JAVA CODING GUIDE LINES

TITLE	**XSL Coding Guide Lines & Best Practices**

REVISION HISTORY

Revision Number	Date	Author(s)	Description
0.1	19 Dec.2023	Rajashekar Tuppada	Initial Draft
0.2	18 Jan 2024	Rajashekar Tuppada	Updated Version
1.0	14 Feb 2024	Rajashekar Tuppada	Final Version

Table of Contents

1 Introduction

This document documents the Java Coding Standards that will be followed during the development of the Mountain Gear Project.
In 95% of coding situations, standards can help make code easier to read. However, in a situation where there is no clearly defined standard DO WHAT MAKES SENSE.

- Do what makes sense to make the code more readable.
- Do what makes sense so that a novice Java programmer can understand what your code is doing.
- Do what makes sense for the client who will ultimately maintain the code.

1.1 Purpose

Standards are critical because they:

- Make code easier to read
- Make code easier to understand
- Provide consistency of documentation (e.g. comments)
- Help future developers get-up-to-speed more quickly
- Improve ease of maintenance

2 Naming conventions

2.1 Package Names

- The entire package name should be in lower case.
- The package name reflects the logical directory structure delimited by periods

Examples:

- com.hybris.web.services
- directory structure com\hybris\web\services

2.2 File name and Class Name

The Java compiler requires that the file be named the same as the class it contains (case sensitive). Only one public class or interface can be specified in a single compilation unit (i.e. file).

Each file must end with a ".java" suffix

Examples:

- MountainGearConstants.java
- PaymntInfoServlet.java
- CacheManager.java

- Use full English description for class name, with the first letter capitalized and the first letter of all subsequent words capitalized.
- For classes that use the "Facade" design pattern, the name should end with "Facade".
  - WishlistDAO.java

- CartFacade.java

- For classes that use the "Factory" design pattern, the name should end with "Factory".
- For Data access use "DAO" as below:
  - WishlistDAO.java (the interface),
  - WishlistDAOImpl.java (implementation)
- For any custom controllers, end the class name with "Controller". Example:
  - AddToCartController.java
  - AddToWishlistController.java

Here is a sample list of class name suffixes, built from browsing existing Java API classes -- this makes a good starting place when thinking of names for service-oriented classes.

Examples:

Acceptor	Factory	Modifier
Assembler	Filter	Parser
Authenticator	Formatter	Reader
Checker	Generator	Scanner
Chooser	Helper	Signer
Compiler	Handler	Tracer
Decoder	Listener	Transformer
Dispatcher	Loader	Transporter
Encoder	Manager	Writer

- CacheManager
- CacheService
- CacheFilter
- MountainGearLogger

2.3 Interface Names

There are coding standards that stipulate that interface names start with a capital "I" so that they are easily distinguished as interfaces. We have dropped this requirement and impose no restrictions on the developer other than that the interface name should accurately reflect the purpose of the interface. Another line of reasoning is that one should not be aware that he / she is using an interface in their code, and to them it should have exactly the same behavior as a regular class.

Examples:

- Destination
- Cacheable

Implementation class for the interface should have Impl suffixed to the interface name. Example if the interface is Cache.java, the implementation class should be CacheImpl.java.

2.4 Method Names

In Java, there are two types of methods in a class.

- Class methods – declared with the "static" keyword. Class methods can be called even if there are no instances of a class being used in the method invocation
- Instance methods – cannot be called unless there is an instance of the object

For both class and instance methods, we will follow the standard Java naming convention

- First letter of the first word is in lower case and the first letter of all subsequent words in uppercase.
- Start methods with an active verb whenever possible

Examples:

- checkPrice()
- searchItem()

Accessor and Mutator methods (a.k.a. "getter" methods and "setter" methods, respectively):

- Use the Java Bean naming specification. The synopsis below should prove sufficient. For more details, visit http://java.sun.com/products/javabeans/docs/spec.html
- Access to member variables from outside a class should always be done through getter and setter method (rather than having public member variables that other classes can freely access)
- Getter and setter should be declared when it makes sense; there is no need to create getter and setter that nobody will ever use. Move getters and setters at the end of class.
- The get methods will be named as follows:

  "get" + <member_variable_name>

- The set methods will be named as follows

  "set" + <member_variable_name>

Example for a class with a member variable named "productId":

- getProductId();
- setProductId(Long productId);

For boolean getter:

- do not use "get" to prefix the accessor
- use "is" + <member_variable_name> to prefix the accessor for readability

Example:

- isActive()
- isPreferred()

A class's layout should be as follows:

- Class variables and constants - declare these in order of visibility by first the private, then protected, and lastly public variables/constants.
- Constructors and constructor helper methods by functionality.
- Methods by functionality.
- Inner classes by functionality.

The different sections of the class should be organized by functionality. For instance, if a constructor uses a private helper method, place the method close to the constructor within the class.

2.5 Constructors and Destructors

- Java requires that the constructor methods use the same name as the class.
- Only use default constructors (i.e. constructors without any parameters) when required by a business case (e.g. do not create a constructor for a rectangle object called Rectangle() unless you can have a valid rectangle object in the system that does not have a height, width, color, etc.). Always have constructors reflect the required attributes for a business object. The default constructor may be required in the class if it needs to be serialized, or the class is referred using reflection API's.
- Java does not have destructors. If you need to free resources, do this in the finalize() method.

  Example:

  CacheManager () is the constructor for the CacheManager class (assuming you can have an Cache Manager without any required arguments).

2.6 Variable Names

For all variables:

- Use a descriptive, unambiguous name, with the first letter in lowercase, and the first letter of all subsequent words in uppercase.

Examples:

- price
- shareName

2.7 Member Variables

In Java, there are two types of member variables for classes.

- Instance variables – every instance can have these member variables. Not declared with "static" keyword
- Class variables – declared with the "static" keyword. Class variables exist even if there are no instances of a class, and the class variables are shared across all instances of a class

For instance variables:

- Always prefix instance variables with "this." (or if necessary, "super") to distinguish it as an instance variable. It is important to distinguish instance variables from local variables, and using Java's built-in OO mechanism for doing so is the best solution. As with all coding standards, this convention needs to be agreed upon and understood by all team members and strictly enforced in all code reviews. The implications of not following this particular standard could be disastrous (see "name hiding" in the local variables section below).

Examples:

- this.price
- super.className

2.8 Declaring constants

2.8.1 CLASS LEVEL CONSTANTS

- Class specific literals- should be defined at the class level as *private static final*
- Constants should be in all capital letters with underscores between words and be declared as static and final. Notice that constants may be objects or primitives (int, boolean). In the case of objects, it is difficult to ensure that they remain constant. In the case of a constant string, I can call the method .toUpperCase(), in which case it is no longer constant. Even though the object may change, we expect them not to and still consider them constants.

- Constants should live in the file where they make the most sense.
- Constants should be listed at the very top of the class (before the constructors).
- If you are hard-coding a value (e.g. if dbname=="test" or while (counter < 3) , you are doing something wrong.

Examples:

- public final static double PI = 3.14;

For class variables:

- Class variables should be preceded with the Class (not the instance/object) name. Although the code will compile if you use the instance name (so beware), if a variable is preceded by the object name it can be immediately recognized as a class variable and not an instance variable or a local variable for a method.

Example:

- MGLocale.langCode

2.8.2 COMMON CONSTANTS CLASS

- All common constants at the application level should be defined in the com.mg.constants.MGConstants class.

2.9 Enumerated Types

- The use of Enums as supported in JDK 6 .For more information refer to the following link http://java.sun.com/javase/6/docs/api/java/util/Enumeration.html

2.10 Local Variables

- Do not name local variables the same as instance variables! This is referred to as "name hiding", and it can introduce dangerous, difficult-to-find bugs in the system. As with all coding standards, this convention needs to be agreed upon and understood by all team members and strictly enforced in all code reviews. The implications of not following this particular standard could be disastrous.
- Loop counters should be called with logical name.
- Avoid using "temp" to prefix a local variable. Call it something that better reflects its purpose.
- Scope variables as narrowly as possible. For instance, if a variable is only used within an if block, declare the variable within the block. Similarly, declare loop iterators within the loop when possible.

Examples:

- currentTotal
- newAccount

```
if(condition) {
  String name;
   ...
}
```

-

```
public double calculateNetSalary(double salary){
        double netSalary;
        double deduction = 0.15;
        netSalary = salary – (salary * deduction);
        return netSalary;
}
```

-

```
int maxLoop = MyClass.MAX_LOOP;
for (int i=0; i < maxLoop; i++) {
                ...
        }
```

2.11 Final Local Variables

If you have local variables, especially if you are using them for looping, declaring them final is fine if the situation calls for it.

3 Formatting

3.1 General

- Use 4 spaces for each indented line. Do not use tabs, as the interpretation of a tab is text editor dependent (Note that many text editors will automatically converts tabs to spaces for you).
- Separate code blocks with a blank line.
- When a statement is too big to fit on one line, indent the second line. Any subsequent lines should use the same indentation as the second line. Do not rely on the text wrapping by the text editor as the code looks ugly. Our standard assumption is that the length of a line is 80 characters.

3.2 Control-of-Flow Statements

- Start the curly brackets at the end of the line that starts the block and align the end brackets with the statement that starts the block.
- Always use brackets around control-of-flow blocks (e.g. "if", "while", etc) even if there is only one line of code within the block. This lessens the risk of error if more code must be later added to the block.
- Second and subsequent "else if" clauses should be on the same level as the opening "if" clause.
- When there are three or more end brackets in a row on their own lines, label the brackets with in-line comments to clarify the block they are ending.

Example of good formatting:


```
/** standard block */

public class MGLogger extends Logger {

  public MGLogger () {
    super();
  }

  public void doSomething() {
    try {
        if (false) {
            System.out.println("blah");
        } else {
            System.out.println("something else");
        }
    } catch (Exception e) {
            System.out.println(e);
    } // end try catch
  } // end doSomething method
} // end class
```

Example of non-acceptable formatting:

```
/** non-standard block */
public class MGLogger extends Logger
{
        public MGLogger () { super(); }
        public void doSomething
        {
                if (false)
                        System.out.println("blah");
                else {
                        System.out.println("something else");
                }
        }
}
```

4 Comments

4.1 How to Javadoc your code

This is a must read:
http://java.sun.com/products/jdk/javadoc/writingdoccomments/index.html

4.2 Classes and interfaces

Every class or interface should have a standard Javadoc class/interface comment header which:

- details the functionality
- references the other classes it uses using Javadoc @see tag
- includes its copyright and date using the Javadoc @copyright tag

- specifies a version using the Javadoc @version tag
- documents the original author and creation date using the Javadoc @author tag

Example:

```
/*
 * HTMLParagraph
 *
 * Purpose of this class is to layout the HTML . . .
 *
 * @author  Name
 * @version 1.1
 *
 * @Copyright (c) 2010 Mountain Gear. All rights
 * reserved.
 * @see  com.mg.html.HTMLComps
 *
 */

public class HTMLParagraph {

   ...

}
```

4.3 Methods

Every method declaration should have a Javadoc method comment header which:

- describes the method functionality , the contract of this method
- explains all the parameters using the Javadoc @param tag
- specifies the return value (if applicable) using the Javadoc @return tag
- defines all exceptions (if applicable) that the method throws using the Javadoc @throws tag

Example:

```
/**

 * Wraps input text with paragraph HTML tags

 * @param content the HTML string that goes

 * between the P tags

 * @return String the input string wrapped with <P> tags
```

```
 */
public String createHTMLParagraph(String content) {
   StringBuffer returnString = new StringBuffer("<P>");
   returnString.append(content);
   returnString.append("</P>");
   return returnString.toString();
}
```

4.4 Member Variables and Constants

Every member variable should have Javadoc comments.

Example:

```
/** the HTML text that is contained within the P tags */
private String contents = null;
```

4.5 Other recommendations

4.5.1 WHICH TYPE OF COMMENT AND WHEN

- Use Javadoc (/** ... */) for documenting classes, interfaces, member functions, member variables, and constants
- Use C-style comments (/* . . . */) to comment out chunks of code.
- Use inline comments (//) to specify business functionality within a method

4.5.2 MULTI-LINE COMMENTS

Whenever you are commenting something that spans multiple lines, use the following format:

Examples:

```
/*
 * This comment is so long, that I cannot fit it on one line.
```

```
 * In order to keep the comment readable, use this format.

 */

// This comment is so long, that I cannot fit it on one line.

// In order to keep the comment readable, don't use this format.
```

4.5.3 SINGLE-LINE COMMENTS

If your comment does not span multiple lines, use the following format

Examples:

```
/* This comment is short and sweet. */

// This comment is short and sweet.
```

4.5.4 REMINDER COMMENTS

- Use comments for any complex chunk of code.
- When you are commenting code that you need to come back to at some time, use the following commenting pattern:

```
/* TODO – Description – Name */
```
to make sure you get to everything you want to.

Example:

```
/* TODO – Remove the stub and replace it with the actual data base access code – Name */
```

5 Access Protection to Methods / Member Variables

Java provides an elegant solution to the namespace conflicts by introducing Java "packages". In addition, packages provide a default access privilege for methods / member variables that allows classes within the same package to access these methods / member variables.

- No member variables should be public (with the exception of public static member variables such as global constants) so that encapsulation is enforced for our code
- We will explicitly declare all methods / member variables as one of the following (except in rare cases) as opposed to using the default access privilege for Java.
  - public
  - protected

- private

This means that there will have to be public accessor methods for protected or private member variables of a class that is being used by another non-subclass class

It is acceptable for subclasses to directly reference protected member variables of a super class using super prefix.

6 Logger Usage

- Use appropriate logger methods in your code – error, warn, info and debug.Use of these methods need to be preceded by a check to see if the log level is enabled. This ensures that we do not take the hit for Object serialization in case the log level is not enabled.
- e.g:

```
if(logger.isDebugEnabled()){
    logger.debug("…");
}
```

References:

- The Elements of Java Style, Allan Vermeulen et al, Cambridge University Press 2000
  - The definitive guide for Java coding standards.
  - http://www.ambysoft.com/elementsJavaStyle.html
- Design Patterns, Gamma, Helm, Johnson & Vlissides, Addison-Wesley 1995
  - http://www.amazon.com/exec/obidos/ASIN/0201633612/o/qid%3D941139992/sr%3D8-1/104-7679297-0353554
- How to Write Doc Comments for the Javadoc Tool
  - http://java.sun.com/j2se/javadoc/writingdoccomments/index.html
- JavaBean Specification
  - http://java.sun.com/products/javabeans/docs/spec.html

Appendix A

A. Common Mistakes from Code Reviews

- Make sure to understand when to use prefix "this." vs. "super."
- Open Javadoc comments with /**, close with */ and use a * as the left border
- Put comments in for commented-out code. Use C-style commenting only to comment out obsolete code. For all other commenting, Javadoc or inline commenting should be used.
- Do not have custom log() methods in the class. Use standard logging service appropriately instead.
- Avoid using scriptlets in JSF/JSP.
- Use correct javadoc for @param in the format:

  Incorrect:

```
@param String

@param ftFirstName
```

  Correct:

```
@param ftFirstName First name of the FT  ???????
```


- Don't initialize objects that are returned by methods, e.g.

```
// Incorrect

 AddressVO address = new AddressVO();

  address = dao.getSeasonalAddress();
```

  In this case no need to initialize address. This should be done like this :

```
// Correct

 AddressVO address = dao.getSeasonalAddress();
```

B. Performance Improvement Guidelines

- Use StringBuffer rather than String for concatenation.

- Using the string class, concatenations are usually performed as follows:

```
String addedString = new String ("Stanford  ");

addedString += "Lost!!";
```

If you were to use StringBuffer to perform the same concatenation, you would need code that looks like this:

```
StringBuffer addedString = new StringBuffer("Stanford")

addedString.append("Lost!!");
```

Using the StringBuffer is significantly faster, especially while doing concatenations.

More Info :

http://www.javaworld.com/javaworld/jw-03-2000/jw-0324-javaperf.html

- Reuse Objects wherever possible

Most container objects (e.g. Hashtables) can be reused rather than created and thrown away. Pool management strategy comes in handy in such kind of a scenario.

While recycling objects, you need to dereference all the elements previously in the container so that you don't prevent them from being garbage collected. This technique negates the employment of resources to create new objects which is a significant bottleneck.

We can have a common utility that will act as a pool manager. This utility can be used from the code as follows:


```
Public void someMethod() {
    HashMap textBoxMap = poolManagerUtility.getHashMap();

    // do Map manipulation stuff

    // extra work of explicitly tell the pool manager that we    have finished
the vector

    poolManagerUtility.returnHashMap(textBoxMap);
}
```

More Info: http://www.oreilly.com/catalog/javapt/chapter/ch04.html

- Use Static final private methods to allow in-lining

  Use of static final modifier enables in-lining which is performance savvy.

```
public static final void log(Object aText) {

   System.out.println(aText.toString());

 }
```

  One negative aspect of doing this is that you will have to compile all the dependent files every time you change the source code of the final method.

- Use System.arraycopy for copying arrays

  Consider the following arrays

```
int[] first = {1, 2, 3};

int[] second = {4, 5, 6};

int[] third = new int[first.length + second.length];
```

  One possible way of copying the contents of the first two arrays into the third one is using loops, such as:

```
 for (int i = 0; i < first.length; i++)
          third[i] = first[i];

     for (int i = 0; i < second.length; i++)
          third[first.length + i] = second[i];
```

  However a better way of accomplishing the same task is by the use of System.arraycopy

```
  System.arraycopy(first, 0, third, 0, first.length);

    System.arraycopy(second, 0, third, first.length,second.length);
```

- Lazy Evaluation

  It can often be optimal in computing to procrastinate computation to the last possible millisecond. You don't create an object until you actually need it. Just before every use, you might execute code roughly like this:

```
if (singleton == null){
```

```
    singleton = new something();
    // further initialization
}
```

This will reduce the scope of the object to minimum possible extent. In some cases, depending on flow, the logic might not even reach here and the object would not even get created.

Following are the best practices:

- Postpone opening files until you are sure you need to read records.
- Mark a list as to-be-sorted, rather than sorting it right away. Don't bother sorting until someone actually makes a request.
- Use Generic data types supported by JDK 5. For more details visit: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
- Minimize data in HTTP sessions
- Do not store redundant variables in the HTTP session unless it is absolutely necessary. As the session data replicated on the cluster, it should not be made unnecessarily heavy.
- Wherever possible, look to minimize synchronization.
- Too much synchronization leads to slow code and/or deadlock.
- If you must synchronize an entire method, use the synchronized method modifier instead of the synchronized block.
- Use vectors wherever there is a need for synchronized arraylist.
- Optimize source code by hand
- Dead code removal :

  Consider the following snippet of code:

```
int j =5;

int[] a = new int[10];

for(int i=10;i<10;i++){
  a[i] = j;

  if(j == 5)        {
      a[i] =25;
    }
  else   {
      a[i] =10;
   }
```

```
}
```

The logic in the loop leads to bigger byte code and piece of code that never executes. The loop always assigns 25 to a[i]. The code should be rewritten like this:

```
int[] a = new int [10];
for(int i=10;i<10;i++){
    a[i] =25;
}
```

The re written loop has smaller byte code and is faster than the original loop.

- Strength reduction

  Strength reduction involves replacing expensive operation with a more efficient one. A common optimization is to use the compound assignment operators.

```
int x=5;

int[] anInt = new int{N};

for(int i=0;i<N;i++){
    anInt[i] = anInt[i] +x;
}
```

The generated byte code for this is slower than

```
int x=5;

int[]anInt = new int{N};

for(int i=0;i<N;i++){
    anInt[i] + = x;
}
```

- Constant folding

  Compiler performs simple constant folding. Data however should be declared as final wherever posible for compiler to optimize it:

```
static int a= 30;

static int b= 60;
```

```
int c = a + b + 100;
```

is compiled into a larger and smaller byte code than:

```
final static int a= 30;

final static int b= 60;

int c = a + b + 100;
```

Because the variables are declared as final the addition takes palce at compile time and the byte code works with a constant value.

- Subexpression Elimination

```
Obj[] someObj = new Obj[N];

someObj[i+j].someMethod(k);

someObj[i+j].someMethod(k+1);

someObj[i+j].someMethod(k+2);

someObj[i+j].someMethod(k+3);

someObj[i+j].someMethod(k+4);

someObj[i+j].someMethod(k+5);
```

Replace with:

```
Obj[] someObj = new Obj[N];

Obj nameObj = new Obj();

nameObj = someObj[i+j];



nameObj.someMethod(k+1);

nameObj.someMethod(k+2);

nameObj.someMethod(k+3);
```

```
nameObj.someMethod(k+4);

nameObj.someMethod(k+5);
```

For many iterations the optimized code is twice as much faster as the original code and it is smaller.

- Common sub-expression elimination
- Loop unrolling

  Unrolling loop has tha advantage of eiliminating the code for the loop construct and branching, thereby resulting faster execution. Disadvantage - more code. This needs to be used judiciously.

  A typical small loop can be:

```
int[] ia = new int int [4];
for (int i=0;i<4;i++)
{
    ia[i] = 10;
}
```

  Unroll it as:

```
int[] ia = new int int [4];

ia[1] = 10;

ia[2] = 10;

ia[3] = 10;

ia[4] = 10;
```

- Algebraic simplification

  Replace expressions like:

```
int x = f*a + f*b + f*c + f*d
```

  as

```
int x = f*(a+b+c+d)
```

- Loop invariant code motion

  Move expressions that are in a loop but can be safely moved outside a loop (especially object creations).

```
int a= 10;

int b = 20;

for (int i=10;i<val;i++)
{
    arr[i] = a+b;
}
```

Rewrite as:

```
int a= 10;

int b = 20;

int c =  a+b;

for (int i=10;i<val;i++)


    arr[i] = c;
```