

SECTION ONE

Java – Interview questions & answers

K E Y A R E A S

- Language Fundamentals **LF**
- Design Concepts **DC**
- Design Patterns **DP**
- Concurrency Issues **CI**
- Performance Issues **PI**
- Memory Issues **MI**
- Exception Handling **EH**
- Security **SE**
- Scalability Issues **SI**
- Coding¹ **CO**

FAQ - Frequently Asked Questions

¹ Unlike other key areas, the **CO** is not always shown against the question but shown above the actual content of relevance within a question.

Java – Fundamentals

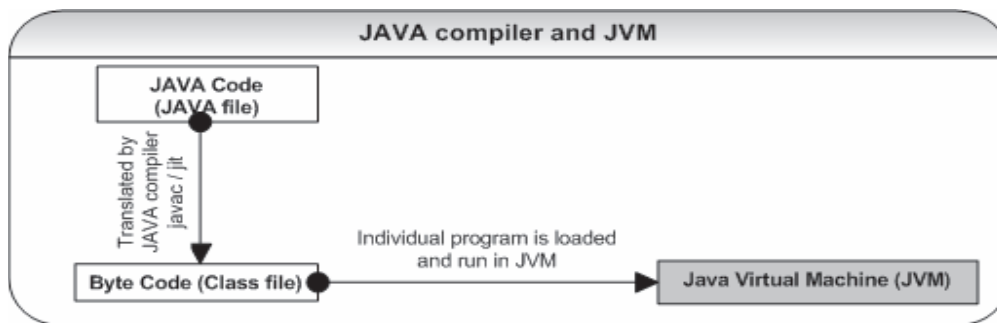
Q 01: Give a few reasons for using Java? [LF](#) [DC](#) [FAQ](#)

A 01: Java is a fun language. Let's look at some of the reasons:

- Built-in support for multi-threading, socket communication, and memory management (automatic garbage collection).
- Object Oriented (OO).
- Better portability than other languages across operating systems.
- Supports Web based applications (Applet, Servlet, and JSP), distributed applications (sockets, RMI, EJB etc) and network protocols (HTTP, JRMP etc) with the help of extensive standardized APIs (Application Programming Interfaces).

Q 02: What is the main difference between the Java platform and the other software platforms? [LF](#)

A 02: 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 a 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.

Q 03: What is the difference between C++ and Java? [LF](#)

A 03: Both C++ and Java use similar syntax and are Object Oriented, but:

- 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**, which allows an object to inherit many method signatures from different interfaces with the condition that the inheriting object must implement those inherited methods. The multiple interface inheritance also allows an object to behave **polymorphically** on those methods. [Refer **Q9** and **Q10** in Java section.]
- 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, which has the finalize() method. This means you do not know when the objects are going to be finalized. **Avoid using finalize() method to release non-memory resources** like file handles, sockets, database connections etc because Java has only a finite number of these resources and you do not know when the garbage collection is going to kick in to release these resources through the finalize() method.
- Java does not include structures or unions because the traditional data structures are implemented as an object oriented framework (Java Collections Framework – Refer **Q16**, **Q17** in Java section).

- All the code in Java program is encapsulated within classes therefore Java does not have global variables or functions.
- C++ requires explicit memory management, while Java includes automatic garbage collection. [Refer **Q37** in Java section].

Q 04: What are the usages of Java packages? **LF**

A 04: 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.

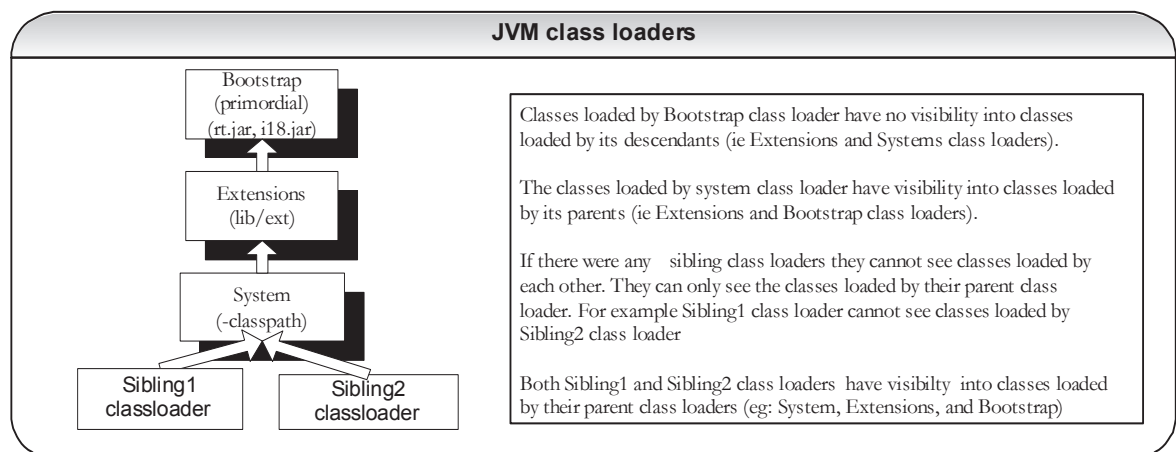
You can create a package as follows with **package** keyword, which is the first keyword in any Java program followed by **import statements**. The `java.lang` package is imported implicitly by default and all the other packages must be explicitly imported.

```
package com.xyz.client ;
import java.io.File;
import java.net.URL;
```

Q 05: Explain Java class loaders? If you have a class in a package, what do you need to do to run it? Explain dynamic class loading? **LF**

A 05: Class loaders are hierarchical. Classes are introduced into the JVM as they are referenced by name in a class that is **already running** in the JVM. So, how is the very first class loaded? The very first class is especially loaded with the help of `static main()` method declared in your class. All the subsequently loaded classes are loaded by the classes, which are already loaded and running. A class loader creates a namespace. All JVMs include at least one class loader that is embedded within the JVM called the primordial (or bootstrap) class loader. Now let's look at non-primordial class loaders. The JVM has hooks in it to allow user defined class loaders to be used in place of primordial class loader. Let us look at the class loaders created by the JVM.

CLASS LOADER	reloadable?	Explanation
Bootstrap (primordial)	No	Loads JDK internal classes, <code>java.*</code> packages. (as defined in the <code>sun.boot.class.path</code> system property, typically loads <code>rt.jar</code> and <code>i18n.jar</code>)
Extensions	No	Loads jar files from JDK extensions directory (as defined in the <code>java.ext.dirs</code> system property – usually <code>lib/ext</code> directory of the JRE)
System	No	Loads classes from system classpath (as defined by the <code>java.class.path</code> property, which is set by the CLASSPATH environment variable or <code>-classpath</code> or <code>-cp</code> command line options)



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.

Q. What do you need to do to run a class with a main() method in a package?

Example: Say, you have a class named "Pet" in a project folder "c:\myProject" and package named com.xyz.client, will you be able to compile and run it as it is?

```
package com.xyz.client;

public class Pet {
    public static void main(String[] args) {
        System.out.println("I am found in the classpath");
    }
}
```

To run → c:\myProject> java com.xyz.client.Pet

The answer is no and you will get the following exception: "Exception in thread "main" java.lang.NoClassDefFoundError: com/xyz/client/Pet". You need to set the classpath. How can you do that? One of the following ways:

1. Set the operating system **CLASSPATH** environment variable to have the project folder "c:\myProject". [Shown in the above diagram as the System –classpath class loader]
2. Set the operating system **CLASSPATH** environment variable to have a jar file "c:/myProject/client.jar", which has the *Pet.class* file in it. [Shown in the above diagram as the System –classpath class loader].
3. Run it with –cp or –classpath option as shown below:

```
c:\>java -cp c:/myProject com.xyz.client.Pet
OR
c:\>java -classpath c:/myProject/client.jar com.xyz.client.Pet
```

Important: Two objects loaded by different class loaders are never equal even if they carry the same values, which mean a class is uniquely identified in the context of the associated class loader. This applies to **singletons** too, where **each class loader will have its own singleton**. [Refer Q51 in Java section for singleton design pattern]

Q. Explain static vs. dynamic class loading?

Static class loading	Dynamic class loading
Classes are statically loaded with Java's "new" operator.	Dynamic loading is a technique for programmatically invoking the functions of a class loader at run time. Let us look at how to load classes dynamically.
<pre>class MyClass { public static void main(String args[]) { Car c = new Car(); } }</pre>	<p>Class.forName (String <i>className</i>); //static method which returns a Class</p> <p>The above static method returns the class object associated with the class name. The string <i>className</i> can be supplied dynamically at run time. Unlike the static loading, the dynamic loading will decide whether to load the class <i>Car</i> or the class <i>Jeep</i> 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.</p> <p>class.newInstance (); //A non-static method, which creates an instance of a //class (i.e. creates an object).</p> <pre>Jeep myJeep = null ; //myClassName should be read from a .properties file or a Constants class. // stay away from hard coding values in your program. CO String myClassName = "au.com.Jeep" ; Class vehicleClass = Class.forName(myClassName) ; myJeep = (Jeep) vehicleClass.newInstance(); myJeep.setFuelCapacity(50);</pre>
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.	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: <ul style="list-style-type: none"> ▪ The forName(..) method in class - Class. ▪ The findSystemClass(..) method in class - ClassLoader. ▪ The loadClass(..) method in class - ClassLoader.

Q. 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. **CO**

```
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
}
```

The following code gives an **Output of** A=5, B=10.

```
public class Test {
    System.out.println("A =" + StaticInitializer.A + ", B =" + StaticInitializer.B);
}
```

Q 06: What is the difference between constructors and other regular methods? What happens if you do not provide a constructor? Can you call one constructor from another? How do you call the superclass's constructor? **LF FAQ**

A 06:

Constructors	Regular methods
Constructors must have the <u>same name as the class name</u> and <u>cannot return a value</u> . The constructors are called only once per creation of an object while regular methods can be called many times. E.g. for a Pet.class	Regular methods can have any name and can be called any number of times. E.g. for a Pet.class.
<code>public Pet() {} // constructor</code>	<code>public void Pet(){} // regular method has a void return type.</code>
	Note: method name is shown starting with an uppercase to differentiate a constructor from a regular method. Better naming convention is to have a meaningful name starting with a lowercase like:
	<code>public void createPet(){} // regular method has a void return type</code>

Q. What happens if you do not provide a constructor? Java does not actually require an explicit constructor in the class description. If you do not include a constructor, the Java compiler will create a default constructor in the byte code with an empty argument. This default constructor is equivalent to the explicit “Pet(){}”. If a class includes one or more explicit constructors like “public Pet(int id)” or “Pet(){}” etc, the java compiler does *not* create the default constructor “Pet(){}”.

Q. Can you call one constructor from another? Yes, by using **this()** syntax. E.g.

```
public Pet(int id) {
    this.id = id; // "this" means this object
}
public Pet (int id, String type) {
    this(id); // calls constructor public Pet(int id)
    this.type = type; // "this" means this object
}
```

Q. How to call the superclass constructor? If a class called “*SpecialPet*” extends your “*Pet*” class then you can use the keyword “**super**” to invoke the superclass's constructor. E.g.

```
public SpecialPet(int id) {
    super(id); //must be the very first statement in the constructor.
}
```

To call a regular method in the super class use: “**super.myMethod();**”. This can be called at any line. Some frameworks based on JUnit add their own initialization code, and not only do they need to remember to invoke

their parent's `setUp()` method, you, as a user, need to remember to invoke theirs after you wrote your initialization code:

```
public class DBUnitTestCase extends TestCase {
    public void setUp() {
        super.setUp();
        // do my own initialization
    }
}

public void cleanUp() throws Throwable
{
    try {
        ... // Do stuff here to clean up your object(s).
    }
    catch (Throwable t) {}
    finally{
        super.cleanUp(); //clean up your parent class. Unlike constructors
                        // super.regularMethod() can be called at any line.
    }
}
```

Q 07: What are the advantages of Object Oriented Programming Languages (OOPL)? **DC FAQ**

A 07: The Object Oriented Programming Languages directly represent the real life objects like *Car*, *Jeep*, *Account*, *Customer* etc. The features of the OO programming languages like **polymorphism**, **inheritance** and **encapsulation** make it powerful. **[Tip: remember pie which, stands for Polymorphism, Inheritance and Encapsulation are the 3 pillars of OOPL]**

Q 08: How does the Object Oriented approach improve software development? **DC**

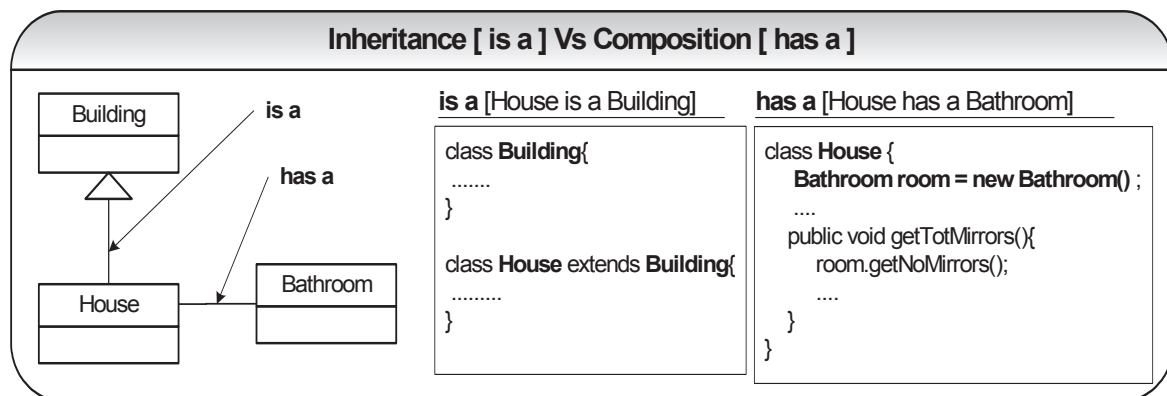
A 08: The key benefits are:

- **Re-use** of previous work: using **implementation inheritance** and **object composition**.
- **Real mapping to the problem domain:** Objects map to real world and represent vehicles, customers, products etc: with **encapsulation**.
- **Modular Architecture:** Objects, systems, frameworks etc are the building blocks of larger systems.

The **increased quality** and **reduced development time** are the by-products of the key benefits discussed above. If 90% of the new application consists of proven existing components then only the remaining 10% of the code have to be tested from scratch.

Q 09: How do you express an '**is a**' relationship and a '**has a**' relationship or explain inheritance and composition? What is the difference between composition and aggregation? **DC FAQ**

A 09: The '**is a**' relationship is expressed with **inheritance** and '**has a**' relationship is expressed with **composition**. Both inheritance and composition allow you to place sub-objects inside your new class. Two of the main techniques for **code reuse** are **class inheritance** and **object composition**.



Inheritance is uni-directional. For example *House is a Building*. But *Building* is not a *House*. Inheritance uses **extends** key word. **Composition:** is used when *House has a Bathroom*. It is incorrect to say *House is a*

Bathroom. Composition simply means using instance variables that refer to other objects. The class *House* will have an instance variable, which refers to a *Bathroom* object.

Q. Which one to favor, composition or inheritance? The guide is that inheritance should be only used when subclass 'is a' superclass.

- Don't use inheritance just to get code reuse. If there is no 'is a' relationship then use composition for code reuse. Overuse of **implementation inheritance** (uses the "extends" key word) can break all the subclasses, if the superclass is modified.
- Do not use inheritance just to get polymorphism. If there is no 'is a' relationship and all you want is **polymorphism** then use **interface inheritance** with **composition**, which gives you **code reuse** (Refer Q10 in Java section for interface inheritance).

What is the difference between aggregation and composition?

Aggregation	Composition
Aggregation is an association in which one class belongs to a collection. This is a part of a whole relationship where a part can exist without a whole. For example a line item is a whole and product is a part. If a line item is deleted then corresponding product need not be deleted. So aggregation has a weaker relationship .	Composition is an association in which one class belongs to a collection. This is a part of a whole relationship where a part cannot exist without a whole. If a whole is deleted then all parts are deleted. For example An order is a whole and line items are parts. If an order is deleted then all corresponding line items for that order should be deleted. So composition has a stronger relationship .

Q 10: What do you mean by polymorphism, inheritance, encapsulation, and dynamic binding? **DC SE FAQ**

A 10: **Polymorphism** – means the ability of a single variable of a given type to be used to reference objects of different types, and automatically call the method that is specific to the type of object the variable references. In a nutshell, polymorphism is a bottom-up method call. The benefit of polymorphism is that it is **very easy to add new classes of derived objects without breaking the calling code** (i.e. `getTotArea()` in the sample code shown below) that uses the polymorphic classes or interfaces. When you send a message to an object even though you don't know what specific type it is, and the right thing happens, that's called **polymorphism**. The process used by object-oriented programming languages to implement polymorphism is called **dynamic binding**. Let us look at some sample code to demonstrate polymorphism: **CO**

Sample code:

```
//client or calling code
double dim = 5.0; //ie 5 meters radius or width
List listShapes = new ArrayList(20);

Shape s = new Circle();
listShapes.add(s); //add circle

s = new Square();
listShapes.add(s); //add square

getTotArea (listShapes,dim); //returns 78.5+25.0=103.5

//Later on, if you decide to add a half circle then define
//a HalfCircle class, which extends Circle and then provide an
//area(). method but your called method getTotArea(...) remains
//same.

s = new HalfCircle();
listShapes.add(s); //add HalfCircle

getTotArea (listShapes,dim); //returns 78.5+25.0+39.25=142.75

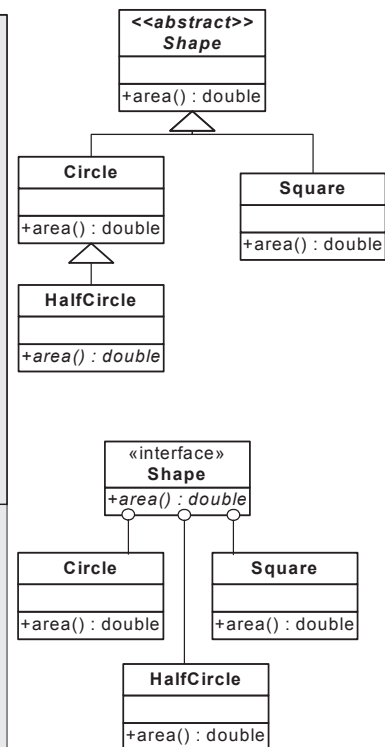
/** called method: method which adds up areas of various
** shapes supplied to it.
**/
public double getTotArea(List listShapes, double dim){
    Iterator it = listShapes.iterator();
    double totalArea = 0.0;
    //loop through different shapes
    while(it.hasNext()) {
        Shape s = (Shape) it.next();
        totalArea += s.area(dim); //polymorphic method call
    }
    return totalArea ;
}
```

For example: given a base class/interface *Shape*, polymorphism allows the programmer to define different `area(double dim)` methods for any number of derived classes such as *Circle*, *Square* etc. No matter what shape an object is, applying the area method to it will return the right results.

Later on *HalfCircle* can be added without breaking your called code i.e. method `getTotalArea(...)`

Depending on what the shape is, appropriate `area(double dim)` method gets called and calculated.

Circle → area is 78.5sqm
Square → area is 25sqm
HalfCircle → area is 39.25 sqm



Inheritance – is the inclusion of behavior (i.e. methods) and state (i.e. variables) of a base class in a derived class so that they are accessible in that derived class. The key benefit of Inheritance is that it provides the formal mechanism for **code reuse**. Any shared piece of business logic can be moved from the derived class into the base class as part of refactoring process to improve maintainability of your code by avoiding code duplication. The existing class is called the *superclass* and the derived class is called the *subclass*. **Inheritance** can also be defined as the process whereby one object acquires characteristics from one or more other objects the same way children acquire characteristics from their parents. **There are two types of inheritances:**

1. Implementation inheritance (aka class inheritance): You can extend an application's functionality by reusing functionality in the parent class by inheriting all or some of the operations already implemented. In Java, you can only inherit from one superclass. Implementation inheritance promotes reusability but improper use of class inheritance can cause programming nightmares by breaking encapsulation and making future changes a problem. With implementation inheritance, the subclass becomes tightly coupled with the superclass. This will make the design fragile because if you want to change the superclass, you must know all the details of the subclasses to avoid breaking them. So when using implementation inheritance, **make sure that the subclasses depend only on the behavior of the superclass, not on the actual implementation**. For example in the above diagram, the subclasses should only be concerned about the behavior known as `area()` but not how it is implemented.

2. Interface inheritance (aka type inheritance): This is also known as subtyping. Interfaces provide a mechanism for specifying a relationship between otherwise unrelated classes, typically by specifying a set of common methods each implementing class must contain. Interface inheritance promotes the design concept of **program to interfaces not to implementations**. This also reduces the coupling or implementation dependencies between systems. In Java, you can implement any number of interfaces. This is more flexible than implementation inheritance because it won't lock you into specific implementations which make subclasses difficult to maintain. So care should be taken not to break the implementing classes by modifying the interfaces.

Which one to use? Prefer interface inheritance to implementation inheritance because it promotes the design concept of **coding to an interface and reduces coupling**. Interface inheritance can achieve **code reuse** with the help of **object composition**. If you look at Gang of Four (GoF) design patterns, you can see that it favors interface inheritance to implementation inheritance. **CO**

Implementation inheritance	Interface inheritance with composition
<p>Let's assume that savings account and term deposit account have a similar behavior in terms of depositing and withdrawing money, so we will get the super class to implement this behavior and get the subclasses to reuse this behavior. But saving account and term deposit account have specific behavior in calculating the interest.</p> <p>Super class <i>Account</i> has reusable code as methods deposit (double amount) and withdraw (double amount).</p> <pre>public abstract class Account { public void deposit (double amount) { System.out.println("depositing " + amount); } public void withdraw (double amount) { System.out.println("withdrawing " + amount); } public abstract double calculateInterest(double amount); }</pre> <pre>public class SavingsAccount extends Account { public double calculateInterest (double amount) { // calculate interest for SavingsAccount return amount * 0.03; } public void deposit (double amount) { super.deposit (amount); // get code reuse // do something else } public void withdraw (double amount) {</pre>	<p>Let's look at an interface inheritance code sample, which makes use of composition for reusability. In the following example the methods <code>deposit(...)</code> and <code>withdraw(...)</code> share the same piece of code in <i>AccountHelper</i> class. The method <code>calculateInterest(...)</code> has its specific implementation in its own class.</p> <pre>public interface Account { public abstract double calculateInterest(double amount); public abstract void deposit(double amount); public abstract void withdraw(double amount); }</pre> <p>Code to interface so that the implementation can change.</p> <pre>public interface AccountHelper { public abstract void deposit (double amount); public abstract void withdraw (double amount); }</pre> <p>class <i>AccountHelperImpl</i> has reusable code as methods deposit (double amount) and withdraw (double amount).</p> <pre>public class AccountHelperImpl implements AccountHelper { public void deposit(double amount) { System.out.println("depositing " + amount); } public void withdraw(double amount) { System.out.println("withdrawing " + amount); } }</pre> <pre>public class SavingsAccountImpl implements Account {</pre>

<pre> super.withdraw (amount); // get code reuse // do something else } } public class TermDepositAccount extends Account { public double calculateInterest (double amount) { // calculate interest for SavingsAccount return amount * 0.05; } public void deposit(double amount) { super.deposit (amount); // get code reuse // do something else } public void withdraw(double amount) { super.withdraw (amount); // get code reuse // do something else } } </pre>	<pre> // composed helper class (i.e. composition). AccountHelper helper = new AccountHelperImpl (); public double calculateInterest (double amount) { // calculate interest for SavingsAccount return amount * 0.03; } public void deposit (double amount) { helper.deposit(amount); // code reuse via composition } public void withdraw (double amount) { helper.withdraw (amount); // code reuse via composition } } public class TermDepositAccountImpl implements Account { // composed helper class (i.e. composition). AccountHelper helper = new AccountHelperImpl (); public double calculateInterest (double amount) { //calculate interest for SavingsAccount return amount * 0.05; } public void deposit (double amount) { helper.deposit (amount) ; // code reuse via composition } public void withdraw (double amount) { helper.withdraw (amount) ; // code reuse via composition } } </pre>
<p>The Test class:</p> <pre> public class Test { public static void main(String[] args) { Account acc1 = new SavingsAccountImpl(); acc1.deposit(50.0); Account acc2 = new TermDepositAccountImpl(); acc2.deposit(25.0); acc1.withdraw(25); acc2.withdraw(10); double cal1 = acc1.calculateInterest(100.0); double cal2 = acc2.calculateInterest(100.0); System.out.println("Savings --> " + cal1); System.out.println("TermDeposit --> " + cal2); } } </pre> <p>The output:</p> <pre> depositing 50.0 depositing 25.0 withdrawing 25.0 withdrawing 10.0 Savings --> 3.0 TermDeposit --> 5.0 </pre>	

Q. 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. **For example:** the composed class **AccountHelperImpl** can be replaced by another more efficient implementation as shown below if required:

```
public class EfficientAccountHelperImpl implements AccountHelper {
    public void deposit(double amount) {
        System.out.println("efficient depositing " + amount);
    }

    public void withdraw(double amount) {
        System.out.println("efficient withdrawing " + amount);
    }
}
```

- 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 superclass 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.

Due to the flexibility and power of object composition, **most design patterns emphasize object composition over inheritance whenever it is possible**. Many times, a design pattern shows a clever way of solving a common problem through the use of object composition rather than a standard, less flexible, inheritance based solution.

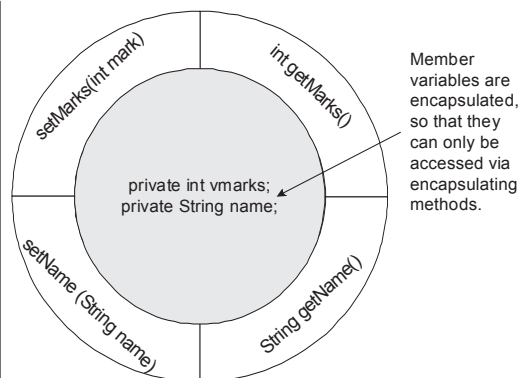
Encapsulation – refers to keeping all the related members (variables and methods) together in an object. Specifying member variables as private can hide the variables and methods. Objects should hide their inner workings from the outside view. Good **encapsulation improves code modularity by preventing objects interacting with each other in an unexpected way**, which in turn makes future development and refactoring efforts easy. **CO**

Sample code

```
Class MyMarks {
    private int vmarks = 0;
    private String name;

    public void setMarks(int mark)
        throws MarkException {
        if(mark > 0)
            this.vmarks = mark;
        else {
            throw new MarkException("No negative
                Values");
        }
    }

    public int getMarks(){
        return vmarks;
    }
    //getters and setters for attribute name goes here.
}
```



Being able to encapsulate members of a class is important for **security** and **integrity**. We can protect variables from unacceptable values. The sample code above describes how encapsulation can be used to protect the *MyMarks* object from having negative values. Any modification to member variable "vmarks" can only be carried out through the setter method *setMarks(int mark)*. This prevents the object "MyMarks" from having any negative values by throwing an exception.

Q 11: What is design by contract? Explain the *assertion* construct? **DC**

A 11: Design by contract specifies the obligations of a calling-method and called-method to each other. Design by contract is a valuable technique, which should be used to build well-defined interfaces. The strength of this programming methodology is that it gets the programmer to **think clearly about what a function does**, what pre and post conditions it must adhere to and also it **provides documentation for the caller**. Java uses the **assert** statement to implement pre- and post-conditions. Java's exceptions handling also support design by contract especially **checked exceptions** (Refer **Q39** in Java section for checked exceptions). In design by contract in addition to specifying programming code to carrying out intended operations of a method the programmer also specifies:

1. Preconditions – This is the part of the contract the **calling-method must agree to**. Preconditions specify the conditions that must be true before a called method can execute. Preconditions involve the system state and the arguments passed into the method at the time of its invocation. **If a precondition fails then there is a bug in the calling-method or calling software component.**

On public methods	On non-public methods
<p>Preconditions on <i>public</i> methods are enforced by explicit checks that throw particular, specified exceptions. You should not use <code>assert</code> to check the parameters of the public methods but can use for the non-public methods. <code>Assert</code> is inappropriate because the method guarantees that it will always enforce the argument checks. It must check its arguments whether or not assertions are enabled. Further, <code>assert</code> construct does not throw an exception of a specified type. It can throw only an <i>AssertionError</i>.</p> <pre>public void setRate(int rate) { if(rate <= 0 rate > MAX_RATE){ throw new IllegalArgumentException("Invalid rate -> " + rate); } setCalculatedRate(rate); }</pre>	<p>You can use assertion to check the parameters of the non-public methods.</p> <pre>private void setCalculatedRate(int rate) { assert (rate > 0 && rate < MAX_RATE) : rate; //calculate the rate and set it. }</pre> <p>Assertions can be disabled, so programs must not assume that <code>assert</code> construct will be always executed:</p> <p>//Wrong: //if assertion is disabled, "pilotJob" never gets removed assert jobsAd.remove(pilotJob);</p> <p>//Correct: boolean pilotJobRemoved = jobsAd.remove(pilotJob); assert pilotJobRemoved;</p>

2. Postconditions – This is the part of the contract the **called-method agrees to**. What must be true after a method completes successfully. Postconditions can be used with assertions in both public and non-public methods. The postconditions involve the old system state, the new system state, the method arguments and the method's return value. **If a postcondition fails then there is a bug in the called-method or called software component.**

```
public double calcRate(int rate) {
    if(rate <= 0 || rate > MAX_RATE){
        throw new IllegalArgumentException("Invalid rate !!! ");
    }

    //logic to calculate the rate and set it goes here

    assert this.evaluate(result) < 0 : this; //message sent to AssertionError on failure
    return result;
}
```

3. Class invariants - what must be true about each instance of a class? A class invariant as an internal invariant that can specify the relationships among multiple attributes, and should be true before and after any method completes. **If an invariant fails then there could be a bug in either calling-method or called-method.** There is no particular mechanism for checking invariants but it is convenient to combine all the expressions required for checking invariants into a single internal method that can be called by assertions. For example if you have a class, which deals with negative integers then you define the **`isNegative()`** convenient internal method:

```
class NegativeInteger {
    Integer value = new Integer (-1); //invariant

    //constructor
    public NegativeInteger(Integer int) {
        //constructor logic goes here
        assert isNegative();
    }

    // rest of the public and non-public methods goes here. public methods should call
    // assert isNegative(); prior to its return

    // convenient internal method for checking invariants.
    // Returns true if the integer value is negative

    private boolean isNegative(){
        return value.intValue() < 0 ;
    }
}
```

The `isNegative()` method should be true before and after any method completes, each public method and constructor should contain the following assert statement immediately prior to its return.

```
assert isNegative();
```

Explain the assertion construct? The assertion statements have two forms as shown below:

```
assert Expression1;
assert Expression1 : Expression2;
```

Where:

- **Expression1** → is a boolean expression. If the *Expression1* evaluates to false, it throws an *AssertionError* without any detailed message.
- **Expression2** → if the *Expression1* evaluates to false throws an *AssertionError* with using the value of the *Expression2* as the error's detailed message.

Note: If you are using assertions (available from JDK1.4 onwards), you should supply the JVM argument to enable it by package name or class name.

```
java -ea[:packagename...[:classname]] or java -enableassertions[:packagename...[:classname]]
java -ea:Account
```

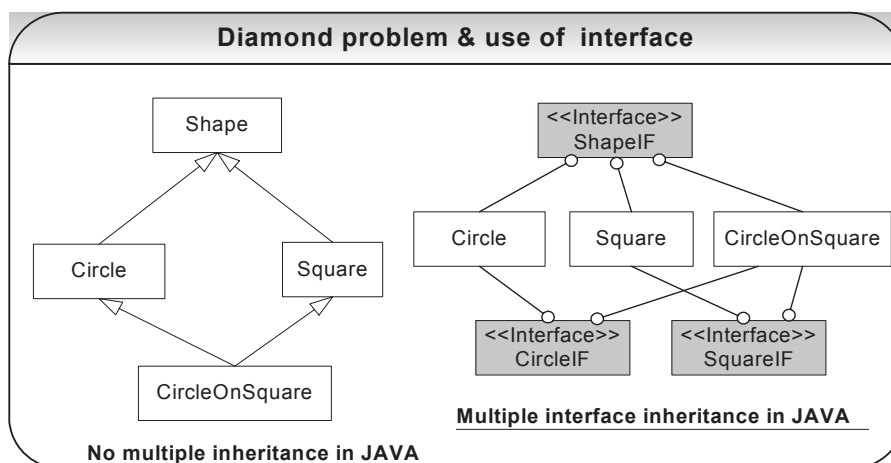
Q 12: What is the difference between an abstract class and an interface and when should you use them? **LF DP DC**
FAQ

A 12: In design, you want the base class to present *only* an interface for its derived classes. This means, you don't want anyone to actually instantiate an object of the base class. You only **want to upcast to it** (implicit upcasting, which gives you polymorphic behavior), so that its interface can be used. This is accomplished by making that class *abstract* using the **abstract** keyword. If anyone tries to make an object of an **abstract** class, the compiler prevents it.

The **interface** keyword takes this concept of an **abstract** class a step further by preventing any method or function implementation at all. You can only declare a method or function but not provide the implementation. The class, which is implementing the interface, should provide the actual implementation. The **interface** is a very useful and commonly used aspect in OO design, as it provides the **separation of interface and implementation** and enables you to:

- Capture similarities among unrelated classes without artificially forcing a class relationship.
- Declare methods that one or more classes are expected to implement.
- Reveal an object's programming interface without revealing its actual implementation.
- Model multiple interface inheritance in Java, which provides some of the benefits of full on multiple inheritances, a feature that some object-oriented languages support that allow a class to have more than one superclass.

Abstract class	Interface
Have executable methods and abstract methods.	Have no implementation code. All methods are abstract.
Can only subclass one abstract class.	A class can implement any number of interfaces.



Q. When to use an abstract class?: In case where you want to use **implementation inheritance** then it is usually provided by an abstract base class. Abstract classes are excellent candidates inside of application frameworks. Abstract classes let you define some default behavior and force subclasses to provide any specific behavior. Care should be taken not to overuse implementation inheritance as discussed in **Q10** in Java section.

Q. When to use an interface?: For polymorphic interface inheritance, where the client wants to only deal with a type and does not care about the actual implementation use interfaces. If you need to change your design frequently, you should prefer using interface to abstract. **CO** Coding to an interface **reduces coupling** and interface inheritance can achieve **code reuse** with the help of **object composition**. **For example:** The Spring framework's dependency injection promotes code to an interface principle. Another justification for using interfaces is that they solve the '**diamond problem**' of traditional multiple inheritance as shown in the figure. Java does not support multiple inheritance. Java only supports **multiple interface inheritance**. Interface will solve all the ambiguities caused by this 'diamond problem'.

Design pattern: Strategy design pattern lets you swap new algorithms and processes into your program without altering the objects that use them. **Strategy design pattern:** Refer **Q11** in How would you go about... section.

Q 13: Why there are some interfaces with no defined methods (i.e. marker interfaces) in Java? **LF FAQ**

A 13: The interfaces with no defined methods act like markers. They just tell the compiler that the objects of the classes implementing the interfaces with no defined methods need to be treated differently. **Example** java.io.Serializable (Refer **Q23** in Java section), java.lang.Cloneable, java.util.EventListener etc. Marker interfaces are also known as "tag" interfaces since they tag all the derived classes into a category based on their purpose.

Q 14: When is a method said to be overloaded and when is a method said to be overridden? **LF CO FAQ**

A 14:

Method Overloading	Method Overriding
Overloading deals with multiple methods in the same class with the same name but different method signatures.	Overriding deals with two methods, one in the parent class and the other one in the child class and has the same name and signatures.
<pre>class MyClass { public void getInvestAmount(int rate) {...} public void getInvestAmount(int rate, long principal) { ... } }</pre>	<pre>class BaseClass{ public void getInvestAmount(int rate) {...} } class MyClass extends BaseClass { public void getInvestAmount(int rate) { ...} }</pre>
Both the above methods have the same method names but different method signatures, which mean the methods are overloaded.	Both the above methods have the same method names and the signatures but the method in the subclass <i>MyClass</i> overrides the method in the superclass <i>BaseClass</i> .
Overloading lets you define the same operation in different ways for different data .	Overriding lets you define the same operation in different ways for different object types .

Q 15: What is the main difference between an ArrayList and a Vector? What is the main difference between HashMap and Hashtable? What is the difference between a stack and a queue? **LF DC PI CI FAQ**

A 15:

Vector / Hashtable	ArrayList / HashMap
Original classes before the introduction of Collections API. <i>Vector</i> & <i>Hashtable</i> are synchronized. Any method that touches their contents is thread-safe.	So if you don't need a thread safe collection, use the <i>ArrayList</i> or <i>HashMap</i> . Why pay the price of synchronization unnecessarily at the expense of performance degradation.

Q. So which is better? As a general rule, prefer *ArrayList/HashMap* to *Vector/Hashtable*. If your application is a multithreaded application and **at least one of the threads either adds or deletes an entry into the collection** then use new Java *collections* API's external synchronization facility as shown below to **temporarily synchronize** your collections as needed: **CO**

```
Map myMap = Collections.synchronizedMap (myMap);           // single lock for the entire map
List myList = Collections.synchronizedList (myList);       // single lock for the entire list
```

J2SE 5.0: If you are using J2SE5, you should use the new “*java.util.concurrent*” package for improved performance because the concurrent package collections are not governed by a single synchronized lock as shown above. The “*java.util.concurrent*” package collections like **ConcurrentHashMap** is threadsafe and at the same time safely permits any number of concurrent reads as well as tunable number of concurrent writes. The “*java.util.concurrent*” package also provides an efficient scalable thread-safe non-blocking FIFO queue like **ConcurrentLinkedQueue**.

J2SE 5.0: The “*java.util.concurrent*” package also has classes like **CopyOnWriteArrayList**, **CopyOnWriteArraySet**, which gives you thread safety with the added benefit of immutability to deal with data that changes infrequently. The **CopyOnWriteArrayList** behaves much like the **ArrayList** class, except that when the list is modified, instead of modifying the underlying array, a new array is created and the old array is discarded. This means that when a caller gets an iterator (i.e. `copyOnWriteArrayListRef.iterator()`), which internally holds a reference to the underlying **CopyOnWriteArrayList** object's array, which is immutable and therefore can be used for traversal without requiring either synchronization on the list `copyOnWriteArrayListRef` or need to `clone()` the `copyOnWriteArrayListRef` list before traversal (i.e. there is no risk of concurrent modification) and also offers better performance.

Array	List / Stack etc
Java arrays are even faster than using an <i>ArrayList/Vector</i> and perhaps therefore may be preferable if you know the size of your array upfront (because arrays cannot grow as Lists do).	<i>ArrayList/Vector</i> are specialized data structures that internally uses an array with some convenient methods like <code>add(..)</code> , <code>remove(...)</code> etc so that they can grow and shrink from their initial size. <i>ArrayList</i> also supports index based searches with <code>indexOf(Object obj)</code> and <code>lastIndexOf(Object obj)</code> methods.
In an array, any item can be accessed.	These are more abstract than arrays and access is restricted. For example, a stack allows access to only last item inserted.

Queue<E> (added in J2SE 5.0)	Stack
First item to be inserted is the first one to be removed.	Allows access to only last item inserted.
This mechanism is called First In First Out (FIFO).	An item is inserted or removed from one end called the “top” of the stack. This is called Last In First Out (LIFO) mechanism.
Placing an item in the queue is called “enqueue or insertion” and removing an item from a queue is called “dequeue or deletion”. Pre J2SE 5.0, you should write your own <i>Queue</i> class with <code>enqueue()</code> and <code>dequeue()</code> methods using an <i>ArrayList</i> or a <i>LinkedList</i> class.	Placing the data at the top is called “pushing” and removing an item from the top is called “popping”. If you want to reverse “XYZ” → ZYX, then you can use a java.util.Stack
J2SE 5.0 has a java.util.Queue<E> interface.	

Q 16: Explain the Java Collections Framework? **LF DP FAQ**

A 16: The key interfaces used by the collections framework are **List**, **Set** and **Map**. The **List** and **Set** extends the **Collection** interface. Should not confuse the **Collection** interface with the **Collections** class which is a utility class.

Set (HashSet, TreeSet)	List (ArrayList, LinkedList, Vector etc)
A Set is a collection with <u>unique elements</u> and prevents duplication within the collection. HashSet and TreeSet are implementations of a Set interface. A TreeSet is an ordered HashSet , which implements the SortedSet interface.	A List is a collection with an <u>ordered sequence of elements</u> and <u>may contain duplicates</u> . ArrayList , LinkedList and Vector are implementations of a List interface. (i.e. an index based)

The Collections API also supports maps, but within a hierarchy distinct from the **Collection** interface. A **Map** is an object that maps keys to values, where the list of keys is itself a collection object. A map can contain duplicate values, but the keys in a map must be distinct. **HashMap**, **TreeMap** and **Hashtable** are implementations of a **Map** interface. A **TreeMap** is an ordered **HashMap**, which implements the **SortedMap** interface.

Q. How to implement collection ordering? **SortedSet** and **SortedMap** interfaces maintain sorted order. The classes, which implement the **Comparable** interface, impose natural order. By implementing **Comparable**, sorting an array of objects or a collection (List etc) is as simple as:

```
Arrays.sort(myArray);
Collections.sort(myCollection); // do not confuse "Collections" utility class with the
                                // "Collection" interface without an "s".
```


For classes that don't implement *Comparable* interface, or when one needs even more control over ordering based on multiple attributes, a **Comparator** interface should be used.

Comparable interface	Comparator interface
<p>The "Comparable" allows itself to compare with another similar object (i.e. A class that implements <i>Comparable</i> becomes an object to be compared with). The method compareTo() is specified in the interface.</p>	<p>The Comparator is used to compare two different objects. The following method is specified in the Comparator interface.</p> <pre>public int compare(Object o1, Object o2)</pre>
<p>Many of the standard classes in the Java library like String, Integer, Date, File etc implement the Comparable interface to give the class a "Natural Ordering". For example String class uses the following methods:</p> <pre>public int compareTo(o) public int compareToIgnoreCase(str)</pre> <p>You could also implement your own method in your own class as shown below:</p> <pre>...imports public class Pet implements Comparable { int petId; String petType; public Pet(int argPetId, String argPetType) { petId = argPetId; this.petType = argPetType; } public int compareTo(Object o) { Pet petAnother = (Pet)o; //natural alphabetical ordering by type //if equal returns 0, if greater returns +ve int, //if less returns -ve int return this.petType.compareTo(petAnother.petType); } public static void main(String[] args) { List list = new ArrayList(); list.add(new Pet(2, "Dog")); list.add(new Pet(2, "Dog")); list.add(new Pet(1, "Parrot")); list.add(new Pet(2, "Cat")); Collections.sort(list); // sorts using compareTo method for (Iterator iter = list.iterator(); iter.hasNext();) { Pet element = (Pet) iter.next(); System.out.println(element); } public String toString() { return petType; } } }</pre> <p>Output: Cat, Dog, Parrot</p>	<p>You can have more control by writing your Comparator class. Let us write a Comparator for the Pet class shown on the left. For most cases natural ordering is fine as shown on the left but say we require a special scenario where we need to first sort by the "petId" and then by the "petType". We can achieve this by writing a "Comparator" class.</p> <pre>...imports public class PetComparator implements Comparator, Serializable{ public int compare(Object o1, Object o2) { int result = 0; Pet pet = (Pet)o1; Pet petAnother = (Pet)o2; //use Integer class's natural ordering Integer pId = new Integer(pet.getPetId()); Integer pAnotherId = new Integer(petAnother.getPetId()); result = pId.compareTo(pAnotherId); //if ids are same compare by petType if(result == 0) { result= pet.getPetType().compareTo (petAnother.getPetType()); } return result; } public static void main(String[] args) { List list = new ArrayList(); list.add(new Pet(2, "Dog")); list.add(new Pet(1, "Parrot")); list.add(new Pet(2, "Cat")); Collections.sort(list, new PetComparator()); for (Iterator iter = list.iterator(); iter.hasNext();){ Pet element = (Pet) iter.next(); System.out.println(element); } } }</pre> <p>Output: Parrot, Cat, Dog.</p> <p>Note: some methods are not shown for brevity.</p>

Important: The ordering imposed by a java.util.Comparator "myComp" on a set of elements "mySet" should be consistent with **equals()** method, which means **for example:**

```
if compare(o1,o2) == 0 then o1.equals(o2) should be true.
if compare(o1,o2) != 0 then o1.equals(o2) should be false.
```

If a comparator "myComp" on a set of elements "mySet" is inconsistent with **equals()** method, then SortedSet or SortedMap will behave strangely and is hard to debug. **For example** if you add two objects o1, o2 to a **TreeSet**

(implements **SortedSet**) such that `o1.equals(o2) == true` and `compare(o1,o2) != 0` the second add operation will return false and will not be added to your set because `o1` and `o2` are equivalent from the `TreeSet`'s perspective. **TIP:** It is always a good practice and highly recommended to keep the Java API documentation handy and refer to it as required while coding. Please refer to **java.util.Comparator** interface API for further details.

Design pattern: **Q. What is an Iterator?** An Iterator is a use once object to access the objects stored in a collection. **Iterator design pattern** (aka Cursor) is used, which is a behavioral design pattern that provides a way to access elements of a collection sequentially without exposing its internal representation.

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

Problem: 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.

```
Collection<String> myCollection = new ArrayList<String>(10);

myCollection.add("123");
myCollection.add("456");
myCollection.add("789");

for (Iterator it = myCollection.iterator(); it.hasNext();) {
    String myObject = (String)it.next();
    System.out.println(myObject);
    if (someConditionIsTrue) {
        myCollection.remove(myObject); //can throw ConcurrentModificationException in single as
                                        //well as multi-thread access situations.
    }
}
```

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.

Solution 4: for single-thread access situation:

Use:

```
it.remove(); // removes the current object via the Iterator "it" which has a reference to
              // your underlying collection "myCollection". Also can use solutions 1-3.
```

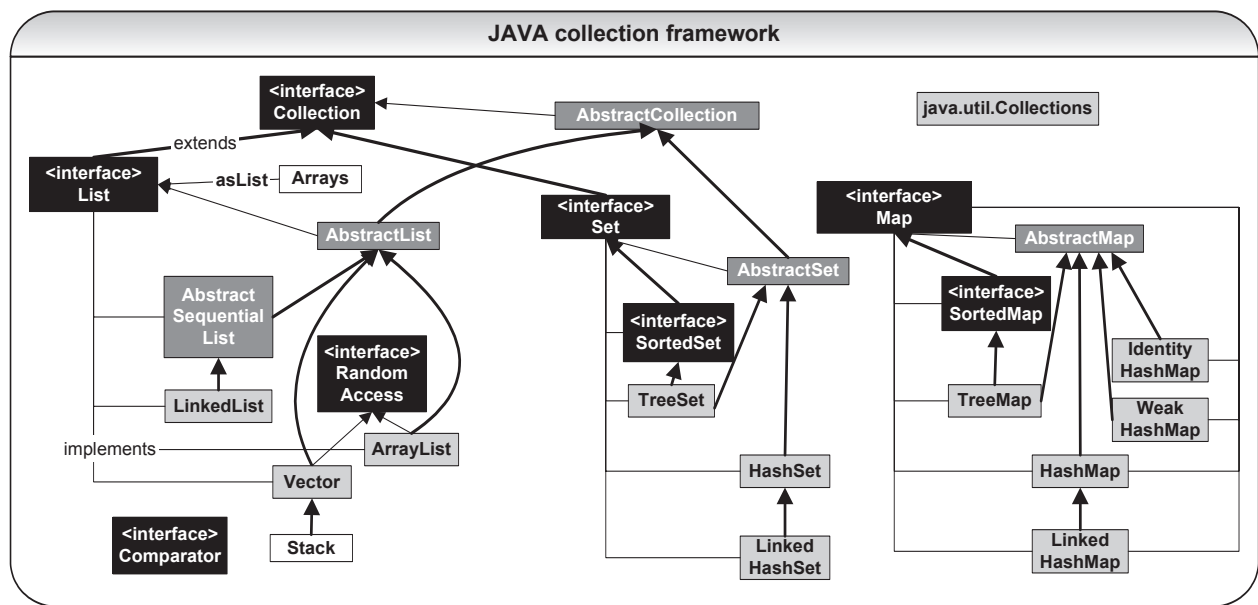
Avoid:

```
myCollection.remove(myObject); // avoid by-passing the Iterator. When it.next() is called, can throw the exception
                                // ConcurrentModificationException
```

Note: If you had used any Object to Relational (OR) mapping frameworks like Hibernate, you may have encountered this exception "ConcurrentModificationException" when you tried to remove an object from a collection such as a `java.util` Set with the intention of deleting that object from the underlying database. This exception is not caused by Hibernate but rather caused by your `java.util.Iterator` (i.e. due to your `it.next()` call). You can use one of the solutions given above.

Q. 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.



(Diagram sourced from: <http://www.wilsonmar.com/1arrays.htm>)

What are the benefits of the Java Collections Framework? Collections framework provides flexibility, performance, and robustness.

- **Polymorphic algorithms** – sorting, shuffling, reversing, binary search etc.
- **Set algebra** - such as finding subsets, intersections, and unions between objects.
- **Performance** - collections have much better performance compared to the older *Vector* and *Hashtable* classes with the elimination of synchronization overheads.
- **Thread-safety** - when synchronization is required, wrapper implementations are provided for temporarily synchronizing existing collection objects. For J2SE 5.0 use *java.util.concurrent* package.
- **Immutability** - when immutability is required wrapper implementations are provided for making a collection immutable.
- **Extensibility** - interfaces and abstract classes provide an excellent starting point for adding functionality and features to create specialized object collections.

Q. What are static factory methods? CO

Some of the above mentioned features like searching, sorting, shuffling, immutability etc are achieved with *java.util.Collections* class and *java.util.Arrays* utility classes. The great majority of these implementations are provided via **static factory methods** in a single, non-instantiable (i.e. private constructor) class. Speaking of **static factory methods**, they are an alternative to creating objects through constructors. Unlike constructors, static factory methods are not required to create a new object (i.e. a duplicate object) each time they are invoked (e.g. immutable instances can be cached) and also they have a more meaningful names like *valueOf*, *instanceOf*, *asList* etc. For example:

Instead of:

```
String[] myArray = {"Java", "J2EE", "XML", "JNDI"};
for (int i = 0; i < myArray.length; i++) {
    System.out.println(myArray[i]);
}
```

You can use:

```
String[] myArray = {"Java", "J2EE", "XML", "JNDI"};
System.out.println(Arrays.asList(myArray)); //factory method Arrays.asList(...)
```

For example: The following static factory method (an alternative to a constructor) example converts a **boolean** primitive value to a **Boolean** wrapper object.

```
public static Boolean valueOf(boolean b) {
    return (b ? Boolean.TRUE : Boolean.FALSE)
}
```

Q 17: What are some of the best practices relating to Java collection? **[BP][PI][CI]**

A 17:

- Use ArrayList, HashMap etc as opposed to Vector, Hashtable etc, where possible to avoid any synchronization overhead. Even better is to use just arrays where possible. If multiple threads concurrently access a collection and **at least one of the threads either adds or deletes an entry into the collection**, then the collection must be externally synchronized. This is achieved by:

```
Map    myMap    = Collections.synchronizedMap (myMap); //conditional thread-safety
List   myList   = Collections.synchronizedList (myList); //conditional thread-safety
// use java.util.concurrent package for J2SE 5.0 Refer Q16 in Java section under ConcurrentModificationException
```

- Set the initial capacity of a collection appropriately (e.g. ArrayList, HashMap etc). This is because *Collection* classes like ArrayList, HashMap etc must grow periodically to accommodate new elements. But if you have a very large array, and you know the size in advance then you can speed things up by setting the initial size appropriately.

For example: HashMaps/Hashtables need to be created with sufficiently large capacity to minimize **rehashing** (which happens every time the table grows). HashMap has two parameters initial capacity and load factor that affect its performance and space requirements. Higher load factor values (default load factor of 0.75 provides a good trade off between performance and space) will reduce the space cost but will increase the lookup cost of myMap.get(...) and myMap.put(...) methods. When the number of entries in the HashMap exceeds the **current capacity * loadfactor** then the capacity of the HasMap is roughly doubled by calling the rehash function. It is also very important not to set the initial capacity too high or load factor too low if iteration performance or reduction in space is important.

- Program in terms of interface not implementation:** **[CO]** For example you might decide a LinkedList is the best choice for some application, but then later decide ArrayList might be a better choice for performance reason. **[CO]**

Use:

```
List list = new ArrayList(100); // program in terms of interface & set the initial capacity.
```


Instead of:

```
ArrayList list = new ArrayList();
```

- Return zero length collections or arrays as opposed to returning null:** **[CO]** Returning null instead of zero length collection (use *Collections.EMPTY_SET*, *Collections.EMPTY_LIST*, *Collections.EMPTY_MAP*) is more error prone, since the programmer writing the calling method might forget to handle a return value of null.
- Immutable objects should be used as keys for the HashMap:** **[CO]** Generally you use a java.lang.Integer or a java.lang.String class as the key, which are immutable Java objects. If you define your own key class then it is a best practice to make the key class an immutable object (i.e. do not provide any setXXX() methods etc). If a programmer wants to insert a new key then he/she will always have to instantiate a new object (i.e. cannot mutate the existing key because immutable key object class has no setter methods). Refer **Q20** in Java section under “**Q. Why is it a best practice to implement the user defined key class as an immutable object?**”
- Encapsulate collections:** **[CO]** In general collections are not immutable objects. So care should be taken not to unintentionally expose the collection fields to the caller.

Avoid where possible	Better approach
<p>The following code snippet exposes the Set “setCars” directly to the caller. This approach is riskier because the variable “cars” can be modified unintentionally.</p> <pre>public class CarYard{ //... private Set<Car> cars = new HashSet<Car>(); //exposes the cars to the caller public Set<Car> getCars() { return cars; } //exposes the cars to the caller public void setCars(Set<Car> cars) {</pre>	<p>This approach prevents the caller from directly using the underlying variable “cars”.</p> <pre>public class CarYard{ private Set<Car> cars = new HashSet<Car>(); //... public void addCar(Car car) { cars.add(car); } public void removeCar(Car car) { cars.remove(car); } }</pre>

<pre> this.cars = cars; } //... } </pre>	<pre> public Set<Car> getCars() { //use factory method from the Collections return Collections.unmodifiableSet (cars); } } </pre>
---	---

- **Avoid storing unrelated or different types of objects into same collection:**  This is analogous to storing items in pigeonholes without any labeling. To store items use **value objects** or **data objects** (as opposed to storing every attribute in an ArrayList or HashMap). Provide wrapper classes around your collections API classes like ArrayList, HashMap etc as shown in better approach column. Also where applicable consider using **composite design pattern**, where an object may represent a single object or a collection of objects. Refer **Q61** in Java section for UML diagram of a composite design pattern. If you are using J2SE 5.0 then make use of “**generics**”. Refer **Q55** in Java section for generics.

Avoid where possible	Better approach
<p>The code below is hard to maintain and understand by others. Also gets more complicated as the requirements grow in the future because we are throwing different types of objects like Integer, String etc into a list just based on the indices and it is easy to make mistakes while casting the objects back during retrieval.</p> <pre> List myOrder = new ArrayList() ResultSet rs = ... While (rs.hasNext()) { List linItem = new ArrayList(); linItem.add (new Integer(rs.getInt("itemId"))); linItem.add (rs.getString("description")); myOrder.add(linItem); } return myOrder; </pre> <p>Example 2:</p> <pre> List myOrder = new ArrayList(10); //create an order OrderVO header = new OrderVO(); header.setOrderId(1001); ... //add all the line items LinItemVO line1 = new LinItemVO(); line1.setLinItemId(1); LinItemVO line2 = new LinItemVO(); Line2.setLinItemId(2); List linItems = new ArrayList(); linItems.add(line1); linItems.add(line2); //to store objects myOrder.add(order); // index 0 is an OrderVO object myOrder.add(linItems); //index 1 is a List of line items //to retrieve objects myOrder.get(0); myOrder.get(1); </pre> <p>Above approaches are bad because disparate objects are stored in the linItem collection in example-1 and example-2 relies on indices to store disparate objects. The indices based approach and storing disparate objects are hard to maintain and understand because indices are hard coded and get scattered across the</p>	<p>When storing items into a collection define value objects as shown below: (VO is an acronym for Value Object).</p> <pre> public class LinItemVO { private int itemId; private String productName; public int getLinItemId(){return accountId ;} public int getAccountName(){return accountName;} public void setLinItemId(int accountId){ this.accountId = accountId } //implement other getter & setter methods } </pre> <p>Now let's define our base wrapper class, which represents an order:</p> <pre> public abstract class Order { int orderId; List linItems = null; public abstract int countLinItems(); public abstract boolean add(LinItemVO itemToAdd); public abstract boolean remove(LinItemVO itemToAdd); public abstract Iterator getIterator(); public int getOrderId(){return this.orderId; } } </pre> <p>Now a specific implementation of our wrapper class:</p> <pre> public class OverseasOrder extends Order { public OverseasOrder(int inOrderId) { this.linItems = new ArrayList(10); this.orderId = inOrderId; } public int countLinItems() { //logic to count } public boolean add(LinItemVO itemToAdd){ ...//additional logic or checks return linItems.add(itemToAdd); } public boolean remove(LinItemVO itemToAdd){ return linItems.remove(itemToAdd); } public ListIterator getIterator(){ return linItems.iterator();} } </pre> <p>Now to use:</p> <pre> Order myOrder = new OverseasOrder(1234) ; </pre>

code. If an index position changes for some reason, then you will have to change every occurrence, otherwise it breaks your application.

The above coding approaches are analogous to storing disparate items in a storage system without proper labeling and just relying on its grid position.

```
LineItemVO item1 = new LineItemVO();
Item1.setItemId(1);
Item1.setProductName("BBQ");

LineItemVO item2 = new LineItemVO();
Item1.setItemId(2);
Item1.setProductName("Outdoor chair");

//to add line items to order
myOrder.add(item1);
myOrder.add(item2);
...
```

Q. How can you code better without nested loops?  Avoid nested loops where possible (e.g. for loop within another for loop etc) and instead make use of an appropriate java collection.

How to avoid nested loops with Java collection classes

Code to test if there are duplicate values in an array.

Avoid where possible -- nested loops

```
public class NestedLoops {
    private static String[] strArray = {"Cat", "Dog", "Tiger", "Lion", "Lion"};

    public static boolean isThereDuplicateUsingLoop() {
        boolean duplicateFound = false;
        int loopCounter = 0;
        for (int i = 0; i < strArray.length; i++) {
            String str = strArray[i];
            int countDuplicate = 0;
            for (int j = 0; j < strArray.length; j++) {
                String str2 = strArray[j];
                if (str.equalsIgnoreCase(str2)) {
                    countDuplicate++;
                }

                if (countDuplicate > 1) {
                    duplicateFound = true;
                    System.out.println("duplicate found for " + str);
                }
                loopCounter++;
            } //end of inner nested for loop

            if (duplicateFound) {
                break;
            }
        } //end of outer for loop

        System.out.println("looped " + loopCounter + " times");
        return duplicateFound;
    }

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

output:
duplicate found for Lion
looped 20 times

Better approach -- using a collections class like a Set

```
public class NonNestedLoop {
    private static String[] strArray = {"Cat", "Dog", "Tiger", "Lion", "Lion"};

    public static boolean isThereDuplicateUsingCollection() {
        boolean duplicateFound = false;
        int loopCounter = 0;

        Set setValues = new HashSet(10); // create a set

        for (int i = 0; i < strArray.length; i++) {
            String str = strArray[i];
            if (setValues.contains(str)) { // check if already has this value
                duplicateFound = true;
                System.out.println("duplicate found for " + str);
            }
            setValues.add(str); // add the value to the set

            loopCounter++;

            if (duplicateFound) {
                break;
            }
        } // end of for loop

        System.out.println("looped " + loopCounter + " times");
        return duplicateFound;
    }

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

output:
duplicate found for Lion
looped 5 times

The approach using a Set is more readable and easier to maintain and performs slightly better. If you have an array with 100 items then nested loops will loop through 9900 times and utilizing a collection class will loop through only 100 times.

Q 18: What is the difference between “==” and equals(...) method? What is the difference between shallow comparison and deep comparison of objects? **LF CO FAQ**

A 18: The questions **Q18**, **Q19**, and **Q20** are vital for effective coding. These three questions are vital when you are using a collection of objects **for Example:** using a java.util.Set of persistable Hibernate objects etc. It is easy to implement these methods incorrectly and consequently your program can behave strangely and also is hard to debug. So, you can expect these questions in your interviews.

== [shallow comparison]	equals() [deep comparison]
<p>The == returns true, if the variable reference points to the same object in memory. This is a “shallow comparison”.</p>	<p>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 ==.</p> <p>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).</p> <p>Overriding the Object class may seem simple but there are many ways to get it wrong, and consequence can be unpredictable behavior. Refer Q19 in Java section.</p>
<p style="text-align: center;">== (identity)</p> <p>If (a== b) → returns false</p> <p>Pet a = new Pet();</p> <p>Pet b = new Pet();</p> <p>If (a== b) → returns true (a,b points to the same object, after a is set to b with a=b)</p> <p>a = b</p>	<p style="text-align: center;">equals() method</p> <p>If (a.equals(b)) → returns true (both objects have same attribute values of id=1 and name="Cat")</p> <p>Pet a = new Pet();</p> <p>Pet b = new Pet();</p> <p>If (a.equals(b)) returns true</p> <p>a = b</p>

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.
```

Let us look at an example:


```

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
        }
    }
}

```

Design pattern: 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.

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

A pool of Strings is maintained by the String class. When the intern() method is invoked equals(...) method is invoked to determine if the String already exist in the pool. If it does then the String from the pool is returned. Otherwise, this String object is added to the pool and a reference to this object is returned. For any two Strings s1 & s2, **s1.intern() == s2.intern()** only if s1.equals(s2) is true.

Q 19: What are the non-final methods in Java Object class, which are meant primarily for extension? **LF CO**

A 19: The non-final methods are **equals()**, **hashCode()**, **toString()**, **clone()**, and **finalize()**. The other methods like **wait()**, **notify()**, **notifyAll()**, **getClass()** etc are final methods and therefore cannot be overridden. Let us look at these non-final methods, which are meant primarily for extension (i.e. inheritance).

Important: The **equals()** and **hashCode()** methods prove to be very important, when objects implementing these two methods are added to collections. If implemented incorrectly or not implemented at all then your objects stored in a collection like a Set, List or Map may behave strangely and also is hard to debug.

Method name	Explanation
equals()	This method checks if some other object passed to it as an argument is equal the object in which this method is invoked. It is easy to implement the equals() method incorrectly, if you do not understand the <u>contract</u> . The contract can be stated in terms of 6 simple principles as follows:
method with public access modifier	<ol style="list-style-type: none"> 1. o1.equals(o1) → which means an Object (e.g. o1) should be equal to itself. (aka Reflexive). 2. o1.equals(o2) if and only o2.equals(o1) → So it will be <u>incorrect</u> to have your own class say "MyPet" to have a equals() method that has a comparison with an Object of class "java.lang.String" class or with any other built-in Java class. (aka Symmetric). 3. o1.equals(o2) && o2.equals(o3) implies that o1.equals(o3) as well → It means that if the first object o1 equals to the second object o2 and the second object o2 is equal to the third object o3 then the first object o1 is equal to the third object o3. For example, imagine that X, Y and Z are 3 different classes. The classes X and Y both implement the equals() method in such a way that it provides comparison for objects of class X and class Y. Now if you decide to modify the equals() method of class Y so that it also provides equality comparison with class Z, then you will be violating this principle because no proper equals comparison exist for class X and class Z objects. So, if two objects agree that they are equal and follow the above mentioned symmetric principle, then

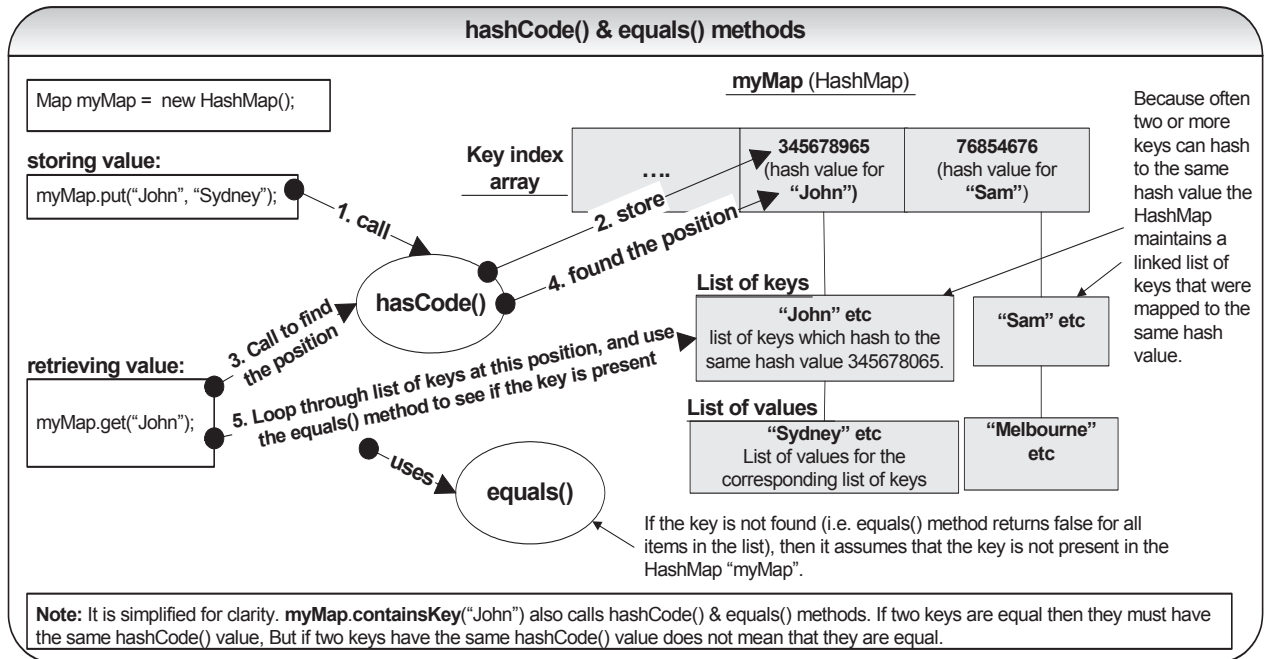
	<p>one of them cannot decide to have a similar contract with another object of different class. (aka Transitive)</p> <ol style="list-style-type: none"> 4. <code>o1.equals(o2)</code> returns the same as long as o1 and o2 are unmodified → if two objects are equal, they must remain equal as long as they are not modified. Similarly, if they are not equal, they must remain non-equal as long as they are not modified. (aka Consistent) 5. <code>!o1.equals(null)</code> → which means that any instantiable object is not equal to null. So if you pass a null as an argument to your object o1, then it should return false. (aka null comparison) 6. <code>o1.equals(o2)</code> implies <code>o1.hashCode() == o2.hashCode()</code> → This is very important. If you define a <code>equals()</code> method then you must define a <code>hashCode()</code> method as well. Also it means that <u>if you have two objects that are equal then they must have the same hashCode. however the reverse is not true (i.e. if two objects have the same hashCode does not mean that they are equal). So. If a field is not used in <code>equals()</code>, then it must not be used in <code>hashCode()</code> method. (<code>equals()</code> and <code>hashCode()</code> relationship)</u> <pre> public class Pet { int id; String name; public boolean equals(Object obj){ if(this == obj) return true; // if both are referring to the same object if ((obj == null) (obj.getClass() != this.getClass())) { return false; } Pet rhs = (Pet) obj; return id == rhs.id && (name == rhs.name (name != null && name.equals(rhs.name))); } //hashCode() method must be implemented here. ... } </pre>
hashCode() method with public access modifier	<p>This method returns a <code>hashCode()</code> value as an Integer and is supported for the benefit of hashing based java.util.Collection classes like Hashtable, HashMap, HashSet etc. If a class overrides the <code>equals()</code> method, it must implement the <code>hashCode()</code> method as well. The general contract of the <code>hashCode()</code> method is that:</p> <ol style="list-style-type: none"> 1. Whenever <code>hashCode()</code> method is invoked on the same object more than once during an execution of a Java program, this method must consistently return the <u>same integer result</u>. The integer result need not remain consistent from one execution of the program to the next execution of the same program. 2. If two objects are equal as per the <code>equals()</code> method, then calling the <code>hashCode()</code> method in each of the two objects <u>must return the same integer result</u>. So, If a field is not used in <code>equals()</code>, then it must not be used in <code>hashCode()</code> method. 3. If two objects are unequal as per the <code>equals()</code> method, each of the two objects can return either two different integer results or same integer results (i.e. <u>if 2 objects have the same <code>hashCode()</code> result does not mean that they are equal, but if two objects are equal then they must return the same <code>hashCode()</code> result</u>). <pre> public class Pet { int id; String name; public boolean equals(Object obj) { //as shown above. } //both fields id & name are used in equals(), so both fields <u>must be used in</u> //hashCode() as well. public int hashCode() { int hash = 9; hash = (31 * hash) + id; hash = (31 * hash) + (null == name ? 0 : name.hashCode()); return hash; } } </pre>
toString()	The <code>toString()</code> method provided by the <code>java.lang.Object</code> returns a string, which consists of the class name

method with public access modifier	<p>followed by an “@” sign and then unsigned hexadecimal representation of the hashCode, for example Pet@162b91. This hexadecimal representation is not what the users of your class want to see.</p> <p>Providing your toString() method makes your class much more pleasant to use and it is recommended that all subclasses override this method. The toString() method is invoked automatically when your object is passed to println(), assert() or the string concatenation operator (+).</p> <pre> public class Pet { int id; String name; public boolean equals(Object obj) { //as shown above. } public int hashCode() { //as shown before } public String toString() { StringBuffer sb = new StringBuffer(); sb.append("id=").append(id); sb.append(",name=").append(name); return sb.toString(); } } </pre>
clone() method with protected access modifier	<p>You should override the clone() method very judiciously. Implementing a properly functioning clone method is complex and it is rarely necessary. You are better off providing some alternative means of object copying (refer Q26 in Java section) or simply not providing the capability. A better approach is to provide a copy constructor or a static factory method in place of a constructor.</p> <pre> //constructor public Pet(Pet petToCopy) { ... } //static factory method public static Pet newInstance(Pet petToCopy) { ... } </pre> <p>The clone() method can be disabled as follows:</p> <pre> public final Object clone() throws CloneNotSupportedException { throw new CloneNotSupportedException(); } </pre>
finalize() method with protected access modifier	<p>Unlike C++ destructors, the finalize() method in Java is unpredictable, often dangerous and generally unnecessary. Use try{} finally{} blocks as discussed in Q32 in Java section & Q45 in Enterprise section. The finalize() method should only be used in rare instances as a safety net or to terminate non-critical native resources. If you do happen to call the finalize() method in some rare instances then remember to call the super.finalize() as shown below:</p> <pre> protected void finalize() throws Throwable { try{ //finalize subclass state } finally { super.finalize(); } } </pre>

Q 20: When providing a user defined key class for storing objects in the HashMaps or Hashtables, what methods do you have to provide or override (i.e. **method overriding**)? **LF PI CO FAQ**

A 20: You should override the equals() and hashCode() methods from the Object class. The default implementation of the equals() and hashCode(), which are inherited from the java.lang.Object uses an object instance's memory location (e.g. MyObject@6c60f2ea). This can cause problems when two instances of the car objects have the same color but the inherited equals() will return false because it uses the memory location, which is different for

the two instances. Also the `toString()` method can be overridden to provide a proper string representation of your object.



Q. What are the primary considerations when implementing a user defined key?

- If a class overrides `equals()`, it must override `hashCode()`.
- If 2 objects are equal, then their `hashCode` values must be equal as well.
- If a field is not used in `equals()`, then it must not be used in `hashCode()`.
- If it is accessed often, `hashCode()` is a candidate for caching to enhance performance.
- It is a best practice to implement the user defined key class as an immutable (refer Q21) object.

Q. Why it is a best practice to implement the user defined key class as an immutable object?

Problem: As per the code snippet shown below if you use a mutable user defined class "UserKey" as a HashMap key and subsequently if you mutate (i.e. modify via setter method e.g. `key.setName("Sam")`) the key after the object has been added to the HashMap then you will not be able to access the object later on. The original key object will still be in the HashMap (i.e. you can iterate through your HashMap and print it – both prints as "Sam" as opposed to "John" & Sam) but you cannot access it with `map.get(key)` or querying it with `map.containsKey(key)` will return false because the key "John" becomes "Sam" in the "List of keys" at the key index "345678965" if you mutate the key after adding. These types of errors are very hard to trace and fix.

```
Map myMap = new HashMap(10);
//add the key "John"
UserKey key = new UserKey("John"); //Assume UserKey class is mutable
myMap.put(key, "Sydney");
//now to add the key "Sam"
key.setName("Sam"); // same key object is mutated instead of creating a new instance.
// This line modifies the key value "John" to "Sam" in the "List of keys"
// as shown in the diagram above. This means that the key "John" cannot be
// accessed. There will be two keys with "Sam" in positions with hash
// values 345678965 and 76854676.
myMap.put(key, "Melbourne");

myMap.get(new UserKey("John")); // key cannot be accessed. The key hashes to the same position
// 345678965 in the "Key index array" but cannot be found in the "List of keys"
```

Solution: Generally you use a `java.lang.Integer` or a `java.lang.String` class as the key, which are immutable Java objects. If you define your own key class then it is a best practice to make the key class an immutable object (i.e. do not provide any `setXXX()` methods in your key class. e.g. no `setName(...)` method in the `UserKey` class). If a programmer wants to insert a new key then he/she will always have to instantiate a new object (i.e. cannot mutate the existing key because immutable key object class has no setter methods).

```

Map myMap = new HashMap(10);
//add the key "John"
UserKey key1 = new UserKey("John"); //Assume UserKey is immutable
myMap.put(key1, "Sydney");

//add the key "Sam"
UserKey key2 = new UserKey("Sam"); //Since UserKey is immutable, new instance is created.
myMap.put(key2, "Melbourne");

myMap.get(new UserKey("John")); //Now the key can be accessed

```

Similar issues are possible with the *Set* (e.g. *HashSet*) as well. If you add an object to a "Set" and subsequently modify the added object and later on try to query the original object it may not be present. `mySet.contains(originalObject)` may return false.

J2SE 5.0 introduces enumerated constants, which improves readability and maintainability of your code. Java programming language enums are more powerful than their counterparts in other languages. **Example:** As shown below a class like "*Weather*" can be built on top of simple enum type "*Season*" and the class "*Weather*" can be made immutable, and only one instance of each "*Weather*" can be created, so that your *Weather* class **does not have to override equals()** and **hashCode()** methods.

```

public class Weather {
    public enum Season {WINTER, SPRING, SUMMER, FALL}
    private final Season season;
    private static final List<Weather> listWeather = new ArrayList<Weather> ();

    private Weather (Season season) { this.season = season;}
    public Season getSeason () { return season;}

    static {
        for (Season season : Season.values()) { //using J2SE 5.0 for each loop
            listWeather.add(new Weather(season));
        }
    }

    public static ArrayList<Weather> getWeatherList () { return listWeather; }
    public String toString(){ return season;} //takes advantage of toString() method of Season.
}

```

Q 21: What is the main difference between a String and a StringBuffer class? **LF PI CI CO FAQ**

A 21:

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.
<pre>//Inefficient version using immutable String String output = "Some text" int count = 100; for(int i =0; i<count; i++) { output += i; } return output;</pre> <p>The above code would build 99 new String objects, of which 98 would be thrown away immediately. Creating new objects is not efficient.</p>	<pre>//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();</pre> <p>The above code creates only two new objects, the <i>StringBuffer</i> and the final <i>String</i> that is returned. <i>StringBuffer</i> expands as needed, which is costly however, so it would be better to initialize the <i>StringBuffer</i> with the correct size from the start as shown.</p>

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.

Q. 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.

Q. What are the benefits of immutable objects?

- Immutable classes can greatly simplify programming by freely allowing you to cache and share the references to the immutable objects without having to defensively copy them or without having to worry about their values becoming stale or corrupted.
- Immutable classes are inherently thread-safe and you do not have to synchronize access to them to be used in a multi-threaded environment. So there is no chance of negative performance consequences.
- Eliminates the possibility of data becoming inaccessible when used as keys in HashMaps or as elements in Sets. These types of errors are hard to debug and fix. Refer **Q20** in Java section under **“Q. Why it is a best practice to implement the user defined key class as an immutable object?”**

Q. How will you write an immutable class? CO

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.
4. The “this” reference is not allowed to escape during construction from the immutable class and the immutable class should have exclusive access to fields that contain references to mutable objects like arrays, collections and mutable classes like `Date` etc by:

- Declaring the mutable references as private.
- Not returning or exposing the mutable references to the caller (this can be done by defensive copying)

Wrong way to write an immutable class	Right way to write an immutable class
<p>Wrong way to write a constructor:</p> <pre>public final class MyImmutable { private final int[] myArray; public MyImmutable(int[] anArray) { this.myArray = anArray; // wrong } public String toString() { StringBuffer sb = new StringBuffer("Numbers are: "); for (int i = 0; i < myArray.length; i++) { sb.append(myArray[i] + " "); } return sb.toString(); } }</pre> <p>// the caller could change the array after calling the constructor.</p> <pre>int[] array = {1,2}; MyImmutable myImmutableRef = new MyImmutable(array); System.out.println("Before constructing " + myImmutableRef); array[1] = 5; // change (i.e. mutate) the element System.out.println("After constructing " + myImmutableRef);</pre> <p>Out put: Before constructing Numbers are: 1 2</p>	<p>Right way is to <u>copy the array before assigning in the constructor</u>.</p> <pre>public final class MyImmutable { private final int[] myArray; public MyImmutable(int[] anArray) { this.myArray = anArray.clone(); // defensive copy } public String toString() { StringBuffer sb = new StringBuffer("Numbers are: "); for (int i = 0; i < myArray.length; i++) { sb.append(myArray[i] + " "); } return sb.toString(); } }</pre> <p>// the caller cannot change the array after calling the constructor.</p> <pre>int[] array = {1,2}; MyImmutable myImmutableRef = new MyImmutable(array); System.out.println("Before constructing " + myImmutableRef); array[1] = 5; // change (i.e. mutate) the element System.out.println("After constructing " + myImmutableRef);</pre> <p>Out put: Before constructing Numbers are: 1 2</p>

After constructing Numbers are: 1 5 As you can see in the output that the "MyImmutable" object has been mutated. This is because the object reference gets copied as discussed in Q22 in Java section. Wrong way to write an accessor. A caller could get the array reference and then change the contents:	After constructing Numbers are: 1 2 As you can see in the output that the "MyImmutable" object has not been mutated. Right way to write an accessor by cloning.
<pre>public int[] getArray() { return myArray; }</pre>	<pre>public int[] getArray() { return (int[]) myArray.clone(); }</pre>

Important: Beware of using the clone() method on a collection like a Map, List, Set etc because they are not only difficult to implement correctly refer **Q19** in Java section but also the default behavior of an object's clone() method automatically yields a shallow copy. You have to deep copy the mutable objects referenced by your immutable class. Refer **Q26** in Java section for deep vs. shallow cloning and **Q22** in Java section for why you will be modifying the original object if you do not deep copy.

Q. How would you defensively copy a Date field in your immutable class?

```
public final class MyDiary {
    private Date myDate = null;

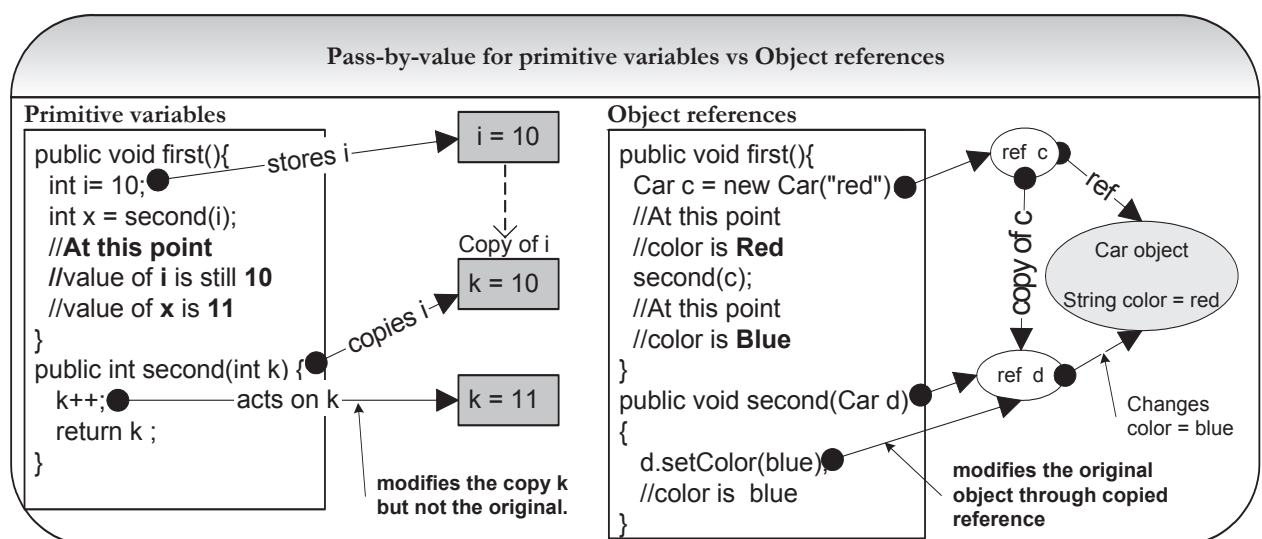
    public MyDiary(Date aDate){
        this.myDate = new Date(aDate.getTime());    // defensive copying by not exposing the "myDate" reference
    }

    public Date getDate() {
        return new Date(myDate.getTime());          // defensive copying by not exposing the "myDate" reference
    }
}
```

Q 22: What is the main difference between pass-by-reference and pass-by-value? **LF PI FAQ**

A 22: Other languages use **pass-by-reference** or pass-by-pointer. But in Java no matter what type of argument you pass the corresponding parameter (primitive variable or object reference) will get a copy of that data, which is exactly how **pass-by-value** (i.e. copy-by-value) works.

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

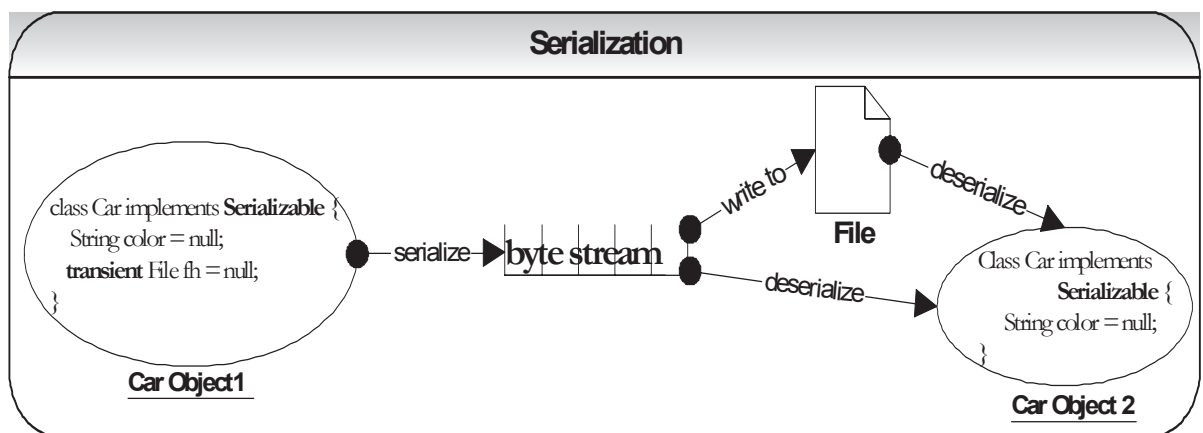
JVM). Hence inter-process communication involves calling method passing objects as arguments to called method **by-value** in a serialized form, which can adversely affect performance due to marshaling and unmarshaling cost.

Note: As discussed in **Q69** in Enterprise section, EJB 2.x introduced local interfaces, where enterprise beans that can be used locally within the same JVM using Java's form of **pass-by-reference**, hence improving performance.

Q 23: What is serialization? How would you exclude a field of a class from serialization or what is a transient variable? What is the common use? What is a serial version id? **LF SI PI FAQ**

A 23: 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.



Serialization can adversely affect performance since it:

- Depends on reflection.
- Has an incredibly verbose data format.
- Is very easy to send surplus data.

Q. When to use serialization? Do not use serialization if you do not have to. 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 (Refer **Q26** in Java section).

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();

```

The **objects stored in an HTTP session should be serializable** to support in-memory replication of sessions to achieve scalability (Refer **Q20** in Enterprise section). Objects are passed in RMI (Remote Method Invocation) across network using serialization (Refer **Q57** in Enterprise section).

Q. What is Java 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:

```
public class Car {
    static final long serialVersionUID = 1L; //assign a long value
}
```

Note: Alternatively you can use the serialver tool comes with Sun's JDK. This tool takes a full class name on the command line and returns the serialVersionUID for that compiled class. **For example:**

```
static final long serialVersionUID = 10275439472837494L; //generated by serialver tool.
```

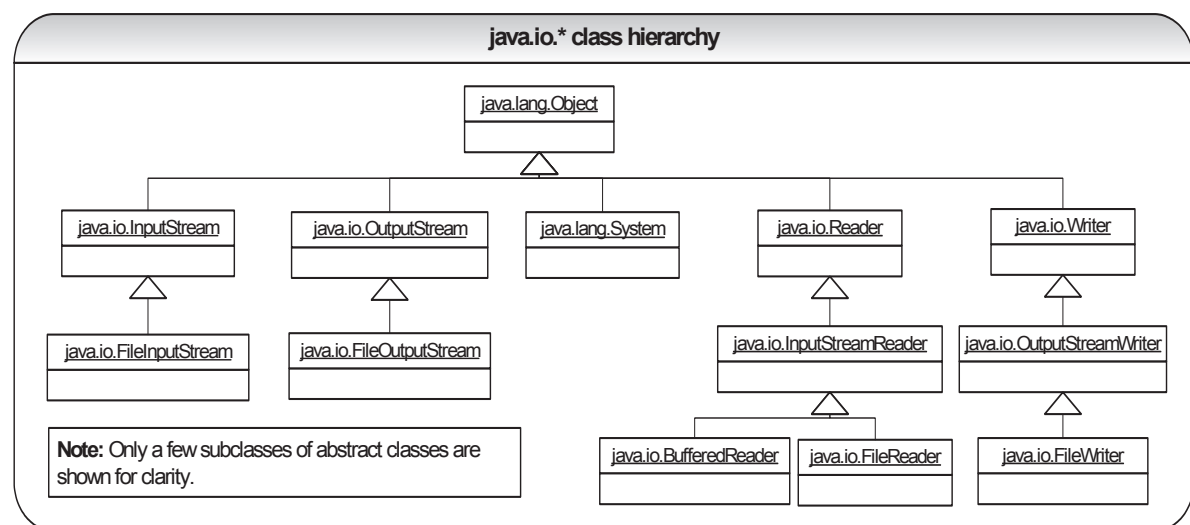
Q 24: Explain the Java I/O streaming concept and the use of the decorator design pattern in Java I/O? **[L F D P I S]**

A 24: Java input and output is defined in terms of an abstract concept called a “**stream**”, which is a sequence of data. 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**

Design pattern: *java.io.** classes use the **decorator design pattern**. The decorator design pattern **attaches responsibilities to objects at runtime**. Decorators are more flexible than inheritance because the **inheritance attaches responsibility to classes at compile time**. The *java.io.** classes use the decorator pattern to construct different combinations of behavior at runtime based on some basic classes.

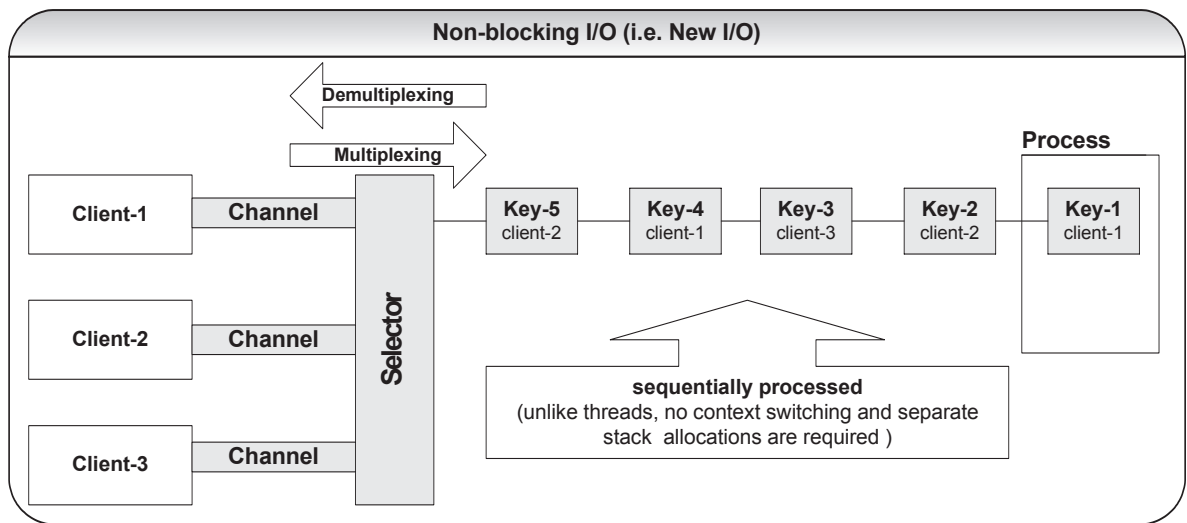
Attaching responsibilities to classes at compile time using subclassing.	Attaching responsibilities to objects at runtime using a decorator design pattern.
Inheritance (aka subclassing) attaches responsibilities to classes at compile time. When you extend a class, each individual changes you make to child class will affect all instances of the child classes. Defining many classes using inheritance to have all possible combinations is problematic and inflexible.	By attaching responsibilities to objects at runtime , you can apply changes to each individual object you want to change. <pre>File file = new File("c:/temp"); FileInputStream fis = new FileInputStream(file); BufferedInputStream bis = new BufferedInputStream(fis);</pre> <p>Decorators decorate an object by enhancing or restricting functionality of an object it decorates. The decorators add or restrict functionality to decorated objects either before or after forwarding the request. At runtime the BufferedInputStream (bis), which is a decorator (aka a wrapper around decorated object), forwards the method call to its decorated object FileInputStream (fis). The “bis” will apply the additional functionality of buffering around the lower level file (i.e. fis) I/O.</p>



Q. 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.

A server's ability to handle several client requests effectively depends on how it uses I/O streams. When a server has to handle hundreds of clients simultaneously, it must be able to use I/O services concurrently. One way to cater for this scenario in Java is to use threads but having almost one-to-one ratio of threads (100 clients will have 100 threads) is prone to enormous **thread overhead and can result in performance and scalability problems due to consumption of memory stacks** (i.e. each thread has its own stack. Refer **Q34, Q42** in Java section) and **CPU context switching** (i.e. switching between threads as opposed to doing real computation.). To overcome this problem, a new set of non-blocking I/O classes have been introduced to the Java platform in java.nio package. 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.

Design pattern: NIO uses a **reactor design pattern**, which demultiplexes events (separating single stream into multiple streams) and dispatches them to registered object handlers. The reactor pattern is similar to an **observer pattern** (aka publisher and subscriber design pattern), but an observer pattern handles only a single source of events (i.e. a single publisher with multiple subscribers) where a reactor pattern handles multiple event sources (i.e. multiple publishers with multiple subscribers). The intent of an observer pattern is to define a one-to-many dependency so that when one object (i.e. the publisher) changes its state, all its dependents (i.e. all its subscribers) are notified and updated correspondingly.

Another sought after functionality of NIO is its ability to **map a file to memory**. There is a specialized form of a Buffer known as "MappedByteBuffer", which represents a buffer of bytes mapped to a file. To map a file to "MappedByteBuffer", you must first get a channel for a file. Once you get a channel then you map it to a buffer and subsequently you can access it like any other "ByteBuffer". Once you map an input file to a "CharBuffer", you can do pattern matching on the file contents. This is similar to running "grep" on a UNIX file system.

Another feature of NIO is its **ability to lock and unlock files**. Locks can be exclusive or shared and can be held on a contiguous portion of a file. But file locks are subject to the control of the underlying operating system.

Q 25: How can you improve Java I/O performance? **PI** **BP**

A 25: 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. **CO**

- 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	With Buffering: yields better performance
<pre>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){}</pre> <p>Note: fis.read() is a native method call to the underlying operating system.</p>	<pre>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){}</pre> <p>Note: bis.read() takes the next byte from the input buffer and only rarely access the underlying operating system.</p>

Instead of reading a character or a byte at a time, the above code with buffering can be improved further by reading one line at a time as shown below:

```
FileReader fr = new FileReader(f);
BufferedReader br = new BufferedReader(fr);
while (br.readLine() != null) count++;
```

By default the **System.out** is line buffered, which means that the output buffer is flushed when a new line character (i.e. "\n") is encountered. This is required for any interactivity between an input prompt and display of output. The line buffering can be disabled for faster I/O operation as follows:

```
FileOutputStream fos = new FileOutputStream(file);
BufferedOutputStream bos = new BufferedOutputStream(fos, 1024);
PrintStream ps = new PrintStream(bos, false);

// To redirect standard output to a file instead of the "System" console which is the default for both "System.out" (i.e.
// standard output) and "System.err" (i.e. standard error device) variables

System.setOut(ps);

while (someConditionIsTrue)
    System.out.println("blah...blah...");
}
```

It is recommended to use logging frameworks like **Log4J** with **SLF4J** (Simple Logging Façade for Java), which uses buffering instead of using default behavior of **System.out.println(.....)** for better performance. Frameworks like Log4J are configurable, flexible, extensible and easy to use.

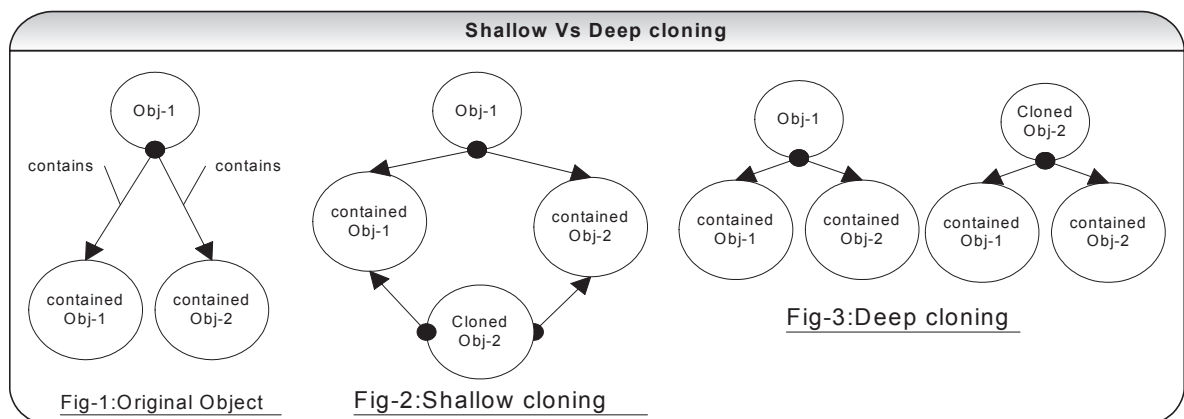
- Use the NIO package, if you are using JDK 1.4 or later, which uses performance-enhancing features like buffers to hold data, memory mapping of files, non-blocking I/O operations etc.
- I/O performance can be improved by minimizing the calls to the underlying operating systems. The Java runtime itself cannot know the length of a file, querying the file system for `isDirectory()`, `isFile()`, `exists()` etc must query the underlying operating system.
- Where applicable caching can be used to improve performance by reading in all the lines of a file into a Java *Collection* class like `ArrayList` or a `HashMap` and subsequently access the data from an in-memory collection instead of the disk.

Q 26: What is the main difference between shallow cloning and deep cloning of objects? **DC LF MI PI**

A 26: 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: If a shallow copy is performed on `obj-1` as shown in fig-2 then it is copied but its contained objects are not. The contained objects `Obj-1` and `Obj-2` are affected by changes to cloned `Obj-2`. Java supports shallow cloning of objects by default when a class implements the *java.lang.Cloneable* interface.

Deep copy: 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.



For example invoking `clone()` method on a collection like *HashMap*, *List* etc returns a shallow copy of *HashMap*, *List*, instances. This means if you clone a *HashMap*, the map instance is cloned but **the keys and values themselves are not cloned**. If you want a deep copy then a simple method is to serialize the *HashMap* to a *ByteArrayOutputStream* and then deserialize it. This creates a deep copy but does require that all keys and values in the *HashMap* are *Serializable*. Main advantage of this approach is that it will deep copy any arbitrary object graph. Refer **Q23** in Java section for deep copying using *Serialization*. Alternatively you can provide a static factory method to deep copy. **Example:** to deep copy a list of *Car* objects.

```
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;
}
```

Q 27: What is the difference between an instance variable and a static variable? How does a local variable compare to an instance or a static variable? Give an example where you might use a static variable? **LF FAQ**

A 27:

Static variables	Instance variables
Class variables are called static variables. There is only one occurrence of a class variable per JVM per class loader. When a class is loaded the class variables (aka static variables) are initialized.	Instance variables are non-static and there is one occurrence of an instance variable in each class instance (i.e. each object). Also known as a member variable or a field .

A static variable is used in the **singleton** pattern. (Refer **Q51** in Java section). A static variable is used with a **final** modifier to define **constants**.

Local variables	Instance and static variables
Local variables have a <u>narrower scope</u> than instance variables.	Instance variables have a narrower scope than static variables.
The lifetime of a local variable is determined by execution path and <u>local variables are also known as stack variables because they live on the stack</u> . Refer Q34 for stack & heap.	<u>Instance and static variables are associated with objects and therefore live in the heap</u> . Refer Q34 in Java section for stack & heap.
For a local variable, it is <u>illegal for code to fail to assign it a value</u> . It is the best practice to declare local variables only where required as opposed to declaring them upfront and cluttering up your code with some local variables that never get used.	Both the static and instance variables always have a <u>value</u> . If your code does not assign them a value then the run-time system will implicitly assign a default value (e.g. null/0/0.0/false).

Note: Java does not support global, universally accessible variables. You can get the same sorts of effects with classes that have static variables.

Q 28: Give an example where you might use a static method? **LF FAQ**

A 28: Static methods prove useful for creating **utility classes**, **singleton classes** and **factory methods** (Refer **Q51**, **Q52** in Java section). Utility classes are not meant to be instantiated. Improper coding of utility classes can lead to procedural coding. **java.lang.Math**, **java.util.Collections** etc are examples of utility classes in Java.

Q 29: What are access modifiers? **LF FAQ**

A 29:

Modifier	Used with	Description
public	Outer classes, interfaces, constructors, inner classes, methods and field variables	A class or interface may be accessed from outside the package. Constructors, inner classes, methods and field variables may be accessed wherever their class is accessed.
protected	Constructors, inner classes, methods, and field variables.	Accessed by other classes in the same package or any subclasses of the class in which they are referred (i.e. same package or different package).
private	Constructors, inner classes, methods and field variables.	Accessed only within the class in which they are declared
No modifier: (Package by default).	Outer classes, inner classes, interfaces, constructors, methods, and field variables	Accessed only from within the package in which they are declared.

Q 30: Where and how can you use a private constructor? **LF FAQ**

A 30: Private constructor is used if you do not want other classes to instantiate the object and to prevent subclassing. The instantiation is done by a public static method (i.e. a static factory method) within the same class.

- Used in the singleton design pattern. (Refer **Q51** in Java section).
- Used in the factory method design pattern (Refer **Q52** in Java section). e.g. **java.util.Collections** class (Refer **Q16** in Java section).
- Used in utility classes e.g. **StringUtils** etc.

Q 31: What is a final modifier? Explain other Java modifiers? **LF FAQ**

A 31: A final class can't be extended i.e. A final class can not be subclassed. A final method can't be overridden when its class is inherited. You can't change value of a final variable (i.e. it is a constant).

Modifier	Class	Method	Variable
static	A static inner class is just an inner class associated with the class, rather than with an instance of the class.	A static method is called by classname.method (e.g Math.random()), <u>can only access static variables</u> .	Class variables are called static variables. There is <u>only one occurrence</u> of a class variable per JVM per class loader.
abstract	An abstract class cannot be instantiated, must be a superclass and a class must be declared abstract whenever one or more methods are abstract.	Method is defined but contains <u>no implementation code</u> (implementation code is included in the subclass). If a method is abstract then the entire class must be abstract.	N/A
synchronized	N/A	Acquires a <u>lock on the class</u> for static methods . Acquires a <u>lock on the instance</u> for non-static methods .	N/A
transient	N/A	N/A	variable should not be serialized.
final	Class cannot be inherited (i.e. extended)	Method cannot be overridden.	Makes the variable immutable.
native	N/A	Platform dependent. No body, only signature.	N/A

Note: Be prepared for tricky questions on modifiers like, what is a “volatile”? Or what is a “const”? Etc. The reason it is tricky is that Java does have these keywords “const” and “volatile” as reserved, which means you can’t name your variables with these names **but modifier “const” is not yet added in the language** and the **modifier “volatile” is very rarely used**.

The “volatile” modifier is used on instance variables that may be modified simultaneously by other threads. The modifier volatile only synchronizes the variable marked as volatile whereas “synchronized” modifier synchronizes all variables. Since other threads cannot see local variables, there is no need to mark local variables as volatile.

For example:

```
volatile int number;
volatile private List listItems = null;
```

Java uses the “final” modifier to declare constants. A final variable or constant declared as “final” has a value that is immutable and cannot be modified to refer to any other objects other than one it was initialized to refer to. So the “final” modifier applies only to the value of the variable itself, and not to the object referenced by the variable. This is where the “const” modifier can come in very **useful if added to the Java language**. A reference variable or a constant marked as “const” refers to an immutable object that cannot be modified. The reference variable itself can be modified, if it is not marked as “final”. The “const” modifier will be applicable only to non-primitive types. The primitive types should continue to use the modifier “final”.

Q. If you want to extend the “java.lang.String” class, what methods will you override in your extending class?

You would be tempted to say equals(), hashCode() and toString() based on **Q19, Q20** in Java section but the “java.lang.String” class is declared final and therefore it cannot be extended.

Q 32: What is the difference between final, finally and finalize() in Java? **LF FAQ**

A 32:

- **final** - constant declaration. Refer **Q31** in Java section.
- **finally** - handles exception. The finally block is optional and provides a mechanism to clean up regardless of what happens within the try block (except System.exit(0) call). Use the finally block to close files or to release other system resources like database connections, statements etc. (Refer **Q45** in Enterprise section)
- **finalize()** - method helps in garbage collection. A **method** that is invoked before an object is discarded by the garbage collector, allowing it to clean up its state. Should not be used to release non-memory resources like file handles, sockets, database connections etc because Java has only a finite number of these resources and you do not know when the garbage collection is going to kick in to release these non-memory resources through the finalize() method. Refer **Q19** in Java Section.

Q 33: Why would you prefer a short circuit “&&, ||” operators over logical “& , |” operators? **LF**

A 33: 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.

There are other ways to check for null but short circuit && operator can simplify your code by not having to declare separate if clauses.

```
if((obj != null) & obj.equals(newObj)) { //can cause a NullPointerException if obj == null
    ...                               // because obj.equals(newObj) is always executed.
}
```

Short-circuiting means that an operator only evaluates as far as it has to, not as far as it can. If the variable 'obj' equals null, it won't even try to evaluate the 'obj.equals(newObj)' clause as shown in the following example. This protects the potential *NullPointerException*.

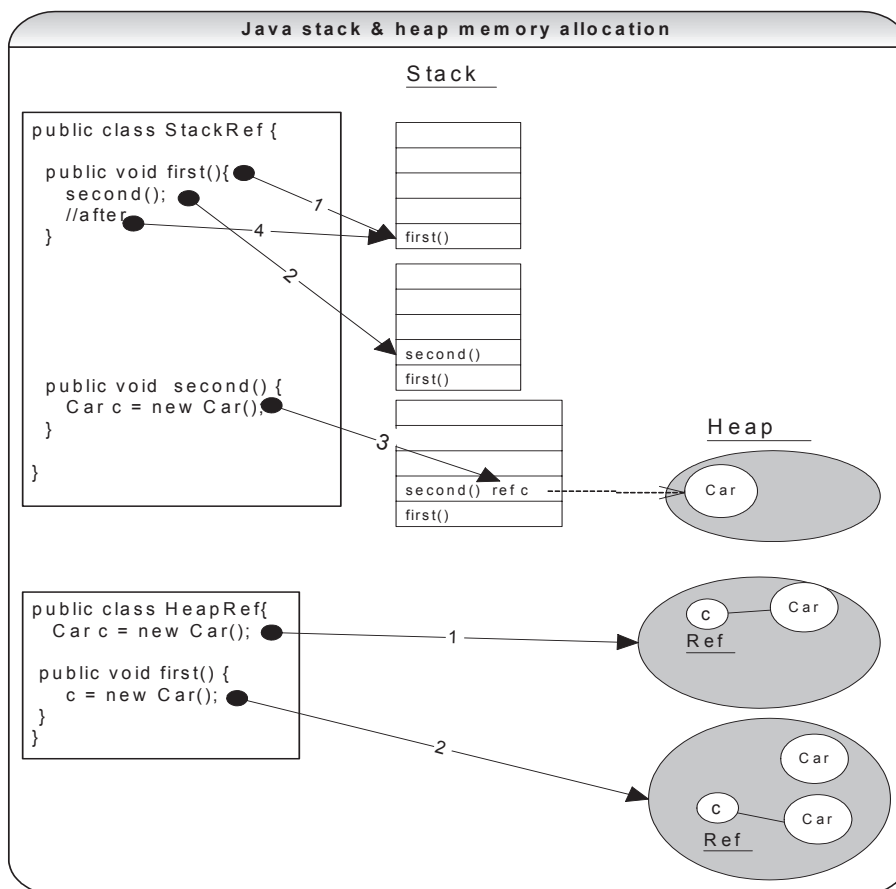
```
if((obj != null) && obj.equals(newObj)) { //cannot get a NullPointerException because
    ...                               //obj.equals(newObj) is executed only if obj != null
}
```

Secondly, short-circuit "&&" and "||" operators can improve performance in certain situations. **For example:**

```
if((number <= 7) || (doComputeIntensiveAnalysis(number) <= 13)) { //the CPU intensive
    ....                               //computational method in bold is executed only if number > 7.
}
```

Q 34: How does Java allocate stack and heap memory? Explain re-entrant, recursive and idempotent methods/functions? **MC**

A 34: 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).

Q 35: Explain Outer and Inner classes (or Nested classes) in Java? When will you use an Inner Class? **LF SE**

A 35: In Java not all classes have to be defined separate from each other. 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.

Q. Where should you use inner classes? 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. Inner classes are used primarily to implement helper classes like Iterators, Comparators etc which are used in the context of an outer class. **CO**

Member inner class	Anonymous inner class
<pre>public class MyStack { private Object[] items = null; ... public Iterator iterator() { return new StackIterator(); } //inner class class StackIterator implements Iterator{ ... public boolean hasNext(){...} } }</pre>	<pre>public class MyStack { private Object[] items = null; ... public Iterator iterator() { return new Iterator { ... public boolean hasNext() {...} } } }</pre>

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 <u>final local variables</u> and <u>final method parameters</u> . Only visible within the block of code that defines it. class Outside { void first() { final int i = 5; class Inside { } } } Outside.class , Outside\$1\$Inside.class

Inner class	Anonymous class	Just like local class, but no name is used. Useful when only one instance is used in a method. Most commonly used in AWT/SWING event model, Spring framework hibernate call back methods etc.	<pre>//AWT example class Outside{ void first(){ button.addActionListener (new ActionListener() { public void actionPerformed(ActionEvent e) { System.out.println("The button was pressed!"); } }); } }</pre> <p><u>Outside.class , Outside\$1.class</u></p>
--------------------	-----------------	---	--

Note: If you have used the **Spring** framework with the **Hibernate** framework (Both are very popular frameworks, Refer section “**Emerging Technologies/Frameworks**”), it is likely that you would have used an anonymous inner class (i.e. a class declared inside a method) as shown below:

```
//anonymous inner classes can only access local variables if they are declared as final
public Pet getPetById(final String id) {
    return (Pet) getHibernateTemplate().execute(new HibernateCallback() {
        public Object doInHibernate(Session session) {
            HibernateTemplate ht = getHibernateTemplate();
            // ... can access variable "id"
            return myPet;
        }
    });
}
```

Q. Are the following valid java statements?

```
Line: OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```

Yes. The above line is valid. It is an instantiation of a **static nested inner class**.

```
OuterClass outerObject = new OuterClass();
Line: OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Yes. The above line is valid. It is an instantiation of a **member inner class**. An instance of an inner class can exist only within an instance of an outer class. The sample code for the above is shown below:

```
public class OuterClass {
    static class StaticNestedClass {
        StaticNestedClass(){
            System.out.println("StaticNestedClass");
        }
    }
    class InnerClass {
        InnerClass(){
            System.out.println("InnerClass");
        }
    }
}
```

Q 36: What is type casting? Explain up casting vs. down casting? When do you get ClassCastException? **LF DP FAQ**

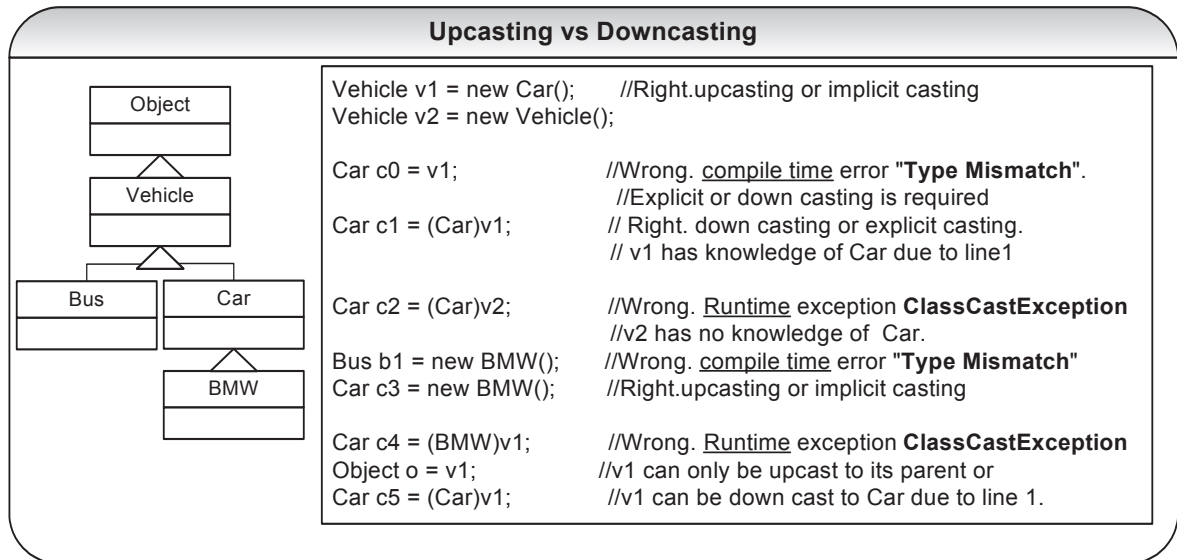
A 36: Type casting means treating a variable of one type as though it is another type.

When up casting **primitives** as shown below from left to right, automatic conversion occurs. But if you go from right to left, down casting or explicit casting is required. Casting in Java is safer than in C or other languages that allow arbitrary casting. Java only lets casts occur when they make sense, such as a cast between a float and an int. However you can't cast between an int and a *String* (is an object in Java).

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.
```

When it comes to object references you can always cast from a subclass to a superclass because a subclass object is also a superclass object. You can cast an object implicitly to a super class type (i.e. **upcasting**). If this were not the case **polymorphism wouldn't be possible**.



You can cast down the hierarchy as well but you must explicitly write the cast and the **object must be a legitimate instance of the class you are casting to**. The **ClassCastException** is thrown to indicate that code has attempted to cast an object to a subclass of which it is not an instance. If you are using J2SE 5.0 then “**generics**” will eliminate the need for casting (Refer **Q55** in Java section) and otherwise you can deal with the problem of incorrect casting in two ways:

- Use the exception handling mechanism to catch **ClassCastException**.

```

try{
    Object o = new Integer(1);
    System.out.println((String) o);
}
catch(ClassCastException cce) {
    logger.log("Invalid casting, String is expected...Not an Integer");
    System.out.println(((Integer) o).toString());
}
        
```

- Use the **instanceof** statement to guard against incorrect casting.

```

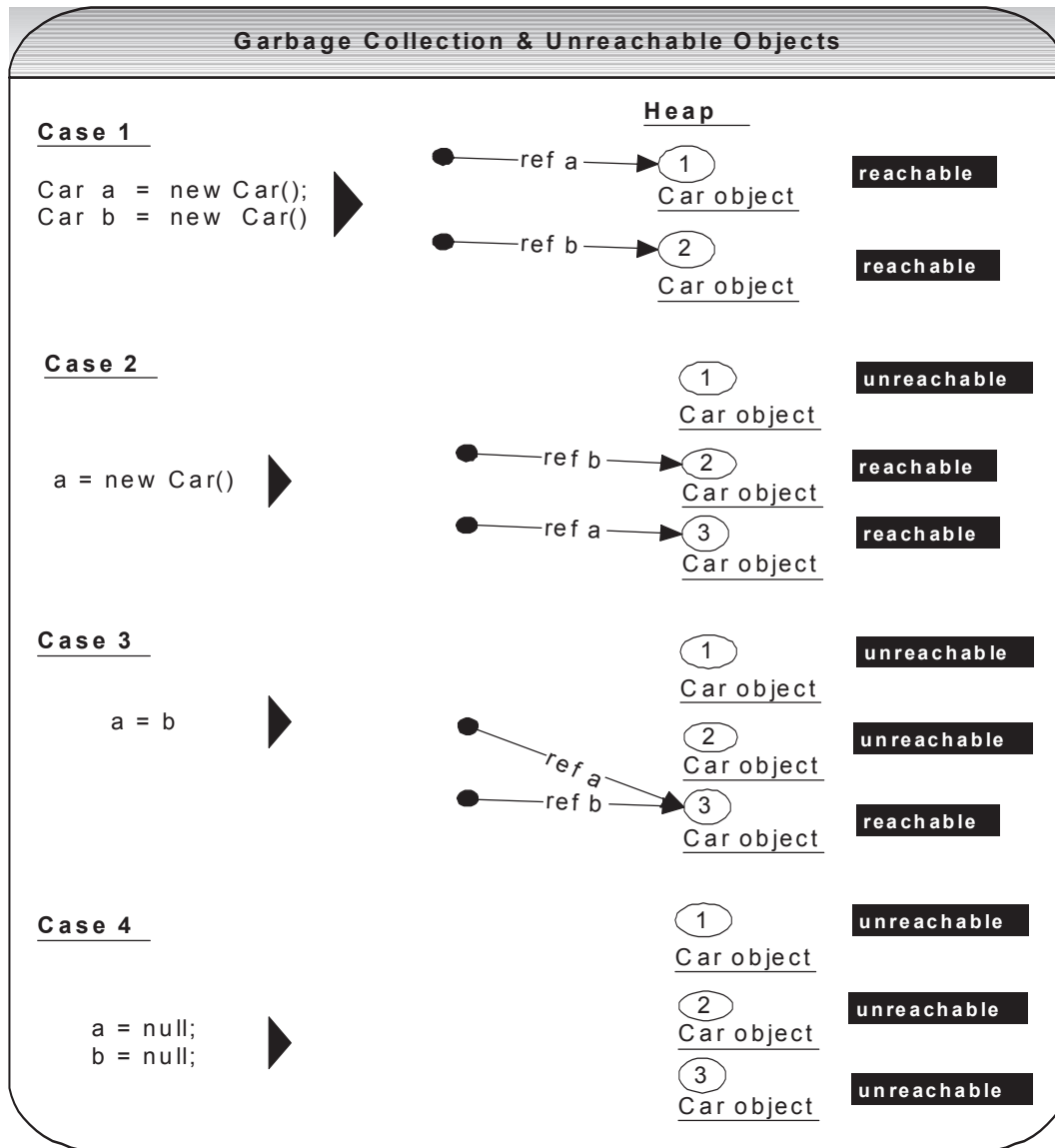
if(v2 instanceof Car) {
    Car c2 = (Car) v2;
}
        
```

Design pattern: The “**instanceof**” and “**typecast**” constructs are shown for the illustration purpose only. Using these constructs can be unmaintainable due to large if and elseif statements and can affect performance if used in frequently accessed methods or loops. Look at using **visitor design pattern** to avoid these constructs where applicable. (Refer **Q11** in How would you go about section...).

Points-to-ponder: You can also get a **ClassCastException** when two different class loaders load the same class because they are treated as two different classes.

Q 37: What do you know about the Java garbage collector? When does the garbage collection occur? Explain different types of references in Java? **LF MI FAQ**

A 37: 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.

What is an unreachable object?

An object's life has no meaning unless something has reference to it. If you can't reach it then you can't ask it to 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.

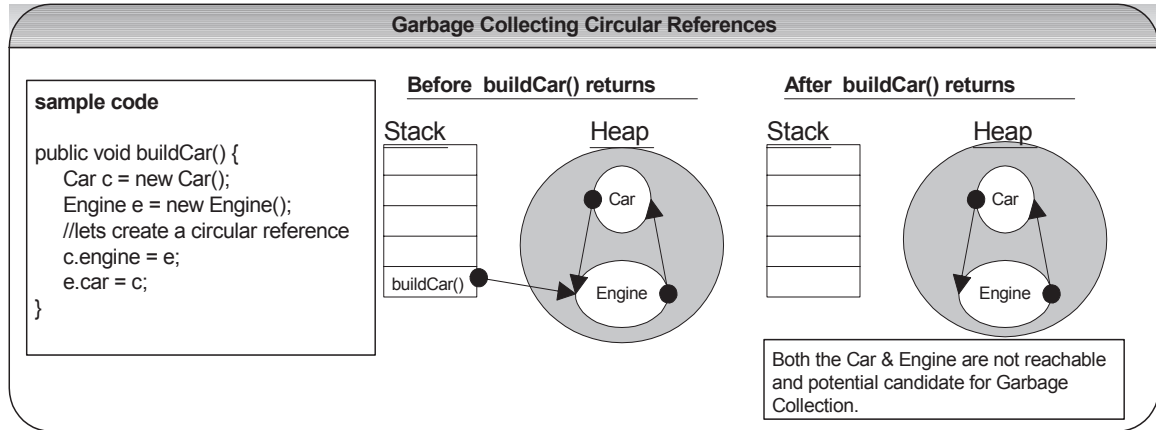
```
java -verbose:gc //reports on each garbage collection event.
```

Explain types of references in Java? *java.lang.ref* package can be used to declare soft, weak and phantom references.

- Garbage Collector won't remove a **strong reference**.
- A **soft reference** will only get removed if memory is low. So it is useful for implementing caches while avoiding memory leaks.
- 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.
- 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.

Q 38: If you have a circular reference of objects, but you no longer reference it from an execution thread, will this object be a potential candidate for garbage collection? **LF MI**

A 38: Yes. Refer diagram below.

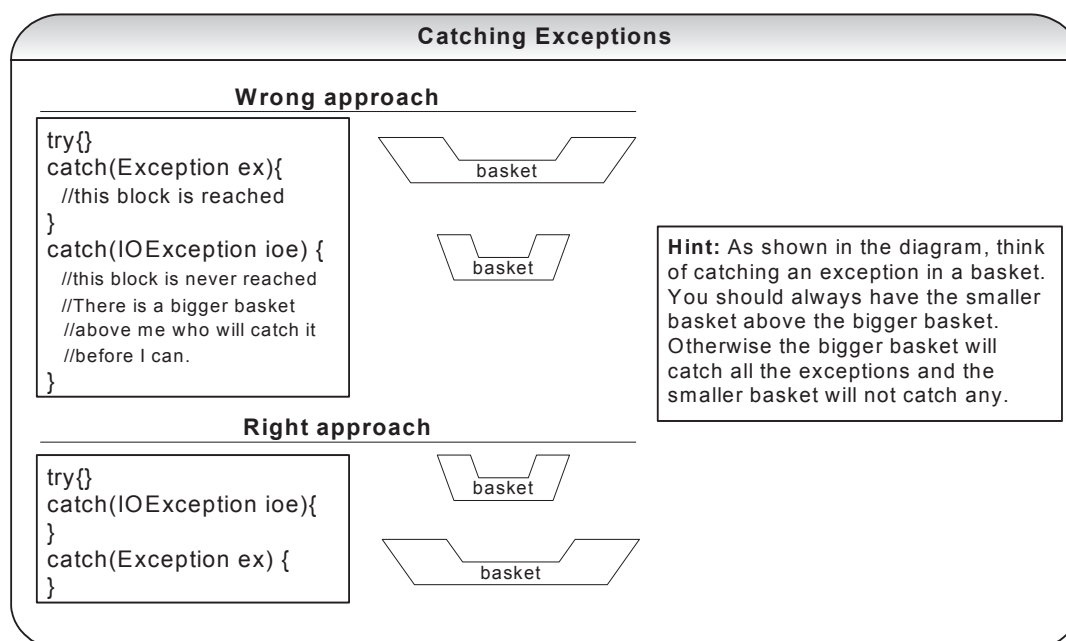


Runtime Exceptions (unchecked exception)

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. 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. In addition, you can create your own `RuntimeException` subclasses and this approach is probably preferred at times because checked exceptions can complicate method signatures and can be difficult to follow.

Q. What are the exception handling best practices: BP**1. Q. Why is it not advisable to catch type “Exception”? CO**

Exception handling in Java is **polymorphic** in nature. 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).



The diagram above is an example for illustration only. In practice it is not recommended to catch type “**Exception**”. We should only catch specific subtypes of the `Exception` class. Having a bigger basket (i.e. `Exception`) will hide or cause problems. Since the `RuntimeException` is a subtype of `Exception`, catching the type `Exception` will catch all the run time exceptions (like `NullPointerException`, `ArrayIndexOutOfBoundsException`) as well.

Example: The `FileNotFoundException` is extended (i.e. inherited) from the `IOException`. So (subclasses have to be caught first) `FileNotFoundException` (small basket) should be caught before `IOException` (big basket).

2. Q. Why should you throw an exception early? CO

The exception stack trace helps you pinpoint where an exception occurred by showing you the exact sequence of method calls that lead to the exception. By throwing your exception early, the exception becomes more accurate and more specific. Avoid suppressing or ignoring exceptions. Also avoid using exceptions just to get a flow control.

Instead of:

```
// assume this line throws an exception because filename == null.
InputStream in = new FileInputStream(fileName);
...
```

Use the following code because you get a more accurate stack trace:

```
...
if(filename == null) {
    throw new IllegalArgumentException("file name is null");
}

InputStream in = new FileInputStream(fileName);
...
```

3. Why should you catch a checked exception late in a catch {} block?

You should not try to catch the exception before your program can handle it in an appropriate manner. The natural tendency when a compiler complains about a checked exception is to catch it so that the compiler stops reporting errors. It is a bad practice to sweep the exceptions under the carpet by catching it and not doing anything with it. The best practice is to catch the exception at the appropriate layer (e.g. an exception thrown at an integration layer can be caught at a presentation layer in a catch {} block), where your program can either meaningfully recover from the exception and continue to execute or log the exception only once in detail, so that user can identify the cause of the exception.

4. Q. When should you use a checked exception and when should you use an unchecked exception?

Due to heavy use of checked exceptions and minimal use of unchecked exceptions, there has been a hot debate in the Java community regarding true value of checked exceptions. Use checked exceptions when the client code can take some useful recovery action based on information in exception. Use unchecked exception when client code cannot do anything. **For example** Convert your SQLException into another checked exception if the client code can recover from it. Convert your SQLException into an unchecked (i.e. RuntimeException) exception, if the client code can not recover from it. (**Note:** Hibernate 3 & Spring uses RuntimeExceptions prevalently).

Important: throw an exception early and catch an exception late but do not sweep an exception under the carpet by catching it and not doing anything with it. This will hide problems and it will be hard to debug and fix. **CO**

A note on key words for error handling:

throw / throws – used to pass an exception to the method that called it.
try – block of code will be tried but may cause an exception.
catch – declares the block of code, which handles the exception.
finally – block of code, which is always executed (except System.exit(0) call) no matter what program flow, occurs when dealing with an exception.
assert – Evaluates a conditional expression to verify the programmer's assumption.

Q 40: What is a user defined exception? **EH**

A 40: 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 {...}`

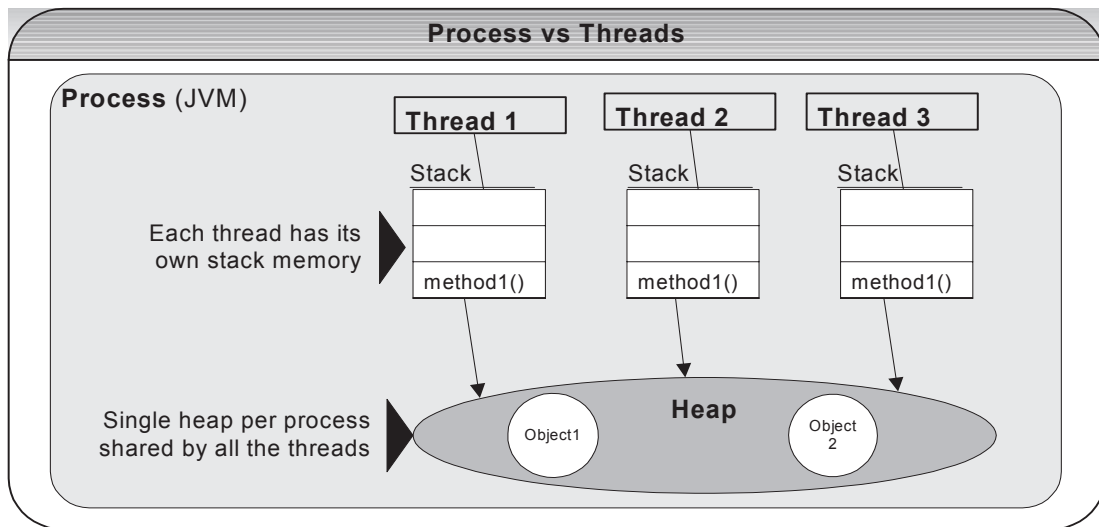
Q 41: What are the flow control statements in Java? **LF**

A 41: The flow control statements allow you to conditionally execute statements, to repeatedly execute a block of statements, or to just change the sequential flow of control.

Flow control types	Keyword
Looping	<p>while, do-while, for</p> <p>The body of the while loop is executed only if the expression is true, so it may not be executed even once:</p> <pre>while(i < 5){...}</pre> <p>The body of the do-while loop is executed at least once because the test expression is evaluated only after executing the loop body. Also, don't forget the ending semicolon after the while expression.</p> <pre>do { ... } while(i < 5);</pre> <p>The for loop syntax is:</p> <pre>for(expr1; expr2; expr3) { // body }</pre> <p>expr1 → is for initialization, expr2 → is the conditional test, and expr3 → is the iteration expression. Any of these three sections can be omitted and the syntax will still be legal:</p> <pre>for(; ;) {} // an endless loop</pre>
Decision making	<p>if-else, switch-case</p> <p>The if-else statement is used for decision-making -- that is, it decides which course of action needs to be taken.</p> <pre>if (x == 5) {...} else {...}</pre> <p>The switch statement is also used for decision-making, based on an integer expression. The argument passed to the switch and case statements should be int, short, char, or byte. The argument passed to the case statement should be a literal or a final variable. If no case matches, the default statement (which is optional) is executed.</p> <pre>int i = 1; switch(i) { case 0: System.out.println("Zero");break; //if break; is omitted case 1: also executed case 1: System.out.println("One");break; //if break; is omitted default: also executed default: System.out.println("Default");break; }</pre>
Branching	<p>break, continue, label:, return</p> <p>The break statement is used to exit from a loop or switch statement, while the continue statement is used to skip just the current iteration and continue with the next. The return is used to return from a method based on a condition. The label statements <u>can lead to unreadable and unmaintainable spaghetti code hence should be avoided</u>.</p>
Exception handling	<p>try-catch-finally, throw</p> <p>Exceptions can be used to define ordinary flow control. <u>This is a misuse of the idea of exceptions, which are meant only for exceptional conditions and hence should be avoided.</u></p>

Q 42: What is the difference between processes and threads? **LF MI CI**

A 42: 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.

Q 43: Explain different ways of creating a thread? **LF** **FAQ**

A 43: 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
    }

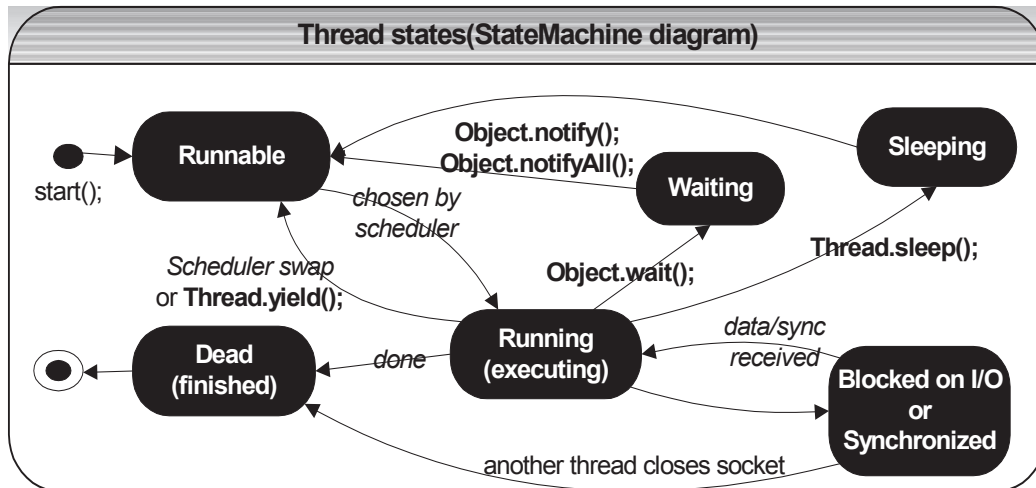
    //let us see how to start the threads
    public static void main(String[] args){
        Thread t1 = new Thread(new Counter());
        Thread t2 = new Thread(new Counter());
        t1.start(); //start the first thread. This calls the run() method.
        t2.start(); //this starts the 2nd thread.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

Q. Which one would you prefer and why? The **Runnable** interface is preferred, as it does not require your object to inherit a thread because when you need multiple inheritance, only interfaces can help you. In the above example we had to extend the **Base** class so implementing **Runnable** interface is an obvious choice. Also note how the threads are started in each of the different cases as shown in the code sample. In an OO approach you

should only extend a class when you want to make it different from its superclass, and change its behavior. By implementing a *Runnable* interface instead of extending the *Thread* class, you are telling to the user that the class *Counter* that an object of type *Counter* will run as a thread.

Q 44: Briefly explain high-level thread states? **LF**

A 44: The state chart diagram below describes the thread states. (Refer **Q107** in Enterprise section for state chart diagram).



(Diagram sourced from: <http://www.wilsonmar.com/1threads.htm>)

- **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.

Q 45: What is the difference between yield and sleeping? What is the difference between the methods `sleep()` and `wait()`? **LF FAQ**

A 45: 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. Refer **Q48** in Java section on thread communication. The call to `sleep(1000)` causes the current thread to sleep for exactly 1 second.

Q 46: How does thread synchronization occurs inside a monitor? What levels of synchronization can you apply? What is the difference between synchronized method and synchronized block? **LF CI PI FAQ**

A 46: 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). Often using a lock on a method level is too coarse. Why lock up a piece of code that does not access any shared resources by locking up an entire

method. Since each object has a lock, dummy objects can be created to implement block level synchronization. The block level is more efficient because it does not lock the whole method.

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

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

    public 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();

    public 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.

Q. Why synchronization is important? 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.

Q. What is a ThreadLocal class? *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. **For ThreadLocal example:** Refer **Q15 (What is a Session?)** in Emerging Technologies/Frameworks section. Refer **Q51** in Java section for *singleton* design pattern.

Q 47: What is a daemon thread? **LF**

A 47: 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 non-daemon 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)` ;

Q 48: How can threads communicate with each other? How would you implement a producer (one thread) and a consumer (another thread) passing data (via stack)? **LF FAQ**

A 48: The `wait()`, `notify()`, and `notifyAll()` methods are used to provide an efficient way for threads to communicate with each other. This communication solves the '**consumer-producer problem**'. This problem occurs when the producer thread is completing work that the other thread (consumer thread) will use.

Example: If you imagine an application in which one thread (the producer) writes data to a file while a second thread (the consumer) reads data from the same file. In this example the concurrent threads share the same resource file. Because these threads share the common resource file they should be synchronized. Also these two threads should communicate with each other because the consumer thread, which reads the file, should wait until the producer thread, which writes data to the file and notifies the consumer thread that it has completed its writing operation.

Let's look at a sample code where **count** is a shared resource. The consumer thread will wait inside the `consume()` method on the producer thread, until the producer thread increments the count inside the `produce()` method and subsequently notifies the consumer thread. Once it has been notified, the consumer thread waiting inside the `consume()` method will give up its waiting state and completes its method by consuming the count (i.e. decrementing the count).

Thread communication (Consumer vs Producer threads)

```
Class ConsumerProducer {

    private int count;

    public synchronized void consume(){
        while(count == 0) {
            try{
                wait()
            }
            catch(InterruptedException ie) {
                //keep trying
            }
        }
        count--; //consumed
    }

    private synchronized void produce(){
        count++;
        notify(); // notify the consumer that count has been incremented.
    }
}
```

Note: For regular classes you can use the *Observer* interface and the *Observable* class to implement the consumer/producer communications with a model/view/controller architecture. The Java programming language provides support for the Model/View/Controller architecture with two classes:

- **Observer** -- any object that wishes to be notified when the state of another object changes.
- **Observable** -- any object whose state may be of interest, and in whom another object may register an interest.

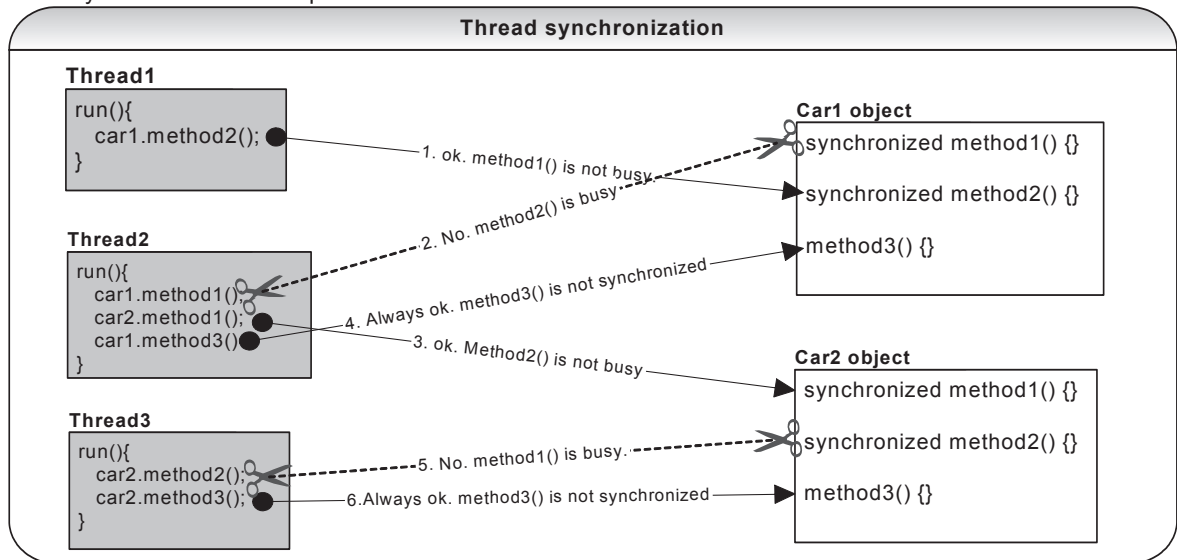
They are suitable for any system wherein objects need to be automatically notified of changes that occur in other objects. **E.g.** Your *ConfigMgr* class can be notified to reload resource properties on change to *.properties file(s).

Q. 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
    }
}
```

Q 49: If 2 different threads hit 2 different synchronized methods in an object at the same time will they both continue? **LF**

A 49: No. Only one method can acquire the lock.



Note: If your job requires deeper understanding of threads then please refer to the following articles by Allen Holub at <http://www.javaworld.com>. There are number of parts (part 1 – Part - 8) to the article entitled “**Programming Java threads in the real world**”. URLs for some of the parts are: <http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads.html>, <http://www.javaworld.com/javaworld/jw-10-1998/jw-10-toolbox.html>, etc.

Q 50: Explain threads blocking on I/O? **LF**

A 50: Occasionally threads have to block on conditions other than object locks. I/O is the best example of this. Threads block on I/O (i.e. enters the waiting state) so that other threads may execute while the I/O operation is performed. When threads are blocked (say due to time consuming reads or writes) on an I/O call inside an object's synchronized method and also if the other methods of the object are also synchronized then the object is essentially frozen while the thread is blocked.

Be sure to not synchronize code that makes blocking calls, or make sure that a non-synchronized method exists on an object with synchronized blocking code. Although this technique requires some care to ensure that the resulting code is still thread safe, it allows objects to be responsive to other threads when a thread holding its locks is blocked.

Note: The `java.nio.*` package was introduced in JDK1.4. The coolest addition is non-blocking I/O (aka NIO that stands for New I/O). Refer **Q24** in Java section for NIO.

Note: **Q51 & Q52** in Java section are very popular questions on design patterns.

Q 51: What is a **singleton** pattern? How do you code it in Java? **DP MI CO FAQ**

A 51: 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;
    }
}
```

To use it:

```
//No matter how many times you call, you get the same instance of the object.

OnlyOne myOne = OnlyOne.getInstance();
```

Note: The constructor must be explicitly declared and should have the private access modifier, so that it cannot be instantiated from outside the class. The only way to instantiate an instance of class *OnlyOne* is through the *getInstance()* method with a public access modifier.

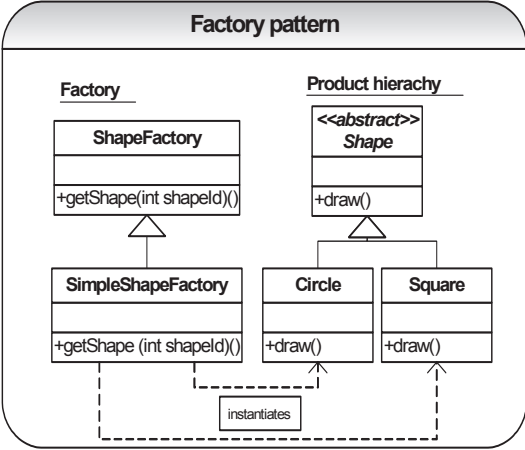
Q. When to use: Use it when only a single instance of an object is required in memory for a single point of access. For example the following situations require a **single point of access**, which gets invoked from various parts of the code.

- 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).

Q. What is the difference between a singleton class and a static class? Static class is one approach to make a class singleton by declaring all the methods as static so that you can't create any instance of that class and can call the static methods directly.

Q 52: What is a factory pattern? **DP CO FAQ**

A 52: A **Factory method pattern** (aka **Factory pattern**) is a creational pattern. The creational patterns abstract the object instantiation process by hiding how the objects are created and make the system independent of the object creation process. An **Abstract factory** pattern is one level of abstraction higher than a factory method pattern, which means it returns the factory classes.

Factory method pattern (aka Factory pattern)	Abstract factory pattern
<p>Factory for what? Factory pattern returns one of the several product subclasses. You should use a factory pattern if you have a super class and a number of subclasses, and based on some data provided, you have to return the object of one of the subclasses. Let's look at a sample code:</p>  <pre>public interface Const { public static final int SHAPE_CIRCLE = 1; public static final int SHAPE_SQUARE = 2; public static final int SHAPE_HEXAGON = 3; }</pre>	<p>An Abstract factory pattern is one level of abstraction higher than a factory method pattern, which means the abstract factory returns the appropriate factory classes, which will later on return one of the product subclasses. Let's look at a sample code:</p> <pre>public class ComplexShapeFactory extends ShapeFactory { throws BadShapeException { public Shape getShape(int shapeTypeId){ Shape shape = null; if(shapeTypeId == Const.SHAPE_HEXAGON) { shape = new Hexagon();//complex shape } else throw new BadShapeException ("shapeTypeId=" + shapeTypeId); return shape; } }</pre> <p>Now let's look at the abstract factory, which returns one of the types of ShapeFactory:</p> <pre>public class ShapeFactoryType throws BadShapeFactoryException { public static final int TYPE_SIMPLE = 1; public static final int TYPE_COMPLEX = 2; public ShapeFactory getShapeFactory(int type) { ShapeFactory sf = null; if(type == TYPE_SIMPLE) {</pre>


```

public class ShapeFactory {
    public abstract Shape getShape(int shapeId);
}

public class SimpleShapeFactory extends
    ShapeFactory throws BadShapeException {
    public Shape getShape(int shapeType){
        Shape shape = null;
        if(shapeType == Const.SHAPE_CIRCLE) {
            //in future can reuse or cache objects.
            shape = new Circle();
        }
        else if(shapeType == Const.SHAPE_SQUARE) {
            //in future can reuse or cache objects
            shape = new Square();
        }
        else throw new BadShapeException
            ("ShapeType="+ shapeType);

        return shape;
    }
}

```

Now let's look at the calling code, which uses the factory:

```
ShapeFactory factory = new SimpleShapeFactory();
```

//returns a Shape but whether it is a Circle or a Square is not known to the caller.

```
Shape s = factory.getShape(1);
s.draw(); // circle is drawn
```

//returns a Shape but whether it is a Circle or a Square is not known to the caller.

```
s = factory.getShape(2);
s.draw(); //Square is drawn
```

```

        sf = new SimpleShapeFactory();
    }
    else if (type == TYPE_COMPLEX) {
        sf = new ComplexShapeFactory();
    }
    else throw new BadShapeFactoryException("No factory!!");

    return sf;
}

```

Now let's look at the calling code, which uses the factory:

```
ShapeFactoryType abFac = new ShapeFactoryType();
ShapeFactory factory = null;
Shape s = null;
```

//returns a ShapeFactory but whether it is a SimpleShapeFactory or a ComplexShapeFactory is not known to the caller.

```
factory = abFac.getShapeFactory(1); //returns SimpleShapeFactory
```

//returns a Shape but whether it is a Circle or a Pentagon is not known to the caller.

```
s = factory.getShape(2); //returns square.
s.draw(); //draws a square
```

//returns a ShapeFactory but whether it is a SimpleShapeFactory or a ComplexShapeFactory is not known to the caller.

//returns a Shape but whether it is a Circle or a Pentagon is not known to the caller.

```
s = factory.getShape(3); //returns a pentagon.
s.draw(); //draws a pentagon
```

Q. Why use factory pattern or abstract factory pattern? Factory pattern returns an instance of several (product hierarchy) subclasses (like **Circle**, **Square** etc), but the calling code is unaware of the actual implementation class. The calling code invokes the method on the interface for example **Shape** and using polymorphism the correct draw() method gets invoked [Refer **Q10** in Java section for polymorphism]. So, as you can see, the factory pattern reduces the coupling or the dependencies between the calling code and called objects like Circle, Square etc. This is a very powerful and common feature in many frameworks. You do not have to create a new **Circle** or a new **Square** on each invocation as shown in the sample code, which is for the purpose of illustration and simplicity. In future, to conserve memory you can decide to cache objects or reuse objects in your factory with no changes required to your calling code. You can also load objects in your factory based on attribute(s) read from an external properties file or some other condition. Another benefit going for the factory is that unlike calling constructors directly, factory patterns have more meaningful names like getShape(...), getInstance(...) etc, which may make calling code more clear.

Q. Can we use the singleton pattern within our factory pattern code? Yes. Another important aspect to consider when writing your factory class is that, it does not make sense to create a new factory object for each invocation as it is shown in the sample code, which is just fine for the illustration purpose.

```
ShapeFactory factory = new SimpleShapeFactory();
```

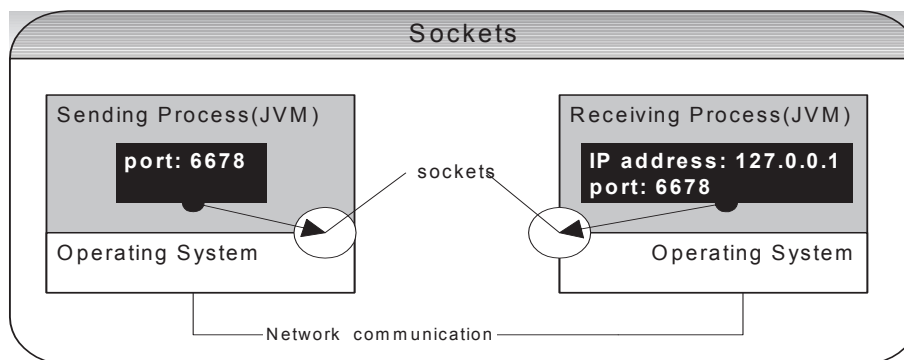
To overcome this, you can incorporate the singleton design pattern into your factory pattern code. The singleton design pattern will create only a single instance of your **SimpleShapeFactory** class. Since an abstract factory pattern is unlike factory pattern, where you need to have an instance for each of the two factories (i.e. **SimpleShapeFactory** and **ComplexShapeFactory**) returned, you can still incorporate the singleton pattern as an access point and have an instance of a **HashMap**, store your instances of both factories. Now your calling method uses a static method to get the same instance of your factory, hence conserving memory and promoting object reuse:

```
ShapeFactory factory = ShapeFactory.getFactoryInstance(); //returns a singleton
factory.getShape();
```

Note: Since questions on singleton pattern and factory pattern are commonly asked in the interviews, they are included as part of this section. To learn more about design patterns refer **Q11, Q12** in How would you go about section...?

Q 53: What is a socket? How do you facilitate inter process communication in Java? **LF**

A 53: 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 first number is a 32-bit (IPV4 or 128-bit is IPV6) integer called the Internet Address (or **IP address**). The second number is a 16-bit integer called the **port** of the socket. The IP address is the location of the machine, which you are trying to connect to and the port number is the port on which the server you are trying to connect is running. The port numbers 0 to 1023 are reserved for standard services such as e-mail, FTP, HTTP etc.



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.

Q 54: How will you call a Web server from a stand alone Java application/Swing client/Applet? **LF**

A 54: Using the **java.net.URLConnection** and its subclasses like HttpURLConnection and JarURLConnection.

URLConnection	HttpClient (i.e. a browser)
Supports HEAD, GET, POST, PUT, DELETE, TRACE and OPTIONS	Supports HEAD, GET, POST, PUT, DELETE, TRACE and OPTIONS.
Does not support cookies.	Does support cookies.
Can handle protocols other than http like ftp, gopher, mailto and file.	Handles only http.

```
public class TestServletWriter {
    public static void main(String[] args) throws Exception{
        String host = "localhost"; //i.e 127.0.0.1
        String protocol = "http"; //request/response paradigm
        int port = 18080;
        String strURL = protocol + "://" + host + ":" + port + "/myRootContext/myServlet";
        java.net.URL servletURL = new java.net.URL(strURL);

        java.net.URLConnection con = servletURL.openConnection();
        con.setDoInput(true);
        con.setDoOutput(true);
        con.setUseCaches(false);
        con.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");

        // Write the arguments as post data
        ObjectOutputStream out = new ObjectOutputStream(con.getOutputStream());

        out.writeObject("Hello Servlet"); //write a serializable object to the servlet.
        out.flush();
        out.close();
    }
}
```