2011

Java Interview Questions

Author – Sanjeev Singh

1. **What are the principle concepts of OOPS?**

   There are four principle concepts upon which object oriented design is based on

   - Abstraction
   - Polymorphism
   - Inheritance
   - Encapsulation

2. **What is Abstraction?**

   Abstraction refers to the act of representing essential features without including the background details or explanations. Let's say you have a method "Calculate Salary" in your Employee class, which takes Employee Id as parameter and returns the salary of the employee for the current month as an integer value. Now if someone wants to use that method. He does not need to care about how Employee object calculates the salary? An only thing he needs to be concern is name of the method, its input parameters and format of resulting member, Right? So abstraction says expose only the details which are concern with the user (client) of your object. So the client who is using your class need not to be aware of the inner details like how you class do the operations? He needs to know just few details.

3. **What is Encapsulation?**

   Encapsulation is a technique used for hiding the properties and behaviours of an object and allowing outside access only as appropriate. It prevents other objects from directly altering or accessing the properties or methods of the encapsulated object. One way to think about encapsulation is as a protective wrapper that prevents code and data from being arbitrarily accessed by other code defined outside the wrapper. Encapsulation defines the access levels for elements of that class. These access levels define the access rights to the data, allowing us to access the data by a method of that particular class itself, from an inheritance class, or even from any other class. There are three levels of access:

4. **What is the difference between abstraction and encapsulation?**

   Abstraction focuses on the outside view of an object (i.e. the interface **Encapsulation** (information hiding) prevents clients from seeing it's inside view, where the behavior of the abstraction is implemented.

   Abstraction solves the problem in the design side while **Encapsulation** is the Implementation.

5. What is Association?

Association is a relationship where all object have their own lifecycle and there is no owner. Let's take an example of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

Points:
- Is a Relationship between objects.
- Objects have independent lifecycles.
- There is no owner.
- Objects can create and delete independently.

6. What is Aggregation?

Aggregation is a specialize form of Association where all object have their own lifecycle but there is ownership and child object cannot belongs to another parent object at the same time.

Let's take an example of Department and teacher. A single teacher cannot belong to multiple departments, but if we delete the department, teacher object will not destroy. We can think about "has-a" relationship.

Points:
- Specialize form of Association.
- has-a relationship between objects
- Object have independent life-cycles
- Parent-Child relationship

7. What is Composition?

Composition is again specialize form of Aggregation. It is a strong type of Aggregation. Here the Parent and Child objects have coincident lifetimes. Child object does not have its own lifecycle and if parent object gets deleted, then all of its child objects will also be deleted.

Let's take again an example of relationship between House and rooms. House can contain multiple rooms there is no independent life of room and any room cannot belongs to two different house if we delete the house room will automatically delete. Let's take another example relationship between Questions and options. Single questions can have multiple options and option can not belong to multiple questions. If we delete questions options will automatically delete.

Points:
• Specialize form of Aggregation
• Strong Type of Aggregation.
• Parent-Child relationship
• only parent object has independent life-cycle.

8. What is Inheritance?

   Inheritance is the process by which objects of one class acquire the properties of objects of another class.

   - A class that is inherited is called a super class.
   - The class that does the inheriting is called a subclass.
   - Inheritance is done by using the keyword extends.
   - The two most common reasons to use inheritance are:
     - To promote code reuse
     - To use polymorphism

9. What is Polymorphism?

   Polymorphism is briefly described as "one interface, many implementations." Polymorphism is a characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form. Polymorphism manifests itself in Java in the form of multiple methods having the same name.

10. How does Java implement polymorphism?

    (Inheritance, Overloading and Overriding are used to achieve Polymorphism in java). Polymorphism manifests itself in Java in the form of multiple methods having the same name.

    - In some cases, multiple methods have the same name, but different formal argument lists (overloaded methods).
    - In other cases, multiple methods have the same name, same return type, and same formal argument list (overridden methods).

11. Explain the different forms of Polymorphism?

There are two types of polymorphism.

1. Compile time polymorphism – Using method overloading
2. Run time polymorphism – Using overriding

12. What is runtime polymorphism or dynamic method dispatch?

In Java, runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a super class. The determination of the method to be called is based on the object being referred to by the reference variable.

13. What is Dynamic Binding?

Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance.

14. What is method overloading?

Method Overloading means to have two or more methods with same name in the same class with different arguments. The benefit of method overloading is that it allows you to implement methods that support the same semantic operation but differ by argument number or type.
Note:

- *Overloaded methods MUST change the argument list*
- *Overloaded methods CAN change the return type*
- *Overloaded methods CAN change the access modifier*
- *Overloaded methods CAN declare new or broader checked exceptions*
- *A method can be overloaded in the same class or in a subclass*

15. What is method overriding?

Method overriding occurs when sub class declares a method that has the same type arguments as a method declared by one of its super class. The key benefit of overriding

is the ability to define behavior that's specific to a particular subclass type.
Note:

- *The overriding method cannot have a more restrictive access modifier than the method being overridden (Ex: You can't override a method marked public and make it protected).*
- *You cannot override a method marked final*
- *You cannot override a method marked static*

16. **Is it possible to override the main method?**

    No, because main is a static method. A static method can't be overridden in Java.

17. **What is Byte Code? Or what gives java it's "write once and run anywhere" nature?**

    All Java programs are compiled into class files that contain byte codes. These byte codes can be run in any platform and hence java is said to be platform independent.

18. **Expain the reason for each keyword of public static void main (String args[])?**

    Public- main (..) is the first method called by java environment when a program is executed so it has to accessible from java environment. Hence the access specifier has to be public.

    static: Java environment should be able to call this method without creating an instance of the class , so this method must be declared as static.

    void: main does not return anything so the return type must be void

    The argument String indicates the argument type which is given at the command line and arg is an array for string given during command line.

19. **What is difference between == and equals?**

    The == operator compares two objects to determine if they are the same object in memory i.e. present in the same memory location. It is possible for two String objects to have the same value, but located in different areas of memory.== compares references while .equals compares contents. The method public boolean equals(Object obj) is provided by the Object class and can be overridden. The default implementation returns true only if the object is compared with itself, which is equivalent to the equality operator == being used to compare aliases to the object.

```
public class EqualsTest {

	public static void main(String[] args) {

		String s1 = "abc";
		String s2 = s1;
		String s5 = "abc";
		String s3 = new String("abc");
		String s4 = new String("abc");
		System.out.println("== comparison : " + (s1 == s5));
		System.out.println("== comparison : " + (s1 == s2));
		System.out.println("Using equals method : " + s1.equals(s2));
		System.out.println("== comparison : " + s3 == s4);
		System.out.println("Using equals method : " + s3.equals(s4));
	}
}
```

Output
== comparison : true
== comparison : true
Using equals method : true
false
Using equals method : true

20. **What if the static modifier is removed from the signature of the main method? Or What if I do not provide the String array as the argument to the method?**

    Program compiles. But at runtime throws an error "NoSuchMethodError".

21. What does final keyword?

    Variables defined in an interface are implicitly final. A final class can't be extended i.e., final class may not be sub classed. This is done for security reasons with basic classes like String and Integer. It also allows the compiler to make some optimizations, and makes thread safety a little easier to achieve. A final method can't be overridden when its class is inherited. You can't change value of a final variable (is a constant).

22. **How do you prevent a method from being overridden?**

    To prevent a specific method from being overridden in a subclass, use the final modifier on the method declaration, which means "this is the final implementation of this method", the end of its inheritance hierarchy.

23. Can we instantiate an interface?

    We can't instantiate an interface directly, but you can instantiate a class that implements an interface.

24. Can we create an object for an interface?

Yes, it is always necessary to create an object implementation for an interface. Interfaces cannot be instantiated in their own right, so you must write a class that implements the interface and fulfil all the methods defined in it.

25. Do interfaces have member variables?

Interfaces may have member variables, but these are implicitly public, static, and final- in other words, interfaces can declare only constants, not instance variables that are available to all implementations and may be used as key references for method arguments as well as only public and abstract modifiers are allowed for methods in interfaces.

26. **What is a marker interface**?

Marker interfaces are those which do not declare any required methods, but signify their compatibility with certain operations. The java.io.Serializable interface and Cloneable are typical marker interfaces. These do not contain any methods, but classes must implement this interface in order to be serialized and de-serialized.

27. What is an abstract class?

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. **Note**:

- *If even a single method is abstract, the whole class must be declared abstract.*
- *Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods.*
- *You can't mark a class as both abstract and final.*

28. What are the differences between Interface and Abstract class?

Abstract Class	Interfaces
An abstract class can provide complete, default code and/or just the details that have to be overridden.	An interface cannot provide any code at all, just the signature.
In case of abstract class, a class may extend only one abstract class.	A Class may implement several interfaces.
An abstract class can have non-abstract methods.	All methods of an Interface are abstract.

An abstract class can have instance variables.	An Interface cannot have instance variables.
An abstract class can have any visibility: public, private, protected.	An Interface visibility must be public (or) none.
If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly.	If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method.
An abstract class can contain constructors .	An Interface cannot contain constructors .
Abstract classes are fast.	Interfaces are slow as it requires extra indirection to find corresponding method in the actual class.

29. **When should I use abstract classes and when should I use interfaces?**

Use Interfaces when…

- You see that something in your design will change frequently.
- If various implementations only share method signatures then it is better to use Interfaces.
- You need some classes to use some methods which you don't want to be included in the class, then you go for the interface, which makes it easy to just implement and make use of the methods defined in the interface.

Use Abstract Class when…

- If various implementations are of the same kind and use common behavior or status then abstract class is better to use.
- When you want to provide a generalized form of abstraction and leave the implementation task with the inheriting subclass.
- Abstract classes are an excellent way to create planned inheritance hierarchies. They're also a good choice for nonleaf classes in class hierarchies.

30. What are the differences between Class Methods and Instance Methods?

Class Methods	Instance Methods
Class methods are methods which are declared as static. The method can be called without creating an instance of the class	Instance methods on the other hand require an instance of the class to exist before they can be called, so an instance of a class needs to be created by using the new keyword. Instance methods operate on specific instances of classes.
Class methods can only operate on class members and not on instance members as class methods are unaware of instance members.	Instance methods of the class can also not be called from within a class method unless they are being called on an instance of that class.
Class methods are methods which are declared as static. The method can be called without creating an instance of the class.	Instance methods are not declared as static.

31. **What are Access Specifier available in Java**?

Java offers four access specifier, listed below in decreasing accessibility:

Public- *public* classes, methods, and fields can be accessed from everywhere.

Protected- *protected* methods and fields can only be accessed within the same class to which the methods and fields belong, within its subclasses, and within classes of the same package.

Default(no specifier)- If you do not set access to specific level, then such a class, method, or field will be accessible from inside the same package to which the class, method, or field belongs, but not from outside this package.

Private- *Private* Methods and fields can only be accessed within the same class to which the methods and fields belong. *Private* methods and fields are not visible within subclasses and are not inherited by subclasses.

32. **What is static block?**

Static block which exactly executed exactly once when the class is first loaded into JVM and it is executed before the constructor is invoked.

33. **What are static variables?**

Variables that have only one copy per class are known as static variables. They are not attached to a particular instance of a class but rather belong to a class as a whole. They are declared by using the static keyword as a modifier.

34. **What are static methods?**

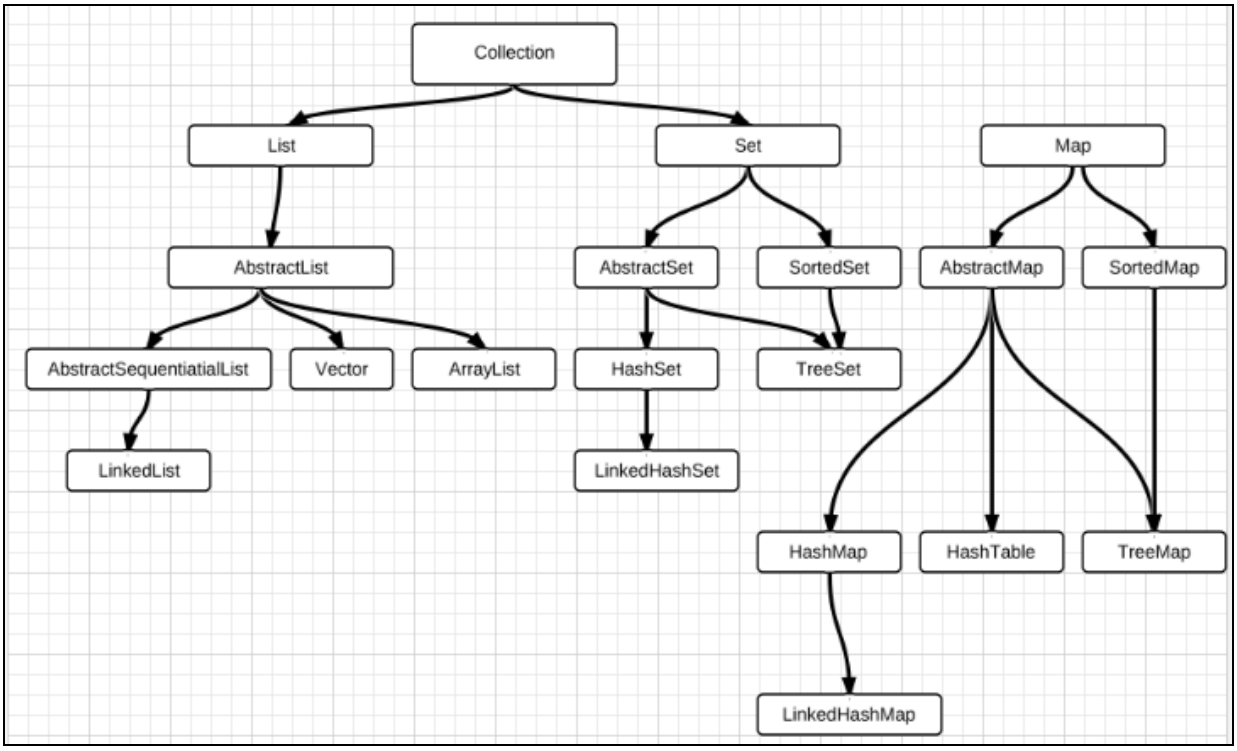
Methods declared with the keyword static as modifier are called static methods or class methods. They are so called because they affect a class as a whole, not a particular instance of the class. Static methods are always invoked without reference to a particular instance of a class.

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

- *A static method can only call other static methods.*
- *A static method must only access static data.*
- *A static method **cannot** reference to the current object using keywords super or this.*

35. **What is an Iterator?**

- The Iterator interface is used to step through the elements of a Collection.
- Iterators let you process each element of a Collection.
- Iterators are a generic way to go through all the elements of a Collection no matter how it is organized.
- Iterator is an Interface implemented a different way for every Collection.



36. What is List

This is an ordered collection (also known as a sequence). The List interface extends the Collection interface to define an ordered collection, permitting duplicates. The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

37. What is ArrayList

It is resizable-array implementation of the List interface. If we need to support random access, without inserting or removing elements from any place to other than the end, then ArrayList offers you the optimal collection.

Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added an ArrayList, its capacity grows automatically.

An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.

This implementation is not synchronized**.** If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized.
List list = Collections.synchronizedList (new ArrayList (...));

The iterators returned by this class's iterator and ListIterator methods are ***fail-fast***: if list is structurally modified at any time after the iterator is created, in any way except through the iterators own remove or add methods, the iterator will throw a ConcurrentModificationException.

38. What is LinkedList

It is Doubly Linked list implementation of the List interface. It may provide better performance than the ArrayList implementation if elements are frequently inserted or deleted within the list.

This implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized.

The iterators returned by this class's iterator and ListIterator methods are ***fail-fast***: if list is structurally modified at any time after the iterator is created, in any way except through the iterators own remove or add methods, the iterator will throw a ConcurrentModificationException.

39. What is difference between ArrayList & LinkedList

1. Adding and deleting at the start and middle of the ArrayList is slow, because all the later elements have to be copied forward or backward. (Using System.arrayCopy()) Whereas Linked lists are faster for inserts and deletes anywhere in the list, since all we do is update a few next and previous pointers of a node.

2. Each element of a linked list (especially a doubly linked list) uses a bit more memory than its equivalent in array list, due to the need for next and previous pointers.

3. ArrayList may also have a performance issue when the internal array fills up. The ArrayList has to create a new array and copy all the elements there. The ArrayList has a growth algorithm of (n*3)/2+1, meaning that each time the buffer is too small it will create a new one of size (n*3)/2+1 where n is the number of elements of the current buffer. Hence if we can guess the number of elements that we are going to have, then it makes sense to create a

ArrayList with that capacity during object creation (using constructor new ArrayList(capacity)). Whereas LinkedList should not have such capacity issues.

40. What is Vector

The Vector class implements a growable array of objects. Each vector tries to optimize storage management by maintaining a capacity and a capacityIncrement. The capacity is always at least as large as the vector size. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation. This implementation is synchronized.

41. What is difference between ArrayList & Vector

1. Vector & ArrayList both classes are implemented using dynamically resizable arrays, providing fast random access and fast traversal. ArrayList and Vector class both implement the List interface.
2. Vector is synchronized whereas ArrayList is not. Even though Vector class is synchronized, still when you want programs to run in multithreading environment using ArrayList with Collections.synchronizedList () is recommended over Vector.
3. ArrayList has no default size while vector has a default size of 10.
4. The Enumerations returned by Vector's elements method are not fail-fast. Whereas ArrayList does not have any method returning Enumerations.

42. What is Set

It is a collection that contains no duplicate elements. This also means that a set can contain at most one null value.

Set has 3 general purpose implementations.

- HashSet
- TreeSet
- LinkedHashSet

43. What is HashSet

The HashSet class implements the Set interface and it uses a hash table data structure for its implementation. This class permits the null element. A HashSet does not guarantee any ordering of the elements. It does not guarantee that the order will remain constant over time.

This implementation is not synchronized. If multiple threads access an HashSet instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized.
Set s = Collections.synchronizedSet (new HashSet (...));

The HashSet uses a hash code to place items in their location. When an object is inserted, it finds the appropriate bucket corresponding to the objects by using getHashCode () method on the key to determine where it should be put in its internal buckets. After that it calls the equals method to compare the equality of objects in the bucket. If an equals returns true with any object of the bucket then the new object is not inserted.

44. What is LinkedHashSet

A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements. Use this class instead of HashSet when you care about the iteration order. When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

This implementation is not synchronized**.** If multiple threads access a LinkedHashSet instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized.
Set s = Collections.synchronizedSet (new LinkedHashSet (...));

45. What is SortedSet

A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the *natural ordering* of its elements.

All elements inserted into in sorted set must implement the Comparable interface (or be accepted by the specified Comparator). Furthermore, all such elements must be mutually comparable: e1.compareTo (e2) (or comparator. compare (e1, e2)) must not throw a ClassCastException for any elements e1 and e2 in the sorted set. Attempts to

violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

46. What is TreeSet

TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order, sorted according to the *natural order* of the elements. Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.
This implementation is not synchronized. If multiple threads access a LinkedHashSet instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized.
SortedSet s = Collections.synchronizedSortedSet (new TreeSet(...));

47. What is Map?

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements.

48. What is HashMap?

It is hash table based implementation of Map interface. *It allows null key and null values*.

The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls. This class makes *no guarantees as to the order of the map*; in particular, it does not guarantee that the order will remain constant over time.

An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method.

Note that this implementation is not synchronized. If multiple threads access this map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally.
Map m = Collections.synchronizedMap (new HashMap(...));

49. What is HashTable?

This class used a hash table data structure to store the data and maps keys to values. Any non-null object can be used as a key or as a value. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hash table, the objects used as keys must implement the hashCode method and the equal's method.
It is synchronized.

An instance of Hashtable has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of *buckets* in the hash table, and the *initial capacity* is simply the capacity at the time the hash table is created. Note that the hash table is *open*: in the case of a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hashtable exceeds the product of the load factor and the current capacity, the capacity is increased by calling the rehash method.

The Iterators returned by the iterator and listIterator methods of the Collections returned by all of Hashtable's "collection view methods" are *fail-fast*. The Enumerations returned by Hashtable's keys and values methods are *not* fail-fast.

50. What is TreeMap?

Red-Black tree based implementation of the SortedMap interface. This class guarantees that the map will be in ascending key order, sorted according to the *natural order* for the key's class.
Note that this implementation is not synchronized**.** If multiple threads access this map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally.
Map m = Collections.synchronizedMap(new TreeMap(...));

51. What is SortedMap?

A map that further guarantees that it will be in ascending key order, sorted according to the *natural ordering* of its keys (see the Comparable interface), or by a comparator provided at sorted map creation time.

All keys inserted into a sorted map must implement the Comparable interface (or be accepted by the specified comparator). Furthermore, all such keys must be *mutually comparable*: k1.compareTo(k2) (or comparator.compare(k1, k2)) must not throw a ClassCastException for any elements k1 and k2 in the sorted map. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

52. What is LinkedHashMap?

Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (*insertion-order*). Note that insertion order is not affected if a key is *re-inserted* into the map. (A key k is reinserted into a map m if m.put(k, v) is invoked when m.containsKey(k) would return true immediately prior to the invocation.)

This class provides all of the optional Map operations, and permits null elements. Note that this implementation is not synchronized.

53. Why to override equals() and hashCode()? And How I can implement both equals() and hashCode() for Set ?

If we are implementing HashSet to store unique object then we need to implement equals () and hashCode method. If two objects are equal according to the equals () method, they must have the same hashCode() value (although the reverse is not generally true).

There are two scenarios

Case 1 : If you don't implement equals() and hashCode() method :

When you are adding objects to HashSet, HashSet checks for uniqueness using equals () and hashCode () method the class (ex. Emp class). If there is no equals () and hashCode () method the Emp class then HashSet checks default Object classes equals () and hashCode () method.

In the Object class , equals method is –

```
public boolean equals(Object obj) {
   return (this == obj);
}
```

Under this default implementation, two references are equal only if they refer to the exact same object. Similarly, the default implementation of hashCode () provided by Object is derived by mapping the memory address of the object to an integer value.

This will fail to check if two EMP object with same employee name. For Example :

```
Emp emp1 = new Emp();
emp1.setName("sat");
Emp emp2 = new Emp();
```

emp2.setName("sat");

Both the objects are same but based on the default above method both objects are different because references and hashCode are different. So in the HashSet you can find the duplicate emp objects.

To overcome the issue equals() and hashcode() method need to override.
Case 2) : If you override equals() and hashcode() method

Example : implement equals and hashcode

```
public class Emp {
   private long empId;
   String name = "";
public boolean equals(Object o) {
      if (o == this)
      return true;
      if (!(o instanceof Emp))
      return false;
      Emp emp = (Emp)o;
      return emp. empId == empId &&
      emp. name == name;

      }

   public int hashCode(){
      int result = 10;
      result = result * new Integer(String.valueOf(empId)).intValue();
      return result;
   }
}
```

In the equals() , it check for name is same or not. This way you can find out the objects are equals or not.
In the hashCode () also it return some unique value for each object. In this way if two Emp object has same empId then it will say both are same object.

Now HashSet store only unique objects.
If you do

```
Emp emp1 = new Emp();
emp1.setName("sat");
Emp emp2 = new Emp();
```

```
emp2.setName("sat");

if(emp1.equals(emp2)){
        System.out.println("equal");
    }
```

This will print : equal

54. What is the difference between java.util.Iterator and java.util.ListIterator?

Iterator: Enables you to traverse through a collection in the forward direction only, for obtaining or removing elements

ListIterator: extends Iterator, and allows bidirectional traversal of list and also allows the modification of elements.

55. Difference between HashMap and HashTable?

1. HashMap allows null keys and null values while HashTable doesn't allow null keys and values.
2. HashMap is not synchronized while HashTable is synchronized.
3. Iterator in the HashMap is fail-fast while the enumerator for the Hashtable isn't.

56. Where will you use Hashtable and where will you use HashMap?

There are multiple aspects to this decision:

1. The basic difference between a Hashtable and HashMap is that, Hashtable is synchronized while HashMap is not. Thus whenever there is a possibility of multiple threads accessing the same instance, one should use Hashtable. While if not multiple threads are going to access the same instance then use HashMap. Non synchronized data structure will give better performance than the synchronized one.

2. If there is a possibility in future that - there can be a scenario when you may require retaining the order of objects in the Collection with key-value pair then HashMap can be a good choice. As one of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you can easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable. Also if you have multiple thread accessing you HashMap then Collections.synchronizedMap () method can be leveraged. Overall HashMap gives you more flexibility in terms of possible future changes.

57. What is the Difference between Enumeration and Iterator interface?

Iterators differ from enumerations in following ways:

1. Enumeration contains 2 methods namely hasMoreElements() & nextElement() whereas Iterator contains three methods namely hasNext(), next(),remove().
2. Iterator adds an optional remove operation, and has shorter method names. Using remove () we can delete the objects but Enumeration interface does not support this feature.

58. What is the difference between Sorting performance of Arrays.sort () vs Collections.sort() ? Which one is faster? Which one to use and when?

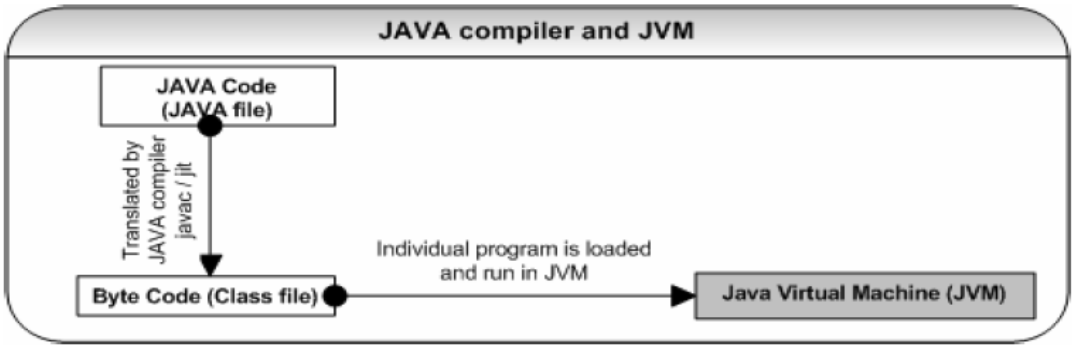
Both methods have same algorithm the only difference is type of input to them. Collections.sort () has a input as List so it does a translation of List to array and vice versa which is an additional step while sorting. So this should be used when you are trying to sort a list. Arrays.sort is for arrays so the sorting is done directly on the array. So clearly it should be used when you have a array available with you and you want to sort it.

59. **What are the differences between the Comparable and Comparator interfaces** ?

Comparable	Comparator
It uses the *compareTo ()* method. *int objectOne.compareTo(objectTwo).*	It uses the *compare ()* method. *int compare(ObjOne, ObjTwo)*
It is necessary to modify the class whose instance is going to be sorted.	A separate class can be created in order to sort the instances.
Only one sort sequence can be created.	Many sort sequences can be created.
It is frequently used by the API classes.	It used by third-party classes to sort instances.

60. What is the main difference between the Java platform and the other software platforms?

Java platform is a software-only platform, which runs on top of other hardware-based platforms like UNIX, NT etc.



The Java platform has 2 components:

Java Virtual Machine (**JVM**) – 'JVM' is software that can be ported onto various hardware platforms. Byte codes are the machine language of the JVM.

Java Application Programming Interface (**Java API**) – set of classes written using the Java language and run on the JVM.

61. What is the difference between C++ and Java?

Both are the object oriented languages.

- Java does not support pointers. Pointers are inherently tricky to use and troublesome.
- Java does not support multiple inheritances because it causes more problems than it solves. Instead Java supports multiple interface inheritance
- Java does not support destructors but adds a finalize () method. Finalize methods are invoked by the garbage collector prior to reclaiming the memory occupied by the object
- All the code in Java program is encapsulated within classes therefore Java does not have global variables or functions

62. What are the usages of Java packages?

It helps resolve naming conflicts when different packages have classes with the same names. This also helps you organize files within your project.

For example: java.io package do something related to I/O and java.net package do something to do with network and so on. If we tend to put all .java files into a single package, as the project gets bigger, then it would become a nightmare to manage all your files.

63. What is Class Loader?

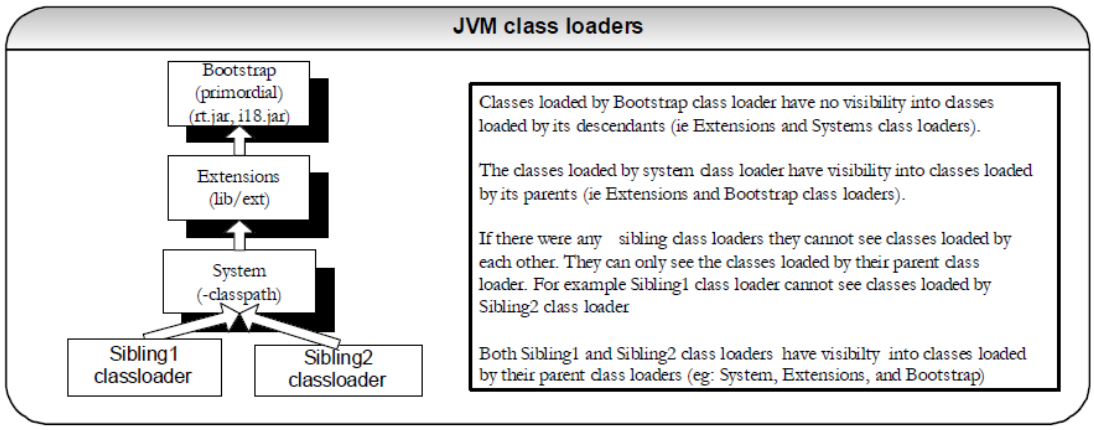
The class which are required for converting the byte code and loaded these classes into the memory is done by ClassLoader. The ClassLoader is the part of the JVM that loads classes into memory. All these classes are loaded by Bootstrap class loader or primordial class loader or through custom class loader.

The bootstrap class loader is an integral part of the JVM and is responsible for loading *trusted* classes (e.g. basic Java class library classes). User-defined class loaders, unlike the bootstrap class loader, are not intrinsic components of the JVM. They are subclasses of the java.util.ClassLoader class that are compiled and instantiated just like any other Java class. Loads JDK internal classes, *java.** packages. (as defined in the sun.boot.class. path system property, typically loads rt.jar and i18n.jar).

There are some class loaders which are created by JVM.
Bootstrap (**primordial**) - Bootstrap class loader loads those classes those which are essential for JVM to function properly. Bootstrap class loader is responsible for loading all core java classes for instance java.lang.*, java.io.* etc. Bootstrap class loader finds these necessary classes from "jdk/jre/lib/rt.jar". Bootstrap class loader cannot be instantiated from JAVA code and is implemented natively inside JVM.

Extensions - The extension class loader also termed as the standard extensions class loader is a child of the bootstrap class loader. Its primary responsibility is to load classes from the extension directories, normally located the "jre/lib/ext" directory. This provides the ability to simply drop in new extensions, such as various security extensions, without requiring modification to the user's class path.

System - The system class loader also termed application class loader is the class loader responsible for loading code from the path specified by the CLASSPATH environment variable. It is also used to load an application's entry point class that is the "static void main ()" method in a class.

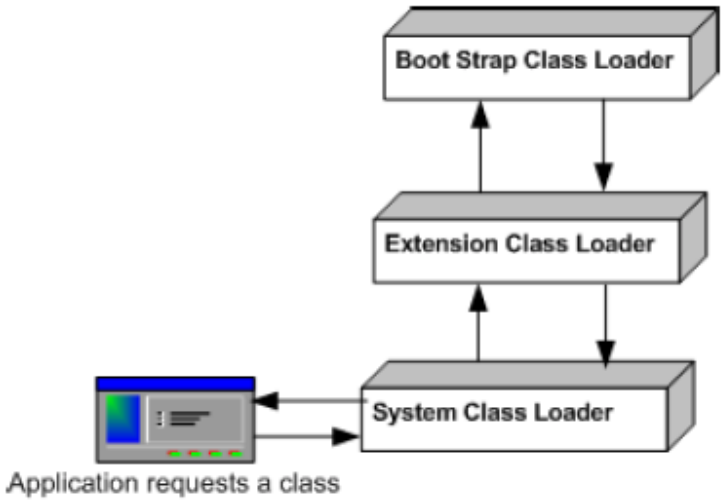


Class loaders are hierarchical and use a **delegation model** when loading a class. Class loaders request their parent to load the class first before attempting to load it

themselves. When a class loader loads a class, the child class loaders in the hierarchy will never reload the class again. Hence **uniqueness** is maintained. Classes loaded by a child class loader have **visibility** into classes loaded by its parents up the hierarchy but the reverse is not true as explained in the above diagram.

64. Can you explain the flow between bootstrap, extension and system class loader?

```
Import Koirala.Interview.Java;
        public class mySimpleClass
        {
          public static void main(String[] args)
        {
            String myStr = "I am going to get a job";
            System.out.println(myStr);
        }}
```



JVM will request the system class loader to load “java.lang.String”. But before he tries to load it will delegate to extension class loader. Extension class loader will pass it to Boot strap class loader. Now Boot Strap class loader does not have any parent so it will try to load “java.lang.String” using “rt.jar”. Now the Boot strap will return the class back using the same chain to the application. Now lets see how “Import Koirala.Interview.Java;” is loaded using the class loaders. For the import statement JVM will make a call to the system class loader who will delegate the same to the extension class loader which will delegate the same to the boot strap class loader. Boot strap loader will not find “Koirala.Interview.Java” and return nothing to the extension class loader. Extension

class loader will also check the same in its path and will not find anything thus returning nothing to the system class loader. System class loader will use its class path and load the class and return the same to the JVM who will then return it to the application.

65. What is Static class loading?

- Classes are statically loaded with Java's 'new 'operator.
- A ***NoClassDefFoundException*** is thrown if a class is referenced with Java's *"new"* operator (i.e. static loading) but the runtime system cannot find the referenced class.

66. What is dynamic class loading?

Class loader at run time. Let us look at how to load classes dynamically. Class.forName (String className); //static method which returns a Class The above static method returns the class object associated with the class name. The string className can be supplied dynamically at run time. Unlike the static loading, the dynamic loading will decide whether to load the class Car or the class Jeep at runtime based on a properties file and/or other runtime conditions. Once the class is dynamically loaded the following method returns an instance of the loaded class. It's just like creating a class object with no arguments class.newInstance (); //A non-static method, which creates an instance of a

//class (i.e. creates an object).

A ***ClassNotFoundException*** is thrown when an application tries to load in a class through its string name using the following methods but no definition for the class with the specified name could be found:
The forName(..) method in class - ***Class***.
The findSystemClass(..) method in class - ***ClassLoader***.
The loadClass(..) method in class - ***ClassLoader***.

67. What are "static initializers" or "static blocks with no function names"?

When a class is loaded, all blocks that are declared static and don't have function name (i.e. static initializers) are executed even before the constructors are executed. As the name suggests they are typically used to initialize static fields.

```
public class StaticInitializer {
   public static final int A = 5;
   public static final int B; //note that it is not → public static final int B = null;
   //note that since B is final, it can be initialized only once.

   //Static initializer block, which is executed only once when the class is loaded.

   static {
      if(A == 5)
         B = 10;
      else
         B = 5;
   }

   public StaticInitializer(){}  //constructor is called only after static initializer block
}
```

68. **Why would you prefer code reuse via composition over inheritance?**

Both the approaches make use of polymorphism and gives code reuse (in different ways) to achieve the same results but:

- The advantage of class inheritance is that it is done statically at compile-time and is easy to use. The disadvantage of class inheritance is that because it is static, implementation inherited from a parent class cannot be changed at run time. In object composition, functionality is acquired dynamically at run-time by objects collecting references to other objects. The advantage of this approach is that implementations can be replaced at run-time. This is possible because objects are accessed only through their interfaces, so one object can be replaced with another just as long as they have the same type.
- Another problem with class inheritance is that the subclass becomes dependent on the parent class implementation. This makes it harder to reuse the subclass, especially if part of the inherited implementation is no longer desirable and hence can break encapsulation. Also a change to a super class can not only ripple down the inheritance hierarchy to subclasses, but can also ripple out to code that uses just the subclasses making the design fragile by tightly coupling the subclasses with the super class. But it is easier to change the interface/implementation of the composed class.

69. **Why do you get a ConcurrentModificationException when using an iterator?**

The java.util Collection classes are fail-fast, which means that if one thread changes a collection while another thread is traversing it through with an iterator the iterator.hasNext() or iterator.next() call will throw ***ConcurrentModificationException.*** Even the synchronized collection wrapper classes *SynchronizedMap* and *SynchronizedList* are only conditionally thread-safe, which means all individual operations are thread-safe but compound operations where flow of control depends on the results of previous operations may be subject to threading issues.

Solutions 1-3: for multi-thread access situation

Solution 1: You can convert your list to an array with list.toArray() and iterate on the array. This approach is not recommended if the list is large.
Solution 2: You can lock the entire list while iterating by wrapping your code within a synchronized block. This approach adversely affects scalability of your application if it is highly concurrent.
Solution 3: If you are using JDK 1.5 then you can use the *ConcurrentHashMap* and *CopyOnWriteArrayList* classes, which provide much better scalability and the iterator returned by ConcurrentHashMap.iterator() will not throw *ConcurrentModificationException* while preserving thread-safety.

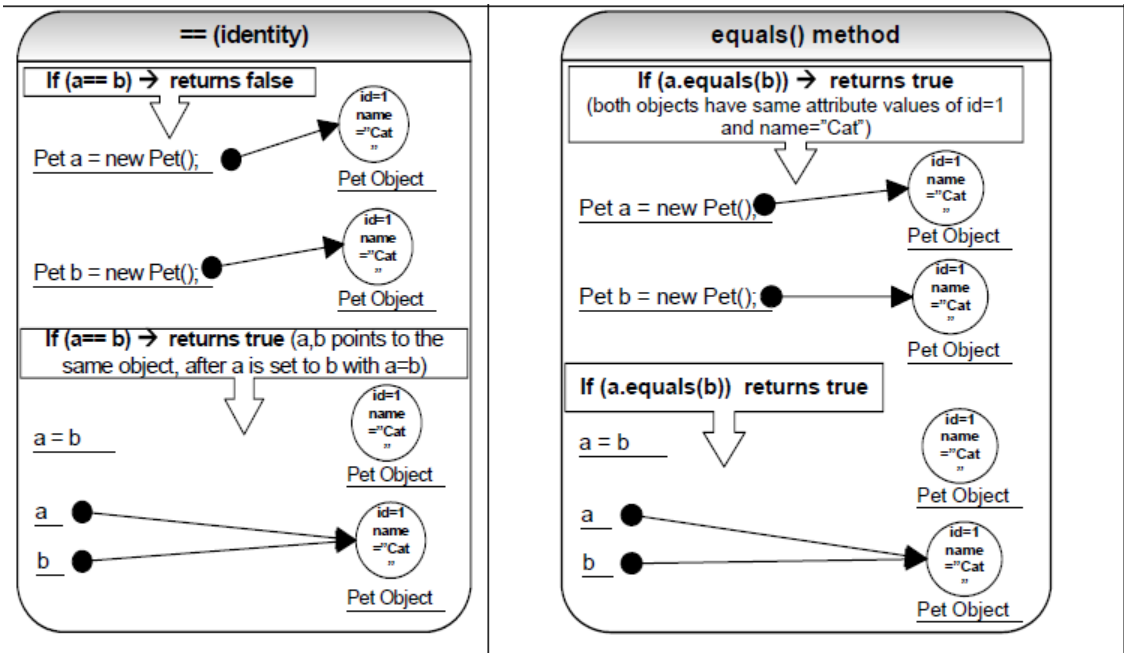
70. What is a list iterator?

The java.util.ListIterator is an iterator for lists that allows the programmer to traverse the list in either direction (i.e. forward and or backward) and modify the list during iteration.

71. **What is the difference between "==" and equals (…) method? What is the difference between shallow comparison and deep comparison of objects?**

The == returns true, if the variable reference points to the same object in memory. This is a "shallow comparison".

The equals () - returns the results of running the equals () method of a user supplied class, which compares the attribute values. The equals() method provides "deep comparison" by checking if two objects are logically equal as opposed to the shallow comparison provided by the operator ==. If equals () method does not exist in a user supplied class then the inherited Object class's equals () method is run which evaluates if the references point to the same object in memory. The object.equals () works just like the "==" operator (i.e. shallow comparison).



Note: String assignment with the "**new**" operator follow the same rule as == and equals( ) as mentioned above.

```
String str = new String("ABC"); //Wrong. Avoid this because a new String instance
                                //is created each time it is executed.
```

Variation to the above rule:

The "literal" String assignment is shown below, where if the assignment value is identical to another String assignment value created then a new String object is not created. A reference to the existing String object is returned.

```
String str = "ABC";     //Right because uses a single instance rather than
                        //creating a new instance each time it is executed.
```

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

        String s1 = new String("A"); //not recommended, use String s1 = "A"
        String s2 = new String("A"); //not recommended, use String s2 = "A"

        //standard: follows the == and equals() rule like plain java objects.

        if (s1 == s2) {  //shallow comparison
            System.out.println("references/identities are equal"); //never reaches here
        }
        if (s1.equals(s2)) {  //deep comparison
            System.out.println("values are equal");   // this line is printed
        }

        //variation: does not follow the == and equals rule

        String s3 = "A";  //goes into a String pool.
        String s4 = "A";  //refers to String already in the pool.

        if (s3 == s4) { //shallow comparison
            System.out.println("references/identities are equal"); //this line is printed
        }
        if (s3.equals(s4)) { //deep comparison
            System.out.println("values are equal"); //this line is also printed
        }
    }
}
```

String class is designed with Flyweight design pattern. When you create a String constant as shown above in the variation, (i.e. String s3 = "A", s4= "A"), it will be checked to see if it is already in the String pool. If it is in the pool, it will be picked up from the pool instead of creating a new one. Flyweights are shared objects and using them can result in substantial performance gains.

72. What is an intern () method in the String class?

This method is used to check whether the newly created string object is already present in String pool or not. S1.intern (s2)

73. What is the main difference between a String and a StringBuffer class?

String	StringBuffer / StringBuilder (added in J2SE 5.0)
String is **immutable**: you can't modify a string object but can replace it by creating a new instance. Creating a new instance is rather expensive.	StringBuffer is **mutable**: use StringBuffer or StringBuilder when you want to modify the contents. ***StringBuilder*** was added in Java 5 and it is identical in all respects to ***StringBuffer*** except that it is not synchronized, which makes it slightly faster at the cost of not being thread-safe.

//Inefficient version using immutable String
String output = "Some text"
Int count = 100;
for(int i =0; i<count; i++) {
output += i;
}
return output;
	//More efficient version using mutable StringBuffer
	StringBuffer output = new **StringBuffer(110)**;// set an initial size of 110
	output.append("Some text");
	for(int i =0; i<count; i++) {
	output.append(i);
	}
	return output.toString();

The above code would build 99 new String objects, of which 98 would be thrown away immediately. Creating new objects is not efficient.	The above code creates only two new objects, the *StringBuffer* and the final *String* that is returned. StringBuffer expands as needed, which is costly however, so it would be better to initialize the *StringBuffer* with the correct size from the start as shown.

Another important point is that creation of extra strings is not limited to overloaded mathematical operator "+" but there are several methods like **concat(), trim(), substring(),** and **replace()** in String classes that generate new string instances. So use StringBuffer or StringBuilder for computation intensive operations, which offer better performance.

74. **What is an immutable object?**

Immutable objects whose state (i.e. the object's data) does not change once it is instantiated (i.e. it becomes a read-only object after instantiation). Immutable classes are ideal for representing numbers (e.g. java.lang. Integer, java.lang.Float, java.lang.BigDecimal etc are immutable objects), enumerated types, colors (e.g. java.awt.Color is an immutable object), short lived objects like events, messages etc.

75. How will you write an immutable class?

Writing an immutable class is generally easy but there can be some tricky situations. Follow the following guidelines:

1. A class is declared final (i.e. final classes cannot be extended).public final class MyImmutable { … }

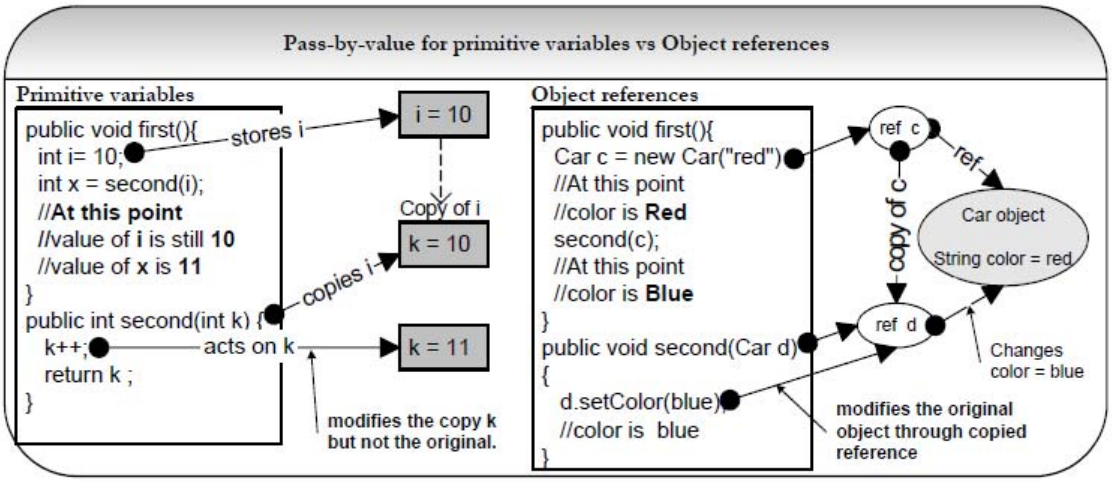
2. All its fields are final (final fields cannot be mutated once assigned).

private final int[] myArray; //do not declare as private final int[] myArray = null;

3. Do not provide any methods that can change the state of the immutable object in any way – not just setXXX methods, but any methods which can change the state.

76. What is the main difference between pass-by-reference and pass-by-value?

In Java, if a calling method passes a reference of an object as an argument to the called method then the passed in reference gets copied first and then passed to the called method. Both the original reference that was passed-in and the copied reference will be pointing to the same object. So no matter which reference you use, you will be always modifying the same original object, which is how the pass-by-reference works as well.

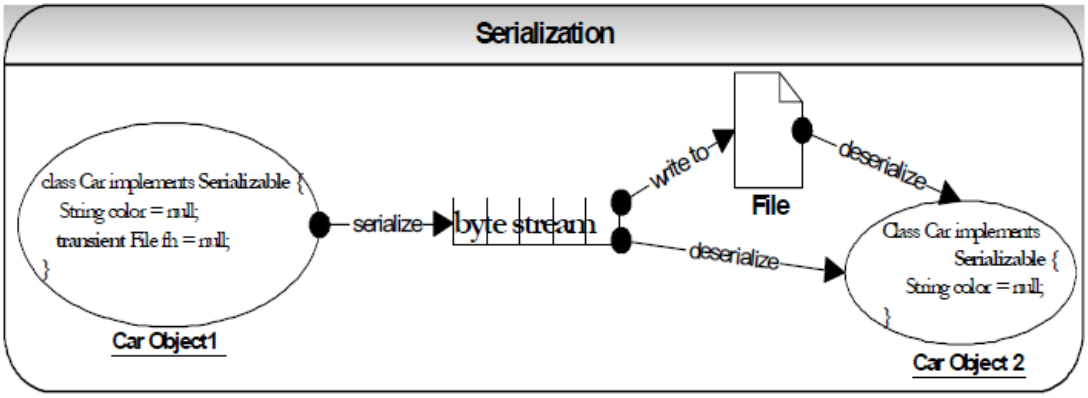


If your method call involves inter-process (e.g. between two JVMs) communication, then the reference of the calling method has a different address space to the called method sitting in a separate process (i.e. separate JVM). Hence inter-process communication involves calling method passing objects as arguments to called method **by-value** in a serialized form, which can adversely affect performance due to marshalling and unmarshaling cost.

77. What is serialization? How would you exclude a field of a class from serialization or what is a transient variable?

Serialization is a process of reading or writing an object. It is a process of saving an object's state to a sequence of bytes, as well as a process of rebuilding those bytes back into a live object at some future time. An object is marked serializable by implementing the java.io.Serializable interface, which is only a marker interface -- it simply allows the serialization mechanism to verify that the class can be persisted, typically to a file.

Transient variables cannot be serialized. The fields marked transient in a serializable object will not be transmitted in the byte stream. An example would be a file handle, a database connection, a system thread etc. Such objects are only meaningful locally. So they should be marked as transient in a serializable class.



A common use of serialization is to use it to send an object over the network or if the state of an object needs to be persisted to a flat file or a database. (Refer Q57 on Enterprise section).

Deep cloning or copy can be achieved through serialization. This may be fast to code but will have performance implications. To serialize the above "Car" object to a file (sample for illustration purpose only, should use try {} catch {} block):

```
Car car = new Car();   // The "Car" class implements a java.io.Serializable interface
FileOutputStream fos = new FileOutputStream(filename);
ObjectOutputStream out = new  ObjectOutputStream(fos);
out.writeObject(car); // serialization mechanism happens here
out.close();
```

All objects implement a clone method

Every object is responsible for cloning itself via its clone () method. So when the parent is called it makes calls to all the referenced objects inside the class and calls its clone method.

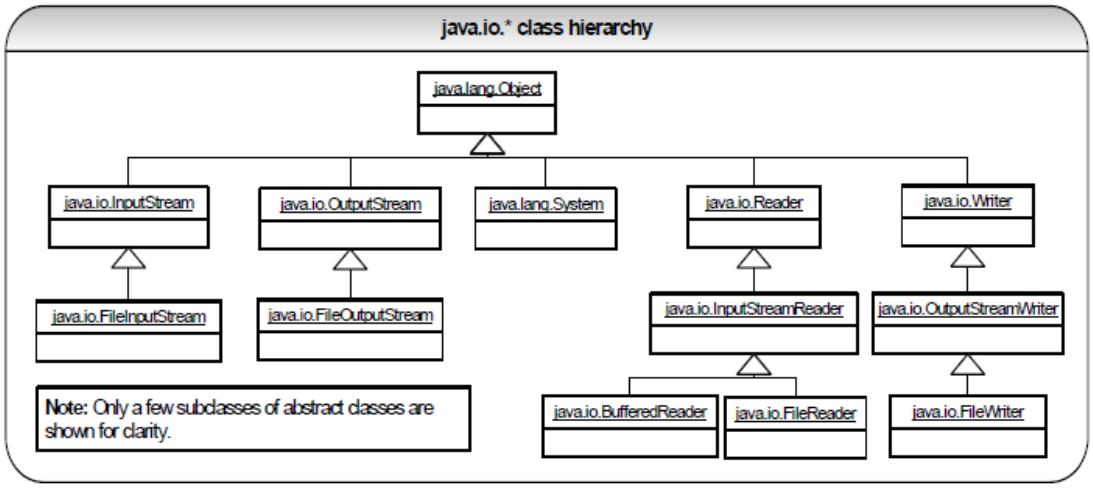
78. What is the common use? What is a serial version id?

Say you create a "Car" class, instantiate it, and write it out to an object stream. The flattened car object sits in the file system for some time. Meanwhile, if the "Car" class is modified by adding a new field. Later on, when you try to read (i.e. deserialize) the flattened "Car" object, you get the **java.io.InvalidClassException** – because all serializable classes are automatically given a unique identifier. This exception is thrown when the identifier of the class is not equal to the identifier of the flattened object. If you really think about it, the exception is thrown because of the addition of the new field. You can avoid this exception being thrown by controlling the versioning yourself by declaring an explicit **serialVersionUID**. There is also a small performance benefit in explicitly declaring your serialVersionUID (because does not have to be calculated). So, it is best practice to add your own **serialVersionUID** to your Serializable classes as soon as you create them as shown below:

79. Explain about File I/O stream? Explain the Java I/O streaming concept?

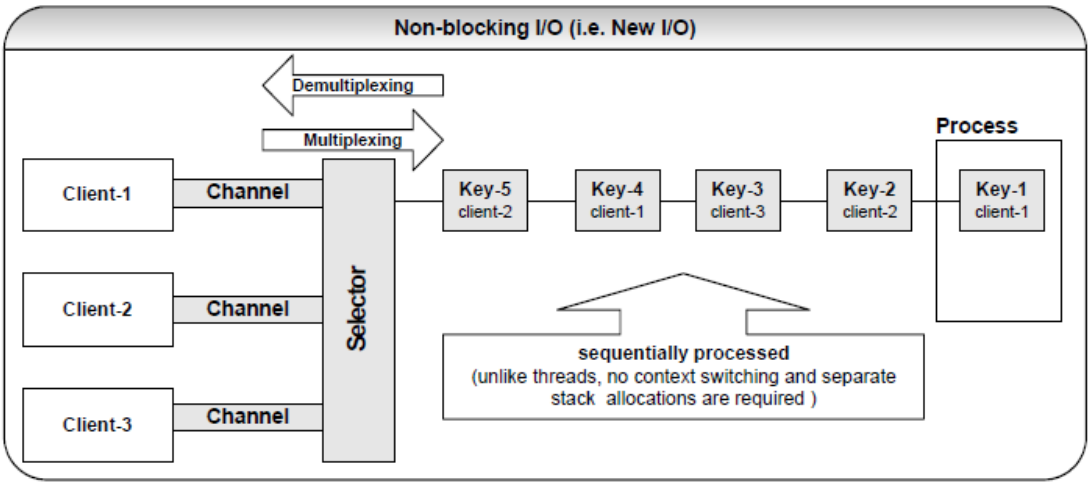
There are 2 kinds of streams.

- Byte streams (8 bit bytes) Abstract classes are: InputStream and OutputStream
- Character streams (16 bit UNICODE) Abstract classes are: Reader and Writer



80. How does the new I/O (NIO) offer better scalability and better performance?

Java has long been not suited for developing programs that perform a lot of I/O operations. Furthermore, commonly needed tasks such as file locking, non-blocking and asynchronous I/O operations and ability to map file to memory were not available. Non-blocking I/O operations were achieved through work around such as multithreading or using JNI. The New I/O API (aka NIO) in J2SE 1.4 has changed this situation.



The non-blocking I/O mechanism is built around Selectors and Channels. Channels, Buffers and Selectors are the core of the NIO.

A channel class represents a bi-directional communication channel (similar to InputStream and OutputStream) between datasources such as a socket, a file, or an application component, which is capable of performing one or more I/O operations such as reading or writing. Channels can be non-blocking, which means, no I/O operation will wait for data to be read or written to the network. The good thing about NIO channels is that they can be asynchronously interrupted and closed. So if a thread is blocked in an I/O operation on a channel, another thread can interrupt that blocked thread.

A Selector class enables multiplexing (combining multiple streams into a single stream) and demultiplexing (separating a single stream into multiple streams) I/O events and makes it possible for a single thread to efficiently manage many I/O channels. A Selector monitors selectable channels, which are registered with it for I/O events like connect, accept, read and write. The keys (i.e. Key1, Key2 etc represented by the SelectionKey class) encapsulate the relationship between a specific selectable channel and a specific selector.

Buffers hold data. Channels can fill and drain Buffers. Buffers replace the need for you to do your own buffer management using byte arrays. There are different types of Buffers like ByteBuffer, CharBuffer, DoubleBuffer, etc.

81. How can you improve Java I/O performance?

Java applications that utilize Input/Output are excellent candidates for performance tuning. Profiling of Java applications that handle significant volumes of data will show significant time spent in I/O operations. This means substantial gains can be

had from I/O performance tuning. Therefore, I/O efficiency should be a high priority for developers looking to optimally increase performance. The basic rules for speeding up I/O performance are

- Minimize accessing the hard disk.
- Minimize accessing the underlying operating system.
- Minimize processing bytes and characters individually.

Let us look at some of the techniques to improve I/O performance. Use buffering to minimize disk access and underlying operating system. As shown below, with buffering large chunks of a file are read from a disk and then accessed a byte or character at a time.

Without buffering : inefficient code

```
try{
   File f = new File("myFile.txt");
   FileInputStream fis = new FileInputStream(f);
   int count = 0;
   int b = 0;
   while((b = fis.read()) != -1){
     if(b== '\n') {
       count++;
     }
   }
    // fis should be closed in a finally block.
    fis.close() ;
 }
catch(IOException io){}
```

Note: fis.read() is a native method call to the underlying operating system.

With Buffering: yields better performance

```
try{
   File f = new File("myFile.txt");
   FileInputStream fis = new FileInputStream(f);
   BufferedInputStream bis = new BufferedInputStream(fis);
   int count = 0;
   int b = 0 ;
   while((b = bis.read()) != -1){
     if(b== '\n') {
       count++;
     }
   }
    //bis should be closed in a finally block.
    bis.close() ;
 }
catch(IOException io){}
```

Note: bis.read() takes the next byte from the input buffer and only rarely access the underlying operating system.

82. What is the main difference between shallow cloning and deep cloning of objects?

Java provides a mechanism for creating copies of objects called cloning. There are two ways to make a copy of an object called shallow copy and deep copy.

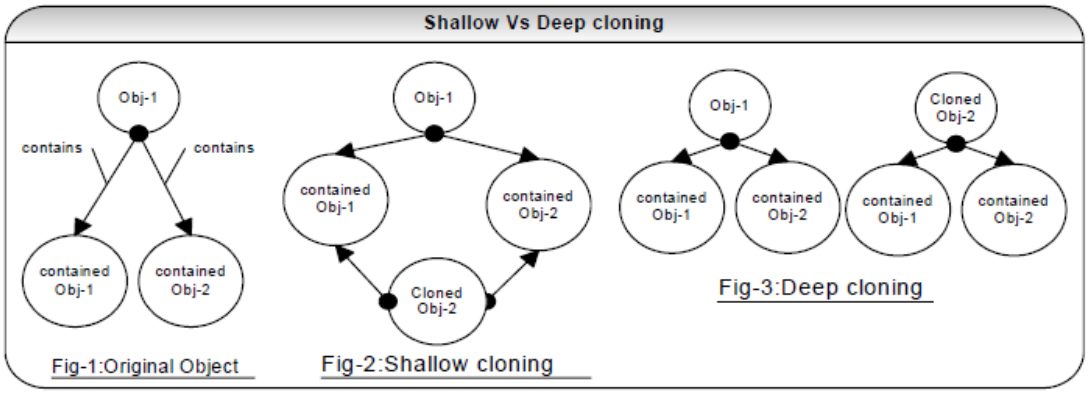
The default behavior of an object's clone () method automatically yields a shallow copy. So to achieve a deep copy the classes must be edited or adjusted.

Shallow copy: Shallow copy is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the references are copied. Thus, if the object you are copying contains references to yet other objects, a shallow copy refers to the same sub objects.

Shallow Copy concept is not applicable to the classes having only primitive data type members as in that case the default cloning will also result into a Deep Copy only.

Deep copy: Deep copy is a complete duplicate copy of an object. If an object has references to other objects, complete new copies of those objects are also made. A deep copy generates a copy not only of the primitive values of the original object, but copies of all sub objects as well, all the way to the bottom. If you need a true, complete copy of the original object, then you will need to implement a full deep copy for the object.

If a deep copy is performed on obj-1 as shown in fig-3 then not only obj-1 has been copied but the objects contained within it have been copied as well. Serialization can be used to achieve deep cloning. Deep cloning through serialization is faster to develop and easier to maintain but carries a performance overhead.



Java supports shallow and deep copy with the Cloneable interface to create copies of objects. To make a clone of a Java object, you declare that an object implements Cloneable, and then provide an override of the clone method of the standard Java Object base class. Implementing Cloneable tells the java compiler that your object is Cloneable. The cloning is actually done by the clone method.

```
public static List deepCopy(List listCars) {
     List copiedList = new ArrayList(10);
     for (Object object : listCars) {   //JDK 1.5 for each loop
        Car original = (Car)object;
        Car carCopied = new Car();  //instantiate a new Car object
        carCopied.setColor((original.getColor()));
        copiedList.add(carCopied);
    }
    return copiedList;
}
```

83. Why would you prefer a short circuit "&&, ||" operators over logical "& , |" operators?

Firstly *NullPointerException* is by far the most common *RuntimeException*. If you use the logical operator you can get a *NullPointerException*. This can be avoided easily by using a short circuit "&&" operator as shown below.

84. How does Java allocate stack and heap memory? Explain re-entrant, recursive and idempotent methods/functions?

Each time an object is created in Java it goes into the area of memory known as **heap**. The primitive variables like int and double are allocated in the **stack** (i.e. Last in First Out queue), if they are local variables and in the **heap** if they are member variables (i.e. fields of a class). In Java methods and local variables are pushed into stack when a method is invoked and stack pointer is decremented when a method call is completed. In a multi-threaded application each thread will have its own stack but will share the same heap. This is why care should be taken in your code to avoid any concurrent access issues in the heap space. The stack is thread-safe because each thread will have its own stack with say 1MB RAM allocated for each thread but the heap is not thread-safe unless guarded with **synchronization** through your code. The stack space can be increased with the **–Xss** option.

All Java methods are automatically **re-entrant**. It means that several threads can be executing the same method at once, each with its own copy of the local variables. A Java method may call itself without needing any special declarations. This is known as a **recursive** method call. Given enough stack space, recursive method calls are perfectly valid in Java though it is tough to debug. Recursive methods are useful in removing iterations from many sorts of algorithms. All recursive functions are re-entrant but not all re-entrant functions are recursive. **Idempotent** methods are methods, which are written in such a way that repeated calls to the same method with the same arguments yield same results. **For example** clustered EJBs, which are written with idempotent methods, can automatically recover from a server failure as long as it can reach another server (i.e. scalable).

85. What are Inner class?

You can put the definition of one class inside the definition of another class. The inside class is called an inner class and the enclosing class is called an outer class. So when you define an inner class, it is a member of the outer class in much the same way as other members like attributes, methods and constructors Code without inner classes is more maintainable and readable. When you access private data members of the outer class, the JDK compiler creates package-access member functions in the outer class for the inner class to access the private members. This leaves a security hole. In general we should avoid using inner classes. Use inner class only when an inner class is only relevant

in the context of the outer class and/or inner class can be made private so that only outer class can access it.

Explain outer and inner classes?

Class Type		Description	Example + Class name
Outer class	Package member class or interface	Top level class. Only type JVM can recognize.	//package scope **class Outside{}** Outside.class

Inner class	static nested class or interface	Defined within the context of the top-level class. Must be static & can access static members of its containing class. No relationship between the instances of outside and Inside classes.	//package scope
			class Outside {
			static class Inside{ }
			}
			
			Outside.class ,Outside$Inside.class

Inner class	Member class	Defined within the context of outer class, but non-static. Until an object of Outside class has been created you can't create Inside.	class Outside{
			class Inside(){}
			}
			
			Outside.class , Outside$Inside.class

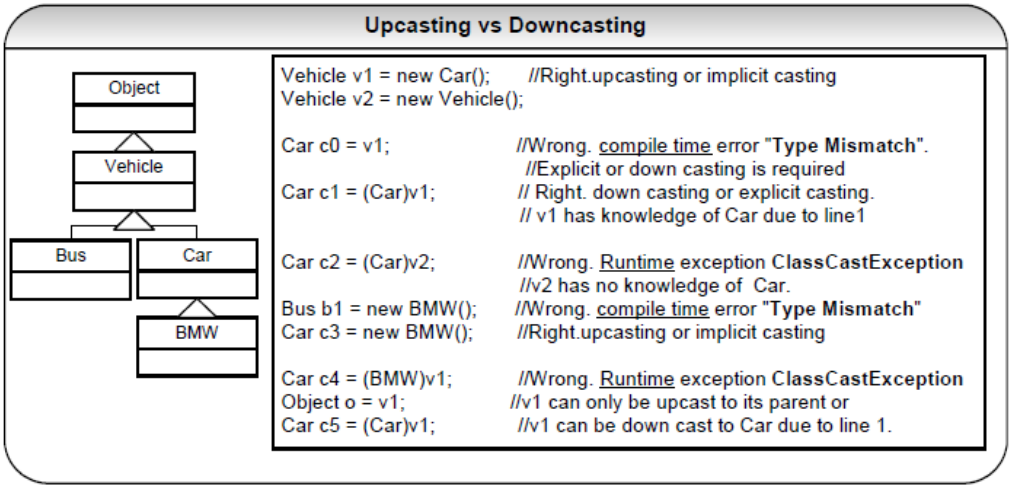
Inner class	Local class	Defined within a block of code. Can use final local variables and final method parameters. Only visible within the block of code that defines it.	class Outside {
			void first() {
			final int i = 5;
			class Inside{}
			}
			}
			
			Outside.class , Outside1Inside.class

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

86. What is type casting?

byte → short → int → long → float → double

```
int i = 5;
long j = i;             //Right. Up casting or implicit casting
byte b1 = i;            //Wrong. Compile time error "Type Mismatch".
byte b2 = (byte) i ;    //Right. Down casting or explicit casting is required.
```



87. What do you know about the Java garbage collector? When does the garbage collection occur?

    Each time an object is created in Java, it goes into the area of memory known as heap. The Java heap is called the garbage collectable heap. The garbage collection cannot be forced. The garbage collector runs in low memory situations. When it runs, it releases the memory allocated by an unreachable object. The garbage collector runs on a low priority daemon (i.e. background) thread. You can nicely ask the garbage collector to collect garbage by calling System.gc() but you can't force it.do anything. Then the object becomes unreachable and the garbage collector will figure it out. Java automatically collects all the unreachable objects periodically and releases the memory consumed by those unreachable objects to be used by the future reachable objects. We can use the following options with the Java command to enable tracing for garbage collection events.

    java -verbose:gc //reports on each garbage collection event.

88. Explain type of references?

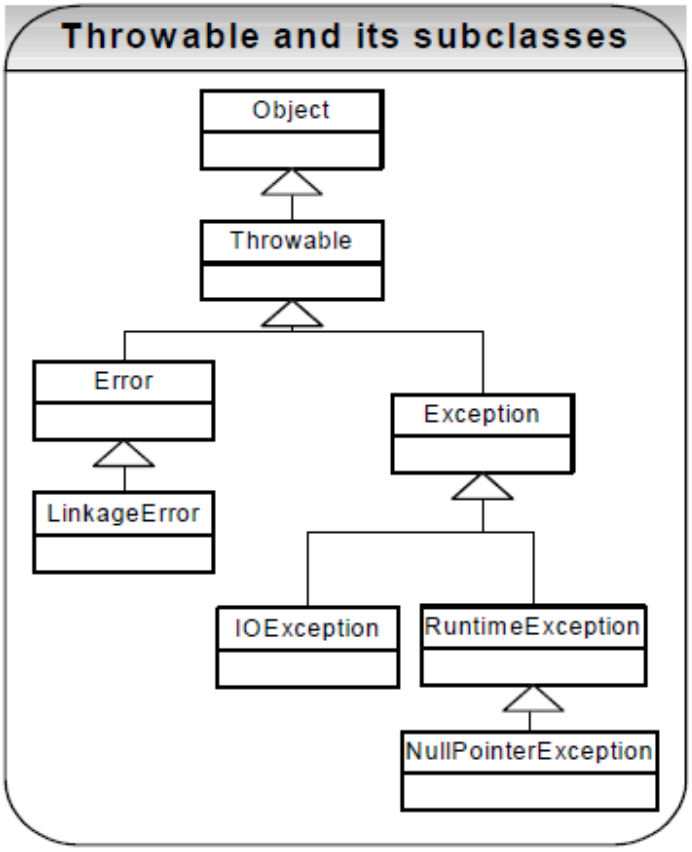
    Strong References – These are those references which are still in used by client and Garbage collector will not remove the strong references.

    Soft References - A ***soft reference will*** only get removed if memory is low. So it is useful for implementing caches while avoiding memory leaks.

    Weak reference - A weak reference will get removed on the next garbage collection cycle. Can be used for implementing canonical maps. The java.util.WeakHashMap implements a HashMap with keys held by weak references.

Phantom References - 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.

89. Discuss the Java error handling mechanism? What is the difference between Runtime (unchecked) exceptions and checked exceptions? What is the implication of catching all the exceptions with the type "Exception"



Errors: When a dynamic linking failure or some other "hard" failure in the virtual machine occurs, the virtual machine throws an Error. Typical Java programs should not catch Errors. In addition, it's unlikely that typical Java programs will ever throw Errors either.

Exception - Exceptions indicate that a problem occurred but that the problem is not a serious JVM problem. An Exception class has many subclasses. These descendants indicate various types of exceptions that can occur. For example, NegativeArraySizeException indicates that a program attempted to create an array with a negative size. One exception subclass has special meaning in the Java language: RuntimeException.All the exceptions except RuntimeException are compiler checked

exceptions. If a method is capable of throwing a checked exception it must declare it in its method header or handle it in a try/catch block. Failure to do so raises a compiler error.

Runtime Exceptions (unchecked exception)

A RuntimeException class represents exceptions that occur within the Java virtual machine (during runtime). An example of a runtime exception is NullPointerException. The cost of checking for the runtime exception often outweighs the benefit of catching it. Exceptions generated from runtime are called unchecked exceptions, since it is not possible for the compiler to determine that your code will handle the exception. Exception classes that descend from RuntimeException and Error classes are unchecked exceptions. Attempting to catch or specify all of them all the time would make your code unreadable and unmaintainable. The compiler allows runtime exceptions to go uncaught and unspecified. If you like, you can catch these exceptions just like other exceptions. However, you do not have to declare it in your
“throws" clause or catch it in your catch clause.

ArithmeticException:- Arithmetic error, such as divide-by-zero.

ArrayIndexOutOfBoundsException :- Array index is out-of-bounds.

ArrayStoreException :- Assignment to an array element of an incompatible type.

ClassCastException:- Invalid cast.

IllegalArgumentException :- Illegal argument used to invoke a method.

IllegalMonitorStateException :- Illegal monitor operation, such as waiting on an unlocked thread.

IllegalStateException :- Environment or application is in incorrect state.

IllegalThreadStateException :- Requested operation not compatible with current thread state.

IndexOutOfBoundsException :- Some type of index is out-of-bounds.

NegativeArraySizeException :- Array created with a negative size.

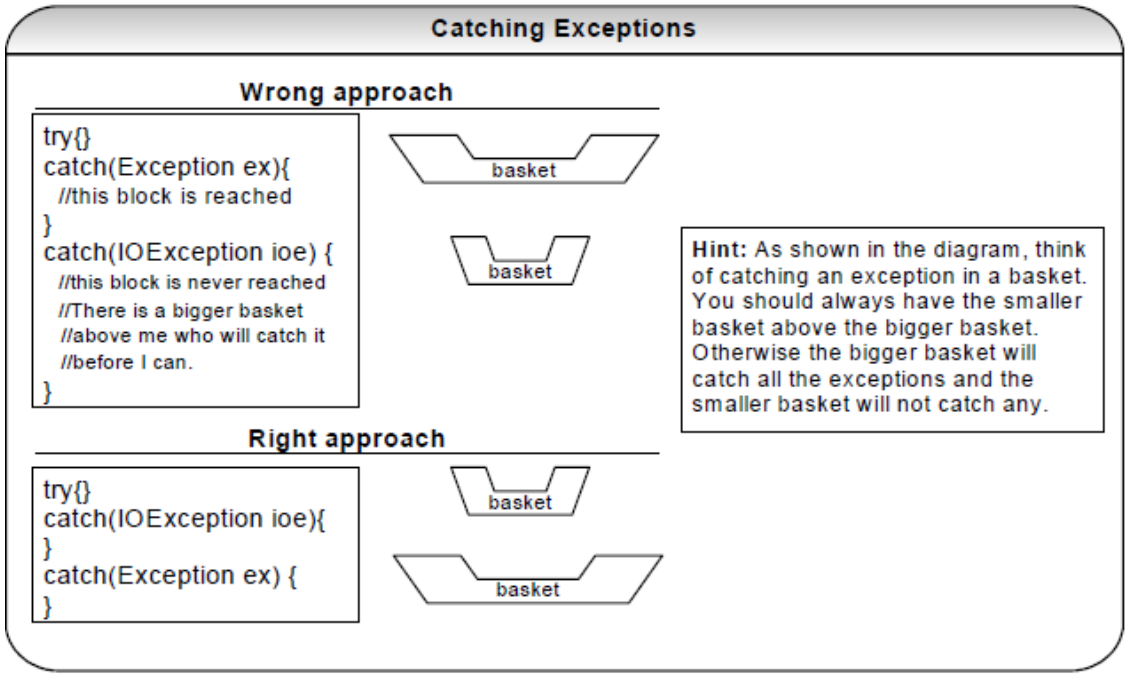
NullPointerException :- Invalid use of a null reference.

NumberFormatException :- Invalid conversion of a string to a numeric format.

SecurityException :- Attempt to violate security.

StringIndexOutOfBounds :- Attempt to access index outside the bounds of a string.

For example if you catch type Exception in your code then it can catch or throw its descendent types like IOException as well. So if you catch the type Exception before the type IOException then the type Exception block will catch the entire exceptions and type IOException block is never reached. In order to catch the type IOException and handle it differently to type Exception, IOException should be caught first (remember that you can’t have a bigger basket above a smaller basket).



90. What is a user defined exception?

User defined exceptions may be implemented by defining a new exception class by extending the Exception class.

```
public class MyException extends Exception {

   /* class definition of constructors goes here */
   public MyException() {
      super();
   }

   public MyException (String errorMessage) {
      super (errorMessage);
   }
}
```

Throw and/or throws statement is used to signal the occurrence of an exception. To throw an exception:

throw new MyException("I threw my own exception.")

To declare an exception: public myMethod() throws MyException {…}

91. What are chained Exception?

The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. Consider a situation in which you are getting null exception because of permission issues. You

would like to know if this exception is associated with some other exception. For chained exceptions there are two constructors and two methods. The constructors are shown here:

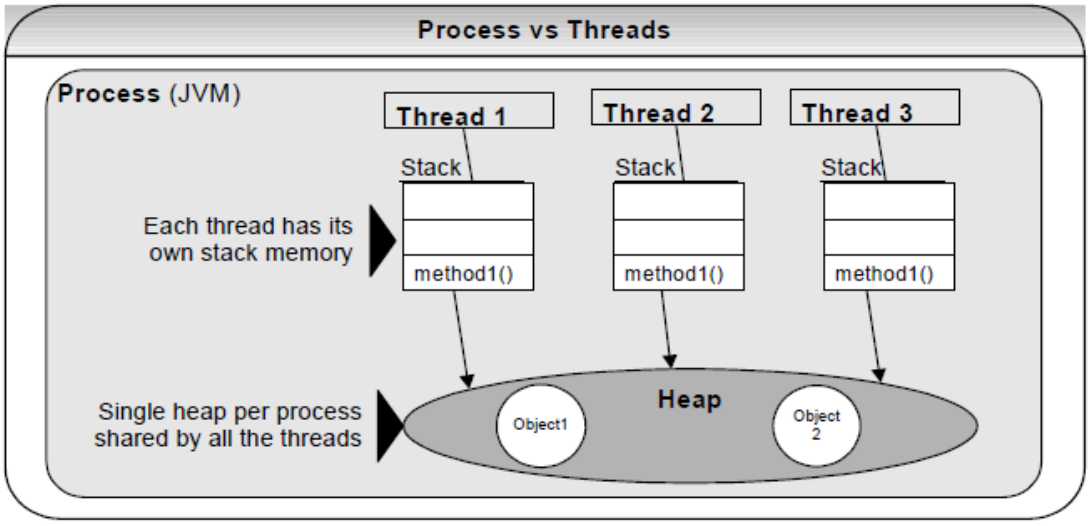
Throwable(Throwable causeExc)

Throwable(String msg, Throwable causeExc)

In the first form, causeExc is the exception that causes the current exception. That is, causeExc is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception.

92. What is the difference between processes and threads?

A process is an execution of a program but a thread is a single execution sequence within the process. A process can contain multiple threads. A thread is sometimes called a lightweight process.



A JVM runs in a single process and threads in a JVM share the heap belonging to that process. That is why several threads may access the same object. Threads **share the heap and have their own stack space**. This is how one thread's invocation of a method and its local variables are kept thread safe from other threads. But the heap is not thread-safe and must be synchronized for thread safety

93. Explain different ways of creating a thread?

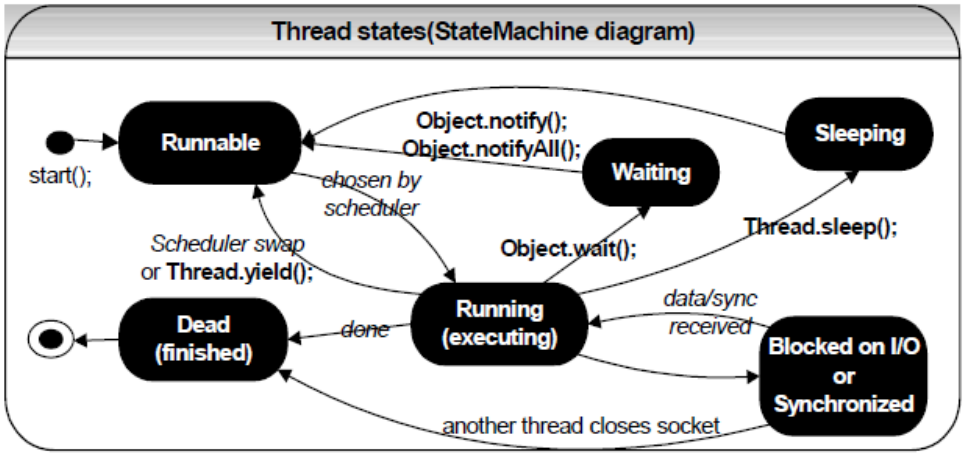
Threads can be used by either:

- Extending the Thread class
- Implementing the Runnable interface

```
class Counter extends Thread {

    //method where the thread execution will start
    public void run(){
        //logic to execute in a thread
    }

    //let's see how to start the threads
    public static void main(String[] args){
        Thread t1 = new Counter();
        Thread t2 = new Counter();
        t1.start();  //start the first thread. This calls the run() method.
        t2.start(); //this starts the 2nd thread. This calls the run() method.
    }
}
```

```
class Counter extends Base implements Runnable {

    //method where the thread execution will start
    public void run(){
         //logic to execute in a thread
    }

    //let us see how to start the threads
    public static void main(String[] args){
         Thread t1 = new Thread(new Counter());
         Thread t2 = new Thread(new Counter());
         t1.start();  //start the first thread. This calls the run() method.
         t2.start(); //this starts the 2nd thread. This calls the run() method.
    }
}
```

94. Briefly explain high-level thread states?



Runnable — waiting for its turn to be picked for execution by the thread scheduler based on thread priorities.

Running: The processor is actively executing the thread code. It runs until it becomes blocked, or voluntarily gives up its turn with this static method *Thread.yield()*. Because of context switching overhead, *yield()* should not be used very frequently.

Waiting: A thread is in a **blocked state** while it waits for some external processing such as file I/O to finish.

Sleeping: Java threads are forcibly put to sleep (suspended) with this overloaded method: Thread.sleep(milliseconds), Thread.sleep(milliseconds, nanoseconds);

Blocked on I/O: Will move to runnable after I/O condition like reading bytes of data etc changes.

Blocked on synchronization: Will move to Runnable when a **lock is acquired.**

Dead: The thread is finished working.

95. What is the difference between yield and sleeping? What is the difference between the methods sleep () and wait()?

    Yield causes the currently executing thread object to temporarily pause and allow other threads to execute. When a task invokes yield (), it changes from running state to runnable state. When a task invokes sleep (), it changes from running state to waiting/sleeping state.

    The method wait (1000), causes the current thread to sleep up to one second. A thread could sleep less than 1 second if it receives the notify () or notifyAll () method call. The call to sleep (1000) causes the current thread to sleep for exactly 1 second.

96. How does thread synchronization occurs inside a monitor? What levels of synchronization can you apply?

    In Java programming, each object has a lock. A thread can acquire the lock for an object by using the ***synchronized*** keyword. The synchronized keyword can be applied in **method level** (coarse grained lock – can affect performance adversely) or **block level of code** (fine grained lock).

```
class MethodLevel {
    //shared among threads
    SharedResource  x, y ;

    pubic void synchronized method1() {
       //multiple threads can't access
    }

    pubic void synchronized method2() {
       //multiple threads can't access
    }

    public void method3() {
      //not synchronized
      //multiple threads can access
    }
}
```

```
class BlockLevel {
    //shared among threads
    SharedResource  x, y ;
    //dummy objects for locking
    Object  xLock = new Object(), yLock = new Object();

    pubic void  method1() {
       synchronized(xLock){
          //access x here. thread safe
       }

       //do something here but don't  use SharedResource  x, y
       // because will not be thread-safe

        synchronized(xLock) {
            synchronized(yLock) {
                //access x,y here. thread safe
            }
         }

        //do something here but don't use  SharedResource  x, y
        //because will not be thread-safe
    }//end of method1
}
```

The JVM uses locks in conjunction with monitors. A monitor is basically a guardian who watches over a sequence of synchronized code and making sure only one thread at a time executes a synchronized piece of code. Each monitor is associated with an object reference. When a thread arrives at the first instruction in a block of code it must obtain a lock on the referenced object. The thread is not allowed to execute the code until it obtains the lock. Once it has obtained the lock, the thread enters the block of protected code. When the thread leaves the block, no matter how it leaves the block, it releases the lock on the associated object.

Without synchronization, it is possible for one thread to modify a shared object while another thread is in the process of using or updating that object's value. This often causes dirty data and leads to significant errors. The disadvantage of synchronization is that it can cause deadlocks when two threads are waiting on each other to do something. Also synchronized code has the overhead of acquiring lock, which can adversely affect the performance.

ThreadLocal is a handy class for simplifying development of thread-safe concurrent programs by making the object stored in this class not sharable between threads. *ThreadLocal* class encapsulates non-thread-safe classes to be safely used in a multi-threaded environment and also allows you to create per-thread-singleton.

97. **What is a daemon thread?**

Daemon threads are sometimes called "service" or "background" threads. These are threads that normally run at a low priority and provide a basic service to a program when activity on a machine is reduced. An example of a daemon thread that is

continuously running is the garbage collector thread. The JVM exits whenever all nondaemon threads have completed, which means that all daemon threads are automatically stopped. To make a thread as a daemon thread in Java myThread.setDaemon(true);

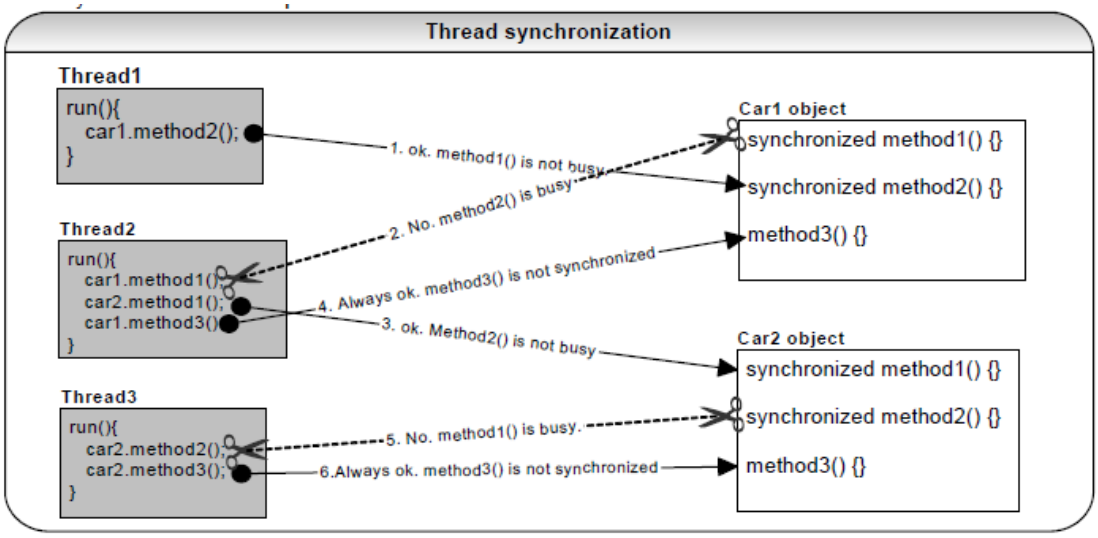
98. What does join method do?

t.join () allows the current thread to wait indefinitely until thread "t" is finished. t.join (5000) allows the current thread to wait for thread "t" to finish but does not wait longer than 5 seconds.

```
try {
  t.join(5000); //current thread waits for thread "t" to complete but does not wait more than 5 sec
  if(t.isAlive()){
     //timeout occurred. Thread "t" has not finished
  }
  else {
     //thread "t" has finished
  }
}
```

99. If 2 different threads hit 2 different synchronized methods in an object at the same time will they both continue?

No. Only one method can acquire the lock



100. **When InvalidMonitorStateException is thrown? Why?**

This exception is thrown when you try to call wait()/notify()/notifyAll() any of these methods for an Object from a point in your program where u are NOT having a lock on that object.(i.e. u r not executing any synchronized block/method of that object

and still trying to call wait()/notify()/notifyAll()). All of these method throws IllegalMonitorStateException. Since this exception is a subclass of RuntimeException so we r not bound to catch it (although u may if u want to). and being a RuntimeException this exception is not mentioned in the signature of wait(), notify(), notifyAll() methods.

101. What is a **singleton** pattern? How do you code it in Java?

A singleton is a class that can be instantiated only one time in a JVM per class loader. Repeated calls always return the same instance. Ensures that a class has only one instance, and provide a global point of access. It can be an issue if singleton class gets loaded by multiple class loaders or JVMs.

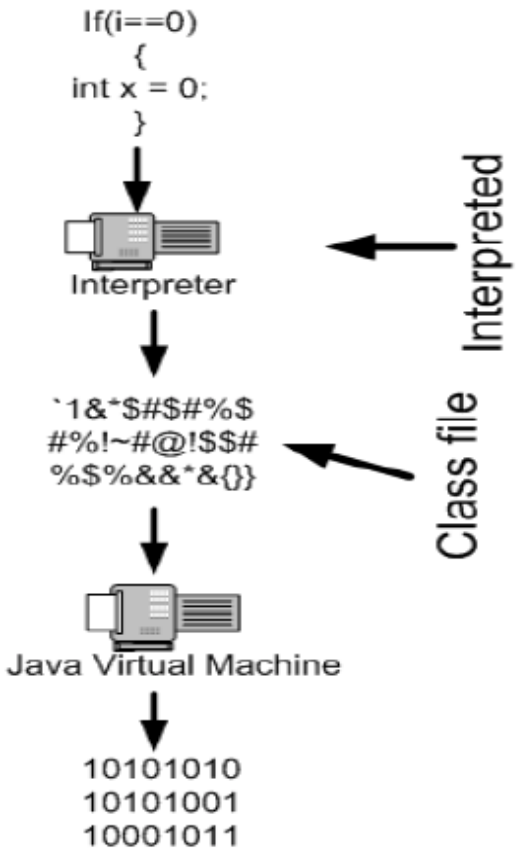
```
public class OnlyOne {

    private static OnlyOne one = new OnlyOne();

    // private constructor. This class cannot be instantiated from outside and
    // prevents subclassing.
    private OnlyOne(){}

    public static OnlyOne getInstance() {
            return one;
    }
}
```

- When to use: Use it when only a single instance of an object is required in memory for a single point of access. For example the following situations require a single point of access, which gets invoked from various parts of the code.Accessing application specific properties through a singleton object, which reads them for the first time from a properties file and subsequent accesses are returned from in-memory objects. Also there could be another piece of code, which periodically synchronizes the in-memory properties when the values get modified in the underlying properties file. This piece of code accesses the in-memory objects through the singleton object (i.e. global point of access).

- Accessing in-memory object cache or object pool, or non-memory based resource pools like sockets, connections etc through a singleton object (i.e. global point of access).

102. **What is JIT (Just-in-Time) Compilation?**

There are two basic ways languages are compiled:-
Compiled Languages
Interpreted Languages


If(i==0)
{
int x = 0;
}
Interpreter
Interpreted
`1&*$#$#%$
#%!~#@!$$#
%$%&&*&{}}
Class file
Java Virtual Machine
10101010
10101001
10001011

interpreted way it generates the class file which is then run by the virtual machine. That means the binary file is generated during runtime. When JVM compiles the class file he does not compile the full class file in one shot. Compilation is done on function basis or file basis. Advantage gained from this is that heavy parsing of original source code is avoided. Depending on need basis the compilation is done. This typ of compilation is termed as JIT or Just-in-Time compilation.

103. **Can you explain "finalize()" method?**

For instance if an object is holding some non-Java resource such as a file handle which you might want to make sure are released before the object is destroyed. For such conditions Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. You can add a finalizer to a class by simply defining the "finalize()" method. JVM calls that method whenever it is about to recycle an object of that class. Inside the "finalize()" method you can specify those actions that must be performed before an object is destroyed. GC runs periodically checking for objects that are no longer referenced. Right before an asset is freed, the Java run time calls the "finalize( )" method on the object. Below is the snippet for finalize method

```
protected void finalize()
{
// Your non java resource deletion code goes here

}
```

104. What is a socket? How do you facilitate inter process communication in Java?

A socket is a communication channel, which facilitates **inter-process communication** (For example communicating between two JVMs, which may or may not be running on two different physical machines)**.** A socket is an endpoint for communication. There are two kinds of sockets, depending on whether one wishes to use a connectionless or a connection-oriented protocol. The connectionless communication protocol of the Internet is called UDP. The connection-oriented communication protocol of the Internet is called TCP. UDP sockets are also called datagram sockets. Each socket is uniquely identified on the entire Internet with two numbers.

The lifetime of the socket is made of 3 phases: **Open Socket** -> **Read and Write to Socket** -> **Close Socket**
To make a socket connection you need to know two things: An IP address and port on which to listen/connect. In Java you can use the ***Socket*** (client side*)* and ***ServerSocket*** *(*Server side*)* classes.

105. How would you improve performance of a Java application?

Resource Pooling - Pool valuable system resources like threads, database connections, socket connections etc. Emphasize on reuse of threads from a pool of threads. Creating new threads and discarding them after use can adversely affect performance. Optimize the pool sizes based on system and application specifications and requirements. Having too many threads in a pool also **can result in performance and scalability problems due to consumption of memory stacks**

Make multithreaded environment - Also consider using multi-threading in your single-threaded applications where possible to enhance performance.

Minimize network overheads by retrieving several related items simultaneously in one remote invocation if possible. Remote method invocations involve a network round-trip, marshaling and unmarshaling of parameters, which can cause huge performance problems if the remote interface is poorly designed.

Lazy loading of Data - Most applications need to retrieve data from and save/update data into one or more databases. Database calls are remote calls over the network. In general data should be **lazily loaded** (i.e. load only when required as opposed to pre-loading from the database with a view that it can be used later) from a database to conserve memory but there are use cases (i.e. need to make several database calls) where **eagerly loading** data and caching can improve performance by minimizing network trips to the database.

Caching – If data is frequently used then we need to use some caching techniques.

106. **what is Assertion?**

Assertion includes a Boolean expression. If the Boolean expression evaluates to false AssertionError is thrown. Assertion helps a programmer to debug the code effectively.
Assertion code is removed when the program is deployed. Below is a code snippet which

checks if inventory object is null.

```
Inventory objInv = null;
// ...
// Get your inventory object reference here
// ...
// Use assert to check if the object is null
// by any chance
assert objInv != null;
```

107. If we have multiple static initializer blocks how is the sequence handled?

You can have more than one static initializer block in your class definition, and they can be separated by other code such as method definitions and constructors. The static initializer blocks will be executed in the order in which they appear in the source code.

108. Can you explain in short how JAVA exception handling works?

Basically there are four important keywords which form the main pillars of exception handling: try, catch, throw and finally. Code which you want to monitor for exception is contained in the try block. If any exception occurs in the try block its sent to the catch block which can handle this error in a more rational manner. To throw an exception manually you need to call use the throw keyword. If you want to put any clean up code use the finally block. The finally block is executed irrespective if there is an error or not. Below is a small error code snippet which describes how the flow of error goes.

109. **Can you explain different exception types?**

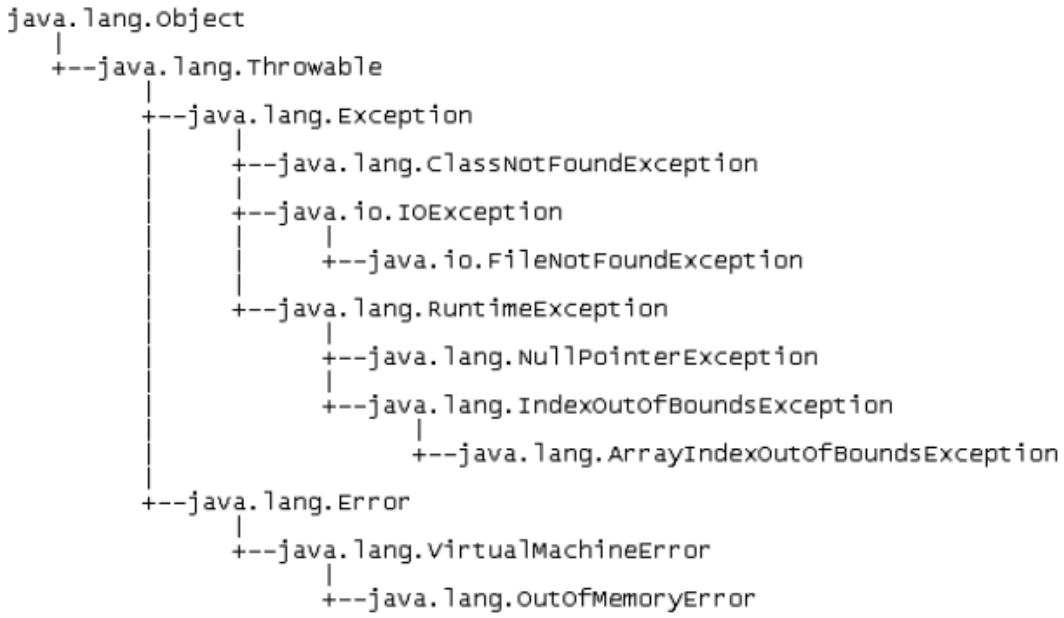


Figure 1.53 : - JAVA exception hierarchy

110. What is Externalizable interface ?

When you use Serializable interface, your class is serialized automatically by default. But with implementing our class with Externizable interface we can override writeObject() and readObject()two methods to control more complex object serailization process.

When you use Externalizable interface, you have a complete control over your class's serialization process. The two methods to be implemented are:

void readExternal(ObjectInput)

void writeExternal(ObjectOutput)

111. What is Auto boxing and unboxing?

This is a newly added feature in J2SE 5.0. Auto boxing is the process in which primitive type is automatically boxed in to equivalent type wrapper. There is no need to explicitly do casting for the same. Auto boxing is vice versa of the same.

Value of the boxed object is converted in to primitive data type. You can see in the below code snippet we have two samples of Auto boxing and Unboxing.

112. **Describe the wrapper classes in Java ?**

Wrapper class is wrapper around a primitive data type. An instance of a wrapper class contains, or wraps, a primitive value of the corresponding type.

Following table lists the primitive types and the corresponding wrapper classes:

Primitive	Wrapper
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

113. **What are the uses of Serialization?**

In some types of applications you have to write the code to serialize objects, but in many cases serialization is performed behind the scenes by various server-side containers.

These are some of the typical uses of serialization:

- To persist data for future use.
- To send data to a remote computer using such client/server Java technologies as RMI or socket programming.
- To "flatten" an object into array of bytes in memory.
- To exchange data between applets and servlets.
- To store user session in Web applications.
- To activate/passivate enterprise java beans.
- To send objects between the servers in a cluster.

114. **How can i tell what state a thread is in ?**

Prior to Java 5, isAlive() was commonly used to test a threads state. If isAlive() returned false the thread was either new or terminated but there was simply no way to differentiate between the two.

Starting with the release of Tiger (Java 5) you can now get what state a thread is in by using the getState() method which returns an Enum of Thread.States. A thread can only be in one of the following states at a given point in time.

NEW	A Fresh thread that has not yet started to execute.
RUNNABLE	A thread that is executing in the Java virtual machine.
BLOCKED	A thread that is blocked waiting for a monitor lock.
WAITING	A thread that is wating to be notified by another thread.
TIMED_WAITING	A thread that is wating to be notified by another thread for a specific amount of time
TERMINATED	A thread whos run method has ended.

115. **What methods java providing for Thread communications ?**

Java provides three methods that threads can use to communicate with each other: wait, notify, and notifyAll. These methods are defined for all Objects (not just Threads). The idea is that a method called by a thread may need to wait for some condition to be satisfied by another thread; in that case, it can call the wait method, which causes its thread to wait until another thread calls notify or notifyAll.

116. **What is the difference between notify and notify All methods?**

A call to notify causes at most one thread waiting on the same object to be notified (i.e., the object that calls notify must be the same as the object that called wait). A call to notifyAll causes all threads waiting on the same object to be notified. If more than one thread is waiting on that object, there is no way to control which of them is notified by a call to notify (so it is often better to use notifyAll than notify).

117. **How do I deserilaize an Object?**

To deserialize an object, perform the following steps:

1. Open the Input stream
2. Chain it with the ObjectInputStream
3. Invoke the readObject() method on ObjectInputStream
4. Typecast into the respective class

```
Java Code

try{
    fIn= new FileInputStream("c:\\emp.ser");
     in = new ObjectInputStream(fIn);

    //de-serializing employee
    Employee emp = (Employee) in.readObject();

    System.out.println("Deserialized " + emp.fName + " "
                + emp.lName + " from emp.ser ");
   }catch(IOException e){
            e.printStackTrace();
   }catch(ClassNotFoundException e){
       e.printStackTrace();  }
```

118. What is enum types?

An *enum type* is a type whose *fields* consist of a fixed set of constants.Because they are constants, the names of an enum type's fields are in uppercase letters.

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

```
switch (day) {
        case MONDAY: System.out.println("Mondays are bad.");
                        break;

        case FRIDAY: System.out.println("Fridays are better.");
                        break;

        case SATURDAY:
        case SUNDAY: System.out.println("Weekends are best.");
                        break;

        default:         System.out.println("Midweek days are so-so.");
                        break;
}
```

```
   public static void main(String[] args) {
           EnumTest firstDay = new EnumTest(Day.MONDAY);
           firstDay.tellItLikeItIs();
           EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
           thirdDay.tellItLikeItIs();
           EnumTest fifthDay = new EnumTest(Day.FRIDAY);
           fifthDay.tellItLikeItIs();
           EnumTest sixthDay = new EnumTest(Day.SATURDAY);
           sixthDay.tellItLikeItIs();
           EnumTest seventhDay = new EnumTest(Day.SUNDAY);
           seventhDay.tellItLikeItIs();


   }
}
```

The output is:

```
Mondays are bad.
Midweek days are so-so.
Fridays are better.
Weekends are best.
Weekends are best.
```

119. Explain JDK 1.5/J2SE 5.0 features

Java 1.5 features

1.Enhanced for loop(for each for loop).
2.Enumeration(enum keyword) - the enum keyword creates a typesafe, ordered list of values (such as day.monday, day.tuesday, etc.). Previously this could only be achieved by non-typesafe constant integers or manually constructed classes (typesafe enum pattern).
3.Assertions added in java 1.5
4.AutoBoxing/Unboxing (like wrapper classes . means automatic convert between primitive to String and vice-versa.) - automatic conversions between primitive types (such as int) and primitive wrapper classes (such as integer).

5.Generics (example: typed Collections, Set(<String>)) - provides compile-time (static) type safety for collections and eliminates the need for most typecasts (type conversion).
6.Varagrs (variable arguments) (example : for printf() function,allows variable number of different arguments) - the last parameter of a method can now be declared using a type name followed by three dots (e.g. Void drawtext(string... Lines)). In the calling code any number of parameters of that type can be used and they are then placed in an array to be passed to the method, or alternatively the calling code can pass an array of that type.
7.StringBuilder class in jdk 1.5 (java.lang package)
8. Metadata - also called annotations; allows language constructs such as classes and methods to be tagged with additional data, which can then be processed by metadata-aware utilities.

120. **Explain Java SE 6 features**

- Support for older win9x versions dropped.
- Scripting lang support: Generic API for integration with scripting languages, & built-in mozilla javascript rhino integration
- Dramatic performance improvements for the core platform, and swing.
- Improved web service support through JAX-WS JDBC 4.0 support
- Java compiler API: an API allowing a java program to select and invoke a java compiler programmatically.
- Upgrade of JAXB to version 2.0: including integration of a stax parser.
- Support for pluggable annotations
- Many GUI improvements, such as integration of swingworker in the API, table sorting and filtering, and true swing double-buffering (eliminating the gray-area effect).

121. **What is JDK and JRE ?**

Java Developer Kit contains tools needed to develop the Java programs, and ***JRE*** to run the programs. The tools include compiler (javac.exe), Java application launcher (java.exe), Appletviewer, etc… Compiler converts java code into byte code. Java application launcher opens a ***JRE***, loads the class, and invokes its main method. You need ***JDK***, if at all you want to write your own programs, and to compile the m. For running java programs, JRE is sufficient. JRE is targeted for execution of Java files

i.e. **JRE** = **JVM** + Java Packages Classes(like util, math, lang, awt,swing etc)+runtime libraries. Java Runtime Environment contains JVM, class libraries, and other supporting files. It does not contain any development tools such as compiler, debugger, etc. Actually JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE. If you want to run any java program, you need to have JRE installed in the system

JDK is mainly targeted for java development. I.e. You can create a Java file (with the help of Java packages), compile a Java file and run a java file