**Java Coding**

**Standards / Guidelines / Best Practices**

### Version 2.0

Sep, 2019

# Revision History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version**  **No.** | **Author** | **Published**  **Date** | **Sections**  **Changed** | **Description of changes** |
| 1.0 | Narayana, Patti Veera Venkata | Aug-2019 | All | Common / important coding standards & guidelines as a ppt for  Java / J2EE |
| 2.0 | Rasool, Ansari Shaik | Sep-2019 | All | Extension of coding standards to cover all critical and commonly used coding standards, guidelines and best practices. Separate coding standards based on technology or file type – Java, JSP, EJB/MDB, JS. Conversion to documents for each technology |

### Table of Content

[Introduction 5](#_TOC_250060)

1. [File Structure & Naming Conventions: 6](#_TOC_250059)

[A1) All Java files to align to the standard file skeleton 6](#_TOC_250058)

[A2) All packages to adhere to the standard naming conventions 8](#_TOC_250057)

[A3) All classes to adhere to the standard naming conventions 8](#_TOC_250056)

[A4) All methods to adhere to the standard naming conventions 8](#_TOC_250055)

[A5) All variables to adhere to the standard naming conventions 9](#_TOC_250054)

1. [Core Programming 10](#_TOC_250053)

[B1) Perform validations on method parameters (passed by user or another application) before executing method logic 10](#_TOC_250052)

B2) Handle all error, boundary, abnormal & special conditions corresponding to the functionality11 B3) Safeguard against the very popular NullPointerException 12

[B4) Code with optimized memory management 14](#_TOC_250051)

[B5) Compare objects for equality the right way 15](#_TOC_250050)

[B6) Use String Buffer concatenation instead of String addition 16](#_TOC_250049)

[B7) Override toString() method for all transfer objects and JavaBeans 17](#_TOC_250048)

[B8) Minimize the accessibility of class variables (member variables) 18](#_TOC_250047)

[B9) Implement Comparable Interface to compare Objects. 19](#_TOC_250046)

[B10) Avoid Shell scripts inside java code 19](#_TOC_250045)

[B11) Use HttpSession correctly 19](#_TOC_250044)

[B12) Do not use application server specific APIs 19](#_TOC_250043)

[B13) Do not hard-code values 20](#_TOC_250042)

B14) Do not modify method parameters that are meant to be “input-only” reference parameters 20 B15) Safeguard against common pitfalls of using control structures 20

1. [Exception Handling & Logging 22](#_TOC_250041)

[C1) Handle checked and un-checked exceptions correctly 22](#_TOC_250040)

[C2) Ensure correct “throws” clause for the methods that detect / propagate exceptions 23](#_TOC_250039)

[C3) Catch all possible checked exceptions that can be thrown from the block of code 24](#_TOC_250038)

[C4) Never throw an exception from finally block 25](#_TOC_250037)

[C5) Never keep a catch block empty 25](#_TOC_250036)

[C6) Handle exception stack trace correctly 25](#_TOC_250035)

[C7) No use of System.out.println / System.err 27](#_TOC_250034)

[C8) Program to logger components using a facade 28](#_TOC_250033)

[C9) Use appropriate logging modes to print various messages to the log file 28](#_TOC_250032)

[C10) Prevent un-necessary heap overload while logging 29](#_TOC_250031)

1. [JDBC 30](#_TOC_250030)

[D1) Use only a DataSource for retrieving a connection in an app server or web server environment](#_TOC_250029)

........................................................................................................................................................ 30

[D2) Use Prepared statements instead of Statements 31](#_TOC_250028)

[D3) Externalize all SQL in a properties / XML file 32](#_TOC_250027)

[D4) Close all database resources (connections, statements, result sets) in finally block 32](#_TOC_250026)

[D5) Do not change default database connection properties in an app server environment 34](#_TOC_250025)

[D6) Use PreparedStatement’s BatchUpdate feature while performing JDBC updates for multiple records 34](#_TOC_250024)

[D7) Do not perform database operations in a loop 35](#_TOC_250023)

1. [Input / Output: 37](#_TOC_250022)

[E1) Close all I/O resources in the finally block 37](#_TOC_250021)

[E2) Optimize I/O performance using appropriate API 38](#_TOC_250020)

1. [Data Structures 39](#_TOC_250019)

[F1) Use Generics with Collection 39](#_TOC_250018)

[F2) Use Non-thread safe collection classes rather than thread-safe collection classes where possible](#_TOC_250017)

........................................................................................................................................................ 39

[F3) Use appropriate collection classes based on requirement 40](#_TOC_250016)

1. [Multi-user aware programming, Multi-threading & Concurrency 41](#_TOC_250015)

[G1) Do not maintain state (member variables) for classes for which shared instances are maintained (i.e. intended to be stateless) 41](#_TOC_250014)

[G2) Updates to the cache or any shared state should always be synchronized 42](#_TOC_250013)

[G3) Ensure minimum performance overhead when using “synchronized” 43](#_TOC_250012)

[G4) Do not use Java Threads in app server environment 44](#_TOC_250011)

[G5) Handle concurrent updates / deletes to a database record 44](#_TOC_250010)

[G6) Adhere to multi-threading best practices when using Java multithreading (in non-app server based Java apps) 44](#_TOC_250009)

1. [Implementation of design practices 48](#_TOC_250008)

[H1) Implement intended design level flexibility 48](#_TOC_250007)

[H2) Align to multi-layered design or separation of concerns 48](#_TOC_250006)

[H3) Implement the design patterns correctly 49](#_TOC_250005)

[H4)Do not assume English as the language while displaying labels or reading data from HttpRequest or formatting data for display on screen 50](#_TOC_250004)

1. [Maintainability, Ease of Support 52](#_TOC_250003)

[I1) Do not duplicate code 52](#_TOC_250002)

[I2) Ensure code that is easy to change 52](#_TOC_250001)

[I3) Implement logging and exception handling standards 52](#_TOC_250000)

J) [Webservice guidelines](#_Webservice_guidelines) 52

J1) [Webservice implementation guidelines](#_J1)_Webservice_implementation) 52

J2) [Error handling](#_J2)_Error_Handling) 52

# Introduction

This manual describes the organization standards, guidelines and best practices to be followed for Java coding. These are required to be followed to ensure –

Uniform, standard and bug-free Java coding

Java code that meets high levels of maintainability, readability, performance and reliability.

It covers the most important ones as well as commonly used organization standards, guidelines & best practices. These standards also accommodate the organization learning from common or critical errors in various Java based projects. Many of these standards are enforced by static analysis tools (JTest, FindBugs, PMD, Check style) which are organization tool standards. In addition, tools help the developer to enforce many other standards, guidelines and best practices that are not covered in this document.

Exhaustive secure coding standards are covered separately in the manual: Standards & Guidelines – Secure Coding.

Any Java file developed needs to align to the standards described here. Separate coding standards exist for EJB/MDBs.

This is intended to be a must-read by all developers / tech leads before doing any Java coding for any projects / products in Virtusa. The standards are further complemented by a code review checklist which includes those standards & guidelines which require manual validation, i.e. those not validated by tools.

Last, this is not meant to be a tutorial on Java programming language. It is required for developers / tech leads to know Java as a pre-requisite.

# File Structure & Naming Conventions:

*While the standards & guidelines described in this chapter are default organization standards, certain projects may deviate based on customer requirements in terms of the description of the standards. All other chapters cover standards & guidelines towards bug-free coding and should not require any deviation / tailoring.*

## A1) All Java files to align to the standard file skeleton

Following should be included in any Java file.

1. Copyright indicating the IPR
2. File header containing module, author, description and revision history
3. Javadocs for the class explaining in brief the functionality of the class
4. Javadocs for each class-level (member) variable explaining in brief the purpose of this variable
5. Javadocs for each method explaining in brief the functionality of the method, parameters, return types and exceptions thrown from this method
6. Constructor for the class

Further,

 All indentation should be in terms of tabs. (The tab should be set to 2 spaces by default in the IDE)

 Opening and closes braces (for class, method, control structures, etc) should be on new lines as depicted.

The body of a class should be indented by a tab

 The body of a method should be indented by a tab

The body of a control structure (for-loop, if-else loop, etc) should be indented by a tab

The file skeleton is provided below as a reference. It is recommended to use the Eclipse bundled in Zing to create new Java files. These files comes with this default file template.

**// FileSkeleton.java**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

**COPYRIGHT NOTICE**

Copyright 2011 Virtusa Software Lab Limited. All rights reserved.

These materials are confidential and proprietary to Virtusa Software Lab Limited and no part of these materials should be reproduced, published, transmitted or distributed in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or stored in any information storage or retrieval system of any nature nor should the materials be disclosed to third parties or used in any other manner for which this is not authorized, without the prior express written authorization of Virtusa Software Lab Limited.

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

* Module Name :Sample Module

\*

* File Name :SampleClass.java

\*

* Description :Description of Class. This is a sample class to explain the skeleton of Java File.

\*

* Version Control Block

\*

* Date Version Author Description

\* --------- -------- --------------- ---------------------------------------------------------------------------------------------

* 01/10/2019 1.0 Dummy User Summarized detail of changes made …..

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ package **com.virtusa**.sampleapp.samplemodule;

import java.util.HashMap;

import **com.virtusa**.commons.logging.Log;

import **com.virtusa**.commons.logging.LogFactory;

/\*\*

* This is a sample class.....

\*

\*/

public class **SampleClass**

{

//Instance of logger

private static final Log **msLog** = LogFactory.getLog(SampleClass.class);

/\*\* Instance of SampleEventClass acting as the "business event handler class"

* + for SampleEvent entity / module. This class typically acts as the model
  + invocation layer and invokes appropriate services to handle the event

\*/

private **mSampleEventClass** impl;

/\*\*

* + Protected constructor to ensures that multiple instances are
  + not created by the user.

\*/

protected **SampleClass**()

{

}

**/\*\***

* + Method for HttpRequest based handling of event: SampleEvent\_Add.
  + This method is invoked by the framework when the Add event is
  + triggered from the SampleEventFrm form in the browser.

\*

* + **@param** pRequest The HttpServletRequest to process
  + **@param** pResponse The HttpServletResponse object to forward
  + **@param** pTransferObj HashMap instance that contains form data and core
  + attributes required for event processing

\*

* + **@return** HashMap instance that is used as the JSP bean for populating
  + the form values in the response screen

\*

* + **@throws** MLException re-throws business errors thrown by SampleEventServiceClass
  + **@throws** MLFatalException re-throws fatal errors thrown by SampleEventServiceClass
  + or on encountering fatal / non-recoverable errors in this method

\*

\*/

public final HashMap **addSampleEvent**(HttpServletRequest **pRequest**,

HttpServletResponse pResponse,

HashMap **pTransferObj**) throws MLException,MLFatalException

{

if(msLog.isDebugEnabled()) mLog.debug("Entering"); String currenycCode = ….

// do something

if(msLog.isDebugEnabled()) mLog.debug("return");

}

}

## A2) All packages to adhere to the standard naming conventions

Package names should always be lowercase with sub-names separated by dot.

 No use of underscores, dashes or any other symbol for any identifier of the package All packages to start with “com.virtusa.”. It is recommended to follow that with product

name or application name, followed by module name, followed by sub-module name (if applicable)

Example covered above.

## A3) All classes to adhere to the standard naming conventions

 First character of the class name should be in capital case. For a multi-word class name, first character of each word to be in capital case

Do not capitalize the entire class

 Give meaningful Class Names relevant to the business functionality provided by the class.

Optional (Recommended) Guidelines –

* Provide a prefix for classes re-usable by multiple applications
* Prefix interfaces with “I”
* Suffix implementations of interfaces with “Impl”

e.g. SampleEJBServiceImpl, ISampleService Example covered above.

## A4) All methods to adhere to the standard naming conventions

 Method names should be in camel case, i.e. first word should be in lower case. For a multi- word method name, the first alphabet of every other word should be in capital case.

Give meaningful method names relevant to the functionality provided by the method.

 Java Beans (Bean classes)

* All “getter” methods should be prefixed with “get”
* All “setter” methods should be prefixed with “set”
* All methods that return a boolean should be prefixed with “is” or “has” Example covered above.

## A5) All variables to adhere to the standard naming conventions

#### General Guidelines

Give meaningful variable names to indicate the purpose / intent of the variable No use of single or double alphabet variable names (e.g. i, j, k, e, etc)

#### Class level (member) variables – Nonstatic

Such variables should be prefixed with “m”

The first alphabet of every word in the variable should be in capital case

#### Class level (member) variables – static

Such variables should be prefixed with “ms”

The first alphabet of every word in the variable should be in capital case

#### Method parameter variables

Such variables should be prefixed with “p”

The first alphabet of every word in the variable should be in capital case

#### Method local variables

Such variables should not have any prefix

Method names should be in camel case, i.e. first word should be in lower case. For a multi- word method name, the first alphabet of every other word should be in capital case.

Example covered above.

# Core Programming

## B1) Perform validations on method parameters (passed by user or another application) before executing method logic

For methods that can be invoked with parameters corresponding to values entered by a user, or parameters passed by another application or component (i.e. public interfaces) perform the following validations on the method parameters -

Null check Mandatory Checks

Data type check (e.g. date, numeric, etc)

Business specific condition check (e.g. status code should be either “T0” or “T1”) If any of the validations fail, throw a checked exception to the caller.

#### Example With Error

public class SampleCmd implements TLCommand

{

public void execute(HashMap pInMap, HashMap pOutMap) throws MLException, MLFatalException

{

//no validation for parameters

PreSampleOp preSampleOp = PreSampleOp.getInstance ((String)pInMap.get(SampleConstants.APPLICATION\_NAME),entityName);

preSampleOp.preSampleModify (pInMap,pOutMap);

// .. processing of modify

}

}

#### Example With Error Rectified

public class SampleCmd implements TLCommand

{

public void execute(HashMap pInMap, HashMap pOutMap) throws MLException, MLFatalException

{

// Perform input validation checks / mandatory checks on the basis of

// Queue name present in the pInMap. Also perform any validation checks

//Perform mandatory checks/business validations.

if(! **SampleValidateInput**.checkMandatoryKeysForCorrect(pInMap))

{

throw new MLException (….);

}

PreSampleOp preSampleOp = PreSampleOp.getInstance ((String)pInMap.get(SampleConstants.APPLICATION\_NAME),entityName);

preSampleOp.preSampleModify (pInMap,pOutMap);

// .. processing of modify

}…

## B2) Handle all error, boundary, abnormal & special conditions corresponding to the functionality

This is one of the most ignored aspects in day-2-day programming. While this is a technology agnostic aspect, as all business logic is implemented in Java, it is very relevant in the current context. While, it is common to code to the intended functionality, following should not be ignored

All conditions / flows required by business functionality

All error scenarios and validations. If any of the error scenarios are encountered, throw a checked exception to the caller

Handling application delimiters while sending / receiving / processing method inputs Handling special characters while reading data at the server side or sending the data for display

#### Example With Error

In this example, Delimiters are not escaped while forming a concatenated string and vice-versa. A bean AutoBean here hold fields for an entity which needs to be stored as a single string in a CLOB field. Thus it has a toString() method which returns a single string in format: “key1=value1|key2=value2…|keyn=valuen”. Here “=“ is the NVDelimiter and “|” is the delimiter. This toString() method however does not ensure that Delimiter or NVDelimiters character occuring in “valuen” is handled

public class AutoBean

{

...

public final String toString(String pDelimiter, String pNVDelimiter)

{

...

StringBuffer lBuf = new StringBuffer(); Iterator lIterator = this.getKeys();

while (lIterator.hasNext())

{

String lKey = (String)lIterator.next(); lBuf.append(lKey); lBuf.append(pNVDelimiter); lBuf.append(get(lKey)); lBuf.append(pDelimiter);

}

...

}

}

...

return lAutoBeanStr;

#### Example With Error Rectified

public class AutoBean

{

...

public final String toString(String pDelimiter, String pNVDelimiter)

{

...

StringBuffer lBuf = new StringBuffer(); Iterator lIterator = this.getKeys();

while (lIterator.hasNext())

{

String lKey = (String)lIterator.next(); lBuf.append(lKey); lBuf.append(pNVDelimiter);

lBuf.append(escapeUnescape(get(lKey),pDelimiter,pNVDelimiter,true)); lBuf.append(pDelimiter);

}

...

}

}

...

return lAutoBeanStr;

## B3) Safeguard against the very popular NullPointerException

 Many methods return null instead of throwing exceptions especially when returning attributes of Java Beans. It is very important to validate the output of such methods before using its attributes / methods. If not done, it may result in null pointer checks and an unhandled exception.

 Throw checked exceptions wherever possible from methods when the value it is supposed to return is null or abnormal. This will prevent the consumers from performing validation checks or null checks explained above.

 When a method is returning a collection element or an array, always return an empty array/collection and not null.

#### Example -1 with Error

public class SampleClass {

...

public void method doAdd() {

....

String lPName = lSampleVO.getPName();

String lProdCode = lProductManufacture.getProductCode(lPName);

}

....

}

#### Example -1 with Error Rectified

public class SampleClass {

...

public void method doAdd() {

....

String lPName = lSampleVO.getPName(); if( lPName !=null){

String lProdCode = lProductManufacture.getProductCode(lPName);

}

....

}

}else {

}

throw new MLException (….);

#### Example -2 with Error

public class ProductManufacture{

….

public String getProductCode(String pPName)

{

if( <business condition>)

{

//business processing

}else{ //if business condition fails the return null return null;

}

return lProdCode;

}

} public class SampleClass {

...

public void method doAdd() {

....

String lPName = lSampleVO.getPName(); if( lPName !=null){

String lProdCode = lProductManufacture. getProductCode(lPName); if(lProdCode != null)

SampleClass.processProduct(lProdCode);

}

....

}

}else {

}

throw new MLException (…);

#### Example -2 with Error Rectified

public class ProductManufacture{

….

public String getProductCode(String pPName) throws MLException

{

if( <business condition>)

{

}else{

}

//business processing

throw new MLException (…..);

return lProdCode;

}

}

public class SampleClass {

...

public void method doAdd() { try{

....

String lPName = lSampleVO.getPName(); if(lPName != null){

String lProdCode = lProductManufacture.getProductCode(lPName); if(lProdCode !=null && lProdCode.compare(...)){

.....

}

}

}

....

}

}catch(MLFatalException mlEx){

....

}catch(MLException ex){

...

}catch(Exception ex)

...

}

## B4) Code with optimized memory management

Object creation is one of the most expensive operations in terms of memory utilization and performance impact. Thus it is very important to follow certain basic standards -

 Do not create memory leaks, i.e. dangling objects which are not garbage collected (A JProbe memory snapshot before the use-case and after the use-case is the best way to validate if there are any memory leaks

Create or initialize an object at the point it is required, i.e. lazy instantiation

For stateless processing by a class, maintain only single instance – either by implementing a singleton pattern or holding the reference of the class in the parent class and continuing the hierarchy till a single instance or entry-point class is hit

 Do not allocate memory for a variable in a loop, if it can be allocated once outside the loop and used for operations within the loop

Use constants for String literals Check logging mode before logging

#### Example With Error

mLog.debug("Set Count….");

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

Object[] cols = (Object[])results.get(i);

**String stage = new String(cols[0]);**

….

}

#### Example With Error Rectified

if(mLog.isDebugEnabled()) mLog.debug("Set Count….");

String stage =””;

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

Object[] cols = (Object[])results.get(i);

**stage = (String)cols[0];**

….

}

Nullify the Objects after use, in case of large heap allocations

When large heap allocations are done in scope of a long method, the memory will be freed only on completion of method. In such cases, it helps to nullify the references of the large heap allocations after use. This will ensure that any garbage collection cycles run during the execution of the method, will re-claim that memory immediately, helping maintain optimum performance.

#### Example With Error

public void doProductProcessing()

{

…..

ProductDataSet lProdDataSet = ProductManger.getProductDataSet(); ProductProcessor.processProducts(lProdDataSet);

…business processing happening

//**lProdDataSet** object is not nullified.Problem araises on long methods with heavy objects

…..

}

#### Example With Error Rectified

public void doProductProcessing()

{

…..

ProductDataSet lProdDataSet = ProductManger.getProductDataSet(); ProductProcessor.processProducts(lProdDataSet); lProdDataSet=null;

…business processing happening

…..

}

## B5) Compare objects for equality the right way

 Use equals method instead of == to check for equality between objects. This applies for Strings also. Equals method check for equality between the state of objects whereas == checks if two object references point to the same instance in the heap

 When comparing an object with a constant value, ensure that the constant value is on left- hand-side to prevent any probable NullPointerException for cases where the object reference may be null

 Override equals() method in transfer objects / Java Beans for precise equality comparison, in the intended manner, between two instances of such class

#### Example-1 With Error

public double calculatePostalCharges(String pCode, String pCountry){ if(pCountry.equals(“India”)){

…..

}

….

}

#### Example-1 With Error Rectified

public void calculateProdCharges(String pCode, pCountry){

if(“India”.equals(pCountry)) // OR (SampleCountry.INDIA.equals(pCountry))

{

…..

}

….

}

#### Example-2 With Error

public class SampleBean implements Serializable { private String mName;

public setName (String pName){ mName= pName;

}

public String getName (String pName){

…

}

//missing equals()

}

#### Example-2 With Error Rectified

public class SampleBean implements Serializable { private String mName;

public setName (String pName){ mName= pName;

}

public String getName (String pName){

…

}

public boolean equals(Object obj) { if(!(obj instanceof SampleBean)){

return false;

}

SampleBean temp= (SampleBean)obj; return this.mName.equals(temp.mName);

}

}

## B6) Use String Buffer concatenation instead of String addition

String buffer concatenation provides much better performance than creating new strings and adding them every time as string variables are immutable.

#### Example With Error

public String getDataString(HaspMap pInMap) throws MLException, MLFatalExceptio {

…..

String lDataString =new String();

lDataString = DataEncoder. Encode(pInMap(SConstants.PKeyName)) lDataString = lDataString + “=” + pInMap.get(“PCode”);

lDataString = lDataString + “&” ;

…..

….

}

#### Example With Error Rectified

public String getDataString(HaspMap pInMap) throws MLException, MLFatalExceptio {

…..

StringBuffer lDataStringBuffer =new StringBuffer(); lDataStringBuffer.append(pInMap(SConstants.PKeyName)) lDataString.append(SConstants.EQUALS);

lDataString.append( pInMap.get(“PCode”)); lDataString.append(SConstants.AND);

…..

….

}

## B7) Override toString() method for all transfer objects and JavaBeans

For any class whose instances hold state, it is required to override the toString() method and provide a meaningful String representation of the state held by that object. *This is a very useful aid in debugging*.

### Example with Error

Class Product{

private String mProdName; private int mProdId;

public void setProdName (String pProdName){ this.mProdName= pProdName;

}

public String getProdName (){ return this. mProdName;

}

//missing toString() method.

}

Class ProductProcessor{

….

public void doProductProcessing(..){ Product product=new Product ();

...

//it will print hashCode of object.

If(mLog.isDebuEnabled()) mLog.debug(“the value is” + product);

}

}

#### Example With Error Rectified

Class Product{

private String mProdName; private int mProdId;

public void setProdName (String pProdName){ this.mProdName= pProdName;

}

public String getProdName (){ return this. mProdName;

}

public String **toString**() {

StringBuilder result = new StringBuilder();

String NEW\_LINE = System.getProperty("line.separator");

result.append(this.getClass().getProdName () + " Object {" + NEW\_LINE);

…… result.append("}");

return result.toString();

}

}

Class ProductProcessor{

….

public void doProductProcessing(..){ Product product=new Product ();

...

//it will print Object’s value String representation.

If(mLog.isDebuEnabled()) mLog.debug(“the value is” + product);

}

}

## B8) Minimize the accessibility of class variables (member variables)

Class variables should never be kept public to prevent consumers from changing its value directly.

#### Example With Error

public class TransferAccount{ public String mProdCode;

…..

}

#### Example With Error Rectified

public class TransferAccount{ private String mProdCode;

// all private member variables would have setter/getter method accordingly on Case-to-case basis

….

}

## B9) Implement Comparable Interface to compare Objects.

Implement Comparable interface to compare objects and sort. It provides fast and efficient method for comparison between objects.

## B10) Avoid Shell scripts inside java code:

Avoid calling the Runtime.exec in order to spawn a process and execute the OS specific commands as this may not be portable because there is no gurantee how the native OS command wil behave on different platforms. Instead, encourage use of Java APIs.

## B11) Use HttpSession correctly

 Ensure use of HttpSession only where required. HttpSession should not be used where HttpRequest could have been used. (e.g. scope of JSP beans should be HttpRequest and not HttpSession)

 Common HttpSession dos & donts –

* Check HttpSession expiry before processing each use-case
* Invalidate HttpSession during application logout
* Do not create new HttpSession for use-cases other than login or the application launch use-case
* Keep HttpSession light. Do not put bulky objects like growable collections or collections of unknown size. To ensure high scalability, HttpSession footprint should be very small
* HttpSession should only be used for holding user specific data that will not change which is retrieved during login or application launch use-case, i.e. as a general rule do not put changing data in HttpSession. If it is required by the use-case, it should be removed as soon as the use-case is complete.

#### Example With Error

In this example, a collection of unknown size put in HttpSession. Here a list of records fetched from database need to be displayed in browser. For this the list is put in HttpSession so that JSP can generate formatted HTML code. HttpServletRequest can be used for this purpose instead.

List list = (List)eventResponse.get("DataList"); session.setAttribute("DataList", list);

#### Example With Error Rectified

List list = (List)eventResponse.get("DataList"); request.setAttribute("DataList", list);

## B12) Do not use application server specific APIs

To keep the application portable on multiple application servers, it is required to refrain from using any application server specific APIs (e.g. Websphere) for optimization or advanced functionality.

Also related is the requirement to not depend on any application server specific leniency in J2EE

implementation. Program to Java / J2EE APIs as required by the specifications.

## B13) Do not hard-code values

Do not hard-code values for attributes like file locations, schema name, deployment related attributes, etc. which are subject to changes on change of environment or during extensions of the product.

These should be externalized to a configuration file or database.

#### Example With Error

This example illustrates hard-coding of deployment attributes. Here the provider URL and Initial Context Factory is hard-coded for getting the InitialContext. Instead the values of these attributes should be read from a configuration file

String providerURL = "t3://localhost:7001";

String initialContextFactory = "weblogic.jndi.WLInitialContextFactory"; Properties prop = new Properties(); prop.put(Context.INITIAL\_CONTEXT\_FACTORY, initialContextFactory); prop.put(Context.PROVIDER\_URL, providerURL);

Context context = new InitialContext(jndiProps);

#### Example With Error Rectified

String providerURL = appConfig.getValue(TlineConstants.DEPLOYCONFIG\_PROVIDERURL);

String initialContextFactory = appConfig.getValue(TlineConstants.DEPLOYCONFIG\_INITIALCONTEXT); Properties prop = new Properties();

prop.put(Context.INITIAL\_CONTEXT\_FACTORY, initialContextFactory); prop.put(Context.PROVIDER\_URL, providerURL);

Context context = new InitialContext(jndiProps);

## B14) Do not modify method parameters that are meant to be “input-only” reference parameters

As objects are passed by reference during method invocations, one can un-knowingly modify a method parameter which was meant to be read-only for the given business requirement context.

## B15) Safeguard against common pitfalls of using control structures

Control structures are required as part of any control or business logic implementation. However, it is important to be aware of certain common pitfalls related to usage of the same -

Nested “for” loops where un-intended loop counter variable used Switch case not having a default case

Using assignment operator by mistake where equality operator was intended in an “if” condition Changing length of an array / collection while iterating, changing a “for” loop counter within the loop

Complicated error-prone “if” conditions. Excessive error-prone nesting of for / if loops

#### Example With Error

This is an example of nested for loops where un-intended loop counter variable is used. It is a common mistake to accidentally type the wrong loop counter variable resulting in an unexpected outcome. This would especially happen with confusing short loop counter variables like i, j, k etc

for (int i = 1; i <= 10; i++)

{

for (int j = 1; i <= 5; j++)

{

...

}

...

}

#### Example With Error Rectified

for (int i = 1; i <= 10; i++)

{

for (int j = 1; j <= 5; j++)

{

...

}

...

}

#### Example With Error

This is an illustration of a Typo of using assignment operator by mistake where equality operator was intended. This typo results in the block of code always getting executed unconditionally. In addition, it will have disastrous effect of changing the value of status variable to null, resulting in some errors in logic elsewhere

if(status = null)

{

...

}

#### Example With Error Rectified

if(status == null)

{

...

}

# Exception Handling & Logging

Incorrect and inappropriate exception handling & logging during development is the main reason for – Reduced monitoring abilities of the application

Reduced troubleshooting, self-detect and self-heal ability in SIT / UAT / Production Compulsion to keep logging level as debug even in SIT / UAT / Production Generation of voluminous log files

This is one of the key aspect to ease production support.

## C1) Handle checked and un-checked exceptions correctly

#### Un-checked exceptions (e.g. NullPointerExceptions):

 Always catch them using a “catch (Exception ex)” block ***only at thread entry points*** – e.g. controller classes, EJB, MDB.

Log in fatal mode. Print the stack trace to log

In case of catching them in controller classes, convert them into an application level checked exception class of fatal category and show an appropriate common message on the screen, indicating un-anticipated system error and requesting the user to try again

 *In case of catching them in EJB/MDB, roll back the transaction context*

**Checked exceptions:** These represent error conditions which can be detected by the application. The logic for detecting certain business exceptions or invalid conditions like infrastructure issues is to be explicitly programmed.

 On detecting an invalid scenario / condition (e.g. invalid user input, invalid method parameters, invalid return values from methods, database problems, network outages, absent files, etc),

* Log the details of the scenario with error or fatal mode depending on the severity of the condition
* Print the stack trace to log - *Note that*, in certain cases, the Java APIs used, e.g. JDBC API throw SQLException which is a checked exception to indicate an invalid condition. This needs to be caught and the stacktrace of this exception should be printed to log file
* Create a application checked exception class instance (with an appropriate error code) and throw

 The method that detect such conditions should have the application checked exception class in their “throws” clause

 Any other parent calling methods in the call stack, should not catch the application checked exception. They should however have the application checked exception in their throws clause. Such exceptions should be finally caught only at thread entry points – e.g. controller classes, EJB, MDB – ***using explicit catch blocks for each such application checked exceptions***

 In case of catching them in controller classes, show an appropriate error message on the screen, indicating one of the known system errors and hint the user on how to rectify the same

 In case of catching them in EJB/MDB, roll back the transaction context and throw the application checked exception from the EJB / MDB

##### Get the error message for the application exceptions from the property files (Resource bundles during display)

## C2) Ensure correct “throws” clause for the methods that detect / propagate exceptions

 Do not have the generic “Exception” class in the “throws” clause

 *Never propagate checked exceptions thrown by Java API from application method using “throws” clause*. Such exceptions should be necessarily caught at source and converted to application checked exception instance as explained above.

#### Example with Error

public class ProductDAO{

…

public void doProcess(….) throws Exception

{

try{

//get connection

// do DB Insert

} catch(SQLException sqlEx) { throw sqlEx;

} catch(Exception ex) {

Log.fatal(“Exception while processing….); Throw ex;

}

//do something

}

}

#### Example With Error Rectified

public class ProductDAO{

…

public void doProcess(….) throws SampleAppFatalException, SampleAppException

{

try{

//get connection

….

If(isRecordExists(….)){

if(mLog.isErrorEnabled()) mLog.error(“Duplicate Record”); throw new SampleAppException(“….”);

}

// do DB Insert

} catch(SQLException sqlEx) {

mLog.fatal(“Exception during Product DAO Processing.”, sqlEx); throw new SampleAppFatalException(“SampleAppExCode.121”);

} finally {

//do resource clean-up

}

//do something

}

}

## C3) Catch all possible checked exceptions that can be thrown from the block of code

In case a block of code throws multiple checked exceptions,

 Catch all of those. Multiple catch blocks can be used in case where all exceptions thrown inside the same try-block are of the same type or subclasses of that type.

The catch clauses must be listed from most specific to most general.

Do not provide only one generic catch block to catch all exceptions, as in – “catch (Exception ex)”

#### Example with Error

public class ProductDAO{

…

public void doProcess(….) throws SampleAppFatalException, SampleAppException, Exception

{

try{

//get connection

// do DB Insert

} catch(SQLException sqlEx) {

mLog.fatal(“Exception during Product DAO Processing.”, sqlEx); throw new SampleAppFatalException(“SampleAppExCode.121”);

} catch(Exception ex){

//do something..

}

//missing exception specific to Batch Processing. finally {

//do resource clean-up

}

//do something

}

}

#### Example With Error Rectified

public class ProductDAO{

…

public void doProcess(….) throws SampleAppFatalException, SampleAppException

{

try{

//get connection

// do DB Insert

//do something

} catch (BatchUpdateException ex)

{

mLog.fatal("Exception while adding multiple records", ex); throw new SampleAppException(“SampleAppExCode.131”);

} catch(SQLException sqlEx)

{

mLog.fatal(“Exception during Product DAO Processing.”, sqlEx); throw new SampleAppFatalException(“SampleAppExCode.121”);

} finally {

//do resource clean-up

}

//do something

}

}

## C4) Never throw an exception from finally block

Finally block is typically required when one wants to close resources opened in the try block, or for any clean-up activity. However, finally block should never throw an exception. In case, any code in the finally block can throw an exception, catch and consume it, logging appropriate details and continue.

## C5) Never keep a catch block empty

A catch block should never be empty. It should have –

Logging of message (describing the error condition) and exception stack trace to log file Handling of exception object as appropriate, like conversion to application checked exception instance. Do not consume an exception without any action

#### Example With Error

public class SampleClass {

public void doFileProcessing (….) throws Exception{ try {

//do something

} catch (java.io.IOException e) {

//empty

} finally {

//empty

}

}

}

#### Example With Error Rectified

public class SampleClass {

public void doFileProcessing (….) throws SampleException, SampleFatalException{ try {

//do something

//do something

} catch (java.io.IOException ioEx) {

mLog.fatal("Exception while processing File", ioEx); throw new SampleException(“SampleAppExCode.122”);

} finally {

//do clean up of File Resource.

}

}

}

## C6) Handle exception stack trace correctly

PrintStacktrace provides the complete stack of the exceptions.

 No use of printStackTrace() as in ex.printStackTrace(). This makes the stack trace details go to the command console of JVM or app server and not to the log file. This impacts troubleshooting and deteriorates performance.

 Always print the stack trace of the exception to the application log file, as in (here “ex” is the exception object) –

*mLog.fatal(“<message describing the operation that failed>”, ex)*

 Do not use “+ ex” while printing exception details to the log. This only causes ex.getMessage() to be printed to the log and not the stackTrace() of the exception object. Again this impacts troubleshooting as the statement which generated the exception cannot be identified

#### Example With Error

public class ProductDAO{

…

public void doProcess(….) throws SampleAppFatalException, SampleAppException

{

try{

//get connection

// do DB Insert

//do something

} catch(ArrayOutOfBoundExceotion ex) {

mLog.fatal("Exception while travsing list of Products ID "+ ex);

} catch (BatchUpdateException ex)

{

mLog.fatal("Exception while adding multiple records"+ ex.getMessage);

} catch(SQLException sqlEx)

{

sqlEx.printStackTrack();

mLog.fatal(“Exception during Product DAO Processing.”);

throw new SampleAppFatalException(“SampleAppExCode.121”);

} finally {

//do resource clean-up

}

//do something

}

}

#### Example With Error Rectified

public class ProductDAO{

…

public void doProcess(….) throws SampleAppFatalException, SampleAppException

{

try{

//get connection

// do DB Insert

//do something

} catch(ArrayOutOfBoundExceotion ex) {

mLog.fatal("Exception while travsing list of Products ID",ex); throw new SampleAppException(“SampleAppExCode.132”);

} catch (BatchUpdateException batchEx)

{

mLog.fatal("Exception while adding multiple records", batchEx); throw new SampleAppException(“SampleAppExCode.131”);

} catch(SQLException sqlEx)

{

mLog.fatal(“Exception during Product DAO Processing.”, sqlEx); throw new SampleAppFatalException(“SampleAppExCode.121”);

} finally {

//do resource clean-up

}

//do something

}

}

## C7) No use of System.out.println / System.err

Printing messages for debugging and troubleshooting of error messages is required. However, such messages **should not strictly be printed on the command line console** of the application server or JVM. This impacts troubleshooting and deteriorates performance. Logger should be used to log any application messages –

Debugging messages Informational messages Business Error conditions Fatal error conditions

A standard logger component like Log4j should be used.

#### Example With Error

public class ProductDAO{

public void doProcess(….) throws SampleAppFatalException, SampleAppException

{

try{

…..

System.out.println(“pInMap”+pInMap);

// Any processing that is required to be done before deleting an existing record preDeleteMultiple(pInMap, pOutMap);

// get the SQL String to be used for deleting a record String lSqlString = getDeleteRecordSqlString(); System.out.println("Delete Query :"+ lSqlString);

…..

//do something

} catch(SQLException sqlEx)

{

System.out.println(“Exception during Product DAO Processing.”+sqlEx); throw new SampleAppFatalException(“SampleAppExCode.121”);

} finally {

//do resource clean-up

}

}

….

}

#### Example With Error Rectified

…

import com.virtusa.commons.logging.Log;

import com.virtusa.commons.logging.LogFactory;

public class ProductDAO

{

private static Log mLog = LogFactory.getLog(ProductDAO.class);

public void doProcess(….) throws SampleAppFatalException, SampleAppException

{

try{

…..

if(mLog.isDebugEnabled()) mLog.debug(“pInMap”+pInMap);

// Any processing that is required to be done before deleting an existing record preDeleteMultiple(pInMap, pOutMap);

// get the SQL String to be used for deleting a record String lSqlString = getDeleteRecordSqlString();

if(mLog.isDebugEnabled()) mLog.debug ("Delete Query :"+ lSqlString);

…..

//do something

} catch(SQLException sqlEx)

{

mLog.fatal(“Exception during Product DAO Processing.”,sqlEx); throw new SampleAppFatalException(“SampleAppExCode.121”);

} finally {

//do resource clean-up

}

}

….

}

## C8) Program to logger components using a facade

Always use a façade to wrap the concrete logger components. Default organization standards for logging are –

virtusa wrapper component as a logging façade (wrapper)  Log4j as the actual logging plug-in configured in the façade

The code should only be programmed to the logging façade and not directly to the logger, i.e.

log4j. This facilitates ease of swapping concrete logger components

## C9) Use appropriate logging modes to print various messages to the log file

The default logger (Log4j) supports various logging modes to log messages, depending on the condition. Use appropriate logging mode while printing messages to the log file –

 Use **debug** mode of logging (*mLog.debug(“<message>”)* ) to log messages for debugging purposes. The debug logs should be meaningful and at well-defined points of execution in code such that it helps the developer during development. It is a common mistake to log value of every variable in debug mode, or a debug log after every statement. This should strictly be

avoided. Not only does it impact performance, it impacts code readability and maintainability. In general, avoid too few or too much of debugging information logged in debug mode.

 Use **info** mode of logging (*mLog.info(“<message>”)*) to log important informational milestones while processing a transaction or general informational messages related to the application

 Use **error** mode of logging (*mLog.error(“<message>”)* ) to log business error / validation conditions, which are anticipated

 Use **fatal** mode of logging (*mLog.fatal(“<message>”)*) to log all unanticipated or fatal error conditions. Fatal mode should result in no logs during normal execution of the application where no errors have resulted.

##### It is a common mistake to log fatal conditions using lesser mode (like debug / info / error) . This should strictly be avoided as logger’s logging level would be fatal in production, as a result of which the messages would not be logged, thereby impacting troubleshooting.

Standards around configuration of Log4j logging levels –

 ***Debug level should be configured only in development environment*** when developer is testing. As the debug mode results in generating a lot of messages and file I/O, it should never be configured during shared environment concurrent user testing, like SIT / UAT / Production

 **Logging levels of info and error are also useful for single developer testing**, where-in after initial code stability, the logging needs to be restricted. Again, it is not recommended for these logging levels to be configured during shared environment concurrent user testing, like SIT / UAT / Production

##### The logging level in SIT / UAT / Production should strictly be fatal. It should be possible to troubleshoot with log level as fatal, with the all the required information printed in fatal mode when the error is encountered.

 A developer needs to test the unit with all logging levels to ensure that logs are not unnecessarily generated or are not missed due to incorrect usage of logging modes

## C10) Prevent un-necessary heap overload while logging

Consider the following logging statement –

*mLog.debug(“Processing the account. ” + “Account details are: “ + ..);*

Here, various String objects are created in heap only to realize that the logging level is set to fatal and does not need to be logged to file. This can be prevented by –

*if mLog.isDebugEnabled() mLog.debug(“Processing the account. ” + “Account details are: “ + ..);*

Thus, for all logging in debug, info and error mode, use the appropriate isXXXEnabled() method – if (*mLog.isDebugEnabled()*) mLog.debug(“…”);

if (*mLog.isInfoEnabled()*) mLog.debug(“…”);

if (*mLog.isErrorEnabled()*) mLog.debug(“…”);

Further, the value of isDebugEnabled() can be stored once as a private static boolean variable in the class.

*private static Boolean isDebug = mLog.isDebugEnabled();*

This variable can then be used for various logging statements, instead of calling the method isDebugEnabled() again and again in the class

*if (isDebug) mLog.debug(“Processing the account. “ + “Account details are:” + …);*

# JDBC

## D1) Use only a DataSource for retrieving a connection in an app server or web server environment

In an app server or web server environment –

Get the handle to Data Source configured in the server using JNDI look-up

While performing lookup, externalise InitialContextFactory, ProviderURL, etc in properties file. Do not hardcode such values in java file.

 Get the connection from the data source. Do not develop custom logic to implement connection pool algorithm.

 No use of DriverManager.getConnection(). For non app server environment, open source database connection pooling implementations like Apache Common’s DBCP should be considered. (<http://commons.apache.org/dbcp)>

#### Example With Error

private Connection getConnection()

{

try {

Connection conn = null;

String driverName = "oracle.jdbc.driver.OracleDriver"; Class.forName(driverName);

// Create a connection to the database String serverName = "127.0.0.1"; String portNumber = "1521";

String sid = "mydatabase";

String url = "jdbc:oracle:thin:@" + serverName + ":" + portNumber + ":" + sid; String username = "username";

String password = "password";

connection = DriverManager.getConnection(url, username, password);

} catch (SQLException e) {

…… e.printStackTrace(); return null;

}

}

#### Example With Error Rectified

public class ConnectionManager

{

public static Connection getConnection

{

/\*\* Read datasource name from config file \*/ String dataSourceName = ...;

Connection conn = null;

...

Context initialContext = new InitialContext();

DataSource datasource = (DataSource)initialContext.lookup(dataSourceName); conn = datasource.getConnection();

...

return conn;

}

}

#### Example With Error

String providerURL = "t3://localhost:7001";

String initialContextFactory = "weblogic.jndi.WLInitialContextFactory"; Properties prop = new Properties(); prop.put(Context.INITIAL\_CONTEXT\_FACTORY, initialContextFactory); prop.put(Context.PROVIDER\_URL, providerURL);

Context context = new InitialContext(jndiProps);

...

#### Example With Error Rectified

Connection related attributes can be obtained from configuration file.

String providerURL = appConfig.getValue(SampelAppConstants.DEPLOYCONFIG\_PROVIDERURL); String initialContextFactory = appConfig.getValue(SampelAppConstants.DEPLOYCONFIG\_INITIALCONTEXT);

Properties prop = new Properties(); prop.put(Context.INITIAL\_CONTEXT\_FACTORY, initialContextFactory); prop.put(Context.PROVIDER\_URL, providerURL);

Context context = new InitialContext(jndiProps);

….

## D2) Use Prepared statements instead of Statements

Use PreparedStatements instead of statements. This ensures -

Better performance. (PreparedStatement precompiles SQL in the cache) Safeguarding against SQL Injection vulnerability

#### Example With Error

public class CurrencyDAO

{

...

public String viewCurrency(CurrencyVO pCurrency)

{

String strSQL = "select currencyName from Currency where currencyCode = "; strSQL += pCurrency.getCurrencyCode();

try

{

Connection conn = ConnectionWrapper.getConnection(); Statement stmt = conn.createStatement(strSQL); ResultSet rst = stmt.executeQuery();

....

}

catch(SQLException sqlException)

{

....

}

}

}

#### Example With Error Rectified

public class CurrencyDAO

{

...

public String viewCurrency(CurrencyVO pCurrency)

{

String strSQL = "select currencyName from Currency where currencyCode = ?"; Connection conn = null;

PreparedStatement pstmt = null; ResultSet rst = null;

try

{

conn = ConnectionWrapper.getConnection(); pstmt = conn.prepareStatement(strSQL); pstmt.setString(1, pCurrency.getCurrencyCode()); rst = pstmt.executeQuery();

....

} catch(SQLException sqlException)

{

} finally

{

}

}

}

....

if(rst != null)

rst.close(); if(pstmt != null)

pstmt.close(); if(conn != null)

ConnectionWrapper.closeConnection(conn);

## D3) Externalize all SQL in a properties / XML file

Instead of defining the sql statements inside the code, they should be placed in properties file/xml file. The code should read the statements from those. This is required for maintainability or SQL tuning.

## D4) Close all database resources (connections, statements, result sets) in finally block

 During JDBC operations, one ends up using / opening following database resources – Connections

PreparedStatements ResultSets

 These resources should be closed unconditionally in the finally block. Closing the resources in the finally block ensures that these resources get closed even in case of SQL Exceptions. Not closing connections results in the very popular “connection leaks” and thereby server outage The order of closing should be – Result Set first, PreparedStatement next and Connection last Any and all such resources should be closed. No assumption to be made on closure of resources by underlying frameworks / drivers / app servers

 The connection should be closed in the same scope in which it is opened, i.e. the method that creates a connection should close it. A method should not close a connection not created by it.

#### Example With Error

String strSQL = "select currencyName from Currency where currencyCode = ?"; Connection conn = null;

PreparedStatement pstmt = null; ResultSet rst = null;

try

{

conn = ConnectionWrapper.getConnection(); pstmt = conn.prepareStatement(strSQL); pstmt.setString(1, pCurrency.getCurrencyCode()); rst = pstmt.executeQuery();

....

rst.close(); pstmt.close();

}

catch(SQLException sqlException)

{

....

}

#### Example With Error Rectified

String strSQL = "select currencyName from Currency where currencyCode = ?"; Connection conn = null;

PreparedStatement pstmt = null; ResultSet rst = null;

try

{

conn = ConnectionWrapper.getConnection(); pstmt = conn.prepareStatement(strSQL); pstmt.setString(1, pCurrency.getCurrencyCode()); rst = pstmt.executeQuery();

....

}

catch(SQLException sqlException)

{

....

}

finally

{

if(rst != null)

rst.close(); if(pstmt != null)

pstmt.close(); if(conn != null))

conn.close();

}

## D5) Do not change default database connection properties in an app server environment

 In app server environment, do not change any default properteis of a connection fetched through data source.

 In an app server environment, the transaction atomicity should be managed by EJB / MDB / User Transaction Context.

 Any programitical changes in connection’s attributes will interfere with app server’s ability to manage transaction. This may lead to un-reliable atomicity handling and also performance degrade.

## D6) Use PreparedStatement’s BatchUpdate feature while performing JDBC updates for multiple records

Sending multiple queries to the database at a time using batch update feature of statement objects reduces the number of JDBC calls and improves performance.

#### Example With Error

PreparedStatement pstmt = connection.prepareStatement(“INSERT INTO customer values(?, ?, ?)”); for(CustomerVO customerVO:arrayList)

{

Pstmt.setString(1, customerVO.getFirstName()); Pstmt.setString(2, customerVO.getMiddleName()); Pstmt.setString(3, customerVO.getFirstName()); Pstmt.execute();

}

#### Example With Error Rectified

PreparedStatement pstmt = connection.prepareStatement(“INSERT INTO customer values(?, ?, ?)”); for(CustomerVO customerVO:arrayList)

{

Pstmt.setString(1, customerVO.getFirstName()); Pstmt.setString(2, customerVO.getMiddleName()); Pstmt.setString(3, customerVO.getFirstName()); Pstmt.addBatch();

}

Pstmt.executeBatch();

## D7) Do not perform database operations in a loop

 Do not perform costly database operations in a loop, e.g.

* Retrieving connections, Database operations, etc. Some methods called in a loop may internally be doing database operations. Multiple alternatives exist to resolve such behaviour based on context
  + Local cache - If the same query is fired move it outside the loop
  + Global cache - If the query is for static data, fire it at server start-up and cache the results
  + Incremental Local cache – Fire the query in the loop for a given criteria based on need. Once fired, cache the results locally so that it does not need to be fired for the next iteration with the same criteria
  + Usage of IN query – Consider firing one query outside the loop using IN operator
  + In case a database operation is still required in a loop, create a connection once outside the loop, rather than repeatedly creating and closing it in the loop

#### Example With Error

In this example, Database query fired in a loop unconditionally. Here a list of value objects is processed. Each value object has a currency code. For each currency code there is an allowable number of decimal points (fetched from database) which is used for processing the value object. Incremental Local cache is demonstrated for this example. Global cache may also have been used for this particular scenario.

private void formatNumericFields(List pValueObjs) throws MLException

{

...

Object obj = null;

for(int count = 0; count < pValueObjs.size(); count++)

{

obj = (ValueObj)pValueObjs.get(count); String cod\_Ccy = obj.getCurrencyCode();

//The call to method below fires a query to get the

// decimals allowed for a currency

int decimalAllowed = Utility.getDecimalAllowed(cod\_Ccy);

.... // main processing

}

...

}

#### Example With Error Rectified

private void formatNumericFields(List pValueObjs) throws MLException

{

...

Object obj = null;

Map currencyCache = new HashMap();

for(int count = 0; count < pValueObjs.size(); count++)

{

obj = (ValueObj)pValueObjs.get(count); String cod\_Ccy = obj.getCurrencyCode();

int decimalAllowed = 0; if(currencyCache.containsKey(cod\_Ccy))

{

}

else

{

decimalAllowed = ((Integer)currencyCache.get(cod\_Ccy)).intValue()

//The call to method below fires a query to get the

// decimals allowed for a currency

decimalAllowed = Utility.getDecimalAllowed(cod\_Ccy);

//cache the decimal allowed value for the currency currencyCache.put(cod\_Ccy, new Integer(decimalAllowed));

}

.... // main processing

}

...

}

# Input / Output:

## E1) Close all I/O resources in the finally block

 During I/O / socket / URL operations, one ends up using / opening many resources, such as – InputStream, OutputStream, FileInputStream, FileOutputStream, etc

 FileReader, FileWriter, BufferedReader, BufferedWriter SocketInputStream, SocketOutputStream, URLConnection, etc

 These resources should be closed unconditionally in the finally block. Closing the resources in the finally block ensures that these resources get closed even in case of Exceptions. Not closing these results in exhaustion of server resources and consequent outage

 Any and all such resources should be closed. No assumption to be made on closure of resources by underlying frameworks / drivers / app servers

 The resources should be closed in the same scope in which it is opened, i.e. the method that creates a resource should close it. A method should not close a resource not created by it.

#### Example With Error

public void readFile()

{

BufferedReader in = new BufferedReader (new FileReader("infilename")); String str;

while ((str = in.readLine()) != null) {

….

}

in.close();

}

#### Example With Error Rectified

public void readFile() {

BufferedReader in = nulll;

try {

in = new BufferedReader(new FileReader("infilename")); String str;

while ((str = inreadLine()) != null) {

….

}

} catch(Exception exp){

// code to handle exception

} finally() {

if(in != null)

in.close();

}

}

## E2) Optimize I/O performance using appropriate API

The major culprit that affects the IO performance is InputStream.read or Reader.read. Reading character by character will affect the performance,hence **BufferedReader should be** used that reads data in buffer.

#### Example with Error

private static void handleCharacters() throws IOException { InputStream in = new FileInputStream(file);

Reader reader = new InputStreamReader(in);

int r;

while ((r = reader.read()) != -1) { char ch = (char) r;

}

// other business logic and closing of resources

}

#### Example With Error Rectified

private static void handleCharacters() throws IOException {

BufferedReader br = new BufferedReader(new FileReader(filename)); while( (strLine = br.readLine())!= null) {

}

// other business logic and closing of resources

}

# Data Structures

## F1) Use Generics with Collection

The Collection interface should be generified. The String collection can contain only String instances. This will allow the compiler to know before hand what kind of data is found in the collection and thus many of runtime problems can be avoided.

#### Example With Error

private static List extractElements(List bag,)

{

List result = new ArrayList(); for(Object e : bag)

{

result.add((String) e);

}

return result;

}

#### Example With Error Rectified

private static List<String> extractElements(List<String> bag,) { List<String> result = new ArrayList<String>(); for(Object e : bag)

{

result.add((String) e);

}

return result;

}

If you try to add other object, or cast the elements in the collection to any other type than String, the compiler will complain.

## F2) Use Non-thread safe collection classes rather than thread-safe collection classes where possible

If the collection is not shared or does not demand multi-thread aware updation of values, it is a good practice to use non-thread safe classes like Arraylist ,HashMap. There is a slight performance overhead for using thread safe collections classes like Vector, HashTable, etc.

Use ArrayList instead of Vector Use HashMap instead of Hashtable

*For synchronized approach, utility methods like synchronizedMap, synchronizedList, synchronizedSet of java.util.Collections class should be used. Since JDK 1.5, java.util.ConcurrentHashMap is introduced that is optimized for concurrent access.*

## F3) Use appropriate collection classes based on requirement

Based on the requirement of project proper collection classes should be chosen.

 Read & delete operations should be performed with appropriate collection classes. For example ArrayList performs read operation with better performance while LinkedList performs better with delete operation.

 Use appropriate collection classes when unique values are expected to be stored. In case of handling unique values Set is preferable over List.

# Multi-user aware programming, Multi-threading & Concurrency

## G1) Do not maintain state (member variables) for classes for which shared instances are maintained (i.e. intended to be stateless)

Most of the times the processing is stateless. Thus many class instances as designed by the application or those provided by frameworks are shared, i.e. one / limited instances are maintained irrespective of the number of users using the application. Typically only transfer objects / value objects need to be necessarily created as a new instance with every transaction / use-case and thus are never shared. Any processing class can have a shared instance.

In such cases where class instances are shared, no state (member variables) should be maintained in the class, as in a multi-user (multi-thread) environment, relying on values of such variables for processing logic will result in ambiguity or inconsistency. Violation of this standard results in serious defects which cannot be reproduced consistently. Examples of such classes are -

Application classes which hold cache Application singletons

Struts Action Classes Stateless session beans Message Driven Beans Spring Beans

Business Logic classes DAOs

The cases where state may be required to be maintained are application classes that hold cache or singletons. In cases where state is required, the next standard described should be enforced.

#### Example With Error

This example illustrates Class for which single instance is maintained contains state that is modified. UserLoginAction is a Struts Action object which unnecessarily uses a member variable during its processing. This will result in random anomalies when tested with multiple users that cannot be reproduced consistently

public class UserLoginAction extends Action

{

String userId; String password;

public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response) throws Exception

{

userId = request.getParameter("username"); password = request.getParameter("password");

// .. handle login event;

return mapping.findForward("success");

}

}

#### Example With Error Rectified

public class UserLoginAction extends Action

{

public ActionForward execute(ActionMapping mapping, ActionForm form,

HttpServletRequest request, HttpServletResponse response) throws Exception

{

String userId = request.getParameter("username"); String password = request.getParameter("password");

// .. handle login event;

return mapping.findForward("success");

}

}

## G2) Updates to the cache or any shared state should always be synchronized

Most of the times the processing is stateless. The cases where state may be required to be maintained are application classes that hold cache or singletons. In such case, any updates to the state should be in a synchronized method / block.

Another example of shared state may be static variables which are non-final. Any updates to such state should also be synchronized.

#### Example With Error

This example illustrates updates to the static cache maintained by a class not synchronized. Here, the class DataCache has a static cache which can be accessed / modified concurrently in a multi-user / multi-threaded environment. Both the accessor and mutator methods should be synchronized. Alternatively, HashMap can also be changed to Hashtable which consists of synchronized methods for put and get.

public class DataCache

{

private static Map cache = new HashMap();

public static void add(String key, Object value)

{

cache.put(key, value);

}

public static Object get(String key)

{

cache.get(key);

}

}

#### Example With Error Rectified

public class DataCache

{

private static Map cache = new HashMap();

public static synchronized void add(String key, Object value)

{

cache.put(key, value);

}

public static synchronized Object get(String key)

{

cache.get(key);

}

}

## G3) Ensure minimum performance overhead when using “synchronized”

To get best performance with concurrent users –

 Use synchronized blocks / methods only where required and the requirements demand serialization of execution by concurrent threads

 Usage of synchronized method or synchronized block depends on locking time span that would result when a thread of execution executes it. When a method is synchronized, the lock is on the instance on which the method is getting executed. If the method is performing multiple activities and only one of those activities need serialized execution in a multi-user scenario, it is recommended to use synchronized blocks rather than synchronizing the entire method (which would increase the locking time span).

#### Example With Error

public synchronised void updateFileStatus() {

// do something

// actual code where exclusive lock is required FileWriter fstream = new FileWriter("fileStatus.txt"); BufferedWriter out = new BufferedWriter(fstream);

// do something

}

#### Example With Error Rectified

public void method() {

// do something

// actual code where exclusive lock is required synchronised(this) {

FileWriter fstream = new FileWriter("out.txt"); BufferedWriter out = new BufferedWriter(fstream);

}

// do something

}

## G4) Do not use Java Threads in app server environment

Do not use core Java threads in app server environment. Achieve the functionality using JMS / MDB and EJB Timer.

Java multithreading should be used only in non-app server based Java applications

## G5) Handle concurrent updates / deletes to a database record

In a multi-user environment, multiple users can update / delete the same database record (entity / transaction record).

Consider a scenario where one user opens a record for modification. At the same time another user also opens the record for update. Both may save the form in approximately the same time frame which may result in updates of the first one getting overwritten without knowledge of it. In such cases, one may maintain a LastModified column whose value is also fetched along with record details. While updating the record, this value is also used along with the record primary key to get handle to the record. Thus after the first update, the second user gets an error message and is notified that the record is already updated by another user. This implies, the second user can open the new record again for modification.

#### Example With Error

This example illustrates updates to a table by multiple concurrent logged-in users not handled

*Update Customer set status = ? where custid = ?*

#### Example With Error Rectified

*Update Customer set status = ? where custid = ?* ***and LastModified = ?***

## G6) Adhere to multi-threading best practices when using Java multithreading (in non-app server based Java apps)

 Do not use Thread.sleep(), unless it is absolutely required. Sleep method blocks the current thread for atleast the number of time slices which is not a good practice. The functionality can be achieved using TimerTask.

#### Example With Error

private void checkStatus(int processId) {

while(true) {

// code to check status from database

// sleet for certain interval Thread.sleep(interval);

}

}

#### Example With Error Rectified

public class Reminder { Timer timer;

public Reminder(int seconds) { timer = new Timer();

timer.schedule(new RemindTask(), seconds\*1000);

}

class RemindTask extends TimerTask { public void run() {

….

timer.cancel(); //Terminate the timer thread

}

}

}

 The ExecutorService interface is among the most commonly used items in the java.util.concurrent package. When finished using an ExecutorService, shut it down .

#### Example With Error

class NetworkService implements Runnable

{

private final ServerSocket serverSocket; private final ExecutorService pool;

public NetworkService(int port, int poolSize) throws IOException

{

..

pool = Executors.newFixedThreadPool(poolSize);

}

public void run() { // run the service try {

for (;;)

{

}

pool.execute(new Handler(serverSocket.accept()));

} catch (IOException ex) {

pool.shutdown();

}

}

} // service was not shutdown

#### Example With Error Rectified

An unused ExecutorService should be shut down to allow reclamation of its resources.

public void shutdownAndAwaitTermination(ExecutorService pool) { pool.shutdown(); // Disable new tasks from being submitted try {

// Wait a while for existing tasks to terminate pool.shutdownNow(); // Cancel currently executing tasks

// Wait a while for tasks to respond to being cancelled if (!pool.awaitTermination(60, TimeUnit.SECONDS))

{

System.err.println("Pool did not terminate");

}

} catch (InterruptedException ie)

{

// (Re-)Cancel if current thread also interrupted pool.shutdownNow();

// Preserve interrupt status Thread.currentThread().interrupt();

}

}

 Nullify the threads after use. Check for availability of thread instance before instantiating a new instance. After the thread usage, nullify the thread so that the thread does not hang and consume the resource. Its always a good practice to set null to the thread instance.

#### Example With Error

public class UpdateApplet extends java.applet.Applet implements Runnable { private Thread updateThread;

int updateInterval=1000;

public void run(){

whiletrue){ try{

....

}

}

public void start(){

updateThread=new Thread(this); updateThread.start();

}

....

}//thread is not stoppedf

#### Example With Error Rectified

public class UpdateApplet extends java.applet.Applet implements Runnable{

….

public void start(){

if(updateThread==null){

updateThread=new Thread(this); updateThread.start();

}

}

public void stop(){

if(updateThread!=null){

UpdateThread.stop(); updateThread=null;

}

}

}

# Implementation of design practices

## H1) Implement intended design level flexibility

 Ensure no compromise of polymorphic behavior

* Program to an interface rather than implementation
* If factories are designed for, retrieve objects using factories rather than direct instantiations

#### Example With Error

This example illustrates not programming to an interface and factory not used. For CRUD persistence of BRANCH entity, a BranchDAO is created which implements a common interface for CRUD operations, i.e. AppCRUDDAO. However, instance of BranchDAO is directly created. Factory is not used for getting this DAO, whereas flexibility of plugging-in customized Branch DAO was required for the product

*BranchDAO dao = new BranchDAO(); dao.addRecord();*

#### Example With Error Rectified

*AppCRUDDAO dao = AppDAOFactory.getDAO(AppConstants.BRANCH\_ENTITY); dao.addRecord()*

## H2) Align to multi-layered design or separation of concerns

 Ensure no violation of multi-layered design or separation of concerns, e.g.

* Technology layers (e.g. EJB, MDB) also perform business processing or database operations
* Business logic classes also perform database operations
* JSPs perform business logic / validations and / or database operations in addition to presentation
* Controller class executes business logic or database operations
* External system integration related code part of business logic or database access classes

#### Example With Error

This example illustrates separation of concerns not met – Technology class doing business operations and database operations

* The class XyzSLSBean is an EJB directly doing business operations
* Besides the business logic is intermingled with data access code

public class XyzSLSBBean implements javax.ejb.SessionBean

{

....

public void businessInterfaceMethod(ValueObject pVObject) throws XYZException

{

....

// .. some business logic if(!pVObject.isHostHoliday())

{

requestAccountBalances(pVObject);

...

}

// .... database operations also happening here conn = ConnectionWrapper.getConnection(); pstmt = conn.prepareStatement(sqlString); pstmt.setString(1,status); pstmt.setLong(2,processId);

...

pstmt.executeUpdate();

....

}

}

#### Example With Error Rectified

public class XyzSLSBBean implements javax.ejb.SessionBean

{

....

private XyzImpl mImpl = new XyzImpl();

public void businessInterfaceMethod(ValueObject pVObject) throws XYZException

{

mImpl.businessInterfaceMethod(pVObject);

}

}

public class XyzImpl

{

public void businessInterfaceMethod(ValueObject pVObject) throws XYZException

{

// business logic for the method if(!pVObject.isHostHoliday())

{

requestAccountBalances(pVObject);

...

}

// delegate db operations to DAO

// Get DAO handle from factory dao.updateBalance(pVObject);

...

}

}

## H3) Implement the design patterns correctly

 All applications require design for some Java / J2EE design patterns. Ensure they are implemented or used properly

* + Implementation of Factory classes, Service Locators, Singletons
  + Have DAOs implement an interface and retrieve from a factory
  + Use ServiceLocator to get handle to EJBs or data sources

#### Example With Error

This example illustrates Factory pattern not implemented as intended

* + The factory class AppDAOFactory always creates a new instance whereas DAOs should be stateless and can be cached
  + It does not get the class name to be used from a configuration file

public static Class AppDAOFactory

{

public static AppDAO getAppDAO(String pEntityName)

{

if(pEntityName.equals("BRANCH")) return new BRANCHDAO();

else if ...

}

}

#### Example With Error Rectified

public static Class AppDAOFactory

{

private static Hashtable mAppDAOs = new Hashtable();

public static AppDAO getAppDAO(String pEntityName)

{

AppDAO dao = (AppDAO)mAppDAOs.get(pEntityName); if(dao == null)

{

...

String className = null;

/\* read the class name to be instantiated from the config file \*/

className = DAOConfigFileLoader.getValue(pEntityName); Class class = Class.forName(className,true,

AppDAOFactory.class.getClassLoader()); dao = (AppDAO)class.newInstance();

...

}

return dao;

}

}

## H4)Do not assume English as the language while displaying labels or reading data from HttpRequest or formatting data for display on screen

Do not assume English as the language (except while printing messages to the log file)

* + Locale aware Java APIs (NumberFormat, SimpleDateFormat) not used in creating numbers

/ dates from String data available in HttpRequest during form submit. Identical scenario while formatting them for display

* + Locale aware Java APIs (MessageFormat, ChoiceFormat) not used for displaying formatted messages
  + Currently this support only English language, we will evolve to other languages other than English if requ

#### Example With Error

This example illustrates locale aware Java APIs not used while reading number & date data from HttpServletRequest

* Date corresponding to Fund\_date not created using a “Locale” aware SimpleDateFormat
* Number corresponding to Amt\_limit not created using a NumberFormat class.

SimpleDateFormat sFormat=new SimpleDateFormat(datePattern);

requestData=request.getParameter("Fund\_date"); if(requestData!=null && !requestData.equals(""))

{

obj.put("Fund\_date", sFormat.parse(requestData));

}

requestData=request.getParameter("Amt\_limit"); if(requestData!=null && !requestData.equals(""))

{

obj.put("Amt\_limit", Integer.parseInt(requestData));

}

##### Example With Error Rectified

SimpleDateFormat sFormat=new SimpleDateFormat(datePattern,locale); requestData=request.getParameter("Fund\_date");

if(requestData!=null && !requestData.equals(""))

{

obj.put("Fund\_date", sFormat.parse(requestData));

}

NumberFormat nFormat = NumberFormat.getInstance(locale);

requestData=request.getParameter("Amt\_limit"); if(requestData!=null && !requestData.equals(""))

{

obj.put("Amt\_limit", new Integer(String.valueOf(nFormat.parse(requestData))));

}

# Maintainability, Ease of Support

## I1) Do not duplicate code

Ensure no duplication of, Methods

Business Logic

Encourage intra-application re-usability with creation of components.

## I2) Ensure code that is easy to change

 Avoid complicated / bulky method implementation – If the method is not understood in one read, it is not maintainable. If the method is performing more than one task, it is not maintainable. As a rule of thumb, a method should be restricted to *100* lines of code statements.

 Avoid lengthy classes – As a rule of thumb, a class should be restricted to *1000* lines of code statements

All classes and methods should have meaningful Javadocs



All methods to have inline comments where required to understand the implementation

## I3) Implement logging and exception handling standards

For ease of support & troubleshooting, it is very important to handle all exceptions, handle them the right way and log errors. As the standards for these are already covered in a separate chapter, they will not be repeated.

# Webservice guidelines

## J1) Webservice implementation guidelines

The Web service name should be a business noun, a business concept, or a business process

Determine the format and media type, which best matches the requirement and client needs

## J2) Error Handling

Always return representation that describes the error in detail, this includes response status, response headers and body containing the description of the error.