**JAVA CODING STANDARDS**

|  |
| --- |
|  |

*Purpose of this document*

This document describes a collection of standards, conventions and guidelines for writing Java code that is easy to understand, to maintain, and to enhance. s

*Important features of this document*

Existing standards from the industry are used wherever possible

The reason behind each standard is explained so that developers can understand why they should follow it.

These standards are based on proven software-engineering principles that lead to improved development productivity, greater maintainability, and greater scalability.

*Target Audience*

Professional Software developers who are involved in:

Writing Java code that is easy to maintain and to enhance

Increasing their productivity

*Why coding standards are important*

Coding standards for Java are important because they lead to greater consistency within code of all developers. Consistency leads to code that is easier to understand, which in turn results in a code, which is easier to develop and maintain. Code that is difficult to understand and maintain runs the risk of being scrapped and rewritten.

***The Prime Directive***

A project requirement may vary from the standards mentioned in this document. **When going against the standards, projects should make sure to document it.**

***1. Naming Convention***

**Use full English descriptors that accurately describe the variable/field/class/interface**

For example, use names like **firstName**, **grandTotal**, or **CorporateCustomer**.

**Use terminology applicable to the domain**

If the users of the system refer to their clients as Customer, then use the term Customer for the class, not client.

**Use mixed case to make names readable**

**Use abbreviations sparingly, but if you do so then use then intelligently and document it**

For example, to use a short form for the word “number”, choose one of **nbr**, **no** or **num**.

**Avoid long names (<15 characters is a good tradeoff)**

**Avoid names that are similar or differ only in case**

***2. Documentation***

**Comments should add to the clarity of code.**

**Avoid decoration, i.e., do not use banner-like comments**

**Document why something is being done, not just what.**

***Java Comments***

|  |  |  |
| --- | --- | --- |
| **Comment Type** | **Usage** | **Example** |
| **Documentation**  Starts with /\*\* and ends with \*/ | Used before declarations of interfaces, classes, member functions, and fields to document them. | /\*\*  \* Customer – a person or  \* organization  \*/ |
| **C style**  Starts with /\* and ends with \*/ | Used to document out lines of code that are no longer applicable. It is helpful in debugging. | /\*  This code was commented out by Ashish Sarin  \*/ |
| **Single line**  Starts with // and go until the end of the line | Used internally within member functions to document business logic, sections of code, and declarations of temporary variables. | **//** If the amount is greater  // than 10 multiply by 100 |

***3. Standards For Member Functions***

***3. 1 Naming member functions***

Member functions should be named using a full English description, using mixed case with the first letter of any non-initial word capitalized. The first word of the member function should be a verb.

**Examples**

openAccount()

printMailingList()

save()

delete()

This results in member functions whose purpose can be determined just by looking at its name.

***3.1.1 Naming Accessor Member Functions***

***3.1.1.1 Getters:*** member functions that return the value of a field / attribute / property of an object.

Use prefix “get” to the name of the field / attribute / property if the field in not boolean

Use prefix “is” to the name of the field / attribute / property if the field is Boolean

A viable alternative is to use the prefix ‘has’ or ‘can’ instead of ‘is’ for boolean getters.

**Examples**

getFirstName()

isPersistent()

***3.1.1.2 Setters:*** member functions that modify the values of a field.

Use prefix ‘set’ to the name of the field.

**Examples**

setFirstName()

***3.1.1.3 Constructors:*** member functions that perform any necessary initialization when an object is created. Constructors are always given the same name as their class.

**Examples**

Customer()

SavingsAccount()

***3.2 Member Function Visibility***

A good design requires minimum coupling between the classes. The general rule is to be as restrictive as possible when setting the visibility of a member function. If member function doesn’t have to be public then make it protected, and if it doesn’t have to be protected than make it private.

***3.3 Documenting Member Functions***

***3.3.1 Member Function Header***

Member function documentation should include the following:

What and why the member function does what it does

What member function must be passed as parameters

What a member function returns

Known bugs

Any exception that a member function throws

Visibility decisions (if questionable by other developers)

How a member function changes the object – it is to helps a developer to understand how a member function invocation will affect the target object.

Include a history of any code changes

Examples of how to invoke the member function if appropriate.

Applicable pre conditions and post conditions under which the function will work properly. These are the assumptions made during writing of the function.

All concurrency issues should be addressed.

- Explanation of why keeping a function synchronized must be documented.

When a member function updates a field/attribute/property, of a class that implements the **Runnable** interface, is not synchronized then it should be documented why it is unsynchronized.

If a member function is overloaded or overridden or synchronization changed, it should also be documented.

**Note:** It’s not necessary to document all the factors described above for each and every member function because not all factors are applicable to every member function.

***3.3.2 Internal Documentation:*** Comments within the member functions

Use C style comments to document out lines of unneeded code.

Use single-line comments for business logic.

Internally following should be documented:

**Control Structures** This includes comparison statements and loops

**Why, as well as what, the code does**

**Local variables**

**Difficult or complex code**

**The processing order** If there are statements in the code that must be executed in a defined order

***3.3.3 Document the closing braces*** If there are many control structures one inside another

***4.0 Techniques for Writing Clean Code:***

**Document the code** Already discussed above

**Paragraph/Indent the code:** Any code between the { and } should be properly indented

**Paragraph and punctuate multi-line statements**

**Example**

Line 1 BankAccount newPersonalAccount = AccountFactory

Line 2 createBankAccountFor(currentCustomer, startDate,

Line 3 initialDeposit, branch)

Lines 2 & 3 have been indented by one unit (horizontal tab)

**Use white space**

A few blank lines or spaces can help make the code more readable.

Single blank lines to separate logical groups of code, such as control structures

Two blank lines to separate member function definitions

**Specify the order of Operations:** Use extra parenthesis to increase the readability of the code using AND and OR comparisons. This facilitates in identifying the exact order of operations in the code

**Write short, single command lines** Code should do one operation per line So only one statement should be there per line

***5.0 Standards for Fields (Attributes / Properties)***

***5.1 Naming Fields***

**Use a Full English Descriptor for Field Names**

Fields that are collections, such as arrays or vectors, should be given names that are plural to indicate that they represent multiple values.

**Examples**

firstName

orderItems

If the name of the field begins with an acronym then the acronym should be completely in lower case

**Example**

sqlDatabase

***5.2 Naming Components***

Use full English descriptor postfixed by the widget type. This makes it easy for a developer to identify the purpose of the components as well as its type.

**Example**

okButton

customerList

fileMenu

newFileMenuItem

***5.3 Naming Constants***

In Java, constants, values that do not change, are typically implemented as *static final* fields of classes. The convention is to use full English words, all in upper case, with underscores between the words

**Example**

MINIMUM\_BALANCE

MAX\_VALUE

DEFAULT\_START\_DATE[[1]](#footnote-1)

***5.4 Field Visibility***

Fields should not be declared public for reasons of encapsulation. All fields should be declared private and accessor methods should be used to access / modify the field value. This results in less coupling between classes as the protected / public / package access of field can result in direct access of the field from other classes

***5.5 Documenting a Field***

Document the following:

**It’s description**

**Document all applicable invariants** Invariants of a field are the conditions that are always true about it. By documenting the restrictions on the values of a field one can understand important business rules, making it easier to understand how the code works / how the code is supposed to work

**Examples** For fields that have complex business rules associated with them one should provide several example values so as to make them easier to understand

**Concurrency issues**

**Visibility decisions** If a field is declared anything but private then it should be documented why it has not been declared private.

***5.6 Usage of Accesors*** Accessors can be used for more than just getting and setting the values of instance fields. Accesors should be used for following purpose also:

**Initialize the values of fields** Use lazy initialization where fields are initialized by their getter member functions.

**Example**

**/\*\***

**\*** Answer the branch number, which is the leftmost four digits of the full account

**\*** number. Account numbers are in the format BBBBAAAAAA.

**\*/**

protected int getBranchNumber()

{

if(branchNumber == 0)

{

// The default branch number is 1000, which is the

// main branch in downtown Bedrock

setBranchNumber(1000);

}

return branchNumber;

}

**Note:**

This approach is advantageous for objects that have fields that aren’t regularly accessed

Whenever lazy initialization is used in a getter function the programmer should document what is the type of default value, what the default value as in the example above.

***5.6.1 Access constant values*** Commonly constant values are declared as *static final* fields. This approach makes sense for “constants” that are stable.

If the constants can change because of some changes in the business rules as the business matures then it is better to use getter member functions for constants.

By using accesors for constants programmer can decrease the chance of bugs and at the same time increase the maintainability of the system.

***5.6.2 Access Collections*** The main purpose of accesors is to encapsulate the access to fields so as to reduce the coupling within the code. Collections, such as arrays and vectors, being more complex than single value fields have more than just standard getter and setter member function implemented for them. Because the business rule may require to add and remove to and from collections, accessor member functions need to be included to do so.

**Example**

|  |  |  |
| --- | --- | --- |
| ***Member function type*** | ***Naming Convention*** | ***Example*** |
| Getter for the collection | getCollection() | getOrderItems() |
| Setter for the collection | setCollection() | setOrderItems() |
| Insert an object into the collection | insertObject() | insertOrderItems() |
| Delete an object from the collection | deleteObject() | deleteOrderItems() |
| Create and add a new object into the collection | newObject() | newOrderItem() |

**Note**

The advantage of this approach is that the collection is fully encapsulated, allowing programmer to later replace it with another structure

It is common to that the getter member functions be *public* and the setter be *protected*

**Always Initialize Static Fields** because one can’t assume that instances of a class will be created before a static field is accessed

***6.0 Standards for Local Variables***

***6.1 Naming Local Variables***

Use full English descriptors with the first letter of any non-initial word in uppercase.

***6.1.1 Naming Streams***

When there is a single input and/or output stream being opened, used, and then closed within a member function the convention is to use **in** and **out** for the names of these streams, respectively.

***6.1.2 Naming Loop Counters***

A common way is to use words like **loopCounters** or simply **counter** because it helps facilitate the search for the counters in the program.

**i, j, k** can also be used as loop counters but the disadvantage is that search for i ,j and k in the code will result in many hits.

***6.1.3 Naming Exception Objects***

The use of letter **e** for a generic exception

***6.2 Declaring and Documenting Local Variables***

Declare one local variable per line of code

Document local variable with an endline comment

Declare local variables immediately before their use

Use local variable for one operation only. Whenever a local variable is used for more than one reason, it effectively decreases its cohesion, making it difficult to understand.It also increases the chances of introducing bugs into the code from unexpected side effects of previous values of a local variable from earlier in the code.

**Note**

Reusing local variables is more efficient because less memory needs to be allocated, but reusing local variables decreases the maintainability of code and makes it more fragile

***7.0 Standards for Parameters (Arguments) to Member Functions***

***7.1 Naming Parameters***

Parameters should be named following the exact same conventions as for local variable

***Name parameters the same as their corresponding fields (if any)***

**Example**

If **Account** has an attribute called **balance** and you needed to pass a parameter representing a new value for it the parameter would be called **balance** The field would be referred to as **this.balance** in the code and the parameter would be referred as **balance**

***7.2 Documenting Parameters***

Parameters to a member function are documented in the header documentation for the member function using the *javadoc @param* tag. It should describe:

What it should be used for

Any restrictions or preconditions

Examples If it is not completely obvious what a parameter should be, then it should provide one or more examples in the documentation

**Note**

Use interface as a parameter to the member function then the object itself.

***Standards for Classes, Interfaces, Packages, and Compilation Units***

***8.0 Standards for Classes***

***8.1 Class Visibility***

Use package visibility for classes internal to a component

Use public visibility for the façade of components

***8.2 Naming classes***

Use full English descriptor starting with the first letter capitalized using mixed case for the rest of the name

***8.3 Documenting a Class***

The purpose of the class

Known bugs

The development/maintenance history of the class

Document applicable variants

The concurrency strategy Any class that implements the interface **Runnable** should

have its concurrency strategy fully described

***8.4 Ordering Member Functions and Fields***

The order should be:

Constructors

private fields

public member functions

protected member functions

private member functions

finalize()

***9.0 Standards for Interfaces***

***9.1 Naming Interfaces***

Name interfaces using mixed case with the first letter of each word capitalized.

Prefix the letter “I” or “Ifc” to the interface name

***9.2 Documenting Interfaces***

The Purpose

How it should and shouldn’t be used

***10.0 Standards for Packages***

Local packages names begin with an identifier that is not all upper case

Global package names begin with the reversed Internet domain name for the organization

Package names should be singular

***10.1 Documenting a Package***

The rationale for the package

The classes in the packages

***11.0 Standards for Compilation Unit (Source code file)***

***11.1 Naming a Compilation Unit***

A compilation unit should be given the name of the primary class or interface that is declared within it. Use the same name of the class for the file name, using the same case.

***11.2 Beginning Comments***

/\*\*

\* Classname

\*

\* Version information

\*

\* Date

\*

\* Copyright notice

\*/

***11.3 Declaration***

|  |  |
| --- | --- |
| Class/interface documentation comment (/\*\*...\*/) | See Documentation standard for class / interfaces |
| 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. |
| 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. |

***11.4 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).

***11.5 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.

***11.5 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.

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 conventional (4 space) indentation makes seeing the body difficult. For example:

//DON'T USE THIS INDENTATION

if ((condition1 && condition2)

|| (condition3 && condition4)

||!(condition5 && condition6)) { //BAD WRAPS

doSomethingAboutIt(); //MAKE THIS LINE EASY TO MISS

}

//USE THIS INDENTATION INSTEAD

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;

***11.6 Declaration***

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

***11.7 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.

***11.8 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.

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!

...

}

...

}

***11.9 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() {}

...

}

**A blank line separates methods**

***11.10 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**

The if-else class of statements should have the following form:

if (condition) {

statements;

}

if (condition) {

statements;

} else {

statements;

}

if (condition) {

statements;

} else if (condition) {

statements;

} else {

statements;

}

**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:

for (initialization; condition; update) {

statements;

}

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 should have the following form:

while (condition) {

statements;

}

An empty while statement should have the following form:

while (condition);

**do-while Statements**

A do-while statement should have the following form:

do {

statements;

} while (condition);

**switch Statements**

A switch statement should have the following form:

switch (condition) {

case ABC:

statements;

/\* falls through \*/

case DEF:

statements;

break;

case XYZ:

statements;

break;

default:

statements;

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.

try {

statements;

} catch (ExceptionClass e) {

statements;

} finally {

statements;

}

**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

Before a block or single-line comment

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 Summary**

|  |  |  |
| --- | --- | --- |
| **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 [[2]](#footnote-2)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; |

**References:**

[www.java.sun.com](http://www.java.sun.com)

1. Java Coding Standards [↑](#footnote-ref-1)
2. Java Coding Standards [↑](#footnote-ref-2)