# Sun Certification for Java Programmer 1.4 Exam Quick Reference Guide Part 1

**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 1 of the complete document.

# Index

# SCJP Quick Reference Guide

## Java Source File, Package, import statements and Class declaration

- Only one top-level public class is allowed per Java source file.
- Top-level classes can have only public or default access.
- Top-level classes can be final or abstract.
- Top-level classes cannot be decl`ared static.
- The name of the Java source file and the name of the top-level public class must be same.
- If there is not a single one top-level public class present in the Java source file, then the Java source file name can be anything.
- If there are more than one public class definitions, compiler will accept the class with the file's name and give an error at the line where the other class is defined.
- It is not required to have a public class definition in a file. In this case, the file's name should be different from the names of classes and interfaces in the file.
- Even an empty file is valid source file.
- Package statement, import statement and class definition must appear in the order given.
- Package declaration is optional. If the package declaration is absent, the classes are added in the default package after compilation.
- Keyword package is not an access modifier.
- Importing packages does not recursively import the sub-packages? Sub-packages are different packages and live within an enclosing package. Classes in the sub-packages cannot access the classes in enclosing packages with default access.

## Keywords and identifiers

- All Java keywords and reserved words are in lower - case.
- strictfp is a new keyword added in Java 2.
- true, false and null are not keywords in Java, they are reserved words.
- Identifiers must start with either letter, $ or _ (underscore) and can have letter, $, _ (underscore) or digit in it.
- Identifies cannot start with the digits (0-9).
- An identifier should not be Java keyword or reserved word.
- Identifiers can be of unlimited length.
- Since all the Unicode characters are valid characters in Java, the identifier can contain any character from the entire Unicode character set.
- '\u000A' and '\u000D' cause problem while compilation, because they do the line break.

**Primitive data types**

| Data Type | Size( in bits ) | Value – range | Default value |
|---|---|---|---|
| **byte** | 8 | -2^7 to ( 2^7 - 1 ) or -128 to 127 | 0 |
| **char** | 16 | 0 to ( 2^16 – 1 ) or \u0000 to \uFFFF | 0 |
| **short** | 16 | -2^15 to ( 2^15 – 1 ) | 0 |
| **int** | 32 | -2^31 to ( 2^31 – 1 ) | 0 |
| **long** | 64 | -2^63 to ( 2^63 – 1 ) | 0L or 0l |
| **float** | 64 | Float.MIN_VALUE to Float.MAX_VALUE | 0.0F or 0.0f |
| **double** | 64 | Double.MIN_VALUE to Double.MAX_VALUE | 0.0D or 0.0d |
| **boolean** | - | true, false | false |

- Primitive data types can be classified as signed integrals and unsigned integrals. Signed integrals represent positive as well as negative numbers.
- Negative integrals are represented in 2's complement form.
- char is the only unsigned integral type in Java and thus cannot represent a negative number.
- Primitive variable declared at the class level will be given the default value.
- If Box is the name of a class, then

  Box myBoxOject; //declare reference to object of Box class
  myBoxObject = new Box(); //allocate a Box object
- Object reference variables are initialized to null by default.

**Literals**

- byte, short, int, long, float and double are signed integrals. boolean and char are unsigned integrals.
- Only the class variables are automatically initialized to their default value. The method variables and the local variables (also called automatic variables) need to be explicitly initialized.
- There are two boolean literals: true and false. These are case-sensitive.
- Character literals are defined as the character in the single quotes.

  char charNumber = 'c';

- Character literals can also be represented using the hexadecimal values.

  char charNumber = '\u4468';

  Where 4468 is the hexadecimal value for some character.

- '\u' should be followed with four digits.
- Applying the ++ operator on the char values increments the ASCII value of the char value.
- Character literals can also be represented using special characters.

  char charNumber = '\r';

- By default, integral literals are of the data type int. Integral literals represent a decimal, octal or hexadecimal value.

  int intNumber = 28; //decimal value

- Integral literals also represent an octal value. An octal literal is identified when there is 0 (zero) at the start of the literal. Octal numbers are represented with 0 to 7 digits. Therefore, there is no number above 7 in the octal system.

  int intNumber = 034; //octal value

- Integral literals can also represent hexadecimal value. Hex literals are case-insensitive.

  int intNumber = 0x1a; //hexadecimal value
  int intNumber = 0x1A; //hexadecimal value
  int intNumber = 0X1a; //hexadecimal value
  int intNumber = 0X1A; //hexadecimal value
- float and double can also be decimal, octal or hexadecimal value.
- By default, floating-point literals are of the data type double.
- A declaration can not be labeled. If a declaration is labeled, the compile-time error will be generated.

## Arrays

- An array is fixed-size ordered collection of homogenous data elements.
- Following are the example of array declaration:

```
int[] intArray; //array declaration
intArray = new int[ 25 ]; //array construction at runtime
int[] intArray = new int[] { 1, 2, 3 }; //array declaration, construction and
// initialization at same time.
int[] intArray = new int[ 3 ] { 1, 2, 3 }; // invalid
```

- An array element created using the new keyword is initialized to its default value.
- An array index starts at 0. So, an array of 10 elements has elements at $0^{th}$ index to $9^{th}$ index. Here, the array[10] will not exist.
- length is property of the array and not a method.

```
int[] intArray = new int[ 25 ];
int arrayLength = intArray.length; //arrayLength will have value 25.
```

- java.lang.Object is the super class of an array. Array understands all method calls of java.lang.Object.
- Arrays in Java are static arrays. The size of the array is to be specified at the compile-time.
- Arrays with zero size can be created. For example, the args array of the main() method will be a zero element array if no command parameters are specified. In this case, args.length will be 0.

## Permitted modifiers

### private, protected, public

- private variables / methods have class level access.
- private methods are not visible in the subclass. It means that a method with the same name in the subclass is an altogether different method, instead of being override.
- protected variables / methods can be accessed in the same class, the same package and in the sub-classes.
- public variables / methods can be accessed from any package.
- Variables / methods that have no explicit access modifier associated with them have default access and can be accessed in the same class and same package.

### abstract

- Classes and methods can be abstract.
- If a class is abstract, no instance of that class can be created.

- If a method is abstract, the subclass should give the implementation for that method.
- Even if a single method of a class is declared abstract, the class itself has to be declared as abstract.
- A class declared abstract may not have not abstract method.

**final**

- Classes, variables and methods can be declared final.
- Class declared as final cannot be sub classed.
- If a variable is declared as final, the value contained in the variable cannot be modified.
- Methods declared as final cannot be overridden.
- For objects declared as final, the reference it stores cannot be changed, but the content of the object it is pointing to, can be changed.
- Method arguments marked final are read-only. Compiler error is generated if it is tried to assign the value to final arguments inside the method.
- Member variables marked final are not initialized by default. They have to be explicitly assigned a value at declaration or in an initializer block.
- static final variables must be assigned to a value in a static initializer block.
- Instance final variables must be assigned a value in an instance initializer or in every constructor.

**static**

- static can applied to nested classes, methods, variables, free-floating code –block (static initializer).
- static methods can access only static variables.
- Access by class name is a recommended way to access static methods/variables.
- static initializer code is run at class- load time.
- Local variables can not be declared as static.
- static methods cannot be abstract.
- Only static modifier can be used with the free-floating blocks. Not even synchronized modifier can be used with the free-floating blocks.
- static methods do not have access to the implicit variable called this.
- A static method may be called without creating an instance of its class.
- Only one instance of static variable will exist for any amount of class instances.
- static blocks are executed before instance blocks and constructors. That is why the non-static variables and methods cannot be accessed through a static block or method. But, static variables can be accessed through a non-static block or method.
- If the static method is being overridden, then method call is bound at the compile time, with the class whose reference variable is taken.

**transient**
- transient modifier applies only to class level variables.
- Local variables cannot be declared as transient.
- During serialization, an object's transient variables are not serialized.
- transient variables may not be final or static. But the compiler allows the declaration and no compile-time error is generated.

**synchronized**
- synchronized keyword can be applied to methods or parts of methods only.
- The synchronized keyword is used to control the access to critical code in multi-threaded programs.

**native**
- native applies only to methods.
- native can be applied to static methods also.
- A native method calls code in a library specific to the underlying hardware.
- native method is like an abstract method. The implementation of the abstract and native method exists some where else, other than the class in which the method is declared.
- native method can pass / return Java objects.
- A native method can not be abstract.
- A native method can throw exceptions.
- Keyword native can be used with synchronized and final, but not with abstract.
- System.loadLibrary is used in static initializer code to load native libraries. If the library is not loaded when the static method is called, an UnsatisfiedLinkError is thrown.

**volatile**
- volatile applies only to variables.
- volatile can be applied to static variables.
- volatile can not be applied to final variables.
- transient and volatile can not come together.
- volatile is used in multi-processor environments.

| Modifier | Class | Inner classes(Except local and anonymous classes) | Variable | Method | Constructor | Free-floating Code block |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| **public** | Y | Y | Y | Y | Y | N |
| **protected** | N | Y | Y | Y | Y | N |
| **default ( no access modifier )** | Y | Y (ok for all) | Y | Y | Y | N |
| **private** | N | Y | Y | Y | Y | N |
| **final** | Y | Y (except anonymous classes) | Y | Y | N | N |
| **abstract** | Y | Y (except anonymous classes) | N | Y | N | N |
| **static** | N | Y | Y | Y | N | Y |
| **native** | N | N | N | Y | N | N |
| **transient** | N | N | Y | N | N | N |
| **synchronized** | N | N | N | Y | N | Y |
| **volatile** | N | N | Y | N | N | N |

**Constructor**

- A default constructor is a constructor with empty argument list.
- A default constructor is automatically generated by the compiler if the class does not have one.
- If an explicit constructor is there in the class, the default constructor is not generated.
- Access mode of the generated constructor is public for public classes, and default for classes with any other access mode.
- If a subclass has a default constructor and a super class has an explicit constructor, the code will not compile, due to the automatic insertion of the super() in the generated constructor by the compiler.
- Parent constructor can be called by using super. <Constructor name (arguments)>.
- Overloaded constructor can be called by using this. <Constructor name (arguments)>.
- Constructors can be invoked only from within the constructors.
- Call to this(…) and super(…) must always be the first statement in the constructor.
- this(…) and super(…) cannot be used in the methods. They can be used only in the constructors.
- Class name can be used as the method name.

- When a method/constructor has an argument which matches two different methods/constructors definitions, then it will always call the most specific one. If the both the arguments are at the same level in the hierarchy chart, then compile-time error is generated.
- A constructor can not call the same constructor from within. Compile-time error is generated.
- Constructors do not have a return type.
- Constructor body can have an empty return statement. Though, void can not be specified with the constructor signature.
- Constructors are not inherited as normal methods. The constructors have to be defined in the class itself.
- Constructors can not be overridden, since they are not inherited.

**Order of code execution (when creating an object)**

- static variables initialization
- static initializer block execution ( in order of declaration, if multiple blocks found )
- constructor header ( super or this – implicit or explicit )
- instance variables initialization / instance initializer block execution
- rest of code in the constructor.

**Assertions**

- An assertion is a statement in the Java that enables to test assumptions about the program.
- Each assertion contains a boolean expression that will be true, when the assertion executes. If the expression is not true, the system will throw an error.
- The assertion statement has two forms:

```
assert Expression1;
assert Expression1: Expression2;
```

First form evaluates the boolean expression 'Expression1' and if it is false, throws an AssertionError with no detail message. The second form is used to provide the detailed message for the AssertionError. The system passes the value of 'Expression2' to the appropriate AssertionError constructor, which uses the String representation of the value as the detailed message of the error.

- By default, assertions are disabled.
- Assertions were introduced in JDK1.4.
- For bringing the assertion mechanism in effect, the class should be compiled with:

javac –source 1.4 <filename.java>

and then executing the class should be done as:

java –ea <class name>

- Once an assertion fails, there is no way to recover from the failure. The program is terminated.
- It is valid to explicitly throw Assertion Error by any subclass, or mention the same in the throws clause, as AssertionError,
- Do not use the assertions for argument checking in public methods.
- Do not use the assertions to do any work that the application requires for correct operation.
- Assertions are commonly used to check the pre conditions, post conditions and class invariants.
  - A pre condition is a constraint that must be true when a method is invoked.
  - A post condition is a constraint that must be true after a method completes successfully.
  - A class invariant is a constraint that tells what must be true about each instance of a class.

**Interfaces**

- All the data members of the interface are public, static and final by default.
- An interface can extend one or more interface, by using the keyword extends.
- An interface can have only public, default and abstract modifiers.
- An interface method can have only public, default and abstract modifiers.
- An interface is loaded in memory only when it is needed for the first time.
- A class which implements an interface needs to provide the implementation of all the methods in that interface.
- If implementation for all the methods declared in the interface are not provided, the class itself has to be declared abstract, otherwise the class will not compile.
- A class can implement more than one interface.
- If a class implements two interfaces and both the interfaces have identical method declaration, it is totally valid.
- If a class implements two interfaces and both the interfaces have identical method names and argument lists, but different return types, then the code will not compile.
- An interface can not be instantiated. An instance of any class that implements the interface can be referred by the object references that use interface type.

- An interface can not be native, static, synchronized, final, private or protected.
- The interface fields can not be private or protected.
- The transient variables and volatile variables can not be members of interfaces.
- extends keyword cannot be used after the implements keyword. The keyword extends must always come before the keyword implements (if used).
- Variables in the interface can not be transient or volatile.
- A class can shadow the variables it inherits from an interface, with its own variables.
- A top-level interface can not be declared as static or final.
- Declaring parameters to be final in the interface is at the method's discretion. This is not part of method signature.
- Classes can declare the methods to be final or synchronized or native, whereas in an interface they can not be specified like that. These are implementation details and interface need not worry about this.
- If an interface specifies an exception list for a method, then the class implementing the interface need not declare the method with the exception list.
- If an interface does not specify any exception list for a method, then the class can not throw any exceptions.
- All interface methods should have public accessibility when implemented in a class.
- A class can implement two interfaces that have a method with the same signature or variables with the same name.

**Argument passing during method calls**

- Arguments of primitive data types- when arguments of primitive data types are passed, first a copy is made and then it is passed. The original copy in the caller method remains unaffected by the called method.
- Object reference as an argument - when argument of object type is passed, a copy of object reference is passed. Therefore, calling method may change the object whose reference is passed as an argument.

**main() method**

- main() is the entry point of the Java application.
- Following are some valid main() method applications:

```
public static void main( String[] args ) {}
public static void main( String args[] ) {}
static public void main( String[] args ) {}
static public void main( String args[] ) {}
```

- main() method takes String array as the argument. The Java runtime system can pass command line parameters to the application through this array.
- For the following main method in the file MyProgram.java,

  public static void main( String[] args ) {}

  the execution of the program from the command – line as

  java MyProgram 12 50 add

  args[ 0 ] will be “12”, args[ 1 ] will be “50” and args[ 2 ] will be “add”.
- The JVM may exit only when all the non-daemon threads exit.
- The JVM may not exit when the main() method ends, as the main() may spawn non-daemon threads, which will prevent the JVM from exiting.
- The main() method is a non-daemon thread. The JVM will not exit till this thread ends.
- Any uncaught exception just ends the thread from which it was thrown. So, an uncaught exception from the main() method (if it is the only non-daemon thread) will exit the JVM.
- main() method can be declared as final.
- main() method can be overloaded.
- A class without main() method or different signature of the main() method will compile. But, it will throw run-time error.
- A class without the main() method can be run by the JVM, if the ancestor class of this class has main() method. i.e. the main() method is inherited.

**hashCode() method**

- The hashCode() method is defined in the Object class.
- The method returns the hash code for the object. The signature of the method is:

  public int hashCode()
- The hashCode method is defined to return distinct integers for distinct objects. This is typically implemented by converting the internal address of the object into an integer. But, this implementation technique is not required by the Java programming language.
- It is necessary to override the hashCode() method in each and every class which overrides equals() method. Otherwise, it will be violation of the general contract.
- Whenever the hashCode() method is invoked on the same object more than once during an execution of an application, the hashCode() method must consistently return the same integer, provided no information used in equals() comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

- If two objects are equal according to the equals( Object ) method, then calling the hashCode method on each of the two objects must produce distinct integer results.
- Equal objects must have equal hash codes.
- A good hash function tends to produce unequal hash codes for unequal objects.

**Garbage collection**

- Java provides built-in automatic memory management, or garbage collection (GC). Garbage collector automatically handles freeing of unused object memory behind the scenes by reclaiming an object only when it can prove that the object is no longer accessible to the running program.
- Garbage collection is a low-priority thread in Java.
- Garbage collection cannot be forced explicitly. Hence, garbage collection is not predictable. The call System.gc() does not force the garbage collection but only suggests that the JVM may make an effort to do garbage collection.
- JVM may do the garbage collection if it running short of memory.
- Garbage Collection is hardwired in Java runtime system. Java runtime system keeps the track of memory allocated. Java runtime system determines if memory is still usable by any live thread. If not, then garbage collection thread will eventually release the memory back to the heap.
- An object is eligible for garbage collection when no object refers to it. An object also becomes eligible when its reference is set to null. Actually all references to the object should be null for it to be eligible.

  ```
  Integer intObject = new Integer (7);
  intObject = null;
  ```

- The objects referred by method variables or local variables are eligible for garbage collection when they go out of scope (i.e. when the method or their container block exits).
- The garbage collection algorithm in Java is vendor-implemented.

**finalize() method**

- All the objects have the finalize() method. This method is inherited from the Object class.
- finalize() method is used to release the system resources other than memory (such as file handles and network connections).
- The signature of the finalize() method is

  ```
  protected void finalize() throws Throwable {}
  ```

- finalize() method is called only once for an object. If any exception is thrown in the finalize() method() the object will still be eligible for garbage collection.
- finalize() method can be called explicitly. But, it does not garbage collect the object.
- finalize() method can be overloaded, but only the original method will be called by gc.
- finalize() method is not implicitly chained. A finalize() method in subclass should call finalize() method in super class explicitly. But the compiler does not enforce this check.

**Operators and Operand types for the operators**

Operands are used to indicate some operation on operands. Depending on the number of operands, operators can be broadly classified as:

- **Unary Operators**
  The unary operators operate on only one operand. Java unary operators are **+, -, ++, --, ~, !** and cast operators.
  - ~ is a unary bitwise inversion operator. It works on integral data types (byte, short, char, int, long). It does not work on boolean data type.
  - ! is a unary bitwise inversion operator. It does inversion for boolean data type.
  - Other unary operators work with the integral data types.
  - ++ is called increment operator. The increment operator increases its operand by one. -- is called as decrement operator. The decrement operator decreases its operand by one.
  - The increment operator and decrement operator are unique in that they can appear both in prefix form and postfix form.
  - In the prefix form, the increment operator and the decrement operator precede the operand. In this form, the operand is incremented or decremented before the value is obtained for use I the expression.
  - In the postfix form, the increment operator and the decrement operator follow the operand. In this form, the previous value of the operand is obtained for use in the expression and then the operand is modified.
  - In the prefix form,

    ```
    int intNumber = 42;
    int intAnotherNumber = ++intNumber;
    ```

    Here, the number intAnotherNumber is set to 43 and the number intNumber is set to 43. The line is equivalent to

    ```
    int intNumber = intNumber + 1;
    int intAnotherNumber = intNumber;
    ```

```
int intNumber = 42;
int intAnotherNumber =  --intNumber;
```

Here, the number intAnotherNumber is set to 41 and the number intNumber is set to 41. The line is equivalent to

```
int intNumber = intNumber - 1;
int intAnotherNumber = intNumber;
```

- In the postfix form,

```
int intNumber = 42;
int intAnotherNumber = intNumber++;
```

Here, the number intAnotherNumber is set to 42 and the number intNumber is set to 43. The line is equivalent to

```
int intAnotherNumber = intNumber;
int intNumber = intNumber + 1;
```

```
int intNumber = 42;
int intAnotherNumber =  intNumber--;
```

Here, the number intAnotherNumber is set to 42 and the number intNumber is set to 41. The line is equivalent to

```
int intAnotherNumber = intNumber;
int intNumber = intNumber - 1;
```

- **Binary Operators**
  The binary operators operate on two operands. Java binary operators can be further classified as:
  - Arithmetic operators
    The operators are related to arithmetic operations. *, /, %, + and - are the arithmetic operators in Java.
    - % operator is generally used with the integers, but can be used with the floating- point numbers also.
    - % and / operator never produce an ArithmeticException when used with the floating-point numbers.
    - All the Arithmetic operators produce results of int, long, float or double type only.
  - **Relational Operators**
    These operators perform comparison operation. <, >, <= and >= does numeric comparison. instanceof operator does the (Object's) type comparison. The type of a relational expression is always boolean.

- **Equality Operators**
  The == (equal to) and != (not equal to) operators are analogous to the relational operators. These operators also perform comparison operation. However, they have lower precedence over relational operators. The result type of an equality expression is always boolean.
- **Bitwise and Logical Operators**
  The bitwise operators and logical operators include the AND operator &, exclusive OR operator ^, and OR operator |. Bit wise binary operators & (AND), | (OR) and ^ (XOR) work on all integral data types and boolean data type.
- **Short circuit && (AND) and || (OR)**
  && and || are also known as Conditional-And and Conditional-OR operations. These operators are introduced for optimization. They work only on boolean operands.
  - In case of ||,

    (expr1) || (expr2) //expr2 is never evaluated if expr1 is true.
  - In case of &&,

    (expr1) && (expr2) //expr2 is never evaluated if expr1 is false.
- **Shift Operators**
  The shift operators include left shift <<, signed right shift >>, and unsigned right shift >>>; There is no such thing as unsigned left shift (<<<).
  - The << operator shifts left. 0 (Zero) bits are introduced at the LSB position.
  - The >> operator performs a signed or arithmetic right shift. The result has 0 (Zero) bits at the MSB, if the original left-hand operand is positive, and has 1 (One) bits at the MSB if the original left-hand operand is negative.
  - The >>> operator shifts right with 0 (Zero) bits introduced at the MSB. The >>> operator changes the initial negative sign (i.e. 1 at the MSB according to the 2's complement) to positive.
  - Using >> operator is equal to dividing the number by 2.
  - Using << is equal to multiplying the number by 2.
  - Using >> and >>> operators yield the same results when the number is positive. The result has 0 (Zero) bits at the MSB.
  - Shift operators never shift more than the number of bits the type of result can have. If the number of shift is greater than or equal to the number of bits in the type of result, RHS operand of the shift operator is reduced to

    RHS op x

where x is the number of bits in type of result (i.e. for int 32 bits and for long 64 bits) and op is the operator.

- **Conditional Operators**
  && (Conditional-And Operator) and || (Conditional-Or Operator). Each operand must be of type boolean. The operands may be expressions evaluating to boolean.
- **Assignment Operator**
  = is simple assignment operator. Assignment operation returns value.

```
int firstNumber = 2;
int secondNumber = 3;
int thirdNumber;
thirdNumber = firstNumber = 2;
```

In above example cNumber is assigned the value 2, because the assignment (firstNumber = 2) returns 2, which is then assigned to thirdNumber.

```
if( a = b)
{
    // do something;
 }
```

The above expression has = and not == operator. Expression (a = b) will be invalid if a and b are of integral types. Because then assignment a = b will return value of that type. However, if a and b are of type boolean, then assignment a = b will return boolean and therefore, if (a = b) will be valid.

- **Ternary Operator – the conditional operator ?:**
  This operator has three-operand expressions. ? appears between the first and second expressions, and : appears between the second and third expressions. The first expression must be of type boolean. The boolean value of first expression decides which of two other expressions should be evaluated. The conditional operator ? : is used in expression as ...

```
a = x ? b : c;
```

The above expression is equivalent to

```
if( x )
{
    a = b;
}
else
```

```
{
    a = c;
 }
```

Expression b and c must result into compatible data types to that of Expression x must be of boolean type.

- **String concatenation operator +**
  + operator works on string operands. If at least one operand expression is of type java.lang.String, then string conversion is performed on the other operand to produce a String at run time. The result is a reference to a newly created String object that is the concatenation of the two string operands. For example,

  System.out.println(2 + " is a company");

  In above example, primitive 2 is converted to String and then after concatenation String "2 is a company" is printed on console.

  Following is the behavior of the String objects, when used with other types in a println statement:

```
System.out.println( 15 + 5 ); //20
System.out.println( "Hello " + 5); //Hello 5
System.out.println( 5 + " Hello" ); //5 Hello
System.out.println( "Hello " + 15 + 5 ); //Hello 155
System.out.println( 15 + 5 + " Hello" ); //20 Hello
```

- **Compound Operators**
  *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^= and |= are compound assignment operators. Compound assignment expression of the form expr1 op= expr2 is equivalent to

  expr1 = (Type)( ( expr1 ) op ( expr2 ) );

  'Type" is the type of expr1. 'op' is the operator. Thus, compound operators have casting implied in it. This type of casting is mostly the narrowing conversion. For example, the following code

  byte byteNumber += 3

  is correct because byteNumber += 3 is same as

  byteNumber = (byte) (byteNumber +3);

```
byte byteNumber = 2;
byteNumber += 3;      // Valid
byte byteNumber = 2;
```

byteNumber = byteNumber + 3; // Invalid, will throw compile time exception.

In expression (byteNumber + 3), byteNumber is promoted from byte to int. So the result of (byteNumber + 3) is int. Since int value cannot be stored into byte variable, exception will be thrown.

**instanceof operator**

- instanceof operator can be used to determine if a reference is an instance of a particular class or interface.
- instanceof operator tests the class of object at run-time.
- instanceof operator returns true if the object denoted by LHS can be cast to the RHS type.
- For the instanceof operator, LHS should be an object reference expression, variable or an array reference. RHS should be a class ( abstract classes are also fine ), an interface or an array type.
- If the LHS and RHS are unrelated, a compile-time error is generated.
- instanceof operator returns true if LHS is a class or subclass of RHS.
- instanceof operator returns true if LHS implements RHS interface.
- instanceof operator returns true if LHS is an array reference and of type RHS.
- instanceof operator always returns false if LHS is null.
- If an object is an instance of a class, it is an instance of all the super classes of this class.
- If the statement

  xObject instanceof YClass

  is not allowed by the compiler, then

  Yclass yObject = ( YClass )xObject

  is not a valid cast expression.
- If the statement

  xObject instanceof YClass
  is allowed by the compiler and returns false, then

  Yclass yObject = ( YClass )xObject

  is a valid cast expression, but throws a ClassCastException at run time.
- If the statement

  xObject instanceof YClass

  is allowed by the compiler and returns true, then

Yclass yObject = ( YClass )xObject

is a valid cast expression and runs fine.

**Operator associativity**

- Operands are evaluated from left to right.
- The operations are executed in the order of precedence and associativity.
- Unary postfix operator and all binary operators (except assignment operator) have left to right associativity.

**equals() method and identity test(==)**

- equals() method is defined in java.lang.Object class. Its signature is:

  public boolean equals(Object anObject);
- The default behavior of equals() is to do identity test.
- Identity test (==) returns true only when both the operands are essentially the same object.
- equals() method returns false when both objects are not of same class. It will not give any compile-time error or throw an exception.
- == can not be used on the objects of two unrelated classes. Compile-time error is generated.
- **Comparison in String class**
  Identity test is not favorable to find whether two strings are equals. String class therefore overrides equals() method. A call to equals() on string will do equality test and return true if the argument passed is of type String and two strings have same contents.
- **Comparison in Boolean wrapper class**
  Boolean wrapper class also overrides equals method(). Overridden equals() does not do identity test, but an equality test. So equals() on Boolean object will return true if the arguments also wraps the same boolean value as the object.

**Precedence of Java operators**

| HIGHEST | | | |
|---|---|---|---|
| () | [] | . | |
| ++ | -- | ~ | ! |
| * | / | % | |

| | | | |
|---|---|---|---|
| + | - | | |
| >> | >>> | << | |
| > | >= | < | <= |
| == | != | | |
| & | | | |
| ^ | | | |
| \| | | | |
| && | | | |
| \|\| | | | |
| ?: | | | |
| = | op= | | |
| LOWEST | | | |

**Conversion and casting**

- Implicit type conversion
  Java performs an implicit/automatic conversion from the type of the expression to a type acceptable in that context. The context may accept a type that is compatible to the type of the expression; as a convenience, rather than requiring the programmer to indicate a type conversion explicitly.
- Casting
  Programmer may do explicit conversion, which is generally called as casting.

**Conversion Contexts**

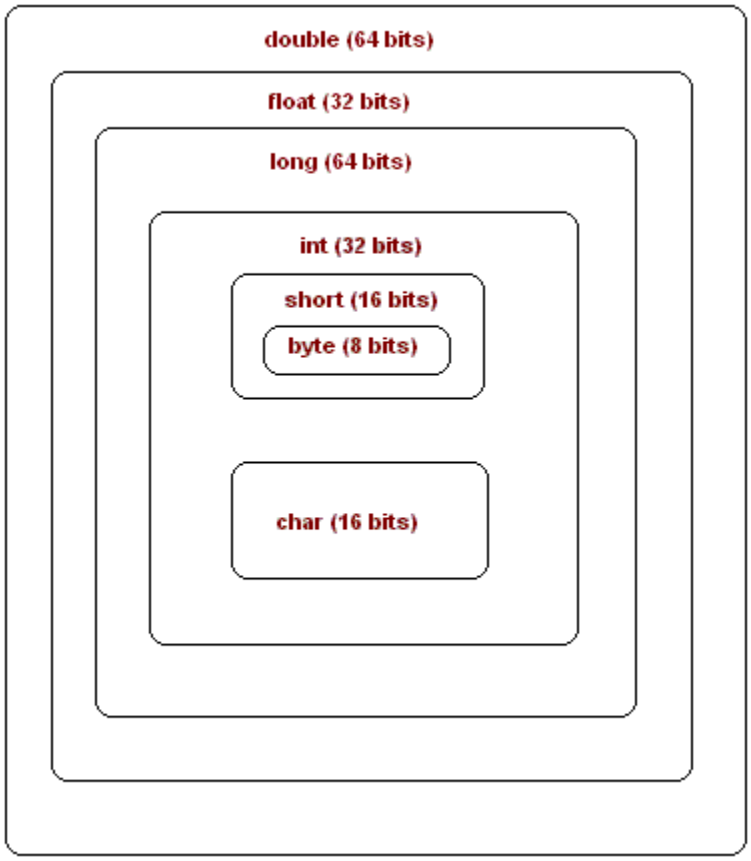
Following conversion contexts define different situations in which implicit conversions occur:

- Assignment conversion
- Method invocation conversion

- Numeric/Arithmetic promotion
- String conversion

**Rules for conversion of primitives in assignment**

a. boolean may not be converted to any other data type.
b. Non-boolean primitive data types can be converted to non-boolean only if it is widening conversion.




- **Widening Conversion**
  Above diagram shows the widening conversions in primitives. Outermost rectangle represents the widest data type of all and innermost represent narrowest of all. For example, rectangle representing **long** contains the rectangle representing **int** indicates that **long** is wider data type than **int**. Thus, int value can be assigned to long variable without any need of explicit casting. For example,

  ```
  int intNumber = 4;
  long longNumber = intNumber; //valid
  ```
- Incompatible data types. Two independent rectangles indicate that the variable cannot be automatically converted. For example,

```
byte byteNumber = 4;
short shortNumber = byteNumber; // valid as short is wider than byte
char charNumber = byteNumber; // Invalid, char and byte are incompatible
```

- Data type is considered wider if it can represent wider range of numbers. The data type is not wider than other data type just because it has more bits. For example, float (represented with 32 bits) is wider than long (represented with 64 bit). This is due to the fact that float can represent wider range of numbers than long.
- casting null will not give any compile-time error or throw an exception, but casting null is of no use.
- Few Exceptions to the rule of conversion
  Integral literals are of type **int** by default. So you may think that

  ```
  byte byteNumber = 2;
  ```

  is invalid as 2 is integral literal and integral literals are by default of type int. Therefore byte byteNumber = 2 will be narrowing conversion (int ==> byte) and hence will throw an exception. However, it is not so.

  ```
  byte byteNumber = 2; //valid
  ```

  Variables of byte type can be assigned any integral literal up to the value -128 to +127 without any need of explicit casting. Similarly,

  ```
  short shortNumber = 2; //valid
  char charNumber = 10; //valid
  ```

**Rules for conversion of primitives during the method call**:

- boolean may not be converted to any other data type.
- Non-boolean primitive data types can be converted to non-boolean only if it is widening conversion.

**Rules for conversion of primitives during the numeric/arithmetic operations**:

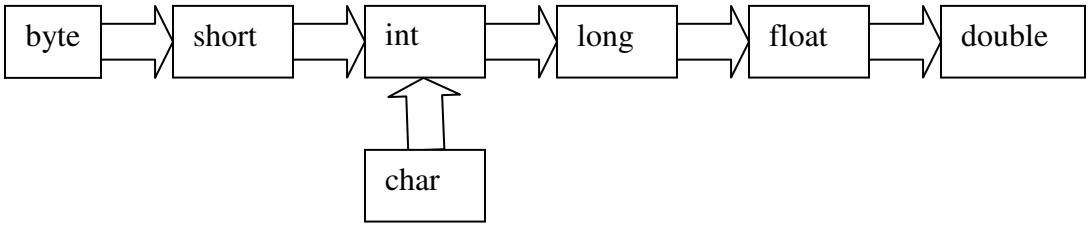
- **Arithmetic Promotions for unary operators**
  - For ++ and—operators, there is no conversion.
  - For unary +, - and ~ , widening conversion happens as variables of type byte, short or char are promoted to int.
- **Arithmetic Promotions for binary operators**
  Both the operands must be of non-boolean primitive type.
- If one of the operand is double, other is promoted to double before operation.
- Else, if one of the operand is float, other is promoted to float before operation

- Else, if one of the operand is long, other is promoted to long before operation.
- Else, both operands are promoted to int before operation.

**Rules for casting of primitive data types**

- Any non-boolean to any non-boolean is allowed.
- boolean to non-boolean casting is NOT allowed.
- Non- boolean to boolean casting is NOT allowed.

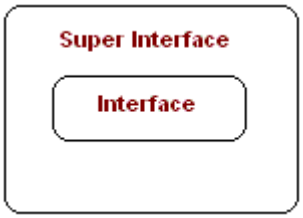
Following is the figure for the allowable primitive conversion:




**Rules for object reference Conversion**

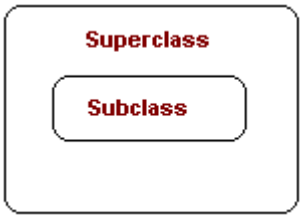
Object reference Conversion rules for assignment and method calls are same. Any object reference can be converted into other object type provided it is a widening conversion.

ChildType childObjectRef = new ChildType;
ParentType parentObjectRef = ChildparentObjectRef; //valid conversion
If ChildType is an interface type, then ParentType must be the super interface of the ChildType.

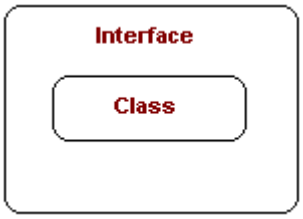



a. If ChildType is a class, and

a. If ParentType is also a class, it must be the super class of ChildType.




b. Alternatively, if ParentType is an interface, ChildType must implement it.




b. If ChildType is an array, ParentType can be of type Object or of type Cloneable, Serializable. Alternatively, ParentType can be an array of compatible type.

**Flow control statements**

- Conditional flow control statements like if, if-else and switch-case,
- Looping statements such as for, while, do-while,
- Exception handling statements like try-catch-finally, throw,
- flow control statements ( with or without label ) like break, continue.

**switch statement:**

The syntax of the switch statement is:

```
switch(expression)
{
    case ConstantExpression: statement(s);
    case ConstantExpression: statement(s);
                .
                .
                .
default: statement(s);
}
```

- The type of the expression must be char, byte, short, or int, or a compile-time error occurs.

- Object reference cannot be passed to switch or case statement. Only constant expressions can be passed. Only expressions that can be evaluated at compile time are allowed to be passed to the switch or case statement. These can be numeric literals or static final variables.
- Every case expression must be evaluated to be unique.
- break statement may be used at the end of a case statement, to discontinue execution. If it is absent, the control falls through to execute all the remaining case statements and default statement (if any).
- There can be at most only one default statement. The order of case statements and default can be anything.
- If none of the case match and there is no default case, switch statement does nothing.
- Variable cannot be inside a switch statement. But the variable can be initialized inside the switch statement.
- break cannot be used without a case in the switch statement.
- All the case statements ( even the default statement ) are optional in the switch statement.
- The keyword continue may occur within the body of a switch statement if it pertains to a loop.

**break and continue statement:**

- A break statement transfers the control out of an enclosing statement. break used within a loop breaks the execution of the current loop. In case of nested loops, the break statement passes the control to the immediate outer loop.
- A continue statement breaks the current iteration and moves to next iteration.
- break and continue with labels:
  - Labels specify the target (statement) for continue and break.
  - continue with label does not jump to the labeled statement but instead jumps to the end of the labeled loop.
  - Same label identifiers can be reused multiple times as long as they are not nested.
  - Label names do not conflict with the same named identifier(variable, method or class name).

## 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