Java concepts

# Core Java

* JDK, JRE, JVM
* JVM

JVM is an acronym for Java Virtual Machine, it is an abstract machine which provides the runtime environment in which java byte code can be executed.

JVMs are available for many hardware and software platforms (so JVM is platform dependent).

* JRE

Java Runtime Environment (JRE) is the implementation of JVM. JRE consists of JVM and java binaries and other classes to execute any program successfully. JRE doesn’t contain any development tools like java compiler, debugger etc. If you want to execute any java program, you should have JRE installed.. It is the implementation of JVM and physically exists.

* JDK

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools. JDK provides all the tools, executable and binaries required to compile, debug and execute a Java Program.

The execution part is handled by JVM to provide machine independence.

* Different ways to create object
* Using new keyword

This is the most common way to create an object in java. I read somewhere that almost 99% of objects are created in this way.

MyObject object = new MyObject ();

* Using Class.forName()

If we know the name of the class & if it has a public default constructor we can create an object in this way.

MyObject object = (MyObject) Class.forName ("subin.rnd.MyObject").newInstance ();

* Using clone()

The clone () can be used to create a copy of an existing object.but must implement Cloneable marker interface

MyObject anotherObject = new MyObject ();

MyObject object = anotherObject.clone();

* Using object deserialization

Object deserialization is nothing but creating an object from its serialized form.

ObjectInputStream inStream = new ObjectInputStream (anInputStream );

MyObject object = (MyObject) inStream.readObject ();

* Internal Architecture of JVM
* Classloader: Classloader is a subsystem of JVM that is used to load class files.
* Class (Method) Area: Class (Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.
* Heap: It is the runtime data area in which objects are allocated.
* Stack: Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as thread. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.
* Program Counter Register: PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.
* Native Method Stack: It contains all the native methods used in the application.
* Execution Engine: It contains:

1) A virtual processor

2) Interpreter: Read byte code stream then execute the instructions.

3) Just-In-Time (JIT) compiler: It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term? Compiler? Refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

* JIT compiler - Java programs consists of classes, which contain platform-neutral byte codes that can be interpreted by a JVM on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling byte codes into native machine code at run time.
* Access modifiers can be - Static, abstract, final, synchronized, transient, native, volatile
* Static Members: Static members are the members with Static keyword in their declarations. Class variables are called as Static variables. These members belong to the class not to the object i.e. they are not instantiated when the class instance is created. The values of these variables are not part of the object state. The static variables are initialized to their default values (if explicit initialization is not specified) at the time of class loading. The Static methods are called as class methods. A static method can directly access other static members in the class. It cannot access instance (non-static) members of the class. But it can always use a reference of the class type to access its members both static and non-static.
* Final Members: Final variable is a constant; its value cannot be changed after its initialization. This applies to instance, static and local variables including parameters that are declared as final. A final variable of primitive data type cannot change its value once it has been initialized. A final variable of a reference type cannot change its reference value once it has been initialized, but the state of the object it denotes can still be changed.Note: Variables defined in Interfaces are implicitly final. Final variables must be initialized before it is used. Final methods in a class are complete i.e. these methods has implementations and hence cannot be overridden in the subclasses.
* Abstract Methods: If method has a keyword abstract in its declaration, then such method/function is called Abstract method. Abstract methods does not have an implementation i.e. method body is not defined; only method prototype is specified in the class definition.

Note: Only instance methods can be declared as abstract. Since Static methods cannot be overridden declaring abstract static method would of no use. A Final method cannot be abstract and vice versa. Methods specified in an Interface are implicitly abstract.

* Synchronized Methods: Multiple threads can be executing in a program and at times they might try to execute several methods on the same object simultaneously. If there is a requirement that only one thread at a time should execute a method in the object, then such methods can be declared as Synchronized. Their execution will be mutually exclusive among all threads. At any given time, at the most one thread will be executing a Synchronized method on an object.

Note: Synchronized methods are also applicable to Static methods of a class.

* Native Methods: Native Methods are also called as foreign methods. Such methods implementation is not defined in Java but in another programming language. These methods are specified in the class as method prototypes with prefix with keyword native, no method body is defined in the Java class.
* Transient Fields: Objects can be stored using serialization. Serialization transforms objects into an output format which is helpful for storing objects. Objects can later be retrieved in the same state as when they were serialized, meaning that fields included in the serialization will have the same values at the time of serialization. Such objects are said to be Persistent. The fields are declared with keyword Transient in their class declaration if its value should not be saved when objects of the class are written to persistent storage.
* Volatile Fields: During execution, complied code might cache the values of fields for efficiency reasons. And as multiple threads will access the same field, caching is not allowed to cause inconsistencies when reading and writing the value in the field. The Volatile modifier can be used to inform the compiler that it should not attempt to perform optimizations on the field which could cause unpredicted results when the field is accessed by multiple threads. It is always read from main memory.
* Static member classes: Classes which are defined with Static modifier inside the top-level class or other Static member class are called Static member classes. It can be instantiated like a normal top-level class; no enclosing instance is required for this class instantiation. All the 4 accessibility modifiers i.e. Public, Protected, Package & Private are applicable to Static member classes’ declaration.
* Non-Static member classes: Classes which are defined without Static modifier inside another class are called non-static member classes. An instance of a non-static member class always has an enclosing instance associated with it.

The accessibility modifiers i.e. Public, Protected, Package & Private, abstract, final are applicable to Non-Static member classes’ declaration.

* Local classes: These classes are defined in the context of a block as in a body of the method or normal local block, just as local variables defined in a method body or local block. An instance of a local class has an enclosing instance associated with it, if it is declared in non-static context.

No accessibility modifiers are applicable for Local classes.

* Anonymous classes: These are defined as expressions and instantiated on the fly. An instance of anonymous class has an enclosing instance associated with it, if it is declared in non-static context. No accessibility modifiers are applicable for Anonymous classes.

Note: A Nested Class or Interface cannot have the same name as any of the enclosing classes or interfaces.

* An upper level class can have only public, default, abstract and final modifiers.
* An inner class can have public, default, protected, private, abstract, static and final as modifiers.
* Inside a class only public, default, protected, private, static, final, transient and volatile variables are permitted.
* Level of priority of access modifiers are public (anywhere accessible) -> protected (within package and to it subclasses) -> default (only within package) -> private (only within class)
* Overriding techniques –

Class Employee extends Person then –

* If Person has a method named ***show*** then child class Employee having a same named method will override it. It should also have the same return type.
* A class cannot have two methods of same name although having different return types.
* A static method in parent class cannot be overridden in the child class.
* If Person p = new Employee () and p.***show*** () (non-static method), p.***print*** () (static method) then in first case it would call Employee’s ***show*** and second case Person’s ***print***.
* It is because static method cannot be overridden.
* If Person.print () and Employee.print () is done then in first case Person’s ***print*** will get called and in second case Employee’s ***print.***
* Final method is inherited but cannot be overridden. If there is method named **finalShow** () in Person then we cannot write a **finalShow** () method in Employee, a compilation error will occur. But we can call the Person’s **finalShow** () method by creating an Employee object (Employee emp = new Employee; emp.**finalShow** ()). In this case Person’s **finalShow** () will get called.
* Constructors cannot be final since a constructor cannot be inherited.
* A final blank variable i.e. **final int a** should always be initialized and it cannot be initialized/modified inside any method. It can be only initialized inside a ***constructor***.
* A static final blank variable i.e. **static final int a** should always be initialized and it cannot be initialized/modified inside any method (not even a static method also). It can be only initialized inside a ***static block***.
* A List<Person> emplCol = new ArrayList<Employee> () is not possible even if Employee extends Person. Always the same type should be maintained while declaring collection objects.
* Suppose if a method as an argument takes ***List<Person>***i.e. ***show (List<Person> emplCol)*** and
* While the invocation of the method we pass a child class list i.e. ***List< Employee > emplCol***, then it will throw a compilation error. It is for the same reason because same type should be maintained as described in the above point.
* However if the argument type can be made generic then we can pass the child class type list also i.e. ***show (List emplCol)***
* Marker interface in Java is interfaces with no field or methods or in simple word empty interface in java is called marker interface. Example of market interface is ***Serializable***, ***Cloneable*** and Remote interface. It looks they are used to indicate something to compiler or JVM. So if JVM sees a Class is Serializable it done some special operation on it, similar way if JVM sees one Class is implement Clonnable it performs some operation to support cloning. Same is true for RMI and Remote interface. So in short Marker interface indicate, signal or a command to Compiler or JVM.
* Singleton vs. Static
* Static class will have its entire member as static only unlike Singleton.
* It can be lazily loaded whereas static will be initialized whenever it is first loaded.
* Singleton object stores in Heap but, static object stores in stack.
* We can clone the object of Singleton but, we cannot clone the static class object.
* Singleton can use the Object Oriented feature of polymorphism but static class cannot.
* ConcurrentHashMap - ConcurrentHashMap is thread safe but does not use locking on complete map. It is fast and has better performance in comparison to Hashtable in concurrent environment.  Retrieval of elements from ConcurrentHashMap does not use locking. It may overlap with update operation. We get the elements of last successfully completed update operation. ConcurrentHashMap does not throw ConcurrentModificationException. Concurrent updates are thread safe. ConcurrentHashMap constructor has an optional concurrency level argument. The default value is 16. This is the estimated number of concurrently updating threads. It is used in internal sizing to accommodate concurrently updating threads.
* Comparator vs Comparable
* Comparator in Java is defined in java.util package while Comparable interface in Java is defined in java.lang package, which very much says that Comparator should be used as an utility to sort objects which Comparable should be provided by default.
* Comparator interface in Java has method public int compare (Object o1, Object o2) which returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. While Comparable interface has method public int compareTo (Object o) which returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
* If you see then logical difference between these two is Comparator in Java compare two objects provided to him, while Comparable interface compares "this" reference with the object specified.
* If any class implement Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and object will be sorted based on their natural order defined by CompareTo method.

Java questions

* Why is a character array preferred over a string for storing passwords?

Since Strings are immutable in Java if you store password as plain text it will be available in memory until Garbage collector clears it and since String are used in String pool for reusability there is pretty high chance that it will  remain in memory for long duration, which pose a security threat. Since anyone who has access to memory dump can find the password in clear text and that's another reason you should always use an encrypted password than plain text. Since Strings are immutable there is no way contents of Strings can be changed because [any change will produce new String](http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html), while if you char [] you can still set all his element as blank or zero. So storing password in character array clearly mitigates security risk of stealing password.

With plain String, you have much higher chances of accidentally printing the password to logs, monitors or some other insecure place. Char [] is less vulnerable.

* How to check java version via a java program?

System.getProperties (“java.version”) or System.getProperties (“java.specification.version”)

* Hash Map collision nicely explained

<http://ydtech.blogspot.in/2010/06/hashmap-hashcode-collision-by-example.html>