**ProCom**

**Java Coding Standards**

**Version 1.0**

**R**

|  |  |
| --- | --- |
| Doc. No.: |  |

**evision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 24 Sep 2009 | 0.01 | Initial Draft | Priyadarshan Patil |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# **1.Introduction**

## **1.1Purpose of this document**

The purpose of this document is to set coding standards for development in Java which will result in java code development of better readability and maintainability.

## **1.2Document organization**

The document is organized as follows:

* Section 1, *Introduction,* describes purpose for having coding standards
* Section 2,Naming standards, describes naming conventions to be followed
* Section 3,Code Layout describes about code layout strategies
* Section 4,Comments describes about how and where to place comments
* Section 5, Statements describes the standard way to write different types of statements in Java for better maintainability of code.
* Section 6, Good Coding Practice describes the about few coding standards

## **1.3Intended Audience**

The intended audience are all the team members of the project.

**2. Sections**

**1. Introduction**

1.1 Why Have Coding Standard?

Coding Standards establish a coherent base for development. Since code is more often reused than developed, having a coding standard eliminates some of the headache associated with maintaining code.

**2. Naming Standards**

There are generally accepted naming conventions employed in Java programming which should be followed. These are general rules to follow when naming classes and their attributes.

2.1 General Standards

Each class, attribute, and method name should be meaningful and descriptive of the information it contains and/or the behavior it performs (i.e., addNewCustomer, calculateTotalAmount). Names may not contain spaces nor may they contain abbreviations. All names must be fully spelled out, except in the case of acronyms. However, acronyms must be treated as normal words e.g. CORBA should be used as Corba. By treating acronyms this way, they fit into all existing naming standards e.g. asCorbaObject, corbaReference etc. If a name is comprised of multiple words, each new word is capitalized (i.e., BidLine, documentCode).

Different from abbreviations, acronyms are allowed.

Java class names should not contain the application name, nor should they include an abbreviation of the application name. The only exceptions to this are the application bootstrap class and the application specific basic servlet/class, which by definition must include the application name (spelled out, not abbreviated).

Avoid:

(If the application name is Phonebook)

PhonebookSearchConversation.java

PhonebookAssociateSearchBLO.java

PhonebookAssociateSearchDAO.java

Preferred:

PhonebookBootstrap.java

PhonebookBasicAction.java

SearchConversation.java

AssociateSearchBLO.java

AssociateSearchDAO.java

For maintainability, and due to operating system limitations, Java package names should be kept as short as possible.

File Names

Public classes must be named the same as the files in which they are defined. Class names should use the same capitalization. The filename for each file should be the public class name with the correct capitalization followed by ".java". (e.g. ContextManager.java should contain the public class ContextManager).

Class Names

Separate multi-word type names using capitalization with no intervening underscore (e.g. ProcessParticipant). Use mixed case for all public class names.

Method Names

Unlike class names, begin methods with a lowercase letter, and separate multi-word method names by starting each new word with an uppercase letter (e.g. calculateElectricalLoad()), except for the constructor, which must have the same name as the class name with the same capitalization (e.g. Settlement()). Use simple, clear names for methods to allow similar operations across different classes to be named consistently.

Instance methods should simply name the operation they perform on their associated object. Methods used to read or write properties (getters and setters), should start with "get" or "set" respectively.

private String getParticipantName(); private void setParticipantName( String newParticipantName\_ );

To indicate a boolean value returned by a method, name the method isXXXX. To indicate the setting of a boolean value, name the method setXXX(argument).

private boolean isValid(); private void setValid( boolean newValid\_ );

Since instance methods are always invoked in the context of an object (with the exception of static methods), their names should not encode the class name or the type of arguments expected. As an example, a String class instance function that returns the length of the string should be simply named length() - it should not be named stringLength(), since the context makes the meaning clear.

For validation type methods, name the methods as canXXXX() (e.g. canDisplay()).

Avoid the use of Object/object in any method name.

Static methods follow the same conventions as other methods.

2.5 Variable Names

Variable names always begin with a lower case letter (i.e., documentCode, name, identifier). Variables may be polymorphic across classes, yet must be unique to its own class (For example, Bid, BidLine, and Customer classes may each have the attribute 'identifier', which is unique to that class).

A variable name should not contain its own class name (Bid should not contain a variable 'bidIdentifier'. The variable should be named, 'identifier'. This will prevent subclasses from inheriting variables that refer to the parent class).

Always declare variables private. When exposing an instance variable to another object, declare the variable private and provide public or protected methods for accessing the variable.

Use descriptive names for variables starting with a lower case letter, and then starting each new word with an uppercase letter.

No Hungarian notation is allowed.

Avoid:

Variable names like sNumber.

Preferred:

Replace the above variable with a more detailed name like storeNumber.

Single character variable names must be replaced with detail names

Avoid:

String s = "This is a test";

Preferred:

Above must be replaced with String testString = "This is a test";

Method arguments variable(s) should end with the suffix, "\_".

These naming conventions enable the rapid location of variable definition, which increases understanding of the code.

public class MyClass

{

private String inputBuffer;

public final void setInputBuffer(final String newBuffer\_ )

{

inputBuffer = newBuffer\_;

} //end setInputBuffer

public final String getInputBuffer()

{

return inputBuffer;

} //end getInputBuffer

...

} //end class MyClass

2.6 Parameter and Local Variable Names

Separate multi-word local variable names using capitalization with no intervening underscore. Local variables should start in lowercase, and variable names should be descriptive of what the variable's purpose is.

\* 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.

\* Never create a temporary variable named "temp". Instead use a name that indicates what the variable is needed for.

Collections should be pluralized (i.e., ending in "s") as in customerAccounts or activeThreads

Setter parameters should be named starting with new and then the variable name

public void createMeterReading() { int counterIndex; Account[] customerAccounts; Account customerAccount; <statements> } //end createMeterReading

Avoid "hiding" names. Name hiding refers to the practice of naming a local variable, argument, or attribute the same as that of another one of greater scope. For example, if you have an attribute called participantName don't create a local variable or parameter called participantName. This makes code difficult to understand and more prone to bugs.

Also, when creating a local variable, always give it a descriptive name indicating its purpose. Never create a temporary variable named "temp".

2.7 Arrays

There are two styles of declaring an array in Java:

1. int integerArray[ ] = { 1, 2, 3, 4, 5 };

2. int[ ] integerArray = { 2, 4, 6, 8 };

Use the latter style; i.e., the int[ ] integerArray style.

2.8 Static Variable Names (i.e. Class Variable Names)

Follow the same rules for static variables as class variables. Always declare static variables private. When exposing a static variable to another object, declare the variable private and provide public or protected static methods for accessing the variable.

Even though you can access static variables (and static methods) through an instance, this should never be done. Always access static variables through public methods using the class.

private static String databaseConnection; public static String getDatabaseConnection { return databaseConnection; } //end getDatabaseConnection

2.9 Constant Names

Use upper case for all constant names, using underscores to separate multiple words.

static final int SPEED\_OF\_WATER = 3.10;

2.10 Exception Names

Exception names should end with the word "Exception". They should use mixed-case starting each new word with an uppercase letter, with no intervening underscore and start with initial capitalization.

2.11 Interface Names

Java Interface names should use mixed-case with the first letter of each word capitalized. Java 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.

2.12 File Suffixes

The following types of files will be used:

File Type Suffix

Java source .java

Java bytecode .class

**3. Code Layout**

A good layout strategy should accurately and consistently represent the logical structure of the code. It should make the code readable, and it should be easy to maintain. The rules in this section are designed to meet those criteria.

3.1 Class Headers

\* Write class headers on a single line if there is room for it

\* If not, break the line before the "extends" and "implements" keywords. Indent succeeding lines.

\* Put the opening brace left aligned on a line by itself

public class OvernightAirDeliveryItem

extends DeliveryItem implements Serializable

{

3.2 Method Headers

\* Write method headers by putting all parameters on one line.

\* If the parameter list won't fit on one line, break the list of parameters onto additional lines, and indent the parameters.

\* Put the opening brace left aligned on a line by itself.

public boolean isReadyToBeShipped(final boolean paidFor\_, final boolean orderFilled\_,

final boolean addressVerified\_, final boolean orderPackaged\_)

{

3.3 Indentation

\* Indentation is one tab. You can set the tab length in the development environment.

\* Use tabs for indentation and spaces for white space. Any tabs typed should not be converted to spaces, so that your formatting will be consistent if code is copied and pasted into other applications or viewed using a different development environment.

3.4 Wrapping Lines

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 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 2 tabs instead.

Here are some examples of breaking method calls:

methodName(longExpression, doubleExpression, floatExpression,

intExpression, stringExpression);

variable = methodName(longExpression,

function(longExpression, doubleExpression));

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.

longName = anotherLongName \* (myLongName + yourLongName - fooLongName) +

4 \* bigLongName; // PREFER

longName = anotherLongName \* (myLongName + yourLongName

- fooLongName) + 4 \* bigLongName; // 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 2 tabs.

//CONVENTIONAL INDENTATION

someMethod(int anArgument\_, Object anotherArgument\_, String yetAnotherArgument\_,

Object andStillAnother\_)

{

...

}

//INDENT 2 TABS TO AVOID VERY DEEP INDENTS

private static synchronized horkingLongMethodName(int anArgument\_,

Object anotherArgument\_, String yetAnotherArgument\_,

Object andStillAnother\_)

{

...

}

Line wrapping for if statements should generally use the 2-tab rule, since conventional (1 tab) indentation makes seeing the body difficult. For example:

//DON'T USE THIS INDENTATION

if ((myCcondition && yourCondition) ||

(anotherCondition && moreConditions) ||

!(thisCondition && thatCondition)) { //BAD WRAPS

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

}

//USE THIS INDENTATION INSTEAD

if ((myCcondition && yourCondition) ||

(anotherCondition && moreConditions) ||

!( thisCondition && thatCondition))

{

doSomethingAboutIt();

}

//OR USE THIS

if ((myCcondition && yourCondition) || (anotherCondition && moreConditions) ||

!( thisCondition && thatCondition))

{

doSomethingAboutIt();

}

3.5 White Space in the Code

\* Blank Lines

Blank lines improve readability by setting off sections of code that are logically related. One blank line should always be used between sections of a source file

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

\* Between functions

\* Between the local variables in a function and its first statement

\* Before a block (see section 5.1.1) or single-line (see section 5.1.2) 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++;

\* }

\* prints("size is " + foo + "\n");

\*

\* The expressions in a FOR statement should be separated by blank spaces.

Example:

for (myExpression; yourExpression; thisExpression)

3.6 Braces

Always use (curly) braces, even for blocks with only one statement. This removes one common source of bugs and eases maintenance:

1. You can insert or remove statements within a block without worrying about adding or removing braces

2. You never have a problem matching else clauses to if clauses.

Example:

Right

if ( clickedRow < currentIndex ) { myTopRow = currentIndex + 1; } //end if ClickedRow else if ( currentIndex < myTopRow ) { myTopRow = currentIndex; } //end else if

Wrong

if (clickedRow < currentIndex) myTopRow = currentIndex + 1; else if (currentIndex < myTopRow ) myTopRow = currentIndex;

This rule applies to the following constructs:

\* for, while and do-while loops

\* if-else statements

\* try, catch and finally clauses

\* synchronized blocks.

Note that the opening brace is always at the start of the next line by itself. This convention makes it easy to see where blocks of code begin and end since opening and closing braces are aligned, and helps prevent unbalanced braces that cause errors.

3.7 Line Lengths and Line Breaks

\* One statement per line.

\* Try to keep line lengths below 120 characters. This rule is not absolute; it is better to have a 130-character line than to break a statement.

\* If you must break a line, indent the continuation line(s).

\* If you must break a line, make it obvious by ending the first line with something that needs a continuation:

1. Break assignments after the assignment operator.

2. Break arithmetic and logical expressions after an operator.

3. Break the line to emphasize major sub-expressions.

4. Break method invocations after the opening parenthesis. If the parameter list still won't fit, break between each parameter or between each logical group of parameters if this seems better.

5. Break method declarations the same way, and put the opening brace on the next line, unindented.

6. If you need to break conditional expressions (e.g., in if or while-statements), follow rules 1 and 2 above, and put the opening brace on the next line, unindented.

\* Using extra variables to hold partial (intermediate) expressions can help you avoid line breaks and at the same time improve readability by making the code self-documenting.

Original condition

if ( clickTime - myPreviousClick < DOUBLECLICK\_TIME && mySelection == rowClicked ) { ... } //end if

Possible rewrite

final long clickInterval = clickTime - myPreviousClick; final boolean doubleClick = ( clickInterval < DOUBLECLICK\_TIME ); final boolean clickedSameRow = ( mySelection = = rowClicked ); if ( doubleClick && clickedSameRow ) { ... } //end if

3.8 Switch/case Layout

\* Indent each case one-tab level from the switch.

\* Use braces to partition case blocks.

\* Indent each statement of a case one-tab level from the case brace.

Switch ( some\_value )

{

case firstCase:

{

foo[ 0 ] = firstValue;

break;

} //end case

case secondCase:

{

foo[ 0 ] = secondValue;

break;

} //end case

case thirdCase:

{

foo[ 0 ] = thirdValue;

break;

} //end case

...

default:

{

foo[ 0 ] = defaultValue;

break;

} //end case

} //end switch

**4. Comments**

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.

4.1 Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments should 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 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. Block comments have an asterisk "\*" at the beginning of each line except the first.

/\*

\* Here is a block comment.

\*/

4.2 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 blank line should precede a single-line comment. Here's an example of a single-line comment:

if (condition)

{

/\* Handle the condition. \*/

...

}

4.3 Trailing Comments

Very short comments can appear on the same line as the code they describe, but 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. Avoid the assembly language style of commenting every line of executable code with a trailing comment.

Here's an example of a trailing comment:

if (a == 2) {

return TRUE; /\* special case \*/

} else {

return FALSE; /\* the alternate \*/

}

4.4 End-Of-Line Comments

The // comment delimiter begins a comment that continues to the new line. It 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 (x > 1) {

// Do a double flip.

...

}

else{

return false; // Explain why here.

}

4.5 Commented Code

To improve readability and maintainability, there should not be any commented lines of code in any class and/or method. All test/debugging code should be removed, and not commented out. Any code that is no longer needed should be removed, and not commented out.

**5. Statements**

5.1 Simple Statements

Each line should contain at most one statement. Example:

argument++; secondArgument--; // AVOID!

Do not use the comma operator to group multiple statements unless it is for an obvious reason. Example:

if (error)

{

Format.print(System.out, "error"), exit(1); //VERY WRONG!

}

5.2 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 singletons, when they are part of a control structure, such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

5.3 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 (condition)

{

statements;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Note - if statements always use braces {}. Avoid the following

error-prone form:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

statement;

5.5 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).

5.6 while Statements

A while statement should have the following form:

while (condition)

{

statements;

}

An empty while statement should have the following form:

while (condition);

5.7 do-while Statements

A do-while statement should have the following form:

do

{

statements;

}

while (condition);

5.8 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.

5.9 try-catch-finally Statements

A try-catch-finally statement should have the following format:

try

{

statements;

}

catch (ExceptionClass e)

{

statements;

}

finally

{

statements;

}

**6. Good Coding Practice**

6.1 Abstract Methods

Use abstract methods for classes which will not be instantiated i.e. which are to be super classes for some class hierarchy. Only define abstract methods for operations that need to be defined in derived classes. Interface methods (for example) are implicitly abstract. If a class implements at least one abstract method, the class should be explicitly declared as abstract.

6.2 Single Entry / Single Exit

Methods should have only a single exit point (exceptions counter this rule); i.e., there should only be one return statement within the method. This method coding style is very clear to follow. In addition, function scope tracing is facilitated by this style, since exit scope declarations have to be defined only once, at the end of the method body, rather than before multiple possible exit points.

6.3 Accessing Instance Variables

Within a class, use methods to access any private instance variables. The advantage of doing this is that the class code will not be deeply affected when implementation changes later on. In general, accessors should be defined final.

public class SavingProduct { private int interest; public final void setInterest(final int interestRate\_ ) { interest = interestRate\_; } //end setInterest public final float getInterest() { return interest; } //end getInterest } //end class SavingProduct

Try not to access a method through another method. (e.g. this.getParticipant().getUnits()). When practical, save the value of the first method in a local variable and access the second method with the local variable.

6.4 Visibility

Be as restrictive as possible when setting the visibility of a method. If a method doesn't have to be public, then make it protected, if it doesn't have to be protected, make it private. Use the table below as a guide.

Visibility Description Proper Usage

Public A public method can be invoked by any other method in any other object or class. When objects and classes outside of the class hierarchy in which the method is defined must access the method.

Protected A protected method can be invoked by any method in the class it was defined, any subclasses of that class, or any class in the same package as the defining class. When the method provides behavior that is needed internally within the class hierarchy but not externally.

package (or default) A method defined with package (or default) protection is accessible to all the other classes in the same package, but not outside that package. Don't use this method of protection.

Private A private method can only be invoked by other methods in the class in which it is defined, but not in the subclasses. When the method provides behavior that is specific to the class. Private methods are often the result of refactoring, also know as reorganizing, the behavior of other methods within the class to encapsulate one specific behavior.

6.5 Class Size/Length

Java classes must not exceed 3,000 Source Lines Of Code (SLOC) (this includes any documentation in the class). Any class longer than 3,000 SLOC must be refactored into smaller classes. The methods in the newly refactored classes should be based on their responsibility to ease maintaining and understanding the code.

6.6 Method Size and Two Page Rule

In order to maintain maximum readability and maintainability, methods must strive to conform to the "Two Page Rule", meaning that the method is not longer than 2 pages when printed. This is achievable using function decomposition, and using helper functions.

A method should do one thing, and the method name should reflect this accurately. If it does more, ensure this is reflected in the method name. If this leads to an ugly method name, reconsider the structure of your code. If you had a function named initializePanelManagerAndReadAccountList, the code would probably benefit from a split into methods named initializePanelManager and readAccountList.

Any method longer than 2 pages must be refactored into smaller methods that can be more easily maintained.

6.7 Exceptions

Generate exceptions for any error conditions not handled by Java or any other exception handlers. A rule of thumb is to generate errors early and often, meaning that it is better to be too extensive in error handling rather than assuming that another portion of the code will catch errors.

Handle exceptions at appropriate hierarchical level. Each level of an application hierarchy should cope with as many errors as it can. However, each level should pass errors it can't cope with to higher levels. A level should not call methods from any higher levels when dealing with an error. This introduces cyclic and unnecessary dependencies between levels of an application. Where applicable, group exceptions. It provides an exception hierarchy. All exceptions should be based on a hierarchy.

A sub-system of an application or a service that has (or may have in the future) several different types of errors should map each error with distinct names derived from a generic base. This allows a user to catch all exceptions at a specific level without having to list all the exception types.

A function may call a File Access framework that throws the following exceptions: FileNotFoundException, FileLockedException, and EndOfFileException. The calling function may decide to handle all the exceptions from the IOException or handle each one individually.

6.8 Bracing and Nesting Styles

For all classes, the declaration shall begin in the leftmost column. The left and right braces for the class must also appear in the leftmost column.

In general, matching beginning and ending braces should always be positioned in the same column. The beginning brace should appear on the line below the control statement. This makes identifying statement blocks easier. All declarations and code must be indented at least one level. A level is defined as one tab stop (i.e. 5 units). Ensure that your development environment is set up not to convert tabs to spaces. This will allow developers to work with code that is spaced to their preference. This will keep formatting the same across different machines and different applications if code is cut and pasted.

6.9 Variables (general usage)

Declare only one variable per line of code.

Right

private int myWidth = 150;

private int myHeight = 50;

Wrong

private int myWidth = 150, myHeight = 50;

All instance and class variables should be private.

Also, declare variables in semantic, not alphabetical, order.

Example

houseNumber streetName city zip inPostalCode outPostalCode

6.10 Instance Variables

All variables should be initialized before they are accessed. Initialization may occur in either a constructor or as a result of lazy initialization. Lazy initialization is the act of initializing variables in their getter methods. This ensures that a variable is not initialized until it is actually needed. This is especially helpful when the object to be retrieved is kept in persistent storage and could be very expensive to build.

6.11 Class Variables

All class variables (also known as static variables) must be initialized when the class is loaded. This can be accomplished through the use of lazy initialization or through static blocks of code, which are automatically run when a class is loaded. See below for an example of a static block of code:

Example of lazy initialization

public String getSpecialName() { if (specialName == null ) { specialName = "special"; } return specialName; } //end getSpecialName

Example of Constructor Initialization

public class Example { // static block of code { marketClearingPrice = ( float ) 15.25; } private static float marketClearingPrice; } //end class Example

6.12 Constants

Business related constants should be declared as static final attributes and always be accessed through class getter methods (methods declared as static). This allows the way a static value is defined to change in the future without affecting any objects that use the value.

6.13 Local Variables

Use local variables to represent one thing only. In other words, do not reuse local variables within a method. Whenever a local variable is used for more that one representation you make your code more difficult to understand. The chance of bugs introduced to your code by other developers also increases. Instead, declare a new descriptive variable for use.

Right

int accountIndex; ... for ( accountIndex = 0; myAccountList.size(); accountIndex++ ) { ... } ... // Swap elements int arrayIndex; arrayElement= ageList[ 0 ]; ageList[ 0 ] = ageList[ 1 ]; ageList[ 1 ] = arrayElement;

Wrong

int i; ... for ( i = 0; i < myAccountList.size(); ++i ) { ... } ... // Swap elements: i = someArray[ 0 ]; someArray[ 0 ] = someArray[ 1 ]; someArray[ 1 ] = i; ...

The two uses of i above on the right have nothing to do with one another. Creating unique variables for each purpose makes your code more readable.

6.14 Collections

Each attribute collection should implement the following getters and setters:

Method Type Naming Convention Example

Getter for the collection getCollection() getMeterReadings()

Setter for the collection setCollection() setMeterReadings()

add an object into the collection addCollection() addMeterReading()

remove an object from the collection removeCollection() removeMeterReading()

6.15 Initialization

\* All variables, including instance and class variables, should be initialized at the point of declaration if possible. Even though all Java declarations have default initialization values (0, null, false), spell this out explicitly.

\* Java allows initialization of arrays using the same syntax as C and C++, by enclosing a comma-delimited set of values in braces.

\* Java 1.1 allows initializer blocks among the declarations. An initializer block is a section of code enclosed in braces. There are two kinds of initializer blocks: static and instance.

Static initializer blocks are executed the first time a class is loaded by a ClassLoader. Error handling is difficult within a static initializer, and as a result static initializer blocks should be avoided if possible. During static initialization (class initialization), things happen in the following order:

1. Class initialization of the superclass is performed, unless it has been done earlier.

2. Static variables are initialized and static initializer blocks are executed. This happens in the order they are listed, from top to bottom. Instance variables, instance initializer blocks and methods don't figure into this.

Note that static and instance initializer blocks are allowed in Java 1.1. Static initializer blocks are executed in order when the class is first instantiated; instance initializer blocks are executed in order after the superclass constructor runs, but before the class constructor runs.

Instance initializer blocks are executed whenever a class is instantiated. During object initialization (instance initialization), things happen in the following order:

1. If this is the first time the class is instantiated, all the class (static) initialization takes place.

2. We enter a constructor. If we have not specified a constructor, the compiler supplies a default constructor with no arguments automatically.

3. The superclass constructor is called. If your constructor does not explicitly invoke a superclass constructor, the default (argument-less) superclass constructor is called anyway.

4. All instance variables are initialized and instance initializer blocks are executed. This happens in the order they are listed, from top to bottom. Class variables, class initializer blocks and methods don't figure into this.

Use initializer blocks to perform any initialization that can't be performed by direct variable initialization; put each initializer block immediately following the variable in question. In the examples below, note that the array can be initialized without using an initializer block, while the vector object requires one because of the calls to the addElement method.

private List listofSomething = new ArrayList();

{ // Instance initializer block

listofSomething.add( someObject );

listofSomething.add( anotherObject );

}

private static int[] multipliers = {

5, 4, 3, 2, 7, 6, 5, 4, 3, 2 };

private static MyClass myClass = new MyClass();

static

{

//Static initializer block

myClass.setValue( someValue );

}

6.16 Wrappers for Primitives

For attributes only, wrappers should be used to wrap primitives. However, this can impact methods that, for example, use boolean logic. Imagine a class that has the following methods defined:

public boolean isDirty()

{

return new Boolean( true );

}

public boolean canSave()

{

return !this.isDirty() || this.isValid();

}

The #canSave() method will not compile because the operator, ||, only works on primitive booleans, not wrapper Booleans. #canSave() must be rewritten so that the #booleanValue() is extracted from the Boolean wrapper returned by #isDirty(), thus :

public Boolean canSave()

{

return new Boolean ( !isDirty().booleanValue

|| isValid().booleanValue() );

}

Other variables, parameters and return types can use primitive types if desired.

6.17 Import Statements

If a class is using three or more import statements from the same package, then the whole package must be imported instead of the individual classes.

For example:

import com.tru.commom.utility.DateUtility;

import com.tru.commom.utility.StringUtility;

import com.tru.commom.utility.CommonUtility;

All the above import statements must be replaced with

import com.tru.commom.utility.\*;

6.18 System Statements

System.out and System.err lines of code must be removed from all methods.

6.19 Use of String vs. StringBuffer

Strings (ex: SQL queries) containing literals, and/or variables with known values, should be built using the String class, not StringBuffer, regardless of how many concatenations with the "+" operator there are. This will improve application performance, as the string literals will be concatenated into a String object at compile time, not at run time.

To improve readability of code, do not concatenate each word of a string literal. Instead, group as many words together as you can before using the "+" operator to concatenate the string literal with variables and/or other string literals.

Avoid:

String sql = "SELECT " + "ITEM\_INFORMATION" + ","

+ "SKN" + "," + " ITEM\_INFORMATION" + "."

+ "CATEGORY" + "," + "ITEM\_INFORMATION"

+ "." + "SUBCATEGORY" + ","

+ "ITEM\_INFORMATION" + "."

+ "SECURITY\_FLAG"

+ " FROM " + dbOwner + "ITEM\_INFORMATION" + ", "

+ dbOwner + "AREA\_INFORMATION"

+ " WHERE " + "ITEM\_INFORMATION" + "."

+ "VENDOR\_NUMBER" + "="

+ "VENDOR\_NAMES" + "." + "VENDOR"

+ " AND " + "ITEM\_INFORMATION" + "."

+ "SKN" + "=" + "AREA\_INFORMATION" + "."

+ "SKN\_AREA";

Preferred:

// declared at the class level, or in a constant file

protected static final String SELECT\_STATE\_SQL =

"select label, value from " + dbOwner + "STATE\_TABLE where value like ?";

// used in your DAO method as follows

PreparedStatement myStatement = null;

try

{

myStatement = connection\_.prepareStatement(SELECT\_STATE\_SQL);

// the remaining part of your try block

}

catch (SQLException sqle)

{

throw new DataAccessException("Error executing query.", sqle);

}

finally

{

close(resultSet);

close(myStatement);

}

// the remaining part of your method

Strings containing values that are not known in advance, cannot be resolved at compile time. Instead, these strings are resolved at run time, and should therefore be built using the StringBuffer class, not String. When values are not known, it is more efficient to use StringBuffer.

Avoid:

public ListResultBean getItemInformation(final String dbOwner\_)

{

String sql = "SELECT ITEM\_INFORMATION.SKN, ITEM\_INFORMATION. CATEGORY, ITEM\_INFORMATION.SUBCATEGORY, ITEM\_INFORMATION.SECURITY\_FLAG FROM " + dbOwner\_ + "ITEM\_INFORMATION," + dbOwner\_ + "AREA\_INFORMATION WHERE ITEM\_INFORMATION.VENDOR\_NUMBER = VENDOR\_NAMES.VENDOR AND ITEM\_INFORMATION.SKN = AREA\_INFORMATION.SKN\_AREA AND ITEM\_INFORMATION.CATEGORY = ?";

// the remaining part of your method

}

Preferred:

public ListResultBean getItemInformation(final String dbOwner\_)

{

StringBuffer query = new StringBuffer(310);

query.append("SELECT ITEM\_INFORMATION.SKN, ITEM\_INFORMATION. CATEGORY, ITEM\_INFORMATION.SUBCATEGORY, ITEM\_INFORMATION.SECURITY\_FLAG FROM ");

query.append(dbOwner\_);

query.append("ITEM\_INFORMATION, ");

query.append(dbOwner\_);

query.append("AREA\_INFORMATION WHERE ITEM\_INFORMATION.VENDOR\_NUMBER = VENDOR\_NAMES.VENDOR AND ITEM\_INFORMATION.SKN = AREA\_INFORMATION.SKN\_AREA AND ITEM\_INFORMATION.CATEGORY = ?" );

// the remaining part of your method

}

When using StringBuffer, you should always initialize the StringBuffer object to a default value. This will improve the performance of your code. Without initialization, the buffer defaults to a size of 16. Each time the buffer exceeds this length, the size is doubled, a new buffer is created, and the contents are copied to the new buffer. This requires extra processing time. You should set the size of the buffer during initialization to be as close to the maximum size that you will need. If the maximum size cannot be determined, you should initialize the buffer using a prime number that is close to the size you think the buffer will be.

Avoid:

public ListResultBean getItemInformation(final String dbOwner\_)

{

StringBuffer query = new StringBuffer();

query.append("SELECT SKN, CATEGORY FROM ");

query.append(dbOwner\_);

query.append("ITEM\_INFORMATION WHERE SKN = ?");

// the remaining part of your method

}

Preferred:

public ListResultBean getItemInformation(final String dbOwner\_)

{

StringBuffer query = new StringBuffer(65);

query.append("SELECT SKN, CATEGORY FROM ");

query.append(dbOwner\_);

query.append("ITEM\_INFORMATION WHERE SKN = ?");

// the remaining part of your method

}

Preferred (Alternate):

public ListResultBean getItemInformation(final String dbOwner\_)

{

StringBuffer query = new StringBuffer("SELECT SKN, CATEGORY FROM ");

query.append(dbOwner\_);

query.append("ITEM\_INFORMATION WHERE SKN = ?);

// the remaining part of your method

}

6.20 toString() Method

Remove any toString() methods in Bean classes used for debugging.

6.21 Static Methods

Use static methods in the Java API where ever applicable for efficiency. Do not instantiate objects for converting one data type to another when a static method is available to do so.

Ex: new Long(resultSet.getLong(TransferDatabaseConstants.CARTON\_ORIGIN\_LOCATION\_CODE)).toString()

can be written as

Long.toString(resultSet.getLong(TransferDatabaseConstants.CARTON\_ORIGIN\_LOCATION\_CODE))

this will avoid the overhead of creating a new object

6.22 Error Handling

Ensure the proper error handling by enclosing the original exceptions when "throwing" the custom exceptions from the catch blocks. Otherwise, it will not help finding the issue in case of error.

Ex: the following code does not properly handle the exception

catch(SQLException sqle)

{

throw new DataAccessException("Could not insert record " + sqle);

}

should be changed to

catch(SQLException sqle)

{

throw new DataAccessException("Could not insert record ", sqle);

}

Don't use printStackTrace() on the exceptions, use proper error handling to log this to application log files.

6.23 Use of FINAL keyword for Variables

One fundamental principle of programming is that, generally, it is best to swap a logic error for a compiler error. Compiler errors tend to be found in seconds and are corrected just as fast. Syntax errors are a good example. A missing semicolon can make things confusing. If the compiler error is something particularly cryptic, the resolution may take as long as a couple of minutes to discover.

Logic errors, on the other hand, seem to have minds of their own, constantly evading detection and dodging your efforts to pin down their cause. They can easily take a thousand times more effort to solve than the worst compiler errors. Worst of all, many logic errors are not found at all and occur only intermittently in sensitive places, which causes your customers to scream for a fix. Logic errors often require you to throw thousands of man-hours at them, only to finally discover that they are minor typos.

The Java keyword final can be instrumental in turning thousands of logic errors into compiler errors without too much effort.

In general, you should always use the keyword final on all variables, regardless of scope (i.e. class, method, etc), that are only assigned one (1) time (i.e. their value is constant). By using the final keyword on these types of variables, you substitute a compile time error for a logic error. This allows you to find errors quickly, rather than hunting through many lines of code when a runtime issue occurs. Additionally, you should also use the keyword final on all method arguments/parameters that do not change in the method.

**7. Revision History**

Name Date Description

Priyadarshan Patil 23 September 2009 Coding standards to be followed for ProCom development

**References:**

1. http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
2. <http://geosoft.no/development/javastyle.html>