JAVA TUTORIAL

SCJP 1.4 EXAM OBJECTIVES (310-035)

Version 1.0
Oct. 30th, 2005

email: deepakgpatel1@yahoo.com

Purpose of this tutorial:

The purpose of this document is to provide you with the important concepts that are ***NOT*** covered in many books. I have tried my best to cover as many points I could in this tutorial.

So I recommend you to go through the entire tutorial at least once, so that you make sure that you are not missing anything.

I will be constantly adding and updating this tutorial with new points and making the previous versions better, so you can email me for your suggestions/feedback on deepakgpatel1@yahoo.com . Latest version of the document can be downloaded from www.geocities.com/deepakgpatel1

Please let me know what you think of this document. Your feedback will help me to improve the future versions. You can also check out the latest versions at www.rajeshpatkar.com

All The Best!

Deepak Patel.

Chapter 1. Language Fundamentals.

- All Java source files end with .java extension.
- A source file is valid if there are no package or *import* statements or *class* or *interface* defined.
- When *class* files are created, they must be placed in directory hierarchy that reflects their *package* names. You must be careful that each component of your *package* name hierarchy is a legitimate directory name on all platforms.
- You must use only alphanumeric characters in *package* names.

Keywords and Identifiers

- A keyword is a word whose meaning is defined by the programming language.
- 'main' is **not** a keyword.
- Keyword *strictfp* is used only for methods and classes. It forces floating points to adhere to IEE754 standards.
- An identifier is a word used by the programmer to name a variable, method, *class*, or a label.
- Keywords may not be used as identifiers, but embedded keywords are allowed – Foex: $int is a valid identifier.
- Identifiers are case sensitive – Foex: value and VALUE are distinct identifiers.
- An identifier **cannot** have a dash (-) in its body.

Primitive Data Types

- Size of primitives should be consistent on all Java implementations.
- Variables of type *boolean* may take only the values *true* or *false*. Their representation size may vary.
- *char* is the only unsigned integer primitive.
- Float.NaN == Float.Nan always results in *false*.
- Float.NaN != Float.NaN always results in *true*.

Literals

- A character literal represents a single Unicode character.
- Default type of any floating point number is *double*
  Foex: *float* f = 1.5; gives Compiler error.
  Hence a floating point literal with no F/f or D/d as suffix defaults to *double.*
- String literal is a sequence of characters enclosed in double quotes.

Arrays

- Arrays are always initialized to default values when created. Even if it is created locally.
- Size of the array is specified at runtime when the array is allocated via *new* keyword.
- Size of the arrays are fixed at the creation they **cannot** be resized. If you want to change the size you can create a new array and assign the old one to it.
- Arrays are objects, you can even execute methods on them.
- Arrays can be used as arguments of a user defined method exactly as any other type.
- Arrayname.length is a property for an array specifying the size of the array.
- Test[][] t = *new* Test [10][];
  While initializing a multidimensional array it is necessary and sufficient to initialize the leftmost dimension for the code to compile.

Importing

- Importing does not mean class loading. Just the name of the class is brought into the source file's namespace.
- Adding more classes with *import* statement does not cause a runtime performance overhead.
- Importing any package doesn't *import* its subpackages.
- When the *import* keyword is followed by *static*, the compiler imports the name of a *static* element of the class.
- Only *public* data may be imported from the classes in external packages.
- As with ordinary imports, *static* imports have only a slight compile-time cost and zero runtime cost.

Class Fundamentals.

- Class level variables are always initialized to default values.
- The main() method must be *public* so that JVM can call it.
- The main() method is *static* so that it can be executed without the necessity of constructing an instance of the application class.
- The argument to main() is a single dimension array of Strings, containing any arguments that the user might have entered on the command line.
- args []/argv [] is local for main method, so not visible outside main.

Variables and Initialization.

- Member/Instance variable of a *class* is created when an instance is created and destroyed when object is destroyed.
- Automatic/Local variable of a method is created on entry of method and destroyed after exiting the method (with an exception for the inner classes).
- Class/Static variable is created when the *class* is loaded and is destroyed when the *class* is unloaded.
- If local variable is not initialized, the compiler gives error and not warning.
- At *class* level --- *int* x, y = 10; ➔ x initialized to 0 and y initialized to 10.
- At method level --- *int* x, y = 10; ➔ x is uninitialized (error) and y initialized to 10.

Garbage Collection.

- Garbage Collector runs in the low priority thread and it **cannot** be forced.
- JVM tries its best to run Garbage Collection under low memory situations.
- The garbage collection algorithm is vendor implemented.
- As soon as object reference is made *null* it is eligible for garbage collection at the very next moment. It doesn't wait for execution thread to end or anything else. Garbage collection is **not** related to execution of thread.

Chapter 2. Operators and Assignments.

- *char* c = '1';
  c >> 1; ➔ **valid**.
- The operator == checks for memory address of 2 object references being compared and **not** their values.
- >>> can only operate on an *int*. For other primitives casting is needed.
- System.out.println(1+1); ➔ **valid**.
- int i = 2 + '2'; ➔ **valid**.
- int i = 0;
  System.out.println(i + '0'); ➔ **invalid**.
- The left argument for an *instanceof* operator can be any object reference expression, usually a variable or an array element, whereas the right operand must be a *class*, *interface* or an array type.
- You cannot use a java.lang.Class object reference or a String representing the name of the class as the right operand for *instanceof* operator.
- The *instanceof* operator can be used to determine if a reference is an instance of a particular primitive wrapper class.
- a = x ? b : c ➔ if (a = x) results in *true*, then a is assigned b, or else a is assigned c.
- Compound assignment operators exist for all binary, non-*boolean* operators: *=, /=, %=, +=, -=, &=, ^=, |=.

Chapter 3. Modifiers.

The Access Modifiers

public

- A *public class*, variable or method may be used in any Java program without restriction.
- In a particular Java file if more then 1 *public class/interface* exists then compiler error results.

private

- *private* keyword **cannot** be applied to top level classes.
- *private* data **cannot** be accessed even in the subclasses of the *class*.

protected

- An instance may read and write *protected* fields that it inherits from its superclass. However, the instance may not read or write *protected* inherited fields from its superclass.
- An instance may call *protected* methods that it inherits from its superclass. However, the instance may not call *protected* methods of other instances.

Overridden methods

- Methods may **not** be overridden to be more *private*.
- A *private* method may be overridden by *private*, default, *protected* or *public* methods.
- A default method may be overridden by default, *protected* or *public* methods.
- A *protected* method may be overridden by *protected* or *public* methods.
- A *public* method may be overridden by only *public* methods.

Other Modifiers

final

- *final class* **cannot** have *abstract* methods.
- *final* features cannot be changed.
- *final class* **cannot** be subclassed.

abstract

- *abstract* method does **not** have an implementation, it has to be implemented by the first concrete subclass of the containing *class*.
- Class with atleast 1 *abstract* method has to be *abstract*. However *abstract class* need **not** have any *abstract* methods
- A *class* that is *abstract* may **not** be instantiated.
- If a parent *class* is declared *abstract*, child *class* has to be declared *abstract* or must define all the *abstract* methods of the parent *class*.
- Interface may or may not have *abstract* keyword.
  Methods of *interface* may or may not have *abstract* keyword.
  The *class* implementing the *interface* should be *abstract* if it doesn't define all the methods.
- *abstract* method **cannot** be *final*, *static* or *private*.
- *abstract* methods do **not** have body.
- *abstract* methods can throw Exceptions.
- It is valid to declare an inherited method as *abstract*.

static

- A *static* method has no '*this'*.
- You need to create an instance of the *class* in order to all any non-*static* methods.
- Static methods **cannot** be overridden to be non-*static* and vice-versa.
- Static methods **cannot** be overridden but they can be overloaded.
- Though a superclass and a subclass can have *static* methods with identical names, arguments and *return* types but it is not considered overriding because the methods are *static*.
- A variable declared *static* inside a method causes a Compiler error. The lifetime of a field within a method is duration of running of method. A *static* field exists only for the *class*. Hence local variables cannot be declared *static*.
- Static method accessing non-*static* variable gives Compiler error.
- Static methods can be *synchronized*, in this case the lock is obtained in the Class object of the *class*.

native

- *native* methods do **not** have body. The body lies outside the JVM.
- Calling a method declared *native* causes runtime error.
- *public static native void* amethod() { ….. } ➔ **invalid** because of { … } . *native* means method has no body.
- *native* is used to get access hardware that Java does not know about.

- It is used to write optimized code for performance in language such as C/C++.
- It violates Java's platform independence.

transient

- *transient* is a keyword used for preventing variables in an object from being serialized.
- *transient* variables **cannot** be declared *final* or *static*.

volatile

- *volatile* is used to inform the compiler that a variable may be changed asynchronously with regards to its current definition(e.g. from a *native* method outside the JVM).
- A variable may be declared *volatile*, in which case a thread must reconcile its working copy of the field with master copy every time it accesses the variable.

synchronized

- To synchronize an entire method, put the *synchronized* keyword in the method's declaration.
- To synchronize a part of a method, using the lock of an arbitrary object, put curly brackets around the code to be *synchronized*, preceded by *synchronized* (thearbitraryobject).
- To synchronize a part of a method, using the lock of the object that owns the method, put curly brackets around the code to be synchronized, preceded by *synchronized* (*this*).
- Synchronized modifier cannot be applied to constructors. However block synchronization can be used within constructors.
- Synchronized method can be inherited to be non-*synchronized* and vice-versa.

Chapter 4. Casting and Conversion.

- Unlike the casting rules, all conversion rules are enforced at compile time.
- Casting *char* to *double* is **valid**.
- Casting an *interface* to a *final class* is **invalid**.
- References are stored in variables, and variables have types that are specified by the programmer at compile time.
- Object reference variables types can be classes, interfaces or arrays.

Primitive Conversion

- A *boolean* **cannot** be converted to any other type.
- *static int* i = -1;
  static double d = 10.1;
  d = i; ➔ valid conversion.
- Unary operators operate on a single value, whereas Binary operators operate on two values.
- Rules for unary operators, depending on the type of the single operand.
  - ✓ If the operand is a *byte*, a *short*, or a *char*, it is converted to an *int* (unless the operator is ++ or --, in which case no conversion happens).
  - ✓ Else there is no conversion.
- Rules for binary operators, depending on the type of the two operands.
  - ✓ If one of the operand is a *double*, the other operand is converted to *double*.
  - ✓ Else if one of the operand is a *float*, the other operand is converted to a *float*.
  - ✓ Else if one of the operand is a *long*, the other operand is converted to a *long*.
  - ✓ Else both the operands are converted to *int*.

Primitives and Casting

- Narrowing conversion runs a risk of losing information, the cast tells the compiler that you accept the risk.
- You cannot cast a *boolean* to any other type, you cannot cast any other type to *boolean*.

Object Reference Conversion

- Reference conversion, like primitive conversion, takes place at compile time, because the compiler has the information it needs to determine whether the conversion is legal.
- Rules for Object Reference Conversion:
  - ✓ An *interface* type can be converted only to an *interface* type or to Object. If the new type is an *interface*, it must be a superinterface of the old type.
  - ✓ A *class* type can be converted to a *class* type or to an *interface* type. If converting to a *class* type, the new type must be a superclass of the old type. If converting to an *interface* type, the old class must implement the *interface*.
  - ✓ An array may be converted to the *class* Object, to the *interface* Cloneable or Serializable, or to an array. Only an array of object reference types can be converted to an array, and the old element type must be convertible to the new element type.
- In general, object reference conversion is permitted when the direction of the conversion is "up" the inheritance hierarchy.

Object Reference Casting

- Only casting of object reference requires run time check, casting of numeric types doesn't.
- Rules for Object Reference Casting:
  - ✓ When both the Oldtype and Newtype are classes, one must be a subclass of the other.
  - ✓ When both Oldtype and Newtype are arrays, both arrays must contain reference types (not primitives), and it must be legal to cast an element of Oldtype to an element of Newtype.
  - ✓ You can always cast between an *interface* and a non-*final* object.
- In general, going "down" the inheritance tree requires an explicit casting.

Chapter 5. Flow Control, Assertions and Exception Handling.

The Loop Constructs

while() loop

- *while*(*int* i < 5) ➔ **invalid**
  Variable declaration inside the loop constructors is valid only for the *for*() loop.

The do loop

- The difference between the *while*() loop and the *do* loop is that this loop executes the body of the loop at least once, because the test is performed at the end of the body.

The for() loop

- The empty *for* loop (*for*(; ;) { }) repeats forever.
- *for*(*int* x = 0 , y=0 ; x + y = 10 ; x++) { … } ➔ **valid**.
- *for* (*int* x = 0 ,*long* y = 0 ; x + y = 10 ; x++) { … } ➔ **invalid**.

The break and continue statements in loops

- The *continue* statement prematurely completes the current iteration of the loop.
- The *break* statement causes the entire loop to be abandoned.
- A label should always be associated with a loop. It cannot appear anywhere in the code.

The Selection Statements

The if()/else Construct

- *if* takes **only** *boolean* arguments.
- The *else* part in the *if*/*else* statement is optional.
- The curly braces are optional if the body is limited to single statement.
- In case of nested *if*/*else* statements, each *else* clause belongs to the closest preceding *if* statement that does not have an *else*.

The switch() Construct

- The variable in the *switch* statement must be either *byte*, *short*, *char*, or an *int*.
- The argument for a *case* statement must be a constant or a constant expression which can be evaluated at compile time and not run time.

Exception Handling

- Error : Incorrect Condition
- Exception: Unusual condition.
- There are two kinds of exceptions: Checked and Runtime exceptions.
- Checked exceptions thrown by method **must** be caught or declared in the *throws* clause of the method or handled by the *try-catch* mechanism.
- You are free to call methods that throw runtime exceptions, without enclosing the calls in *try* blocks or adding *throws* to the method declaration.
- Methods that *throw* exceptions not declared in base *class* cannot be overridden, but can be overloaded.
- The *finally* block's code is guaranteed to execute in nearly all circumstances except the following:
  - ✓ The death of the current thread.
  - ✓ Execution of System.exit().
  - ✓ Turning off the computer.
- If both *catch* and *finally* blocks are defined, *catch* **must** precede *finally* block.
- A *try* block does not need to be followed by a *catch* block, but a *catch* block must always be associated with a *try* block.
- A *finally* block can never stand on its own i.e. without being associated with *try* block.
- It is legal to issue a *return* statement inside the *try* block.
- Examples of runtime exceptions are: ArrayIndexOutOfBoundsException,NullPointerException, IllegalArgumentException, ArithmeticException,NumberFormatException and ClassCastException.

Assertions

- In assertions, Expression 1 should always be *boolean* (not Boolean), else compiler error results.
- In assertions Expression 2 should be an expression. If it is a method call, then the method should *return* a value such as *int* or String. It should **not** *return void* or any wrapper *class*, else compiler error results.

- By compiling the code with javac –source 1.3 Application.java we can use the keyword *assert* as an identifier.
- Assertions can be enabled/disabled programmatically using ClassLoader, besides the usual command line switches.
- Even though assertions are disabled, they still increase the size of *byte* code of the *class*.
- The command line switches –esa or –enablesystemassertions can be used for enabling assertions in all system classes.
  -dsa or –disablesystemassertions disables the assertions in all system classes.
  -ea or –enableassertions switches enables assertions for all classes except for the system classes.
- If *assert* condition fails it *throws* Assertion error, so it may not be enclosed in *try-catch* blocks, though it is valid.
- Once *assert* condition fails rest of the statements are **not** executed.
- It is not mandatory to declare Error or its subclass in the *throws* clause of the method can *throw* it irrespective of what its superclass implementation throws.
- Java insists that assertion mechanism should not be used for checking parameters of a *public* method except for postconditions.
- Class AssertionError does not define any methods of its own, all of its methods are inherited from its parents.
- AssertionError is present in *package* java.lang.
- Assertions should not be used to control flow of execution.

Chapter 6. Objects and Classes.

Coupling and Cohesion

- Coupling: When object A is tightly coupled to object B, a programmer who wants to use or modify A is required to have an inappropriately extensive expertise in how to use B.
- Cohesion: It is the degree to which a *class* or method resists being broken down into smaller pieces. Cohesion is desirable.

Methods, Overloading and Overriding

- Having same name and different arguments is the necessary and sufficient condition for methods to be called Overloaded.
- There are no constraints on method *return* type, accessibility and exception lists for Overloading.
- Overloaded methods exist in any number in the same *class*.
- Overriding methods **must** have argument list of identical type and order.
- Methods may not be overridden to be more *private*.
- The *return* type must be the same as, or a subclass of, the superclass version's *return* type.
- The method may *throw* only checked exception types that are same as, or subclass of, exception types thrown by the original method.
- Each parent *class* method may be overridden once at most in any one subclass.
- Same name ,same arguments and different *return* type ➔ Illegal
  It's neither overriding nor overloading.

Constructors and Subclassing

- The superclass constructor must be called before any reference is made to any part of the subclass object.
- Constructors can only be invoked from within constructors.
- A Constructor is a method with no *return* type (not even *void*) and having same name as *class*.
- Constructors can be marked with any visibility modifiers.
- A class with all its constructors as private can't be extended and instantiated by any other class.
- The access modifier for the default constructor provided by the compiler is the same as that of the *class* to which the constructor belongs.
- Constructor code executes starting from oldest ancestor *class* downwards.

- Constructors cannot be overridden but can be overloaded. Overloading a constructor is a key technique to allow multiple ways of initializing classes.
- Java does not provide zero parameter constructor (i.e. having an uninitialized variable in the argument list of the constructor).
- Child *class* **cannot** access methods of the *class* Parent to its Parent using "*super*". You may access direct parent *class*, but classes further up the hierarchy are not visible.
- *this*() and *super*() can never be called in the same constructor.
- When an object is constructed the variables are initialized first and then the constructor is executed.
- Class *class* has no *public* constructor
- Constructors can *throw* Exceptions.
- Constructors **cannot** be *abstract*, *final*, *native*, *static*, *synchronized*.
- class A{}
  class B *extends* A{}
  class C *extends* B{}
  When C is invoked, A is executed first.
- class A{}
  class B *extends* A{
  A a = *new* B();
  }
  When 'a 'is used for accessing variables, variables of A are accessed.
  When 'a' is used for accessing methods, methods of B are accessed.

Inner classes

- Inner classes defined inside a *class* can have any access modifiers and can be declared *static*.
- Inner classes defined inside a method cannot be declared with any access modifier and cannot be *static*.
- Classes defined inside a method can access any variables of the enclosing *class*, but only *final* variables of the enclosing method.
- An Inner *class* may extend any other *class*.
- *static* Inner Class is also called "top level nested *class*".
- Variables inside the inner classes cannot be *static* unless the inner *class* itself is *static*.
- Inner *class* gets put into its own .class output file, using the format Outer$Inner.class.
- Anonymous *class* has no name, so no constructor.

Chapter 7.Threads.

Thread Fundamentals

- Thread is a subset of process. A multithreaded program is a process which has more than 1 threads.
- Threading and garbage collection are platform dependent and hence not suitable for Realtime Programming.
- 'main' is a thread with default priority 5.
- The Runnable Interface describes only one method.
- In the class which extends Thread if there is no run() method defined, then on calling the start() method, run() method of the Thread class which does nothing is called and there will be no output.
- Once a thread is dead, it cannot be started again, if you want the thread's task to be performed again, you have to construct and start a new thread instance.
- The dead thread continues to exist, you can access its data and call its methods.
- Thread scheduler chooses the highest-priority waiting thread to execute. If the threads in the waiting pool are having same priorities then any arbitrary thread is made to run. There is no guarantee that the thread chosen will be the one that has been waiting the longest.
- The specifics of how thread priorities affect scheduling are platform dependent.
- Threads inherit their priority from their parent thread.
- ThreadGroup class instance allows threads to be manipulated as groups.
- It is not necessary that all the Threads are of same type in a ThreadGroup.
- start(),run() and toString() methods are the instance methods of thread class.
- sleep(),join() and wait() methods all *throw* InterruptedException.
- You can invoke start() method only once for each thread, invoking it again results in an IllegalThreadStateException at runtime.

Daemon Threads

- The garbage collector is a daemon thread.
- The main is a non-daemon thread.
- Threads created by daemon threads are daemon threads.
- Threads created by non-daemon threads are non-daemon threads.
- The JVM runs until the only live threads are daemons.

Controlling Threads

yield()

- When the yield() method is called on a thread, the scheduler runs a thread with same or higher priority, if it is in the Ready state.
- If no other threads are waiting, then the thread that just yielded will get to continue execution **immediately**.
- The yield() method is a *static* method of the thread class.

sleep()

- There are 2 overloaded sleep methods in Thread class.
  - ✓ sleep(*int* milliseconds)
  - ✓ sleep(*int* milliseconds, *int* nanoseconds).
- sleep() and yield are *static* methods of Thread class. wait() and notify() are the non-*static* methods belonging to Object *class* which is inherited by Thread class.
- When a sleep() method is defined for a thread then InterruptedException has to be caught. If no exception is caught or if any other exception is caught then Compiler error will be thrown.
- When a thread has finished sleeping, it does not continue execution immediately. It enters the Ready state and will start execution only when thread scheduler tells it to do so.

Monitors, Waiting and Notifying

- A monitor is an object that can block and revive threads.
- An Object has only 1 lock which controls the access to all *synchronized* methods and blocks.
- The thread must have lock on the object on which wait() is to be invoked otherwise IllegalMonitorStateException results.
- If wait() is not called in *synchronized* code then IllegalMonitorStateException occurs.
- wait() method *throws* an InterruptedException.
- The thread that calls wait() method gives up the CPU, gives up the lock and goes into the monitors waiting pool.
- On execution of notify() method, one arbitrarily chosen thread gets moved out of the monitors waiting pool and into the Seeking Lock State.
- The thread that was notified must reacquire the monitors lock before it can proceed.

Synchronizing a Part of a method.

- To synchronize an entire method, put the *synchronized* keyword in the methods declaration.
- To synchronize a part of a method, using the lock of an arbitrary object, put curly brackets around the code to be *synchronized*, preceded by *synchronized* (thearbitraryobject).
- To synchronize a part of a method, using the lock of the object the owns the method, put curly brackets around the code to be *synchronized*, preceded by *synchronized* (*this*).

Chapter 8. The java.lang and java.util packages.

- The Java compiler automatically imports all the classes in the java.lang *package* into every source file.
- java.lang.Void class is *final* and has a *private* constructor, hence it cannot be extended or instantiated.

The Object Class

- The Object *class* is ancestor of all Java classes.
- If a *class* does not contain the *extends* keyword in its declaration, the compiler builds a *class* that *extends* directly from Object.
- All java objects have toString() method.
- The toString() method will create a new object every time it is called.
- We can use equals() method for objects of different classes.
- (obj1.equals(obj2)) returns *true* if and only if (obj1 == obj2).
- If 2 objects are equal then hashcodes returned by them are same, but converse is **NOT** true. If 2 objects are unequal, their hashcodes could possibly be same.
- All variables in hashCode() method should be present in the equals() method.
- All variables in equals() method need not be present in hashCode() method.
- *public int* hashCode() and *public boolean* equals(Object obj),these methods must be implemented correctly by a *class* if its objects have to behave consistently and properly with all Java collection classes and interfaces, or for that matter with any other classes.
- equals() method overridden in class Boolean returns true if and only if the object passed to it is non-null Boolean and represents the same boolean value as this object.
- equals() method overridden in class String returns true if and only if the argument passed to it is non-null and is a String object that represents the same sequence of characters as this object.
- Classes which do not implement equals() method inherits the method from Object *class*. Foex: StringBuffer *class* , java.lang.Number class.

The Math Class

- Java's Math *class* contains a collection of methods and 2 constants.
- The 2 constants are Math.PI and Math.E. They are declared to be *public*, *static*, *final* and *double*.
- The Math *class* is declared *final*, so it **cannot** be extended.
- The constructor for Math *class* is *private*, so it **cannot** be instantiated.

Strings

- **Strings are immutable**.
- String s1 = “immutable”; ➔ the string is added to the pool of strings of the Java class. The compiler does not create a new copy every time. Instead it uses the existing one from the pool.
- String s2 = *new* String(“immutable”); ➔ a totally new object is created. Extra memory is allocated to such strings.
- String comparison is case sensitive.
- Strings are not implemented as *char* array.
- String class if *final* by default and hence **cannot** be subclassed.
- String s = “Bicycle”; ➔ s.substring(1, 3);➔”ic”.
- StringBuffer and StringBuilder classes are mutable.
- StringBuffer doesn’t override equals method. Hence the method returns *true* only when comparing references to the same single object.
- StringBuffers are threadsafe. StringBuilders are **NOT**.
- StringBuffer and StringBuilder implement java.lang.Appedable.
- StringBuffer sb = “abcd”; ➔ invalid

The Wrapper Classes

- Wrapper classes **cannot** be used like primitives.
- All wrapper classes are immutable, i.e. once created their values cannot be changed.
- There is not wrapper class for String as wrapper classes are only for primitive types.
- Byte, Double, Float, Integer, Long, Short, BigDecimal, BigInteger all extend java.lang.Number.
- Void, Boolean and Character extend java.lang.Object.
- *new* Boolean(“TRUE”) produces a Boolean Object that represents *true*.
  new Boolean(“*true*”) produces a Boolean Object that represents *true*.
  new Boolean(“TrUe”) produces a Boolean Object that represents *true*.
  new Boolean(“anything”) produces a Boolean Object that represents *false*.
- Unlike other wrapper classes Boolean does not *throw* an exception if *null* is passed to its constructor. It represents *false*.
- All the wrapper classes except Character have *static* method called valueOf(String s). It returns a wrapper instance of the same type as the *class* whose method was called.
- The parseXXX() method is *static*. It takes String arguments and returns the corresponding primitive type. It throws NumberFormatException.
- The getXXX() method is *static*. It takes a String argument that is the name of a system property and returns the value of the property.

- Each **numeric** wrapper class defines the following set of "typeValue()" methods for converting the primitive value in the wrapper object to a value of any numeric primitive data type.
  - ✓ *public byte byte*Value()
  - ✓ *public short short*Value()
  - ✓ *public int int*Value()
  - ✓ *public long long*Value()
  - ✓ *public float float*Value()
  - ✓ *public double double*Value()
- String val = *null*;
  int x = Integer.parseInt(val); ➔NullPointerException
- When a String is passed to the parseInt(String s) method, it should have only digits, except that it can start with '-' to indicate negative.
- valueOf(String s) is a *static* method of *class* Integer but it is perfectly legal to invoke it on an Object reference.
- *boolean* by default is *false*, while Boolean by default is *null*, because Boolean is an object.
- For all standard wrapper classes (x.equals(y)) always returns *false* if x and y are objects of different classes, though they may wrap the same value.
- All the wrapper classes are *final* and implement Serializable *interface*.
- Java's wrapper classes are useful in situations where a primitive must act like an object.

The Collections Framework

- Collection is an *interface* not a *class*. (It's Collection and **NOT** Collections).
- The List and Set interfaces extend java.util.Collection, whereas Map does not.
- Vector and Hashtable are the 2 collection classes that are threadsafe or *synchronized*.
- Collections classes like ArrayList and HashMap must be wrapped in Collections.synchronizedList or Collections.synchronizedMap if synchronization is desired.
- The 6 core interfaces in Java's collections framework are Collection, Set, SortedSet, List, Map, and SortedMap.

Lists

- List keeps its elements in order in which they were added.
- Vector, Stack, ArrayList and LinkedList are the classes which implement List.
- java.util.List *interface* contains a few methods in addition to those inherited from the java.util.Collection superinterface.

Sets

- Sets may **not** contain duplicate elements.
- Sets – The add method returns *false* if you attempt to add an element with duplicate value. No exception is thrown.
- All methods defined in Set are also defined in java.util.Collection it does **not** contain any additional methods of its own.
- Set allows atmost one *null* element. Though some implementations of set *interface* prohibits addition of *null* elements.
- HashSet offers constant time performance for basic operations like add, remove, contains and size.
- Class TreeSet *implements interface* java.util.SortedSet.
- TreeSets rely on all their elements implementing the java.lang.Comparable *interface*.
- HashSet allows '*null'* element
- HashSet maintains a doubly linked list.
- java.util.LinkedHashSet *extends* HashSet and provides constant time performance.

Maps

- Java Map – An Interface that ensures that implementing classes cannot contain duplicate keys.
- java.util.LinkedHashMap can be used for maintaining the entries in order in which they were last accessed.
- In identityHashMap 2 keys k1 and k2 are equal if and only if (k1 == k2). It violates the general contract of *interface* Map.
- HashMap and Hashtable have keys which can be any object.
- HashMap allows '*null'* as a valid key or value Hashtable doesn't. It throws NullPointerException.
- Hashcode may be negative.
- WeakHashMap is a Hashtable based Map implementation.
- The entries of the class java.lang.WeakHashMap can be Garbage collected without you explicitly deleting them.