ORACLE
Consulting

Coding Guidelines

Oracle Identity and Access Management

Coding Guidelines

Oracle Identity and Access Management

Table of Contents

Preface

This document embodies a unified set of standards to be used on IAM projects throughout Oracle Consulting. It addresses the core constructs and structures of the Java language, rather than application of the language (a set of guideline documents are being prepared for the major Java-related technologies within Oracle such as the ADF and IAM frameworks). The standards will be revisited and reissued regularly as they are adopted and exercised by new projects.

This document assumes that the reader has a basic knowledge of Java. Its purpose is not to teach the language but to advise best practice with regard to maintainable and robust code.

The purpose of the guide is to provide comprehensive team member Orientation Guide to use during Project Kickoff and afterward for the on boarding of new team members. This will help to streamline the on-boarding process, ensure consistent communications and reduce the project manager and team leads workload during the on-boarding process.

Background

Development of code on any project should be considered a team activity. Any individual developer does not own code, but will more likely evolve through successive iterations of design and build often being updated by several people. As a result, it is imperative that developers use guidelines to aid the development process.

These Java coding standards supersede the existing draft Oracle Java coding standards [1] when used within Oracle Consulting. (No work is being done on the draft standards, hence will never progress from draft status and will not be updated to reflect the current Java specification).

This document should be used in conjunction with the Sun Java Coding Standards [3]. In some instances, it overrides recommendations from Sun, based on past Oracle project experience. When this document fails to cover a topic, recommendations from Sun should be followed.

Confidentiality

Typographical Conventions

The following text conventions are used in this document.

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.

Convention	Meaning
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.
hyperlink	A link to another section of the document or to an external document or website.

Conventions Directory Variables

The following table explains variables that might be used in this document.

Variable	Meaning
JAVA_HOME	The location where the supported Java Development Kit (JDK) was installed.
ORACLE_BASE	The base directory where Oracle products are installed.
ORACLE_HOME	The location for a product's binaries. For the application tier host computers, it should be stored on a shared disk.

Symbol Conventions

The following table explains symbols that might be used in this document.

Convention	Meaning
[ ]	Contains optional arguments and command options.
{ | }	Contains a set of choices for a required command option.
${ }	Indicates a variable reference.
-	Joins simultaneous multiple keystrokes.
+	Joins consecutive multiple keystrokes.
>	Indicates menu item selection in a graphical user interface.

Related Documents

No.	Document
1	Draft Oracle Java Coding Standards (http://www-apps.us.oracle.com/java/codestand.htm)
2	How to Write Doc Comments For Javadoc (https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html)
3	Sun Java Coding Standards (http://java.sun.com/docs/codeconv/index.html)
4	Thinking In Java, Bruce Eckel (http://www.bruceeckel.com/)

Principles

This document follows two broad IT conventions in order to create a consistent set of naming guidelines.

Convention over Configuration

Convention over Configuration - also known as Coding by Convention (as defined by Wikipedia) is a software design paradigm which seeks to decrease the number of decisions that developers need to make, gaining simplicity, but not necessarily losing flexibility. To some readers the fact an Oracle and ADF document is trying to dictate convention over configuration might seem rather comical given ADF through its heavy use of XML files is all configuration. However, the leading principle of decreasing the number of decisions developers need to make is easily adaptable into this guide.

No Hungarian Notation

Hungarian notation (as defined by Wikipedia) is a code convention of adding prefixes or suffixes to objects to indicate the object types or use. For example for an integer counter field, it would be named iCounter. While it's recognized some programmers & languages have a preference for using Hungarian notation, because it is arguable not the norm in Java programming, and it can lead to inconsistent names when code is modified or ported, it will not be adopted for this document as it is counter to the stated goal "Be flexible enough to adapt to change". There are some scenarios where the principles conflict with each other and it is necessary to apply a precedence order to the rules. For example when JDeveloper creates an ADF View Object, it includes a “View” suffix in the View Object’s name. This violates the No Hungarian Notation principle. However, to change the name violates the Convention over Configuration principle. To resolve this conflict, the document puts precedence on Convention over Configuration before the No Hungarian Notation rule. This has the added benefit that when you introduce junior programmers to your ADF team who have just undertaken ADF training, or a new ADF programmer joins your team from another organization, they will be familiar with the default conventions used by JDeveloper over a unique set of guidelines that require you to modify all names.

General Guidelines

Formatting Code

Use two spaces of indentation per nesting level. Use spaces rather than tabs to avoid problems with different tab definitions on printers and editors. The length of each source line should not exceed 80 characters.

JDeveloper should be set up to use these defaults.

Go to **Tools** | **Preferences** | **Code Editor** | **Code Style**

Comments

Comments must be produced in parallel with source code development, not tagged on afterwards. Comments should be used as follows:

- Javadoc comments (`/**  */`) should be used to generate appropriate class, method and variable documentation (see section X for more detail).
- Single-line comments (`//`) should be used for any non-documentation comments within code. If there are several consecutive lines of comment, each must begin with `//`.
- C-style comment markers (`/* */`) should only be used to comment out blocks of code. This should only be done as part of the development process or debugging, delivered modules should not contain any commented out code.

Source Files

A Java source file may contain only one public class or interface. When top-level (i.e. package level) non-public classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

In addition to top-level classes, defined in the context of the package, inner classes can be defined within the context of a parent class. This includes member classes, local classes and anonymous classes (see [4] for more information). There are some situations in which inner classes useful (for example, local and anonymous classes are often used in event-handling code). However, they can also make the code difficult to follow. In general, consider the justification for an inner class carefully (for example, the class will not be available outside the context of the parent class) and ensure that the inner class is fully commented.

Organize the material in each source file as follows:

-

The final three categories form a repeating group. Public member variables, constructors and methods appear first. This is then followed by protected member variables, constructors and methods, and so on for default (package level) and private visibilities. Within each set of method definitions, group ‘get’ and ‘set’ functions for a particular attribute together and group other logically related methods together.

File Header

Begin each file with a non-javadoc comment including:

- Disclaimer, if applicable
- Copyright information, if applicable.
- Information identifying the file including the project, subsystem and filename.
- A list of the classes or interfaces defined in the file, which will always include the one public class or interface included in the file plus any non-public top-level classes and any inner classes.
- History table listing dates, authors, and summaries of changes. Change summaries can often be generated automatically by source control systems, for example PVCS replaces the label Log with the change history (although this is only useful if developers enter meaningful comments when checking files in).

For example:

```
/*
  Oracle Deutschland BV & Co. KG

  This software is the confidential and proprietary information of
  Oracle Corporation. ("Confidential Information"). You shall not
  disclose such Confidential Information and shall use it only in
  accordance with the terms of the license agreement you entered
  into with Oracle.

  ORACLE MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY
  OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
  TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
  PURPOSE, OR NON-INFRINGEMENT. ORACLE SHALL NOT BE LIABLE FOR ANY
  DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
  DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

  Copyright © 2014. All Rights reserved

  ------------------------------------------------------------------------

  System      :  Oracle Access Authentication Plug-In Library
  Subsystem   :  Common Shared Runtime Facilities

  File        :  AccessPluginError.java

  Purpose     :  This file implements the interface
                 AccessPluginError.


  Revisions    Date        Editor      Comment
  -----------+-----------+-----------+---------------------------------
  3.1.0.0      2014-28-10  DSteding    First release version
```

Import Statements

Although Java allows the use of an asterisk as a wildcard character in import statements, its use should be avoided, since it increases the chance of namespace clashes between packages. Instead, explicitly import each class required. This also makes maintenance easier, since a fully qualified import statement allows you to find documentation relating to the class quickly.

To improve clarity, imports should be listed in order of increasing package specialization with respect to the project. `java` imports should be listed first, followed by `javax` imports and any other non-Oracle imports. Then Oracle imports should be listed, again in order of increasing specialization - packages developed on other projects first followed by packages developed on the current project. Multiple imports from a particular package should be listed alphabetically.

For example:

```
import java.util.Stack;
import java.util.Enumeration;

import oracle.abc.Utility;

import oracle.iam.access.Employee;
import oracle.iam.access.Location;
```

JDeveloper can automatically sort the import statements in a class, removing duplicates and unused imports. Right click in the code editor and select *Organize imports...* and then one of the options.

Note that in general import statements should be used rather than the use of fully qualified class names within code. This is unwieldy and, since the class name may be mentioned in several places, difficult to maintain as the code evolves. The use of two identically named types (for example, `java.util.Date` and `java.sql.Date` is an exceptional case in which the more frequently used type should be imported and the least used type fully qualified within the code.

Java Standards

Packages

Applicability

All classes must belong to a particular package, i.e. each source file must contain a package statement. If using J2SE 1.4 and later all classes must be in a package.

Naming

Package names should be all lower case.

Package names should begin with `oracle` followed by the project name (on external projects it may be more appropriate to start package names with the customer's name).

Package names belonging always to a specific product line.

Product Line	Package
Identity Manager	`oracle.iam.identity`
Access Manager	`oracle.iam.access`
Unified Directory	`oracle.iam.directory`
Internet Directory	`oracle.iam.directory`
Virtual Directory	`oracle.iam.directory`
Integration Directory	`oracle.iam.directory`

Classes

Declarations

At the beginning of the class definition, any extends declarations on the same line as the class declaration and any implements on the following line, for example,

```
public class GeneratorEngine extends    Object
                             implements Generator {
```

Naming

Class names should be a noun or noun phrase clearly representing its purpose but not overly long. Class names should be written in UpperCamelCase i.e. the first letter of every word in the name should be capitalized. For example,

```
```

Constructor

A default constructor should be explicitly supplied for each class, unless there is a good reason for not providing one (in which case the reason must be documented in the code).

Be sure to call the appropriate superclass constructor, using super() and the relevant arguments, as necessary.

Final classes

Since the final keyword prevents any further derived classes of a class it requires a strong justification to be used (typically, security; optimization reasons are not sufficient). As a case in point, `java.util.Vector` is declared as a final class, which means that you cannot derive from it, even though it would be very useful to do so.

Documentation

Every class in a source file must be preceded by javadoc comments that describe the purpose of the class. This should include the author, file version and links to any related packages (the source control system may be able to generate the first two automatically). The first sentence of the javadoc should be on a separate line as it appears as a summary of the class javadoc.

```
/**
 ** Object to encapsulate content settings in a project's configuration.
 ** <br>
 ** This object is added to the common data of a project when it is first
 ** given a provider.
 ** <br>
 ** Provides access to content set information such as the root content
 ** path under which the objects .xml files are stored.
 ** <br>
 ** This class use a hash data structure intended to be used as the
 ** generic storage for project metadata that can be marshalled to and
 ** from a persistent form without depending on custom marshalling code.
 **
 ** @author dieter.steding@oracle.com
 ** @version 12.2.1.3.42.60.84
 ** @since   12.2.1.3.42.60.83
 */
```

Interfaces

Naming

Interface names should be descriptive adjectives clearly representing their purpose but not overly long. According to the defined principles, they should never begin with a capital ‘I’ to visualize that this is an interface rather than a class. The first letter of every word in the name should be capitalized.

Documentation

Interfaces should be documented as described for classes.

Methods

All methods should have one specific task and be small in length. This will encourage methods to be generic and enhance the possibility of code reuse.

Method prototypes should have the opening and closing braces at the same level, for example:

```

```

Miscellaneous

Magic Numbers

There should be no hard coded ‘magic numbers’ within source code. All constant values should be declared as static final class variables and a comment included with the variable definition as to what it is used for.

Initialization

All variables should be explicitly initialized, at their point of declaration. Without explicit initialization the default type initialization is used, which may cause problems if someone subsequently changes the variable’s type.

One statement per line.

Code should be laid such that there is only one statement per line. This improves readability and maintainability and makes debugging using a source code debugger easier. For example the following code fragment:

```
if (state) { setEnabled(state); }
```

should be written as follows:

```
if (state) {
 setEnabled(state);
}
```

Ternary Operator

Used wisely, the ternary operator yields concise, readable code. It should only be used when the expressions involved are simple and their purpose clear. Parentheses should be used to aid readability. For example,

```
String stateLabel = (a.isModified()) ? "Modified" : "";
```

Dangling Else Problem

Braces must always be used to delimit the branches of an if-else statement, even if the branch consists of a single statement. This avoids the so-called ‘dangling else’ problem where even though the code layout and indentation suggests one mode of execution the reality is different. For example, consider the following code fragment:

```
if (a == 1)
 if (b == 3)
  myString = “hello”;
else
 myString = “goodbye”;
```

The indentation suggests that the `else` clause is intended to match the first `if` (i.e. `a == 1` but in fact it will be interpreted in the context of the second `if` (i.e. `b == 3`. This ambiguity is avoided if braces are used:

```
if (a == 1) {
 if (b == 3) {
  myString = “hello”;
 }
}
else {
 myString = “goodbye”;
}
```

The same principle applies to other flow control and looping constructs such as switch statements (see section xxx.xxx), for, do-while and try-catch statements, in that all blocks of code used in these constructs must be delimited by braces - even if the block only consists of one line of code.

Switch Statements

The `case` labels in a switch statement should be indented to the next level as the `switch` label - these are all parts of the same statement, much as the `if` and `else` labels in an if-else statement have the same level of indentation.

Each case statement must have its own `break` statement, unless it is explicitly intended that execution fall through to the next case statement. If fall through is intended a comment must be included stating this.

All switch statements should have a `default` case statement; even if at the time of writing, no circumstance is envisaged where the `default` will be called. As programs evolve, domains inevitably change and the presence of a `default` statement will help catch potential problems.

The code associated with each `case` statement should be delimited by its own set of braces. This avoids unexpected problems with regard to the scope of variables. For example, the following code fragment will not compile, since variable `i` is declared twice at switch-level scope:

```
...
```

```
case 1: int i = 0;
    // Some code using i...
    break;
case 2: int i = 0; // Will not compile
    // Some further code using i...
    break;
```

whereas with the addition of case-level braces the code will compile:

```
...
case 1: {
     int i = 0;
     // Some code using i...
    }
    break;
case 2: {
     int i = 0;
     // Some further code using i...
    }
    break;
```

Empty Statements

There are situations where a statement is not required even though in general a statement would exist. Typical this involves a looping construct where the necessary processing is done in the predicate of the construct itself. Such scenarios must be fully commented since they may not be immediately obvious to anyone maintaining the code. In addition, consideration must be given to rewriting the code in such a way as to improve its legibility. For example:

```
for (int i = 0; i < 10; a[i++] = i) {
 // intentionally left blank
}
```

should be written

```
for (int i = 0; i < 10; i++) {
 a[i] = i + 1;
}
```

String Manipulation

Intensive manipulation of strings should be performed using the `StringBuilder` class rather than `String` as this is usually more efficient. Instances of `String` are immutable, and operations such as concatenation are achieved using `StringBuilder` objects behind the scenes. For example,

```
String s = new String();
s = "One,";
s = s + "two,";
s = s + "three";
```

```
System.out.println(s);
```

will probably result in the creation of several temporary `StringBuilder` objects, whilst a single `StringBuilder` object can be used to achieve the same effect:

```
StringBuilder builder = new StringBuilder();
builder.append("One,");
builder.append("two,");
builder.append("three");
System.out.println(builder.toString());
```

Garbage Collection

Java's garbage collector in general can be relied upon to reclaim the memory holding an object when no handles to that object remain. However sometimes handles to an object continue to exist even when the object has ceased to be useful (for example, an object created and used at the start of a method but ignored for the rest of the method). When large objects or arrays to become useful it is good practice to explicitly set all handles to them to null. This will allow the garbage collector to reclaim the associated memory as soon as possible.

A class can define a finalize() method which will be called by the garbage collector when an instance of the class is being reclaimed. However, Java makes no guarantees about when garbage collection will occur or what order objects will be collected in. Indeed, an object may not be garbage collected at all. This means that the finalize() method is of very little use and in particular should not be used to free non-memory resources such as file descriptors associated with an object.

The try/catch/finally method discussed in section xxx.xxx should be used to free resources instead. For example a database connection should be closed in a finally block so it will be released irrespective of whether the JDBC code completed successfully or not.

Static Declarations

When the value of a static class variable is changed, all instances of the class (within a single Java Virtual Machine (JVM)) see the new value. Care should be taken when using static member variables to ensure that this is the desired behavior.

Note

Static final variables are set at the point of declaration (to act as immutable constants) and cannot be changed later.

The use of static methods should also be considered carefully, since static methods can only reference static class variables and for the reasons already discussed these are potentially problematic.

Classes used in a completely static manner are reasonable when implementing pools of objects, shared resources and factory methods. However, bear in mind that a static attribute is only static within a single JVM, which means they cannot be relied upon in multi-server and multi-process environments such as Oracle Application Server. To ensure that classes such as these are only used in a static manner and never instantiated, the class constructor can be declared private.

When referring to a static attribute or class method, use the form:

```
ClassName.ATTRIBUTE_NAME
```

Exception Handling

Make sure that exceptions are handled at the correct level. If a method does not have sufficient knowledge to know how to treat an exception, it should pass it up to its calling method to deal with.

Catch blocks should only handle checked Exceptions that are thrown in the preceding try blocks. A catch block should not catch `java.lang.Exception` or `java.lang.Throwable` this would lead to masking Errors or RuntimeExceptions. The only exception to this rule is if the module is some form of server that must continue operation irrespective of any fault.

Each catch block must either handle the Exception or pass it up by rethrowing it wrapped as another Exception. Catch blocks should always log the Exception to aid debugging.

Methods should never declare that they throw `java.lang.Exceptiony` as this forces code that uses it to catch `java.lang.Exception`. Methods should always declare that they throw a specific sub class of Exception. In most cases, this would be a custom Exception for the module.

```
try {
 // Do something
}
catch (IOException e) {
 // do something specific for this type of exception
 e.printStackTrace();
 throw new SpecificException(e);
}
finally {
 // if anything always has to be done, even if an exception
 // is caught, do it here...
}
```

Further, if the number of exceptions to be caught is large, or the processing involved is substantial, consider abstracting it out into a separate method.

The `finally` part of a try/catch statement always called even if an exception is thrown in the `try` part. If your code locks or grabs a resource, such as opening a file or a database connection, you should use a try/catch/finally construct to ensure the resource always freed. The general construct would be:

```
// grab resource here
try {
 // use resource here
}
// repeat for all exceptions
catch (IOException e) {
 // handle exception
}
finally {
 // free the resource here
}
```

Argument checking

A useful defensive programming technique is for all methods to check the validity of their arguments, throwing an `IllegalArgumentException` when an invalid argument is received. This is illustrated in the example below.

Note
Since `IllegalArgumentException` is derived from `RuntimeException` it is not necessary to declare that the function may throw this exception.

```
public void addMoney(int total) {
 if (total < 0)  {
  throw new IllegalArgumentException("'total' cannot be negative");
 }

 // ...do something with ‘total’
}
```

Thread Safety

It is possible that Java code written on a project will be run in multi-threaded mode, the same instance of a class will be accessed by more than one client via a thread... E.g. Servlets in WebLogic or JSF Action classes. For this reason, thread safety must be a major consideration.

Consider the following class definition:

```
public class MyClass {
 public void myFunction()  {
  int l = 10;
  // Some code...
 }

 private static String NAME = null;
 private float value = 1;
}
```

In this example, in a single JVM:

- All threads running through all instances of this class see the same value of the static variable `NAME`.
- All threads running though a particular instance of this class see the same value of the instance variable `value`
- All threads get their own stack, so each thread has its own copy of any local variables such as `l` in `myFunction()`.

As described in section xxx.xxx, static variables should generally be avoided even in single-threaded situations, since they are shared across class instances. Further, in multi-threaded mode instance variables should be avoided unless you intend all threads running through a single class instance to see the same value. Put another

way, two threads running through a single class instance cannot maintain different states, i.e. different values of static or instance class variables.