

---

# Java certification success, Part 1: SCJP

Skill Level: Introductory

Pradeep Chopra  
Cofounder  
WHIZlabs Software

06 Nov 2003

This tutorial is designed to prepare programmers for the Sun Certified Java Programmer (SCJP) 1.4 exam, providing a detailed overview of all the exam's main objectives.

## Section 1. Before you start

### About this tutorial

This tutorial is a guide to help you become a Sun certified Java programmer. It is organized in the same way as the Sun Certified Java Programmer (SCJP) 1.4 exam and provides a detailed overview of all of the exam's main objectives. Throughout the tutorial, simple examples are provided to illustrate the important concepts covered in the exam.

At the end of each section, exercises are provided to test your knowledge of the main concepts covered in that section. At the end of the tutorial, useful resources, such as recommended books, articles, tutorials, training, and specifications for the exam, are also listed.

If you are a programmer interested in enhancing your skills and your resume, this tutorial is for you. The tutorial assumes you have familiarity with the Java programming language.

### About the SCJP 1.4 exam

The SCJP 1.4 exam is the first in a series of Java certification exams offered by Sun Microsystems, and for many programmers it is the first step to becoming established

as a competent Java developer.

The exam tests the knowledge of Java fundamentals and requires in-depth knowledge of the syntax and semantics of the language. Even experienced Java programmers can benefit from preparing for the SCJP exam. You get to learn very subtle and useful tips you might not have been aware of, even after many years of Java programming.

---

## Section 2. Declarations and access control

### Arrays

Arrays are dynamically created objects in Java code. An array can hold a number of variables of the same type. The variables can be primitives or object references; an array can even contain other arrays.

#### Declaring array variables

When we declare an array variable, the code creates a variable that can hold the reference to an array object. It does not create the array object or allocate space for array elements. It is illegal to specify the size of an array during declaration. The square brackets may appear as part of the type at the beginning of the declaration or as part of the array identifier:

```
int[] i;           // array of int
byte b[];          // array of byte
Object[] o,        // array of Object
short s[][];       // array of arrays of short
```

#### Constructing arrays

You can use the `new` operator to construct an array. The size of the array and type of elements it will hold have to be included. In the case of multidimensional arrays, you may specify the size only for the first dimension:

```
int [] marks = new int[100];
String[][] s = new String[3][];
```

#### Initializing arrays

An array initializer is written as a comma-separated list of expressions, enclosed within curly braces:

```
String s[] = { new String("apple"), new String("mango") };
int i[][] = { {1, 2}, {3,4} };
```

An array can also be initialized using a loop:

```
int i[] = new int[5];
for(int j = 0; j < i.length; j++)
{
    i[j] = j;
}
```

### Accessing array elements

Arrays are indexed beginning with 0 and ending with  $n-1$ , where  $n$  is the array size. To get the array size, use the array instance variable called `length`. If you attempt to access an index value outside the range 0 to  $n-1$ , an `ArrayIndexOutOfBoundsException` is thrown.

## Declaring classes, variables, and methods

Now let's look at ways we can modify classes, methods, and variables. There are two kinds of modifiers -- access modifiers and non-access modifiers. The access modifiers allow us to restrict access or provide more access to our code.

### Class modifiers

The access modifiers available are `public`, `private`, and `protected`. However, a top-level class can have only `public` and default access levels. If no access modifier is specified, the class will have default access. Only classes within the same package can see a class with default access. When a class is declared as `public`, all the classes from other packages can access it.

Let's see the effect of some non-access modifiers on classes. The `final` keyword (see [Java keywords and identifiers](#) for more on keywords) does not allow the class to be extended. An `abstract` class cannot be instantiated, but can be extended by subclasses:

```
public final class Apple {...}
class GreenApple extends Apple {}    // Not allowed, compile time error
```

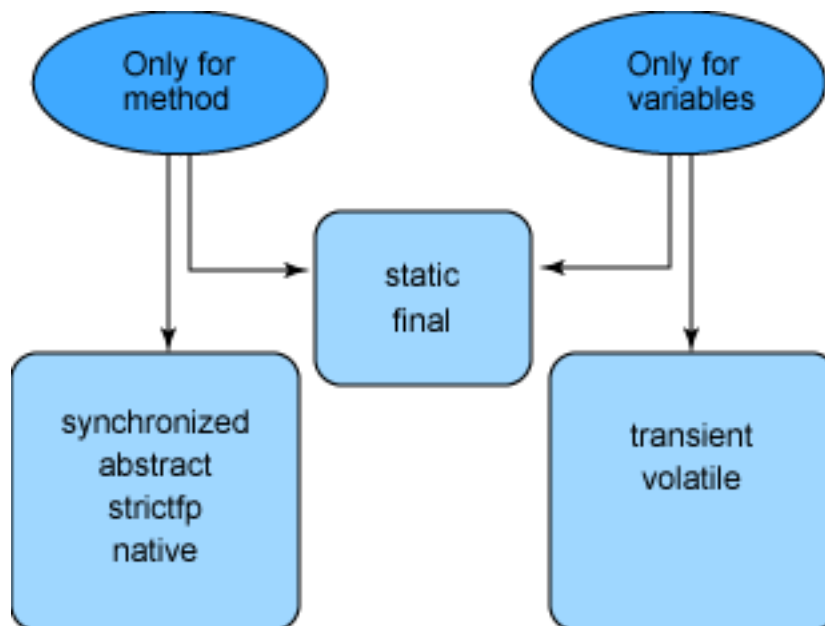
### Method and variable modifiers

All the access modifiers can be used for members of a class. The `private` members can only be accessed from inside the class. The `protected` members can only be accessed by classes in the same package or subclasses of the class. The `public` members can be accessed by any other class.

If there is no access modifier specified, these members will have default access and only other classes in the same package can access them.

Now let's explore other modifiers that can be applied to member declarations. Some of them can be applied only to methods while some can be applied only to variables, as illustrated in the figure below:

### Figure 1. Modifiers for methods and variables



A `synchronized` method can be accessed by only one thread at a time. `Transient` variables cannot be serialized. An `abstract` method does not have an implementation; it has to be implemented by the first concrete subclass of the containing class. The class containing at least one `abstract` method has to be declared as `abstract`. However, an `abstract` class need not have any `abstract` methods in it:

```
public abstract class MyAbstractClass
{
    public abstract void test();
}
```

The `native` modifier indicates that the method is not written in the Java language, but in a native language. The `strictfp` keyword (see [Java keywords and identifiers](#) for more information on keywords), which is used only for methods and classes, forces floating points to adhere to IEEE754 standard. A variable may be declared `volatile`, in which case a thread must reconcile its working copy of the field with the master copy every time it accesses the variable.

`Static` variables are shared by all instances of the class. `Static` methods and variables can be used without having any instances of that class at all:

```
class StaticTest
{
    static int i = 0;
    static void getVar()
    {
        i++;
        System.out.println(i);
    }
}

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

```
    {  
        StaticTest.getVar();  
    }  
}
```

## Constructors

A constructor is used when creating an object from a class. The constructor name must match the name of the class and must not have a return type. They can be overloaded, but they are not inherited by subclasses.

### Invoking constructors

A constructor can be invoked only from other constructors. To invoke a constructor in the same class, invoke the `this()` function with matching arguments. To invoke a constructor in the superclass, invoke the `super()` function with matching arguments. When a subclass object is created, all the superclass constructors are invoked in the order starting from the top of the hierarchy.

### Default constructors

The compiler creates the default constructor if you have not provided any other constructor in the class. It does not take any arguments. The default constructor calls the no-argument constructor of the superclass. It has the same access modifier as the class.

However, the compiler will not provide the default constructor if you have written at least one constructor in the class. For example, the following class has a constructor with two arguments defined in it. Here the compiler will give an error if we try to instantiate the class without passing any arguments because the default constructor is not available:

```
class Dot  
{  
    int x, y;  
    Dot(int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
}
```

If you invoke the default constructor of your class and the superclass does not have a constructor without any arguments, your code will not compile. The reason is that the subclass default constructor makes an implicit call to the no-argument constructor of its superclass. For instance:

```
class Dot  
{  
    int x, y;  
    Dot(int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
}  
  
class MyDot extends Dot { }  
class Test  
{  
    public static void main(String args[])  
    {  
        MyDot dot=new MyDot();  
    }  
}
```

## Summary

In this section, we covered the important concepts that are included in the first objective. We discussed the valid ways to declare and construct single and multi-dimensional arrays. When you understand the effect of modifiers in methods and classes, make sure you know the legal combinations of modifiers. For example, you cannot declare a `final` method as `abstract`. We also learned about constructors. Remember that the compiler provides the default no-argument constructor only when you don't write any constructors.

## Exercise

### Question:

What will be the result of an attempt to compile and run the following program?

```
class Box  
{  
    int b,w;  
    void Box(int b,int w)  
    {  
        this.b = b;  
        this.w = w;  
    }  
}  
  
public class MyBox extends Box  
{  
    MyBox()  
    {  
        super(10,15);  
        System.out.println(b + "," + w);  
    }  
    static public void main(String args[])  
    {  
        MyBox box = new MyBox();  
    }  
}
```

### Choices:

- A. Does not compile; main method is not declared correctly
- B. Prints 10,15
- C. Prints 0,0

- D. None of the above

**Correct choice:**

- D

**Explanation:**

The program does not compile because there is no matching constructor in the base class for the `super(10,15)` call from the subclass constructor. `void Box(int b, int w)` is not a constructor, because a return type is given. If it had been a constructor, the variables `w` and `h` would have been initialized to 10 and 15. The program would have compiled correctly and printed 10,15. There is no error in the declaration of the `main()` method; `static` and `public` can appear in any order.

---

## Section 3. Flow control, assertions, and exception handling

### Flow control statements

The flow control statements allow you to conditionally execute statements, to repeatedly execute a block of statements, or to just change the sequential flow of control.

#### **if/else statement**

The `if/else` statement is used for decision-making -- that is, it decides which course of action needs to be taken. The syntax is:

```
if(boolean expression)
{
    // do this if the expression is true
}
else
{
    // do this if the expression is false
}
```

The `else` part in the `if/else` statement is optional. The curly braces are optional if the body is limited to a single statement. (Note that we cannot use numeric values to represent true and false as we do in C/C++.) For instance:

```
if (x == 5) {}           // compiles, executes body if x is equal
to 5
if (x = 0) {}           // does not compile, because expression
is non-boolean
if (x = true) {}        // compiles, but the body is always
```

```
executed
```

In the case of nested `if/else` statements, each `else` clause belongs to the closest preceding `if` statement that does not have an `else`.

### **switch statement**

The `switch` statement is also used for decision-making, based on an integer expression. The argument passed to the `switch` and `case` statements should be `int`, `short`, `char`, or `byte`. The argument passed to the `case` statement should be a literal or a `final` variable. If no case matches, the default statement (which is optional) is executed.

When a `break` statement is encountered, the control moves out of the `switch` statement. If no `break` statement is given, all the `case` statements are executed until a `break` is encountered or the `switch` statement ends. For instance:

```
int x = 1;
switch(x)
{
    case 1: System.out.println("one");
    case 2:      System.out.println("two");
}
// both one and two are printed
```

Notice the position of the `default` statement. Though the `default` statement is usually placed at the end of all the `case` options, it is not mandatory as you can see in the example below:

```
int i = 1;
switch(i)
{
    default:
        System.out.println("Default");
        break;
    case 0:
        System.out.println("Zero");
        break;
    case 1:
        System.out.println("One");
        break;
}
```

Note that the control comes to the `default` statement only if none of the cases match. However, it is a good practice to have the `default` statement at the end itself.

### **Loop statements**

The three types of Java looping constructs are `while`, `do-while`, and `for`.

#### ***while loop***

The syntax of the `while` loop is:

```
while(boolean expression) {}
```



The body of the `while` loop is executed only if the expression is true, so it may not be executed even once:

```
while(false){}    // body never executed
while(1) {}       // code will not compile, not a boolean
```

### ***do-while loop***

The syntax of the `do-while` loop is:

```
do { } while(boolean expression);
```

The body of the `do-while` loop is executed at least once because the test expression is evaluated only after executing the loop body. Also, don't forget the ending semicolon after the `while` expression.

### ***for loop***

The syntax of the `for` loop is:

```
for(expr1; expr2; expr3)
{
    // body
}
```

`expr1` is for initialization, `expr2` is the conditional test, and `expr3` is the iteration expression. Any of these three sections can be omitted and the syntax will still be legal:

```
for( ; ; ) {} // an endless loop
```

There can be more than one initialization expression and more than one iteration expression, but only one test expression.

### ***break and continue***

The `break` statement is used to exit from a `loop` or `switch` statement, while the `continue` statement is used to skip just the current iteration and continue with the next.

In the case of nested loops, the `break` statement exits from the innermost loop only. If a `break` keyword (see [Java keywords and identifiers](#) for more information on keywords) is combined with a label, a `break` statement will exit out of the labeled loop:

```
outer: for (int i = 0; i < 10; i++)
{
    while(y > 0)
    {
        break outer;
    }
}
```

```
}  
// breaks from the for loop if y > 0
```

## Assertions

An assertion is a statement containing a boolean expression that is assumed to be true when the statement is executed. The system reports an `AssertionError` if the expression evaluates to false. It is used for debugging purposes:

```
assert(a > 0); // throws an AssertionError if a <= 0
```

Assertions can be in two forms:

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

`Expression1` should always result in a boolean value.

`Expression2` can be anything that results in a value. The value is used to generate a `String` message that displays more debugging information. For instance:

```
class AssertionTest  
{  
    int a ;  
    public void func()  
    {  
        assert (a < 0): "a is positive" ;  
        // Remaining code  
    }  
}
```

In the above code, the assertion expression is true if `a` is negative. If `a` is positive, an `AssertionError` is thrown with the message "a is positive."

Assertions are disabled by default. To compile with assertions enabled, you need to use the `source 1.4` flag:

```
javac -source 1.4 Test.java
```

To enable assertions at runtime, use the `-enableassertions` or `-ea` flags.

For selective disabling of assertions at runtime, use the `-da` or `-disableassertions` flags.

For enabling assertions in the system classes, use the `-esa` or `-dsa` flags. Assertions can also be enabled or disabled on a package basis.

### Appropriate and inappropriate use of assertions

You can place an assertion at any location that you don't expect to be reached normally. Assertions can be used to validate the parameters passed to a private method. However, assertions should not be used to validate parameters passed to public methods because a public method must check its arguments regardless of whether assertions are enabled or not. However, you can test postconditions with assertions in both public and non-public methods. Also, assertions should not change the state of a program in any manner.

See [Resources](#) for more information on assertions.

## Exception handling

Exceptions are Java objects; exception classes are derived from `java.lang.Throwable`.

An exception is thrown to denote an abnormal occurrence like accessing an array index that exceeds the size of the array. Exceptions can be checked or unchecked. Subclasses of `RuntimeException` and `Error` are called *unchecked exceptions*. All other classes, which inherit from `Exception`, are *checked exceptions*. Any method that might throw a checked exception must either declare the exception using the `throws` keyword (see [Java keywords and identifiers](#) for more information on keywords) or handle the exception using a `try/catch` block.

### **try / catch block**

The code that might throw an exception is enclosed in the `try` block. One or more `catch` clauses can be provided to handle different exception types:

```
try
{
    // code that might throw exceptions
}
catch(Exception e)
{
    //code to handle the exception
}
```

The more specific exception type should be caught first. For instance, we have to order the `catch` blocks from specific to most general, otherwise a compile-time error occurs.

For example, assume that `MyException` inherits from `Exception`:

```
try
{
    throw new MyException();
}
catch(Exception e)
{
    System.out.println("Exception");
}
catch(MyException e)
{
    System.out.println("MyException");
}
```

```
}
```

Here the compiler complains that the `catch` clause for `MyException` is unreachable because all exceptions thrown will be caught by the first general `catch` clause.

### **finally block**

The `finally` block can also be provided to perform any cleanup operation. If an exception is thrown, any matching `catch` clauses are executed, then control comes to the `finally` block, if one is provided. The syntax of the `finally` block is as follows:

```
try
{
    // code that throws exceptions
}
catch(Exception e)
{
    // handle exception
}
finally
{
    // cleanup code
}
```

The `finally` block is executed even if no exception is thrown. The only case in which this does not happen is when `System.exit()` is invoked by the `try` or `catch` blocks. A `try` block should have either a `catch` block or a `finally` block, but it's not required to have both.

### **Declaring and throwing exceptions**

To throw an exception explicitly from the code, use the `throw` keyword.

The checked exceptions that a method can throw must be declared using the `throws` keyword (see [Java keywords and identifiers](#) for more information on keywords). For instance:

```
void f() throws MyException
{
    if(x > y) throw new MyException(); // MyException is a subclass of Exception
}
```

## **Summary**

In this section, we saw the proper usage of flow control statements like `if/else` and `while` loops. Make sure that the arguments passed to these statements are legal. Note the difference between `break` and `continue` statements. We discussed assertions and the ways to disable and enable them. Be clear about when assertions should be used and when using them is inappropriate. We also learned about checked/unchecked exceptions and how to handle or raise exceptions in the code. Remember that the `finally` block would be reached in most occasions

except in some special cases.

## Exercise

### Question:

What will be printed out if you attempt to compile and run the following code?

```
int i = 3;
switch (i)
{
    default:
        System.out.println("default");
    case 0:
        System.out.println("zero");
    case 1:
        System.out.println("one");
        break;
    case 2:
        System.out.println("two");
}
```

### Choices:

- A. default
- B. default, zero, one
- C. Compiler error
- D. No output displayed

### Correct choice:

- B

### Explanation:

Although it is normally placed last, the `default` statement does not have to be placed as the last option in the case block. Since there is no case label found matching the expression, the default label is executed and the code continues to fall through until it encounters a `break` statement.

---

## Section 4. Garbage collection

### Behavior of the garbage collector

A Java programmer does not have to worry about memory management, because it is automatically taken care of by the garbage collector. The Java virtual machine (JVM) decides when to run the garbage collector. The garbage collector is a low priority thread that runs periodically, releasing memory used by objects that are not needed anymore.

The garbage collection (GC) algorithm varies from one JVM to another. There are different algorithms being used, like reference counting or the mark and sweep algorithm. See [Resources](#) for more information on garbage collection.

### Running the garbage collector

The JVM usually runs the garbage collector when the level of available memory is low. However, the garbage collector cannot ensure that there will always be enough memory.

If there is not enough memory even after the garbage collector reclaims memory, the JVM throws an `OutOfMemoryError` exception. Note that the JVM will definitely run the garbage collector at least once before throwing this exception.

While you can *request* that the garbage collector run, it is important to note that you can never *force* this action.

### Requesting that the garbage collector run

To request garbage collection, you can call either of the following methods:

```
System.gc()  
Runtime.getRuntime().gc()
```

## Eligibility for garbage collection

An object is eligible for garbage collection when no live thread can access it.

An object can become eligible for garbage collection in different ways:

- If the reference variable that refers to the object is set to null, the object becomes eligible for garbage collection, provided that no other reference is referring to it.
- If the reference variable that refers to the object is made to refer to some other object, the object becomes eligible for garbage collection, provided that no other reference is referring to it.
- Objects created locally in a method are eligible for garbage collection when the method returns, unless they are exported out of the method (that is, returned or thrown as an exception).
- Objects that refer to each other can still be eligible for garbage collection if no live thread can access either of them.

Consider the following example:

```
public class TestGC
{
    public static void main(String [] args)
    {
        Object o1 = new Integer(3);           // Line 1
        Object o2 = new String("Tutorial");    // Line 2
        o1 = o2;                               // Line 3
        o2 = null;                             // Line 4
        // Rest of the code here
    }
}
```

In this example, the `Integer` object initially referred to by the reference `o1` becomes eligible for garbage collection after line 3. This is because `o1` now refers to the `String` object. Even though `o2` is now made to refer to null, the `String` object is not eligible for garbage collection because `o1` refers to it.

## Finalization

Java technology allows you to use the `finalize()` method to do the necessary cleanup before the garbage collector removes the object from memory. This method is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. It is defined in the `Object` class, thus it is inherited by all classes. A subclass overrides the `finalize()` method to dispose of system resources or to perform other cleanup:

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

If an uncaught exception is thrown by the `finalize()` method, the exception is ignored and finalization of that object terminates.

The `finalize()` method will be invoked only once in the lifetime of an object.

You can use an object's `finalize()` method to make it ineligible for garbage collection. Note, however, that the garbage collector will not run `finalize()` for this object again.

The `finalize()` method will always be invoked once before the object is garbage collected. However, the `finalize()` method might never be invoked for an object in its lifetime, because there is no guarantee that it will get garbage collected.

## Summary

In this section, we discussed the concept of garbage collection, which is the Java language's memory management technique. We saw that garbage collection cannot be forced. We also learned the different ways in which objects become eligible for garbage collection and that the `finalize()` method is invoked on an object before

it is removed by the garbage collector.

## Exercise

### Question:

How many objects will be eligible for garbage collection after line 7?

```
public class TutorialGC
{
    public static void main(String [] args)
    {
        Object a = new Integer(100); // Line1
        Object b = new Long(100);    // Line2
        Object c = new String("100"); // Line3
        a = null;                    // Line4
        a = c;                       // Line5
        c = b;                       // Line6
        b = a;                       // Line7
        // Rest of the code here
    }
}
```

### Choices:

- A. 0
- B. 1
- C. 2
- D. 3
- E. Code does not compile

### Correct choice:

- B

### Explanation:

Among the three objects created in lines 1, 2, and 3, only the `Integer` object is eligible for garbage collection at the end of line 7. The reference variable `a`, which originally referred to the `Integer` object is made to refer to the `String` object in line 5. So the `Integer` object is eligible for garbage collection after line 5, since there are no variables referring to it. The variables `b` and `c` refer to the `String` and `Long` objects in lines 6 and 7, so they are not eligible for garbage collection.

---

## Section 5. Language fundamentals



## Package and class declarations

A *package* represents a group of classes. A `package` statement should be the first valid statement in the source file. If there is no `package` statement, the classes in the source file belong to the default unnamed package, or else they belong to the named package. Only one `package` statement is allowed in a source file.

In a source file, there can be only one public class, and the name of the file should match that of the class. An `import` statement allows you to use a class directly instead of referring to it using its fully qualified name. `Import` statements must come after any `package` statement and before the class declarations, as shown below:

```
package com.abc;           // package statement
import java.net.*;         // wild card import (imports all
classes in the package)
import java.util.Vector;   // class import (only the given
class is imported)
```

### Nested class declarations

A *nested* class or *inner* class is a class defined inside another class. Nested classes can be non-static, method-local, anonymous, or static.

### Non-static inner classes

A non-static inner class definition does not use the `static` keyword (see [Java keywords and identifiers](#) for more information on keywords). It is defined within the scope of the enclosing class, as follows:

```
class MyClass
{
    class MyInner{}
}
```

To instantiate a non-static inner class, you need an instance of the outer class.

A non-static inner class has access to all the members of the outer class. From inside the outer instance code, use the inner class name alone to instantiate it:

```
MyInner myInner = new MyInner();
```

From outside the outer instance code, the inner class name must be included in the outer instance:

```
MyClass myClass = new MyClass();
MyClass.MyInner inner = myClass.new MyInner();
```

Modifiers that can be applied to non-static inner classes are `final`, `abstract`,

`public`, `private`, and `protected`.

### Method local inner classes

A method local inner class can be instantiated in the method where it is defined. It can also be in a constructor, a local block, a static initializer, or an instance initializer. It cannot access the local variables of the enclosing method unless they are `final`. The only modifiers for such a class are `abstract` or `final`.

### Anonymous inner classes

Anonymous inner classes are inner classes that have no name. They can extend a class or implement an interface. For instance:

```
button1.addMouseListener(new MouseAdapter()  
{  
    public void mouseClicked(MouseEvent e)  
    {  
        button1_mouseClicked(e);  
    }  
});
```

This example shows an anonymous inner class. It is important to note that the class has been defined within a method argument.

### Static nested classes

A static nested class is a static member of the enclosing class. It does not have access to the instance variables and methods of the class. For instance:

```
class MyOuter  
{  
    static class MyNested{  
    }  
}  
class Test  
{  
    public static void main(String args[]){  
        MyOuter.MyNested n = new MyOuter.MyNested();  
    }  
}
```

As this example shows, a static nested class does not need an instance of the outer class for instantiation.

## Interfaces

An interface is like a public class that has only abstract and public methods. The variables declared in an interface are implicitly `public`, `static`, and `final`. For instance:

```
interface MyInterface  
{  
    void f();  
}  
class MyClass implements MyInterface  
{
```

```
    public void f() {}
}
```

A concrete class implementing an interface has to implement all the methods declared in it.

Interface methods are implicitly `public` and cannot be declared `static`. All the overriding rules are applicable for the implemented methods. A class can implement more than one interface. An interface can extend any number of interfaces:

```
main() method
```

To run the class as a Java application, there must be a method called `main` that takes an array of `String` objects as arguments. The `main()` method must be `public` and `static` and should not return anything. Note that the modifiers `static` and `public` can be in any order:

```
static public void main(String[] abc) {} // Valid
```

### Command-line arguments

The `String` array argument of the `main()` method contains the command line arguments. The length of the array will be equal to the number of command line arguments. If you try to access elements outside the range 0 to length-1, an `ArrayIndexOutOfBoundsException` is thrown. For instance:

```
class Test
{
    public static void main(String args[])
    {
        System.out.println(args[0]);
    }
}
```

When you use the `java Test Hello` command to invoke this code, the output is "Hello."

## Java keywords and identifiers

Keywords are reserved words that are predefined in the language; see the table below. All the keywords are in lowercase.

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static

strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while

Note: `assert` is a new keyword since J2SE 1.4

An identifier is composed of a sequence of characters where each character can be a letter, a digit, an underscore, or a dollar sign.

An identifier declared by the programmer cannot be a Java keyword and it cannot start with a digit. For instance:

```
number, $$abc, _xyx    // Legal Identifiers
12cc, ss-he           // Illegal identifiers
```

## Ranges of primitive types

The size and range of all primitive types are listed in the table below. The positive range is one less than the negative range because zero is stored as a positive number. For the boolean datatype, there is no range; it can only be true or false.

Primitive Type	Size	Range of Values
byte	8 bits	$-2^7$ to $2^7 - 1$
short	16 bits	$-2^{15}$ to $2^{15} - 1$
int	32 bits	$-2^{31}$ to $2^{31} - 1$
long	64 bits	$-2^{63}$ to $2^{63} - 1$
char	16 bits	0 to $2^{16} - 1$
float	32 bits	
double	64 bits	

## Literals

Integer literals can be decimal, octal, or hexadecimal.

Octal literals begin with zero and only digits 0 through 7 are allowed. For instance:

```
011
```

Hexadecimal literals begin with `0x` or `0X`, the digits allowed are 0 through 9 and `a` through `f` (or `A` through `F`). For instance:

```
0x0001;
```

All integer literals are of type `int` by default. To define them as long, we can place a suffix of `L` or `l` after the number.

For char literals, a single character is enclosed in single quotes. You can also use the prefix `\u` followed by four hexadecimal digits representing the 16-bit Unicode character:

```
'\u0045', 'A', 'a'
```

For float literals, we need to attach the suffix *F* or *f*. For instance:

```
float f = 23.6F;  
double d = 23.6;
```

In the case of double literals, we can add *D* or *d* to the end, however this is optional.

### Uninitialized variables

Static and instance variables of a class are initialized with default values for the particular datatype. Local variables are never given a default value; they have to be explicitly initialized before use. When an array is created, all its elements are initialized to their default values, even if the array is declared locally.

## Summary

In this section, we saw the correct way to declare packages, classes, interfaces, and the `main()` method. We have also seen the effect of using variables without initializing them. Make sure you can identify the keywords in Java code. Be clear about the ranges of primitive types and how to use literals.

## Exercise

### Question:

What will be the result of compiling and running the following program with the command **java Test hello**?

```
public class Test  
{  
    static int i;  
    static public void main(String[] args)  
    {  
        do  
        {  
            System.out.println(args[++i]);  
        } while (i < args.length);  
    }  
}
```

### Choices:

- A. Prints "hello"

- B. Prints "Test"
- C. Code does not compile
- D. Throws an `ArrayIndexOutOfBoundsException`

**Correct choice:**

- D

**Explanation:**

The variable `i` is automatically initialized to 0. Inside the loop, `i` is incremented by 1 first before printing the value of `args[i]`. Here the `args` array has only one element -- "hello." Attempting to access the second element causes the exception.

---

## Section 6. Operators and assignments

### Using operators

#### Assignment operators

You can assign a primitive variable using a literal or the result of an expression. The result of an expression involving integers ( `int`, `short`, and `byte` ) is always at least of type `int`. Narrowing conversions are not allowed, as they would result in the loss of precision. For instance:

```
byte b = 5;
byte c = 4;
byte d = b + c;           // does not compile because int
                           // cannot fit in a byte

float f = (float)35.67;    // implicitly double, so casting to
float                                     float
float f = 35.67F;
```

A boolean cannot be assigned any type other than boolean.

If we assign an existing object reference to a reference variable, both reference variables refer to the same object. It is legal to assign a child object to a parent object, but the reverse is not allowed. You can also assign an object reference to an interface reference variable, if the object implements that interface. For instance:

```
class Base{}
public class Child extends Base
{
    public static void main(String argv[])
    {
```

```

    Child child = new Child();
    Base base = new Base();
    base = child;    // Compiles fine
    child = base;    // Will not compile
}
}

```

### instanceof operator

The `instanceof` operator is used to check if an object reference belongs to a certain type or not. The type can either be an interface or a superclass. For instance:

```

class A{}
class B extends A
{
    public static void main(String args[])
    {
        B b = new B();
        if(b instanceof A)
            System.out.println("b is of type A"); // "b is of type A" is printed
    }
}

```

Note that an object is of a particular interface type if any of its superclasses implement that interface.

### The plus (+) operator

The plus (+) operator is typically used to add two numbers together. If the operands are `String` objects, the plus (+) operator is overridden to offer concatenation. For instance:

```

String s1 = "hello ";
String s2 = "World ";
System.out.println(s1 + s2); // prints "hello world"

```

If one of the operands is a `String` and the other is not, the Java code tries to convert the other operand to a `String` representation. If the operand is an object reference, then the conversion is performed by invoking its `toString()` method:

```

int i = 10;
String s1 = " rivers";
String s2 = i + s1;
System.out.println(s2); // prints "10 rivers"

```

### Increment and decrement operators

Two shortcut arithmetic operators are `++`, which increments its operand by 1, and `--`, which decrements its operand by 1.

- `++` increment operator (increments value by 1)
- `--` decrement operator (decrements value by 1)

These operators can appear before the operand (prefix) or after the operand (postfix). In the case of the prefix style, the value of the operand is evaluated after

the increment/decrement operation. For postfix, the value of the operand is evaluated before the increment/decrement operation, as shown below:

Prefix (operator is placed before the operand):

```
int i = 10;
int j = ++i;           // j and i are 11
System.out.println(--i); // prints 10
```

Postfix (operator is placed after the operand):

```
int i = 10;
int j = i++;           // j is 10, i becomes 11
System.out.println(i--); // prints 11
```

## Shift operators

Shift operators shift the bits of a number to the right or left, thus producing a new number:

- `>>` right shift (fills the left bit with whatever value the original sign bit held)
- `<<` left shift (fills the right bit with zeros)
- `>>>` unsigned right shift (fills the left bits with zeros)

For instance:

```
8 >> 1;
Before shifting: 0000 0000 0000 0000 0000 0000 0000 1000
After shifting:  0000 0000 0000 0000 0000 0000 0000 0100
```

Note that decimal equivalent of the shifted bit pattern is 4.

## Bitwise operators

Java provides four operators to perform bitwise functions on their operands:

- The `&` operator sets a bit to 1 if both operand's bits are 1.
- The `^` operator sets a bit to 1 if only one operand's bit is 1.
- The `|` operator sets a bit to 1 if at least one operand's bit is 1.
- The `~` operator reverses the value of every bit in the operand.

For instance:

```
class Test
{
    public static void main(String args[])
    {
        int x = 10 & 9; // 1010 and 1001
        System.out.println(x);
    }
}
```



```
    } // prints 8, which is the decimal equivalent of 1000
}
```

The bitwise operators `&` and `|` can be used in boolean expressions also.

### Logical operators

The four logical operators are `&`, `&&`, `|`, and `||`.

If both operands are true, then the `&` and `&&` operators return `true`.

If at least one operand is true, then the `|` and `||` operators return `true`.

The `&` and `|` operators always evaluate both operands. `&&` and `||` are called *short circuit* operators because if the result of the boolean expression can be determined from the left operand, the right operand is not evaluated. For instance:

```
if ((++x > 2) || (++y > 2)) { x++; }
// y is incremented only if (++x > 2) is false
```

Note that `||` and `&&` can be used only in logical expressions.

## The equals() method

We can use the `==` operator to compare the values of primitive variables and determine if they are equal. However, if object reference variables are compared using the `==` operator, it returns `true` only if the reference variables are referring to the same object. To check for the equality of two objects, the `Object` class provides the `equals(Object obj)` method, which can be overridden to return `true` for logically equal objects. The default implementation of the `equals()` method in the `Object` class returns `true` only if an object is compared to itself -- that is, it behaves similarly to the `==` operator. Classes like `String` and `Boolean` have overridden the `equals()` method, while `StringBuffer` has not. For instance:

```
String s1 = "abc";
String s2 = new String("abc");
if (s1 == s2) // returns false
    System.out.println("Same Object Reference");
if (s1.equals(s2)) // returns true
    System.out.println("Equal Objects"); // prints "Equal Objects"
```

## Passing variables into methods

If the variable passed is a primitive, only a copy of the variable is actually passed to the method. So modifying the variable within the method has no effect on the actual variable.

When you pass an object variable into a method, a copy of the reference variable is actually passed. In this case, both variables refer to the same object. If the object is

modified inside the method, the change is visible in the original variable also. Though the called method can change the object referred by the variable passed, it cannot change the actual variable in the calling method. In other words, you cannot reassign the original reference variable to some other value. For instance:

```
class MyClass
{
    int a = 3;
    void setNum(int a)
    {
        this.a = a;
    }
    int getNum()
    {
        return a;
    }
}
class Test
{
    static void change(MyClass x)
    {
        x.setNum(9);
    }
    public static void main(String args[])
    {
        MyClass my = new MyClass();
        change(my);
        System.out.println(my.getNum()); // Prints 9
    }
}
```

In the above example, you can see that changing the object referred to by `x` changes the same object referred to by `my`.

## Summary

In this section, we discussed the various operators and assignment rules. You should understand the difference between bitwise operators ( `&` and `|` ) and logical operators ( `&&` and `||` ). Be clear about how equality is tested with the `==` operator and the significance of the `equals()` method for the objects. Also, remember that objects are passed by a copy of the reference value, while primitives are passed by a copy of the primitive's value.

## Exercise

### Question:

What will be the result of compiling and running the following code fragment?

```
Integer i= new Integer("10");
if (i.toString() == i.toString())
    System.out.println("Equal");
else
    System.out.println("Not Equal");
```

**Choices:**

- A. Compiler error
- B. Prints "Equal"
- C. Prints "Not Equal"
- D. None of the above

**Correct choice:**

- C

**Explanation:**

The `toString()` method returns the `String` equivalent of this `String` object. It creates a new object each time it is called. The `==` operator compares the bit patterns of the two object references and not the actual `String` contents. Thus the comparison returns `false`, the `else` statement is executed, and "Not Equal" is printed out.

---

## Section 7. Overriding, overloading, and object orientation

### Encapsulation

*Encapsulation* is the concept of hiding the implementation details of a class and allowing access to the class through a public interface. For this, we need to declare the instance variables of the class as private or protected. The client code should access only the public methods rather than accessing the data directly. Also, the methods should follow the Java Bean's naming convention of `set` and `get`.

Encapsulation makes it easy to maintain and modify code. The client code is not affected when the internal implementation of the code changes as long as the public method signatures are unchanged. For instance:

```
public class Employee
{
    private float salary;
    public float getSalary()
    {
        return salary;
    }
    public void setSalary(float salary)
    {
        this.salary = salary;
    }
}
```

## IS-A relationship

The IS-A relationship is based on inheritance. In Java programming, it is implemented using the keyword `extends`. For instance:

```
public class Animal {...}
public class Cat extends Animal
{
    // Code specific to a Cat. Animal's common code is
    inherited
}
```

Here `Cat extends Animal` means that `Cat` inherits from `Animal`, or `Cat` is a type of `Animal`. So `Cat` is said to have an IS-A relationship with `Animal`.

## HAS-A relationship

If an instance of class `A` has a reference to an instance of class `B`, we say that class `A` HAS-A `B`. For instance:

```
class Car
{
    private Engine e;
}
class Engine {}
```

Here the `Car` class can make use of the functionality of the `Engine` class, without having the `Engine` class's specific methods in it. This gives us a chance to reuse the `Engine` class in multiple applications. Thus the HAS-A relationship allows us to have specialized classes for specific functions.

## Polymorphism

*Polymorphism* means "any forms." In object-oriented programming, it refers to the capability of objects to react differently to the same method. Polymorphism can be implemented in the Java language in the form of multiple methods having the same name. Java code uses a late-binding technique to support polymorphism; the method to be invoked is decided at runtime.

*Overloaded* methods are methods that have the same name, but different argument lists. *Overriding*, on the other hand, occurs when a subclass method has the same name, same return type, *and* same argument list as the superclass method.

### Overloading

As we mentioned, it is mandatory for overloaded methods to have the same names but different argument lists. The arguments may differ in type or number, or both. However, the return types of overloaded methods can be the same or different. They may have different access modifiers and may throw different checked exceptions.

Consider a `print()` method used to output data. The data might be `int`, `boolean`, or `char`. For each type of data, the `print()` method should be implemented differently. We can use overloaded methods as shown here:

```
void print(int i) {}  
void print(boolean b) {}  
void print(char c) {}
```

The method invoked depends on the type of argument passed. A subclass can overload the methods, which it inherits from its superclass. Constructors can be overloaded, which allows us to instantiate a class with different types of arguments. In the following example, the subclass constructor invokes the overloaded constructor of the superclass, which takes an integer argument:

```
class Base  
{  
    Base() {}  
    Base(int a)  
    {  
        System.out.println(a);  
    } //Overloaded constructors  
}  
  
class Derived  
{  
    Derived(int a, int b){ super(a); }  
}
```

## Overriding

A subclass can redefine a method that it inherits from its superclass. Now the method actually called depends on the type of the invoking object at runtime. The overriding method must have the same name, arguments, and return type as the overridden method.

The overriding method cannot be less public than the overridden method. The overriding method should not throw new or broader checked exceptions that are not declared by the original method. In the following example, the overridden version of the `print()` method is invoked because the invoking object is an instance of the derived class:

```
class Base  
{  
    void print()  
    {  
        System.out.println("Base");  
    }  
}  
class Derived extends Base  
{  
    void print()  
    {  
        System.out.println("Derived");  
    }  
    public static void main(String args[])  
    {  
        Base obj = new Derived();  
        obj.print();           // "Derived" is printed  
    }  
}
```

To invoke the superclass version of an overridden method from the subclass, use `super.methodName()`. In the above example, the subclass can use the

functionality of the superclass `print()` method by calling `super.print()`.

Methods declared as `final` cannot be overridden by subclasses. Even though constructors can be overloaded, they cannot be overridden because they are not inherited.

## Summary

In this section, we discussed the advantages of encapsulation. You saw that inheritance relationships are also known by the term "IS-A" relationships. A "HAS-A" relationship indicates that one class holds a reference to another. You should be able to identify the type of relationship from a given scenario. Be clear about the distinction between overriding and overloading.

## Exercise

### Question:

Which of the choices below can legally be inserted at the "insert code here" position in the following code?

```
class Parent
{
    public void print(int i)
    {
    }
}
public class Child extends Parent
{
    public static void main(String argv[])
    {
    }
    // insert code here
}
```

### Choices:

- A. `public void print(int i, byte b) throws Exception {}`
- B. `public void print(int i, long i) throws Exception {}`
- C. `public void print(long i) {}`
- D. `public void print(int i) throws Exception {}`
- E. `public int print(int i)`

### Correct choices:

- A, B, and C

### Explanation:

Option D will not compile because it attempts to throw a checked exception that is not declared in the `Parent` class. Option E will not compile because only the return type is different; the argument list and method name are the same. This is not allowed in both overriding and overloading. Options A, B, and C have different argument lists, so they represent *overloading*, not *overriding*. Because they can throw any exceptions, they are legal.

---

## Section 8. Threads

### Creating threads

Threads are objects in the Java language. A thread can be defined by extending the `java.lang.Thread` class or by implementing the `java.lang.Runnable` interface. The `run()` method should be overridden and should have the code that will be executed by the new thread. This method must be public with a `void` return type and should not take any arguments. See [Resources](#) for more information on threads.

#### Extending thread

If we need to inherit the behavior of the `Thread` class, we can have a subclass of it. In this case, the disadvantage is that you cannot extend any other class. For instance:

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Inside run()");
    }
}
```

Now the `Thread` object can be instantiated and started. Even though the execution method of your thread is called `run`, you do not call this method directly. Instead, you call the `start()` method of the `thread` class. When the `start()` method is called, the thread moves from the new state to the `Runnable` state. When the `Thread Scheduler` gives the thread a chance to execute, the `run()` method will be invoked. If the `run()` method is called directly, the code in it is executed by the current thread and not by the new thread. For instance:

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

#### Implementing Runnable

To implement `Runnable`, you need to define only the `run()` method:

```
class MyRunnable implements Runnable
{
    public void run()
    {
        System.out.println("Inside run()");
    }
}
```

Next, create an instance of the class and construct a thread, passing the instance of the class as an argument to the constructor:

```
MyRunnable mc = new MyRunnable();
Thread t = new Thread(mc);
```

To start the thread, use the `start()` method as shown here:

```
t.start();
```

## Thread constructors

The `Thread` constructors available are:

- `Thread()`
- `Thread(String name)`
- `Thread(Runnable runnable)`
- `Thread(Runnable runnable, String name)`
- `Thread(ThreadGroup g, Runnable runnable)`
- `Thread(ThreadGroup g, Runnable runnable, String name)`
- `Thread(ThreadGroup g, String name)`

## Thread states

Threads can be in one of the following states:

**New.** After the thread is instantiated, the thread is in the New state until the `start()` method is invoked. In this state, the thread is not considered alive.

**Runnable.** A thread comes into the runnable state when the `start()` method is invoked on it. It can also enter the runnable state from the running state or blocked state. The thread is considered alive when it is in this state.

**Running.** A thread moves from the runnable state into the running state when the thread scheduler chooses it to be the currently running thread.



**Alive, but not runnable.** A thread can be alive but not in a runnable state for a variety of reasons. It may be waiting, sleeping, or blocked.

**Waiting.** A thread is put into a waiting state by calling the `wait()` method. A call to `notify()` or `notifyAll()` may bring the thread from the waiting state into the runnable state. The `sleep()` method puts the thread into a sleeping state for a specified amount of time in milliseconds, as follows:

```
try {
    Thread.sleep(3 * 60 * 1000); // thread sleeps for 3 minutes
} catch (InterruptedException ex){}
```

**Blocked.** A thread may enter a blocked state while waiting for a resource like I/O or the lock of another object. In this case, the thread moves into the runnable state when the resource becomes available.

**Dead.** A thread is considered dead when its `run()` method is completely executed. A dead thread can never enter any other state, not even if the `start()` method is invoked on it.

## Important thread methods

The `sleep()` method causes the current thread to sleep for a given time in milliseconds:

```
public static void sleep(long millis) throws InterruptedException
```

The `yield()` method causes the current thread to move from the running state to the runnable state, so that other threads may get a chance to run. However, the next thread chosen for running might not be a different thread:

```
public static void yield()
```

The `isAlive()` method returns `true` if the thread upon which it is called has been started but not moved to the dead state:

```
public final boolean isAlive()
```

When a thread calls `join()` on another thread, the currently running thread will wait until the thread it joins with has completed:

```
void join()
void join(long millis)
void join(long millis, int nanos)
```

## Thread synchronization

Every object in Java code has one lock, which is useful for ensuring that only one thread accesses critical code in the object at a time. This synchronization helps prevent the object's state from getting corrupted. If a thread has obtained the lock, no other thread can enter the synchronized code until the lock is released. When the thread holding the lock exits the synchronized code, the lock is released. Now another thread can get the object's lock and execute the synchronized code. If a thread tries to get the lock of an object when another thread has the lock, the thread goes into a blocked state until the lock is released.

### **Synchronized**

You can use the `synchronized` keyword to declare a method as synchronized. This method cannot be accessed by multiple threads simultaneously. The `synchronized` keyword can also be used to mark a block of code as synchronized. For this, the argument passed should be the object whose lock you want to synchronize on. The syntax is shown below:

```
synchronized(obj)
{
    // here add statements to be synchronized
}
// obj is the object being synchronized
```

### **wait(), notify(), and notifyAll()**

The `wait()`, `notify()`, and `notifyAll()` methods are defined in the `java.lang.Object` class.

When the `wait()` method is invoked, a thread releases the lock on an object and moves from the running state to the waiting state. The `notify()` method is used to signal one of the threads waiting on the object to return to the runnable state. It is not possible to specify which of the waiting threads should be made runnable. The `notifyAll()` method causes all the waiting threads for an object to return to the runnable state.

A thread can invoke `wait()` or `notify()` on a particular object only if it currently holds the lock on that object. `wait()`, `notify()`, and `notifyAll()` should be called only from within the synchronized code.

## Summary

We saw how threads are defined in Java code either by implementing the `Runnable` interface or by extending the `Thread` class. For the exam, you should know all the `Thread` constructors. You should also know how a thread moves into and out of the blocked state. To prevent multiple threads from running the same object's method, we can use the `synchronized` keyword.

## Exercise

### Question:

Which of the following statements is *not* true about threads?

### Choices:

- A. If the `start()` method is invoked twice on the same `Thread` object, an exception is thrown at runtime.
- B. The order in which threads were started might differ from the order in which they actually run.
- C. If the `run()` method is directly invoked on a `Thread` object, an exception is thrown at runtime.
- D. If the `sleep()` method is invoked on a thread while executing synchronized code, the lock is not released.

### Correct choice:

- C

### Explanation:

Statement C is the correct answer; it is not true about threads. If the `run()` method is directly invoked on a `Thread` object, no exception is thrown at runtime. However, the code written in the `run()` method would be executed by the current thread, not by a new thread. So the correct way to start a thread is by invoking the `start()` method, which causes the `run()` method to be executed by the new thread. However, invoking the `start()` method twice on the same `Thread` object will cause an `IllegalThreadStateException` to be thrown at runtime, so A is true.

B is true because the order in which threads run is decided by the thread scheduler, no matter which thread was started first. D is true because a thread will not relinquish the locks it holds when it goes into the sleeping state.

---

## Section 9. Fundamental classes in the `java.lang` package

### Using the `Math` class

The `Math` class is `final` and all the methods defined in the `Math` class are

, which means you cannot inherit from the `Math` class and override these methods. Also, the `Math` class has a private constructor, so you cannot instantiate it.

The `Math` class has the following methods: `ceil()`, `floor()`, `max()`, `min()`, `random()`, `abs()`, `round()`, `sin()`, `cos()`, `tan()`, and `sqrt()`.

The `ceil()` method returns the smallest double value that is not less than the argument and is equal to a mathematical integer. For instance:

```
Math.ceil(5.4)           // gives 6
Math.ceil(-6.3)          // gives -6
```

The `floor()` method returns the largest double value that is not greater than the argument and is equal to a mathematical integer. For instance:

```
Math.floor(5.4)           // gives 5
Math.floor(-6.3)          // gives -7
```

The `max()` method takes two numeric arguments and returns the greater of the two. It is overloaded for `int`, `long`, `float`, or `double` arguments. For instance:

```
x = Math.max(10,-10); // returns 10
```

The `min` method takes two numeric arguments and returns the smaller of the two. It is overloaded for `int`, `long`, `float`, or `double` arguments. For instance:

```
x = Math.min(10,-10); // returns -10
```

The `random()` method returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

The `abs()` method returns the absolute value of the argument. It is overloaded to take an `int`, `long`, `float`, or `double` argument. For instance:

```
Math.abs(-33)           // returns 33
```

The `round()` method returns the integer closest to the argument (overloaded for `float` and `double` arguments). If the number after the decimal point is less than 0.5, `Math.round()` returns the same result as `Math.floor()`. In all the other cases, `Math.round()` works the same as `Math.ceil()`. For instance:

```
Math.round(-1.5) // result is -1
Math.round(1.8)  // result is 2
```

## Trigonometric functions

The `sin()`, `cos()`, and `tan()` methods return the sine, cosine, and tangent of an angle, respectively. In all three methods, the argument is the angle in radians. Degrees can be converted to radians by using `Math.toRadians()`. For instance:

```
x = Math.sin(Math.toRadians(90.0)); // returns 1.0
```

The `sqrt()` function returns the square root of an argument of type `double`. For instance:

```
x = Math.sqrt(25.0); // returns 5.0
```

If you pass a negative number to the `sqrt()` function, it returns `NaN` ("Not a Number").

## String and StringBuffer

### Immutability of String objects

As you know, `String`s are objects in Java code. These objects, however, are *immutable*. That is, their value, once assigned, can never be changed. For instance:

```
String msg = "Hello";  
msg += " World";
```

Here the original `String` "Hello" is not changed. Instead, a new `String` is created with the value "Hello World" and assigned to the variable `msg`. It is important to note that though the `String` object is immutable, the reference variable is not.

### String literals

In Java code, the `String` literals are kept in a memory pool for efficient use.

If you construct two `String`s using the same `String` literal without the `new` keyword, then only one `String` object is created. For instance:

```
String s1 = "hello";  
String s2 = "hello";
```

Here, both `s1` and `s2` point to the same `String` object in memory. As we said above, `String` objects are immutable. So if we attempt to modify `s1`, a new modified `String` is created. The original `String` referred to by `s2`, however, remains unchanged.

### StringBuffer

`StringBuffer` objects are *mutable*, which means their contents can be changed. For instance:

```
Stringbuffer s = new StringBuffer("123");  
s.append("456"); // Now s contains "123456"
```

You can see how this differs from the `String` object:

```
String s = new String("123");  
s.concat("456"); // Here s contains "123"
```

## Wrapper classes

Wrapper classes correspond to the primitive data types in the Java language. These classes represent the primitive values as objects. All the wrapper classes except `Character` have two constructors -- one that takes the primitive value and another that takes the `String` representation of the value. For instance:

```
Integer i1 = new Integer(50);  
Integer i2 = new Integer("50");
```

The `Character` class constructor takes a `char` type element as an argument:

```
Character c = new Character('A');
```

Wrapper objects are immutable. This means that once a wrapper object has a value assigned to it, that value cannot be changed.

### The `valueOf()` method

All wrapper classes (except `Character`) define a static method called `valueOf()`, which returns the wrapper object corresponding to the primitive value represented by the `String` argument. For instance:

```
Float f1 = Float.valueOf("1.5f");
```

The overloaded form of this method takes the representation base (binary, octal, or hexadecimal) of the first argument as the second argument. For instance:

```
Integer I = Integer.valueOf("10011110",2);
```

### Converting wrapper objects to primitives

All the numeric wrapper classes have six methods, which can be used to convert a numeric wrapper to any primitive numeric type. These methods are `byteValue`, `doubleValue`, `floatValue`, `intValue`, `longValue`, and `shortValue`. An example is shown below:

```
Integer i = new Integer(20);
```

```
byte b = i.byteValue();
```

### Parser methods

The six parser methods are `parseInt`, `parseDouble`, `parseFloat`, `parseLong`, `parseByte`, and `parseShort`. They take a `String` as the argument and convert it to the corresponding primitive. They throw a `NumberFormatException` if the `String` is not properly formed. For instance:

```
double d = Double.parseDouble("4.23");
```

It can also take a `radix(base)` as the second argument:

```
int i = Integer.parseInt("10011110",2);
```

### Base conversion

The `Integer` and `Long` wrapper classes have methods like `toBinaryString()` and `toOctalString()`, which convert numbers in base 10 to other bases. For instance:

```
String s = Integer.toHexString(25);
```

## Summary

We discussed the usage and signature of the various utility methods of the `Math` class. You are expected to memorize these for the exam. We also saw how `String` objects are immutable, while `StringBuffer` objects are not. The wrapper classes provide utility methods that allow you to represent primitives as objects. Make sure you are familiar with all the parsing and conversion methods.

## Exercise

### Question:

What will be the result of an attempt to compile and run the following program?

```
class Test
{
    public static void main(String args[])
    {
        String s1 = "abc";
        String s2 = "abc";
        s1 += "xyz";
        s2.concat("pqr");
        s1.toUpperCase();
        System.out.println(s1 + s2);
    }
}
```

**Choices:**

- A. "abcxyzabc"
- B. "abcxyzabcpqr"
- C. "ABCXYZabcpqr"
- D. "ABCXYZabc"
- E. Code does not compile

**Correct choice:**

- A

**Explanation:**

This code compiles and runs without errors, printing "abcxyzabc." In the given code, `s1` and `s2` initially refer to the same `String` object "abc." When "xyz" is concatenated to `s1`, a new `String` object "abcxyz" is created and `s1` is made to refer to it. Note that `s2` still refers to the original `String` object "abc," which is unchanged. The `concat()` and `toUpperCase()` methods do not have any effect, because the new `String` objects created as a result of these operations do not have any references stored. So `s1` contains "abcxyz" and `s2` contains "abc" at the end, making A the correct result.

---

## Section 10. The Collections framework

### Collections and collections interfaces

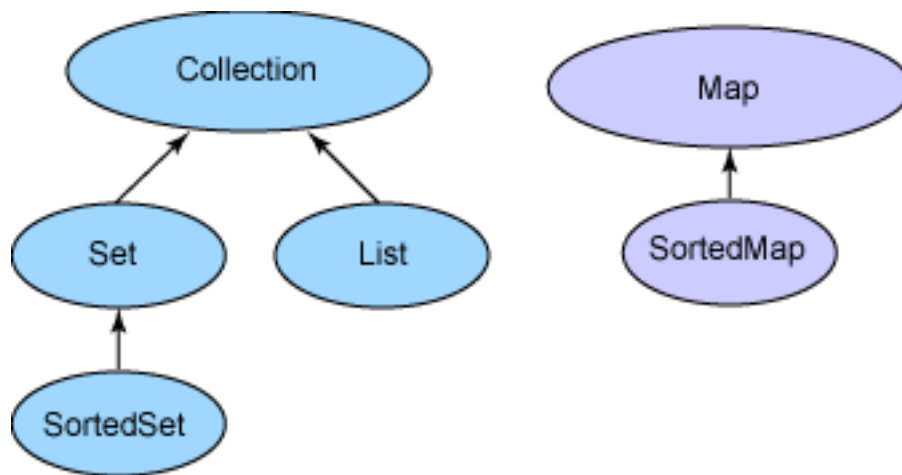
`Collections` are objects used to group together and manipulate multiple data elements. They can dynamically grow and shrink, which is their advantage over arrays. `Collections` provide methods to add objects to a collection, remove objects from it, check if an object is present in it, retrieve objects from it, and iterate through it. See [Resources](#) for more information on the Collections framework.

**Collection interfaces**

The inheritance hierarchy of the core `collection` interface is shown here:

**Figure 2. Inheritance hierarchy of the core collection interface**





`Collection` is the root interface, from which the `Set` and `List` interfaces extend. The `Map` interface does not extend the `Collection` interface. Also, do not confuse the `Collection` *interface* with the `Collections` *class*, which holds static utility methods for collections.

Let's analyze the basic behavior of these interfaces. The `List` interface represents ordered collections while `Set` cannot contain duplicate elements. The `Map` interface matches unique keys to values. `SortedSet` holds elements in sorted order while `SortedMap` orders the mapping in the sorted order of keys. The classes that implement these interfaces are listed below:

<b>Set</b>	<code>HashSet</code>	<code>TreeSet</code>	<code>LinkedHashSet</code>	
<b>List</b>	<code>Vector</code>	<code>ArrayList</code>	<code>LinkedList</code>	
<b>Map</b>	<code>HashMap</code>	<code>HashTable</code>	<code>TreeMap</code>	<code>LinkedHashMap</code>

### Set classes

The classes implementing the `Set` interface do not allow duplicate elements.

A `HashSet` is not ordered or sorted. This class offers constant time performance for basic operations like `add` and `remove`. `TreeSet` arranges the elements in ascending element order, sorted according to the natural order of the elements.

A `LinkedHashSet` is an ordered `HashSet`, which gives the elements in the order of insertion or last access. For instance:

```

LinkedHashSet linkSet = new LinkedHashSet();
linkSet.add("mango");
linkSet.add("apple");
linkSet.add("mango");
linkSet.add("banana");
Iterator i = linkSet.iterator();
while(i.hasNext())
    System.out.print(i.next());    // Prints "mango apple banana"
  
```

### List classes

A `List` is an ordered collection, which allows positional access and search.

The classes implementing `List` are ordered by index position. An `ArrayList` enables fast iteration and constant speed positional access. A `Vector` is similar to `ArrayList`, only slower because it is synchronized. `LinkedList` allows fast insertion and deletion at the beginning or end. It is commonly used for implementing stacks and queues. For instance:

```
ArrayList list = new ArrayList();
list.add("mango");
list.add("apple");
list.add("mango");
list.add("banana");
Iterator i = list.iterator();
while(i.hasNext())
    System.out.print(i.next());    // Prints "mango apple mango banana"
```

### Map classes

The classes implementing the `Map` interface map unique keys to specific values.

The `HashMap` class is not sorted or ordered. It allows one null key and many null values.

`Hashtable` is similar to `HashMap`, but does not allow null keys and values. It's also slower than `HashMap` because it is synchronized. The J2SE 1.4 `LinkedHashMap` class iterates by insertion or last accessed order. It allows one null key and many null values.

`TreeMap` is a map in ascending key order, sorted according to the natural order for the key's class.

### *The equals() method of java.lang.Object*

The `equals()` method, shown below, should implement an equivalence relation:

```
public boolean equals(Object obj)
```

In addition, the `equals()` method is:

- Reflexive: For any reference value `x`, `x.equals(x)` should return `true`.
- Symmetric: For any reference values `x` and `y`, `x.equals(y)` should return `true` *only if* `y.equals(x)` returns `true`.
- Transitive: For any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- Consistent: For any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equal comparisons on the object is modified.

For any non-null reference value `x`, `x.equals(null)` should return `false`.

### ***The hashCode() method of java.lang.Object***

The `hashCode()` method, shown below, of the `java.lang.Object` class returns the hash code value for the object. The hash code value of an object gives an integer that can be used by some collection classes like `HashSet` and `Hashtable` to map its objects into buckets in memory. The process of hashing enables efficient storage and retrieval of objects from the collection.

```
public int hashCode()
```

If two objects are equal as given by the `equals()` method, then their hash codes must be the same. So whenever `equals()` is overridden, `hashCode()` also should be implemented.

However, the reverse is not required. Two objects that are not equal can have the same hash code. This is because it is not always possible to ensure unique hash codes. However, it is better to have distinct hash codes for distinct objects to improve efficiency and performance.

The object comparison logic used by the `equals()` method usually involves instance variables of the two objects. The same instance variables should also be used by the `hashCode()` implementation. For instance:

```
public class Test
{
    int i;
    public boolean equals(Object o)
    {
        Test t = (Test)o;
        if (t.i == this.i)
            return true;
        return false;
    }
    public int hashCode()
    {
        return i * 31;
    }
}
```

Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used in equal comparison on the object is modified. It is inappropriate to involve a random number when computing the hash code, because it would not return the same hash code for multiple invocations of the method. The hash code of a null element is defined as zero.

When you implement the `equals()` and `hashCode()` methods, it is not appropriate to use transient variables.

## Summary

We learned about the different types of collections like `Lists`, `Sets`, and `Maps`. It is important to understand the behavior of each of them and the situation in which they are preferred or required. We also saw the significance of the `equals()` and `hashCode()` methods. You should know the appropriate ways to implement these methods.

## Exercise

### Question:

Which is the most suitable Java collection class for storing various companies and their stock prices? It is required that the class should support synchronization inherently.

### Choices:

- A. Hashtable
- B. HashMap
- C. LinkedHashMap
- D. HashSet
- E. TreeMap

### Correct choice:

- A

### Explanation:

To store the company names and their corresponding stock prices, a `Map` implementation would be appropriate because it stores key-value pairs. `HashSet` is not the correct answer because it does not implement the `Map` interface. `Hashtable`, `HashMap`, `TreeMap`, and `LinkedHashMap` are the classes that implement the `Map` interface. Among these, only the `Hashtable` class is inherently synchronized.

---

## Section 11. Summary

### Summary

This wraps up the SCJP 1.4 tutorial. It would be advisable to test each new concept you learned with the help of small practice examples. This will help foster your

understanding of the material presented in this tutorial.

The most difficult questions in the exam center around threads and garbage collection. Pay extra attention to these chapters. Remember that the garbage collector and thread scheduler implementations are platform specific.

It is not enough if you just memorize the syntax and APIs that are part of the exam. It is very important to acquire a deep understanding of the core concepts covered in the exam objectives.

Regardless of your reasons for going after certification -- whether you want the professional advantage or the personal challenge (or whether getting certified is a job requirement) -- your approach to the task will determine the success of its outcome. Good preparation, attention to details, and a positive attitude are essential if you want to pass the SCJP exam.

I hope this tutorial is helpful in preparing you for your exam, and I wish you the best of luck.

I would like to acknowledge Seema Manivannan for her invaluable contribution in writing this tutorial.

## Resources

- - Khalid Mughal and Rolf Rasmussen's [A Programmer's Guide to Java Certification](#) (Addison-Wesley, 1999) is a great reference if you're seeking a structured learning format.
  - Kathy Sierra and Bert Bates's [Sun Certified Programmer & Developer for Java 2 Study Guide \(Exam 310-035 & 310-027\)](#) is another useful study guide.
  - William Brogden's [Java 2 Exam Prep, Second Edition \(Exam: 310-025\)](#) and [Java 2 Exam Cram, Second Edition \(Exam 310-025\)](#) are both excellent books. Previously published by The Coriolis Group, the series is being relaunched by Que Publishing. *Exam Cram* is a quick review, whereas the older *Exam Prep* is a fairly comprehensive introduction to Java language fundamentals.
  - [William Brogden's home page](#) contains a nice selection of Java certification information and resources.
  - Check out " [An SCJP 1.4 certification primer](#) " ( *developerWorks*, June 2003) by Pradeep Chopra, the author of this tutorial.
  - The " [Java Collections Framework](#) " tutorial ( *developerWorks*, April 2001) is helpful if you need to brush up on the Java Collections API.
  - The [Programming With Assertions](#) study guide, also from Sun, is recommended reading before you take the new SCJP 1.4 exam.
  - In his monthly *developerWorks* column, [Magic with Merlin](#) , John Zukowski covers new features of JDK 1.4, including one installment titled " [Working with assertions](#) " ( *developerWorks*, February 2002).
  - For more on garbage collection, see the following articles from *developerWorks*:
    - In " [Mash that trash](#) ", you'll learn more about how incremental compaction reduces pause times.
    - The three-part series " [Sensible sanitation](#) " describes the garbage collection strategy employed by the IBM 1.2 and 1.3 Developer Kits for the Java Platform.
    - In his October 2003 installment of *Java theory and practice*, Brian Goetz provides " [A brief history of garbage collection](#). "
    - In " [Fine-tuning Java garbage collection performance](#), " learn how to detect and troubleshoot garbage collection issues.
  - If you're interested in learning more about threading in the Java language, " [Introduction to Java threads](#) " by Brian Goetz ( *developerWorks*, September 2002) explores threading basics, exchanging data between threads, controlling threads, and how threads can communicate with each other.

- Marcus Green is a well-regarded Java certification guru. His [Java certification site](#) is home to an SCJP tutorial, free mock examinations, a certification discussion forum, and information about his Java certification training courses.
- The [IBM Education Web site](#) is a good place to start if you're looking for personalized Java programming training.
- Whizlabs Software specializes in IT certification simulators, including the [Whizlabs Java Certification \(SCJP 1.4\) Exam Simulator](#) for J2SE 1.4.
- Whizlabs Software also offers [instructor-led, online training for Java Certification \(SCJP 1.4\)](#), delivered live by the author of the SCJP 1.4 exam simulator, as well as [Java certification discussion forums](#).
- [JavaRanch](#) ("a friendly place for Java greenhorns") hosts a community discussion forum and the SCJP rules roundup, along with a free certification mock exam.
- The [IBM professional certifications](#) site lists the available certification programs, categorized by brand.
- Sun Microsystems's [J2SE 1.2 and J2SE 1.4 API specs](#) are essential reading for the SCJP exam.
- You can learn quite a lot about the `hashCode()` and `equals()` methods by reading Chapter 3 of [Effective Java](#), available online courtesy of Sun Microsystems.
- See the [developerWorks Java technology tutorials index](#) for a complete listing of free tutorials.
- You can also find hundreds of articles about every aspect of Java programming in the [developerWorks Java technology zone](#).



# Programming With Assertions

---

An *assertion* is a statement in the Java™ programming language that enables you to test your assumptions about your program. For example, if you write a method that calculates the speed of a particle, you might assert that the calculated speed is less than the speed of light.

Each assertion contains a boolean expression that you believe will be true when the assertion executes. If it is not true, the system will throw an error. By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors.

Experience has shown that writing assertions while programming is one of the quickest and most effective ways to detect and correct bugs. As an added benefit, assertions serve to document the inner workings of your program, enhancing maintainability.

This document shows you how to program with assertions. It covers the topics:

[Introduction](#)

[Putting Assertions Into Your Code](#)

[Compiling Files That Use Assertions](#)

[Enabling and Disabling Assertions](#)

[Compatibility With Existing Programs](#)

[Design FAQ](#)

---

## Introduction

The assertion statement has two forms. The first, simpler form is:



```
assert Expression1 ;
```

where *Expression*<sub>1</sub> is a boolean expression. When the system runs the assertion, it evaluates *Expression*<sub>1</sub> and if it is false throws an [AssertionError](#) with no detail message.

The second form of the assertion statement is:

```
assert Expression1 : Expression2 ;
```

where:

*Expression*<sub>1</sub> is a boolean expression.

*Expression*<sub>2</sub> is an expression that has a value. (It cannot be an invocation of a method that is declared void.)

Use this version of the assert statement to provide a detail message for the `AssertionError`. The system passes the value of *Expression*<sub>2</sub> to the appropriate `AssertionError` constructor, which uses the string representation of the value as the error's detail message.

The purpose of the detail message is to capture and communicate the details of the assertion failure. The message should allow you to diagnose and ultimately fix the error that led the assertion to fail. Note that the detail message is *not* a user - level error message, so it is generally unnecessary to make these messages understandable in isolation, or to internationalize them. The detail message is meant to be interpreted in the context of a full stack trace, in conjunction with the source code containing the failed assertion.

Like all uncaught exceptions, assertion failures are generally labeled in the stack trace with the file and line number from which they were thrown. The second form of the assertion statement should be used in preference to the first only when the program has some additional information that might help diagnose the failure. For example, if *Expression*<sub>1</sub> involves the relationship between two variables *x* and *y*, the second form should be used. Under these circumstances, a reasonable candidate for *Expression*<sub>2</sub> would be "x: " + *x* + ", y: " + *y*.

In some cases *Expression*<sub>1</sub> may be expensive to evaluate. For example, suppose you

write a method to find the minimum element in an unsorted list, and you add an assertion to verify that the selected element is indeed the minimum. The work done by the assert will be at least as expensive as the work done by the method itself. To ensure that assertions are not a performance liability in deployed applications, assertions can be enabled or disabled when the program is started, and are disabled by default. Disabling assertions eliminates their performance penalty entirely. Once disabled, they are essentially equivalent to *empty statements* in semantics and performance. See [Enabling and Disabling Assertions](#) for more information.

The addition of the assert keyword to the Java programming language has implications for existing code. See [Compatibility With Existing Programs](#) for more information.

---

## Putting Assertions Into Your Code

There are many situations where it is good to use assertions. This section covers some of them:

[Internal Invariants](#)

[Control - Flow Invariants](#)

[Preconditions, Postconditions, and Class Invariants](#)

There are also a few situations where you should *not* use them:

Do *not* use assertions for argument checking in public methods.

Argument checking is typically part of the published specifications (or *contract*) of a method, and these specifications must be obeyed whether assertions are enabled or disabled. Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (such as `IllegalArgumentException`, `IndexOutOfBoundsException`, or `NullPointerException`). An assertion failure will not throw an appropriate exception.

Do *not* use assertions to do any work that your application requires for correct operation.

Because assertions may be disabled, programs must not assume that the

boolean expression contained in an assertion will be evaluated. Violating this rule has dire consequences. For example, suppose you wanted to remove all of the null elements from a list `names`, and knew that the list contained one or more nulls. It would be wrong to do this:

```
// Broken! - action is contained in assertion
assert names.remove(null);
```

The program would work fine when asserts were enabled, but would fail when they were disabled, as it would no longer remove the null elements from the list. The correct idiom is to perform the action before the assertion and then assert that the action succeeded:

```
// Fixed - action precedes assertion
boolean nullsRemoved = names.remove(null);
assert nullsRemoved; // Runs whether or not asserts are enabled
```

As a rule, the expressions contained in assertions should be free of *side effects*: evaluating the expression should not affect any state that is visible after the evaluation is complete. One exception to this rule is that assertions can modify state that is used only from within other assertions. [An idiom that makes use of this exception](#) is presented later in this document.

## Internal Invariants

Before assertions were available, many programmers used comments to indicate their assumptions concerning a program's behavior. For example, you might have written something like this to explain your assumption about an else clause in a multiway if - statement:

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else { // We know (i % 3 == 2)
    ...
}
```

You should now use an assertion whenever you would have written a comment that asserts an invariant. For example, you should rewrite the previous if - statement like

this:

```

if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
    ...
}

```

Note, incidentally, that the assertion in the above example may fail if *i* is negative, as the `%` operator is not a true *modulus* operator, but computes the *remainder*, which may be negative.

Another good candidate for an assertion is a switch statement with no default case. The absence of a default case typically indicates that a programmer believes that one of the cases will always be executed. The assumption that a particular variable will have one of a small number of values is an invariant that should be checked with an assertion. For example, suppose the following switch statement appears in a program that handles playing cards:

```

switch(suit) {
    case Suit.CLUBS:
        ...
        break;

    case Suit.DIAMONDS:
        ...
        break;

    case Suit.HEARTS:
        ...
        break;

    case Suit.SPADES:
        ...
}

```

It probably indicates an assumption that the suit variable will have one of only four values. To test this assumption, you should add the following default case:

```
default:
    assert false : suit;
```

If the `suit` variable takes on another value and assertions are enabled, the `assert` will fail and an `AssertionError` will be thrown.

An acceptable alternative is:

```
default:
    throw new AssertionError(suit);
```

This alternative offers protection even if assertions are disabled, but the extra protection adds no cost: the `throw` statement won't execute unless the program has failed. Moreover, the alternative is legal under some circumstances where the `assert` statement is not. If the enclosing method returns a value, each case in the `switch` statement contains a `return` statement, and no `return` statement follows the `switch` statement, then it would cause a syntax error to add a default case with an assertion. (The method would return without a value if no case matched and assertions were disabled.)

## Control - Flow Invariants

The previous example not only tests an invariant, it also checks an assumption about the application's flow of control. The author of the original `switch` statement probably assumed not only that the `suit` variable would always have one of four values, but also that one of the four cases would always be executed. It points out another general area where you should use assertions: place an assertion at any location you assume will not be reached. The assertions statement to use is:

```
assert false;
```

For example, suppose you have a method that looks like this:

```
void foo() {
    for (...) {
        if (...)
            return;
    }
    // Execution should never reach this point!!!
}
```

Replace the final comment so that the code now reads:

```
void foo() {
    for (...) {
        if (...)
            return;
    }
    assert false; // Execution should never reach this point!
}
```

---

Note: Use this technique with discretion. If a statement is unreachable as defined in the Java Language Specification ( [JLS 14.20](#)), you will get a compile time error if you try to assert that it is not reached. Again, an acceptable alternative is simply to throw an `AssertionError`.

---

## Preconditions, Postconditions, and Class Invariants

While the `assert` construct is not a full - blown *design - by - contract* facility, it can help support an informal design - by - contract style of programming. This section shows you how to use asserts for:

[Preconditions](#) — what must be true when a method is invoked.

- [Lock - Status Preconditions](#) — preconditions concerning whether or not a given lock is held.

[Postconditions](#) — what must be true after a method completes successfully.

[Class invariants](#) — what must be true about each instance of a class.

### Preconditions

By convention, preconditions on *public* methods are enforced by explicit checks that throw particular, specified exceptions. For example:

```
/**
 * Sets the refresh rate.
```

```

*
* @param rate refresh rate, in frames per second.
* @throws IllegalArgumentException if rate <= 0 or
*         rate > MAX_REFRESH_RATE.
*/
public void setRefreshRate(int rate) {
    // Enforce specified precondition in public method
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
        throw new IllegalArgumentException("Illegal rate: " + rate);

    setRefreshInterval(1000/rate);
}

```

This convention is unaffected by the addition of the assert construct. Do not use assertions to check the parameters of a public method. An assert is inappropriate because the method guarantees that it will always enforce the argument checks. It must check its arguments whether or not assertions are enabled. Further, the assert construct does not throw an exception of the specified type. It can throw only an `AssertionError`.

You can, however, use an assertion to test a *nonpublic* method's precondition that you believe will be true no matter what a client does with the class. For example, an assertion *is* appropriate in the following "helper method" that is invoked by the previous method:

```

/**
 * Sets the refresh interval (which must correspond to a legal frame rate).
 *
 * @param interval refresh interval in milliseconds.
 */
private void setRefreshInterval(int interval) {
    // Confirm adherence to precondition in nonpublic method
    assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;

    ... // Set the refresh interval
}

```

Note, the above assertion will fail if `MAX_REFRESH_RATE` is greater than 1000 and the client selects a refresh rate greater than 1000. This would, in fact, indicate a bug in the library!

## Lock - Status Preconditions

Classes designed for multithreaded use often have non - public methods with preconditions relating to whether or not some lock is held. For example, it is not uncommon to see something like this:

```
private Object[] a;

public synchronized int find(Object key) {
    return find(key, a, 0, a.length);
}

// Recursive helper method - always called with a lock on this object
private int find(Object key, Object[] arr, int start, int len) {
    ...
}
```

A static method called `holdsLock` has been added to the `Thread` class to test whether the current thread holds the lock on a specified object. This method can be used in combination with an `assert` statement to supplement a comment describing a lock - status precondition, as shown in the following example:

```
// Recursive helper method - always called with a lock on this.
private int find(Object key, Object[] arr, int start, int len) {
    assert Thread.holdsLock(this); // lock - status assertion
    ...
}
```

Note that it is also possible to write a lock - status assertion asserting that a given lock *isn't* held.

## Postconditions

You can test postcondition with assertions in both public and nonpublic methods. For example, the following public method uses an `assert` statement to check a post condition:

```
/**
 * Returns a BigInteger whose value is (this - 1 mod m).
 */
```



```

* @param m the modulus.
* @return this-1 mod m.
* @throws ArithmeticException m <= 0, or this BigInteger
*         has no multiplicative inverse mod m (that is, this BigInteger
*         is not relatively prime to m).
*/
public BigInteger modInverse(BigInteger m) {
    if (m.signum <= 0)
        throw new ArithmeticException("Modulus not positive: " + m);

    ... // Do the computation

    assert this.multiply(result).mod(m).equals(ONE) : this;
    return result;
}

```

Occasionally it is necessary to save some data prior to performing a computation in order to check a postcondition. You can do this with two assert statements and a simple inner class that saves the state of one or more variables so they can be checked (or rechecked) after the computation. For example, suppose you have a piece of code that looks like this:

```

void foo(int[] array) {
    // Manipulate array
    ...

    // At this point, array will contain exactly the ints that it did
    // prior to manipulation, in the same order.
}

```

Here is how you could modify the above method to turn the textual assertion of a postcondition into a functional one:

```

void foo(final int[] array) {

    // Inner class that saves state and performs final consistency check
    class DataCopy {
        private int[] arrayCopy;

        DataCopy() { arrayCopy = (int[]) array.clone(); }

        boolean isConsistent() { return Arrays.equals(array, arrayCopy); }
    }
}

```

```

    }

    DataCopy copy = null;

    // Always succeeds; has side effect of saving a copy of array
    assert ((copy = new DataCopy()) != null);

    ... // Manipulate array

    // Ensure array has same ints in same order as before manipulation.
    assert copy.isConsistent();
}

```

You can easily generalize this idiom to save more than one data field, and to test arbitrarily complex assertions concerning pre - computation and post - computation values.

You might be tempted to replace the first assert statement (which is executed solely for its side - effect) by the following, more expressive statement:

```
copy = new DataCopy();
```

Don't make this replacement. The statement above would copy the array whether or not asserts were enabled, violating the principle that assertions should have no cost when disabled.

## Class Invariants

A class invariant is a type of [internal invariant](#) that applies to every instance of a class at all times, except when an instance is in transition from one consistent state to another. A class invariant can specify the relationships among multiple attributes, and should be true before and after any method completes. For example, suppose you implement a balanced tree data structure of some sort. A class invariant might be that the tree is balanced and properly ordered.

The assertion mechanism does not enforce any particular style for checking invariants. It is sometimes convenient, though, to combine the expressions that check required constraints into a single internal method that can be called by assertions. Continuing the balanced tree example, it might be appropriate to implement a private method that checked that the tree was indeed balanced as per the dictates of the data

structure:

```
// Returns true if this tree is properly balanced
private boolean balanced() {
    ...
}
```

Because this method checks a constraint that should be true before and after any method completes, each public method and constructor should contain the following line immediately prior to its return:

```
assert balanced();
```

It is generally unnecessary to place similar checks at the head of each public method unless the data structure is implemented by native methods. In this case, it is possible that a memory corruption bug could corrupt a "native peer" data structure in between method invocations. A failure of the assertion at the head of such a method would indicate that such memory corruption had occurred. Similarly, it may be advisable to include class invariant checks at the heads of methods in classes whose state is modifiable by other classes. (Better yet, design classes so that their state is not directly visible to other classes!)

## Advanced Uses

The following sections discuss topics that apply only to resource - constrained devices and to systems where asserts must not be disabled in the field. If you have no interest in these topics, skip to the next section, ["Compiling Files that Use Assertions"](#).

### Removing all Trace of Assertions from Class Files

Programmers developing applications for resource - constrained devices may wish to strip assertions out of class files entirely. While this makes it impossible to enable assertions in the field, it also reduces class file size, possibly leading to improved class loading performance. In the absence of a high quality JIT, it could lead to decreased footprint and improved runtime performance.

The assertion facility offers no direct support for stripping assertions out of class files. The `assert` statement may, however, be used in conjunction with the "conditional compilation" idiom described in [JLS 14.20](#), enabling the compiler to eliminate all traces of these asserts from the class files that it generates:

```
static final boolean asserts = ... ; // false to eliminate asserts

if (asserts)
    assert <expr> ;
```

## Requiring that Assertions are Enabled

Programmers of certain critical systems might wish to ensure that assertions are not disabled in the field. The following static initialization idiom prevents a class from being initialized if its assertions have been disabled:

```
static {
    boolean assertsEnabled = false;
    assert assertsEnabled = true; // Intentional side effect!!!
    if (!assertsEnabled)
        throw new RuntimeException("Asserts must be enabled!!!");
}
```

Put this static - initializer at the top of your class.

---

## Compiling Files That Use Assertions

In order for the `javac` compiler to accept code containing assertions, you must use the `-source 1.4` command - line option as in this example:

```
javac -source 1.4 MyClass.java
```

This flag is necessary so as not to cause source [compatibility](#) problems.

## Enabling and Disabling Assertions

By default, assertions are disabled at runtime. Two command - line switches allow you to selectively enable or disable assertions.

To enable assertions at various granularities, use the `-enableassertions`, or `-ea`, switch. To disable assertions at various granularities, use the `-disableassertions`, or `-da`, switch. You specify the granularity with the arguments that you provide to the switch:

- no arguments

- Enables or disables assertions in all classes except system classes.

- packageName...*

- Enables or disables assertions in the named package and any subpackages.

- ...

- Enables or disables assertions in the unnamed package in the current working directory.

- className*

- Enables or disables assertions in the named class

For example, the following command runs a program, `BatTutor`, with assertions enabled in only package `com.wombat.fruitbat` and its subpackages:

```
java -ea:com.wombat.fruitbat... BatTutor
```

If a single command line contains multiple instances of these switches, they are processed in order before loading any classes. For example, the following command runs the `BatTutor` program with assertions enabled in package `com.wombat.fruitbat` but disabled in class `com.wombat.fruitbat.Brickbat`:

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat BatTutor
```

The above switches apply to all class loaders. With one exception, they also apply to *system classes* (which do not have an explicit class loader). The exception concerns the switches with no arguments, which (as indicated above) do not apply to system classes. This behavior makes it easy to enable asserts in all classes except for system classes, which is commonly desirable.

To enable assertions in all system classes, use a different

switch: - enablesystemassertions, or - esa. Similarly, to disable assertions in system classes, use - disablesystemassertions, or - dsa.

For example, the following command runs the BatTutor program with assertions enabled in system classes, as well as in the com.wombat.fruitbat package and its subpackages:

```
java - esa - ea:com.wombat.fruitbat...
```

The assertion status of a class (enabled or disabled) is set at the time it is initialized, and does not change. There is, however, one corner case that demands special treatment. It is possible, though generally not desirable, to execute methods or constructors prior to initialization. This can happen when a class hierarchy contains a circularity in its static initialization.

If an assert statement executes before its class is initialized, the execution must behave as if assertions were enabled in the class. This topic is discussed in detail in [the assertions specification](#).

## Compatibility With Existing Programs

The addition of the assert keyword to the Java programming language does not cause any problems with preexisting binaries (.class files). If you try to compile an application that uses assert as an identifier, however, you will receive a warning or error message. In order to ease the transition from a world where assert is a legal identifier to one where it isn't, the compiler supports two modes of operation in this release:

source mode 1.3 (default) — the compiler accepts programs that use assert as an identifier, but issues warnings. In this mode, programs are *not* permitted to use the assert statement.

source mode 1.4 — the compiler generates an error message if the program uses assert as an identifier. In this mode, programs *are* permitted to use the assert statement.

Unless you specifically request source mode 1.4 with the - source 1.4 flag, the compiler operates in source mode 1.3. *If you forget to use this flag, programs that use the new assert statement will not compile.* Having the compiler use the old semantics as its default behavior (that is, allowing assert to be used as an identifier)

was done for maximal source compatibility. Source mode 1.3 is likely to be phased out over time.

---

## Design FAQ

Here is a collection of frequently asked questions concerning the design of the assertion facility.

[General Questions](#)

[Compatibility](#)

[Syntax and Semantics](#)

[The AssertionError Class](#)

[Enabling and Disabling Assertions](#)

### General Questions

1. Why provide an assertion facility, given that one can program assertions atop the Java programming language with no special support?

Although ad hoc implementations are possible, they are of necessity either ugly (requiring an if statement for each assertion) or inefficient (evaluating the condition even if assertions are disabled). Further, each ad hoc implementation has its own means of enabling and disabling assertions, which lessens the utility of these implementations, especially for debugging in the field. As a result of these shortcomings, assertions have never become a part of the culture among engineers using the Java programming language. Adding assertion support to the platform stands a good chance of rectifying this situation.

2. Why does this facility justify a language change, as opposed to a library solution?

We recognize that a language change is a serious effort, not to be undertaken lightly. The library approach was considered. It was, however, deemed

essential that the runtime cost of assertions be negligible if they are disabled. In order to achieve this with a library, the programmer is forced to hard - code each assertion as an if statement. Many programmers would not do this. Either they would omit the if statement and performance would suffer, or they would ignore the facility entirely. Note also that assertions were contained in James Gosling's original specification for the Java programming language. Assertions were removed from the Oak specification because time constraints prevented a satisfactory design and implementation.

3. Why not provide a full - fledged *design - by - contract* facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?

We considered providing such a facility, but were unable to convince ourselves that it is possible to graft it onto the Java programming language without massive changes to the Java platform libraries, and massive inconsistencies between old and new libraries. Further, we were not convinced that such a facility would preserve the simplicity that is the hallmark of the Java programming language. On balance, we came to the conclusion that a simple boolean assertion facility was a fairly straight - forward solution and far less risky. It's worth noting that adding a boolean assertion facility to the language doesn't preclude adding a full - fledged design - by - contract facility at some time in the future.

The simple assertion facility does enable a limited form of [design - by - contract style programming](#). The assert statement is appropriate for nonpublic precondition, postcondition and class invariant checking. Public precondition checking should still be performed by checks inside methods that result in particular, documented exceptions, such as `IllegalArgumentException` and `IllegalStateException`.

4. In addition to boolean assertions, why not provide an assert - like construct to suppress the execution of an entire block of code if assertions are disabled?

Providing such a construct would encourage programmers to put complex assertions inline, when they are better relegated to separate methods.

## Compatibility



1. Won't the new keyword cause compatibility problems with existing programs that use assert as an identifier?

Yes, for source files. (Binaries for classes that use assert as an identifier will continue to work fine.) To ease the transition, we implemented [a strategy](#) whereby developers can continue using assert as an identifier during a transitional period.

2. Doesn't this facility produce class files that cannot be run against older JREs?

Yes. Class files will contain calls to the new ClassLoader and Class methods, such as desiredAssertionStatus. If a class file containing calls to these methods is run against an older JRE (whose ClassLoader class doesn't define the methods), the program will fail at run time, throwing a NoSuchMethodError. It is generally the case that programs using new facilities are not compatible with older releases.

## Syntax and Semantics

1. Why allow primitive types in *Expression<sub>2</sub>*?

There is no compelling reason to restrict the type of this expression. Allowing arbitrary types provides convenience for developers who, for example, want to associate a unique integer code with each assertion. Further, it makes this expression feel like the argument of System.out.println(...), which is seen as desirable.

## The AssertionError Class

1. When an AssertionError is generated by an assert statement in which *Expression<sub>2</sub>* is absent, why isn't the program text of the asserted condition used as the detail message (for example, "height < maxHeight")?

While doing so might improve out - of - the - box usefulness of assertions in some cases, the benefit doesn't justify the cost of adding all those string

constants to .class files and runtime images.

2. Why doesn't an `AssertionError` allow access to the object that generated it? Similarly, why not pass an arbitrary object from the assertion to the `AssertionError` constructor in place of a detail message?

Access to these objects would encourage programmers to attempt to recover from assertion failures, which defeats the purpose of the facility.

3. Why not provide context accessors (like `getFile`, `getline`, `getMethod`) on `AssertionError`?

This facility is best provided on `Throwable`, so it may be used for all throwables, not just just assertion errors. We enhanced `Throwable` with the `getStackTrace` method to provide this functionality.

4. Why is `AssertionError` a subclass of `Error` rather than `RuntimeException`?

This issue was controversial. The expert group discussed it at length, and came to the conclusion that `Error` was more appropriate to discourage programmers from attempting to recover from assertion failures. It is, in general, difficult or impossible to localize the source of an assertion failure. Such a failure indicates that the program is operating "outside of known space," and attempts to continue execution are likely to be harmful. Further, convention dictates that methods specify most runtime exceptions they may throw (with `@throws` doc comments). It makes little sense to include in a method's specification the circumstances under which it may generate an assertion failure. Such information may be regarded as an implementation detail, which can change from implementation to implementation and release to release.

## Enabling and Disabling Assertions

1. Why not provide a compiler flag to completely eliminate assertions from object files?

It is a firm requirement that it be possible to enable assertions in the field, for

enhanced serviceability. It would have been possible to also permit developers to eliminate assertions from object files at compile time. Assertions can contain side effects, though they should not, and such a flag could therefore alter the behavior of a program in significant ways. It is viewed as good thing that there is only one semantics associated with each valid Java program. Also, we want to encourage users to leave asserts in object files so they can be enabled in the field. Finally, the spec demands that assertions behave as if enabled when a class runs before it is initialized. It would be impossible to offer these semantics if assertions were stripped from the class file. Note, however, that the standard "conditional compilation idiom" described in JLS 14.20 can be used to achieve this effect for developers who really want it.

2. Why do the commands that enable and disable assertions use package - tree semantics instead of the more traditional package semantics?

Hierarchical control is useful, as programmers really do use package hierarchies to organize their code. For example, package - tree semantics allow assertions to be enabled or disabled in all of Swing at one time.

3. Why does `setClassAssertionStatus` return a boolean instead of throwing an exception if it is invoked when it's too late to set the assertion status (that is, if the named class has already been initialized)?

No action (other than perhaps a warning message) is necessary or desirable if it's too late to set the assertion status. An exception seems unduly heavyweight.

4. Why not overload a single method name to take the place of `setDefaultAssertionStatus` and `setAssertionStatus`?

Clarity in method naming is for the greater good. Overloading tends to cause confusion.

5. Why not tweak the semantics of `desiredAssertionStatus` to make it more "programmer friendly" by returning the actual assertion status if a class is already initialized?

It's not clear that there would be any use for the resulting method. The method

isn't designed for application programmer use, and it seems inadvisable to make it slower and more complex than necessary.

6. Why is there no `RuntimePermission` to prevent applets from enabling/disabling assertions?

While applets have no reason to call any of the `ClassLoader` methods for modifying assertion status, allowing them to do so seems harmless. At worst, an applet can mount a weak denial - of - service attack by enabling assertions in classes that have yet to be initialized. Moreover, applets can only affect the assert status of classes that are to be loaded by class loaders that the applets can access. There already exists a `RuntimePermission` to prevent untrusted code from gaining access to class loaders (`getClassLoader`).

7. Why not provide a construct to query the assert status of the containing class?

Such a construct would encourage people to inline complex assertion code, which we view as a bad thing. Further, it is straightforward to query the assert status atop the current API, if you feel you must:

```
boolean assertsEnabled = false;  
assert assertsEnabled = true; // Intentional side - effect!!!  
// Now assertsEnabled is set to the correct value
```

8. Why does an `assert` statement that executes before its class is initialized behave as if assertions were enabled in the class?

Few programmers are aware of the fact that a class's constructors and methods can run prior to its initialization. When this happens, it is quite likely that the class's invariants have not yet been established, which can cause serious and subtle bugs. Any assertion that executes in this state is likely to fail, alerting the programmer to the problem. Thus, it is generally helpful to the programmer to execute all assertions encountered while in this state.



[Copyright](#) © 2002 [Sun Microsystems, Inc.](#) All Rights Reserved.