**New Codington Festival Online Project -   
Application Development Standards Pilot 2**

**Project:** New Codington Festival Online Project

**System:** Festival Event Registration System – Release 1

Table of Contents

[1 Introduction 3](#_Toc319330770)

[2 Java Coding Standards 3](#_Toc319330771)

[2.1 Code Structure and Format 3](#_Toc319330772)

[2.1.1 Class Structure and Organization 3](#_Toc319330773)

[2.1.2 Class Headers 6](#_Toc319330774)

[2.1.3 Method Headers 7](#_Toc319330775)

[2.1.4 Indentation 7](#_Toc319330776)

[2.1.5 White Space and Blank Lines 8](#_Toc319330777)

[2.1.6 Aligning Assignments Statements 9](#_Toc319330778)

[2.1.7 Variable Declarations 9](#_Toc319330779)

[2.1.8 Line Lengths and Line Breaks 10](#_Toc319330780)

[2.1.9 Method Size 11](#_Toc319330781)

[2.1.10 Bracing and Nesting 12](#_Toc319330782)

[2.1.11 Switch/Case Layout 14](#_Toc319330783)

[2.1.12 Conditionals 14](#_Toc319330784)

[2.1.13 Loops 15](#_Toc319330785)

[2.2 Java Naming Standards 16](#_Toc319330786)

[2.2.1 General Standards 16](#_Toc319330787)

[2.2.2 Package Names 17](#_Toc319330788)

[2.2.3 Class Names 17](#_Toc319330789)

[2.2.4 Abstract Class Names 17](#_Toc319330790)

[2.2.5 Interface Names 17](#_Toc319330791)

[2.2.6 Interfaces and Implementations 18](#_Toc319330792)

[2.2.7 Method Names 18](#_Toc319330793)

[2.2.8 Variable Names 19](#_Toc319330794)

[2.2.9 Instance Variable Names 19](#_Toc319330795)

[2.2.10 Local Variable Names 19](#_Toc319330796)

[2.2.11 Constant Names 20](#_Toc319330797)

[2.3 Programming Styles 20](#_Toc319330798)

[2.3.1 Java Programming Style 20](#_Toc319330799)

[2.3.2 Abstract Methods 20](#_Toc319330800)

[2.3.3 Instance Variables 20](#_Toc319330801)

[2.3.4 Static Variable Names (i.e. Class Variable Names) 21](#_Toc319330802)

[2.3.5 Visibility Types 22](#_Toc319330803)

[2.3.6 Constants 22](#_Toc319330804)

[2.3.7 Local Variables 23](#_Toc319330805)

[2.3.8 Notes on Specific Keywords and Constructs 23](#_Toc319330806)

[2.3.9 Using Java Collections 25](#_Toc319330807)

[2.3.10 Generics 26](#_Toc319330808)

[2.3.11 BigDecimal for Financial Calculations 27](#_Toc319330809)

[2.3.12 Annotations 27](#_Toc319330810)

[2.4 Documentation in Java Programs 28](#_Toc319330811)

[2.4.1 Comment Types 28](#_Toc319330812)

[2.4.2 Project Documentation Standards 29](#_Toc319330813)

[2.4.3 General JavaDoc Usage 31](#_Toc319330814)

[2.5 Good Coding Practices 35](#_Toc319330815)

[2.5.1 Decomposition 35](#_Toc319330816)

[2.5.2 Readability 35](#_Toc319330817)

[2.5.3 Comments 35](#_Toc319330818)

[2.5.4 Interfaces and Dependencies 36](#_Toc319330819)

[2.5.5 Process 36](#_Toc319330820)

[2.5.6 Exception Handling 36](#_Toc319330821)

[2.6 References 36](#_Toc319330822)

[3 Java Naming Standards 37](#_Toc319330823)

[3.1 Web Application Style 37](#_Toc319330824)

[3.1.1 Configuration Files 37](#_Toc319330825)

[3.1.2 Presentation Tier 38](#_Toc319330826)

[3.1.3 Business Tier 39](#_Toc319330827)

[3.1.4 Data Access Tier 42](#_Toc319330828)

[3.2 Batch Application Style 42](#_Toc319330829)

[3.2.1 Configuration Files 42](#_Toc319330830)

[3.2.2 Job Tier 42](#_Toc319330831)

[3.3 Integration Application Style 43](#_Toc319330832)

[3.3.1 Configuration Files 43](#_Toc319330833)

[3.3.2 Integration Tier 43](#_Toc319330834)

# Introduction

This document provides the Application Development Standards for the New Codington Festival Online Project / FERS System. This serves as a guide to the standards and procedures to be used for application and technical architecture development.

# Java Coding Standards

Good standards help to ensure that code is consistent across developers and easy to understand. Additionally, standards help to ensure application code is flexible and maintainable in the long run. The following sections define the Java coding standards and best practices.

## Code Structure and Format

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.

### Class Structure and Organization

The code units within the class should be organized as follows:

//Copyright information

package package\_name

import class\_names

import static static\_member\_name

class header

public class class\_name

constants (using static final)

static variables (using static)

private instance variables

private constructors

public instance methods

getters / setters

instance methods

protected instance methods

getters / setters

instance methods

private instance methods

getters / setters

instance methods

static methods

inner class header

private class class\_name

**Notes:**

1. All instance and class variables should be private. Have public getters and setters for those that you wish to make accessible outside the class.
2. Define instance methods in a logical order (as determined by the developer) for each public, protected, and private section.
3. Only import the classes that are necessary for the class being written. (e.g. If your program references the class Component, you should only import java.awt.Component rather than import java.awt.\*)
4. Getter and setter functions of the same scope (e.g. public) should be defined adjacent to each other in pairs.
5. Use of finalize is discouraged as relying on the garbage collection process is unpredictable.

For example:

/\*\*

\* Visitor.java Feb 23, 2012

\*

\* © 2012 Accenture. All Rights Reserved.

\* The trademarks used in these materials are the properties of their respective owners.

\* This work is protected by copyright law and contains valuable trade secrets and

\* confidential information.

\*/

package com.accenture.test;

/\*\*

\* The visitor Class represents a visitor entity in the system.

\* A visitor has a first and last name. The first name can be validated

\* to verify that the name is less than the maximum length.

\*

\* @author <<author name>>

\*

\*/

public class Visitor {

//

// constants

//

public static final int MAXIMUM\_NAME\_LENGTH = 40;

//

// instance variables

//

private String firstName;

private String lastName;

//

// constructor

//

public Customer() {

}

//

// instance methods

//

/\*\*

\* The isFirstNameValid method verifies if the firstName is less than the

\* maximum length of a Name field.

\*

\* @return Boolean true if the first Name is valid, false if it is not.

\*/

public boolean isFirstNameValid() {

return (this.firstName.length() <= Visitor.MAXIMUM\_NAME\_LENGTH);

}

//

// getters and setters

//

/\*\*

\* Returns the value of the instance's first name variable.

\*

\* @return String Visitor's first name

\*/

public String getFirstName() {

return this.firstName;

}

/\*\*

\* Sets the value of the instance's first name variable.

\*

\* @param String

\* Visitor's first name

\*/

public void setFirstName(String newFirstName) {

this.firstName = newFirstName;

}

/\*\*

\* Returns the value of the instance's last name variable.

\*

\* @return String Visitor's last name

\*/

public String getLastName() {

return this.lastName;

}

/\*\*

\* Sets the value of the instance's last name variable.

\*

\* @param String

\* Visitor's last name

\*/

public void setLastName(String newLastName) {

this.lastName = newLastName;

}

}

### Class Headers

The class header consists of the package name, the import section and copyright information section. In addition, there should be a comment block describing the class, author, and date of creation. This comment block should be in a format suitable for generating JavaDoc documentation. This is followed by the actual class declaration and opening brace.

In the import section, list each imported class explicitly. Import statements should be sorted with the most fundamental packages first (i.e. java, javax, org, com), and grouped with associated packages together and one blank line between groups. For example java. comes first, javax. comes next, then everything else and imports within each group are in lexicographic order. Static imports must be at the end of a group and in lexicographic order amongst themselves.

Example:

|  |  |
| --- | --- |
| Right | Wrong |
| import java.awt.Frame; import java.awt.Graphics; import java.awt.event.WindowAdapter; import java.awt.event.WindowEvent; import java.applet.AppletContext; | import java.awt.\*; import java.awt.event.\*; import java.applet.\*; |

The class comment should follow this format:

Example:

/\*\*

\* This class defines all the common constants used by the

\* frameworks

\*

\* @author <author\_name>

\*/

In the class comment author name should be added. Author is not mandatory, it can be dropped of you want to rely on version control system for that.

### Method Headers

* Write method header in a single line. Wrap method as appropriate.
* The "throws" clause should appear after the parenthesis or on its own line. If it appears on its own line it should be indented four spaces.
* Put the opening brace left aligned on the same line.
* Method headers should start with a JavaDoc header describing the method, as the following format defines:

/\*\*

\* <Method description>

\*

\* @param parameter-name description

\* @return return-description

\* @throws exception-decription

\* @see com.accneture.sample.SuperObject#get

\*/

Example

public boolean isReadyToBeShipped(boolean paidFor,

boolean orderFilled, boolean addressVerified) throws

MySpecificException {

### Indentation

* Indentation should be set one tab shift.
* Use tabs for indentation and spaces for white space. Indentation should be similar in java code, HTML, JSP and javascript files.

### White Space and Blank Lines

Add one space in the following places:

* between operators
* after comma in method declarations and invocations
* after semicolons in for-loops
* before and after the assignment operator
* between a keyword and a parenthesis.

No space in the following places:

* between a method name and the opening parenthesis
* between opening and closing parentheses in a function declaration or invocation with an empty parameter list
* between opening and closing square brackets in an array declaration where the number of elements is not specified

This example illustrates the above rules

public void invertFirstName(String aName) {

String invertName[];

int nameLength = aName.length() - 1;

char nameElement;

for (int counter = (nameLength - 1); counter >= 0; counter--) {

nameElement = aName.substring(counter, 1);

invertName[nameLength - counter] = nameElement;

}

}

Use blank lines to separate "paragraphs" of related code lines.

Use a single blank line to separate logical groups of code (like control statements).

Use single blank lines to separate method definitions.

Example :

public void addVisitor(Visitor newVisitor) throws InvalidVisitorException() {

if (newVisitor.isValidated()) {

this.visitorList().add(newVisitor);

} else {

throw new InvalidVisitorException();

}

}

public Visitor getVisitor(int VisitorKey) {

return this.VisitorList().getElement(identifier);

}

### Aligning Assignments Statements

* Align the = of related assignment statements. This sets them off as a group and shows clearly that they are related.
* Do not align the = of unrelated statements. Such alignment gives an erroneous impression of relatedness.

Example:

|  |  |
| --- | --- |
| Right | Wrong |
| panelWidth = 90; panelHeight = 30; selectedIndex = 0; finalIndex = 12; | panelWidth    = 90; panelHeight   = 30; selectedIndex = 0; finalIndex    = 12; |

### Variable Declarations

Declare only one variable per line of code.

Example:

|  |  |
| --- | --- |
| Right | Wrong |
| private int myWidth = 150; private int myHeight = 50; | private int myWidth = 150, myHeight = 50; |

Visibility of all instance and class variables should be kept to the minimum. Relevant justification in the comments must be provided when the variables are not declared private. Also, declare variables in semantic, not alphabetical, order.

Example

houseNumber

streetName

city

zip

inPostalCode

outPostalCode

Arrays must be created in the following style:

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

Use the int[] integerArray style.

### Line Lengths and Line Breaks

Follow these rules on line lengths and breaks:

* One statement per line.
* Try to keep line lengths below 80 characters.
* 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:
  + Break assignments after the assignment operator.
  + Break arithmetic and logical expressions after an operator.
  + Break the line to emphasize major sub-expressions.
  + 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.
  + Break method declarations the same way, and put the opening brace on the same line,
  + 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 same line, un-indented.
* 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.
* Do not use ternary statements (x ? y : z) - these break the "one statement per line" rule and are difficult to read and understand.

### Method Size

In order to maintain maximum readability and maintainability, methods should strive to have methods that are 25 or less statements long. This is achievable using function decomposition and using helper methods.

A method should preferably do one thing, and the method name should reflect this accurately. If it does more, ensure that 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 initPanelManagerAndReadAccountList, the code would probably benefit from a split into methods named initializePanelManager and readAccountList.

Example: Too Long

public void openAndPrintFile(String aFileName) {

TextFile textFile = null;

try {

textFile = new TextFile();

textFile.open(aFileName);

} catch(IOException e) {

String errorString = "Error opening file " + aFileName;

errorString = errorString + " " + e.toString();

System.out.println(errorString);

}

if(textFile.isOpen()) {

while(!textFile.atEOF()) {

String line = textFile.readln();

System.out.println(line);

}

textFile.close();

}

}

Example: Decomposed Method

public TextFile openFile(String aFileName) throws IOException {

TextFile aFile = new TextFile();

textFile.open(aFileName);

return aTextFile;

}

public void printFile(TextFile aFile) {

if(textFile.isOpen()) {

while(!textFile.atEOF()) {

String line = textFile.readln();

System.out.println(line);

}

textFile.close();

}

}

### Bracing and Nesting

For all classes, the declaration shall begin in the leftmost column. The opening brace should be on the same line and the closing brace should begin a newline, indented to match its corresponding opening statement. In case of an empty statement the “}” should appear immediately after the “{“. All declarations and code must be indented at least one level (four spaces).

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 | Wrong |
| if (clickedRow < currentIndex) {  myTopRow = currentIndex + 1;  } else if (currentIndex < myTopRow) {  myTopRow = currentIndex;  **}** | 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

Programming constructs must have the following form:

**For loops:**

for (<control variable initialization>;

<test condition>; <increment control variable>) {

<statements>

}

**While loops:**

while (<condition>) {

<statements>

}

**If constructs:**

if (<condition>) {

<statements>

}

**If ... else constructs:**

if (<condition>) {

<statements>

} else {

<statements>

}

**If ... else constructs:**

if (<condition>) {

<statements>

} else if (<condition>) {

<statements>

} else {

<statements>

}

**try ... catch constructs:**

try {

<statements>

} catch (SomeException variableName) {

<statements>

}

### Switch/Case Layout

Never allow flow control to "fall through" from one case label to the next by omitting the break statement. If you feel an urge to do this because of common code, consider factoring out the common code in a new helper method.

Formatting switch statements:

* Indent each case one indentation level from the switch.
* Indent the statements that belong to a switch, one statement to a line.

switch (someValue) {

case case1:

value1++;

break;

case case2:

value2++;

break;

case case3:

value3++;

break;

...

}

### Conditionals

Complex conditions can be hard to read and understand. One way to alleviate this is by using extra boolean variables. In the first fragment below, the meaning of the test is not obvious; in the second, it is crystal clear:

Murky :

if (element < 0 || MAX\_ELEMENTS < element || element == lastElement) {

...

}

Clear:

final boolean finished = (element < 0 || MAX\_ELEMENTS < element);

final boolean repeatedEntry = (element == lastElement);

if (finished || repeatedEntry) {

...

}

This approach both simplifies and documents complex expressions, making them easier to program without errors and easier to maintain.

* Always use parentheses to indicate operator precedence. They cost nothing, and save code readers from looking up precedence.
* If you code a chain of if-then statements, code the most common cases first.
* Strive to minimize the number of branches in your code. Whenever you find yourself dealing with a special case, take a moment to consider if it is possible to handle the problem in a more general fashion. Linear code is far easier to test.
* Make conditional blocks of code short enough to view all at once. Follow the 10 statement rule.
* Limit nesting to three levels.
* Compare boolean values to true or false implicitly, not explicitly:

|  |  |
| --- | --- |
| Right | Wrong |
| if (valid) {     ... } | if (valid == true) {     ... } |
| if (!valid) {     ... } | if (valid == false) {     ... } |

* When comparing against null always place the variable before the conditional operator:

|  |  |
| --- | --- |
| Right | Wrong |
| myVariable != null | null != myVariable |

### Loops

Integer loop index variables can be named with a single letter such as i, j, or k, in keeping with common conventions. This is the only exception to the naming standards for variables.

Prefer a "for" loop whenever possible. The advantages of the "for" loop are that it collects the loop control in a single place, and that it allows you to declare a loop control variable that is not accessible outside the loop.

Example:

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

Never modify the loop control variable inside the "for" loop. If this becomes necessary, use a "while" loop instead. Consider the example above: If the purpose of the loop were to delete selected items from the list, a "for" loop would be inappropriate since you wouldn't increment the loop control variable consistently:

Wrong

for (int item = 0; item < list.size(); item++) {

MyClass item = (MyClass) list.elementAt(item);

if (item.isOldAndTired()) {

list.removeElementAt(item);

item--; //Loop control is off limits!

}

}

Right

List<MyClass> list = ...;  
for (Iterator<MyClass> itr = list.iterator(); itr.hasNext(); ) {  
    MyClass item = list.next();  
    if (item.shouldDelete()) {  
       itr.remove();  
    }  
}

* Make loops as short as possible. If the loop code grows too long, consider restructuring the code using helper methods.
* Limit nesting to three levels.

#### For Each Loops

Use for-each loops to reduce error-proneness of iterators and index variables when iterating over collections and arrays.

|  |  |
| --- | --- |
| Right | Wrong |
| for (TimerTask t : c) {  t.cancel();  } | for (Iterator < TimerTask > I = c.iterator(); i.hasNext();) {  i.next().cancel();  } |

## Java Naming Standards

### General Standards

Each class, attribute, and method name must be meaningful and descriptive of the information it contains and/or the behavior it performs. Names may not contain spaces nor may they contain abbreviations. All names must be fully spelled out, except in the case of acronyms which are allowed in our code. Acronyms must be either industry standards or from the project Glossary in order to be used. Acronyms will be capitalized to follow the Sun naming convention for class and method names (i.e. RMIException and SSLSocket). Variable names (even those containing acronyms) will follow the usual convention of always beginning with a lower-case letter.

Example:

try {  
    //something   
} catch(RMIException rmiException) {   
    //do something   
}

### Package Names

Packages should use the following naming convention:

com.<organization-name>.<func-layer>.<func-domain>.<stereotype>.<sub-stereotype>

* <organization-name> commonly refers to the organization or framework name (e.g. accenture).
* <func-layer> represents functional area in the application (e.g. foundation).
* <func-domain> represents functional are (e.g. common).
* <stereotype> represents the components within the deploy layer (e.g. codetable).
* <sub-stereotype> optional, it represents the sub-component within the component (e.g. configuration).

Example: com.accenture.foundation.common.persistence, com.accenture.foundation.common.codetable.database

### Class Names

Class names must begin with a capital letter and separate multi-word names using capitalization (e.g. ProcessParticipant). Use camel case for all public class names.

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

Exception classes must end with the word "Exception". Separate multi-word exception names using capitalization with no intervening underscore and start with initial capitalization.

### Abstract Class Names

Abstract class names should start with the word "Abstract"; e.g. Abstract<Name>.java

Examples: AbstractJpaDao.java, AbstractController.java

### Interface Names

Java Interface names, like class names, must use camel-case with the first letter of each word capitalized. Java Interface names shall be a descriptive adjective, such as Runnable or Cloneable, although descriptive nouns, such as Singleton and DataInput, are also common. Only one interface will be defined per file.

### Interfaces and Implementations

Make concrete classes names very specific or use the suffix "Impl" to identify implementing classes.

public class VisitorDaoImpl extends AbstractDao implements VisitorDao

Example of concrete class names to highlight the use of different technologies.

public class VisitorJpaDao extends AbstractJpaDao implements VisitorDao

public class VisitorJdbcDao implements VisitorDao

### Method Names

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

Instance methods shall simply name the operation they perform on their associated object and should not encode the class name or the type of arguments expected. The use of "Object" or "object" should be avoided in method names as well. As an example, a String class instance function that returns the length of the string would be simply named length() - it would not be named stringLength(), since the context makes the meaning clear.

Static methods follow the same conventions as other methods but act on classes rather than objects.

Some specific rules for method naming include:

For example:

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(arg).

For example:

private boolean isValid();

private void setValid(boolean newValid);

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

When calling instance methods within a class, begin with the "this" keyword. If the method is in a parent class begin with the "super" keyword.

Methods for collections shall implement the following getters and setters:

|  |  |  |
| --- | --- | --- |
| Method Type | Naming Convention | Example |
| Getter for the collection | getCollection() | getMeterReadings() |
| Setter for the collection | setCollection() | setMeterReadings() |
| Insert an object into the collection | addCollection() | addMeterReading() |
| Delete an object from the collection | removeCollection() | removeMeterReading() |

### Variable Names

Variable names always begin with a lower case letter (i.e., documentCode, name, identifier). Collections shall be pluralized (i.e., ending in "s") as in VisitorAccounts or activeThreads.

### Instance Variable Names

A variable name shall not contain its own class name. (Place should not contain a variable 'placeIdentifier'. The variable should instead be named, 'identifier'. This will prevent subclasses from inheriting variables that refer to the parent class. Variables may be polymorphic across classes, yet must be unique to its own class. (For example, Place, Building, and Zoo classes may each have the attribute 'identifier', which is unique to that 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.

### Local Variable Names

Some key things to consider when working with local variables include:

Parameter names shall avoid, where possible, having the type implied. One should specify some sort of role in the name of the parameter e.g. newVisitor. This removes ambiguities when there is more than one parameter of the same type e.g. 'newVisitor', 'existingVisitor' rather than 'aVisitor1', 'aVisitor2' etc.

Do not "hide" 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.

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

Variable names i, j, k, and l are permissible as loop control variables.

For example:

public void createMeterReading() {  
    int counterIndex;  
    Account[] VisitorAccounts;  
    Account newVisitorAccount;  
  
 <statements>  
}

For Exception variables, specifically:

Within a try-catch, one can use the 'e' abbreviation to describe the exception being caught, e.g.:

try {  
    //some statements that will cause a throw  
} catch (SpecialException e) {  
    //react to exception e  
}

### Constant Names

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

For example:

static final int SPEED\_OF\_WATER = 3.10;

## Programming Styles

### Java Programming Style

This section describes a number of key concepts that are important in following a programming style that is consistent across the project. Most of these concepts are not project-specific but rather apply across projects to almost any development done in Java.

### Abstract Methods

Make methods abstract 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.

### Instance Variables

All variables must 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 expensive to build.

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.

Example

public class SavingProduct {

private BigDecimal interest;

public setInterest(BigDecimal interestRate) {

this.interest = interestRate;

}

public BigDecimal getInterest() {

return this.interest;

}

}

Try not to access a method through another method, e.g. this.getParticipant().getUnits(). If such a method chaining must be used frequently, the "Law of Demeter" suggests that a new method be created for "this", e.g. getParticipantUnits(). The idea is that a calling object should not need to know how the receiving object works or is composed.

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

Follow the same rules for static variables as instance 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.

Example

private static String databaseConnection;

public static String getDatabaseConnection() {

return databaseConnection;

}

### Visibility Types

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.

|  |  |  |
| --- | --- | --- |
| Visibility | Description | Proper Usage |
| **public** | A public method can be invoked by any other method in any other object or class. | When the method must be accessed by objects and classes outside of the class hierarchy in which the method is defined. |
| **Protected** | A protected method can be invoked by any method in the class in which 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 (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. | Package-level visibility makes methods and attributes accessible only by the containing class or another class in the same package. This visibility level is usually used for methods that need to be accessed within the same package but should not be externally used by client classes. |
| **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 reorganizing the behavior of other methods within the class to encapsulate one specific behavior. |

### Constants

Constants must be declared as static final attributes. If they need to be accessed outside of the class, they should be made public and then accessed as a public attribute.

### 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 than 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.

Example:

|  |  |
| --- | --- |
| Right | Wrong |
| int accountIndex;  …  //Code to get calculate certain account  …  Account account = accountList.get(accountIndex);  int newAccountNumber;  …  newAccountNumber = //Assign new account number  account.setAccountNumber(newAccountNumber); | int accountIndex;  …  //Code to get calculate certain account  …  Account account = accountList.get(accountIndex);  …  accountIndex = //Assign new account number  account.setAccountNumber(accountIndex); |

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.

### Notes on Specific Keywords and Constructs

#### Final

The final keyword is a relative of the C++ keyword const (though not the same). You can apply it to classes, methods and all kinds of variables:

* A final class may not be sub classed.
* A final method may not be overridden.
* A final variable may never be changed.

Using final on a class or method may have an optimization effect as well. The compiler may be able to perform in lining or compile-time linking instead of dynamic linking at run-time.

Limit the usage of final for classes and methods because it's hard to forecast extensions needs.

All variables (including method parameters) that can be final should be final. In the case of constants, this may allow in lining by the compiler, and it is in any case an excellent documentation tool.

#### Return

A method that returns a value should only have a single return statement. Having multiple return statements in a method makes the code less readable because it is less clear to the reading which value will be returned. Having one return statement at the end of the method allows a reader to easily find what is being returned.

#### Constructors

There shall normally be only one main constructor in a class. Additional convenience constructors may be defined, but they shall be implemented in terms of the main constructor. The point of this is to avoid duplicate code:

**Main Constructor**

public MultiLineLabel(String label, int marginWidth, int marginHeight, int textAlignment,

int fixedSize) {

this.breakLabel(strLabel);

myMarginWidth = marginWidth;

myMarginHeight = marginHeight;

myTextAlignment = textAlignment;

myFixedWidth = fixedSize;

}

**Incorrect convenience constructor (repeats code from above)**

public MultiLineLabel(String strLabel) {

this.breakLabel(strLabel);

myMarginWidth = 0;

myMarginHeight = 0;

myTextAlignment = this.LEFT;

myFixedWidth = 0;

}

**Correct convenience constructor**

public MultiLineLabel(String strLabel) {

this(strLabel, 0, 0, this.LEFT, 0);

}

#### Threads

Debugging and profiling can be made significantly more effective by naming all threads explicitly. Therefore, make sure to always use the Thread constructors that take a name parameter, e.g. use Thread(String name) instead of Thread().

Example:

|  |  |
| --- | --- |
| Right | Wrong |
| Thread mainThread =  new Thread("Main"); | Thread mainThread = new Thread(); |

### Using Java Collections

Using the Java Collections framework (java.util.\*) plays a big role in any development project. It is of utmost importance to always use the correct Collection for the correct task. Additionally, with all development it is important to abstract away details to make a system more flexible to future change. As such, when using Collections, one should always use one of the core three interfaces (Map, List, or Set) whenever possible rather than concrete implementations, for example, HashMap, ArrayList or HashSet. ALL method signatures should receive and return a collection interface unless there is a very good reason not to. Following this standard allows us to change the collection implementations used in the application without impacting the whole system.

|  |  |
| --- | --- |
| Right | Wrong |
| public Map listToMap(List myList) {  Map myMap = new HashMap();  ...  //do translation  return myMap;  } | public HashMap listToMap(ArrayList myList) {  HashMap myMap = new HashMap();  ...  //do translation  return myMap;  } |

Additionally, the Collection implementations Vector and Hashtable should NEVER be used. These collections are antiquated and tend to perform slower than the newer options because they are synchronized. Use ArrayList and HashMap instead.

### Generics

Generics allow a type or method to operate on objects of various types while providing compile-time type safety eliminating the need for casting.  This not only reduces clutter. It also reduces the possibility of a run time error, adding compile time type safety to Collections Framework.

* A collection object should be defined as follows with no casting in the statements.

|  |  |
| --- | --- |
| Right | Wrong |
| List< Integer > myList = new LinkedList<Integer>();  myList.add(0);  Integer x = myList.iterator().get(0); | List myList = new LinkedList();  myList.add(0);  Integer x = (Integer) myList.iterator().get(0); |

* Correct usage of wild cards when there is a need for super type and the type of the object is unknown. The “?” stands for an unknown type,

Here is an example the method prints all the items in a collection that demonstrates this.

|  |  |
| --- | --- |
| Right | Wrong |
| void printCollection(Collection<?> coll) {  for (Object obj : coll) {  System.out.println(obj);  }  } | Void printCollection(Collection<Object> coll) {  for (Object obj : coll) {  System.out.println(obj);  }  } |

* Bounded Wild cards are used when will there is a need to accept generic super types but are bound to a specific class it extends.

In this example the super keyword allows any class that subclasses E, but the API is only meant to use the functions of the superclass E. This is called a lower bound.

Comparator<? **super** E> c is an example of a lower bounded wildcard.

However, if your API returns E you should use the upper bound wildcard.

Comparator<? **extend** E> c is an example of a upper bounded wildcard*.*

* Generic Methods be used when method declarations can be generic - that is they can be parameterized by one or more type parameters.

The example below is a method that takes an array of objects and a collection and puts all objects in the array into the collection.

static <T> void arrayToCollection(T[] array, Collection<T> collection) {

for (T object : array) {

c.add(object);

}

}

For more details please refer <http://download.oracle.com/javase/tutorial/extra/generics/index.html>

### BigDecimal for Financial Calculations

BigDecimal should be used for financial calculations instead of double or float. The BigDecimal class provides operations for arithmetic, scale manipulation, rounding, comparison, hashing, and format conversion. The toString() method provides a canonical representation of a BigDecimal.

The BigDecimal class gives its user complete control over rounding behavior. If no rounding mode is specified and the exact result cannot be represented, an exception is thrown; otherwise, calculations can be carried out to a chosen precision and rounding mode by supplying an appropriate MathContext object to the operation. In either case, eight rounding modes are provided for the control of rounding. Using the integer fields in this class (such as ROUND\_HALF\_UP) to represent rounding mode is largely obsolete; the enumeration values of the RoundingMode enum, (such as RoundingMode.HALF\_UP) should be used instead.

BigDecimal constructor that takes in a String is recommended among the other allowed arguments.

More information on BigDecimal can be found on [Java API](http://java.sun.com/j2se/1.5.0/docs/api/java/math/BigDecimal.html).

### Annotations

* @Override - Use @Override to indicate to the compiler that the method overrides a method in the super class.

class Parent {

public void doSomething(float a, float b) {

…

}

}

class Child extends Parent {

@Override

public void doSomething(float a, float b) {

…

}

}

* @Deprecated – Use @Deprecated to a method or type to indicate that it is deprecated.

class MyTest {

@Deprecated.

public void doSomething(float a, float b) {

…..

}

}

* @SuppressWarnings – This annotation is used to suppress compiler warnings and must not be used.

## Documentation in Java Programs

Comments should add to the clarity of your code. The reason why you document your code is to make it more understandable to you, your co-workers, and to any other developer who may come after you. Well-documented code is more readable and easier to maintain. When commenting code, you should:

* Only inline comment confusing/interesting calls
* All methods must have documentation comments
* Ensure that comments are valuable, not redundant

Along with documenting code, every effort should be made to make code self-documenting. This means writing code and naming variables and methods such that it is easy to understand the flow and use of code. There should be no spelling mistakes in the documentation or variable names.

### Comment Types

Java has three styles of comments:

1. Documentation comments start with /\*\* and end with \*/. Use documentation comments directly before declarations of interfaces, classes, and methods to document them. Documentation comments are processed by JavaDoc (see below) to create external documentation for a class.   
   Use documentation comments to capture the business logic that your class performs. Never include a summary of the interactions with other layers or stereotypes, since this is already well documented elsewhere.   
   For example, here are the right and wrong ways to document a class called SearchParticipantAction:   
     
   **Right**

/\*\*

\* This class validates that the user

\* entered either an SSN or a combination

\* of first and last name, then initiates

\* a search for a list of related Participants.

\*

\*@authorJohn Doe

\*/

**Wrong**

/\*\*

\* This class sends a DTO to ParticipantService,

\* and returns a PaginatedResultSet.

\*

\*@authorJohn Doe

\*/

1. Multi-line comments start with /\* and end with \*/. Use C Style comments to document out lines of code that are no longer applicable, but you want to keep just in case requirements change in the future.   
     
   **Example**

/\*

John Doe commented out this code

on 06/24/04 and replaced it

with the following code.

<the old source code>

\*/

1. Single-line comments start with // and go until the end of the source-code line. Use single line comments internally within methods to document business logic, sections of code, and declarations of temporary variables.   
     
   Example

// Update the effectively of

// the now invalid participant

// object.

Included in Sun's Java Development Kit (JDK) is a program called Java Doc that processes Java code files and produces external documentation, in the form of HTML files, for your Java programs

### Project Documentation Standards

#### Class Descriptions

Every class or interface definition shall be preceded by a Java Doc description. The comment block will have a general class description, authors, and the date the unit was created.

Class descriptions should follow the format:

/\*\*

\* The class summary goes here, and the first sentence

\* ends with a period. Additional sentences go here

\* and may include HTML for formatting.

\*

\* @author <your\_name>

\*/

#### Method Descriptions

Every method except for the most trivial of private methods shall be preceded by a JavaDoc comment. These comments shall also be kept updated, reflecting any changes that have been made.

The first sentence of the description must be a business-focused overview. Details shall be included in any following sentences. The following constructs are used to provide extra details where appropriate:

* The @param tag shall be used for every parameter in the method.
* The @return tag shall be used to describe the return type, if any.
* The @throws tag shall be used to describe any exceptions the method may throw.
* The @see tag may be used to create a "see also" link to other similar classes or methods.

Method comments shall follow the format:

/\*\*

\* <Method description>

\*

\* @param parameter-name description

\* @return return-description

\* @throws exception-description

\* @see example.SimilarObject#similarMethod()

\*/

Example:

/\*\*

\* This method assigns a new Location to the Role.

\*

\* @param newLocation The new Location to assign.

\*/

#### Variables

Although variable names should be chosen to indicate their purpose, if needed for clarity an inline comment (//) describing the variable can be placed on the line with the variable declaration.

Example

public class Window {

...

private Window parent; //the Window I was created from

}

#### Other Comments

In-line comments should be used to explain pieces of code that are complex or potentially confusing. In addition, all items that need to be revisited or are incomplete shall be marked with "//TODO:" so that these comments can be easily found.

Example:

public void myMethod() {

//TODO: implement this method fully

}

### General JavaDoc Usage

Comments must be formatted in a certain way in order for the JavaDoc utility to extract them. All comments intended for JavaDoc must be preceded by /\*\* and concluded with \*/. Place these symbols on separate lines with an additional \* separating each line of comments:

/\*\*

\* Some comments.

\* Some more comments.

\*/

Developers shall run the JavaDoc utility on their code prior to a code review to ensure that the JavaDoc that results is well-formed. Additionally, the build process will generate JavaDoc for all code in the version control tool on at least a nightly basis. All errors and warnings while generating the documentation should be eliminated. Also, all hyper-links in the generated documentation should be tested for validity.

#### First Sentence

The first sentence in a JavaDoc description is considered summary text for your method or class. JavaDoc considers your first sentence as being all text up to the first period that is followed by a blank, tab, or line terminator, or at the first tag:

/\*\*

\* This is the first sentence

\* This is still the first sentence. This is the next sentence.

\*/

This is important because JavaDoc creates a summary for each member that is documented. Each summary is included in the "Index" sections of the final output (Constructor Index, Field Index, Method Index etc.) That summary comprises the first sentence. So, the example above would look like this in the JavaDoc member summary:

This is the first sentence This is still the first sentence.

Be aware of your first sentence. Make note that it is used for the summary text of your method.

#### JavaDoc Tags

JavaDoc also provides tags to include additional information in the comments. Use the following JavaDoc tags to document your code:

|  |  |  |
| --- | --- | --- |
| **Tag** | **Usage** | **Purpose** |
| @author **name** | Classes, Interfaces | Indicates the author(s) for a given piece of code. Use one tag per author. |
| @throws **name description** | Methods | Describes the exceptions a method throws. Use one tag per exception and give the name of the exception. |
| @param **name description** | Methods | Used to describe the parameter passed to a method, including its type/class and its usage. One tag per parameter. |
| @return **description** | Methods | Describes the return value, if any, of a method. Describe the type/class and the potential use(s) of the return value. |
| @see **ClassName** | Classes, Interfaces, Methods, Attributes | Generates a hypertext link to the specified class in the generated documentation. Use a fully qualified class name. |
| @see **ClassName#methodName()** | Classes, Interfaces, Methods, Attributes | Generates a hypertext link to the specified class method in the generated documentation. Use a fully qualified class name. |

**Clarification on the @see tag**

The @see tag allows JavaDoc content to contain a hyperlink to another JavaDoc description for a class or a method.

@see is not needed to refer to an interface or abstract class whose methods you are implementing. JavaDoc automatically generates a "specified by" section that provides this information. For example, in your Service EJB's bean implementation class, do not use @see to refer to the Activity methods it implements.

Use @see to link to another class or method that provides similar or related functionality. For example, in the business methods of your Service EJB's remote interface, use @see to link to the Activity that specifies your business method. JavaDoc will not automatically generate this link, since only the EJB's implementation class directly implements the Activity interface.

The follow examples are all valid @see links:

@see java.lang.String

@see String

@see String#equals()

@see java.lang.Object#wait(int)

@see Character#MAX\_RADIX

@see <a href="spec.htm">Java Spec</a>

The pound sign (#) separates the name of a class from the name of one of its fields, methods, or constructors. If there are multiple methods of the same name (the method is overloaded), a list of arguments must be included to fully identify the method to link to. For instance, the following examples each identify a unique method on the java.lang.Object class:

@see java.lang.Object#wait()

@see java.lang.Object#wait(long timeout)

@see java.lang.Object#wait(long timeout, int nanos)

Additionally, when within a class, the class name can be implicit. The following example would look for the method myMethod() in the class that it was defined:

@see #myMethod()

#### JavaDoc Tags

JavaDoc alone is quite restrictive with its formatting capabilities. It does not consider carriage returns or asterisks as new lines. The only thing that causes a new line break is a JavaDoc tag (see above). Therefore, it is suggested that HTML tags are used within JavaDoc to add formatting when it helps to clarify the information to be contained in the JavaDoc.

All HTML used in JavaDoc code must conform to the HTML Coding Standards.

The following examples demonstrate the most common HTML tags found in JavaDoc.

<br/> tag - break  
If you want to start a new line in your main JavaDoc body, use the <br/> tag:

/\*\*

\* This is the first sentence. This sentence will start immediately after

\* the first sentence with no line break.<br/> This sentence will start on a

\* new line.

\*/

<p> tag – paragraph

If you want to start a new paragraph, use the <p> tag. Subsequently, the end of a paragraph should be marked with the </p> tag:

/\*\*

\* This is the first sentence.

\* <p>This is a paragraph. It contains lots of sentences.</p>

\* <p>This is another paragraph. It also contains lots of sentences, but

\* there is a distinct gap between this paragraph and the first one.</p>

\* <p>Here is some more text. There will also be a gap between this and

\* the last paragraph.</p>

\*/

* **<ul> tag - unordered list**  
  HTML gives the ability to create bulleted lists. These are also known as unordered lists. Use this tag to start a list, and then use the </ul> tag to finish the list. Each list item must be preceded with the <li> tag and ended with the </li> tag:

/\*\*

\* This is the first sentence. Here is an unordered list of items:

\* <ul>

\* <li>January</li>

\* <li>February</li>

\* <li>March</li>

\* </ul>

\* That was nice.

\*/

* <ol> tag - ordered list  
  Similar to the <ul> tag, <ol> creates a numbered list. Each list item is numbered automatically for you:

/\*\*

\* This is the first sentence. Here is a numbered list of items:

\* <ol>

\* <li>Perform this</li>

\* <li>Then this</li>

\* <li>Finish with this</li>

\* </ol>

\* That was nice.

\*/

HTML, like JavaDoc, ignores carriage returns. Therefore, the output from Example 1 below is no different from the output of Example 2:

Example 1

/\*\*

\* This is the first sentence.

\* <br/>

\* New line.<p>

\* New paragraph

\* </p>

\* <ul><li>

\* Item 1

\* </li>

\* <li>

\* Item2

\* </li></ul>

\*/

Example 2

/\*\*

\* This is the first sentence.

\* <br/>New line.<p>New paragraph</p>

\* <ul>

\* <li>Item 1</li>

\* <li>Item 2</li>

\* </ul>

\*/

For detailed information on writing JavaDocs, please refer to the Oracle JavaDoc website

[http://www.oracle.com/technetwork/java/javase/documentation](http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html)

## Good Coding Practices

### Decomposition

* Keep methods to **25** statements in length
* Emphasize variability. For example:

|  |  |
| --- | --- |
| Right | Wrong |
| if (condition) {  result = "a";  } else {  result = "b";  }  call (result, 1, 2); | if (condition) {  call("a", 1, 2);  } else {  call("b", 1, 2);  } |

### Readability

* Know your coding standards
* Keep code as readable in English as possible
* Use suitable names
* Break up long statements
* Absolutely no abbreviations (acronyms are OK)
* Specify receiver, including "this"
* One return statement in a function
* Don't compare to true in conditionals
* *don'tforgetwhitespaceforreadability* (don't forget whitespace for readability)

### Comments

* Javadoc comments for all classes and methods
* Inline comments for interesting things
* Comment examples
* Don't mention external users of the method in comments.

### Interfaces and Dependencies

* Carefully differentiate public and private
* Use getters and setters instead of making variables public or protected
* Throw exceptions early and throw exceptions often
* Be specific in imports
* Create helper methods
* Methods should declare intention (isPending, etc.)

### Process

* Don't blame the language for shortcomings
* Get it right the first time
* Don't write code you would be ashamed to show others
* Writing good code will become natural

### Exception Handling

There must be an exception handling framework defined for an application. The exception handling framework must provide a basic hierarchy for exceptions as well as the basic exceptions needed on a project.

Exception strategy must meet the following requirements:

1. Consistency in exception handling is important because it is essential for the application’s usability and maintainability.
2. Take into account the different types of exceptions that can occur. The application needs to handle architecture exceptions, Java API exceptions from Java API calls made directly by the application and exceptions from 3rd party libraries called directly by the application.
3. Identify base application exceptions and define how they should be handled (e.g. appropriate error messages that need to be sent to the user).

## References

1. Java Coding Standards:
   * [http://www.oracle.com/technetwork/java/codeconv](http://www.oracle.com/technetwork/java/codeconv-138413.html)
2. JSP Coding guidelines - <http://java.sun.com/developer/technicalArticles/javaserverpages/code_convention/>
3. Other links
   * [http://www.oracle.com/technetwork/java](http://www.oracle.com/technetwork/java/index.html)
   * [http://www.oracle.com/technetwork/java Developer Tutorials and Training](http://www.oracle.com/technetwork/java/index-jsp-135888.html)
   * [http://download.oracle.com/javase/tutorial](http://download.oracle.com/javase/tutorial/index.html)
   * [http://download.oracle.com/javaee/6/tutorial](http://download.oracle.com/javaee/6/tutorial/doc/)
4. AFP J Reference links

* [AFP-J Exception Handling Guidelines](https://collaboration.accenture.com/display/JAVA/Exception+Handling+Guidelines)
* [AFP-J Overview of Logging Framework](https://collaboration.accenture.com/display/JAVA/Overview+of+Logging+Framework)
* [AFP-J Enumerated Types Guidelines](https://collaboration.accenture.com/display/JAVA/Enumerated+Types+-+Overview+and+Guidelines)

# Java Naming Standards

## Web Application Style

### Configuration Files

#### Presentation Tier Configuration Files

Configuration files should be named as illustrated in Table 1.

Table 1. Web Configuration Files

| File | Format or Example | Location |
| --- | --- | --- |
| Web Deployment Descriptor | web.xml | src/main/webapp/WEB-INF/ |
| Spring Web Application Context (main application context for web applications) | web-application-config.xml | src/main/webapp/WEB-INF/ |
| Spring Web Application Context for the action DispatcherServlet | <servlet name>-servlet.xml | src/main/webapp/WEB-INF/ |
| Spring MVC Configuration | webmvc-config.xml | src/main/webapp/WEB-INF/ |
| Spring Security Context | security-context.xml | src/main/webapp/WEB-INF/ |
| Sitemesh configuration | decorators.xml | src/main/webapp/WEB-INF/ |
| Strtus Menu configuration | menu-config.xml | src/main/webapp/WEB-INF/ |
| Spring Web Flow Configuration | webflow-config.xml | src/main/webapp/WEB-INF/ |
| Spring Web Flow – Flow Definitions (single file per flow) | <flow-name>.xml | src/main/webapp/WEB-INF/flows/<flow-name>/ |
| Spring Web Flow – Bean Definitions (single file per flow) | <flow-name>-beans.xml | src/main/webapp/WEB-INF/flows/<flow-name>/ |
| Site text/labels properties file. | sitetext\*.properties | src/main/resources/ |

#### Business Tier Configuration Files

Configuration files should be named as illustrated in Table 2.

Table 2. Business Tier Configuration Files

| File | Format or Example | Location |
| --- | --- | --- |
| Business Services Context | business-service-context.xml | src/main/resources/ |
| Rules Engine Context | rules-context.xml | src/main/resources/ |

#### Data Access Tier Configuration Files

Configuration files should be named as illustrated in Table 3.

Table 3. Data Access Tier Configuration Files

| File | Format or Example | Location |
| --- | --- | --- |
| Data Access Context | dao-context.xml | src/main/resources/ |
| Integration Context | aij-context.xml | src/main/resources/ |

#### Domain Configuration Files

Configuration files should be named as illustrated in Table 4.

Table 4. Domain Configuration Files

| File | Format or Example | Location |
| --- | --- | --- |
| Persistence Context | persistence.xml | src/main/resources/META-INF |
| Named Query Configuration | {persistence-unit}-named-query.xml | src/main/resources/ |

### Presentation Tier

#### URLs

Table 5. URL Name Examples

| Format | Examples | Description |
| --- | --- | --- |
| /prepare<Entity>Search | /prepareClaimSearch | Takes the user to a blank search page. |
| /search<Entity> | /searchClaim | Sends the search criteria to the server and returns a list of search results. |
| /find<Entity> | /findClaim | Retrieves a single entity in either View or Update Mode. |
| /save<Entity> | /saveClaim | Updates the information in the database. |
| /create<Entity> | /createClaim | Inserts a new record in the database. |
| /update<Entity> | /updateClaim | Updates an existing record in the database |
| /delete<Entity> | /deleteClaim | Removes the entity from the user's view. |

Example 1. Concrete URL

http://www.accenture.com/myPerformance/saveObjective/

NOTE: Mixed cases are supported in URLs. Please follow Java case naming conventions (first word is all lower case, 2nd word begins with an upper case letter).

#### HTML and JSP Files

Table 2 provides examples for commonly operations. Depending of the use case being solved, other names may be needed. Keep all page names consistent with the naming convention standard: <action><Entity>.jsp

Table 6. HTML and JSP File Name Examples

|  |  |  |
| --- | --- | --- |
| Format | Example | Description |
| create<Entity>.jsp | createClaim.jsp | Adds new data |
| update<Entity>.jsp | updateClaim.jsp | Changes current state data |
| delete<Entity>.jsp | deleteClaim.jsp | Removes data |
| view<Entity>.jsp | viewClaimOverview.jsp | Displays data and does not allow update |
| search<Entity>.jsp | searchClaim.jsp | Finds items that fulfill criteria |
| search<Entity>Results.jsp | searchClaimResults.jsp | Displays a collection of data associated with a search criteria |
| maintain<Entity>.jsp | maintainClaim.jsp | Multi-role page (view, update, delete, create). |

Table 73. Other HTML and JSP File Standards

| Component | Format or Example |
| --- | --- |
| Directory Structure for JSPs | src/main/webapp/WEB-INF/pages/<functionalArea>/<subFunctionalArea>/<pageName>.jsp |
| Page Title | Same as page name. |
| Form Name (Form tag attribute) | <PageName>Form |

#### JavaScript Files

Table 8. JavaScript Examples

| Component | Format or Example |
| --- | --- |
| JavaScript File Naming Standard | <functionalArea>\_<subFunctionalArea>.js |
| Method Name | methodName() (Same as Java Naming Standards) |
| JavaScript File Location | <web\_project>: src/main/webapp/js |

### Business Tier

#### Business Services Class Names

The name of a Business Implementation must be the same as the Service Interface that it implements. The implementation will use the subfix Impl; e.g., AccountManagementServiceImpl.

All classes and interfaces must follow the AFP-J Java Coding Standards.

#### Service Method Names

For methods that are within the context of the entity (entity name contain in the class name), do not add the entity name as part of the method name. For example, given a class named AccountManagementService.java, name a method that updates data update() instead of updateAccount().

Table 9. Transactional Method Names

|  |  |  |
| --- | --- | --- |
| Format | Example | Description |
| create<Entity>(...) | create(…)  createAccount(…) | Creates new data. |
| update<Entity>(...) | update(…)  updateAccount(…) | Change current state of the data. (Edit or Update) |
| delete<Entity>(...) | delete(…)  deleteAccount(…) | Remove data. |

The entity name on Service methods is optional. If the service operation is limited to a single entity the name of the entity is not needed on the methods name.

Table 10. Search Method Names

|  |  |  |
| --- | --- | --- |
| Format | Example | Description |
| findByPrimaryKey(<key>) | findByPrimaryKey(<key>) | Retrieves one record for the given primary key. |
| find(...) | find(...) | Retrieves only one item per criteria. |
| find<Entity>(...) | findCase(...) | Retrieves only one item per criteria. |
| search(...) | search(...) | Retrieves a collection of data per criteria which is non-specific. |
| searchFor<Entity>(...) | searchForCase(...) | Retrieves a collection of data per criteria which is non-specific. |
| searchFor<Entity>By<Entity>(...) | searchForCaseByParticipant(...) | Retrieves a collection of data per criteria which is specific. |

Table 11. Functional Method Names

|  |  |  |
| --- | --- | --- |
| Format | Example | Description |
| <businessFunction>(...) | calculateMonthlyPayment(...) | Executes a business function on some data. |
| <businessFunction><Preposition><Entity>(...) | calculateMonthlyPaymentByParticipant(...) | Executes a business function on some data in relation to an entity. |

Note: Different combinations of <Preposition><Entity> may be used to extend this. Example preposition words are: For, By.

#### Business Object (BO)

A Business Object represents a business concept found in the Logical Data Model. Example: ACCOUNT or VISITOR.

##### BO Class Names

The name of a Business Object starts with the name of the entity and ends with Bo. For example, if the entity ACCOUNT exists in the Logical Data Model, then the name of the Business Object that represents it should be AccountBo.

##### BO Method Names

A Business Object may have relational methods that add or remove its relationships with other entities.

Table 12. Relational Method Names

|  |  |  |
| --- | --- | --- |
| Format | Example | Description |
| add<Entity>(...) | addParticipant(...) | Adds new entity relationship to the Business Object. |
| remove<Entity>(...) | removeParticipant(...) | Removes a specific entity relationship from the Business Object. |
| removeAll<Entity>(...) | removeAllParticipants(...) | Removes all entity relationships from the Business Object. |

Table 13. Functional Method Names

| Format | Example | Description |
| --- | --- | --- |
| <businessFunction>(...) | calculateMonthlyPayment(...) | Executes a business function on some data. |
| <businessFunction><Preposition><Entity>(...) | calculateMonthlyPaymentByParticipant(...) | Executes a business function on some data in relation to an entity. |

Note: Different combinations of <Preposition><Entity> may be used to extend this. Example Preposition words are: For, By, etc.

#### Data Transfer Object

Data Transfer Objects (DTOs) are serializable objects that transfer data from the Presentation tier to the Business tier. DTO uses the following naming convention:

<PageName>Dto

SearchAccountDto

##### DTO Method Names

Composite and Atomic DTOs will have get and set methods for each property stored in the DTO in accordance with the Java Beans specification. For example, for a property such as Name, a DTO will have methods getName() and setName(). In addition, Composite DTOs will have factory methods to produce Atomic DTOs from the data contained within the Composite. These factory methods will be named as follows:

create<EntityName>Dto()

createAccountDto()

Note that a Composite DTO may contain several create methods, one for each Atomic DTO the Composite DTO contains.

### Data Access Tier

#### Data Access Object (DAO)

##### DAO Class Names

DAO Implementation must be the same as the Dao Interface that it implements. DAO Interfaces use the following naming convention <Entity>Dao, e.g. VisitorDao. DAO implementations use the subfix Impl; e.g., VisitorDaoImpl.

##### DAO Method Names

Table 14. CRUD Method Names

|  |  |
| --- | --- |
| Format | Description |
| store(<Entity>) | Stores the updated domain/entity object in the database. |
| delete(<Entity>) | Removes a domain/entity object from the database. |
| refresh(<Entity>) | Refresh an object from the underlying DB |

Table 15. Search Method Names

|  |  |  |
| --- | --- | --- |
| Format | Example | Description |
| findByPrimaryKey(<key>) | findByPrimaryKey(<key>) | Retrieves one record for the given primary key. |
| find(...) | find(...) | Retrieves only one item per criteria. |
| search(...) | search(...) | Retrieves a collection of data per criteria which is non-specific. |
| searchFor<Entity>By<Entity>(...) | searchForCaseByParticipant(...) | Retrieves a collection of data per criteria which is specific. |

## Batch Application Style

### Configuration Files

Configuration files should be named as illustrated in Table 16.

Table 16. Batch Configuration Files

| File | Format or Example | Location |
| --- | --- | --- |
| Job Configuration | <job name>.xml | src/main/resources/<job name>/ |

### Job Tier

Table 17. Batch Job Components

|  |  |
| --- | --- |
| Component | Format |
| ItemProcessor implementation | <action>ItemProcessor.java |
| ItemWriter implementation | <action>ItemWriter.java |
| Tasklet implementation | <action>Tasklet.java |
| Listener implementations | <action><listener interface name> |

## Integration Application Style

### Configuration Files

The configuration files for integration should be named as illustrated in Table 18.

Table 18. Integration Configuration Files

| File | Format or Example | Location |
| --- | --- | --- |
| Spring-integration context configuration files | integration-context.xml | src/main/resources |

### Integration Tier

Table 19. Extendable service class names

|  |  |
| --- | --- |
| Component | Format |
| Outbound Adapter class | <protocol/technology>OutboundAdapter.java |
| Inbound Adapter class | <protocol/technology>InboundAdapter.java |
| Transformer class | <technology/direction>Transformer.java |
| Gateway class | <protocol/technology>Gateway.java |
| Remote Service Interface | <Provider>RSI.java |