

Moody's Investors Service

New York

**Java Coding Standards & Guidelines**

****

Revision History

| Date | Version | Description | Author |
| --- | --- | --- | --- |
| 10-Oct-2012 | 1.0 | Initial Version | Tata consultancy Services |
| 13-Jun-2013 | 1.1 | Incorporated Prantik’s Feedback   1. Added the section 3.11.2 – Single Concept per Test 2. Changed the line length limit from 80 to 120 in section 4.4.3 3. Added sections ‘5.3.12 - Do not write Redundant Comments’ and ‘5.3.13 - Dont comment out code, delete it’ 4. Added the reference ‘Clean Code A Handbook of Agile Software Crafts’ under section 7 | Tata consultancy Services |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1 Introduction 5](#_Toc358939654)

[1.1 Purpose 5](#_Toc358939655)

[2 Overview 6](#_Toc358939656)

[3 Java naming standards 7](#_Toc358939657)

[3.1 Naming Java applications and libraries 7](#_Toc358939658)

[3.2 Identifiers naming standards 8](#_Toc358939659)

[3.3 File naming standards 9](#_Toc358939660)

[3.4 Types naming standards 9](#_Toc358939661)

[3.5 Class naming standards 9](#_Toc358939662)

[3.6 Interface naming standards 9](#_Toc358939663)

[3.7 Method naming standards 10](#_Toc358939664)

[3.8 Variables naming standards 11](#_Toc358939665)

[3.9 Constants naming standards 13](#_Toc358939666)

[3.10 Exception naming standards 14](#_Toc358939667)

[3.11 Unit test class naming standards 17](#_Toc358939668)

[3.12 Package naming standards 18](#_Toc358939669)

[4 Java layout standards 23](#_Toc358939670)

[4.1 Java source files layout standards 23](#_Toc358939671)

[4.2 Java types layout standards 24](#_Toc358939672)

[4.3 Java statements layout standards 26](#_Toc358939673)

[4.4 Other Java layout standards 29](#_Toc358939674)

[5 Java commenting standards 32](#_Toc358939675)

[5.1 Doc comments 32](#_Toc358939676)

[5.2 Comment types 32](#_Toc358939677)

[5.3 Usage guidelines 37](#_Toc358939678)

[6 Java coding guidelines 40](#_Toc358939679)

[6.1 Best practices 40](#_Toc358939680)

[6.2 Make exceptions informative 45](#_Toc358939681)

[6.3 Consider the ordering of object comparisons 46](#_Toc358939682)

[6.4 Use short-circuit evaluation 46](#_Toc358939683)

[7 References 47](#_Toc358939684)

[8 Approvals 48](#_Toc358939685)

# Introduction

This document provides an outline of Java standards and guidelines that need to be followed to improve readability and maintainability of the source code and also to ensure the consistency within and across the projects.

## Purpose

This document lists recommendations for Java development.

This document will require periodic reviews and updates from design and development teams, to incorporate new standards or update guidelines without losing the objective of consistency. Contribution to improve the content of this document is very much appreciated.

Audience

The audience of these standards are the professional software developers to

* Develop Java applications that are easy to maintain.
* Minimize the review and rework effort.
* Increase the individual and team productivity.

# Overview

Why have standards anyway?

Standards are important for a number of reasons:

* 80% of the lifetime cost of a piece of software goes to maintenance.
* Hardly any software is maintained for its whole life by the original author.
* Coding standards improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
* Standards reduce training management and effort.

There are millions of ways to implement standards.

The question always is: What is the right way?

|  |
| --- |
| The rule to follow when developing Java code |
| If the people working on the project strongly believe in a certain convention, it can be used, but always use it consistently throughout the project. |

The recommendations in this document are based on established standards collected from a number of sources as listed in the References section.

This document mainly focuses on detailing out below java standards

1. Naming standards – naming conventions to be followed in java projects
2. Layout or formatting standards – settings to be used while formatting the source code
3. Commenting standards – various types of comments to be added to the source code
4. Coding guidelines – some of the best practices that are being followed by java community

# Java naming standards

We will be discussing naming conventions throughout this document, so let’s set the stage with a few basics:

1. **Use full English descriptors that accurately describe the variable, field or class.** For example, use names like firstName, grandTotal. Although names like x1, y1, or fn are easy to type because they’re short, they do not provide any indication of what they represent and result in code that is difficult to understand, maintain, and enhance.
2. **Use terminology applicable to the domain.** If your users refer to their clients as customers, then use the term Customer for the class, not Client. Many developers will make the mistake of creating generic terms for concepts when perfectly good terms already exist in the industry/domain.
3. **Use mixed case to make names readable.** You should use lower case letters in general, but capitalize the first letter of class names and interface names, as well as the first letter of any non-initial word.
4. **Use abbreviations sparingly, but if you do so then use them intelligently.** This means you should maintain a list of standard short forms (abbreviations), you should choose them wisely, and you should use them consistently. For example, if you want to use a short form for the word number, then choose no.
5. **Avoid long names (< 15 characters is a good idea).** Although the class name PhysicalOrVirtualProductOrService might seem to be a good class name at the time, this name is simply too long and you should consider renaming it to something shorter, perhaps something like Offering.
6. **Avoid names that are similar or differ only in case.** For example, the variable names persistentObject and persistentObjects should not be used together, nor should anSqlDatabase and anSQLDatabase.

Note: Rules defined further on in this document explain how to choose between potential variants of a name in specific usage scenarios.

## Naming Java applications and libraries

### Use business requirements and organization’s business terms as the reference

When naming Java applications or libraries, always use business requirements and organization’s business terms as the reference.

Source: Sun Standards

### Java application and library names use UpperCamelCase

By default, the Java application or library name should be indicative of the application and should be written in UpperCamelCase letters. In case of conflict, the name should be a more specific and descriptive name.

Source: Sun Standards

### The name must not contain versioning information

The name does not have to contain versioning information, because the software versioning system that you use (for instance CVS or Subversion) will take care of the version and you will know on which version (branch or trunk) you are working.

|  |
| --- |
| Application or library name: SmartFormsFramework |

Source: Sun Standards

### The packaged distribution must contain versioning information

Java applications and libraries are normally packaged in jar files and it is important that the packaged jar file contain the application or library name as well as the version. The version must be added to the application name in the format:

–<major release>.<minor release>.<fixpack number>

|  |
| --- |
| Packaged name: SmartFormsFramework-1.0.0.jar |

Source: Sun Standards

## Identifiers naming standards

These naming standards apply to all identifiers like class names, variables, etc.

### Compound words use CamelCase

Identifiers in Java made up of more than one word should use the CamelCase naming convention.

Source: Sun Standards

### Do not use the underscore to delimit words.

With HTML-based documentation and automatic documentation generators like doxygen and javadoc, method and class names are often hyperlinked. On one page of documentation if you click on a class name, it jumps you to the page describing that class. Since the html convention is to underline links, any underscores in class or method names tend to get lost. For example, consider get\_radius (with an underscore separator) and get radius (with no underscore). They look the same. The underscore has been lost. So, to avoid the lost underscore problem with html-based documentation, we recommend that underscores not be used in names.

Source: Sun Standards

### Do not abbreviate

Use whole words - avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as url or html).

Source: Sun Standards

### Abbreviations and acronyms are not in uppercase

Do not express acronyms or abbreviations in uppercase in identifier names.

Using all uppercase for the base name will give conflicts with other naming conventions. In the example below, a variable of this type would have to be named dVD, hTML etc. which obviously is not very readable. Another problem is that when the name is connected to another, the readability is seriously reduced. The word following the acronym does not stand out, as it should.

|  |
| --- |
| exportHtmlSource(); // NOT: exportHTMLSource();  openDvdPlayer(); // NOT: openDVDPlayer(); |

Source: GeoSoft

### Identifier names are English

All identifier names should be written in English.

English is the preferred language for international development. Constraining names to a common language will avoid translation issues in a multi-cultural environment. It will further reduce the risk of ambiguous names that may be expressed the same but have different meanings in different languages.

Source: GeoSoft

## File naming standards

### File names should have the same name as the primary type

A java source code file name should match the name of the primary contained class or interface in both name and case. Use the file extension .java

|  |
| --- |
| MyClass.java //encapsualtes MyClass |

Source: Sun Standards

## Types naming standards

### Type names are nouns written in UpperCamelCase

Names of types (classes and interfaces) should be nouns written in UpperCamelCase. Try to keep your class names simple and descriptive.

This practise avoids type and package name conflicts.

Source: Sun Standards

## Class naming standards

### Class names are nouns written in UpperCamelCase

Names of classes should be nouns written in UpperCamelCase. Try to keep your class names simple and descriptive.

Use whole words and avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as url or html).

|  |
| --- |
| class Intermediary; |

Source: Sun Standards

## Interface naming standards

### Interface names should be a descriptive adjective written in UpperCamelCase

Java interface names should be written in UpperCamelCase.

Interface names should be a descriptive adjective, such as Runnable or Cloneable, although descriptive nouns, such as Singleton and DataInput, are also common.

Only one interface should be defined per file.

|  |
| --- |
| interface Storeable; |

Source: Sun Standards

## Method naming standards

### Methods are verbs written in lowerCamelCase

Methods should be verbs and written using the lowerCamelCase naming convention.

This is identical to variable names, but methods in Java are already distinguishable from variables by their specific form. This also follows the convention that internal elements of a class use lowerCamelCase.

|  |
| --- |
| run();  runFast();  getBackground(); |

Source: Sun Standards

### Avoid repeating object name in the method name

In a method name the name of the object is implicit and should be avoided in the method name.

|  |
| --- |
| line.getLength(); //NOT: line.getLineLength(); |

### Use getters and setters as field accessors

Prefix methods that expose field values with get.

Prefix methods that set field values with set.

By following this naming convention you make it obvious that you are returning or setting a field of an object. This also complies with the Java bean accessor syntax.

|  |
| --- |
| getFirstName();  setFirstName(String aName); |

### Use the is prefix for boolean methods

All boolean getters must be prefixed with the word is.

This is the naming convention for boolean methods and variables used by Sun for the Java core packages. When writing Java beans this convention is actually enforced for methods.

|  |
| --- |
| isPersistent()  isString()  isCharacter() |

There are a few alternatives to the is prefix that fit better in some situations. These are has, can and should prefixes. Use them judiciously.

|  |
| --- |
| hasLicense()  canEvaluate()  shouldAbort() |

Source: GeoSoft, Ambler

### The term compute can be used in methods where something is computed

This gives the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he/she might consider caching the result. Consistent use of the term enhances readability.

|  |
| --- |
| valueSet.computeAverage();  matrix.computeInverse(); |

Source: GeoSoft

### The term find can be used in methods where something is looked up

This gives the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability.

|  |
| --- |
| vertex.findNearestVertex();  matrix.findMinElement(); |

Source: GeoSoft

### The term initialize can be used where an object or concept is established

The American initialize should be preferred over the English initialise.

The abbreviation initmust be avoided.

|  |
| --- |
| printer.initializeFontSet(); |

Source: GeoSoft

### Default interface implementations can be prefixed by Default

It is not uncommon to create a simplistic class implementation of an interface providing default behaviour to the interface methods. The convention of prefixing these classes by Default has been adopted by Sun for the Java library.

|  |
| --- |
| class DefaultTableCellRenderer  implements TableCellRenderer {  :  } |

Source: GeoSoft

### Functions (methods returning an object) should be named after what they return and procedures (void methods) after what they do

This increases readability. It makes it clear what the unit should do and especially all the things it is *not* supposed to do. This makes it easier to keep the code clean of side effects.

Source: GeoSoft

## Variables naming standards

### Variable names are in lowerCamelCase

Variables should use the lowerCamelCase naming convention. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Variable names should not start with underscore \_ or dollar sign $ characters, even though both are allowed.

This makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration Line line;

|  |
| --- |
| Line line;  float myWidth; |

Source: Sun Standards

### No suffix should be used for variables representing an entity number

Variables representing an entity number should be suffixed with No.

The notation is taken from mathematics where it is an established convention for indicating an entity number.

|  |
| --- |
| tableNo, employeeNo |

Source: GeoSoft

### Avoid prefixing instance variable or member names

Avoid using the C++ convention of prefixing instance or member variables with m\_ (for example, m\_name). It is ugly and inconsistent with other Java conventions (underscores are normally only used in constants in Java).

### Variables with a large scope should have long names, variables with a small scope can have short names

Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are i, j, k, m, n and for characters c and d.

Iterator variables should be called i, j, k, etc.

|  |
| --- |
| while (Iterator i = points.iterator(); i.hasNext(); ) {  :  }  for (int i = 0; i < nTables; i++) {  :  } |

The notation is taken from mathematics where it is an established convention for indicating iterators.

Variables named j, k etc. should be used for nested loops only.

Source: GeoSoft

### Collection names are expressed in plural form

Plural form should be used on names representing a collection of objects. This enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on the object.

|  |
| --- |
| Collection points; // of Point  int[] values; |

Source: GeoSoft

### Avoid negated boolean variable names

Do not use negated expressions in boolean variable names. The problem arise when the logical not operator is used and double negative arises. It is not immediately apparent what !isNotError means.

|  |
| --- |
| boolean isError; //NOT: isNotError  boolean isFound; //NOT: isNotFound |

Source: GeoSoft

### Never create a temporary variable named temp

Instead use a name that indicates what the variable is needed for.

### Arrays

Use the following style to declare an array:

|  |
| --- |
| int[] integerArray = { 2, 4, 6, 8 }; // Use this  int integerArray[] = { 1, 2, 3, 4, 5 }; // NOT this |

### Parameter names should avoid, where possible, having the type implied

One should imply some sort of role in the name of the parameter e.g. newCustomer. This removes ambiguities when there is more than one parameter of the same type e.g. newCustomer, existingCustomer rather than aCustomer1, aCustomer2 etc.

### JFC (Java Swing) variables should be suffixed by the element type

This enhances readability since the name gives the user an immediate clue of the type of the variable and thereby the available resources of the object.

|  |
| --- |
| widthScale  nameTextField  leftScrollbar  mainPanel  fileToggle  minLabel  printerDialog |

Source: GeoSoft

## Constants naming standards

### Constants are all upper case

The names of variables declared class constants should be all uppercase with words separated by underscores ("\_").

|  |
| --- |
| static final int MIN\_WIDTH = 4;  static final int MAX\_WIDTH = 999;  static final int GET\_THE\_CPU = 1; |

Source: Sun Standards

### Associated constants should be prefixed by a common type name

This indicates that the constants belong together and what concept the constants represents.

|  |
| --- |
| final int COLOR\_RED = 1;  final int COLOR\_GREEN = 2;  final int COLOR\_BLUE = 3; |

An alternative to this approach is to put the constants inside an interface effectively prefixing their names with the name of the interface.

|  |
| --- |
| interface Color {  final int RED = 1;  final int GREEN = 2;  final int BLUE = 3;  } |

Source: GeoSoft

With newer version of JDK, it is recommended that Enum feature is used instead of interface features with the same naming standards as of using an interface.

## Exception naming standards

### Exception class names are suffixed with Exception

All exception class names should follow the class naming convention and should end in the word Exception.

Exception classes with names ending in Exception are runtime problems that may be dealt with by the application. A FileNotFoundException, for instance, simply indicates that a requested file doesn't exist yet.

We recommend that new classes for normal exceptions use names ending with Exception.

|  |
| --- |
| class MyCustomApplicationException extends Exception; |

### Error class names are suffixed with Error

All error class names should follow the class naming convention and should end in the word Error.

Exception classes with names ending in Error indicate very serious problems that are normally fatal. An OutOfMemoryError, for instance, is usually not recoverable.

We recommend that new classes for fatal exceptions use names ending with Error.

|  |
| --- |
| class MyFatalError extends Error; |

### Exception names should conform to existing Java examples

Since exceptions are also classes, their class names should follow the class naming conventions. We recommend that new exception classes use names that follow the Java class naming conventions.

|  |
| --- |
| Example Java exception class names from Sun’s Java distribution |
| Object  |  +---->Throwable  |  +---->Error  | |  | +---->LinkageError  | | |  | | +---->ClassCircularityError  | | |  | | +---->ClassFormatError  | | | |  | | | +---->UnsupportedClassVersionError  | | |  | | +---->ExceptionInInitializerError  | | |  | | +---->IncompatibleClassChangeError  | | | |  | | | +---->AbstractMethodError  | | | |  | | | +---->IllegalAccessError  | | | |  | | | +---->InstantiationError  | | | |  | | | +---->NoSuchFieldError  | | | |  | | | +---->NoSuchMethodError  | | |  | | +---->NoClassDefFoundError  | | |  | | +---->UnsatisfiedLinkError  | | |  | | +---->VerifyError  | |  | +---->ThreadDeath  | |  | +---->VirtualMachineError  | |  | +---->InternalError  | |  | +---->OutOfMemoryError  | |  | +---->StackOverflowError  | |  | +---->UnknownError  |  +---->Exception  |  +---->ClassNotFoundException  |  +---->CloneNotSupportedException  |  +---->IllegalAccessException  |  +---->InstantiationException  |  +---->InterruptedException  |  +---->IOException  | |  | +---->CharConversionException  | |  | +---->EOFException  | |  | +---->FileNotFoundException  | |  | +---->InterruptedIOException  | |  | +---->ObjectStreamException  | | |  | | +---->InvalidClassException  | | |  | | +---->InvalidObjectException  | | |  | | +---->NotActiveException  | | |  | | +---->NotSerializableException  | | |  | | +---->OptionalDataException  | | |  | | +---->StreamCorruptedException  | | |  | | +---->WriteAbortedException  | |  | +---->SyncFailedException  | |  | +---->UnsupportedEncodingException  | |  | +---->UTFDataFormatException  |  +---->NoSuchFieldException  |  +---->NoSuchMethodException  |  +---->RuntimeException  | |  | +---->ArithmeticException  | |  | +---->ArrayStoreException  | |  | +---->ClassCastException  | |  | +---->ConcurrentModificationException  | |  | +---->EmptyStackException  | |  | +---->IllegalArgumentException  | | |  | | +---->IllegalThreadStateException  | | |  | | +---->NumberFormatException  | |  | +---->IllegalMonitorStateException  | |  | +---->IllegalStateException  | |  | +---->IndexOutOfBoundsException  | | |  | | +---->ArrayIndexOutOfBoundsException  | | |  | | +---->StringIndexOutOfBoundsException  | |  | +---->MissingResourceException  | |  | +---->NegativeArraySizeException  | |  | +---->NoSuchElementException  | |  | +---->NullPointerException  | |  | +---->SecurityException  | |  | +---->UnsupportedOperationException  |  +---->TooManyListenersException |

Source: Sun Standards

## Unit test class naming standards

### Unit test class names are suffixed with Test

All unit test class names should follow the naming convention for classes, have the same base name as the target class being tested and should end in the word Test.

|  |
| --- |
| class ClientBeanTest; //Tests the class ClientBean |

### Single Concept per Test

We don’t want long test functions that go testing one miscellaneous thing after another. The best rule would be to minimize the number of asserts per concept and test only one concept per test function.

Source: Clean Code A Handbook of Agile Software Craftsmanship

## Package naming standards

### Package names are in lower case

All components of a unique package name should be written in all-lowercase ASCII letters.

Because Java uses the same dot syntax for packages and classes, if a package name and a class name are the same, the compiler can't figure out what to do. There is no syntactic workaround. Thus, packages and classes have different naming conventions.

**Exception:** In the Java distribution, the only exception to this naming convention is javax.rmi.CORBA, which introduces uppercase letters because of CORBA's own naming conventions.

|  |
| --- |
| package java.io;  package java.lang;  package com.moodys.sff.util; |

Source: Sun Standards

### Identifiers are separated by periods

To make package names more readable, Sun suggests that the identifiers in package names be separated by periods. For example, the package name java.awt is comprised of two identifiers, java and awt.

Source: Ambler

### Package names are singular

The common convention is to use singular names for packages.

|  |
| --- |
| package interface.screen; //NOT: interface.screens; |

Source: Ambler

### Package names should only contain alphabetic characters

Only use alphabetic characters in package names.

|  |
| --- |
| package package.example.first; //NOT: package.example.1; |

Source: Sun Standards

### Package names should conform to existing Java examples

Package names should conform to existing Java examples when appropriate. e.g., for I/O classes, make an io package similar to java.io.

|  |
| --- |
| Example Java package names from Sun’s Java distribution |
| java.applet  java.awt  java.awt.color  java.awt.datatransfer  java.awt.dnd  java.awt.event  java.awt.font  java.awt.geom  java.awt.im  java.awt.im.spi  java.awt.image  java.awt.image.renderable  java.awt.print  java.beans  java.beans.beancontext  java.io  java.lang  java.lang.annotation  java.lang.instrument  java.lang.management  java.lang.ref  java.lang.reflect  java.math  java.net  java.nio  java.nio.channels  java.nio.channels.spi  java.nio.charset  java.nio.charset.spi  java.rmi  java.rmi.activation  java.rmi.dgc  java.rmi.registry  java.rmi.server  java.security  java.security.acl  java.security.cert  java.security.interfaces  java.security.spec  java.sql  java.text  java.util  java.util.concurrent  java.util.concurrent.atomic  java.util.concurrent.locks  java.util.jar  java.util.logging  java.util.prefs  java.util.regex  java.util.zip  javax.accessibility  javax.activity  javax.crypto  javax.crypto.interfaces  javax.crypto.spec  javax.imageio  javax.imageio.event  javax.imageio.metadata  javax.imageio.plugins.bmp  javax.imageio.plugins.jpeg  javax.imageio.spi  javax.imageio.stream  javax.management  javax.management.loading  javax.management.modelmbean  javax.management.monitor  javax.management.openmbean  javax.management.relation  javax.management.remote  javax.management.remote.rmi  javax.management.timer  javax.naming  javax.naming.directory  javax.naming.event  javax.naming.ldap  javax.naming.spi  javax.net  javax.net.ssl  javax.print  javax.print.attribute  javax.print.attribute.standard  javax.print.event  javax.rmi  javax.rmi.CORBA  javax.rmi.ssl  javax.security.auth  javax.security.auth.callback  javax.security.auth.kerberos  javax.security.auth.login  javax.security.auth.spi  javax.security.auth.x500  javax.security.cert  javax.security.sasl  javax.sound.midi  javax.sound.midi.spi  javax.sound.sampled  javax.sound.sampled.spi  javax.sql  javax.sql.rowset  javax.sql.rowset.serial  javax.sql.rowset.spi  javax.swing  javax.swing.border  javax.swing.colorchooser  javax.swing.event  javax.swing.filechooser  javax.swing.plaf  javax.swing.plaf.basic  javax.swing.plaf.metal  javax.swing.plaf.multi  javax.swing.plaf.synth  javax.swing.table  javax.swing.text  javax.swing.text.html  javax.swing.text.html.parser  javax.swing.text.rtf  javax.swing.tree  javax.swing.undo  javax.transaction  javax.transaction.xa  javax.xml  javax.xml.datatype  javax.xml.namespace  javax.xml.parsers  javax.xml.transform  javax.xml.transform.dom  javax.xml.transform.sax  javax.xml.transform.stream  javax.xml.validation  javax.xml.xpath  org.ietf.jgss  org.omg.CORBA  org.omg.CORBA\_2\_3  org.omg.CORBA\_2\_3.portable  org.omg.CORBA.DynAnyPackage  org.omg.CORBA.ORBPackage  org.omg.CORBA.portable  org.omg.CORBA.TypeCodePackage  org.omg.CosNaming  org.omg.CosNaming.NamingContextExtPackage  org.omg.CosNaming.NamingContextPackage  org.omg.Dynamic  org.omg.DynamicAny  org.omg.DynamicAny.DynAnyFactoryPackage  org.omg.DynamicAny.DynAnyPackage  org.omg.IOP  org.omg.IOP.CodecFactoryPackage  org.omg.IOP.CodecPackage  org.omg.Messaging  org.omg.PortableInterceptor  org.omg.PortableInterceptor.ORBInitInfoPackage  org.omg.PortableServer  org.omg.PortableServer.CurrentPackage  org.omg.PortableServer.POAManagerPackage  org.omg.PortableServer.POAPackage  org.omg.PortableServer.portable  org.omg.PortableServer.ServantLocatorPackage  org.omg.SendingContext  org.omg.stub.java.rmi  org.w3c.dom  org.w3c.dom.bootstrap  org.w3c.dom.events  org.w3c.dom.ls  org.xml.sax  org.xml.sax.ext  org.xml.sax.helpers |

Source: Sun Standards

### Standard Java distribution packages

The standard java distribution packages from Sun begin with the identifier ‘java’ or ‘javax’. Sun has reserved this right so that the standard java packages are named in a consistent manner regardless of the vendor of your Java development environment.

Source: Ambler

### Package names must begin with the reversed internet domain name

The prefix of a unique package name is always written in all-lowercase ASCII letters and should be: com.moodys.

Source: Ambler

# Java layout standards

## Java source files layout standards

These guidelines are intended to assist developers with organizing the files, directories and code associated with an application in a logical fashion. By structuring your applications as shown in these guidelines will make it easier to manage and maintain projects, especially when multiple developers contribute to the same project or projects are maintained during an extended lifetime. Having predefined, consistent and standard layouts save time.

Following these conventions will help developers establish an overall structure for application artefacts and code.

### File layout

Java source files have the following ordering:

* File comments
* Package and import statements
* Type declarations

Source: Sun Standards

### File comments layout

All source files should begin with a c-style comment that lists the copyright notice.

|  |
| --- |
| Example |
| /\*  \* Copyright Moody's Corporation. All Rights Reserved.  \* This software is the proprietary information of Moody's Corporation.  \* Use is subject to license terms.  \*/ |

Source: Sun Standards

### Package statement is obligatory

All source files should belong to a specific package by including the package statement as the first non-comment line of each Java source file.

Types belonging to the null package cannot be imported.

|  |
| --- |
| package com.moodys.mypackage; |

Source: GeoSoft

### Sort and group import statements

Import statements should follow the package statement and be sorted and grouped consistently.

Use the following guidelines to sort and group imports statements:

* Group associated packages together with a blank line between each package group
* Declare imports from the java and javax packages first
* Sort the remaining packages lexicographically

|  |
| --- |
| import java.io.BufferedReader;  import java.net.HttpURLConnection;  import javax.swing.ActionMap;  import javax.swing.event.ChangeEvent;  import org.antlr.stringtemplate.StringTemplate;  import org.antlr.stringtemplate.StringTemplateGroup; |

Source: GeoSoft

## Java types layout standards

### Type layout

Order classes and interfaces in the following manner:

* Type documentation comment
* The class or interface statement
* Type implementation comment
* Class variables in order public, protected, package, private
* Instance variables in order protected, package, private
* Constructors
* Methods grouped by functionality

|  |
| --- |
| /\*\*  \* Class description goes here.  \*  \* @version 1.00  \* @author Firstname Lastname  \*  \*/  public class Blah extends SomeClass {  /\* A class implementation comment can go here. \*/  /\*\* classVar1 documentation comment \*/  public static int classVar1;  /\*\*  \* classVar2 documentation comment that happens to be  \* more than one line long  \*/  private static Object classVar2;  /\*\* instanceVar2 documentation comment \*/  protected int instanceVar2;  /\*\* instanceVar3 documentation comment \*/  private Object[] instanceVar3;  /\*\*  \* ...constructor Blah documentation comment...  \*/  public Blah() {  // ...implementation goes here...  }  /\*\*  \* ...method doSomething documentation comment...  \*/  public void doSomething() {  // ...implementation goes here...  }  /\*\*  \* ...method doSomethingElse documentation comment...  \* @param someParam description  \*/  public void doSomethingElse(Object someParam) {  // ...implementation goes here...  }  } |

Source: Sun Standards

### Declare one variable per line

One variable declaration per line is recommended since it encourages commenting.

|  |
| --- |
| int level; // indentation level  int size; // size of table |

Source: Sun Standards

### Declare variables at beginning of blocks

Put declarations only at the beginning of blocks.

A block is any code surrounded by curly braces { and }. Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

Note: The one exception to the rule is indexes of for loops, which in Java can be declared in the for statement.

|  |
| --- |
| void myMethod() {  int int1 = 0; //beginning of method block  if (condition) {  int int2 = 0; //beginning of “if” block  ...  }  } |

Source: Sun Standards

### Method formatting guidelines

The following formatting rules apply to methods:

* No space between a method name and the parenthesis ( starting its parameter list
* Open brace { appears at the end of the same line as the declaration statement
* Closing brace } starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the } should appear immediately after the {
* Methods are separated by a blank line

|  |
| --- |
| class Sample extends Object {  int ivar1;  int ivar2;  Sample(int i, int j) {  ivar1 = i;  ivar2 = j;  }  int emptyMethod() {}  ...  } |

Source: Sun Standards

## Java statements layout standards

### Indent enclosed statements

Enclosed statements of a compound statement should be indented one more level.

Source: Sun Standards

### Braces around all block statements

Braces should be used around all block statements.

This should be applied even to single statement blocks in control structures to avoid accidentally introducing bugs due to forgetting to add braces.

Source: Sun Standards

### Opening brace at end of line

The opening brace of a compound statement should be placed at the end of the line that begins the compound statement.

|  |
| --- |
| if (condition) {  doStuff();  } |

Source: Sun Standards

### One statement per line

Each line should contain at most one statement.

|  |
| --- |
| argv++; // Correct  argc--; // Correct  argv++; argc--; // AVOID! |

Source: Sun Standards

### White space

Blank spaces should be used in the following circumstances:

* A keyword followed by a parentheses should be separated by a space
* After commas in argument lists
* Around binary and ternary operators
* Never around unary operators
* After semi-colons in a for statement
* After casts

|  |
| --- |
| while (true); //between keyword and parentheses  a += c + d; //around binary operators  n++; //never around unary operators  for (expr1; expr2; expr3); //after semi-colons in if statement  myMethod((byte) aNum, (Object) x); //after casts |

Source: Sun Standards

### The if statement layout

The if statement layout should have the following form:

|  |
| --- |
| if (condition) {  statements;  }  if (condition) {  statements;  }  else {  statements;  }  if (condition) {  statements;  }  else if (condition) {  statements;  }  else {  statements;  }  //AVOID the following form  if (condition)  statement; |

Source: Sun Standards

### The for statement layout

A for statement should have the following form:

|  |
| --- |
| for (initialization; condition; update) {  statement;  }  //Empty for statement should take the following form:  for (initialization; condition; update); |

Source: Sun Standards

### The while statement layout

A while statement should have the following form:

|  |
| --- |
| while (condition) {  statements;  }  //Empty while statement  while (condition); |

Source: Sun Standards

### The do-while statement layout

A do-while statement should have the following form:

|  |
| --- |
| do {  statements;  } while (condition); |

Source: Sun Standards

### The switch statement layout

A switch statement should have the following form:

(Note: Add a comment whenever a case falls through as illustrated in the example.)

|  |
| --- |
| switch (condition) {  case ABC:  statements;  /\* falls through \*/  case DEF:  statements;  break;  case XYZ:  statements;  break;  default:  statements;  break;  } |

Source: Sun Standards

### The try-catch statement layout

A try-catch statement should have the following format:

|  |
| --- |
| try {  statements;  }  catch (ExceptionClass e) {  statements;  }  try {  statements;  }  catch (ExceptionClass e) {  statements;  }  finally {  statements;  } |

Source: Sun Standards

## Other Java layout standards

### Blank lines

Two blank lines should be used in the following circumstances:

* Between sections of a source file
* Between class and interface declarations

Single blank lines should be used:

* Between methods
* Between the local variables in a method and its first statement
* Before a block or single-line comment
* Between logical sections inside a method to improve readability

Source: Sun Standards

### Indent 4 spaces

The unit of indentation is 4 spaces.

Source: Sun Standards

### Line length is limited to 120 columns

Lines that exceed 120 columns should be wrapped.

It improves readability when unintentional line breaks and horizontal scrolls are avoided when passing a file between programmers.

Source: Sun Standards

### Line wrapping guidelines

Lines should be wrapped consistently and the existence of wrapping should be obvious.

When an expression will not fit on a single line, break it according to these general principles:

* Break after a comma
* Break after an operator
* Prefer higher-level breaks to lower-level breaks
* Align the new line with the beginning of its expression
* If the above rules lead to confusing code or to code indented too far to the right, indent 8 spaces instead
* Wrapped if statements should be indented 8 spaces to avoid obscuring the body

|  |
| --- |
| someMethod(longExpression1, longExpression2, longExpression3,  longExpression4, longExpression5);  var = someMethod1(longExpression1,  someMethod2(longExpression2,  longExpression3));  //Breaking arithmetic expressions at higher level  longName1 = longName2 \* (longName3 + longName4 – longName5) +  4 \* longName6; //PREFERRED  longName1 = longName2 \* (longName3 + longName4 -  longName5) + 4 \* longName6; //AVOID  //Conventional indentation  someMethod(int anArg, Object anotherArg, String yetAnotherArg,  Object andStillAnother) {  ...  }  //Indent 8 spaces to avoid very deep indents  private static synchronized horkingLongMethodName(int anArg,  Object anotherArg, String yetAnotherArg,  Object andStillAnother) { ... }  //Wrapped if statements indented 8 spaces  if ((condition1 && condition2) ||  (condition3 && condition4) ||  !(condition5 && condition6)) {  doSomething();  } |

Source: Sun Standards

# Java commenting standards

## Doc comments

A doc comment is written in HTML and must precede a class, field, constructor or method declaration. It is made up of two parts -- a description followed by block tags.

|  |
| --- |
| /\*\*  \* The first sentence is the Summary sentence.  \* Description follows in one or more sentences and can include inline tags  \* and html markup.  \* <p>  \* This is a new paragraph containing more information about the item. A link  \* to another type or method can be included in a {@link package.class#member} tag.  \*  \* @tag Comment for the tag  \*/ |

Note the following attributes of the doc comment (as illustrated in the sample above):

* A doc comment is indicated by a begin-comment delimiter (/\*\*)
* The first sentence found in the doc comment is placed in the summary table and index by the Javadoc tool. Use this sentence to write a short summary of the item being commented.
* The remaining sentences before the @tags contain the full description.
* Delimit new paragraphs with a <p> on its own line.
* A link to another type or method is included in a {@link package.class#member} tag.
* @tags are separated from the rest of the comment by a single blank line.
* A @tag consists of the tag name followed by a tag comment.

See the Javadoc reference documentation at <http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html> for detailed reference documentation.

## Comment types

Javadoc can generate output originating from four different types of "source" files:

* Source code files for Java classes (.java) - these contain class, interface, field, constructor and method comments.
* Package comment files - these contain package comments
* Overview comment files - these contain comments about the set of packages
* Miscellaneous unprocessed files - these include images, sample source code, class files, applets, HTML files, and whatever else you might want to referenced from the previous files.

We choose to concentrate only on producing comments for source code files.

A source code file can contain the following types of comments:

* Copyright notice
* Type header comments
* Type implementation comments
* Method header comments
* Method implementation comments
* Inline comments

In the following sample the relative location of each these comments is illustrated.

|  |
| --- |
| /\*  \* Copyright notice  \*/  <package and import statements>  /\*\*  \* Type header comment  \*/  <Type declaration> {  /\* Type implementation comment \*/  /\*\*  \* Method header comment  \*/  <Method declaration> {  /\* Method implementation comment (only for classes) \*/  //inline comments  <some code > //inline comments  /\* inline  comments \*/  }  } |

### Copyright notice

Every java source code file must contain a copyright information at the top of the file, i.e., before the package statement. The following copyright statement must be used. Note that the begin-comment delimiter (/\*\*) is not used in the copryright notice.

|  |
| --- |
| /\*  \* Copyright Moody's Corporation. All Rights Reserved.  \* This software is the proprietary information of Moody's Corporation.  \* Use is subject to license terms.  \*/ |

### Type header comment

The type header comment must include class summary, description, author and version information.

|  |
| --- |
| /\*\*  \* The summary sentence.  \* One or more description sentences.  \*  \* [@author](http://java.sun.com/j2se/javadoc/writingdoccomments/#@param) Author name  \* [@author](http://java.sun.com/j2se/javadoc/writingdoccomments/#@param) Another author  \* @version 1.0  \*/  <type declaration> {  …  } |

Multiple authors are specified with a separate @author tag for each author on its own line. The @version tag specifies the version of the class. Optionally @since, @deprecated and @see tags can be included as well. See the reference documentation for details on how to use these tags.

#### @author

Include a @author tag for each author of a type. If you edit a type that does not include your name in a @author tag, add a new @author tag below the last with your name.

#### @version

Include a @version tag that contains the version number of the class in the form <major>.<minor>

#### @since

This tag need only be specified if the type has been introduced at a later stage than the rest of the package. It must include a version number of the form <major>.<minor> that indicates the point at which the type was introduced to the package.

#### @deprecated

The @deprecated description in the first sentence should at least tell the user when the API was deprecated and what to use as a replacement. Only the first sentence will appear in the summary section and index. Subsequent sentences can also explain why it has been deprecated. When generating the description for a deprecated API, the Javadoc tool moves the @deprecated text ahead of the description, placing it in italics and preceding it with a bold warning: "Deprecated". An @see tag (for Javadoc 1.1) or {@link} tag (for Javadoc 1.2 or later) should be included that points to the replacement type:

|  |
| --- |
| /\*\*  \* @deprecated As of JDK 1.1, replaced by  \* {@link package.class}  \*/ |

#### @see

Adds a "See Also" heading with a link or text entry that points to *reference*. A doc comment may contain any number of @see tags, which are all grouped under the same heading.

|  |
| --- |
| /\*\*  \* @see package.class label  \*/ |

### Method header comment

The method header comment must include summary, description and required tags.

|  |
| --- |
| /\*\*  \* The summary sentence.  \* One or more description sentences.  \*  \* [@param](http://java.sun.com/j2se/javadoc/writingdoccomments/#@param) <parameter name> <parameter description>  \* [@param](http://java.sun.com/j2se/javadoc/writingdoccomments/#@param) <parameter name> <parameter description>  \* [@return](http://java.sun.com/j2se/javadoc/writingdoccomments/#@param) <return description>  \* @throws <exception class> <exception description>  \* @since <version>  \*/  <method declaration> {  …  } |

For neatness tab tag comments so that they line up with each other like in the extract above.

The following rules apply to tags:

#### @param

An @param tag is required for each parameter to a method.

The @param tag is followed by the name (not data type) of the parameter, followed by a description of the parameter. By convention, the first noun in the description is the data type of the parameter. (Articles like "a", "an", and "the" can precede the noun.) An exception is made for the primitive int, where the data type is usually omitted. Additional spaces can be inserted between the name and description so that the descriptions line up in a block. Dashes or other punctuation should not be inserted before the description, as the Javadoc tool inserts one dash.

Parameter names are lowercase by convention. The data type starts with a lowercase letter to indicate an object rather than a class. The description begins with a lowercase letter if it is a phrase (contains no verb), or an uppercase letter if it is a sentence. End the phrase with a period only if another phrase or sentence follows it.

|  |
| --- |
| **Examples** |
| @param x the x-coordinate, measured in pixels (a phrase on its own)  @param x the x-coordinate. Measured in pixels. (a phrase followed by a sentence) |

Note: Order multiple @param tags in the order the parameters are declared in the method.

#### @return

Omit @return for methods that return void and for constructors; include it for all other methods, even if its content is entirely redundant with the method description. Having an explicit @return tag makes it easier for someone to find the return value quickly. Whenever possible, supply return values for special cases (such as specifying the value returned when an out-of-bounds argument is supplied).

Use the same capitalization and punctuation as you used in @param.

#### @throws

@throws is required for every checked exception thrown by the method. Also include any unchecked exceptions that the caller might reasonably catch. Errors should not be documented as they are unpredictable.

#### @since

An @since tag is only required if the method was introduced in a later version of the class.

#### @deprecated

The @deprecated description in the first sentence should at least tell the user when the API was deprecated and what to use as a replacement. Only the first sentence will appear in the summary section and index. Subsequent sentences can also explain why it has been deprecated. When generating the description for a deprecated API, the Javadoc tool moves the @deprecated text ahead of the description, placing it in italics and preceding it with a bold warning: "Deprecated". An @see tag (for Javadoc 1.1) or {@link} tag (for Javadoc 1.2 or later) should be included that points to the replacement method:

|  |
| --- |
| /\*\*  \* @deprecated As of JDK 1.1, replaced by  \* {@link package.class#method}  \*/ |

#### @see

Adds a "See Also" heading with a link or text entry that points to *reference*. A doc comment may contain any number of @see tags, which are all grouped under the same heading.

|  |
| --- |
| /\*\*  \* @see package.class #member label  \*/ |

### Implementation comments

If required, include implementation comments for a type or method directly after the declaration. These comments are targeted at developers who need to maintain the item and include implementation details that assist with maintenance. Do not mark these comments with a begin-comment delimiter since we don’t want consumers of the class to see the nasty implementation details.

|  |
| --- |
| /\*\*  \* Type header comment  \*/  <type or method declaration> {  /\* Nasty implementation details useful for maintenance of the class  \* but not to consumers of the class.  \*/  …  } |

## Usage guidelines

### The First Sentence

The first sentence of each doc comment should be a summary sentence, containing a concise but complete description of the API item. This means the first sentence of each member, class, interface or package should be a concise description. The Javadoc tool copies this first sentence to the appropriate member, class/interface or package summary. This makes it important to write crisp and informative initial sentences that can stand on their own.

This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first tag (as defined below).

### Implementation independence

Write comment descriptions to be implementation-independent, but specifying such dependencies where necessary.

### Automatic reuse of doc comments

Automatic reuse of doc comments occurs in three cases:

* When a method in a class overrides a method in a superclass
* When a method in an interface overrides a method in a superinterface
* When a method in a class implements a method in an interface

In the first two cases, if a method m () overrides another method, The Javadoc tool will generate a subheading "Overrides" in the documentation for m (), with a link to the method it is overriding.

In the third case, if a method m() in a given class implements a method in an interface, the Javadoc tool will generate a subheading "Specified by" in the documentation for m(), with a link to the method it is implementing.

In all three of these cases, if the method m () contains no doc comments or tags, the Javadoc tool will also copy the text of the method it is overriding or implementing to the generated documentation for m (). So if the documentation of the overridden or implemented method is sufficient, you do not need to add documentation for m (). If you add any documentation comment or tag to m (), the "Overrides" or "Specified by" subheading and link will still appear, but no text will be copied.

### Use <code> style for keywords and names

Keywords and names are offset by <code>...</code> when mentioned in a description. This includes:

* Java keywords
* package names
* class names
* method names
* interface names
* field names
* argument names
* code examples

### Use in-line links economically

You are encouraged to add links for API names (listed immediately above) using the {@link} tag. It is not necessary to add links for all API names in a doc comment. Because links call attention to themselves (by their colour and underline in HTML, and by their length in source code doc comments), it can make the comments more difficult to read if used profusely. We therefore recommend adding a link to an API name if the user might actually want to click on it for more information (in your judgment), and only for the first occurrence of each API name in the doc comment (don't bother repeating a link)

### Omit parentheses for the general form of methods and constructors

When referring to a method or constructor that has multiple forms, and you mean to refer to a specific form, use parentheses and argument types. For example, ArrayList has two add methods: add (Object) and add (int, Object). The add (int, Object) method adds an item at a specified position in this arraylist.

However, if referring to both forms of the method, omit the parentheses altogether. It is misleading to include empty parentheses, because that would imply a particular form of the method. The intent here is to distinguish the general method from any of its particular forms. Include the word "method" to distinguish it as a method and not a field.

|  |
| --- |
| The add method enables you to insert items. (preferred)  The add () method enables you to insert items. (avoid when you mean "all forms" of the add method) |

### Okay to use phrases instead of complete sentences

This holds especially in the initial summary and in @param tag descriptions.

### Use 3rd person (descriptive) not 2nd person (prescriptive)

The description is in 3rd person declarative rather than 2nd person imperative.

|  |
| --- |
| Gets the label. (preferred)  Get the label. (avoid) |

### Method descriptions begin with a verb phrase

A method implements an operation, so it usually starts with a verb phrase:

|  |
| --- |
| Gets the label of this button. (preferred)  This method gets the label of this button. (avoid) |

### Class/interface/field descriptions can omit the subject and simply state the object

These API often describe things rather than actions or behaviors:

|  |
| --- |
| A button label. (preferred)  This field is a button label. (avoid) |

### Use "this" instead of "the" when referring to an object created from current class

For example, the description of the getToolkit method should read as follows:

|  |
| --- |
| Gets the toolkit for this component. (preferred)  Gets the toolkit for the component. (avoid) |

### Do not write Redundant Comments

Comments should say things that the code cannot say for itself.

A comment is redundant if it describes something that adequately describes itself. For example:

|  |
| --- |
| i++; // increment i (avoid) |

Another example is a Javadoc that says nothing more than (or even less than) the function signature:

|  |
| --- |
| /\*\* (avoid)  \* @param sellRequest  \* @return  \* @throws ManagedComponentException  \*/  public SellResponse beginSellItem(SellRequest sellRequest) throws ManagedComponentException |

Source: Clean Code A Handbook of Agile Software Crafts

### Dont comment out code, delete it

Code must not be left commented out. No one will know how old it is or whether it is meaningful or not. Yet no one will delete it because everyone assumes someone else needs it or has plans for it. That code sits there and rots, getting less and less relevant with every passing day. It pollutes the modules that contain it and distracts the people who try to read it. Commented-out code must not be present and if any are found, must be deleted.

Source: Clean Code A Handbook of Agile Software Crafts

# Java coding guidelines

## Best practices

Coding standards for Java are important because they lead to greater consistency within your code and the code of your teammates. Greater consistency leads to code that is easier to understand, which in turn means it is easier to develop and to maintain. This reduces the overall cost of the applications that you create.

What is good code? These are a few of its characteristics:

* Good code is extensible without drastic modification. It's easy to add features without tearing it apart.
* Good code is easy to read and maintain.
* Good code is well documented.
* Good code makes it hard to write bad code around it.
* Good code is easy to test.
* Good code is easy to debug.
* Good code contains no code duplication.
* Good code gets re-used.

### Avoid star import

Avoid importing everything from a package.

Importing all types from a package leads to tight coupling between packages and might lead to problems when a new version of a library introduces name clashes.

|  |
| --- |
| import java.io.\*; //AVOID  import java.io.BufferedReader; //PREFERRED |

Source: Sun Standards

### Initialize variables on declaration

Variables should be initialized where they are declared.

This ensures that variables are valid at any time. The only exception to this rule is when the initial value of the variable is based on some computation occurring first. In this case leave the value un-initialized, do not set a phony value.

Source: Sun Standards

### Variables have smallest possible scope

Variables should be declared within the smallest possible scope.

Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

Source: Sun Standards

### Do not assign several variables

Avoid assigning several variables in a single statement.

|  |
| --- |
| fooBar.fChar = barFoo.lchar = 'c'; //AVOID |

Source: Sun Standards

### Fields are private

Fields should be declared private.

The concept of Java information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C++ struct). In this case it is appropriate to make the class instance variables public.

Source: Ambler

### Avoid hiding declarations

Avoid local variable declarations that hide variables at higher levels.

|  |
| --- |
| int count;  ...  myMethod() {  if (condition) {  int count = 0; //AVOID!  ...  }  ...  } |

Source: Sun Standards

### Access static member through class identifier

Avoid using an object to access a class (static) variable or method. Use a class name instead.

|  |
| --- |
| classMethod(); //OK  AClass.classMethod(); //OK  anObject.classMethod(); //AVOID |

Source: Sun Standards

### Avoid magic numbers

Numerical literals should be avoided, except for -1, 0 and 1, which can appear in a for loop as counter values.

Source: Sun Standards

### Use accessors for constants

Use an accessor to retrieve the value of a business constant, rather than using a static final field.

So-called business constants change over time because the underlying business rules change.

Source: Ambler

### Keep methods concise

Methods should be short and to the point.

Two good rules of thumb are:

* The 30 second rule: another developer should be able to read and understand what the method does within 30 seconds
* A method should not take up more than a screen (approx. 30 lines)

If these rules of thumb are exceeded consider refactoring to multiple methods.

Source: Ambler

### Do not embed assignments

Do not use embedded variable assignments.

|  |
| --- |
| d = (a = b + c) + r; //AVOID |

Source: Sun Standards

### Normal case in if, exception in else

The nominal case should be put in the if-part and the exception in the else-part of an if statement

Makes sure that the exceptions do not obscure the normal path of execution. This is important for both the readability and performance.

|  |
| --- |
| boolean isError = readFile(fileName);  if (!isError) {  //nominal path  } else {  //exception path  } |

Source: GeoSoft

### Avoid executable statements in conditionals

Conditionals with executable statements are simply very difficult to read. This is especially true for programmers new to Java.

|  |
| --- |
| file = openFile (fileName); // NOT: if ((file = openFile (fileName)) != null) {  if (file != null) {  ...  } |

Source: GeoSoft

### Avoid break and continue in loops

The use of break and continue in loops should be avoided.

These statements should only be used if they prove to give higher readability than their structured counterparts.

Source: GeoSoft

### Only loop control statements must be included in the for() construction

This increases maintainability and readability. It makes a clear distinction of what controls and what is contained in the loop.

|  |
| --- |
| sum = 0;  for (i = 0; i < 100; i++) {  sum+= value[i];  }  //NOT!!  for (i = 0, sum = 0; i < 100; i++) {  sum+= value[i];  } |

Source: GeoSoft

### Use parentheses to indicate precedence

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

|  |
| --- |
| if (a == b && c == d) // AVOID!  if ((a == b) && (c == d)) // RIGHT |

Source: Sun Standards

### Parentheses around test in ? operator

If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesised.

|  |
| --- |
| (x >= 0) ? x : -x; |

Source: Sun Standards

### Checked vs Unchecked Exceptions

Use a checked exception if calling code can do something sensible with the exception. Use an unchecked exception if the exception is fatal, or if callers won't gain by catching it. Remember that a J2EE container (such as a web container) can be relied on to catch unchecked exceptions and log them.

Checked exceptions:

* Represent invalid conditions in areas *outside the immediate control of the program* (invalid user input, database problems, network outages, absent files)
* Are subclasses of Exception
* Methods are *obliged* to establish a policy for all checked exceptions thrown by its implementation (either pass the checked exception further up the stack, or handle it somehow)

Unchecked exceptions:

* Represent *defects* in the program (often invalid arguments passed to a non-private method)
* Are subclasses of [RuntimeException](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/RuntimeException.html), and are usually implemented using IllegalArgumentException, NullPointerException, or IllegalStateException
* Methods are *not* obliged to establish a policy for the unchecked exceptions thrown by its implementation

Runtime exceptions can occur anywhere in a program and in a typical one they can be very numerous. Having to add runtime exceptions in every method declaration would reduce a program's clarity. Thus, the compiler does not require that you catch or specify runtime exceptions (although you can).

One case where it is common practice to throw a RuntimeException, is when the user calls a method incorrectly. For example, a method can check if one of its arguments is incorrectly null. If an argument is null, the method might throw a NullPointerException, which is an unchecked exception.

The following table can be used as a guide when choosing whether to throw checked or unchecked exceptions.

|  |  |  |
| --- | --- | --- |
| **Question** | **Example** | **Recommendation if the answer is yes** |
| Should all callers handle this problem? Is the exception essentially a second return value for the method? | Spending limit exceeded in a processInvoice() method | Define and used a checked exception and take advantage of Java's compile-time support. |
| Will only a minority of callers want to handle this problem? | JDO exceptions | Extend RuntimeException. This leaves callers the choice of catching the exception, but doesn't force all callers to catch it. |
| Did something go horribly wrong? Is the problem unrecoverable? | A business method fails because it can't connect to the application database | Extend RuntimeException. We know that callers can't do anything useful besides inform the user of the error. |
| Still not clear? |  | Extend RuntimeException. Document the exceptions that may be thrown and let callers decide which, if any, they wish to catch. |

The only danger in using unchecked exceptions is that the exceptions may be inadequately documented. When using unchecked exceptions, be sure to document all exceptions that may be thrown from each method, allowing calling code to choose to catch even exceptions that you expect will be fatal.

Source: The Java Tutorial, Expert one-on-one J2EE Design and Development

## Make exceptions informative

Consider the case of exceptions of the same class reflecting different problems, but distinguished only by their message strings. These are unhelpful to Java code catching them. Exception message strings are of limited value: they may be helpful to explain problems when they appear in log files, but they won't enable the calling code to react appropriately, if different reactions are required, and they can't be relied on for display to users. When different problems may require different actions, the corresponding exceptions should be modeled as separate subclasses of a common superclass. Sometimes the superclass should be abstract. Calling code will now be free to catch exceptions at the relevant level of detail.

Include as much context information as possible with exceptions. If an exception probably results from a programming error, try to include information on how to rectify the problem.

Separate error messages for display to users from exception code, by including an error code with exceptions. When it's time to display the exception, the code can be resolved: for example, from a properties file.

Source: Expert one-on-one J2EE Design and Development

### Return statements do not use parentheses

A return statement with a value should not use parentheses unless they make the return value more obvious in some way.

|  |
| --- |
| return;  return myDisk.size();  return (size ? size : defaultSize); |

Source: Sun Standards

### Structure matches intent

Try to make the structure of your program match the intent.

|  |
| --- |
| //instead of  if (booleanExpression) {  return true;  } else {  return false;  }  //use  return booleanExpression;  //instead of  if (condition) {  return x;  }  return y;  //use  return (condition ? x : y); |

Source: Sun Standards

## Consider the ordering of object comparisons

The following two lines of code will produce the same result in normal operation:

|  |
| --- |
| if (myStringVariable.equals(MY\_STRING\_CONSTANT))  if (MY\_STRING\_CONSTANT.equals(myStringVariable)) |

However, the second form is more robust. What if myStringVariable is null? The second condition will evaluate to false, without error, while the first will throw a NullPointerException. It's usually a good idea to perform object comparisons by calling the equals() method on the object less likely to be null. If it's an error for the other object to be null, perform an explicit check for null and throw the appropriate exception (which won't be NullPointerException).

Source: Expert one-on-one J2EE Design and Development

## Use short-circuit evaluation

Sometimes we can rely on Java's short-circuit evaluation of Boolean expressions to avoid potential errors: for example, with null objects. Consider the following code fragment:

|  |
| --- |
| if ( (o != null) && (o.getValue() < 0)) |

This is safe even if the object o is null. In this case, the second test won't be executed, as the condition has already evaluated to false. Of course, this idiom can only be used if it reflects the intention of the code. Something quite different might need to be done (besides evaluating this condition to false) if o is null. However, it's a safe bet that we don't want a NullPointerException.

An alternative is to perform the second check in an inner if statement, only after an outer if statement has established that the object is non-null. However, we don't recommend this approach unless there is some other justification for the nested if statements (which, however, there often will be), as statement nesting adds complexity.

Source: Expert one-on-one J2EE Design and Development

# References

* [Java Code Conventions](http://java.sun.com/docs/codeconv/) (Sun Microsystems)
* [The Java Tutorial](http://java.sun.com/docs/books/tutorial/) (Sun Microsystems)
* [Java Programming Style Guidelines](http://geosoft.no/development/javastyle.html) (GeoSoft – Geotechnical Software Services)
* [Writing robust Java Code](http://www.ambysoft.com/downloads/javaCodingStandards.pdf) (Ambysoft)
* [How to write doc comments for the Javadoc tool](http://java.sun.com/j2se/javadoc/writingdoccomments/) (Sun Microsystems)
* Clean Code A Handbook of Agile Software Crafts – Robert C. Martin (Prentice Hall)

# Approvals

**From:** Biswas, Prantik   
**Sent:** Thursday, June 13, 2013 3:33 PM  
**To:** Mishra, Ranjan (non-empl)  
**Cc:** Vytla, Veerraju (non-empl)  
**Subject:** RE: Coding Standards

Thanks for the prompt action my comments and even incorporating them in the Standards.

For the deviation, I am in agreement with Veera on the 120 column limit.

Please treat this as my approval on the suggestions incorporated in the doc.

Thanks and regards

.................................................  
**Prantik Biswas**  
Sr. Software Engineer

212.553.1167 tel

732.425.7197 mobile

212.298.6033 fax  
[prantik.biswas@moodys.com](mailto:prantik.biswas@moodys.com)  
  
Moody’s Investors Service

7 World Trade Center  
250 Greenwich Street  
New York, NY 10007   
USA  
[www.moodys.com](http://www.moodys.com/)

.................................................