**What are “static initializers” or “static blocks with no function names”?**

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

public class 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);

}

What is the difference between an instance variable and a static variable? Give an example where you might use

a static variable?

**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**.

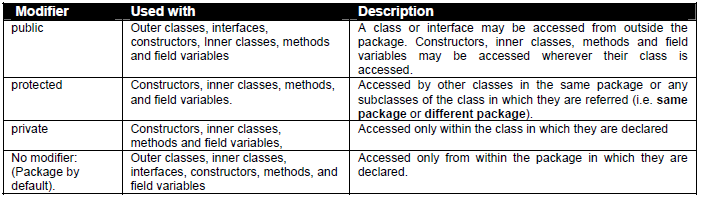
Give an example where you might use a static method?

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.

What are access modifiers?



Where and how can you use a private constructor?

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).

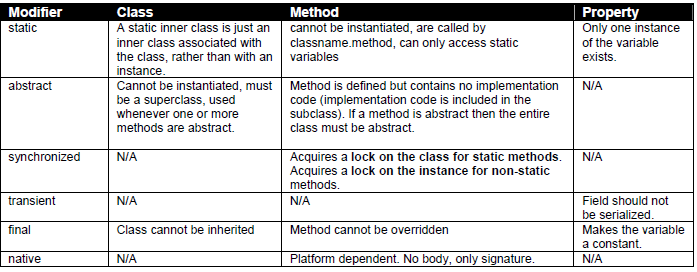
􀂃 Used in the factory method pattern (Refer **Q46** in Java section).

􀂃 Used in utility classes e.g. StringUtils etc.

What is a final modifier? Explain other Java modifiers?

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).



**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”.

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

􀂃 **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.

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?

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 ]**

**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 main difference between pass-by-reference and pass-by-value?

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

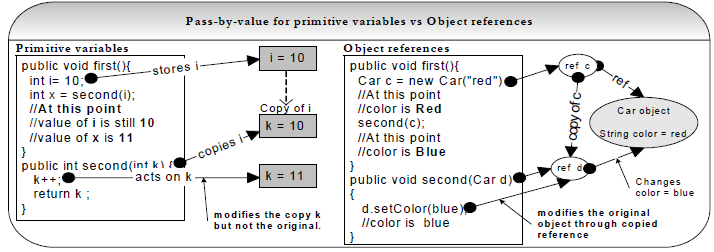
exactly how **pass-by-value** (i.e. copy-by-value) works.

In Java, if a calling method passes a reference of an object as an argument to the called method then the **passedin**

**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**.



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.

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

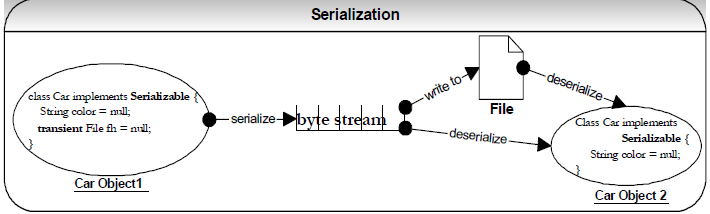
What is the common use? **LF SI PI**

**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).

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

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

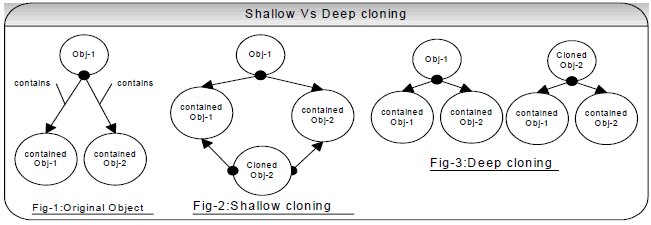
are not. The contained objects Obj-1 and Obj-2 are affected by changes to cloned Obj-2. Java supports shallow

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.

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

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

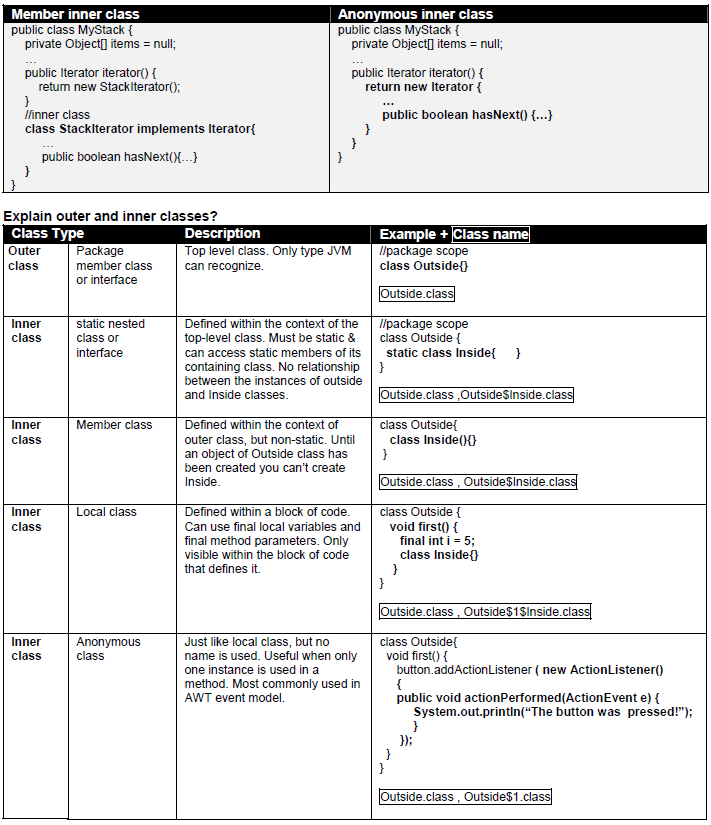
in the outer class for the inner class to access the private members. This leaves a **security hole**. In general **we**

**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.



**Java – Key Points**

Java is an object oriented (OO) language, which has built in support for multi-threading, socket communication,

automatic memory management (i.e. garbage collection) and also has better portability than other languages across

operating systems.

􀂃 Java class loaders are **hierarchical** and use a **delegation model**. The classes loaded by a child class loader have

**visibility** into classes loaded by its parents up the hierarchy but the reverse is not true.

􀂃 Java does not support **multiple implementation inheritance** but supports **multiple interface inheritance**.

􀂃 **Polymorphism**, **inheritance** and **encapsulation** are the 3 pillar of an object-oriented language.

􀂃 Code reuse can be achieved through either **inheritance** (“is a” relationship) or **object composition** (“has a”

relationship). Favour object composition over inheritance.

􀂃 When using **implementation inheritance**, make sure that the **subclasses depend only on the behaviour of the**

**superclass**, not the actual implementation. An **abstract** base class usually provides an implementation inheritance.

􀂃 Favour **interface inheritance** to **implementation inheritance** because it promotes the deign concept of **coding to**

**interface** and **reduces coupling**. The interface inheritance can achieve code reuse through **object composition**.

􀂃 Design by contract specifies the obligations of a calling-method and called-method to each other using **preconditions**,

**post-conditions** and **class invariants**.

􀂃 When using Java collection API, prefer using *ArrayList* or *HashMap* as opposed to *Vector* or *Hashtable* to **avoid any**

**synchronization overhead**. The *ArrayList* or *HashMap* can be externally synchronized for concurrent access by

multiple threads.

􀂃 Set the initial capacity of a collection appropriately and program in terms of interfaces as opposed to

implementations.

􀂃 When providing a user defined key class for storing objects in *HashMap*, you should override **equals()**, and

**hashCode()** methods from the *Object* class.

􀂃 *String* class is immutable and *StringBuffer* and *StringBuilder* classes are mutable. So it is more efficient to use a

*StringBuffer* or a *StringBuilder* as opposed to a *String* in a computation intensive situations (ie. in for, while loops).

􀂃 **Serialization** is a process of writing an object to a file or a stream. **Transient** variables cannot be serialized.

􀂃 Java I/O performance can be improved by using buffering, minimising access to the underlying hard disk and

operating systems. Use the NIO package for performance enhancing features like non-blocking I/O operation, buffers

to hold data, and memory mapping of files.

􀂃 Each time an object is created in Java it goes into the area of memory known as **heap**. The primitive variables are

allocated in the **stack** if they are local method variables and in the **heap** if they are class member variables.

􀂃 Threads **share the heap spaces** so it is **not thread-safe** and the threads have **their own stack space**, which is

**thread-safe**.

􀂃 The **garbage collection cannot be forced**, but you can nicely ask the garbage collector to collect garbage.

􀂃 There two types of exceptions **checked** (ie compiler checked) and **unchecked** (Runtime Exceptions). It is not

advisable to catch type *Exception*.

􀂃 A **process** is an execution of a program (e.g. JVM process) but a **thread** is a single execution sequence within the

process.

In Java each object has a lock and a thread can acquire a lock by using the **synchronized** key word. The

synchronization key word can be applied in **method level** (coarse-grained lock) or **block level** (fine-grained lock

which offers better performance) of code.

􀂃 Threads can communicate with each other using **wait(), notify(), and notifyAll()** methods. This communication

solves the **consumer-producer** problem.

􀂃 Sockets are communication channels, which facilitate inter-process communication.

􀂃 Swing uses the **MVC paradigm** to provide loose coupling and action **architecture** to implement a shared behaviour

between two or more user interface components.

􀂃 Swing components should be accessed through an **event-dispatching thread**. There is a way to access the Swing

event-dispatching thread from outside event-handling or drawing code, is using *SwingUtilities’* **invokeLater()** and

**invokeAndWait()** methods.

􀂃 A signed applet can become a trusted applet, which can work outside the sandbox.

􀂃 In Java typically memory leak occurs when an object of longer life cycle has a reference to objects of a short life

cycle.

􀂃 Threads can be created in Java by either extending the *Thread* class or implementing the *Runnable* interface.