SECTION ONE

Java – Interview questions & answers

KEY AREAS

- 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
- Coding[1] CO

Popular Questions: Q01, Q04, Q07, Q08, Q13, Q16, Q17, Q18, Q19, Q25, Q27, Q29, Q32, Q34, Q40, Q45, Q46, Q50, Q51, Q53, Q54, Q55, Q63,Q64, Q66, **Q67**

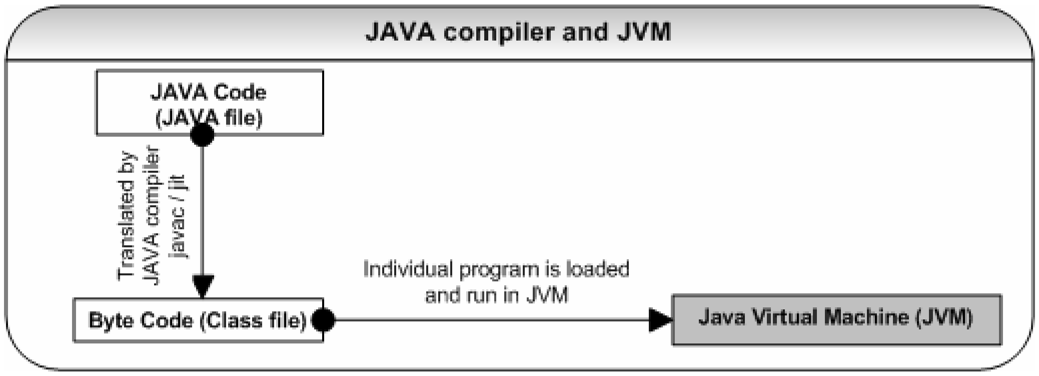
1 Unlike other key areas, the CO is not always shown against the question but shown above the actual subsection of relevance within a question.

Java – Language Fundamentals

Q 01: Give a few reasons for using Java? LF DC
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 standardised APIs (Application Program 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.


JAVA compiler and JVM
JAVA Code (JAVA file)
Translated by JAVA compiler javac / jit
Byte Code (Class file)
Individual program is loaded and run in JVM
Java Virtual Machine (JVM)

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**) -

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 **Q 8** and **Q 10** in Java section.]
- Java does not support destructors but rather 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 collection framework – Refer **Q14**, **Q15** in Java section).

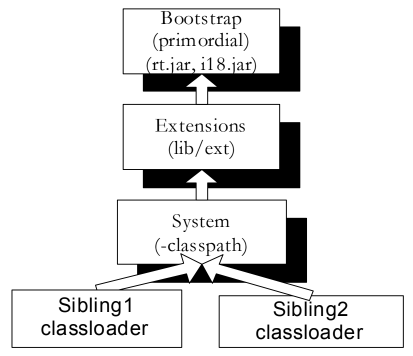
- All the code in Java program is encapsulated within classes therefore Java does not have global variables or functions.
- C++ requires explicit memory management, while Java includes automatic garbage collection. [Refer **Q32** in Java section].

Q 04: Explain Java class loaders? Explain dynamic class loading? LF

A 04: 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 specially loaded with the help of *static main()* method declared in your class. All the subsequently loaded classes are loaded by the classes, which are already loaded and running. 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, *java.** packages. (as defined in the sun.boot.class.path system property, typically loads rt.jar and i18n.jar)
Extensions	No	Loads jar files from JDK extensions directory (as defined in the java.ext.dirs system property – usually lib/ext directory of the JRE)
System	No	Loads classes from system classpath (as defined by the java.class.path property, which is set by the **CLASSPATH** environment variable or –classpath or –cp command line options)

JVM class loaders



Classes loaded by Bootstrap class loader have no visibility into classes loaded by its descendants (ie Extensions and Systems class loaders).

The classes loaded by system class loader have visibility into classes loaded by its parents (ie Extensions and Bootstrap class loaders).

If there were any sibling class loaders they cannot see classes loaded by each other. They can only see the classes loaded by their parent class loader. For example Sibling1 class loader cannot see classes loaded by Sibling2 class loader

Both Sibling1 and Sibling2 class loaders have visibilty into classes loaded by their parent class loaders (eg: System, Extensions, and Bootstrap)

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.

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 **Q45** in Java section for singleton design pattern]

Explain static vs. dynamic class loading?

Static class loading	Dynamic class loading

Classes are statically loaded with Java's "new" operator.
 	
`class MyClass {`
`   public static void main(String args[]) {`
`      Car c = new Car();`
`   }`
`}`
	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.
	
	Class.forName (String *className*); //static method which returns a Class
	
	The above static method returns the class object associated with the class name. The string *className* can be supplied dynamically at run time. Unlike the static loading, the dynamic loading will decide whether to load the class *Car* or the class *Jeep at* runtime based on a properties file and/or other runtime

	conditions. Once the class is dynamically loaded the following method returns an instance of the loaded class. It's just like creating a class object with no arguments.
	
	class.newInstance (); //A non-static method, which creates an instance of a class (i.e. creates an object).
	
	Jeep myJeep = null ;
	//myClassName should be read from a properties file or Constants interface.
	//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);

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: ▪ The forName(..) method in class - ***Class***. ▪ The findSystemClass(..) method in class - ***ClassLoader***. ▪ The loadClass(..) method in class - ***ClassLoader***.

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 StaticInitilaizer {
  public static final int A = 5;
  public static final int B;

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

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

  public StaticInitilaizer(){}  // 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 =" + StaticInitilaizer.A + ", B =" + StaticInitilaizer.B);
}
```

Q 05: What are the advantages of Object Oriented Programming Languages (OOPL)? **DC**
A 05: 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 **P**olymorphism**, I**nheritance and **E**ncapsulation are the **3 pillars** of OOPL]

Q 06: How does the Object Oriented approach improve software development? **DC**
A 06: The key benefits are:

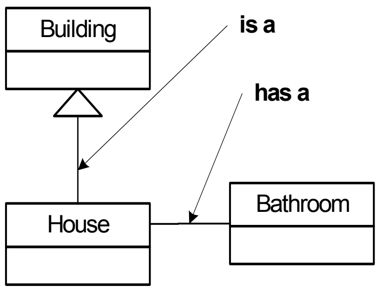
- ***Re-use*** of previous work: using **implementation inheritance** and **object composition**.
- ***Real mapping to the problem domain:*** Objects map to real world and represent vehicles, customers, products etc: with **encapsulation**.
- ***Modular Architecture:*** Objects, systems, frameworks etc 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 07: 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

A 07: 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 a] Vs Composition [has a]



is a [House is a Building]

```
class Building{
 .......
}

class House extends Building{
 .........
}
```

has a [House has a Bathroom]

```
class House {
    Bathroom room = new Bathroom() ;
    ....
   public void getTotMirrors(){
       room.getNoMirrors();
       ....
   }
}
```

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.

Which one to use? 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 **Q8** 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 deleted then all corresponding line items for that order should be deleted. So **composition has a stronger relationship**.

Q 08: What do you mean by polymorphism, inheritance, encapsulation, and dynamic binding? DC

A 08: ***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:

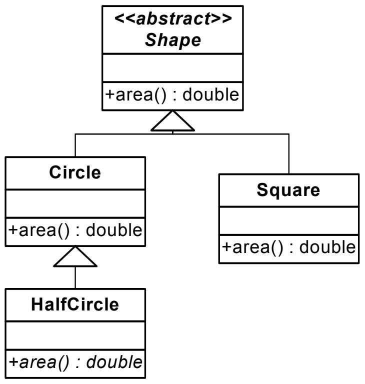
```
//client or calling code
double dim = 5.0; //ie 5 meters radius or width
List listShapes = new ArrayList(20);

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 ;
}
```

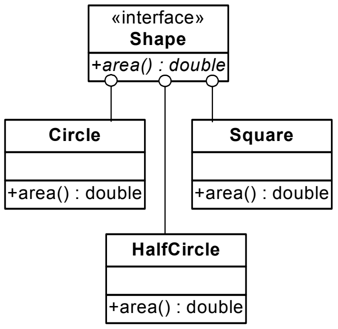
For example: given a base class/interface *Shape*, polymorphism allows the programmer to define different area(double dim1) methods for any number of derived classes such as *Circle*, *Square* etc. No matter what shape an object is, applying the area method to it will return the right results.

Later on *HalfCicle* 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 behaviour (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 applications' 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 behaviour of the superclass, not on the actual implementation**. For example in the above diagram the subclasses should only be concerned about the behaviour 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 favours interface inheritance to implementation inheritance. **CO**

Implementation inheritance

Let's assume that savings account and term deposit account have a similar behaviour in terms of depositing and withdrawing money, so we will get the super class to implement this behaviour and get the subclasses to reuse this behaviour. But saving account and term deposit account have specific behaviour in calculating the interest.

```
public abstract class Account {

  public void deposit(double amount) {
    //deposit logic
  }

  public void withdraw(double amount) {
    //withdraw logic
  }

  public abstract double calculateInterest(double  amount);

}
```

```
public class SavingsAccount extends Account {

   public double calculateInterest(double amount)  {
       //calculate interest for SavingsAccount
  }
}
```

```
public class TermDepositAccount extends Account {

    public double calculateInterest(double amount) {
       //calculate interest for TermDeposit
    }
 }
```

The calling code can be defined as follows for illustration purpose only:

```
public class Test {
  public static void main(String[] args) {
    Account acc1 = new SavingsAccount();
    acc1.deposit(5.0);
    acc1.withdraw(2.0);

    Account acc2 = new TermDepositAccount();
    acc2.deposit(10.0);
    acc2.withdraw(3.0);

    acc1.calculateInterest(500.00);
    acc2.calculateInterest(500.00);
  }
}
```

Interface inheritance

Let's look at an **interface inheritance** code sample, which makes use of **composition** for reusability. In the following example the methods deposit(…) and withdraw(…) share the same piece of code in *AccountHelper* class. The method calculateInterest(…) has its specific implementation in its own class.

```
public interface Account {
    public abstract void deposit(double amount);
    public abstract void withdraw(double amount);
    public abstract int getAccountType();
}

public interface SavingsAccount extends Account{
   public abstract double calculateInterest(double amount);
}

public interface TermDepositAccount extends Account{
   public abstract double calculateInterest(double amount);
}
```

The classes **SavingsAccountImpl, TermDepositAccountImpl** should implement the methods declared in its interfaces. The class **AccountHelper** implements the methods deposit(…) and withdraw(…)

```
public class SavingsAccountImpl implements SavingsAccount{
    private int accountType = 1;

   //helper class which promotes code reuse through composition
   AccountHelper helper = new AccountHelper();

   public void deposit(double amount) {
         helper.deposit(amount, getAccountType());
   }
   public void withdraw(double amount) {
         helper.withdraw(amount, getAccountType());
   }
   public double calculateInterest(double amount) {
         //calculate interest for SavingsAccount
    }
   public int getAccountType(){
        return accountType;
   }
}
```

```
public class TermDepositAccountImpl implements
                                        TermDepositAccount {
     private int accountType = 2;

    //helper class which promotes code reuse through composition
    AccountHelper helper = new AccountHelper();

    public void deposit(double amount) {
          helper.deposit(amount, getAccountType());
    }
    public void withdraw(double amount) {
          helper.withdraw(amount, getAccountType());
    }
    public double calculateInterest(double amount) {
             //calculate interest for TermDeposit
     }
    public int getAccountType() {
         return accountType;
    }
}
```

The calling code can be defined as follows for illustration purpose only:

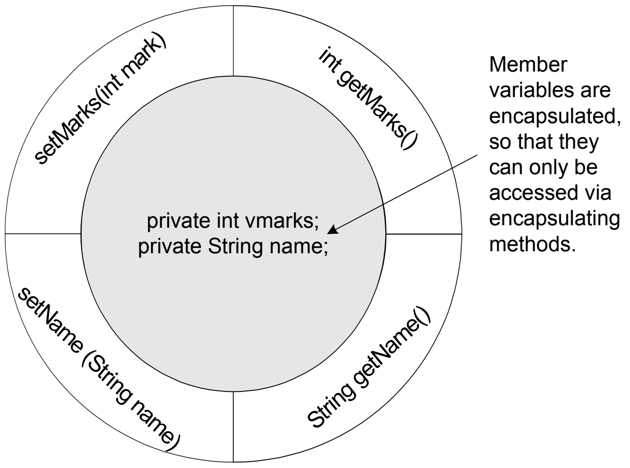
```
public class Test {
    public static void main(String[] args) {
        Account acc1 = new SavingsAccountImpl();
        acc1.deposit(5.0);

        Account acc2 = new TermDepositAccountImpl();
        acc2.deposit(10.0);

        if (acc1.getAccountType() == 1) {
           ((SavingsAccount) acc1).calculateInterest(500.00);
        }

        if (acc2.getAccountType() == 2) {
              ((TermDepositAccount) acc2).calculateInterest(500.00);
        }
     }
}
```

Encapsulation – refers to keeping all the related members (variables and methods) together in an object. Specifying members 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.

Being able to encapsulate members of a class is important for **security** and **integrity**. We can protect variables from unacceptable values. The sample code below 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. **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.
}
```



Q 09: What is design by contract? Explain the *assertion* construct? **DC**

A 09: 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 **Q34** 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

Preconditions on *public* methods are enforced by explicit checks that throw particular, specified exceptions. You **should not use *assertion* to check the parameters of the public methods** but can use for the non-public methods. ***Assert*** 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, assert construct does not throw an exception of a specified type. It can throw only an *AssertionError*.
 	
`public void setRate(int rate) {`
`  if(rate <= 0 || rate > MAX_RATE){`
`      throw new IllegalArgumentException("Invalid rate → " + rate);`
`  }`
`  setCalculatedRate(rate);`
`}`
	You can use assertion to check the parameters of the non-public methods.
	
	`private void setCalculatedRate(int rate) {`
	`  assert (rate > 0 && rate < MAX_RATE) : rate;`
	`     //calculate the rate and set it.`
	`}`
	
	Assertions can be disabled, so programs must not assume that assert construct will be always executed:
	
	`//Wrong: if assertion is disabled, CarpenterJob never`
	`//Get removed`
	`assert  jobsAd.remove(PilotJob);`
	
	`//Correct:`
	`boolean pilotJobRemoved =  jobsAd.remove(PilotJob);`
	`assert pilotJobRemoved;`

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;   //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 errors' 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.

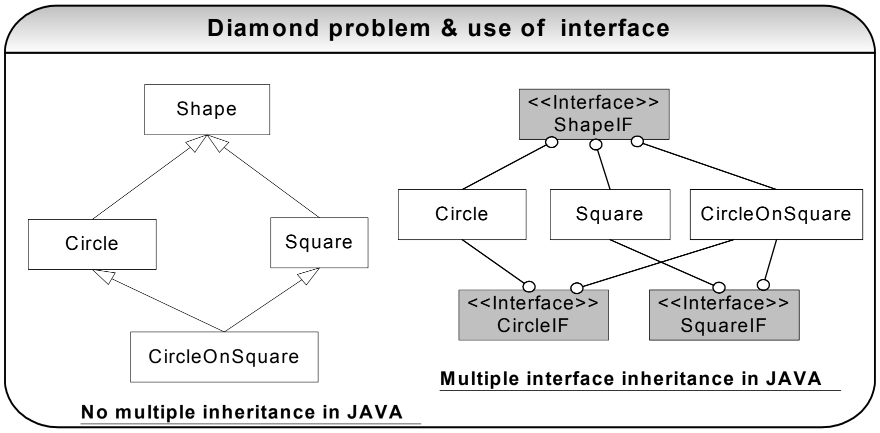
```
Java -ea[:packagename...|:classname] or Java -enableassertions[:packagename...|:classname]
Java –ea:Account
```

Q 10: What is the difference between an abstract class and an interface and when should you use them? **LF** **DP** **DC**

A 10: 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 behaviour), 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.
Can have instance variables, constructors and any visibility: public, private, protected, none (aka package).	Cannot have instance variables, constructors and can have only public and none (aka package) visibility.

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 behaviour and force subclasses to provide any specific behaviour. Care should be taken not to overuse implementation inheritance as discussed in **Q8** in Java section.

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**. 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 inheritances. 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 11: Why there are some interfaces with no defined methods (i.e. marker interfaces) in Java? LF
A 11: 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*** Serializable (Refer **Q19** in Java section), Cloneable etc

---

Q 12: When is a method said to be overloaded and when is a method said to be overridden? LF CO
A 12:

Method Overloading	Method Overriding

Overloading deals with multiple methods in the same class with the same name but different method signatures.
 	
`class MyClass {`
`    public void getInvestAmount(int rate) {…}`
 	
`    public void getInvestAmount(int rate, long principal)`
`    { … }`
`}`
 	
Both the above methods have the same method names but different method signatures, which mean the methods are overloaded.
	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.
	
	`class BaseClass{`
	`    public void getInvestAmount(int rate) {…}`
	`}`
	
	`class MyClass extends BaseClass {`
	`    public void getInvestAmount(int rate) { …}`
	`}`
	
	Both the above methods have the same method names and the signatures but the method in the subclass *MyClass* overrides the method in the superclass *BaseClass*.

Overloading lets you define the **same operation in different ways for <u>different data</u>**.	Overriding lets you define the **same operation in different ways for <u>different object types</u>**.

Q 13: What is the main difference between an ArrayList and a Vector? What is the main difference between Hashmap and Hashtable? LF DC PI CI
A 13:

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

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 c*ollection* API's external synchronization facility as shown below to **temporarily synchronize** your collections as needed: CO

```
Map myMap = Collections.synchronizedMap (myMap);
List  myList = Collections.synchronizedList (myList);
```

Java arrays are even faster than using an *ArrayList/Vector* and perhaps therefore may be preferable. *ArrayList/Vector* internally uses an array with some convenient methods like add(..), remove(…) etc.

Q 14: Explain the Java Collection framework? LF DP

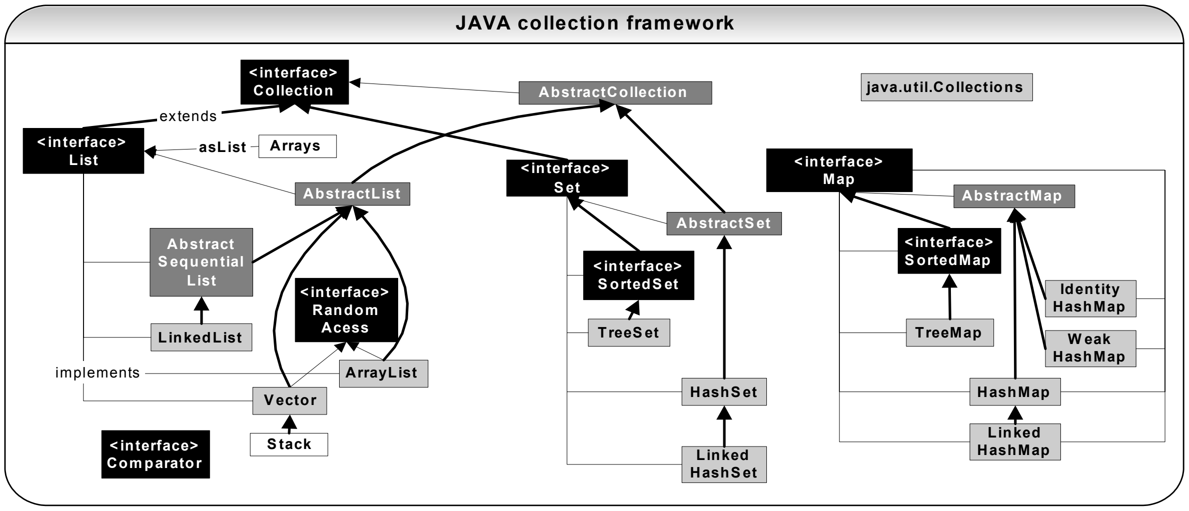
A 14: The key interfaces used by the collection framework are ***List****,* ***Set*** and ***Map***. The *List* and *Set* extends the ***Collection*** interface. Should not confuse the *Collection* interface with the ***Collections*** class which is a utility class.

A ***Set*** is a collection with unique elements and prevents duplication within the collection. ***HashSet*** and ***TreeSet*** are implementations of a *Set* interface. A ***List*** is a collection with an ordered sequence of elements and may contain duplicates. ***ArrayList*****,** ***LinkedList*** and ***Vector*** are implementations of a *List* interface.

The Collection 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.

How to implement collection ordering? ***SortedSet*** and ***SortedMap*** interfaces maintain sorted order. The classes, which implement the ***Comparable*** interface, impose natural order. 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.

Design pattern: **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 behavioural design pattern that provides a way to access elements of a collection sequentially without exposing its internal representation.



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

What are the benefits of the Java collection framework? Collection 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.
- **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 15: What are some of the best practices relating to Java collection? **BP** **PI** **CI**

A 15:

- Use ArrayLists, 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);
```

```
List  myList = Collections.synchronizedList (myList);
```

- 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 minimise **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**: 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();
```

- **Avoid storing unrelated or different types of objects into same collection**: This is analogous to storing items in pigeonholes without any labelling. To store items use **value objects** or **data objects** (as oppose to storing every attribute in an ArrayList or HashMap). Provide wrapper classes around your collection 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 **Q52** in Java section for UML diagram of a composite design pattern. **CO**

Avoid where possible

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.

```
List myOrder = new ArrayList()

ResultSet rs = …

While (rs.hasNext()) {

    List lineItem = new ArrayList();

    lineItem.add (new Integer(rs.getInt("itemId")));
    lineItem.add (rs.getString("description"));
    ….
    myOrder.add( lineItem);
}

return myOrder;
```

Example 2:

```
List myOrder =  new ArrayList(10);

//create an order
OrderVO header = new OrderVO();
header.setOrderId(1001);
…
//add all the line items
LineItemVO line1 = new LineItemVO();
line1.setLineItemId(1);
LineItemVO line2 = new LineItemVO();
Line2.setLineItemId(2);
```

Better approach

When storing items into a collection define value objects as shown below: (**VO** is an acronym for **V**alue **O**bject).

```
public class LineItemVO {
  private int itemId;
  private String productName;

  public int getLineItemId(){return accountId ;}
  public int getAccountName(){return accountName;}

  public void setLineItemId(int accountId ){
    this.accountId = accountId
  }
   //implement other getter & setter methods
}
```

Now let's define our base wrapper class, which represents an order:

```
public abstract class Order {
      int orderId;
      List lineItems = null;

      public abstract int countLineItems();
      public abstract boolean add(LineItemVO itemToAdd);
      public abstract boolean remove(LineItemVO itemToAdd);
      public abstract Iterator  getIterator();
      public int  getOrderId(){return this.orderId; }
}
```

Now a specific implementation of our wrapper class:

```
public class OverseasOrder extends Order {
      public OverseasOrder(int inOrderId) {
            this.lineItems = new ArrayList(10);
            this.orderId = inOrderId;
      }
```

```
List lineItems = new ArrayList();
lineItems.add(line1);
lineItems.add(line2);

//to store objects
myOrder.add(order);// index 0 is an OrderVO object
myOrder.add(lineItems);//index 1 is a List of line items

//to retrieve objects
myOrder.get(0);
myOrder.get(1);
```

Above approaches are bad because disparate objects are stored in the **lineItem** 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 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 labelling and just relying on its grid position.

```
    public int countLineItems() { //logic to count }

    public boolean add(LineItemVO itemToAdd){
        …//additional logic or checks
        return  lineItems.add(itemToAdd);
    }

    public boolean remove(LineItemVO itemToAdd){
        return  lineItems.remove(itemToAdd);
    }

    public ListIterator getIterator(){ return lineItems.Iterator();}
}
```

Now to use:

```
Order myOrder = new OverseasOrder(1234) ;

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 16: 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

A 16: 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 colour 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. ***Points to consider:***

- 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.

Note: Java 1.5 introduces enumerated constants, which improves readability and maintainability of your code. Java programming language enums are more powerful than their counterparts in other languages. E.g. 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()) {
            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 17: What is the main difference between a String and a StringBuffer class? LF PI CI CO
A 17:

String	StringBuffer / StringBuilder
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 synchronised, which makes it slightly faster at the cost of not being thread-safe.

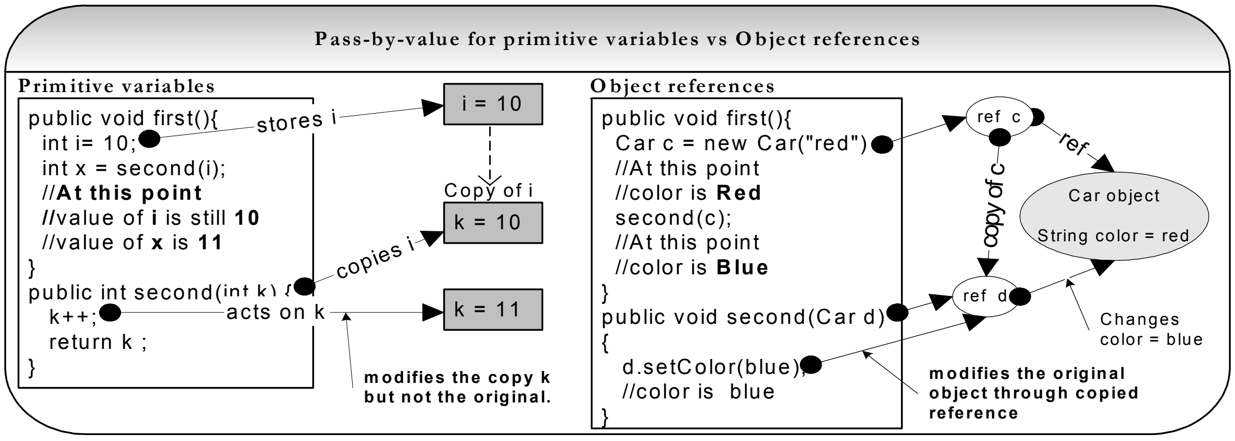
//Inefficient version using immutable String
String output = "Some text"
Int count = 100;
for(int I =0; i<count; i++) {
   output += i;
}
return output;
	//More efficient version using mutable StringBuffer
	StringBuffer output = new StringBuffer(**110)**;
	Output.append("Some text");
	for(int I =0; i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

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

Another important point is that creation of extra strings is not limited to 'overloaded mathematical operators' ("+") 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 18: What is the main difference between pass-by-reference and pass-by-value? LF PI
A 18: 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**.



Pass-by-value for primitive variables vs Object references

Primitive variables

```
public void first(){
  int i= 10;
  int x = second(i);
  //At this point
  //value of i is still 10
  //value of x is 11
}
public int second(int k) {
   k++;
   return k ;
}
```

Object references

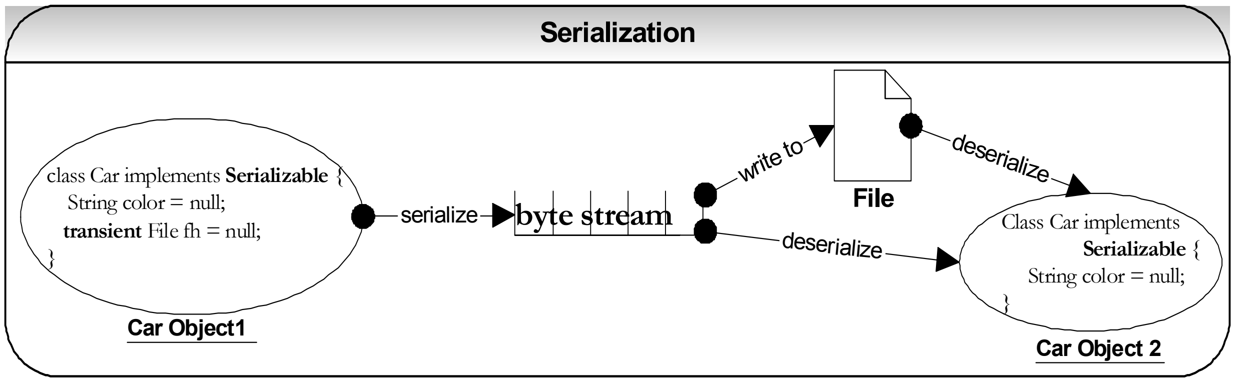
```
public void first(){
  Car c = new Car("red")
  //At this point
  //color is Red
  second(c);
  //At this point
  //color is Blue
}
public void second(Car d)
{
   d.setColor(blue);
   //color is  blue
}
```

If your method call involves inter-process (e.g. between two JVMs) communication, then the reference of the calling method has a different address space to the called method sitting in a separate process (i.e. separate JVM). Hence inter-process communication involves calling method passing objects as arguments to called method by-value in a serialized form, which can adversely affect performance due to marshalling and unmarshalling 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 19: 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? LF SI PI

A 19: 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 or a database connection. 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.

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 **Q22** in Java section).

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 20: Explain the Java I/O streaming concept and the use of the decorator design pattern in Java I/O? LF DP PI SI

A 20: 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.

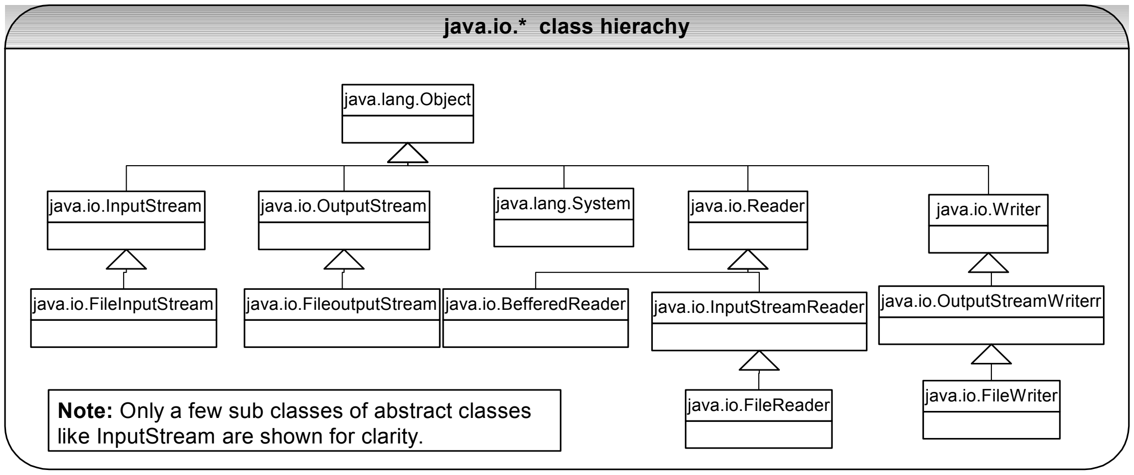
- Byte streams (8 bit bytes) → Abstract classes are: **InputStream** and **OutputStream**
- Character streams (16 bit UNICODE) → Abstract classes are: **Reader** and **Writer**

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 behaviour 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.
	
	File **file** = new File("c:/temp");
	FileInputStream **fis** = new FileInputStream(**file**);
	BufferedInputStream bis = new BufferedInputStream(**fis**);
	
	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.



The New I/O (NIO): more scalable 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 and CPU context switching**. 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 *InputStrean* 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.

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.

A **Selector** class is responsible for multiplexing (combining multiple streams into a single stream) by allowing a single thread to service multiple channels. Each *Channel* registers events with a *Selector*. When events arrive from clients, the Selector demultiplexes (separating a single stream into multiple streams) them and dispatches the events to corresponding Channels. To achieve non-blocking I/O a *Channe*l class must work in conjunction with a *Selector* class.

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 21: How can you improve Java I/O performance? $\boxed{\text{PI}}$ $\boxed{\text{BP}}$

A 21: Java applications that utilise 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

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

Let us look at some of the techniques to improve I/O performance. $\boxed{\text{CO}}$

- Use **buffering** to minimise disk access and underlying operating system. As shown below, with buffering large chunks of a file are read from a disk and then accessed a byte or character at a time.

Without buffering : inefficient code

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

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

With Buffering: yields better performance

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

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

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 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);
System.setOut(ps);

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

It is recommended to use logging frameworks like **Log4J** or **apache commons logging**, which uses buffering instead of using default behaviour of **System.out.println(…..)** for better performance. Frameworks like Log4J are configurable, flexible, extensible and easy to use.

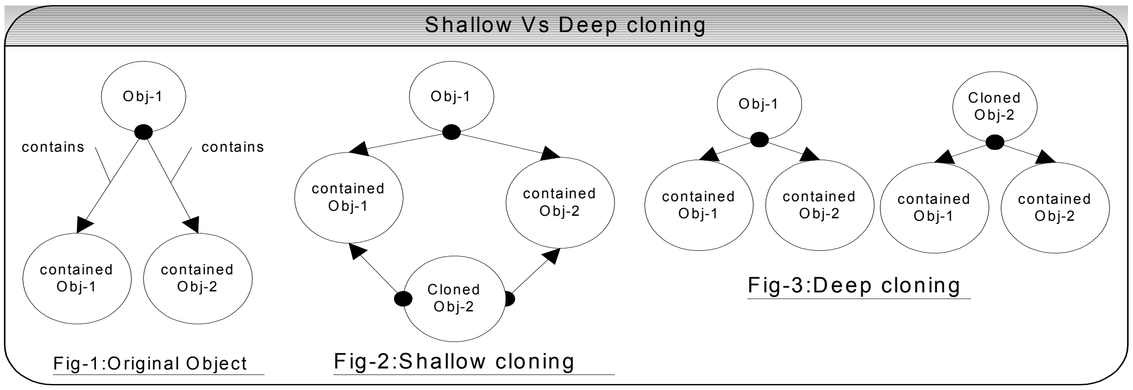
- Use the NIO package, if you are using JDK 1.4 or later, which uses performance-enhancing features like buffers to hold data, memory mapping of files, non-blocking I/O operations etc.
- I/O performance can be improved by minimising the calls to the underlying operating systems. The Java runtime itself 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 an ArrayList or a HashMap and subsequently access the data from an in-memory collection instead of the disk.

Q 22: What is the main difference between shallow cloning and deep cloning of objects? **DC** **LF** **MI** **PI**

A 22: The default behaviour 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 *HashMap* returns a shallow copy of *HashMap* instance, which means **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 *ByteArrayOutputSream* and then deserialize it. This creates a deep copy but does require that all keys and values in the *HashMap* are Serializable. Its primary advantage is that it will deep copy any arbitrary object graph.

List some of the methods supported by Java object class? clone(), toString(), equals(Object obj), hashCode() → refer **Q16** in Java section, wait(), notify() → refer **Q42** in Java section, finalize() etc.

Q 23: What is the difference between an instance variable and a static variable? Give an example where you might use a static variable? **LF**

A 23:

Static variable	Instance variable
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 initialised.	Instance variables are non-static and there is one occurrence of an instance variable in each class instance (i.e. each object).

A static variable is used in the **singleton** pattern. (Refer **Q45** in Java section). A static variable is used with a **final** modifier to define **constants**.

Q 24: Give an example where you might use a static method? **LF**

A 24: Static methods prove useful for creating **utility classes, singleton classes** and **factory methods** (Refer **Q45**, **Q46** 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 25: What are access modifiers? LF

A 25:

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 26: Where and how can you use a private constructor? LF

A 26: Private constructor is used if you do not want other classes to instantiate the object. The instantiation is done by a public static method within the same class.

- Used in the singleton pattern. (Refer **Q45** in Java section).
- Used in the factory method pattern (Refer **Q46** in Java section).
- Used in utility classes e.g. StringUtils etc.

Q 27: What is a final modifier? Explain other Java modifiers? LF

A 27: A final class can't be extended i.e. A final class may 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	Property
static	A static inner class is just an inner class associated with the class, rather than with an instance.	cannot be instantiated, are called by classname.method, can only access static variables	Only one instance of the variable exists.
abstract	Cannot be instantiated, must be a superclass, used whenever one or more methods are abstract.	Method is defined but contains no implementation code (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 **lock on the class for static methods**. Acquires a **lock on the instance for non-static** methods.	N/A
transient	N/A	N/A	Field should not be serialized.
final	Class cannot be inherited	Method cannot be overridden	Makes the variable a constant.
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 member variables that may be modified simultaneously by other threads. Since other threads cannot see local variables, there is no need to mark local variables as volatile. E.g. **volatile** int number; **volatile** private List listItems = null; etc. The modifier volatile only synchronizes the variable marked as volatile whereas "synchronized" modifier synchronizes all variables.

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 28: What is the difference between final, finally and finalize() in Java? LF

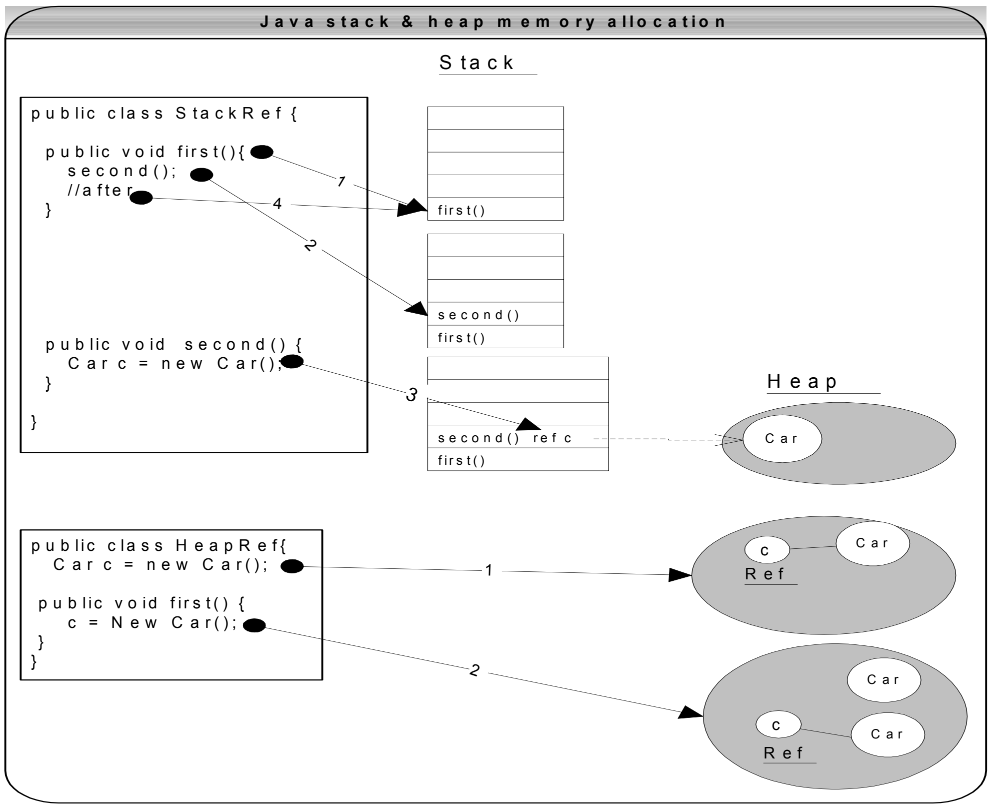
A 28:

- **final** - constant declaration. Refer **Q27** 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.

Q 29: How does Java allocate stack and heap memory? Explain re-entrant, recursive and idempotent methods/functions? MI CI

A 29: 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**, if they are local method variables and in the **heap** if they are member variables (i.e. fields of a class). In Java methods 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 threadsafe (each thread will have its own stack) but the heap is not threadsafe unless guarded with synchronisation through your code.

A method in stack is **re-entrant** allowing multiple concurrent invocations that do not interfere with each other. A function is **recursive** if it calls itself. Given enough stack space, recursive method calls are perfectly valid in Java though it is tough to debug. Recursive functions 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.



Q 30: Explain Outer and Inner classes (or Nested classes) in Java? When will you use an Inner Class? LF

A 30: 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.

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. $\boxed{\text{CO}}$

Member inner class	**Anonymous inner class**

```
public class MyStack {
   private Object[] items = null;
   …
   public Iterator iterator() {
      return new StackIterator();
   }
   //inner class
   class StackIterator implements Iterator{
       …
       public boolean hasNext(){…}
   }
}
```
	```
	public class MyStack {
	   private Object[] items = null;
	   …
	   public Iterator iterator() {
	      return new Iterator {
	          …
	          public boolean hasNext() {…}
	      }
	   }
	}
	```

Explain outer and inner classes?

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

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{ }
			}
			
			$\boxed{\text{Outside.class ,Outside\$Inside.class}}$

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(){}
			}
			
			$\boxed{\text{Outside.class , Outside\$Inside.class}}$

Inner class	Local class	Defined within a block of code. Can use final local variables and final method parameters. Only visible within the block of code that defines it.	class Outside {
			  void first() {
			    final int i = 5;
			    class Inside{}
			  }
			}
			
			$\boxed{\text{Outside.class , Outside\$1\$Inside.class}}$

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 event model.	class Outside{
			  void first() {
			    button.addActionListener **(new ActionListener()**
			    {
			    public void actionPerformed(ActionEvent e) {
			       System.out.println("The button was pressed!");
			       }
			    });
			  }
			}
			
			$\boxed{\text{Outside.class , Outside\$1.class}}$

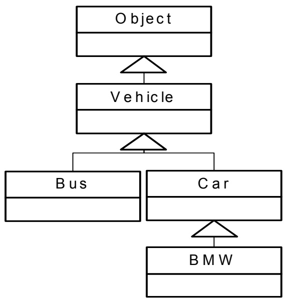
Q 31: What is type casting? Explain up casting vs. down casting? When do you get ClassCastException? LF DP
A 31: Type casting means treating a variable of one type as though it is another type.

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**.

Upcasting vs Downcasting Objects



```
Vehicle v1 = new Car();       //Right.upcasting or implicit casting
Vehicle v2 = new Vehicle();

Car c0 = v1;                  //Wrong. compile time error "Type Mismatch".
                              //Explicit or down casting is required
Car c1 = (Car)v1;             // Right. down casting or explicit casting.
                              // v1 has knowledge of Car due to line1

Car c2 = (Car)v2;             //Wrong. Runtime exception ClassCastException
                              //v2 has no knowledge of Car.
Bus b1 = new BMW();           //Wrong. compile time error "Type Mismatch"
Car c3 = new BMW();           //Right.upcasting or implicit casting

Car c4 = (BMW)v1;             //Wrong. Runtime exception ClassCastException
Object o = v1;                //v1 can only be upcast to its parent or
Car c5 = (Car)v1;             //v1 can be down cast to Car due to line 1.
```

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. We 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;
}
```

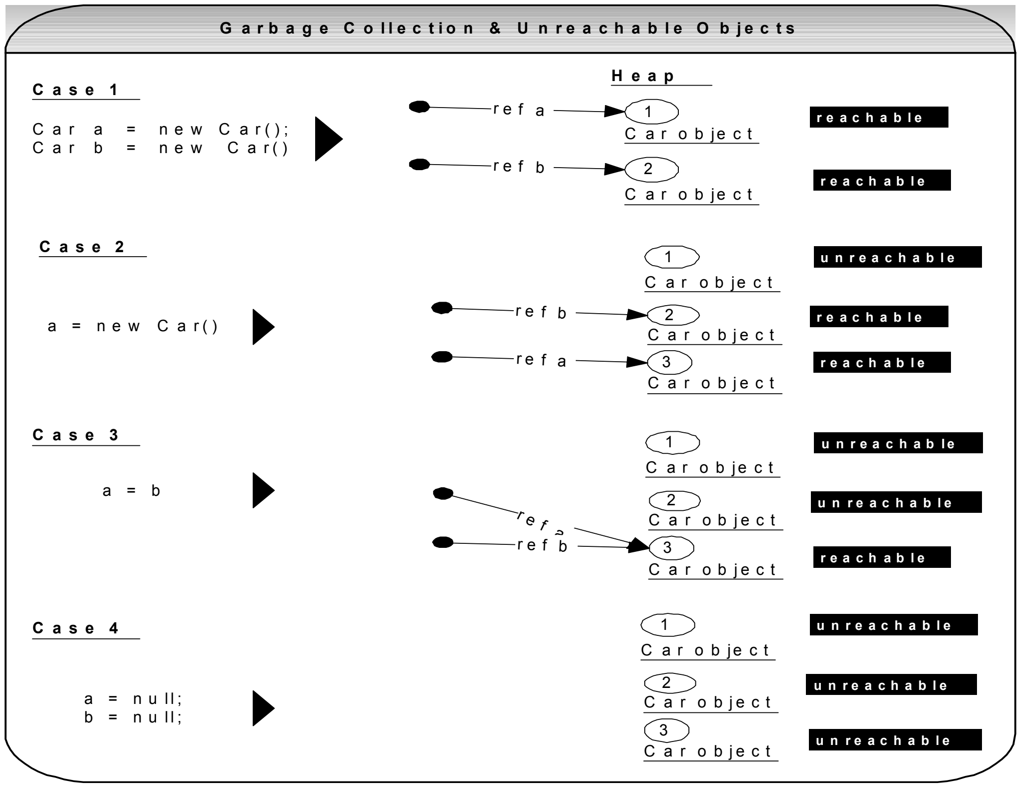
Design pattern: The “**instanceof**” and “**typecast”** constructs are shown for the illustration purpose only. Using these constructs can be unmaintainable due to large if and elseif statements and can affect performance if used in frequently accessed methods or loops. Look at using **visitor design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

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

Q 32: What do you know about the Java garbage collector? When does the garbage collection occur? Explain different types of references in Java? LF MI

A 32: 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 (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.

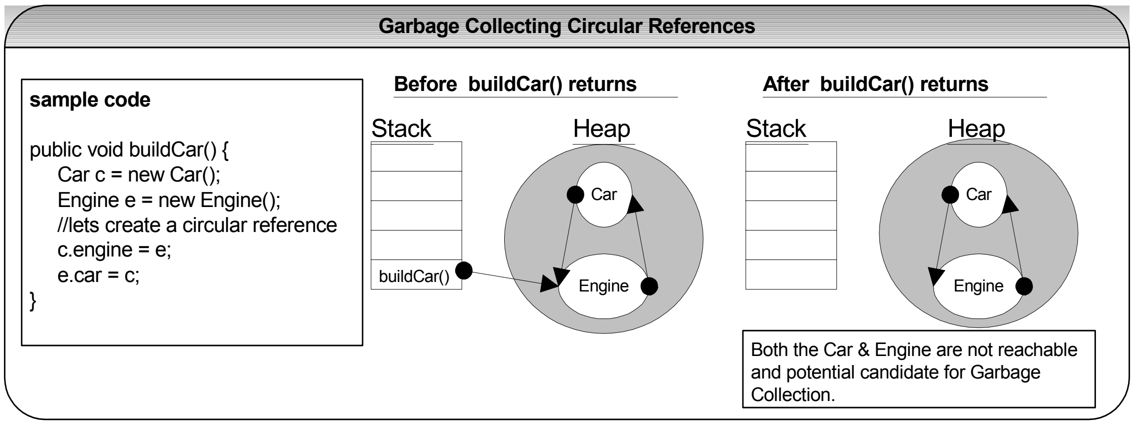
-verbose:gc reports on each garbage collection event.

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 33: 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 33: Yes. Refer diagram below.


Garbage Collecting Circular References
Before buildCar() returns
After buildCar() returns
Stack
Heap
buildCar()
Car
Engine
Both the Car & Engine are not reachable and potential candidate for Garbage Collection.

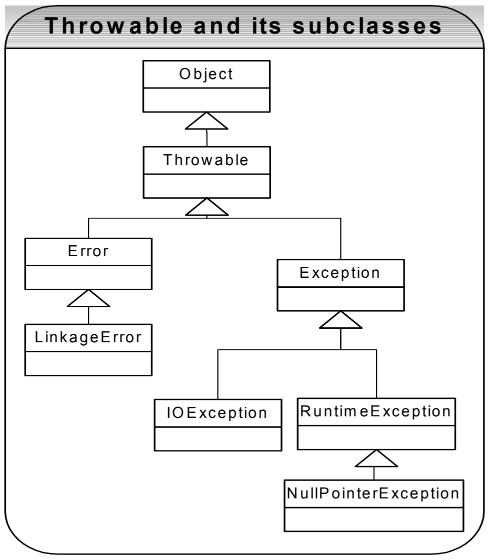
```
sample code

public void buildCar() {
    Car c = new Car();
    Engine e = new Engine();
    //lets create a circular reference
    c.engine = e;
    e.car = c;
}
```

Q 34: Discuss the Java error handling mechanism? What is the difference between Runtime (**unchecked**) exceptions and **checked** exceptions? What is the implication of catching all the exceptions with the type "*Exception*"? **EH** **BP**

A 34:

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

Exceptions: Most programs throw and catch objects that derive from the Exception class. Exceptions indicate that a problem occurred but that the problem is not a serious JVM problem. An Exception class has many subclasses. These descendants indicate various types of exceptions that can occur. For example, NegativeArraySizeException indicates that a program attempted to create an array with a negative size. One exception subclass has special meaning in the Java language: RuntimeException. All the exceptions except RuntimeException are compiler checked exceptions. If a method is capable of throwing a checked exception it must declare it in its method header or handle it in a try/catch block. Failure to do so raises a compiler error. So checked exceptions can, at compile time, greatly reduce the occurrence of unhandled exceptions surfacing at runtime in a given application at the expense of requiring large throws declarations and encouraging use of poorly-constructed try/catch blocks. Checked exceptions are present in other languages like C++, C#, and Python.



Runtime Exceptions (unchecked exception)

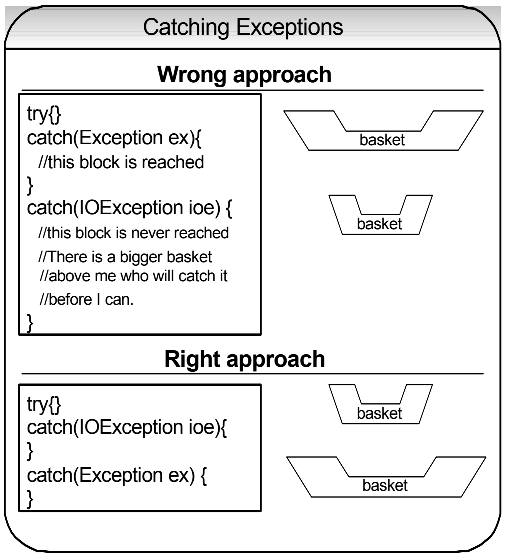
A RuntimeException class represents exceptions that occur within the Java virtual machine (during runtime). An example of a runtime exception is NullPointerException. The cost of checking for the runtime exception often outweighs the benefit of catching it. Attempting to catch or specify all of them 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.

Exception handling best practices: **BP**

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 below 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, ArrayIndexOut-OfBounds-Exception) as well.



```
try{}
catch(Exception ex){
  //this block is reached
}
catch(IOException ioe) {
  //this block is never reached
  //There is a bigger basket
  //above me who will catch it
  //before I can.
}
```

```
try{}
catch(IOException ioe){
}
catch(Exception ex) {
}
```

Hint: as shown in the figure, think of catching an exception in a basket. You should always have the smaller basket above the bigger one. Otherwise the bigger basket will catch all the exceptions and smaller baskets will not catch any.

Why should you throw an exception early? **CO**

The exception stack trace helps you pinpoint where an exception occurred by showing us 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:

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

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);
…
```

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. 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.

Note: 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 and convert your SQLException into an unchecked (i.e. RuntimeException) exception, if the client code cannot do anything about it.

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 35: What is a user defined exception? EH
A 35: User defined exceptions may be implemented by defining a new exception class by extending the *Exception* class.

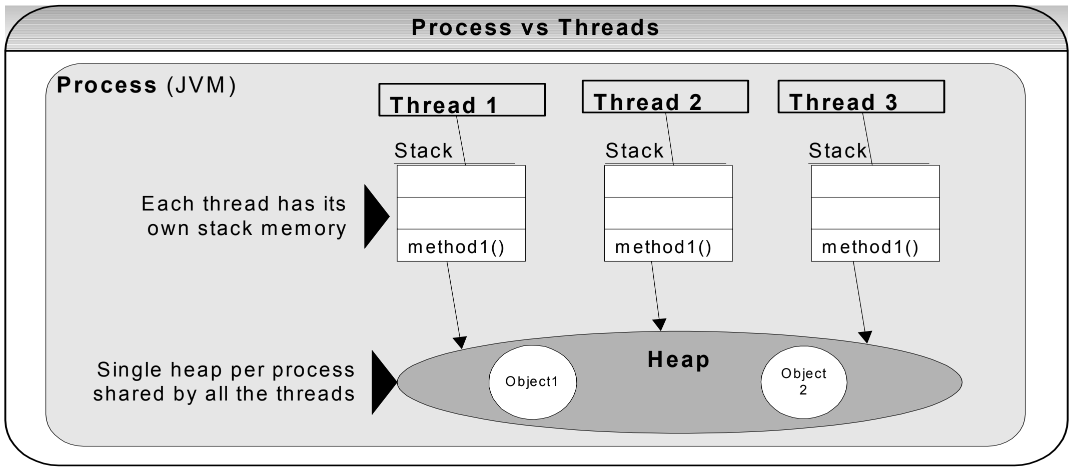
```
public class MyException extends Exception {

  /* 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. Throw an exception:

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

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

Q 36: What is the difference between processes and threads? LF MI CI
A 36: 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 37: Explain different ways of creating a thread? LF
A 37: Threads can be used by either :

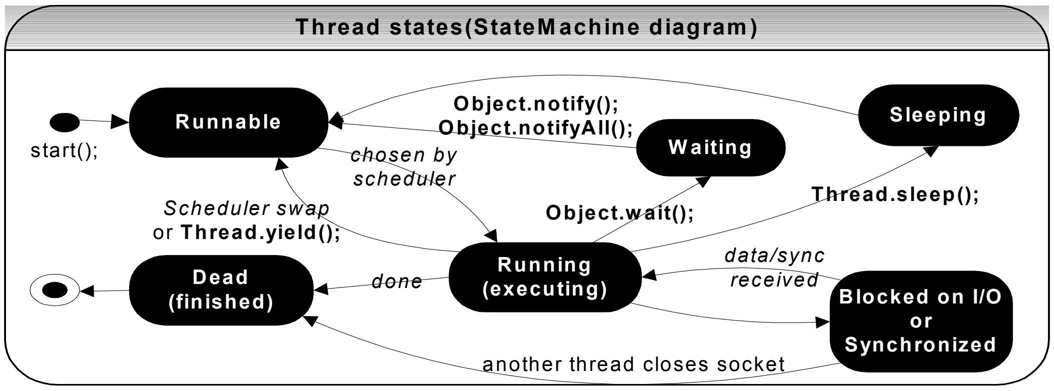
- Extending the ***Thread*** class
- Implementing the ***Runnable*** interface.

```
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
   }
}
```

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.

Q 38: Briefly explain high-level thread states? LF
A 38: 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 schedular 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 39: What is the difference between yield and sleeping? LF
A 39: 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.

Q 40: 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
A 40: 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 ;

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

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

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

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

    pubic void  method1() {
       synchronized(xLock){
          //access x here. thread safe
      }
       //do something here but don't  use
SharedResource  x, y ;

        synchronized(xLock) {
           synchronized(yLock) {
               //access x,y here. thread safe
           }
        }
        //do something here but don't use
SharedResource  x, y ;
    }
}
```

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.

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 the performance.

Q 41: What is a daemon thread? **LF**
A 41: Daemon threads are sometimes called "service" threads. These are threads that normally run at a low priority and provide a basic service to a program or programs when activity on a machine is reduced. An example of a daemon thread that is continuously running is the garbage collector thread. This thread is provided by the JVM.

Q 42: 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**
A 42: 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.

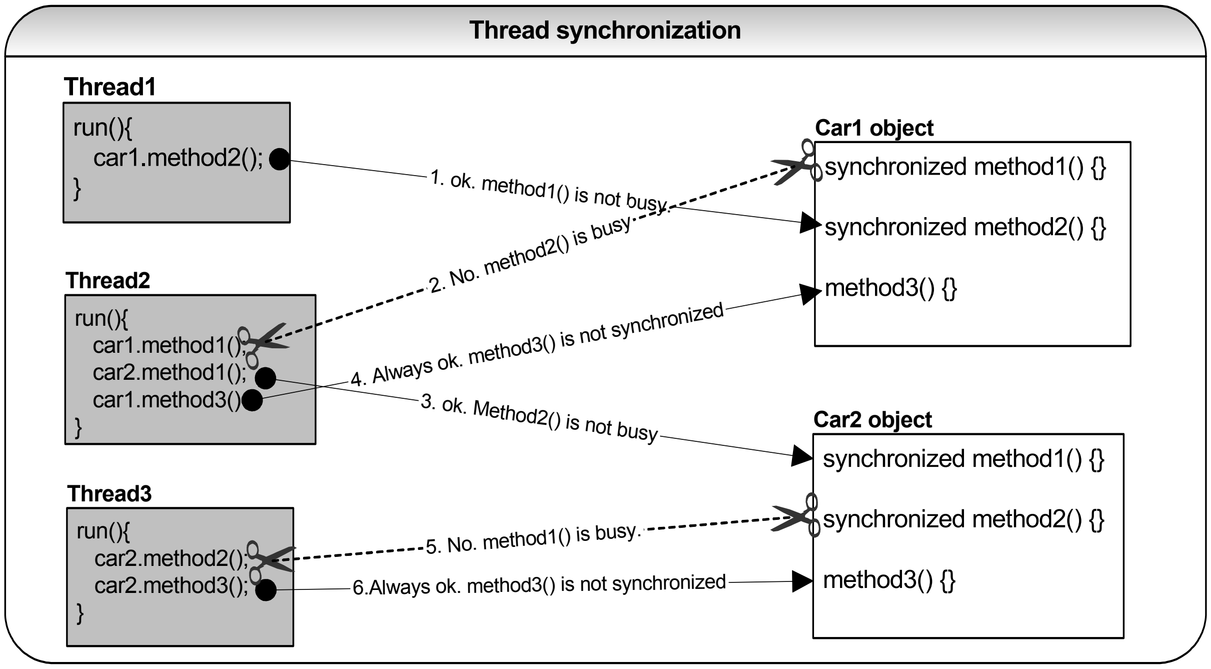
Example: If you imagine an application in which one thread (the producer) writes data to a file while a second thread (the consumer) reads data from the same file. In this example the concurrent threads share the same resource file. Because these threads share the common resource file they should be 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.
  }
}
```

Q 43: If 2 different threads hit 2 different synchronized methods in an object at the same time will they both continue? **LF**
A 43: No. Only one method can acquire the lock.


Thread synchronization
Thread1
run(){
car1.method2();
}
Thread2
run(){
car1.method1();
car2.method1();
car1.method3()
}
Thread3
run(){
car2.method2();
car2.method3();
}
1. ok. method1() is not busy.
2. No. method2() is busy.
3. ok. Method2() is not busy
4. Always ok. method3() is not synchronized
5. No. method1() is busy.
6.Always ok. method3() is not synchronized
Car1 object
synchronized method1() {}
synchronized method2() {}
method3() {}
Car2 object
synchronized method1() {}
synchronized method2() {}
method3() {}

Q 44: Explain threads blocking on I/O? LF

A 44: 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 nonblocking I/O (aka NIO that stands for New I/O). Refer **Q20** in Java section for NIO.

Note: Q45 & **Q46** are very popular questions on design patterns.

Q 45: What is a **singleton** pattern? How do you code it in Java? DP MI CO

A 45: 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.

```
public class OnlyOne {

   private static OnlyOne one = new OnlyOne();

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

   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 out side the class. The only way to instantiate an instance of class *OnlyOne* is through the **getInstance()** method with a public access modifier.

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).

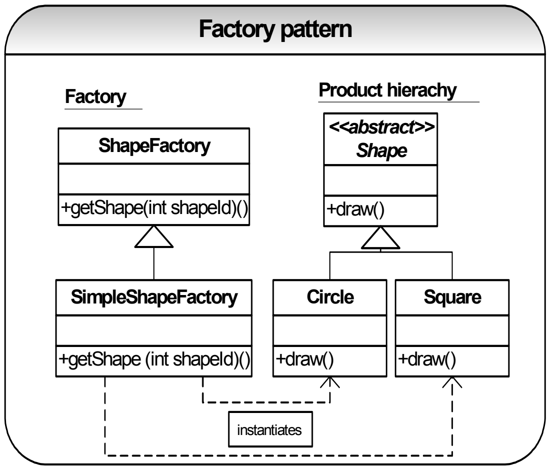
What is the difference between a singleton class and a static class? Static class is one approach to make a class singleton by declaring the class as "final" so that it cannot be extended and declaring all the methods as static so that you can't create any instance of the class and can call the static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

Q 46: What is a factory pattern? DP CO

A 46: 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)

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 sub-classes, and based on some data provided, you have to return the object of one of the subclasses. Let's look at a sample code:



```
public interface Const {
   public static final int SHAPE_CIRCLE =1;
   public static final int SHAPE_SQUARE =2;
   public static final int SHAPE_HEXAGON =3;
}

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

public class SimpleShapeFactory extends
         ShapeFactory  throws BadShapeException  {
```

Abstract factory pattern

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:

```
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;
   }
}
```

Now let's look at the abstract factory, which returns one of the types of ShapeFactory:

```
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) {
          sf = new SimpleShapeFactory();
       }
       else if (type == TYPE_COMPLEX) {
          sf = new ComplexShapeFactory();
       }
       else throw new BadShapeFactoryException("No factory!!");
    public Shape getShape(int shapeTypeId){
        Shape shape = null;
        if(shapeTypeId == Const.SHAPE_CIRCLE) {
            //in future can reuse or cache objects.
            shape = new Circle();
        }
        else if(shapeTypeId == Const.SHAPE_SQUARE) {
            //in future can reuse or cache objects
            shape = new Square();
        }
        else throw new BadShapeException
                ("ShapeTypeId="+ shapeTypeId);

        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
```

```
        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.
factory = abFac.getShapeFactory(2);
//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
```

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 **Q8** 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.

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. Ge/tFactoryInstance();
factory.getShape();
```

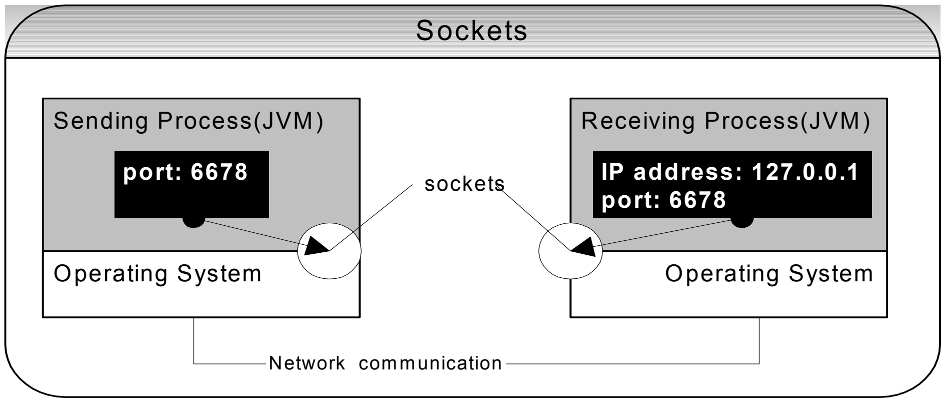
Note: Since questions on singleton pattern and factory pattern are commonly asked in the interviews, they are included as part of this section. To learn more about design patterns refer **Q11** in How would you go about section…?

Q 47: What is a socket? How do you facilitate inter process communication in Java? LF

A 47: 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 128-bit 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 48: How will you call a Web server from a stand alone Java application? LF
A 48: Using the **java.net.URLConnection** and its subclasses like HttpURLConnection and JarURLConnection.

URLConnection	**HttpClient (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.

Java – Swing

Q 49: What is the difference between AWT and Swing? LF DC
A 49: Swing provides a richer set of components than AWT. They are 100% Java-based. There are a few other advantages to Swing over AWT:

- Swing provides both additional components like JTable, JTree etc and added functionality to AWT-replacement components.
- Swing components can change their appearance based on the current "look and feel" library that's being used.
- Swing components follow the **Model-View-Controller** (MVC) paradigm, and thus can provide a much more flexible UI.
- Swing provides "extras" for components, such as: icons on many components, decorative borders for components, tool tips for components etc.
- Swing components are lightweight (less resource intensive than AWT).