.NotSerializableException: com.rithus.serialization.Wall  
    at java.io.ObjectOutputStream.writeObject0(Unknown Source)  
    at java.io.ObjectOutputStream.defaultWriteFields(Unknown Source)  
This is because Wall is not serializable. Two solutions are possible.  
  
Make Wall transient. Wall object will not be serialized. This causes the wall object state to be lost.  
Make Wall implement Serializable. Wall object will also be serialized and the state of wall object along with the house will be stored.  
Option 1  
  
class House implements Serializable {  
    public House(int number) {  
        super();  
        this.number = number;  
    }  
  
    transient Wall wall;  
    int number;  
}  
Option 2  
  
class Wall implements Serializable {  
    int length;  
    int breadth;  
    int color;  
}  
With both these programs, earlier main method would run without throwing an exception. If you try de-serializing, In Example2, state of wall object is retained whereas in Example1, state of wall object is lost.  
  
***Question : Are the constructors in an object invoked when it is de-serialized?***  
***Answer :***No. When a class is De-serialized, initialization (constructor’s, initializer’s) does not take place. The state of the object is retained as it is.  
  
***Question : What happen to the parent class data if the parent class has not implemented the serializable interface?***  
***Answer***: The data is not persisted along with the persistent object, and parent class constructor is called during the de-serialization.  
  
***Question : What happen to the parent class data if the parent class has also implemented the serializable interface?***  
***Answer***: The data is persisted along with the persistent object, and parent class constructor is not called during the de-serialization.  
  
***Question : Are the values of static variables stored when an object is serialized?***  
***Answer*** : Static Variables are not part of the object. They are not serialized, after de-serializing the object, if we try to access the value of static variable is most probably not same when we persisted that object, so if we try to access it using the serialized object, latest value will be shown, that if some other process has changed the data, then value which was last modified will be visible to displayed.  
  
***Question : If the parent class has implemented the serializable interface, then does it mean that child class are also serializable?***  
***Answer*** : Yes, that is normal inheritance principle that, child class will own most of the property of parent, which includes interface, public/protected method/attribute. So when a parent class implements Serializable then all the child class in bottom most hierarchy will be serializable.

**Exception :**

**Question:**  **Can we use Throwble instead of Exception, if yes then will there be any adverse effect?**  
**Answer :**Only diff is we may expect the error to be caught, and next thing is not good practice.  
  
**Question : Will the (Exception e) able catch runtimeException?**  
**Answer**: Yes, being parent of all Exception it can catch the **Runtime** exception as well.    
  
***Question : What are the adverse effect of catching Exception exception?***  
***Answer*** : Being parent of all exception, it will try to handle any exception, which we may want to handle non generic way.  
It will also catch the Runtime Exception as it is parent of that exception as well.  
It may also hide all the clarity on what are the other possible exception and hence give the code management team a difficulty while refactoring, as those scenario may be ignored or will be missed while handling.  
It gives really no clarity on what might have caused the real error unless we use debugger and see the step by step  and figure out what went wrong.

**Question: Explain the user defined Exceptions?**  
**Answer:** User defined Exceptions are the separate Exception classes defined by the user for specific purposed. An user defined can created by simply sub-classing it to the Exception class. This allows custom exceptions to be generated (using throw) and caught in the same way as normal exceptions.  
Example:  
class myCustomException extends Exception {  
   // The class simply has to exist to be an exception  
} 

**Question: How does a try statement determine which catch clause should be used to handle an exception?**  
**Answer :**When an exception is thrown within the body of a try statement, the catch clauses of the try statement are examined in the order in which they appear. The first catch clause that is capable of handling the exception is executed. The remaining catch clauses are ignored.

**Question: Does it matter in what order catch statements for FileNotFoundException and IOExceptipon are written?**

**Answer:** Yes, it does. The FileNoFoundException is inherited from the IOException. Exception's subclasses have to be caught first.

**Question: What is user-defined exception in java ?**

**Answer:** User-defined expections are the exceptions defined by the application developer which are errors related to specific application. Application Developer can define the user defined exception by inheriting the Exception class as shown below. Using this class we can throw new exceptions.

Example:  
class myCustomException extends Exception {  
   // The class simply has to exist to be an exception  
}

***Java Example :***public class noFundException extends Exception {

 }

Throw an exception using a throw statement:

 public class Fund {

 ... public Object getFunds() throws noFundException {

 if (Empty()) **throw**new noFundException(); ...

 }

 }

User-defined exceptions should usually be checked.

**Question: What is the difference between checked and Unchecked Exceptions in Java ?**

**Answer:** All predefined exceptions in Java are either a checked exception or an unchecked exception. Checked exceptions must be caught using try .. catch() block or we should throw the exception using throws clause. If you dont, compilation of program will fail.

***Java Exception Hierarchy***+--------+ | Object | +--------+ | | +-----------+ | Throwable | +-----------+ / \ / \ +-------+ +-----------+ | Error | | Exception | +-------+ +-----------+ / | \ / | \ \\_\_\_\_\_\_\_\_/ \\_\_\_\_\_\_/ \ +------------------+ unchecked checked | RuntimeException | +------------------+ / | | \ \\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ unchecked

**Question : Can user defined exception be also part of RuntimeException?**  
**Answer :**Yes it can be, while defining the class we need to extend the RuntimeException class or any subclass of RuntimeException class.  
  
***Question : Give example of User Defined Runtime Exception.***  
***Answer*** : Below is the example of the same

|  |
| --- |
| **class** MyUserDefinedRuntimeException **extends** RuntimeException{    }  **public** **static** **void** sayhello()  {          System.***out***.println("Parent saying hello");  **throw** **new** MyUserDefinedRuntimeException();  } |

***Question : Give example of User Defined Exception.***  
***Answer*** : Below is the example of the same\\

|  |
| --- |
| **class** MyUserDefinedException **extends** Exception{  }  **public** **static** **void** sayhello()  {  **try** {                  System.***out***.println("Parent saying hello");  **throw** **new** MyUserDefinedException();          } **catch** (MyUserDefinedException e) {                  // **TODO**: handle exception          }  } |

|  |
| --- |
| **class** MyUserDefinedException **extends** Exception{  }  **public** **static** **void** sayhello()  **throws** MyUserDefinedException{          System.***out***.println("Parent saying hello");  **throw** **new** MyUserDefinedException();  } |

**Question: What is the purpose of the finally clause of a try-catch-finally statement?**  
**Answer:** The finally clause is used to provide the capability to execute code no matter whether or not an exception is thrown or caught.  
 **Question: What classes of exceptions may be caught by a catch clause?**  
**Answer:** A catch clause can catch any exception that may be assigned to the Throwable type. This includes the Error and Exception types.

**Question: Can an exception be rethrown?**

**Answer:**Yes, an exception can be rethrown.

**Question : How to rethrow Exception?**

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

**int** a[][] = **new** **int**[3][];

**try** {

*recursion*();

                                } **catch** (Throwable t) {

                                                System.*out*.println("out of stack error");

**try** {

**throw** t;

                                                } **catch** (Throwable e) {

                                                                e.printStackTrace();

                                                }

                                }

                }

**public** **static** **void** recursion(){

*recursion*();

                }

The Method recursion will cause Stackoverflow error.

Output : out of stack error

java.lang.StackOverflowError

                at com.example.Main.recursion(Main.java:25)

**Question: When is the finally clause of a try-catch-finally statement executed?**

**Answer:**The finally clause of the try-catch-finally statement is always executed unless the thread of execution terminates or an exception occurs within the execution of the finally clause.

**Question: What class of exceptions are generated by the Java run-time system?**

**Answer:**The Java runtime system generates Runtime Exception and Error exceptions.

**Question: What is the relationship between a method's throws clause and the exceptions that can be thrown during the method's execution?**

**Answer:**A method's throws clause must declare any checked exceptions that are not caught within the body of the method.

**Question: What classes of exceptions may be thrown by a throw statement?**  
**Answer:**A throw statement may throw any expression that may be assigned to the Throwable type.

**Question: What happens if an exception is not caught?**

**Answer:**An uncaught exception results in the uncaughtException() method of the thread's ThreadGroup being invoked, which eventually results in the termination of the program in which it is thrown.

**Question: What happens if a try-catch-finally statement does not have a catch clause to handle an exception that is thrown within the body of the try statement?**

**Answer:**The exception propagates up to the next higher level try-catch statement (if any) or results in the program's termination.

**Question: Can try statements be nested?**  
Try statements may be tested.

**Question: What is difference between error and exception**

**Answer:**Error occurs at runtime and cannot be recovered, Outofmemory is one such example. Exceptions on the other hand are due conditions which the application encounters such as FileNotFound exception or IO exceptions

**Question: When do we say an exception is handled**

**Answer:**When an exception is thrown in a try block and is caught by a matching catch block, the exception is considered to have been handled  
  
**Question: When do we say an exception is not handled**

**Answer:**There is no catch block that names either the class of exception that has been thrown or a class of exception that is a parent class of the one that has been thrown, then the exception is considered to be unhandled, in such condition the execution leaves the method directly as if no try has been executed  
  
**Question: In what sequence does the finally block gets executed**

**Answer:**If you put finally after a try block without a matching catch block then it will be executed after the try block irrespective of final status of the try block execution, i.e completed successfully or encountered some exception.  
If it is placed after the catch block and there is no exception then also it will be executed after the try block  
If there is an exception and it is handled by the catch block then it will be executed after the catch block  
  
**Question: What can prevent the execution of the code in finally block**

**Answer:**

* The death of thread
* Use of system.exit()
* Turning off the power to CPU
* An exception arising in the finally block itself

**What are the rules for catching multiple exceptions**  
A more specific catch block must precede a more general one in the source, else it gives compilation error  
Only one catch block, that is first applicable one, will be executed  
  
**Question: What does throws statement declaration in a method indicate**

**Answer:**This indicates that the method throws some exception and the caller method should take care of handling it  
  
**Question: What are checked exception**

**Answer:**Checked exceptions are exceptions that arise in a correct program, typically due to user mistakes like entering wrong data or I/O problems  
  
**Question: What are runtime exceptions**

**Answer:**Runtime exceptions are due to programming bugs like out of bond arrays or null pointer exceptions.  
  
**Question: How will you handle the checked exceptions**

**Answer:**You can provide a try/catch block to handle it. OR Make sure method declaration includes a throws clause that informs the calling method an exception might be thrown from this particular method. When you extend a class and override a method, can this new method throw exceptions other than those that were declared by the original method. No it cannot throw, except for the subclasses of those exceptions.  
  
**Question: Is it legal for the extending class which overrides a method which throws an exception, not o throw in the overridden class**

**Answer:**Yes it is perfectly legal  
  
**Question: Explain the user defined Exceptions?**

**Answer:**User defined Exceptions are the separate Exception classes defined by the user for specific purposed. An user defined can created by simply sub-classing it to the Exception class. This allows custom exceptions to be generated (using throw) and caught in the same way as normal exceptions.

**Example:**  
  
class myCustomException extends Exception {  
// The class simply has to exist to be an exception  
} 

**Question: What is the purpose of the finally clause of a try-catch-finally statement?  
Answer:** The finally clause is used to provide the capability to execute code no matter whether or not an exception is thrown or caught.  
  
**Question: What classes of exceptions may be caught by a catch clause?  
Answer:** A catch clause can catch any exception that may be assigned to the Throwable type. This includes the Error and Exception types.  
 **Question:What is the difference between throw and throws keywords?  
Answer:** The *throw* keyword denotes a statement that causes an exception to be initiated. It takes the Exception object to be thrown as argument. The exception will be caught by an immediately encompassing try-catch construction or propagated further up the calling hierarchy. The *throws* keyword is a modifier of a method that designates that exceptions may come out of the mehtod, either by virtue of the method throwing the exception itself or because it fails to catch such exceptions that a method it calls may throw.  
 ***Question: Does the code in finally block get executed if there is an exception and a return statement in a catch block?  
Answer*:**  If an exception occurs and there is a return statement in catch block, the finally block is still executed. The finally block will not be executed when the System.exit(1) statement is executed earlier or the system shut down earlier or the memory is used up earlier before the thread goes to finally block.

***Question : What is the restriction for the checked exception?***

***Answer*** : There is no restriction for checked Exception but if there is case when code will never throw checked exception then compiler will  restrict us to catch it, so we can only catch the checked exception if it has real possibility to happen.

E.g.

**try** {

**int** a[][] = **new** **int**[3][];

*recursion*();

} **catch** (ClassNotFoundException e) {

                                                // **TODO**: handle exception

}

Above code will give compilation error : "Unreachable catch block for ClassNotFoundException. This exception is never thrown from the try statement  body."

Hence compiler is stating that, the catch block will never be reached as there is no such code in the try block which can ever make it to reach the catch block.

But alternatively

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

**int** a[][] = **new** **int**[3][];

*recursion*();

}

The above code will not give any compilation error as there is no unreachable code.

***Question: It is said that the code in a finally clause will never fail to execute, Is there any example where it fails to execute?***

***Answer*:** Here is the example code where the finally clause code will not execute.

|  |
| --- |
| public class testFinally{                      public static void main(String[] args){                                           System.out.println("Executing the program");                                             try {                                                               System.out.println("In the try block");                                                   System.exit(1);                          } finally {                                                               System.out.println("In the finally.");                                                 }                           }       } |

***Question: How do you intercept and thereby control exceptions***

***Answer*:**We can do this by using try/catch/finally blocks You place the normal processing code in try block  
You put the code to deal with exceptions that might arise in try block in catch block Code that must be executed no matter what happens must be place in finally block

**What is the difference between checked and Unchecked Exceptions in Java?**

All predefined exceptions in Java are either a checked exception or an unchecked exception. Checked exceptions must be caught using try .. catch() block or we should throw the exception using throws clause. If you don't, compilation of program will fail.

**Java Exception Hierarchy**

                    +--------+

                    | Object |

                    +--------+

                          |

                          |

                   +-----------+

                   | Throwable |

                   +-----------+

                    /         \

                   /           \

          +-------+          +-----------+

          | Error |          | Exception |

          +-------+          +-----------+

           /  |  \               / |       \

         \\_\_\_\_\_\_\_\_/           \\_\_\_\_\_\_/      \

                                       +------------------+

          unchecked            checked | RuntimeException |

                                       +------------------+

                                             /   |    |      \

                                            \\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

                                                 unchecked

**Question : Explain the rules of throws Exception in terms of Overriding?**

**Answer**: Below code can demonstrate that.

**Question : In terms of overriding, what will happen if child class is throwing exception but parent class method is not?**  
**Answer**: It's illegal so compiler will give error.  
  
**Question : In terms of overriding, what will happen if Parent class is throwing exception but child  class method is not?**  
**Answer**: It's legal and compilation will be successful.  
  
**Question : In terms of overriding, what will happen if Parent class method is throwing Runtime exception but child  class method is not?**  
**Answer**: It's legal and compilation will be successful.  
  
**Question : In terms of overriding, what will happen if Child class is throwing Runtime exception but Parent class method is not?**  
**Answer**: It's legal and compilation will be successful.

**JVM Related**

***Question :   When are static variables loaded in memory ?***

***Answer***. They are loaded at runtime when the respective Class is loaded.

***Question :   What is a String Pool ?***

***Answer***. String pool (String intern pool) is a special storage area in Java heap. When a string is created and if the string already exists in the pool, the reference of the existing string will be returned, instead of creating a new object and returning its reference.

***Question :  how many objects are created with this code ?***

***String s =new String("abc");***

***Answer***. Two objects will be created here. One object creates memory in heap with new operator and second in stack constant pool with "abc".

***Question :   Which are the different segments of memory ?***

***Answer***.

* Stack Segment - contains local variables and Reference variables(variables that hold the address of an object in the heap.
* Heap Segment - contains all created objects in runtime, objects only plus their object attributes (instance variables)
* Code Segment -  The segment where the actual compiled Java bytecodes resides when loaded.

***Question : .  Which memory segment loads the java code ?***

***Answer***. Code segment.

***Question : .  Does garbage collection guarantee that a program will not run out of memory?***

***Answer***. Garbage collection does not guarantee that a program will not run out of memory. It is possible for programs to use up memory resources faster than they are garbage collected. It is also possible for programs to create objects that are not subject to garbage collection

***Question : .  Describe what happens when an object is created in Java ?***

***Answer***.

* Memory is allocated from heap to hold all instance variables and implementation-specific data of the object and its superclasses. Implemenation-specific data includes pointers to class and method data.
* The instance variables of the objects are initialized to their default values.
* The constructor for the most derived class is invoked. The first thing a constructor does is call the constructor for its superclasses. This process continues until the constructor for java.lang.Object is called, as java.lang.Object is the base class for all objects in java.
* Before the body of the constructor is executed, all instance variable initializers and initialization blocks are executed. Then the body of the constructor is executed. Thus, the constructor for the base class completes first and constructor for the most derived class completes last.

**Quetion : .  Describe, in general, how java's garbage collector works ?**

Ans. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is known as garbage collection. The Java runtime environment supports a garbage collector that periodically frees the memory used by objects that are no longer needed. The Java garbage collector is a mark-sweep garbage collector that scans Java's dynamic memory areas for objects, marking those that are referenced. After all possible paths to objects are investigated, those objects that are not marked (i.e. are not referenced) are known to be garbage and are collected.

**Quetion : Can I import same package/class twice? Will the JVM load the package twice at runtime?**

Ans. One can import the same package or same class multiple times. Neither compiler nor JVM complains wil complain about it. And the JVM will internally load the class only once no matter how many times you import the same class.

**Quetion : Different types of memory used by JVM ?**

Ans. Class , Heap , Stack , Register , Native Method Stack.

**Quetion : .  What is a class loader ? What are the different class loaders used by JVM ?**

Ans. Part of JVM which is used to load classes and interfaces. Bootstrap , Extension and System are the class loaders used by JVM.

**Quetion : .  Explain java.lang.OutOfMemoryError ?**

Ans. This Error is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.

**Quetion : .  Is JVM, a compiler or interpretor ?**

Ans. Its an interpretor.

**Quetion : Difference between loadClass and Class.forName ?**

Ans. loadClass only loads the class but doesn't initialize i.e. it will not call any static block declared inside the class, whereas Class.forName execute all the static blocks and also initialize the static variable declared inside the class after loading it.

**Quetion : Should we override finalize method ?**

Ans. Finalize is used by Java for Garbage collection. It should not be done as we should leave the Garbage Collection to Java itself.

**Question : Which kind of memory is used for storing object member variables and function local variables ?**

Ans. Local variables are stored in stack whereas object variables are stored in heap.

**Question : Why do member variables have default values whereas local variables don't have any default value ?**

Ans. member variable are loaded into heap, so they are initialized with default values when an instance of a class is created. In case of local variables, they are stored in stack until they are being used.

***Question : Why Java don't use pointers ?***

***Answer :***. Pointers are vulnerable and slight carelessness in their use may result in memory problems and hence Java intrinsically manage their use.

***Question : What are various types of Class loaders used by JVM ?***

***Answer*** :

*Bootstrap*- Loads JDK internal classes, java.\* packages.

*Extensions*- Loads jar files from JDK extensions directory - usually lib/ext directory of the JRE

*System*- Loads classes from system classpath.

***Question : How are classes loaded by JVM ?***

***Answer***. Class loaders are hierarchical. The very first class is specially loaded with the help of static main() method declared in your class. All the subsequently loaded classes are loaded by the classes, which are already loaded and running.

***Question : Difference between static vs. dynamic class loading?***

***Answer :***

*static loading* - Classes are statically loaded with Java’s “new” operator.

*dynamic class loading* - Dynamic loading is a technique for programmatically invoking the functions of a class loader at run time.

Class.forName (Test className);

***Question : What are strong, soft, weak and phantom references in Java ?***

***Answer***. Garbage Collector won’t remove a strong reference. A soft reference will only get removed if memory is low. A weak reference will get removed on the next garbage collection cycle. A phantom reference will be finalized but the memory will not be reclaimed. Can be useful when you want to be notified that an object is about to be collected.

***Question : Name few tools for probing Java Memory Leaks ?***

***Answer :***. JProbe, OptimizeIt

**Question : .  Which memory areas does instance and static variables use ?**

Ans. instance variables are stored on stack whereas static variables are stored on heap.

***Question : What is PermGen or Permanent Generation ?***

***Answer***. The memory pool containing all the reflective data of the java virtual machine itself, such as class and method objects. With Java VMs that use class data sharing, this generation is divided into read-only and read-write areas. The Permanent generation contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

***Question : .  What is metaspace ?***

***Answer***. The Permanent Generation (PermGen) space has completely been removed and is kind of replaced by a new space called Metaspace. The consequences of the PermGen removal is that obviously the PermSize and MaxPermSize JVM arguments are ignored and you will never get a java.lang.OutOfMemoryError: PermGen error.

***Question : .  What are the disadvantages of using arrays ?***

***Answer***. Arrays are of fixed size and have to reserve memory prior to use. Hence if we don't know size in advance arrays are not recommended to use.

Arrays can store only homogeneous elements.

Arrays store its values in contentious memory location. Not suitable if the content is too large and needs to be distributed in memory.

There is no underlying data structure for arrays and no ready made method support for arrays, for every requirement we need to code explicitly

***Question : .  Can we call the garbage collector explicitly ?***

***Answer***. Yes, We can call garbage collector of JVM to delete any unused variables and unreferenced objects from memory using gc( ) method. This gc( ) method appears in both Runtime and System classes of java.lang package.

***Question : .  What are different ways to create String Object? Explain.***

***Answer*** :

String str = new String("abc");

String str1 = "abc";

When we create a String using double quotes, JVM looks in the String pool to find if any other String is stored with same value. If found, it just returns the reference to that String object else it creates a new String object with given value and stores it in the String pool.

When we use new operator, JVM creates the String object but don’t store it into the String Pool. We can use intern() method to store the String object into String pool or return the reference if there is already a String with equal value present in the pool.

***Question : How substring() method of String class create memory leaks?***

***Answer***. substring method would build a new String object keeping a reference to the whole char array, to avoid copying it. Hence you can inadvertently keep a reference to a very big character array with just a one character string.

***Question : How Java provide high Performance ?***

***Answer***. Java uses Just-In-Time compiler to enable high performance. Just-In-Time compiler is a program that turns Java bytecode into instructions that can be sent directly to the processor.

***Question : Why is Java considered Portable Language ?***

***Answer*** :  Java is a portable-language because without any modification we can use Java byte-code in any platform(which supports Java). So this byte-code is portable and we can use in any other major platforms.

***Question : .  How to find if JVM is 32 or 64 bit from Java program. ?***

***Answer***: You can find JVM - 32 bit or 64 bit by using System.getProperty() from Java program.

**Inner Classes**

***Question : What modifiers may be used with an inner class that is a member of an outer class?***  
***Answer***: All kind of public/protected/private/static/abstract and final, only thing is final and abstract is not possible.  
  
***Question : Give the example of the static and non static inner class.***  
***Answer***: Below is the example of non static inner class.

|  |
| --- |
| **public** **class** OuterClass {  **public** **static** **void** main(String args[]) { //Step 1: Create the OuterClass Object first                  OuterClass outerClass = **new** OuterClass(); //Step 2: Creating the object of Inner class using the outerClass.**new**operator                  InnerClass innerClass = outerClass.**new** InnerClass();           }  **class** InnerClass {             }  } |

Example of static inner class

|  |
| --- |
| **public** **class** OuterClass {  **public** **static** **void** main(String args[]) {                  //Example of static inner class, and creating the object using the                  //Outer Class itself.                  InnerClass innerClass = **new** OuterClass.InnerClass();           }  **static** **class** InnerClass {             }  } |

***Question : Can a static nested class have access to the enclosing class static methods or static variables?***  
***Answer***:  Yes. Look at the below code example.

|  |
| --- |
| **public** **class** OuterClass {  **static** **int** *i* = 1;  **static** **void** outerClassMethod() {                System.***out***.println(2);         }  **public** **static** **void** main(String args[]) {                // Example of static inner class, and creating the object using the                // Outer Class itself.                InnerClass innerClass = **new** OuterClass.InnerClass();                innerClass.innerClassMethod();         }  **static** **class** InnerClass {  **void** innerClassMethod() {  *outerClassMethod*();                       System.***out***.println(*i*);                }         }  } |

***Question : Can a static nested class have access to the enclosing class non-static or instance methods or instance variables?***  
***Answer***:  No. Look at the below code example.

|  |
| --- |
| **public** **class** OuterClass {  **int** i = 1;  **void** outerClassMethod() {                System.***out***.println(2);         }  **public** **static** **void** main(String args[]) {                // Example of static inner class, and creating the object using the                // Outer Class itself.                InnerClass innerClass = **new** OuterClass.InnerClass();                innerClass.innerClassMethod();         }  **static** **class** InnerClass {  **void** innerClassMethod() {                   //Compilation error : Cannot make a static reference to the                   //non-static method outerClassMethod() from the type OuterClass                   outerClassMethod();                   //Compilation error : Cannot make a static reference to                   //the non-static field i                   System.***out***.println(i);                }         }  } |

***Question : Can a non-static nested class have access to the enclosing class static methods or static variables?***  
***Answer***:  Yes. Look at the below code example.

|  |
| --- |
| **public** **class** OuterClass {  **static** **int** *i* = 1;  **static** **void** outerClassMethod() {                System.***out***.println(2);         }  **public** **static** **void** main(String args[]) {                InnerClass innerClass = **new** OuterClass().**new** InnerClass();                innerClass.innerClassMethod();         }  **class** InnerClass {  **void** innerClassMethod() {                       //static method call : possible  *outerClassMethod*();                       //static variable call : possible                       System.***out***.println(*i*);                }         }  } |

***Question : Can a non-static nested class have access to the enclosing class's non-static or instance methods or instance variables?***  
***Answer***:  Yes. Look at the below code example.

|  |
| --- |
| **public** **class** OuterClass {  **int** *i* = 1;  **void** outerClassMethod() {                System.***out***.println(2);         }  **public** **static** **void** main(String args[]) {                InnerClass innerClass = **new** OuterClass().**new** InnerClass();                innerClass.innerClassMethod();         }  **class** InnerClass {  **void** innerClassMethod() {                       //non static method call : possible  *outerClassMethod*();                       //non static variable call : possible                       System.***out***.println(*i*);                }         }  } |

***Question : Can a non-static nested class have access to the enclosing class non-static methods or instance variables with the scope other than public like private or protected or default?***  
***Answer :***Yes, there will be no problem accessing them in the inner classes, look at the below code example.

|  |
| --- |
| **public** **class** OuterClass {  **private** **int** i = 1;  **protected** **void** outerClassMethod() {                System.***out***.println(2);         }  **public** **static** **void** main(String args[]) {                InnerClass innerClass = **new** OuterClass().**new** InnerClass();                innerClass.innerClassMethod();         }  **public** **class** InnerClass {  **void** innerClassMethod() {                       //protected method call : possible                       outerClassMethod();                       //private variable call : possible                       System.***out***.println(i);                }         }  } |

***Question : What will happen if the non static inner class has default/friendly/package scope?***  
***Answer***:  Everything is fine as long as we are trying to create an object of Outer class as the Outer Class is public class so can be accessed from inside the same package and also from other packages.  
Now let's talk in terms of the object of inner class, as long as we are in the same package or class we will able to create the instance like

|  |
| --- |
| **public** **class** OuterClass {  **private** **int** i = 1;  **protected** **void** outerClassMethod() {                System.***out***.println(2);         }  **public** **static** **void** main(String args[]) {                InnerClass innerClass\_1 = **new** OuterClass().**new** InnerClass();                innerClass\_1.innerClassMethod();                //OR                OuterClass outerClass = **new** OuterClass();                InnerClass innerClass = outerClass.**new** InnerClass();                innerClass.innerClassMethod();         }  **class** InnerClass {  **public** **void** innerClassMethod() {                       //static method call : possible                       outerClassMethod();                       //static variable call : possible                       System.***out***.println(i);                }         }  } |

But when we try to access from another package we may have problem

|  |
| --- |
| **public** **class** Demo **extends** OuterClass {  **public** **static** **void** main(String[] args) {                // Creating class is usual thing and has no issue                OuterClass outerClass = **new** OuterClass();                // compilation error : if we try to access the inner class.                // - The type InnerClass is not visible                // - The type OuterClass.InnerClass is                // not visible                InnerClass innerClass = outerClass.**new** InnerClass();         }  } |

***Question : What will happen if the non static inner class has protected scope?***  
***Answer***:  We will able to create the object of Outer or Enclosing class but we may not able to create the instance of inner class or nested class inside the classes belongs to diff packages. We will get the following compilation error, while we try to create the object of inner class like "The constructor OuterClass.InnerClass() is not visible" even after extending the enclosing class the as the inheritance will not able to pull down the constructor.  
  
***Question : What will happen if the static inner class has protected scope?***  
***Answer***: The constructor OuterClass.InnerClass() is not visible.  
look at the code example to demo the same.

|  |
| --- |
| **public** **class** OuterClass {  **static** **private** **int** *i* = 1;  **static** **protected** **void** outerClassMethod() {                System.***out***.println(2);         }  **public** **static** **void** main(String args[]) {                InnerClass innerClass = **new** OuterClass.InnerClass();                innerClass.innerClassMethod();         }  **protected** **static**  **class** InnerClass {  **public** **void** innerClassMethod() {                       //static method call : possible  *outerClassMethod*();                       //static variable call : possible                       System.***out***.println(*i*);                }         }  } |

creating the demo class for verifying the same

|  |
| --- |
| **import** core.java.innerclass.pkg1.OuterClass;  **import** core.java.innerclass.pkg1.OuterClass.InnerClass;  **public** **class** Demo **extends** OuterClass {  **public** **static** **void** main(String[] args) {                 //Compilation fails                 //Compilation error : The constructor OuterClass.InnerClass() is not visible                 InnerClass innerClass = **new** OuterClass.InnerClass();         }  } |

***Question : Can the inner class access the variable and methods of the super class extended by the Outer Class?***  
***Answer*** : public and protected methods and variable can be accessed from the non static inner class. In case of the static inner class it can access only static variable from outer class and super class of the outer class.  
  
***Question : What is the Scenario when you really want to use inner class?***  
***Answer*** : <https://docs.oracle.com/javase/tutorial/java/javaOO/whentouse.html>

**Threads and Synchronization**

***Question : Can we declare a class as synchronized?***  
***Answer***: No.  
  
***Question : Can we declare a variable as synchronized?***  
***Answer***: No.

***Question : Can we declare a method as synchronized?***

***Answer*:**Yes

***Question : What is lifecycle of threads in Java.***

***Answer*:**

* **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
* **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
* **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
* **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires  or when the event it is waiting for occurs.
* **Terminated:**A runnable thread enters the terminated state when it completes its task or otherwise terminates.

***Question : Can we declare the run() like public void  run() throws InterruptedException ?***  
***Answer***: No, we cannot as run() in Thread Class and Runnable Interface doesn't throws  **InterruptedException,**So while overriding if parent class/Interface method is not throwing any exception then the child class method can only throw Runtime Exception or No Exception.

***What is the difference between preemptive scheduling and time slicing?***  
**Answer**: Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.  
  
**What is difference between wait() and sleep() method?**

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| 1) The wait() method is defined in Object class. | The sleep() method is defined in Thread class. |
| 2) wait() method releases the lock. | The sleep() method doesn't releases the lock. |

|  |
| --- |
| **public** **static** **void** main(String[] args) {                Thread t1 = **new** Thread();                t1.start();                t1.setDaemon(**true**);  } |

***Can we start a thread twice?***  
***Answer***: No. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.  
  
***What is Daemon Thread in Java?***  
Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.  
There are many java daemon threads running automatically e.g. gc, finalizer etc.  
You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.  
  
***What are the methods for Java Daemon thread by Thread class?***  
***Answer***: The java.lang.Thread class provides two methods for java daemon thread.  
Method Description  
*public void setDaemon(boolean status)* is used to mark the current thread as daemon thread or user thread.  
*public boolean isDaemon()* is used to check that current is daemon.

***Can we make an already running thread as demon thread?***

***Answer :***No, we can not , we need to make it demon before calling the start(), otherwise it will throw an Illegal state exception.

|  |
| --- |
| **public** **static** **void** main(String[] args) {                Thread t1 = **new** Thread();                t1.start();                t1.setDaemon(**true**);  } |

|  |
| --- |
| Exception in thread "main" java.lang.IllegalThreadStateException         at java.lang.Thread.setDaemon(Unknown Source)         at core.java.threads.UserDemonThread.main(UserDemonThread.java:8) |

So correct way is the following code

|  |
| --- |
| **public** **static** **void** main(String[] args) {         Thread t1 = **new** Thread();         t1.setDaemon(**true**);         t1.start();  } |

***What is Shutdown Hook?***  
***Answer***: The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.  
  
***Show code example of the Shutdown Hook.***  
***Answer***: Following the is the code to show the shutdown hook.  
First creating the thread MyThread which will be attached to the shutdown hook, so that upon the shutdown, the thread's run method will be called.

|  |
| --- |
| **public** **class** MyThread **implements** Runnable {  **private** **int** i;  **public** MyThread(**int** i){  **this**.i =i;         }         @Override  **public** **void** run() {                System.***out***.println("I am Running : " + i);         }  } |

Now creating the main method to demo the same.

|  |
| --- |
| **public** **static** **void** main(String[] args) {                //Creating the basic thread                Thread t  = **new** Thread(**new** MyThread(1));                //Getting the runtime object from the Runtime factory class                Runtime r = Runtime.*getRuntime*();                //Attaching the recently created thread to the Runtime for the shutdown hook,                //so that this will be called upon the JVM Shutdown.                r.addShutdownHook(t);                System.***out***.println("Starting the task");  **try** {                       Thread.*sleep*(5000);                } **catch** (InterruptedException e) {                       e.printStackTrace();                }                System.***out***.println("Completed the task");                //Once the task get completed the JVM is going to shut down,                //This the time when runnable thread will start and execute the                //run method.                //r.halt(0);         } |

***What will happen if we uncomment last line from the above code i.e. we call r.halt(0); after the comments ended in the above code?***

***Answer***: In that case the shutdown hook will not be called.

***What is Interrupting a Thread?***  
***Answer:***If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.  
  
***Question : What is ThreadLocal in java?***  
***Answer :***Thread Local can be considered as a scope of access, like a request scope or session scope. It’s a thread scope. You can set any object in Thread Local and this object will be global and local to the specific thread which is accessing this object. Global and local!!? Let me explain:  
  
*Values stored* in Thread Local are global to the thread, meaning that they can be accessed from anywhere inside that thread. If a thread calls methods from several classes, then all the methods can see the Thread Local variable set by other methods (because they are executing in same thread). The value need not be passed explicitly. It’s like how you use global variables.  
  
*Values stored* in Thread Local are local to the thread, meaning that each thread will have it’s own Thread Local variable. One thread can not access/modify other thread’s Thread Local variables.  
  
***Question : What is Volatile keyword in java?***  
***Answer*** : The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.  
Actually, since Java 5 the volatile keyword guarantees more than just that volatile variables are written to and read from main memory. I will explain that in the following sections.  
The Java volatile keyword guarantees visibility of changes to variables across threads. This may sound a bit abstract, so let me elaborate.  
  
In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs.

What is Executor Service  
What is Countdownlatch  
What is Cyclic Barrier  
When to use CyclicBarrier in Java?  
What is race condition in multithreading.  
Difference between Runnable and Thread in java  
How to Stop Thread in Java  
Why wait and notify methods are declared in Object Class?  
How to Solve Producer Consumer Problem in Java  
Why wait and notify needs to called from Synchronized Context?  
Difference between ConcurrentHashMap and Hashtable in Java  
Difference between invokeAndWait and InvokeLater in java Swing.  
  
Difference between wait and sleep in Java  
When to use CyclicBarrier in Java?  
What is race condition in multithreading.  
Difference between Runnable and Thread in java  
How to Stop Thread in Java  
Why wait and notify methods are declared in Object Class?  
How to Solve Producer Consumer Problem in Java  
Why wait and notify needs to called from Synchronized Context?  
Difference between ConcurrentHashMap and Hashtable in Java  
Difference between invokeAndWait and InvokeLater in java Swing.  
Difference between wait and sleep in Java  
When to use ThreadLocal variable in Java  
Difference between Runnable and Thread in Java  
How to stop Thread in Java  
How to fix deadlock in Java  
How to find race conditions in Java  
How to write Thread-safe code in Java  
  
http://javarevisited.blogspot.com/2012/07/countdownlatch-example-in-java.html#ixzz3k6Zg7w8Q  
http://javarevisited.blogspot.in/2015/07/how-to-do-group-by-in-java-8.html

**Programing Riddle**

**Question: Titanic- Find the First non repeating character.**  
**Question: Find last 5 characters of a String**  
**Question:Can we create outofmemory Error?**  
**Question : Fibonacci Series with recursion.**  
**Question: Can a for statement loop indefinitely?**  
**Question : Write the program to find the list of prime number from 0 to N.**  
**Approach : 1**

|  |
| --- |
| **package** core;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** PrimeNumberDemo {  **public** **static** **void** main(String[] args) {                PrimeNumberContainer container = **new** PrimeNumberContainer(200);                container.showPrimeNumberFromList();                System.***out***.println("Total Iteration = "+container.totalCount);         }  }  **class** PrimeNumberContainer{  **public** **int** totalCount=1;         List<Integer> primeNumbers = **null**;    **public** PrimeNumberContainer(**int** n){                generateNNumbersOfPrimenumbers(n);         }    **public** **void** showPrimeNumberFromList(){  **for**(Integer primeNumber : primeNumbers){                       System.***out***.println(primeNumber);                }         }    **private** **void** generateNNumbersOfPrimenumbers(**int** numberOfPrimeToBeGenerated){                primeNumbers = **new** ArrayList<>(numberOfPrimeToBeGenerated);  **int** initialValue = 3;  **while**(primeNumbers.size() <= numberOfPrimeToBeGenerated ){                       isPrime(initialValue);                       initialValue+=2;                }         }    **public** **boolean** isPrime(**int** numberToBeVerifiedIfItIsPrime){    **if**(numberToBeVerifiedIfItIsPrime % 2 == 0 ){                       totalCount++;  **return** **false**;                }  **for**(**int** i=3 ; i <= Math.*sqrt*(numberToBeVerifiedIfItIsPrime) ; i+=2)                {  **if**((numberToBeVerifiedIfItIsPrime % i) == 0){  **return** **false**;                       }                       totalCount++;                }                primeNumbers.add(numberToBeVerifiedIfItIsPrime);  **return** **true**;         }  } |

**Using the sieve of Eratosthenes**

|  |
| --- |
| **package** core;  **import** java.util.ArrayList;  **import** java.util.List;  **import** java.util.concurrent.CopyOnWriteArrayList;  **public** **class** PrimeNumberDemoUsingCopyOnWriteArrayList {  **public** **static** **void** main(String[] args) {                PrimeNumberContainerWithCopyOnWriteArrayListnumberContainerWithCopyOnWriteArrayList =  **new** PrimeNumberContainerWithCopyOnWriteArrayList(200);                System.***out***.println(numberContainerWithCopyOnWriteArrayList.removeNonPrimeFromList());                System.***out***.println(numberContainerWithCopyOnWriteArrayList.primeNumbers);         }  }  **class** PrimeNumberContainerWithCopyOnWriteArrayList{         List<Integer> primeNumbers = **null**;  **public** PrimeNumberContainerWithCopyOnWriteArrayList(**int** n){                addAllNumbersBetween2ToN(n);         }    **public** **void** addAllNumbersBetween2ToN(**int** n){                primeNumbers = **new** CopyOnWriteArrayList<Integer>();  **for**(**int** i =2 ; i <= n ; i++){                       primeNumbers.add(i);                }         }    **public** **int** removeNonPrimeFromList(){  **int** totalIteration = 0;  **int** j=0;  **for**( j=0; j < primeNumbers.size() ; j++)  **for**(**int** i=j+1 ; i < primeNumbers.size() ; i++){  **if**(primeNumbers.get(i) % primeNumbers.get(j) == 0){                             primeNumbers.remove(i);                       }                       totalIteration++;                }  **return** totalIteration;           }  } |

**Answer:** Yes, a for statement can loop indefinitely. For example, consider the following: for(;;) ;  
<http://javaconceptoftheday.com/how-to-reverse-the-string-with-preserving-the-position-of-spaces/>  
<http://conceptlearningcentre.com/index.php/2015/08/03/string-rotation-program/>  
  
  
  
**JDBC**  
  
***Question : Write A sample Program for the JDBC.***  
***Answer***:  
*Step 1 : Load the driver*

|  |
| --- |
| **static** {  **try** {                Class.*forName*("oracle.jdbc.driver.OracleDriver");         } **catch** (ClassNotFoundException ex) {                ex.printStackTrace();         }  } |

*Step 2 : Get the connection*

|  |
| --- |
| DriverManager.*getConnection*("jdbc:oracle:thin:@localhost:1521:xe", "HR", "password"); |

**Step 3: Run some SQL Command**

|  |
| --- |
| @Override  **public** List<Contact> findAll() {         List<Contact> result = **new** ArrayList<Contact>();         Connection connection = **null**;    **try** {                connection = getConnection();                PreparedStatement statement = connection.prepareStatement("select \* from contact");                ResultSet resultSet = statement.executeQuery();  **while** (resultSet.next()) {                       Contact contact = **new** Contact();                       contact.setId(resultSet.getLong("id"));                       contact.setFirstName(resultSet.getString("first\_name"));                       contact.setLastName(resultSet.getString("last\_name"));                       contact.setBirthDate(resultSet.getDate("birth\_date"));                       result.add(contact);                }         } **catch** (SQLException ex) {                ex.printStackTrace();         } **finally** {                closeConnection(connection);         }  **return** result;  } |

***Why PreparedStatement are faster than statement?***  
***Answer*:**The Prepared Statement is a slightly more powerful version of a Statement, and should always be at least as quick and easy to handle as a Statement.  
The Prepared Statement may be parametrized  
  
Most relational databases handles a JDBC / SQL query in four steps:  
1.Parse the incoming SQL query  
2. Compile the SQL query  
3. Plan/optimize the data acquisition path  
4. Execute the optimized query / acquire and return data  
  
A Statement will always proceed through the four steps above for each SQL query sent to the database. A Prepared Statement pre-executes steps (1) - (3) in the execution process above. Thus, when creating a Prepared Statement some pre-optimization is performed immediately. The effect is to lessen the load on the database engine at execution time.  
  
Precompilation and DB-side caching of the SQL statement leads to overall faster execution and the ability to reuse the same SQL statement in batches.  
  
Automatic prevention of SQL injection attacks by builtin escaping of quotes and other special characters. Note that this requires that you use any of the PreparedStatement setXxx() methods to set the values and thus don't inline the values in the SQL string by string-concatenating.  
They are pre-compiled (once), so faster for repeated execution of dynamic SQL (where parameters change).  
  
Database statement caching boosts DB execution performance.  
  
Databases store caches of execution plans for previously executed statements. This allows the database engine to reuse the plans for statements that have been executed previously. Because PreparedStatement uses parameters, each time it is executed it appears as the same SQL, the database can reuse the previous access plan, reducing processing. Statements "inline" the parameters into the SQL string and so do not appear as the same SQL to the DB, preventing cache usage.  
  
Binary communications protocol means less bandwidth and faster comms calls to DB server.  
  
Prepared statements are normally executed through a non-SQL binary protocol. This means that there is less data in the packets, so communications to the server is faster. As a rule of thumb network operations are an order of magnitude faster than disk operations which are an order of magnitude faster than in-memory CPU oeprations. Hence, any reduction in amount of data sent over the network will have a good effect on overall performance.  
  
They protect against SQL injection, by escaping text for all the parameter values provided.  
  
They provide stronger separation between the query code and the parameter values (compared to concatenated SQL strings), boosting readability and helping code maintainers quickly understand inputs and outputs of the query.  
  
In java, can call getMetadata() and getParameterMetadata() to reflect on the result set fields and the parameter fields, respectively  
  
In java, intelligently accepts java objects as parameter types via setObject, setBoolean, setByte, setDate, setDouble, setDouble, setFloat, setInt, setLong, setShort, setTime, setTimestamp - it converts into JDBC type format that is comprehendible to DB (not just toString() format).  
  
In java, accepts SQL ARRAYs, as parameter type via setArray method  
  
In java, accepts CLOBs, BLOBs, OutputStreams and Readers as parameter "feeds" via setClob/setNClob, setBlob, setBinaryStream, setCharacterStream/setAsciiStream/setNCharacterStream methods, respectively  
  
In java, allows DB-specific values to be set for SQL DATALINK, SQL ROWID, SQL XML, and NULL via setURL, setRowId, setSQLXML ans setNull methods  
  
In java, inherits all methods from Statement. It inherits the addBatch method, and additionally allows a set of parameter values to be added to match the set of batched SQL commands via addBatch method.  
  
In java, a special type of PreparedStatement (the subclass CallableStatement) allows stored procedures to be executed - supporting high performance, encapsulation, procedural programming and SQL, DB administration/maintenance/tweaking of logic, and use of proprietary DB logic & features  
 ***Question :  How to call the Stored Procedure using JDBC?***  
***Answer* :**Code sample to call the stored procedure.

|  |
| --- |
| /\*\*   \* **@author** Vikash   \*   \*/  **package** com.sql.sp.jdbc;  **import** java.sql.CallableStatement;  **import** java.sql.Connection;  **import** java.sql.Date;  **import** java.sql.DriverManager;  **public** **class** JDBCDemoForStoredProcedure{  **public** **static** **void** main(String[] args) **throws** Exception{    Connection connection = DriverManager.*getConnection*("jdbc:oracle:thin:@localhost:1521:xe","HR","password");    CallableStatement callableStatement = connection.prepareCall("call contactInsert(?,?,?)");    callableStatement.setString(1, "King");    callableStatement.setString(2, "George");    callableStatement.setDate(3, **new** Date(0));    callableStatement.execute();    connection.close();  }  } |

***Question :  How to call the SQL Function using JDBC?***  
***Answer* :**Code sample to call the SQL Function, we are calling very common in built function of oracle called sysdate().

|  |
| --- |
| /\*\*   \* **@author** Vikash   \*   \*/  **package** com.sql.sp.jdbc;  **import** java.sql.CallableStatement;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.Types;  **public** **class** JDBCDemoForFunction{  **public** **static** **void** main(String[] args) **throws** Exception{    Connection connection=          DriverManager.*getConnection*("jdbc:oracle:thin:@localhost:1521:xe","HR","password");    CallableStatement callableStatement = connection.prepareCall("{? = call sysdate()}");    callableStatement.registerOutParameter(1, Types.***DATE***);    callableStatement.execute();    System.***out***.println(callableStatement.getDate(1));    connection.close();   }  } |

**ExecutorService Example**

Here is a simple Java ExectorService example:

ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

First an ExecutorService is created using the newFixedThreadPool() factory method. This creates a thread pool with 10 threads executing tasks.

Second, an anonymous implementation of the Runnable interface is passed to the execute() method. This causes the Runnable to be executed by one of the threads in the ExecutorService.

**Task Delegation**

Here is a diagram illustrating a thread delegating a task to an ExecutorService for asynchronous execution:

|  |
| --- |
| A thread delegating a task to an ExecutorService for asynchronous execution. |
| **A thread delegating a task to an ExecutorService for asynchronous execution.** |

Once the thread has delegated the task to the ExecutorService, the thread continues its own execution independent of the execution of that task.

**ExecutorService Implementations**

Since ExecutorService is an interface, you need to its implementations in order to make any use of it. TheExecutorService has the following implementation in the java.util.concurrent package:

* [**ThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/threadpoolexecutor.html)
* [**ScheduledThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html)

**Creating an ExecutorService**

How you create an ExecutorService depends on the implementation you use. However, you can use theExecutors factory class to create ExecutorService instances too. Here are a few examples of creating anExecutorService:

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

**ExecutorService Usage**

There are a few different ways to delegate tasks for execution to an ExecutorService:

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...)

I will take a look at each of these methods in the following sections.

**execute(Runnable)**

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. Here is an example of executing a Runnable with an ExecutorService:

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use aCallable for that (explained in the following sections).

**submit(Runnable)**

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Futureobject can be used to check if the Runnable as finished executing.

Here is a ExecutorService submit() example:

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

**submit(Callable)**

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.

The Callable's result can be obtained via the Future object returned by the submit(Callable) method. Here is an ExecutorService Callable example:

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

The above code example will output this:

Asynchronous Callable

future.get() = Callable Result

**invokeAny()**

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.

If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.

Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

This code example will print out the object returned by one of the Callable's in the given collection. I have tried running it a few times, and the result changes. Sometimes it is "Task 1", sometimes "Task 2" etc.

**invokeAll()**

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.

Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a Future to tell the difference.

Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

**ExecutorService Shutdown**

When you are done using the ExecutorService you should shut it down, so the threads do not keep running.

For instance, if your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.

To terminate the threads inside the ExecutorService you call its shutdown() method. The ExecutorService will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down. All tasks submitted to the ExecutorService before shutdown()is called, are executed.

If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.