**Java Coding Conventions and other Standards**

# Table of Contents

[1 Table of Contents 2](#_Toc207429497)

[1 Document History 6](#_Toc207429498)

[2 Introduction 7](#_Toc207429499)

[2.1 Spirit of Standards 7](#_Toc207429500)

[2.2 References 7](#_Toc207429501)

[3 General Standards 8](#_Toc207429502)

[3.1 Design by Contract 8](#_Toc207429503)

[3.2 Fault Detection and Recovery 8](#_Toc207429504)

[3.3 Self-Documentation 9](#_Toc207429505)

[3.4 Method Granularity 9](#_Toc207429506)

[3.5 Concurrency 9](#_Toc207429507)

[3.6 Resource Management 9](#_Toc207429508)

[3.7 Contract Requirements 9](#_Toc207429509)

[3.8 Packages 9](#_Toc207429510)

[3.9 Interfaces and Abstract Classes 10](#_Toc207429511)

[3.10 Visibility 10](#_Toc207429512)

[3.11 Final Classes and Methods 10](#_Toc207429513)

[3.12 Final Members 10](#_Toc207429514)

[3.13 Immutable Classes 10](#_Toc207429515)

[3.14 Nested Top-level Classes and Interfaces 11](#_Toc207429516)

[3.15 Static Class Members 11](#_Toc207429517)

[3.16 Constants 11](#_Toc207429518)

[3.17 JavaBeans 11](#_Toc207429519)

[4 Documentation 12](#_Toc207429520)

[4.1 Documentation Comments 12](#_Toc207429521)

[4.2 Banner Comments 14](#_Toc207429522)

[4.3 Other Comments 14](#_Toc207429523)

[5 Formatting 15](#_Toc207429524)

[Spaces 15](#_Toc207429525)

[Blank Lines 15](#_Toc207429526)

[5.1 Indentation 15](#_Toc207429527)

[5.2 Line Length 15](#_Toc207429528)

[5.3 Braces 16](#_Toc207429529)

[6 Naming 17](#_Toc207429530)

[6.1 Name Forms 17](#_Toc207429531)

[6.2 Naming Rules 17](#_Toc207429532)

[6.3 Classes 17](#_Toc207429533)

[6.4 Methods 17](#_Toc207429534)

[6.5 Fields 18](#_Toc207429535)

[7 Packages 19](#_Toc207429536)

[7.1 Naming 19](#_Toc207429537)

[8 Source Files 20](#_Toc207429538)

[8.1 Naming 20](#_Toc207429539)

[8.2 Organization 20](#_Toc207429540)

[9 Classes 22](#_Toc207429541)

[9.1 Class Documentation 22](#_Toc207429542)

[9.2 Class Organization 22](#_Toc207429543)

[9.3 Part of Class/Interface Declaration 22](#_Toc207429544)

[9.4 Notes 22](#_Toc207429545)

[9.5 Initialization 23](#_Toc207429546)

[9.6 java.lang.Object Methods 23](#_Toc207429547)

[9.7 java.io.Serializable 23](#_Toc207429548)

[9.8 Method Documentation 24](#_Toc207429549)

[9.9 Field Documentation 24](#_Toc207429550)

[9.10 Implementation of Preconditions and Postconditions 24](#_Toc207429551)

[9.11 Passing of References 24](#_Toc207429552)

[10 Interfaces 25](#_Toc207429553)

[10.1 Interface Documentation 25](#_Toc207429554)

[10.2 Method Documentation 25](#_Toc207429555)

[11 Code 26](#_Toc207429556)

[11.1 Comments 26](#_Toc207429557)

[11.2 Deficiencies 26](#_Toc207429558)

[11.3 Expressions 26](#_Toc207429559)

[11.4 Local Variables 26](#_Toc207429560)

[11.5 Local and Anonymous Classes 27](#_Toc207429561)

[11.6 Arrays 27](#_Toc207429562)

[11.7 Use of Exceptions 27](#_Toc207429563)

[11.8 Assertions 27](#_Toc207429564)

[11.9 Resource Management 28](#_Toc207429565)

[11.10 Testing Equality 28](#_Toc207429566)

[11.11 switch Statements 28](#_Toc207429567)

# Document History

|  |  |  |  |
| --- | --- | --- | --- |
| 08/21/2008 | 1.0 | Initial Draft |  |

# Introduction

This document specifies standards for the development of Java code. These standards are a tool that aids in the rapid development of high-quality software by promoting consistency, helping developers avoid common mistakes, making code easier to read and maintain, and focusing reviews on correctness rather than style

## Spirit of Standards

These standards are intended to define norms that help developers to efficiently deliver high-quality software. They may, at times, conflict with other goals of the development organization. Developers can and should make reasoned trade-offs against these standards when necessary.

If you need to violate a standard, document the reason clearly. If you are unsure about the application of a standard, document what you decided. This will make the job of understanding your code at a later time much easier. You should also be prepared to explain any such decision at a code review.

## References

References herein are not part of this specification unless otherwise specified.

Some general Java coding standards include Sun’s *Java Code Conventions* [<http://java.sun.com/docs/codeconv/>] and Doug Lea’s *Draft Java Coding Standard* [<http://g.oswego.edu/dl/html/javaCodingStd.html>].

# General Standards

## Design by Contract

One of the most important principles in the development of reliable software is *design by contract*. An introduction to design by contract is available at [<http://www.eiffel.com/doc/manuals/technology/contract/>].

Design by contract states that every software element has a particular job to do. It can either succeed or fail in that job – there is no in-between state. In order for the correctness of the element – its ability to succeed at its job – to be determinable, the job must be specified in terms of the responsibility of its clients and the obligations of the element.

Design by contract explicitly specifies that a software element does not have to handle cases where its clients have failed to meet their responsibilities. This is in direct contrast to *defensive programming*, which suggests that a software element should try to handle all cases. Defensive programming generally leads to overly complex, untestable software and often provides only inconsistent, useless handling of an unspecified subset of all possible cases.

In object-oriented software development, design by contract is best applied at the level of class and interface methods. Therefore, all methods must have a clearly specified contract. Unfortunately, most programming languages do not support the full design-by-contract methodology. Therefore the details of the contract requirements are language-specific.

## Fault Detection and Recovery

A *fault* is the run-time deviation of a software system from its specification. A fault is almost always the result of one or more *defects* in the software, which in turn result from one or more *errors* in design or coding.

Some faults simply produce erroneous results at the outputs of the system, but many faults avalanche into conditions detectable at run time. A fault may be detected because of run-time checking of contracts, or because it triggers a fault in a fundamental operation, such as a memory or array access. It is a goal of good software development to maximize the chances of detecting faults as close as possible to the defect that caused them.

Unfortunately, regardless of the degree of fault detection, by the time a fault is detected, the software system is generally in an indeterminate state. This makes recovery – putting the system back into a known and useful state – very difficult. Many software systems do not attempt recovery, but simply reflect the detected fault as a failure. This is almost always an invitation for further faults, farther removed from the original defect. A recovery strategy must take into account that arbitrary system state may be corrupted. The more global the recovery strategy, the more likely it is to succeed. In fact, the most reliable recovery strategy is to restart the software system. Of course, this may discard a great deal of previous work, but it is important to realize that any other strategy risks further cascading the effects of the original, probably undetected, fault.

Regardless of the handling of a detected fault, it is crucial that as much information as possible be collected and communicated to the developer responsible for finding and removing the defects. It is a goal of good software development to find and fix the defect or defects responsible for a detected fault from a single occurrence of that fault.

Therefore, all contract *preconditions* and *postconditions* should be tested at run-time wherever practical. Additionally, assumptions implied by the design or coding of a piece of software should be verified at run-time wherever practical through the use of *assertions*.

Each software system must have a documented strategy for handling detected faults and for collecting enough information to support a one-fault-to-fix philosophy.

## Self-Documentation

Each element of a software system should be sufficiently documented to make its use and maintenance reasonably obvious. If possible, all documentation should be kept with the element that is being documented. It is the developer’s responsibility to provide this documentation and to update it when necessary.

## Method Granularity

Each method on an interface or class should perform a single, coherent function. It is usually best to separate methods that perform actions from actions that query the state of the object.

## Concurrency

Every software element is implemented with some assumption about concurrency. Often this assumption is implicit. This leads to faults that are extremely difficult to reproduce and debug. Therefore, all software elements must have a documented, explicit concurrency context. This requirement applies to all classes, interfaces, and methods. The form of this documentation is language-specific.

## Resource Management

If a class or method allocates resources, it must ensure, either through language mechanisms or contracts, that the resources will be deallocated. For example, in Java, a finally-block is often used to deallocate resources such as database connections.

## Contract Requirements

Every method must have a contract that specifies the semantics of the method. The contract must be complete enough that clients of the method do not need to read the implementation code or comments. In Java, the contract comprises zero or more preconditions and postconditions, and a prose description.

## Packages

Each package should have some coherent purpose that serves as a guideline to what classes and subpackages it should contain.

## Interfaces and Abstract Classes

An interface should be defined whenever an aspect of a class can be abstracted into a form that could be implemented on another class. Interfaces should be defined instead of abstract classes unless significant implementation functionality is provided and the defined methods represent the primary functionality of the expected classes.

## Visibility

First, non-final fields should always be private. If such variables need to be accessed from other classes, even within the same package, accessor methods should be provided.

In a maximally-modular world, the behavior of a class would be defined solely by the Java interfaces it implements, and it would provide no other non-private members. However, this would eliminate the ability to reuse any code by inheritance and would force the designer to define Java interfaces for all inter-class interactions. Instead, we allow Java classes to define methods and make them available to various other classes.

Each of the following sets of methods is conceptually an interface, even if it is not defined as a Java interface.

* The set of public methods is the interface for unrelated classes in other packages.
* The set of public and protected methods is the interface for subclasses in other packages.
* The set of public and package-private and protected methods is the interface for all classes in the same package.

Each of these sets should be a coherent interface. The class documentation should discuss the purpose of each distinct set. Unless there is a clear reason, the class should have only public and private methods.

## Final Classes and Methods

The design of each class should take into account how it may be subclassed. If there is no clear purpose to subclassing, the class should be final. In cases where subclassing is permitted, allowing some methods to be overridden may not make sense. These methods should be final.

## Final Members

Class members should be declared final if their values should not change after construction.

## Immutable Classes

It is often useful to make a class immutable, as references to instances of the class can be replicated without the danger of changes to the object having inadvertent consequences. The objects are also inherently thread-safe. Immutable classes should be clearly labeled as such in the class comment, and should generally impose a condition that all subclasses must be immutable.

All member variables of an immutable class should be final.

The Object.clone method on an immutable object, if implemented, should simply return the same object.

## Nested Top-level Classes and Interfaces

Nested top-level classes and interfaces should be used for concepts that are very closely tied to the outer class, such as an iterator and a collection.

## Static Class Members

Non-final static fields should be avoided.

## Constants

Compile-time constants are declared in Java as static final fields.

Note that, in many cases, constants are not truly constant over the life of the code. These should be implemented as a method that returns the desired value, because changing a final field to either a non-final field or a method requires recompilation of every class that uses the value.

## JavaBeans

Use of the standard Bean naming conventions will permit access to the public methods and properties of a class as a Bean.

# Documentation

## Documentation Comments

All classes, interfaces, and members must have a documentation comment (“doc comment”). Doc comments express the contract that a class or interface meets. The use of doc comments keeps the documentation of this contract with the code that implements it and in a form that can be extracted.

The formatting of doc comments generally follows the JavaSoft standard [<http://java.sun.com/docs/codeconv/>].

It is the developer’s responsibility to provide complete doc comments, as they are the specification of the interface, class, or method’s contract. Without this contract, the software cannot be judged as to its correctness.

### Tab Characters

Horizontal tab characters should not be used in doc comments, as their rendering in text editors and browsers is inconsistent.

### HTML Elements

Only the following HTML elements may be used within doc comments: p, i, b, ul, ol, li, sub, sup, a, table, code and its associated subelements, and br.

### @deprecated

Deprecated classes, interfaces, or methods are marked with a @deprecated doc comment. See the JavaSoft guidelines on deprecation for a discussion of when to deprecate [<http://java.sun.com/j2se/1.4/docs/guide/misc/deprecation/deprecation.html>].

### Locally-defined Doc Comment Tags

This section describes several locally-defined doc comment tags adopted by Middleware development team.

#### @precondition

This doc comment is used on an interface or class method to document a precondition. The text of the doc comment should be a Java expression that is legal within the context of the method and must evaluate to true in order for an invocation of the method to be legal.

The precondition expression must not use any members whose visibility is less than that of the method being defined. For example, a precondition on a public method may only use other public methods in its precondition. This ensures that a client can use the precondition expression directly to know whether it would be an error to call this method.

It is acceptable to use a comment that describes the precondition in words in place of the expression. This form should only be used when the condition is extremely difficult to express as a Java expression. Since, in this case, a client cannot dynamically verify the condition, it must be a condition that the client can satisfy structurally. Put // before the comment to indicate that it is a comment. Java expressions or pseudo-expressions embedded within the comment should be put into code font, as in other Javadoc comments.

#### @postcondition

This doc comment is used on an interface or class method to document a postcondition. The text of the doc comment should be a Java expression that is legal within the context of the method – with one addition – and must evaluate to true when the method has completed (except through an exception). The one addition to legal Java syntax is that the keyword return may be used as an expression to denote the return value of the method.

When used within an interface, the postcondition expression must not use any members whose visibility is less than that of the method being defined. For example, a postcondition on a public method may only use other public methods in its postcondition. This restriction does not apply to postconditions on methods in classes.

It is acceptable to use a comment that describes the postcondition in words in place of the expression. This form should only be used when the condition is difficult to express as a Java expression. Put // before the comment to indicate that it is a comment. Java expressions or pseudo-expressions embedded within the comment should be put into code font, as in other Javadoc comments.

Here are two standard forms for comment style postconditions.

@postcondition return != null

@postcondition // count == old count + 1

The second postcondition indicates that the current value of count, after invocation of the method, must be one greater than the old value, before the method invocation.

### Tag Order

The order of tags in a method doc comment is: @param, @return, @throws, @precondition, @postcondition, @see, @since, @deprecated.

### Formatting

Add a blank line between each type of tag in a method doc comment. Do not add a blank line between @pre tags and @post tags. If the text of a tag wraps to another line, indent the wrapped line three spaces.

/\*\*

\* Creates a new configuration variable and adds it to this build job.  
 \*   
 \* @param name the variable's name  
 \* @param description the variable's description or <code>null</code> if

\* there is none  
 \* @param allowedValues the variable's allowed values or a zero-length   
 \* array if there are no allowed values  
 \*   
 \* @return the newly created <code>ConfigurationVariable</code>  
 \*   
 \* @precondition name != null  
 \* @precondition name.length() > 0  
 \* @precondition allowedValues != null  
 \* @postcondition // return != null  
 \*/

## Banner Comments

Sometimes it is useful to mark off part of a class declaration and provide comments about that part. Banner comment markers can be used to do this.

//----------------------------------------------------------------------------  
// The following methods are defined by the ServletRequest interface and are  
// implemented by delegating to the wrapped HttpServletRequest object.  
  
/\*\*  
 \* Returns the size of the request entity.  
 \*/  
public int getContentLength() {  
 return httpRequest.getContentLength();  
}  
  
/\*\*  
 \* Returns the Internet Media Type of the request entity data.  
 \*/  
public String getContentType() {  
 return httpRequest.getContentType();  
}  
  
//----------------------------------------------------------------------------

## Other Comments

Use C-style comments (/\* \*/) within a Java source file only to comment out lines of code that are not currently applicable, but which you want to keep.

Use the C++, single-line comment style (//) for all other comments within a Java source file. This reduces the possibility of accidentally commenting-out code. The double front-slash must always be followed by a space.

# Formatting

Consistent formatting improves the readability of code and allows the reader to quickly focus on the content of the code and not its form.

## Spaces

Spaces are used to improve readability of code. Except at the beginning or end of a line, a single space should be used

* between a language keyword, such as if or while, and an opening parenthesis,
* after each comma or semi-colon,
* before and after every binary operator except “.”.

No space should be used

* between a method name and the opening parenthesis of the argument list,
* after an opening parenthesis or before the closing parenthesis,
* between an unary operator, such as ++ or cast, and its operand, or
* before a comma or semi-colon.

## Blank Lines

Blank lines improve readability of code by delimiting sections of code. In general, one blank line should be used

* between methods,
* before a block or single-line comment, and
* between logical sections within a method.

## Indentation

Each lexical scope is indented one additional level, where each level is four spaces.

Horizontal tab characters should not be used in code, as their rendering in text editors is inconsistent.

Within a switch statement, the case labels are indented one level and the code blocks are indented an additional level, even if braces are not used.

## Line Length

A Java source file should not contain lines longer than 100 characters.

In general, each simple statement should appear on a line by itself. When a statement will not fit on a line, the statement should be split after a comma or an operator. In general, the second and subsequent lines of a single statement should be indented so that the items separated by the comma or operator line up vertically. If, however, this results in hard-to-read code or code that would overrun the right margin, it is acceptable to indent subsequent lines by one indent level.

## Braces

Braces must always be used around a statement block that is part of another statement, such as an if, else, or while statement, unless the entire statement is kept on a single line. This avoids an error when an additional statement is added to the statement block.

Braces should not be used after a case label to emphasize the fact that fall-through occurs unless a break statement is included. If braces are needed to create a local scope, the break statement must be outside this scope.

If the statement introducing a statement block fits on a single line, then the opening brace should be placed at the end of that line to avoid excessive vertical whitespace. If the statement introducing a statement block takes up multiple lines, then the opening brace should be placed on a line by itself to make the code easier to read.

A closing brace is always placed on a line by itself, indented the same amount as the statement where the opening brace occurs.

// The starting brace is at the end of the line containing the introductory  
// statement, because the introductory statement fits on a single line.  
if (numberOfErrors > 10) {  
 ...

}

// The starting brace is on a line by itself, because the statement

// introducing the method block takes up more than one line.  
  
public static someMethodWithALongName(ParameterType parameter1,   
 ParameterType parameter2)  
 throws SomeException, SomeOtherException, SomeThirdException  
{  
 ...

}

# Naming

## Name Forms

One of three name forms is used for all identifiers:

|  |  |  |  |
| --- | --- | --- | --- |
| **Form** | **Rules** | **Example** | **Usage** |
| Lower-case name | Begins with lower-case letter. Subsequent words have single initial capital. No underscores. | numberOfItems | local variables, formal parameters,  methods, |
| Upper-case name | Begins with upper-case letter. Subsequent words have single initial capital. No underscores. | SentData | classes, interfaces |
| All upper-case name | All upper-case characters. Words are separated by an underscore. | WEIGHT\_VERY\_HEAVY | constants |
| Lower-case name | Begins with and follows the Lower-case name rule | patientName | member variables |

## Naming Rules

Boolean identifiers should generally begin with one of the words “is” or “has”.

When an acronym or abbreviation is used in a mixed-case identifier, the second and subsequent letters should never be capitalized, even if they would be in normal usage. For example, use isUsaId instead of isUSAID.

Random abbreviations, such as “num” for number, should be avoided. If acceptable abbreviations are adopted in the future, they will be defined in a separate document.

## Classes

Classes are named using the upper-case name form. Classes should be given meaningful names, usually a noun phrase, such as DatabaseHelper. Classes that represent a collection of objects should have a plural name that identifies the collected objects, such as Utilities.

Classes that represent an exception – those that inherit from Exception – should be named by a noun phrase ending with “Exception”. No other classes should be named “Exception”.

Inner classes are named using the same rules as outer classes.

## Methods

Methods are named using the lower-case name form. Methods should be given meaningful names, usually a verb phrase, such as printData. Accessors for non-boolean properties should be named get*X* andset*X*. Accessors for boolean properties should be named is*X* andset*X*.

int getSize() { … }

void setSize(int newSize) { … }

boolean isReady() { … }

void setReady(boolean makeReady) { … }

## Fields

Most fields are named using the lower-case name form prepended. Member variables should be given meaningful names.

Static final fields should be named using the all-upper-case name form

# Packages

## Naming

Packages are named with a sequence of single-word, all-lower-case identifiers, separated by the period character.

# Source Files

## Naming

The Java compilation system requires that the name of a source file be the name of the public class or interface defined therein plus a “.java” suffix.

## Organization

Each Java source file contains the following elements in the following order.

### Header Block

Every Java source file begins with the following header block, which contains a line-length ruler, file version identifier, and a copyright notice.

/\*

\* The information in this document is subject to change without notice and does

\* not represent a commitment by Headstrong Corporation. The software and/or

\* databases described in this document are furnished under a license agreement and

\* may be used or copied only in accordance with the terms of the agreement.

\*

\* Copyright © 2008 Headstrong Corporation

\* All rights reserved.

\*

\* $Id: $

\* $Revision: $

\* $Author: $

\* $Change: $

\* $DateTime: $

\*/

You must augment the copyright notice to include the current year whenever you modify a source file.

### Imports

Any needed import statements follow the package statement. Import only those classes or packages needed. Do not copy a huge list from another source file. Using wildcard (\*) notation to import all of the public classes in a package should be avoided.

Do not import the java.lang package, as it is automatically imported.

### Class or Interface Definition

A single class or interface definition follows the imports. Only one class or interface definition is permitted per file. Any associated private classes or interfaces should be nested top-level classes or interfaces rather than separate private classes or interfaces.

# Classes

## Class Documentation

Every class must be preceded by a documentation comment that describes the purpose and general usage of the class and its relationship to other classes, including non-private inner classes.

A class may contain a general comment describing implementation as the first item within its declaration.

## Class Organization

The class definition is organized to simplify maintenance. Clients of the class should use the javadoc-generated documentation for reference, not the class source code.

The following table describes the parts of a class or interface declaration, in the order that they should appear.

|  |  |  |
| --- | --- | --- |
|  | Part of Class/Interface Declaration | Notes |
| 1 | Class/interface documentation comment (/\*\*...\*/) |  |
| 2 | Class or interface statement |  |
| 3 | Class/interface implementation comment (/\*...\*/) | This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment. |
| 4 | Class (static) variables | First the public class variables, then the protected, then package level (no access modifier), and then the private. |
| 5 | Instance variables | First public, then protected, then package level (no access modifier), and then private. |
| 6 | Constructors |  |
| 7 | Methods | These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier. |

|  |
| --- |
|  |

The serialization identifier is an exception to this rule and is always placed at the bottom of the class definition.

## Initialization

All fields needing non-default initialization should be initialized in their declaration statement if possible. Otherwise, each static and non-static fields needing non-default initialization should be initialized in an initializer immediately following the field declaration if possible. The use of initializers is preferred to initialization in a constructor because there can be multiple constructors.

## java.lang.Object Methods

The designer of a class should consider the appropriate semantics for the equals method.

Classes that might be used as a key to a hash table should override Object.hashCode.

## java.io.Serializable

Any class that implements java.io.Serializable must define a static, final member variable serialVersionUID to define the serialization identifier. The value of this identifier should not change over the lifetime of the class.

// Provides the serialization identifier.

static final long serialVersionUID = -5663010379579870382L;

The serialver utility from the Sun Java SDK can be used to generate a serialization identifier value.

Serializable classes are restricted to certain evolutionary changes:

* adding fields
* adding classes to the inheritance hierarchy
* removing classes from the inheritance hierarchy
* adding writeObject/readObject methods that first call the default serialization methods defaultReadObject/defaultWriteObject
* removing writeObject/readObject methods that had called the default serialization methods defaultReadObject/defaultWriteObject
* adding java.io.Serializable - This is equivalent to adding types. There will be no values in the stream for this class so its fields will be initialized to default values. The support for subclassing nonserializable classes requires that the class's supertype have a no-arg constructor and the class itself will be initialized to default values. If the no-arg constructor is not available, the NotSerializableException is thrown.
* removing java.io.Serializable from a superclass that has a no-arg constructor
* changing the access to a field
* changing a field from static to nonstatic or transient to nontransient
* adding or removing methods.

## Method Documentation

Every method must be, preceded by a documentation comment that describes the purpose and usage of the method. This comment should contain enough information, including examples if necessary, for a developer to use the method without reference to the source code. The comment for a non-final method should describe the appropriate conditions for overriding the method in a subclass.

A method may contain a general comment describing its implementation as the first item within its declaration.

## Field Documentation

Fields should almost never have public or package visibility. Each field should, therefore, have a simple comment that describes its purpose.

## Implementation of Preconditions and Postconditions

Each precondition and postcondition specified for a method in non-comment form must be implemented as a Java assert. Do not check multiple preconditions or postconditions in a single assert statement, as that makes debugging more difficult when a condition fails.

Since precondition and postcondition assertions are a verification mechanism, and could theoretically be removed without affecting the semantics of the method, the assert statements must have no side effects.

## Passing of References

Methods that take references to mutable objects as parameters or that return such object references must be clearly documented as to the semantics of those references. If the class maintains a reference to the same object, changes made to the object outside the class will affect future use of the object within the class. In many cases, the object should be cloned or copied.

# Interfaces

## Interface Documentation

Every interface must be preceded by a documentation comment that describes the purpose and general usage of the interface and its relationship to other interfaces and classes. This comment should contain enough information for a developer to determine the applicability of the interface.

## Method Documentation

Every method must be preceded by a documentation comment that describes the purpose and usage of the method. This comment should contain enough information, including examples if necessary, for a developer to use the method without reference to the source code.

# Code

## Comments

Source code is read many more times than it is written, so it is vital that it be readable and comprehensible. Code should be commented liberally as to purpose and method.

## Deficiencies

If code is unfinished or needs to be updated in some way in the future, use TODO with a comment to describe, in detail, how the code needs to be changed. If there is an issue in the issue tracking system that corresponds to the unfinished code, list its issue ID in parentheses directly after the TODO. For example,

// TODO (BUG 1898) When we upgrade to version 2.1 of the Java Servlet   
// API, the code that follows should be changed to use the   
// log(Throwable) method.

Note that if a method does not meet its contract due to unfinished code, then in addition to having a TODO comment in the code, there should be a note in the method's documentation comment.

## Expressions

Make expressions as simple as possible. When possible, split a complex expression into multiple steps, or use parentheses to make the expression more readable. Avoid expressions with embedded side effects – make those into separate statements (e.g., numberOfItems++;). Do not embed an assignment in an expression. Do not cascade assignments in a single statement.

## Local Variables

Local variables are named using the lower-case name form. In general, local variables should be given meaningful names. However, it is acceptable to use i, j, or k to name an iterator declared in the initialization part of a for statement, and e to name an exception in a catch clause. Do not give a local variable a name that hides another variable.

Local variables should be declared as close as practical to their first use. Each one should be declared in a separate statement, and, unless it is obvious, each declaration should have a comment describing the purpose of the variable. If possible, a local variable should be initialized in the declaration statement.

Do not use a local variable for more than one purpose within a method.

## Local and Anonymous Classes

Local classes may be used only where a class is needed only in a single method and the class can be defined in a few simple lines of code. An anonymous class should be used if the local class would be used only in a single place.

A more significant class should be declared as a private nested top-level class.

## Arrays

Use the notation *Type*[] *variable\_name* when specifying an array type.

public String[] getWarningMessages() {  
 int[] messageCodes;  
 ...  
}

## Use of Exceptions

Java provides two types of exceptions: checked and unchecked. Checked exceptions, which are subtypes of java.lang.Exception, but not java.lang.RuntimeException, must be declared in the signature of a method. Unchecked exceptions, which are subtypes of java.lang.Error or java.lang.RuntimeException, do not need to be declared.

A checked exception is always used to indicate the anticipated failure of a method to meet its contract. It is appropriate to catch such an exception if, and only if, the catching code has a recovery action so that it can meet its own contract.

An unchecked exception always indicates a fault, and therefore a defect in the code. For this reason, it is almost never allowable to catch an unchecked exception without re-throwing it. Only if a complete software fault-tolerance strategy, including appropriate fault logging and tracking, is implemented may an unchecked exception be caught and handled. Therefore, do not specify java.lang.Throwable, java.lang.Error, any subtype of java.lang.Error, java.lang.RuntimeException, or any subtype of java.lang.RuntimeException in a catch clause unless the clause re-throws the exception.

## Assertions

There are often many implicit assumptions made in a piece of code, such as “this method call will always return true in this case”. These assumptions should always be documented, but, if possible, they should be made explicit in a form that can be checked at run time. An *assertion* is a boolean expression that must be true in order for the containing method to continue to execute correctly. The failure of an assertion is always a fault.

An assertion is implemented by a Java assert statement.

assert numberOfElements > 0;

The assertion expression must have no side effects.

## Resource Management

Any method that acquires a non-object resource, such as a lock, must make provision to release it in the case of any exception. This is the only case where catching an unchecked exception is allowed. In this case, the exception must be re-thrown once the resource is released.

## Testing Equality

Use Object.equals to test object equality; use == only for primitive types.

## switch Statements

Every switch statement must have a default case label. If the default case should never be reached, it should contain the statement “assert false”.

Fall-through from one block of code into another in a switch statement is strongly discouraged, but may be done if a //FALLTHROUGH comment is placed at the point of the missing break statement.

switch (choice) {  
 case FIRST\_CHOICE:  
 case SECOND\_CHOICE: // No FALLTHROUGH needed.  
 // Do nothing in either case.  
 break;  
 case THIRD\_CHOICE:  
 doSomething();  
 //FALLTHROUGH // Needed here. But better to rewrite to avoid.  
 case FOURTH\_CHOICE:   
 doSomethingElse();  
 break;  
 default:  
 assert false;   
 break; // Always use break.  
}