Sun Certification for Java Programmer 1.4 Exam Quick Reference Guide

Part 2

Vrukesh V. Panse
Email: vrukesh.panse@wipro.com

Embedded & Product Engineering Solutions
Mobile Computing Group

Abstract

This document is a quick reference to the concepts covered in the Sun Certified Programmer for Java 2 Platform 1.4 exam. The document attempts to cover all the aspects of the Certification exam. The purpose of the document is not to discuss in-depth details. Hence very few examples which contain code have been provided. This document is best read after a first round study of the Java programming language.

Information in this document has been obtained from sources believed to be reliable. Still, the author does not guarantee the accuracy or completeness of any technical information.

Please feel free to contact me for any technical query related to the document at mailto:vrukesh.panse@wipro.com

Introduction

To quote from the Sun Java website, "*Getting certified is a great way to invest in your professional development and to help boost your career potential*." I prepared for the Sun Certification for Java Programmer 1.4 Exam with the same in my mind. On my way to achieving the certificate, I made use of several sources of information like website links, books, help documents…

I found it difficult to find a single source of information. Every website link, every book, every document had its own pros and cons. I prepared this document with single objective in mind: to find the concepts to study for the exam at a single source.

This document is a quick reference guide and not an exhaustive resource of information. This document is best read after a first round study of the Java programming language. It will do no good for the layman's knowledge. While preparing the document, it is assumed that the reader knows all the basic terminology associated with Java.

The objectives of the exam cover a vast strecth of the Java programming language. No wonder, I could not keep the Quick Reference short!!! So, to improve the readability, the complete document is split into two parts.

This is part 2 of the complete document.

Index

SCJP Quick Reference Guide

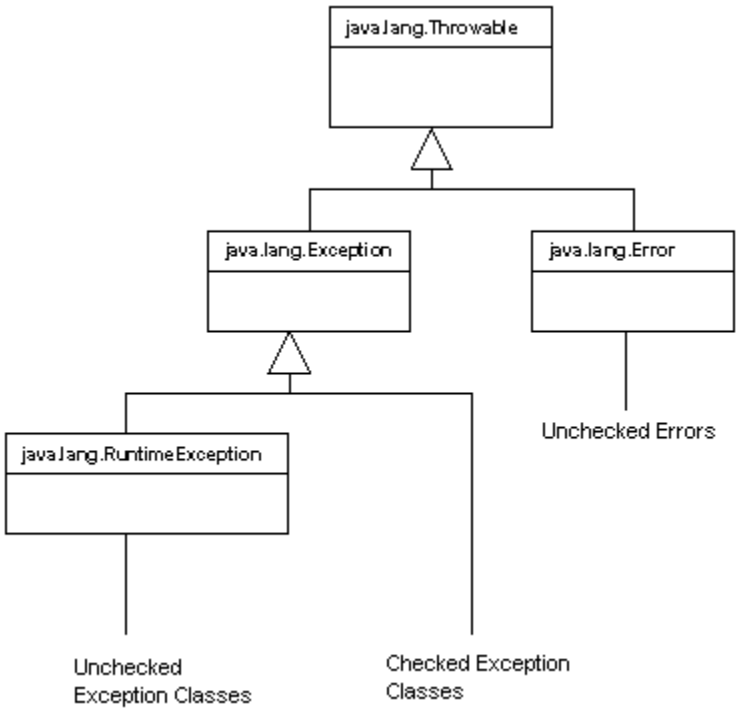
Exceptions

- Whenever a program violates the syntax/semantics of Java, the JVM signals this error to the program as an exception.
- When an exception occurs, the flow is transferred from that point to a new point that can be specified by the programmer.
- An exception is said to be thrown from the point where it occurred and is said to be caught at the point to which control is transferred.
- Programs can also throw the exceptions explicitly, using throw statements.
- Throwing an exception consists of creating an exception object and handling it to the java runtime system. Only objects of java.lang.Throwable class can be thrown by JVM or by using throw statement. For example, throw new SomeException("<Exception message>");

  //SomeException must be the subclass of java.lang.Throwable.
- Programmers can use the exception-handling constructs like (try-catch and try-catch-finally) to manage foreseen errors.
- With exception handling, Java separates the error handling code from regular code, making the code more clean, readable and manageable.
- In Java, unreachable code generates compile-time error.
- Local variable which is already declared in an enclosing block, is visible in a nested block. Therefore, that variable can not be re-declared inside the nested block.
- A local variable in a block may be re-declared in another local block, if the blocks are disjoint.

Errors and Exceptions

- There are two types of exceptions, an Error and an Exception.
- An Error and an Exception are the direct subclasses of the Throwable class.
- Errors are handled by java system. Instances of errors are results of internal errors inside the java runtime. Errors are rare and usually fatal.
- All programmer-declared exceptions must be the subclasses of Exception class.



Checked and Unchecked Exceptions

- **Checked Exceptions**
  - A checked exception is direct or indirect subclass of Exception excluding class RuntimeException and its subclasses.
  - Checked Exceptions are checked by the compiler to see if these exceptions are properly caught or specified. If not, the code will fail to compile.
  - Checked exception forces client program to deal with the scenario in which an exception may be thrown.
  - Checked exceptions must be either declared or caught at compile time.
- **Unchecked Exceptions**
  - Unchecked exceptions are RuntimeException and all of its subclasses.
  - Class java.lang.Error and its subclasses also are unchecked.
  - Unchecked Exceptions are not checked by the compiler. The compiler will not insist that client program/method should declare each exception thrown by a method or even handle it.
  - Runtime exceptions do not need to be caught or declared.
  - RuntimeException is a direct subclass of Exception. RuntimeException class and its subclasses represent exceptions, which may be thrown during normal operation of the JVM at runtime.

- ClassCastException occurs when the code attempts casting of an object to an incompatible type. For example,

```
Object anObject = new Integer(10);
System.out.println((String) anObject); // ClassCastException at runtime.
```

Handling an exception

For handling exception programmatically, enclose the statements that might throw an exception within a try block. General syntax of the try-catch-finally statements is:

```
try
{
. < Some statements which throw the exception>
        .
 }
catch ( Throwable exception )
{
 }
finally
{
 }
```

- **The try {..} block**
  - A try statement executes a block and oversees the execution of enclosed statements for exceptions. It also defines the scope for the exception handlers (defined in catch clause).
  - The try block must be accompanied by at least one catch block or one finally block.
- **The catch(Throwable throwable){..} - clause of try block**
  - The catch clause contains the exception handling code.
  - The catch statement takes only the objects of Throwable type as an argument.
  - There can be more than one catch block. The exception caught in them should be specific to generic from top to bottom. If a generic exception class (e.g. Exception, Throwable) is used to catch the exception before a specific class, all the exceptions can be caught by them and catch block below them become unreachable. Hence, the code will not compile.
  - Statements in a catch block are executed whenever a particular exception it is catching actually occurs within the try block.
- **The finally {..} clause**
  - A try block can have a finally clause which must come after all the catch clauses.
  - The finally clause always gets executed:

        - In case the try block completed normally i.e. no exception occurred within try block.
        - If exception occurs within try block and one of the catch blocks handled that exception.
        - In case, an exception occurred within try block that none of the catch block could handle.
    - The code in finally clause is always executed, unless the thread executing the try code dies. e.g. a call to System.exit().
- The finally clause helps in general cleanup and disposal of system resources at the time of try block exit.
- throws clause in method declaration. It is sometimes desirable for method to declare which exceptions it is going to throw instead of handling them. The client of the method can then decide how to handle the exception. The throws keyword is used to identify the list of possible exceptions that a method might throw.
- When an exception occurs in the program, and the care is not taken to deal with the exception by catching it, the execution of the program jumps to the end of the current method and the execution goes to caller of the method and again the execution jumps to the end of the called method. This goes on until the execution of the program reaches the top of the execution stack, where the thread dies.
- If the explicit exception handling is done, the thrown exception is caught in the catch block, where it can be handled properly. After that, finally block is executed, if one exists, and the execution continues after the end of the try-catch-finally block.
- ArithmeticException is thrown only in case of int and everything that is implicitly converted to int being divided by zero.

Method Overloading

- If two (or more) methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but different signatures, then the method name is said to be overloaded.
- The signature of a method consists of the name of the method and the number and types of formal parameters in particular order.
- A class must not declare two methods with the same signature, or a compile-time error occurs.
- Only the method name is reused in overloading, so method overloading is actually method name overloading.
- Overloaded methods may have arguments with different types and order of the arguments may be different.
- Overloaded methods are not required to have the same return type or the list of thrown exceptions.
- Overloading is particularly used while implementing several methods that implement similar behavior but for different data types.

- Overloaded methods are independent methods of a class and can call each other just like any other method.
- A static method can be overloaded by a non-static method.
- Constructor Overloading
  - Constructors can also be overloaded, as they are also methods.
  - One constructor can call another overloaded constructor of the same class by using this keyword.
  - One constructor can call constructor of its super class by using the super keyword.
  - For example, Java has some classes with overloaded constructors. The Wrapper class for int has two overloaded constructors. One constructor takes int as argument and another takes String argument.

```
Integer integerObject = new Integer(2);
Integer anotherintegerObject = new Integer("2005");
```

Method Overriding

- When a class defines a method with same method name, argument types, argument order and return type as a method in its super class, it is called method overriding.
- The method in the super class is said to be overridden by the method in the subclass.
- Overriding method actually replaces the behavior in super class for a subclass.
- **Rules for overriding**
  - Methods overriding cannot be declared more private than the super class method. This means that the method cannot be overridden with more restrictive access modifier.
  - Any exceptions declared in overriding method must be of the same type as those thrown by the super class, or a subclass of that type.
  - Methods declared as final cannot be overridden.
  - An overriding method can be declared as final as the keyword final only suggests that this method cannot be further overridden.
  - Methods declared as private cannot be overridden as they are not visible outside the class.
  - A static method cannot be overridden by a non-static method.
  - A static method can be overridden by a static method.
  - A static variable can be overridden by a non-static variable in the child class.
  - synchronized methods can be overridden to be non-synchronous. synchronized behavior affects only the original class.
- Overriding method can call the overridden method (just like any other method) in its super class with keyword super. The call super.method() will invoke the method of immediate super class.

- Though keyword super is used to refer super class, method call super.super.method() is invalid.
- The direction shown in the figure below is only permitted from the parent classes to sub-classes for overriding:

  private -> default -> protected -> public
  parent classes sub classes

Difference between method overloading and method overriding

Sr. No.	Method Overloading	Method Overriding
1	Overloaded methods supplement each other.	An overriding method replaces the method it overrides.
2	Overloading methods may have different argument list of different types.	An overriding method must have argument list of identical type and order.
3	The return type may be different for overloaded methods.	The return type must be same as that of the overridden method.
4	Since overloading methods are essentially different methods, there are no restrictions on the exceptions they can throw.	The overriding method must not throw checked exceptions which cannot be thrown by the original method.
5	Accessibility may vary freely.	Overriding methods ca not be more private than the overridden method.
6	There is no limit on the number of overloaded methods a class can have.	Each parent class method may be overridden at most once in any sub-class. That is, there can not be two identical methods in the same class.

- Variables are shadowed and methods are overridden.
- When a class member variable is accessed through an object reference, the variable to be selected depends on the declared class of the object reference variable. However, when a method is called on through an object reference, the actual method invoked depends on the type of an object reference.

- A subclass can have a variable with the same name as a variable in the parent class. This is called as shadowing the parent class variable. It is not overriding of variable.
- Late Binding : In any method call, actual method to invoke is determined by the 'Class' (or type) of an object on which the method is called and not on the type of variable holding the object reference. In addition, this is decided at runtime.
- Method is bound at run time, where as variables are bound at compile time. It is the type of object which tells which method is to be called. It is the type of reference which tells which variable is to be called.
- Methods access the variables only in the context of the class of the object they belong to. If a sub-class method calls explicitly a super class method, the super class method always will access the super class variable. Super class methods will not access the shadowing variables declared in the sub classes, since they do not know about them. But, the method accessed will be again subject to the dynamic look-up. It is always decided at the run-time which implementation is called.
- Only static methods are resolved at compile-time.
- It is not at all possible to access a method in a super-super-class from a sub class.
- private members are not inherited, but they do exist in the sub classes.
- Since the private methods are not inherited, they are not overridden.
- A method in a sub class with the same signature as a private method in the super class, is essentially a new method, independent from the super class, since the private method in the super class is not visible in the sub class.

'is-a' relationship - Generalization (Inheritance)
Generalization is the result of abstracting the common attributes and behavior from a group of closely related classes and moving them into a common super class. A common test for generalization is the IS-A test. Generalization may also look for following phrases in problem definition to identify the inheritance.

- "is a"
- "is a type of"
- "is a kind of"

For example, if part of problem definition says "Dog is an animal", implementation will more likely have Animal class and Dog class will be the subclass of the Animal class.

'has-a' relationship - Composition
Composition is the relationship between a class and its constituent parts. Composition passes the "HAS-A" test. For example, "An Automobile has an Engine".

Inner Classes

An inner class is a type of nested class that is not explicitly or implicitly declared static.

- An inner class can be a subclass of the outer class.
- Inner classes can be declared as private, protected and static.
- An interface can be defined inside a class just like an inner class.
- One enclosing class can have multiple instances of inner classes.
- Inner classes can have synchronized modifier for their methods. But, calling those methods obtains lock for the inner object only, not the outer object.
- If it is need to synchronize an inner class method based on outer object, outer object, outer object lock must be obtained explicitly. Locks on outer and inner objects are independent.
- An inner class variable can shadow an outer class variable. In this case, an outer class variable can be referred as (outerclassname.this.variablename).
- Outer class variables are accessible within the inner class, but they are not inherited. They do not become the members of inner class.
- Outer class can not be referred using super and outer class variables can not be access using this.
- An inner class variable can shadow an outer class variable.

Types of Inner class are:

- **Local inner class**
  - A local class is a nested class that is not a member of any class and that has a name.
  - Every local class declaration statement is contained by a block.
  - The scope of a local class declared in a block is within the rest of the block.
  - Local class defined in a method has access to method variables which are final and also to the outer class's member variables.
  - Local inner class cannot de declared public, protected or private.
  - Local inner class can implement interfaces.
- **Anonymous inner class**
  - Anonymous inner class does not have name.
  - It can extend a class or implement an interface but cannot do both at the same time.
  - The definition, construction and first use of an anonymous inner class is at same place.
  - Programmer cannot define specific constructor for anonymous class, but can pass arguments (to call implicit constructor of its super class.)
  - An anonymous class is never abstract.
  - An anonymous class is always an inner class.
  - An anonymous class is never static.

  - An anonymous class is always implicitly final.
  - Anonymous class defined in a method has access to method variables which are final and also to the outer class's member variables.
- **Non-static member class**
  - A member class is an inner class whose declaration is directly enclosed in another class or interface declaration.
  - Member inner class can be public, private, protected or default/friendly.
  - Non-static member inner class has access to member variables of the outer class.
  - Non-static member inner class can be instantiated on the instance of the outer class.
  - Non-static inner class cannot have static members.
  - Non-static inner class can have static final variable, since it will be treated as constant by the compiler.
  - When the inner class is non-static, enclosing class instance is required to access the inner class member.
- **Static nested classes**
  - Nested class can be static. It has access to only static member variables of the outer class.
  - static nested class may be instantiated / accessed without the instance of the enclosing class. It is accessed just like any other static member variable of a class.
  - An object of the static inner class cannot be made directly in the static method of outer class. It has to be referred with the reference of object of outer class.
- **Inner classes within method**
  - Inner classes within a method can not be static.
  - Inner classes within a method can have only default access. The public, protected or public modifiers are not allowed for the inner classes within a method.
  - Inner classes within a method can have final or abstract modifiers.
  - Inner classes within a method can access any data member from the enclosing class.
  - Inner classes within the methods can access only the final variables in the parameter list or the final variable defined inside the enclosing method.

Entity	Declaration context	Accessibility modifiers	Outer instance	Direct access to enclosing context	Defines static or non-static members

package level class	as package member	public or default	No	N/A	both static and non-static
Top-level static nested class	as static class member	All	No	static members in enclosing context	both static and non-static
Non-static inner class	as non-static class member	All	Yes	all members in enclosing context	only non-static
Local class (non-static)	in block with non-static context	None	Yes	all members in enclosing context and local final variables	only non-static
Local class (static)	in block with static context	None	No	static members in enclosing context and local final variables	only non-static
Anonymous class (non-static)	in block with non-static context	None	Yes	all members in enclosing context and local final variables	only non-static
Anonymous class (static)	in block with static context	None	No	static members in enclosing	only non-static

				context and local final variables	
Package level interface	as package member	public or default	No	N/A	static variables and non-static method prototypes
Top-level interface (static)	as static class member	all	No	static members in enclosing context	static variables and non-static method prototypes

Threads

A Java thread is an execution context or a lightweight process. It is a single sequential flow of control within a program. Programmer may use java thread mechanism to execute multiple tasks at the same time.

Thread class and run() method

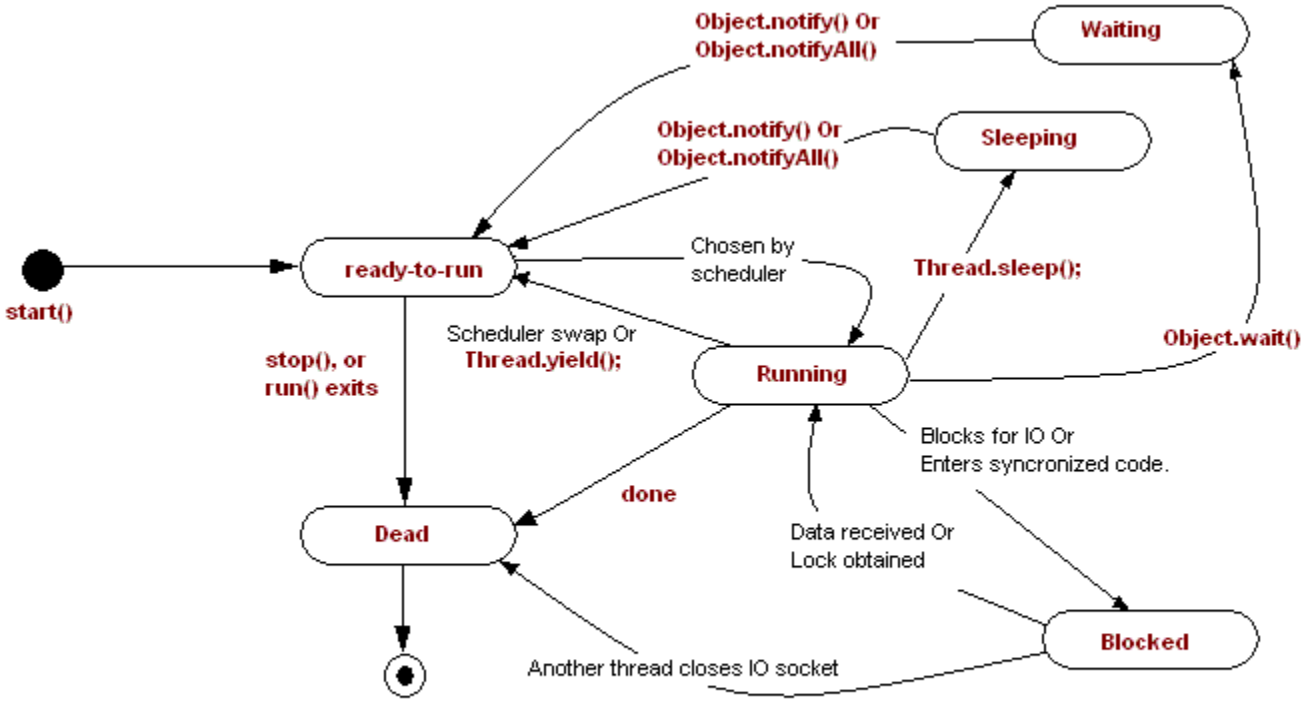
- Basic support for threads is in the java.lang.Thread class. It provides a thread API and all the generic behavior for threads. These behaviors include starting, sleeping, running, yielding, and having a priority.
- Thread class implements the Runnable interface.
- The run() method is defined in the Runnable interface.
- When the Thread class is extended, it is not compulsory to implement the run() method. If the run() method is not provided, the run() of the Thread class is executed, which does nothing.
- The signature of the run() method is:

  public void run()
- The run() method gives a thread something to do. Its code should implement the thread's running behavior. There are two ways of creating a customized thread:
  - Sub classing java.lang.Thread and Overriding run() method.
  - Implementing the java.lang.Runnable Interface.
- The Thread will not be created without calling the start() method. Simply calling run() method is not sufficient for creating a thread.

- Calling the start() method will implicitly call the run() method.

The Life Cycle of a Thread

The following diagram illustrates the various states that a Java thread can be in at any point during its life and which method calls cause a transition to another state.



- Ready-to-run
  A thread starts its life cycle with a call to start(). For example,

```
MyThread threadObject = new MyThread();
threadObject.start();
```

A call to start() will not immediately start thread's execution but rather will move it to pool of threads waiting for their turn to be picked for execution. The thread scheduler picks one of the ready-to-run threads based on thread priorities.

- **Running**
  The thread code is being actively executed by the processor. It runs until it is swapped out, becomes blocked, or voluntarily give up its turn with this static method Thread.yield(). Even if it is called on any thread object, it causes the currently executing thread to give up the CPU.

- **Waiting**
  A call to java.lang.Object's wait() method causes the current thread object to wait. The thread remains in "Waiting" state until some another thread invokes notify() or the notifyAll() method of this object. The current thread must own this object's monitor for calling the wait().
- **Sleeping**
  Java thread may be forced to sleep (suspended) for some predefined time. The signature of the sleep method is Thread.sleep(milliseconds) and Thread.sleep(milliseconds, nanoseconds). Static method sleep() only guarantees that the thread will sleep for predefined time and be running some time after the predefined time has been elapsed. For example, a call to sleep(60) will cause the currently executing thread to sleep for 60 milliseconds. This thread will be in ready-to-run state after that. It will be in "Running" state only when the scheduler will pick it for execution. The thread will run some time after 60 milliseconds.
- **Blocked on I/O**
  A Java thread may enter this state while waiting for data from the IO device. The thread will move to Ready-to-Run after I/O condition changes (such as reading a byte of data).
- **Blocked on Synchronization**
  A Java thread may enter this state while waiting for object lock. The thread will move to Ready-to-Run when a lock is acquired.
- **Dead**
  A Java thread may enter this state when it is finished working. It may also enter this state if the thread is terminated by an unrecoverable error condition.

Thread Synchronization

Problems may occur when two threads are trying to access/modify the same object. To prevent such problems, Java uses monitors and the synchronized keyword to control access to an object by a thread.

- **Monitor**
  - Monitor is any class with synchronized code in it.
  - Monitor controls its client threads using, wait() and notify() (or notifyAll()) methods.
  - wait() and notify() methods must be called in synchronized code.
  - Monitor asks client threads to wait if it is unavailable.
  - Normally a call to wait() is placed in while loop. The condition of while loop generally tests the availability of monitor. After waiting, thread resumes execution from the point it left.
- **Synchronized code and Locks**
  - Each Object has a lock. This lock can be controlled by at most one thread at time. Lock controls the access to the synchronized code.

- When an executing thread encounters a synchronized statement, it goes in blocked state and waits until it acquires the object lock. After that, it executes the code block and then releases the lock. While the executing thread owns the lock, no other thread can acquire the lock. Thus the locks and synchronization mechanism ensures proper execution of code in multiple threading.

Conditions that might prevent the thread from executing:

- Methods like wait(), sleep() and yield().
- Entering the synchronized region can force the thread to pause until it obtains a lock on the object.
- CPU scheduling can cause the thread to pause for some time.
- deadlocks can cause the thread to pause. If two threads are unable to recover from a deadlock, they will be in that state forever.
- deadly embrace is a scenario, where one thread has a lock on object A and waiting to acquire lock on object B. At the same time, another thread has a lock on object B and waiting to acquire lock on object A.

Thread Priority

A thread's priority is specified with an integer from 1 (the lowest) to 10 (the highest), Constants Thread.MIN_PRIORITY and Thread.MAX_PRIORITY can also be used. By default, the setPriority() method sets the thread priority to 5, which is the Thread.NORM_PRIORITY.

```
Thread threadObject = Thread.currentThread();
int currentPriority = threadObject.getPriority();
threadObject.setPriority( currentPriority + 1 );
```

- If the thread is holding a lock and went to a sleeping state, it does not loose the lock.
- Java does not provide any mechanisms for detection or control of deadlock situations, so the programmer is responsible for avoiding them.

Object class

Class java.lang.Object is the root of the java class hierarchy. Every class has Object as a super class. All objects, including arrays, inherit the methods of this class.
Important methods in this class are:

- toString() method of Object class returns string representation of the object. The signature of the method is:

```
public String toString()
```

- equals() method of Object class compare the object references. The method only compare object references, it returns **true** only when both object references are pointing to the same object. The signature of the method is:

  public boolean equals(Object o)
- hashCode() method of Object class returns distinct integers for distinct objects. This is typically implemented by converting the internal (memory) address of the object into an integer. The syntax of the method is:

  public native int hashCode()
- The relation between equals() method and hashcode() is such that if two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same integer result.
- It is generally necessary to override the hashCode() whenever equals() is overridden.
- Object class has supporting methods for java threading mechanism. These methods are namely wait(), notify and notifyAll().

Math Class

- Math is final class, so it cannot be sub classed.
- Math class cannot be instantiated since all methods are static.
- All constants and methods of Math class are public and static. They can be accessed by using the class name Math.
- The constructor of the Math class is private.
- random() method of Math class returns a value of type double such that

  0.0 <= value < 1.0
- abs() returns the absolute value of argument value. abs() is overloaded for int, long, float and double. The signature of the method is:

  public static int abs(dataType dataTypeValue)
- Truncating (floor) and rounding(ceil,round) functions. ceil() method returns the smallest (closest to negative infinity) double value that is not less than the argument and is equal to a mathematical integer. floor() method returns the largest (closest to positive infinity) double value that is not greater than the argument and is equal to a mathematical integer. round() method is overloaded for float and double arguments. It rounds to the nearest integer and return value of type int or long.
- The ceil() and floor() methods accept double as the parameter. ceil() and floor() are not overloaded for other data types. The signature of the methods is:

public static double ceil(double value)
public static double floor(double value)

- round() is overloaded for float and double arguments. It rounds to the nearest integer and returns value of type int or long.
- Trigonometric functions sin(), cos(), tan() returns the trigonometric sine, cosine and tangent of the angle respectively. These methods take the double argument and return a double value.
- All the methods that implement Trigonometry are native.
- All the Trigonometric methods take the angle input in radians.

Angle in Degrees * PI / 180= Angle in Radian

- sqrt() returns the correctly rounded positive square root of a double value. If the argument is NaN or less than zero, then the result is NaN.
- Anything modulus 0.0 or -0.0 is NaN.
- Anything divide by 0.0 is Infinity.
- Anything divide by -0.0 is –Infinity.
- 0.0 or -0.0 divide by 0.0 or -0.0 is NaN.
- 0.0 divide by anything is 0.0.
- -0.0 divide by anything is -0.0.

Wrapper classes

- java.lang package provides standard library classes that are closely related to primitives. These are called as Wrappers.
- These classes are Byte, Short, Character, Integer, Long, Float, Double and Boolean.
- Wrapper class names differ from the primitives only in the initial upper case letter.
  Exceptions are Integer, which wraps **int** and Character, which wraps **char** values.
- All wrapper objects can be constructed by passing a String except Character wrapper class for char data type. Most of these constructors throw NumberFormatException, which is a runtime exception.
- Wrapper classes override equals() method. equals() on wrappers return false if both the objects are not instances of the same class and even when the value these objects are wrapping has the same numerical value.
- All wrapper objects are immutable. Once an object is created, the wrapped primitive value cannot be changed.
- Wrapper classes are **final** and hence cannot be sub classed.

String Class

- java.lang.String is a **final** class hence it cannot be sub classed.
- String object represents a sequence of 16-bit Unicode characters.

- Strings are immutable which means a String object has a constant (unchanging) value.
- String literals are created and stored on the pool of literals.
- String objects are immutable. Hence, none of the methods of the String class will modify the String object.
- equals() in String class compares Strings. String's equal() method checks if the argument if of type string, if not it returns false. It returns true if the argument is not null and is a String object that represents the same sequence of characters as this object.
- The statement will create two String objects.

```
String strObject1= new String( "MyString" );
String strObject2 = strObject1;
```

Here, both strObject1 and strObject2 will contain the same reference, so both will be equal by the equality test(==).

- The statement will create two String literals.

```
String strObject1 = "MyString";
String strObject2 = "MyString";
```

Here, both strObject1 and strObject2 are String literals, instead of String objects. So, once "MyString" for strObject1 is created in the String pool, it will be reused in case of creating strObject2. It means that both will be equal by the equality test(==).

- The statement will create two String objects

```
String strObject1 = new String( "MyString" );
String strObject2 = new String( "MyString" );
```

Here, both strObject1 and strObject2 are String objects. Both are created at different memory locations on the heap. So, both will not be equal by the equality test(==), though they point to the same String literal in the String pool.

StringBuffer

- java.lang.StringBuffer is a **final** class hence it cannot be sub classed.
- It is mutable, which means its value can be changed.

- StringBuffer class does not override the equals() method. Therefore, it uses Object class' equals(), which only checks for equality of the object references. StringBuffer equals() does not return true even if the two StringBuffer objects have the same contents.

Collection

A collection is an object that represents a group or collection of objects. The collections framework consists of collection interfaces and some general purpose implementations of these interfaces.

Collection Interfaces

There are six collection interfaces. The most basic interface is Collection. Three interfaces that extend Collection are Set, List and SortedSet. The other two collection interfaces, Map and SortedMap, do not extend Collection, as they represent mappings rather than true collections. Collection interfaces can be categorized as:

- Set is an interface, which models mathematical abstraction. Set provides an API for a collection that contains no duplicate elements. SortedSet interface extends this interface for ordered collection of unique elements.
- List is an interface for ordered collection. Lists typically allow duplicate elements. ArrayList, Vector, LinkedList are the some of the classes which implement this interface.
- Map is an interface for a collection of key-value pairs. It is not an ordered collection. Repetitions are not allowed. It also provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. Interface SortedMap extends this interface for ordered key-value pairs. Classes HashMap, Hashtable are some implementations of this interface.

Benefits and constraints of using different data structures:

COLLECTION TYPE	BENEFITS	CONSTRAINTS
Arrays	Data access is fast	Inefficient if number of elements grows.
	Good for ordered data, which is not changed or searched frequently.	Inefficient if an element is to be inserted in the middle of collection.

		Provides no special search mechanism.
Linked List	Allows efficient insertion / deletion at any location.	Slower while accessing elements by index.
	Allows arbitrary growth of collection.	Provides no special search mechanism.
	Applying order to the elements is easy.	
Tree	Allows efficient insertion / deletion at any location.	Ordering is peculiar and some comparison mechanism is required.
	Allows arbitrary growth of collection.	Searching is not efficient for unevenly distributed data.
	Better and efficient search mechanism for evenly distributed data.	
HashTable	Efficient searching.	Ordering is peculiar and some comparison mechanism is required.
	Allows arbitrary growth of collection.	Searching is not efficient for unevenly distributed data.
	Good access mechanism.	

Sites

- http://suned.sun.com/US/catalog/courses/CX-310-035.html
- http://in.sun.com/education/bundles/in/sun_cert.html
- http://www.javaranch.com/mock.jsp
- http://www.javaprepare.com/
- http://www.vivek.4mg.com/javacertification/jsexam.htm
- http://java.about.com/cs/javacertification/tp/topscjpexams.htm
- http://www.jchq.net/
- http://certification.about.com/library/quiz/java/blscjp14_intro.htm
- http://www.javacaps.com/scjp_netlinks.html
- http://www.geocities.com/velmurugan_p/
- http://www.javacaps.com/scjp_tutorial.html