

Hewlett-Packard Company

Prepared by: PGS

Table of Contents

[1 Why Coding Standards 3](#_Toc246477789)

[1.1 File Suffixes 3](#_Toc246477790)

[1.2 File Organization 3](#_Toc246477791)

[1.2.1 Class and Interface 4](#_Toc246477792)

[1.2.2 Implementation Comment Formats 7](#_Toc246477793)

[1.2.3 Declarations 11](#_Toc246477794)

[1.2.4 Statements 12](#_Toc246477795)

[2 Synchronization 17](#_Toc246477796)

[2.1 White Space 17](#_Toc246477797)

[2.2 Naming Conventions 18](#_Toc246477798)

[2.3 Programming Practices 20](#_Toc246477799)

[2.4 Miscellaneous Practices 20](#_Toc246477800)

[2.5 Tips 21](#_Toc246477801)

[3 Exception Handling Standards: 22](#_Toc246477802)

[Common Problems and Best Practices with Java 22](#_Toc246477803)

[3.1 Some Common Problems 23](#_Toc246477804)

[3.1.1 Problem 1: Empty Catch Blocks 23](#_Toc246477805)

[3.1.2 Problem 2: Meaningless Throws Clauses 23](#_Toc246477806)

[3.1.3 Problem 3: Loss of Stack Information 24](#_Toc246477807)

[3.2 Best Practice Guidelines 24](#_Toc246477808)

[3.2.1 Usage of Unchecked Exceptions 25](#_Toc246477809)

[3.2.2 Usage of Checked Exceptions 26](#_Toc246477810)

[3.2.3 Exception Logging 27](#_Toc246477811)

[4 Best Practices with Java 28](#_Toc246477812)

[4.1 Optimization techniques in Loops 28](#_Toc246477813)

[4.2 Optimization techniques in Object creation 28](#_Toc246477814)

[4.3 Optimization techniques in String and StringBuffer 28](#_Toc246477815)

[4.4 Optimization techniques in Serialization 29](#_Toc246477816)

[4.5 Optimization techniques in I/O 29](#_Toc246477817)

[4.6 Optimization techniques in Collections 29](#_Toc246477818)

[4.7 Optimization techniques in Synchronization 29](#_Toc246477819)

[4.8 Optimization techniques in using final keyword 29](#_Toc246477820)

[References 30](#_Toc246477821)

1. Why Coding Standards

Code conventions are important to programmers for a number of reasons:

1. 80% of the lifetime cost of a piece of software goes to maintenance.
2. Hardly any software is maintained for its whole life by the original author.
3. Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
4. If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.
5. For the conventions to work, every person writing software must conform to the code conventions
6. Below are some java code conventions Programmers can follow while coding.
   1. File Suffixes

Java Software uses the following file suffixes:

| File Type | Suffix |
| --- | --- |
| Java source | java |
| Java bytecode | .class |

Frequently used file names include:

| File Name | Use |
| --- | --- |
| BUILD.XML | We use ant to build our application package. |
| README | The preferred name for the file that summarizes the contents of a particular Directory. |

* 1. File Organization

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

Files longer than 2000 lines are cumbersome and should be avoided.

**Java Source Files**

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files have the following ordering:

* Beginning Comments
* Package and import statements
* Class and interface declarations

Example: Beginning Comments

**Beginning Comments**

All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice:

/\*

\* This class represents a sample to demonstrate how to provide

\* class-level comments

\*

\* Version information

\*

\* Date

\*

\* Copyright notice

\*/

Note: The above text to be replaced as shown in example

Error! Reference source not found.Error! Reference source not found.**Beginning Comments (Examples)**

Error! Reference source not found.Error! Reference source not found.**Beginning Comments (Examples)**

[Click here to see an example](#one_example_beginning_comments)

**Package and Import Statements**

The first non-comment line of most Java source files is a package statement. After that, import statements can follow. For example:

package java.awt;

import java.awt.peer.CanvasPeer;

/\* added additonal content \*/

Now add the external classes import statement. Please use always fully qualified imports (no \*). When you have a conflict then import none or the best one and fully qualify the other class(es) when you use them (example: java.util.Date and java.sql.Date).

Note: The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.

* + 1. Class and Interface

**Declarations**

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

An example that includes comments:

| Part of Class/Interface Declaration | Notes |
| --- | --- |
| Class/interface documentation comment (/\*\*...\*/) |  |
| Class or interface statement |  |
| Class/interface implementation comment (/\*...\*/), if necessary | This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment. |
| Class (static) variables | First the public class variables, then the protected, then package level (no access modifier), and then the private. |
| Instance variables | First public, then protected, then package level (no access modifier), and then private. |
| Constructors |  |
| 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. |

**Class Organization Standards**

The contents of the class should be ordered as given below.

Beginning Comments

Package Statement

Import Statements (Each section of imports should be in alphabetical order)

Classes of java.\* packages

Classes of javax.\* packages

Classes of third-party packages

Classes of application architecture packages

Classes of application business packages

Class-level Comments

Public Class Declaration

Constant Declarations

public

protected

default

private

Static Variable Declarations

public

protected

default

private

Instance Variable Declarations

public

protected

default

private

Constructors

Default

Copy

Parameterized

Methods (each method to be preceded by method-level comments) grouped by functionality (and not by access specifier)

**Indentation**

Four spaces should be used as the unit of indentation. The exact construction of the indentation

(spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

**Line Length**

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length-generally no more than 70 characters.

**Wrapping Lines**

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

* Break after a comma.
* Break before an operator.
* Prefer higher-level breaks to lower-level breaks.
* Align the new line with the beginning of the expression at the same level on the previous line.
* If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

[Click here to see more examples on wrapping](#_Wrapping_Lines(Examples))

**Comments**

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /\*...\*/, and //. Documentation comments (known as "doc comments") are Java-only, and are delimited by /\*\*...\*/. Doc comments can be extracted to HTML files using the javadoc tool.

Implementation comments are mean for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective. To be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or no obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters.

Comments should never include special characters such as form-feed and backspace.

* + 1. Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

**Block Comments**

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

/\*

\* Here is a block comment.

\*/

Block comments can start with /\*-, which is recognized by indent (1) as the beginning of a block comment that should not be reformatted. Example:

/\*-

\* Here is a block comment with some very special

\* Formatting that I want indent (1) to ignore.

\*

\* One

\* Two

\* Three

\*/

Note: If you don't use indent (1), you don't have to use /\*- in your code or make any other concessions to the possibility that someone else might run indent (1) on your code.

**Single-Line Comments**

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java code:

if (condition) {

/\* Handle the condition. \*/

...

}

**Trailing Comments**

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here's an example of a trailing comment in Java code:

if (a == 2) {

return TRUE; /\* special case \*/

} else {

return is Prime(a); /\* works only for odd a \*/

}

**End-Of-Line Comments**

The // comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

if (foo > 1) {

// Do a double-flip.

...

}

else {

return false; // Explain why here.

}

//if (bar > 1) {

//

// // Do a triple-flip.

// ...

//}

//else {

// return false;

//}

**Documentation Comments**

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters /\*\*...\*/, with one comment per class, interface, or member. This comment should appear just before the declaration:

/\*\*

\* The Example class provides...

\*/

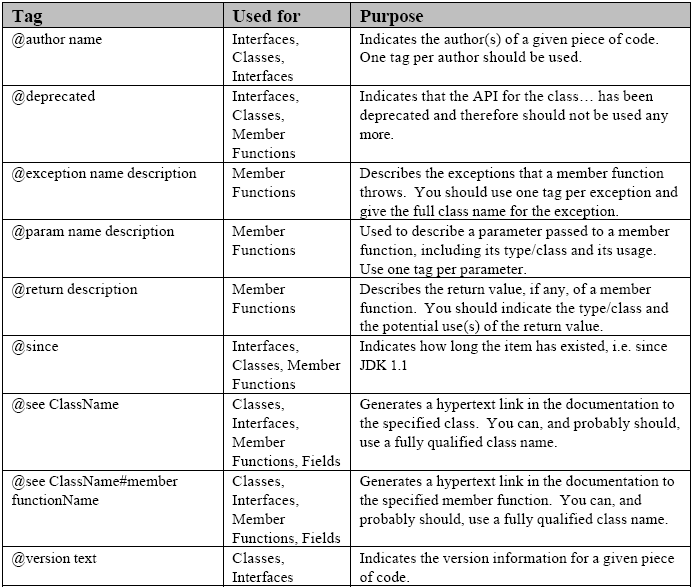
public class Example { ...

Notice that top-level classes and interfaces are not indented, while their members are. The first line of doc comment (/\*\*) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter.

If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately after the declaration. For example, details about the implementation of a class should go in in such an implementation block comment following the class statement, not in the class doc comment.

Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration after the comment.

The chart below gives a brief overview of the current javadoc tags:



**/\* Added new content to support Javadoc standards in Java \*/**

**JavaDoc for Methods**

/\*\*

\* prompts for and reads an array of integers from the standard

\* input. Repeatedly prompts for an integer, then asks the user if

\* s/he is done. This repeats until the user is done, or until the

\* maximum number of integers is reached. If the user enters characters

\* that are not integer format, the method will ask him/her to try

\* again.

\* <p>

\* Limitation: If the user answers the "are you done?" question with

\* any character but a lower-case 'n', it is considered a yes.

\*

\* @param inputArray the array into which the integers are read

\* @return the number of integers read

\*/

private static int readArray(int inputArray[]){}

[Click here for more information on javadoc tags](#_Javadoc_Tags(More_Information))

* + 1. Declarations

**Number per Line**

One declaration per line is recommended since it encourages commenting. In other words,

int level; // indentation level

int size; // size of table

is preferred over

int level, size;

Do not put different types on the same line. Example:

int foo, fooarray[]; //WRONG!

Note: The examples above use one space between the type and the identifier.

Another acceptable alternative is to use tabs, e.g.:

int level; // indentation level

int size; // size of table

Object currentEntry; // currently selected table entry

**Initialization**

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

**Placement**

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.

[Click here to see an example on declarations placement](#_Declarations_Placements(Eaxmples))

**Class and Interface Declarations**

When coding Java classes and interfaces, the following formatting rules should be followed:

* 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 "{"

class Sample extends Object {

int ivar1;

int ivar2;

Sample (int i, int j) {

ivar1 = i;

ivar2 = j;

}

int emptyMethod() {}

...

}

* Methods are separated by a blank line

**Method Declaration Standards**

**Samples of Acceptable Method Declarations**

/\*\*

\* This method is a sample to demonstrate the declaration

\* of methods

\* @param ArrayList The list of courses to attend

\* @param type The type of employee

\* @return The highest priority training for employee type

\*/

public String getHighestPriorityTraining(ArrayList courses

, int type) {

:

:

}

/\*

\* This method is a sample to demonstrate the declaration

\* of methods

\* @param ArrayList The list of the courses to attend

\* @param type The type of employee

\* @return The highest priority training for employee type

\*/

private String getHighestPriorityTraining(ArrayList courses

, int type) {

:

:

}

* + 1. Statements

**Simple Statements**

Each line should contain at most one statement. Example:

argv++; // Correct

argc--; // Correct

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

**Compound Statements**

Compound statements are statements that contain lists of statements enclosed in braces "{statements }". See the following sections for examples.

* The enclosed statements should be indented one more level than the compound statement.
* The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
* Braces are used around all statements; even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

**Return Statements**

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

Example:

return;

return myDisk.size();

return (size ? size : defaultSize);

**if, if-else, if else-if else Statements**

**Samples of Acceptable if Statement Format**

if (condition1) {

abc = 100;

xyz = 200;

}

if (condition1) {

// A single line is also enclosed within flower braces

abc = 100;

}

**Samples of Acceptable if-else Statement Format**

if (condition1) {

abc = 100;

xyz = 200;

} else {

abc = 300;

xyz = 400;

}

**Samples of Acceptable Nested if-else Statement Format**

if (condition1) {

abc = 100;

xyz = 200;

} else if (condition2) {

abc = 300;

xyz = 400;

if (condition2) {

abc = 800;

} else {

xyz = 800;

}

} else {

abc = 500;

xyz = 600;

}

**Samples of Acceptable Ternary Operator Format**

abc = (condition1) ? xyz : def;

abc = (condition1) ? xyz

: def;

abc = (condition1)

? xyz

: def;

Note: if statements always use braces {}. Avoid the following error-prone form:

if (condition) //AVOID! THIS OMITS THE BRACES {}!

statement;

**for Statements**

A for statement should have the following form:

**Acceptable for Statement Formats**

for (initialization; condition; expression) {

statement1;

statement2;

:

}

for (initialization;

condition;

expression) {

statement1;

statement2;

:

}

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

for (initialization; condition; update);

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

**while Statements**

A **while Statement Format** should have the following form:

while (condition) {

statement1;

statement2;

:

}

An empty while statement should have the following form:

while (condition);

**do-while Statements**

A do-while statement should have the following format:

do {

statement1;

statement2;

:

} while (condition);

**Switch Statements**

**Samples of Acceptable switch Statement Format**

switch (condition) {

case abc :

statement1;

statement2;

break;

case xyz :

statement1;

statement2;

break;

/\* The default case should always be present even

\* if it is empty and should be ended with a break

\* statement

\*/

default :

statement1;

statement2;

break;

}

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /\* falls through \*/ comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

**Try-catch Statements**

A try-catch statement should have the following format:

try {

statements;

} catch (ExceptionClass e) {

statements;

}

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

Sample try-catch-finally Block

try {

abc = Integer.parseInt(“abc”);

:

:

} catch(NumberFormatException nfe) {

:

:

} catch(IOException ioe) {

:

:

} finally {

:

:

}

/\* Added new section for Synchronization usage \*/

1. Synchronization

Synchronization describes the set of mechanisms or process for preventing undesirable interleaving of operations or interference between concurrent threads.

A thread that is no longer executing because it is delayed or waiting on some synchronization mechanism is blocked. Once unblocked, awakened, or notified, a thread becomes runnable and eligible for further execution.

Two basics uses exist for thread synchronization: to protect the integrity of shared data and to communicate changes efficiently in program state between cooperating threads.

**Avoid synchronization.**

Synchronization is expensive. It takes time to acquire and release the synchronization objects necessary to synchronize a section of code. Moreover, synchronization serializes access to an object, minimizing concurrency. We have be really think twice before we use synchronize and should only synchronize when it’s truly necessary.

Do not arbitrarily synchronize every public method. If it does not --- if the method only operates on its local variables, parameters, or synchronized objects, then synchronized is not required.

Do not synchronized classes that provide fundamental data types or structures.

Avoid unnecessary synchronization when reading or writing instance variables.

* 1. White Space

**Blank Lines**

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

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

One blank line should always be used in the following circumstances:

* Between methods
* Between the local variables in a method and its first statement
* Between logical sections inside a method to improve readability

**Blank Spaces**

Blank spaces should be used in the following circumstances:

* A keyword followed by a parenthesis should be separated by a space. Example:

while (true) {

...

}

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

* A blank space should appear after commas in argument lists.
* All binary operators except. Should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

a += c + d;

a = (a + b) / (c \* d);

while (d++ = s++) {

n++;

}

printSize("size is " + foo + "\n");

* The expressions in a for statement should be separated by blank spaces. Example:

for (expr1; expr2; expr3)

* Casts should be followed by a blank space. Examples:

myMethod((byte) aNum, (Object) x);

myMethod((int) (cp + 5), ((int) (i + 3))

* 1. Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

| Identifier Type | Rules for Naming | Examples |
| --- | --- | --- |
| Packages | The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.  Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names. | com.sun.eng  com.apple.quicktime.v2  edu.cmu.cs.bovik.cheese |
| Classes | Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML). | class Raster; class ImageSprite; |
| Interfaces | Interface names should be capitalized like class names. | interface RasterDelegate; interface Storing; |
| Methods | Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized. | run(); runFast(); getBackground(); |
| Variables | Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore \_ or dollar sign $ characters, even though both are allowed.  Variable names should be short yet meaningful. 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. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters. | int i;  char c;  float myWidth; |
| Constants | The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("\_"). (ANSI constants should be avoided, for ease of debugging.) | static final int MIN\_WIDTH = 4;  static final int MAX\_WIDTH = 999;  static final int GET\_THE\_CPU = 1; |

* 1. Programming Practices

Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten-often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a struct instead of a class (if Java supported struct), then it's appropriate to make the class's instance variables public.

**Referring to Class Variables and Methods**

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

classMethod(); //OK

AClass.classMethod(); //OK

anObject.classMethod(); //AVOID!

**Constants**

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

**Variable Assignments**

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

fooBar.fChar = barFoo.lchar = 'c'; // AVOID!

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

if (c++ = d++) { // AVOID! (Java disallows)

...

}

should be written as

if ((c++ = d++) != 0) {

...

}

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

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

should be written as

a = b + c;

d = a + r;

* 1. Miscellaneous Practices

**Parentheses**

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

**Returning Values**

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

if (booleanExpression) {

return true;

} else {

return false;

}

should instead be written as

return booleanExpression;

Similarly,

if (condition) {

return x;

}

return y;

should be written as

return (condition ? x : y);

**Expressions before `?' in the Conditional Operator**

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

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

**Special Comments**

Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken.

* 1. Tips

These are practices, which should be followed when applicable to improve the quality of deployed code.

**Avoid return in the middle of a method**

Returns in the middle of a method are confusing in many situations. It is also easy for a maintainer to write code after the return assuming that it will be called.

//DO NOT DO

If( x == null) {

Return;

}

//code to do logic

return ;

//INSTEAD DO

if(x != null) {

//code to do logic

....

}

return;

**Avoid using import xxx.\***

Import statements which end in "\*" should be avoided. Whenever possible specific classes should be imported. If import xxx.\* is used it should only be done for thoroughly used and well known packages like java.awt.

Use of import xxx.\* has two impacts on the code:

• Longer compile time, since javac has to search all imported classes for dependencies

• Problems with classes of the same name imported more than once.

**Comment } for long blocks**

When ending a long block of code, 20 lines or more, the closing "}" should have a comment after it noting where the block started. This should also be used in situations where four or more levels of indenting are present.

For example

if (a == b) {

//start code here

...

} //if(a == b)

NOTE: Do not leave classes in an inconsistent state

Any method on a class should be able to be called at any time in any order and still produce reasonable results.

For example

* After construction any method on the class should be callable and produce reasonable results.
* If two variables in a field are related, any method changing one should update the other as well.
* While, to get a desirable result there may be a sequence of methods, which should be called; calling them any order should produce reasonable results.

**Catch Throwable appropriately**

Any Throwable, other than subclasses of Exception, are not required to be caught. They should normally be caught only in the top level of UI(User Interface) code. When caught they should display to the user an error message and if possible gracefully exit the application. These are not normally errors that can be handled and a program should rarely continue after encountering one.

**Use StringBuffer for non constant strings when appropriate**

The class String is a constant string. Appending information to it, or reassigning it causes another memory allocation to take place. When doing repeated concatenation of strings this memory overhead can be very expensive. This is not the case with StringBuffer. It allows append with little additional memory overhead.

**Do not try to optimize by combining statements**

Do not do double assignments, or combine multiple expressions into one to improve performance. This is the job of the compiler. In most cases it does a better job and leaves the code less confusing

**Nullify temporary references, which are pointing to large amounts of memory**

While there are no "memory leaks" in Java, there are still memory problems. If a reference, which is no longer needed, is held to a chunk of memory it will not be freed till that reference goes out of scope. Therefore, when temporary references to large amounts of memory are no longer being used nullify them.

**Close resources**

When done with a file or other resource, ensure close is called. While this may be done by finalize when garbage collection runs on the object it will lock a resource till then.

**Make sure you are really comparing Strings**

When comparing strings make sure you are comparing the contents and not the references. The == operator on a String compares the references and not the contents of the string. If you want to know if the contents of two strings are equal use the String.compareto or String.equals methods.

1. Exception Handling Standards:

### Common Problems and Best Practices with Java

Abstract. Many real-world Java applications fall short of Java’s excellent exception handling capabilities. In this article we point out several design and programming mistakes encountered in industry projects.

* The JVM rarely crashes, even as a result of serious problems in the code or runtime environment and thus in most cases successfully delivers valuable error information.
* Stack trace information support is built into the exception classes and does not need proprietary extensions.
* Checked exceptions are a way to emphasize through compiler support which exceptions of an API could and should be treated in the caller’s code.
  1. Some Common Problems
     1. Problem 1: Empty Catch Blocks

Empty catch blocks found in Java applications in many cases look like the following code snippets:

DateFormat format = DateFormat.getDateInstance(SHORT);

...

private final static String DEFAULT\_DATE\_STRING=”01.01.1900”;

...

try {

Date defaultDate=format.parse(DEFAULT\_DATE\_STRING);

...

} catch( ParseException pexc ) {

// Cannot happen: if it works once, it will always work!

}

The idea behind this code is that the only argument to the parse method is a constant chosen by the programmer to fit the used DateFormat, and therefore the ParseException will never be thrown in practice; hence, no code to deal with this kind of exception is needed.

Although this reasoning might appear sound at first glance, it is not, for two reasons:

* During maintenance of this code a programmer might switch to a different DateFormat and forget to change the constant accordingly
* The above code depends on the Locale. It might therefore be broken by installation on a different machine without any code changes.

In both cases a ParseException will be thrown and caught silently and the primary problem will remain undetected. If defaultDate is uninitialized at that time this will lead to a follow-up NullPointerException. Otherwise the program will continue with a wrong value of defaultDate. In both cases the original problem will be difficult to identify.

Therefore empty catch blocks in most cases are a serious threat to maintainability and reliability, two main criteria of software quality. The problem with empty catch blocks arises because the exception is *checked*, i.e. not inherited from RuntimeException. Programmers must either declare it in a throws clause or provide a catch block. Both choices seem inadequate for an exception that is not expected to occur.

In the Best Practice section, it is shown how this problem can be solved using unchecked exceptions in an adequate manner.

* + 1. Problem 2: Meaningless Throws Clauses

Many projects define a common base class for application-specific exceptions:

public class OurAppBaseException extends Exception{…}

This can be a good idea in order to define common behaviour of all exceptions. It is a design bug, however, to replace specific throws clauses in all methods of the application by blanket declarations such as this:

public void method1 throws OurAppBaseException {...}

public String method2 throws OurAppBaseException {

...

} catch( FileNotFoundException e ) {

throw new OurAppBaseException(e.getMessage( ));

}

}

It is the purpose of checked exceptions to provide specific information to the caller about exceptions that might be thrown in a method, and to have exception handling formally checked by the compiler.

The above practice undermines that purpose while formally maintaining the syntax of checked exceptions. To fully appreciate this argument note that if OurAppBaseException were unchecked (inherited from RuntimeException), all throws clauses could be dropped from all methods without changing their semantics. This is a legitimate choice (compare C++ !) but it should not be mixed up with adequate usage of checked exceptions.

* + 1. Problem 3: Loss of Stack Information

In many cases it is desirable to change an exception’s class by catching it and rethrowing a different exception object. One good reason for this, among others, is that in many cases low-level exceptions do not correspond to a method’s abstraction level:

private void init() throws InitializationException {

...

} catch( FileNotFoundException fnfexc) {

throw new InitializationException(fnfexc.getMessage());

}

}

Even in cases where this is a good idea, it is a problem that valuable information about the original exception will not be propagated. In the example above, the programmer tried to limit the problem by copying the original message text to the new exception. In this case other pieces of information, such as the original exception’s class and stack trace, are still lost. Its full information content can only be preserved by a technique called exception chaining (or nesting). Exception chaining is generically supported in Java 1.4.

* 1. Best Practice Guidelines

/\* Added additional content for Exception Handling \*/

Java distinguishes between two types of exception. Compiler insists that they be caught or explicitly re thrown. Unchecked or runtime exceptions and need not be caught (although they can be caught and propagate up the call stack in the same way as checked exception).

Unchecked, runtime exceptions are typically used to report serious unexpected errors that may indicate an error in the program’s logic. Run-time exceptions are usually throws because of programming errors, such as failed assertion, using an out-of-bound index, dividing by zero, or referencing a null pointer.

Use checked exceptions to report errors that may occur, however rarely, under normal program operations. Checked exceptions indicate a serious problem that should not occur under normal conditions. The caller must catch this exception. Depending upon the application, a program may be able to recover from a checked exception, that is, it doesn’t indicate a fundamental flaw in the program’s logic.

**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, 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 (and they almost always do not do so)

* + 1. Usage of Unchecked Exceptions

There is a misunderstanding that unchecked exceptions have no place in Java programming, it is emphasized that they stand equal beside checked exceptions. Just to make it clear again:

Note: Exceptions that signal an untreatable situation should be unchecked.

This concept is present throughout the Java core packages where NullPointerException, IndexOutOfBoundsException and other program bugs are inherited from RuntimeException and thus are unchecked. Program bugs should be fixed not treated in code.

As these exceptions can be thrown anywhere in your code but you don't want to declare them in all your methods, it's a good idea to make them unchecked. Going one step further you should define the following exception for your application:

public class ProgrammingException extends RuntimeException

This exception class is useful for all higher-level programming bugs. One important example is the empty catch block problem mentioned above:

try {

Date defaultDate=format.parse(DEFAULT\_DATE\_STRING);

...

} catch( ParseException pexc ) {

// If this exception is thrown, I got something wrong

throw new ProgrammingException(“bad init value”,pexc);

}

Compared to the original claim that the exception 'cannot be thrown', the idea remains the same but is expressed in a much more fault-tolerant way.

Note also in this example that exception chaining is used, so that the information associated with the original ParseException is not lost. Besides programming bugs, there are other obvious uses for unchecked exceptions.

Installation and configuration problems cannot not be treated in code either. Provide unchecked exception classes for them and use them as in this example:

try {

FileInputStream fin=new FileInputStream(configfilename);

...

} catch( FileNotFoundException fnfexc ) {

throw new InstallationException(“missing file”,fnfexc);

}

The FileNotFoundException and the ParseException above are checked because from the perspective of the Java core packages, they might be treatable depending on the application context. In the example contexts shown, it is clear that they are not treatable. Therefore they are chained into unchecked exceptions which simplifies their further handling and adds information to them. The fact that a programmer linked a certain FileNotFoundException to an InstallationException will certainly help the system administrator to analyze the problem once that exception chain has been written to a log file.

* + 1. Usage of Checked Exceptions

**Checked exceptions signal an exceptional situation which might be treatable.**

Because knowledge inside the called method is incomplete, the decision is left to the caller. A checked exception helps to ensure that the caller accepts this responsibility. This rule can be applied to the following situations:

* Exceptions which signal a problem at the level of a method's purpose (including 'business' level exceptions) are usually checked because by definition, their treatment is out of the method's scope and left to the caller.
* Badly formatted user input can be signalled by checked exceptions, because displaying an exception message and waiting for the user's retry is a kind of treatment.
* Any technical exception inside a server could be converted to a checked InternalServerException in the server's API, because the client might try to solve the situation by retrying or by calling a different server instance.

Let's start from a method which is used to buy shares in a stock trading application:

public void buyStock(int numOfShares, StockSymbol stock)

If this method can fail because there is not enough money on the cash account it must throw a checked exception:

public void buyStock(...) throws NotEnoughCashException

If we know from context that this method will only be called after a check of the cash account, this knowledge must be documented by the caller of the method with a ProgrammingException:

try {

trader.buyStock(num,symbol);

} catch( NotEnoughCashException necexc ) {

// Cash should have been checked before

throw new ProgrammingException(“no cash check”,necexc);

}

As much as an empty catch block, it would be a design bug to eliminate the NotEnoughCashException from the method's API due to this knowledge. This would lead to a fragile application with weak diagnostic capabilities.

If the application does not check cash before calling the buyStock() method, all the methods on the call stack need to propagate the NotEnoughCashException in their own throws clauses until it finally reaches the user interface and a message is displayed to the user. In this case the checked exception needs some reference to an error message. For many practical reasons, it usually is not adequate to display the hardcoded exception message text to the user; the exception message should only be used to communicate information to developers, whereas the display of error messages to users is a task of the presentation level, and should be implemented at that level.

A mechanism for identifying errors with sufficient granularity, so that the right error messages can be selected for them, should be implemented at the level of an application-specific checked exception base class, for example by including a member field for an error code that can be used as an index into a table of messages.

**Feedback: This is too descriptive section, rather than giving coding guidelines, it reflects more information about java fundamentals of exceptions.**

**Why: The above detailed descriptive section does not depict exact coding guidelines.**

**If any more details, with snippet is needed, then append it in separate document.**

**Here is suggestion, to remove about details with following points.**

* In a catch block avoid using the generic class Exception. For each try block use specific catch blocks based on what can go wrong in your code.
* Do not use Exception handling for anything other than exception handling like to control the flow of your program.
* Whenever you are using a throws clause always use the specific subclass of Exception like FileNotFoundException rather than using throws Exception.
* Use exception handling generously-Very little overhead is imposed by using exception handling mechanism unless an exception occurs. But when an exception occurs it imposes an overhead in terms of execution time.
* Always use the finally block to release the resources like a database connection, closing a file or socket connection etc. This  prevents resource leaks even if an exception occurs.
* When using method calls always handle the exceptions in the method where they occur, do not allow them to propagate to the calling method unless it is specifically required. It is efficient to handle them locally since allowing them to propagate to the calling method takes more execution time.
* Do not use Exception handling in loops. It is better to place loops inside try/catch blocks than vice versa. Here is an code snippet that gives bench mark.
  + 1. Exception Logging

An important part of exception handling is exception logging. This is particularly true for multi-tier and web applications where an exception on its way from the source to the user screen can traverse several virtual machines. In many cases the user who finally receives an error message has neither enough information nor interest to provide a useful error report to the administrators of all the systems contributing to a service. It is therefore essential that each system keep its own log, in which exceptions can be tracked as part of an ongoing quality assurance process.

To make these log files as readable as possible, it is very helpful to log an exception exactly once into a logfile. The right time to log an exception in Java is the latest possible, because the JVM most probably will not crash before and by chaining exceptions gain information content while they travel up the call stack. Exceptions should therefore only be logged when they

* are eventually being treated (caught with no rethrow)
* are leaving a physical tier / virtual machine through a remote call
* are leaving a logical tier that writes its own logfile

If additional 'early' log entries of the same exception are really needed, they should at least use a lower log level for distinction.

For programming exceptions the log should contain as much information about the state of the system as is available. For example, when errors are detected at the web presentation level, the log entry should contain, at least, a dump of the internal states of all of the objects available in the servlet API, incuding the request, response and session objects, including the contents of their attribute tables. The states of all other available objects, including EJB's, should be logged as well. We see no need for restraint here, since the cause for the problem could potentially be anywhere in the system, and a resolution of the problem has highest priority. Although the performance impact might not be negligible, it does not hit the system as long as it functions properly. Of course, a well-structured format for log entries makes extensive diagnostic information easier to examine.

1. Best Practices with Java
   1. Optimization techniques in Loops

* When using arrays it is always efficient to copy arrays using System.arraycopy () than using a loop.
* It is better to avoid accessing array elements in a loop the better option would be to use a temporary variables inside the loop and modify the array values out of the loop. It is fast to use a variable in a loop than accessing an array element.
* When using short circuit operators to test for loop termination tests always put the expression that will most likely evaluate to false at extreme left. This saves all the following expressions from being tested in case there is an && operator and if there are only || operators then put the expression which is most likely to evaluate to true in the extreme left.
* Avoid using try-catch inside the loops instead place the loops inside the try-catch for better performance
  1. Optimization techniques in Object creation
* Whenever you are done with an object make that reference null so that it is eligible for garbage collection.
* Whenever possible avoid using class variables, use local variables since accessing local variables is faster than accessing class variables.
  1. Optimization techniques in String and StringBuffer
* Create strings as literals instead of creating String objects using ‘new’ key word whenever possible
* Use String.intern() method if you want to add number of equal objects whenever you create String objects using ‘new’ key word.
* Use StringBuffer instead of String wherever possible for concatenation.
* To compare a constant string to a variable, use   
  if (“John”.equals(strName))..   
  This will work even if strName is null. However if you use it the normal way  
  i.e. strName.equals(“John”),it will give NullPointerException if strName is null.
  1. Optimization techniques in Serialization
* Use ‘transient’ key word for unnecessary variables that need not be read from/written into streams.
  1. Optimization techniques in I/O
* Reading and writing data using default behavior of some streams that is byte by byte read/write causes slow performance.
* Buffered input streams and buffered output streams increase performance.
  1. Optimization techniques in Collections
* Use Collections with proper initialization if you don’t want thread safe for the collection whenever you add/remove/access objects at end and middle of collection.
* Use ListIterator than Iterator and Enumeration for List types, gives flexibility of traversing both the sides.
  1. Optimization techniques in Synchronization
* Synchronizing on methods rather than on code block is slightly faster.
  1. Optimization techniques in using final keyword
* Use the final modifier on instance-variable definitions to create immutable internally accessible objects.
* Final classes can be faster, as they cannot be extended by any other class.
* Declare method arguments final, if they are not modified in the method.

References

[Ass02] Programming With Assertions,

http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html

[Blo01] Joshua Bloch:Effective Java,Programming Language Guide,Addison Wesley,2001

[Goe01] Brian Goetz:Exceptional practices,Part 2,

http//developer.java.sun.com/developer/technicalArticles/Programming/exceptions2/

[JLS00] Java Language Specification,

http://java.sun.com/docs/books/jls/second\_edition/html/exceptions.doc.html

[Log02] Java Logging APIs,http://java.sun.com/j2se/1.4/docs/guide/util/logging/index.html

[Ven98] Bill Venners: Exceptions in Java,

<http://www.javaworld.com/javaworld/jw-07-1998/jw-07-exceptions.html>

Error! Reference source not found.Error! Reference source not found.**Beginning Comments (Examples)**

/\*\*

\* Created on <<Month>> <<Day>>,<<Year>>

\*

\* Classname

\* Description of the class

\* @version $Revision: 1.3 $

\* @author [{full name}](mailto:%7bemail%7d).

\* Date

\*

\* @see SomeRelatedClass

\*

\* <PRE>

\*

\* <<<< THE EXACT COPYRIGHT NOTICE IS BEING WORKED ON>>>>

\*

\* </PRE>

\*/

[Click here to go back to the section](#_Beginning_Comments_1)

**Wrapping Lines (Examples)**

Here are some examples of breaking method calls:

someMethod(longExpression1, longExpression2, longExpression3,

longExpression4, longExpression5);

var = someMethod1(longExpression1,

someMethod2(longExpression2,

longExpression3));

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

longName1 = longName2 \* (longName3 + longName4 - longName5)

+ 4 \* longname6; // PREFER

longName1 = longName2 \* (longName3 + longName4

- longName5) + 4 \* longname6; // AVOID

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

//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) {

...

}

Line wrapping for if statements should generally use the 8-space rule since the conventional (4 space) indentation makes seeing the body difficult. For example:

//DON'T USE THIS INDENTATION

**Example of Poor Wraps**

public void mySampleMethod(String arg1, String arg2,

String arg3) {

:

:

if (condition1

&& (condition2

|| condition3

|| condition4)

|| !condition5) {

abc = 100;

}

:

:

}

//USE THIS INDENTATION INSTEAD

**Example of Good Wraps**

public void mySampleMethod(String arg1

, String arg2

, String arg3) {

:

:

if (condition1

&& (condition2 || condition3 || condition4)

|| !condition5) {

abc = 100;

}

:

:

}

if ((condition1 && condition2)

|| (condition3 && condition4)

||!(condition5 && condition6)) {

doSomethingAboutIt();

}

//OR USE THIS

if ((condition1 && condition2) || (condition3 && condition4)

||!(condition5 && condition6)) {

doSomethingAboutIt();

}

Here are three acceptable ways to format ternary expressions:

alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta

: gamma;

alpha = (aLongBooleanExpression)

? beta

: gamma;

[Click here to go back to the section](#_Wrapping_Lines_1)

**Javadoc Tags(More Information)**

**Tags for Methos**   
For each parameter to a method, include a param tag:  
  
        @param <name of parameter>      short description of parameter

For example,   
    @param size number of elements in the array

If a method has several parameters, the param tags should appear in the same order as the parameters are declared.  A method with no parameters will have no param tags. After the param tag(s), if any, include a return tag unless your return type is void (no return value):

@return    short description of return value

For example:   
    @return true if the value was found in the array

If your method throws an exception, you should also include an exception tag:

        @exception <name of exception>     description of circumstances under which the exception is thrown   
For example:   
    @exception NumberFormatException raised if the user's input is not in valid integer format

If your method throws more than one exception, they should appear in alphabetical order by exception name.  Exception tags should follow param and return tags.

**JavaDoc for Fields**

    /\*\*   
     \* The number of guest in a reserved product.  This variable must not be   
  \* negative or greater than 9.   
     \*/

public int totalGuest;

NOTE: The above is a guide for creating class javadocs. There are certain sections (, , ...) which are meant to be replaced by the individual developer as it applies to the class at hand.

Document all public (and mostly protected) members and methods in the class with JavaDoc except you did not add new functionality during overwriting a method (comming from an Interface or a base class). Especially note the following:

When Parameter can or cannot be null and also what it means when null is allowed.

When Return values can be or are never null.

Document side effects.

Useful is also when you mention if and where another method or the overwritten method is called. Because this is open-source it is not that important but when you only rely on the documentation it can avoid endless calls or other mistakes when you have to call the overwritten method.

Please add JavaDoc to every method/class.

Refer to this <http://java.sun.com/j2se/javadoc/writingdoccomments/> link for java doc guide.**Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.**

**Feedback: Append the above reference link in the section References.**

**Why: Basic contents are enough for normal java coding guidelines. No need to provide elaborative description.**

[Click here to go back to the section](#_Documentation_Comments)

**Declarations Placement (Examples)**

void myMethod() {

int int1 = 0; // beginning of method block

if (condition) {

int int2 = 0; // beginning of "if" block

...

}

}

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

for (int i = 0; i < maxLoops; i++) { ... }

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

int count;

...

myMethod() {

if (condition) {

int count = 0; // AVOID!

...

}

...

}

[Click here to go back to the section](#_Placement)