	Document # : TCS-DEL-iQMS-25

Java Coding Standards

Version 1.0

Delhi

February 2003

DOCUMENT RELEASE NOTICE

Java Coding Standards

Document Detail :

Name	Version No.	Description
Java Coding Standards	1.0	

Revision details:

Action taken (add/del/chg)	Preceding page No	New page No	Revision Description

This document and any revised pages are subject to document control. Please keep them up-to-date using the release notice from the distributor of the document.

Approved by : _____ Date :

Authorised by : _____ Date :

TATA CONSULTANCY SERVICES
PTI Building, Parliament Street
New Delhi – 11 00 01

About the document

Purpose

This manual contains important guidelines and coding standards for JAVA which will be used in projects using Java platforms.

Intended Audience

This document may be used to set up project specific coding standards for JAVA, if applicable.

Assumptions

The document assumes that the user has knowledge of the Java and object oriented programming. How to program in Java, the mechanics of the Java language and object oriented concepts are not covered in this document.

References

Java API specifications http://java.sun.com

Table Of Contents

This document contains a total of 18 pages

1. Coding Conventions

This document lays down the general conventions for Java usage for program development.

1.1 Commenting

The following chart describes the three types of java comments and suggested uses for them

Comment Type	Usage	Example
Documentation (Javadoc)	The javadoc utility process java code files and produce external documentation in the form of HTML files for the java program ..The details of javadoc is given below.	/** Customer – A customer is any person or organization that we sell services and products to. */
C style	Use C-style comments to document out lines of code that are no longer applicable, but that you want to keep just in case you users change their minds, or because you want to temporarily turn it off while debugging.	/* This code was commented out by RNA on Dec 9, 1997 because it was replaced by the preceding code. Delete it after two years if it is still not applicable. . . . (the source code) */

Single line/ non java doc comment	Use single line comments internally within methods to document business logic, sections of code, and declarations of temporary variables.	// Apply a 5% discount to all invoices
		// over $1000 as defined by the Sarek
		// generosity campaign started in
		// Feb. of 1995.

Use the following Javadoc tags where appropriate:

Tag	Used For	Purpose
@author name	Classes, Interface	Indicates the author(s) of the given piece of code. One tag per author should be used.
@exception name description	Member Functions	Describes the exception the member functions throws. One tag should be used for one exception with detail description.
@param name description	Member Function	Used to describe a parameter passed to a member function. One tag per parameter can be used.
@return description	Member Function	Describes the return value.
@see ClassName	Classes, Interfaces , Member Functions , Fields	Generate the hypertext link in the documentation to the specified class .
@see ClassName#member functionName	Classes, interfaces Member Functions , Fields	Generate the hypertext link in the documentation to the specified class .

1.2 Structure

In general, the code layout should be as follows :
//--- Class Header
//--- Packages
//--- Import Files
//--- Class Definitions
//--- Variables
//--- Method Header
//--- Method Definitions
//--- End of Source File

Class Header

```
/**---------------------------------------------------------------------------
Class Name:
Description:
Date of Creation:
Extends:
Implements:
Author:
---------------------------------------------------------------------------
Update Log:
Date:          By:          Details:
---------------------------------------------------------------------------*/
```

Packages

Create a new java package for each self-contained project or group of related functionality. If the package is part of a larger project, organize your packages under a reasonable named top-level package name. Package names should be all lower-case. Package names should conform to java examples when appropriate. Example: java.util, java.math.

Provide an index.html file in each directory briefly outlining the purpose and structure of the package.

Import Files Section

This contains all the system and user defined imported files used by the source file. The IMPORT path for the compiler environment will be appropriately defined in the "makefile". Wide Imports must not be used

i.e. Do not use
import java.awt.*;

Variables Section

Only the variables used in this file, i.e. the Instance & Static variables will be defined here.

Method Header

```
/**---------------------------------------------------------------------------
Method Name:
Description:
Date of Creation:
Input Arguments:
Return Parameters:
Exception Used:
---------------------------------------------------------------------------
Update Log:
Date:          By:     Details:
---------------------------------------------------------------------------*/
```

Inline Comments

All source code should be adequately commented. At least one space should be left after the end of the source line before starting the comments. Since all functions will have the header section, only code handling complex processing will require inline comments.
For example,

```
nUserId = function(); //comment
```

"//" operator should be used for inline comments.

For multiple level nesting, comments should be added at the end of the loop.

Eg:-

```
while (X == Y)
{
} // end while (X == Y)
```

1.3 Java Documentation Conventions

A really good rule of thumb to follow regarding documentation is to ask yourself if you've never seen the code before, what information would you need to effectively understand the code in a reasonable amount of time.

In general, the following recommendations should be followed.

- Comments should add to the clarity of your code
- Avoid decoration, i.e. don't use banner-like comments
- Keep comments simple
- Write the documentation before you write the code
- Document why something is being done, not just what.

1.3.1 What To Document

The following chart summarizes what to document regarding each portion of Java code that is written.

Item	What to Document
Arguments/ Parameters	The type of the parameter What it should be used for Any restrictions or preconditions Examples
Attributes/ fields/properties	The description of the attribute The type of an attribute Document all applicable in variants Examples Concurrency issues visibility decisions
Classes	The purpose of the class, its visibility and package name. The development/maintenance history of the class.
Compilation units	Each class/interface defined in the class, including a brief description The file name and/or identifying information Copyright information
Interfaces	The purpose How it should and shouldn't be used
Local variables	Its use/purpose
Methods – Documentation	What and why the method does what it does What a method must be passed as parameters What a method returns Any exceptions that a method throws Visibility decisions How a method changes the object Include a history of any code changes Examples of how to invoke the method if appropriate Applicable preconditions and post conditions Document all concurrency
Methods – Internal Comments	Control structures Why, as well as what, the code does Local variables Difficult or complex code The processing order
Package	The rationale for the package The classes in the package

Table 2 - Documentation Required

1.4 Code

1.4.1 Formatting the code

- Tab characters must not be used in source code. Spaces must be used instead. Since the code is truly 'portable' we cannot infer the platforms that the code will need to run, and the sophistication of the editors and viewers on those platforms, as such it is safer to use spaces.
    - Methods are separated by two lines of white space.
    - Comments are separated by methods by one line of white space.
    - Long and/or complex code blocks should have closing brace commented to clearly identify the block the brace is, in fact, closing.
    - All class variables are declared at the top of the class.
    - Methods should be declared in the following order: Constructors, Finalizers, Initializers, then blocks of public methods with their private helper methods.
    - Instance variables should be organized in the following order: public, protected, private.
    - Deep nesting levels should be avoided.

Examples

IF / ELSE

Place the IF keyword and conditional expression on the same line.

```
if (expression) {
statement;
}
else {
   statement;
}
```

WHILE

```
while (expression) {
   statement;
}
```

DO..WHILE

```
do {
      statement;
  } while (expression);
```

SWITCH

```
  switch (expression) {
 case n:
     statement;
     break;
   case x:
     statement;
     // Continue to default case
   default:          //always add the default case
     statement;
     break;
 }
```

TRY/CATCH/FINALLY

```
   try {
   }
catch (ExceptionClass e) {
   statement;
     statement;
   }
finally {
     statement;
   }
```

1.4.2 Coding Guidelines

There are many conventions and standards, which are critical to the maintainability and enhance ability of the code. It is essential to program in such a manner that other programmers in the team can understand the code easily. Making the code understandable to others is of utmost importance.

Convention Target	Convention
Accessor methods	Use accessors for obtaining and modifying all attributes Use accessors for 'constants" For collections, add methods to insert and remove items Whenever possible, make accessors protected, not public
Attributes	Attributes should always be declared private Don't directly access attributes, instead use accessor methods Don't hide names always initialize static attributes.
Classes	Minimize the public and protected interfaces Declare the attributes and methods of a class in the following order: • Static Blocks if any • Constructors • finalize() • public methods • protected methods • private methods • private attributes

Convention Target	Convention
Local variables	Don't hide names Declare one local variable per line of code Document local variables with an inline comment Declare local variables in the beginning of methods.
Methods	Document your code Paragraph your code

Table 3 - Coding Guidelines

1.4.3 Suggestions :

S.No.	Description
1.	Minimize * forms of import. Be precise about what you are importing. Check that all declared imports are actually used.
2.	When sensible, consider writing a main for the principal class in each program file. The main should provide a simple unit test or demo.
3.	For self-standing application programs, the class with main should be separate from those containing normal classes. Hard-wiring an application program in one of its component class files hinders reuse.
4.	Consider writing template files for the most common kinds of class files you create: Applets, library classes, application classes.
5.	If you can conceive of someone else implementing a'class's functionality differently, define an interface, not an abstract class. Generally, use abstract classes only when they are "partially abstract"; i.e., they implement some functionality that must be shared across all subclasses.
6.	Consider whether any class should implement Cloneable and/or Serializable.
7.	Declare a class as final only if it is a subclass or implementation of a class or interface declaring all of its non-implementation-specific methods. (And similarly for final methods).
8.	Minimize reliance on implicit initializers for instance variables (such as the fact that reference variables are initialized to null).
9.	Minimize statics (except for static final constants).
10.	Generally prefer long to int, and double to float. But use int for compatibility with standard Java constructs and classes (for the major example, array indexing, and all of the things this implies, for example about maximum sizes of arrays, etc).
11.	Use final and/or comment conventions to indicate whether instance variables that never have their values changed after construction are intended to be constant (immutable) for the lifetime of the object (versus those that just so happen not to get assigned in a class, but could in a subclass).

S.No.	Description
12.	Generally prefer protected to private.
13.	Minimize direct internal access to instance variables inside methods. Use protected access and update methods instead (or sometimes public ones if they exist anyway).
14.	Avoid giving a variable the same name as one in a superclass.
15.	Prefer declaring arrays as Type[] arrayName rather than Type arrayName[].
16.	Ensure that non-private statics have sensible values even if no instances are ever created. (Similarly ensure that static methods can be executed sensibly.) Use static intitializers (static { ... }) if necessary.
17.	Write methods that only do “one thing". In particular, separate out methods that change object state from those that just rely upon it. For a classic example in a Stack, prefer having two methods Object top() and void removeTop() versus the single method Object pop() that does both.
18.	Define return types as void unless they return results that are not (easily) accessible otherwise. (i.e., hardly ever write “return this").
19.	Avoid overloading methods on argument type.
20.	Prefer synchronized methods to synchronized blocks.
21.	If you override Object.equals, also override Object.hashCode, and vice-versa.
22.	Override readObject and WriteObject if a Serializable class relies on any state that could differ across processes, including, in particular, hashCodes and transient fields.
23.	If you think that clone() may be called in a class you write, then explicitly define it (and declare the class to implement Cloneable).
24.	Always document the fact that a method invokes wait.
25.	Whenever reasonable, define a default (no-argument) constructor so objects can be created via Class.newInstance().
26.	Prefer abstract methods in base classes to those with default no-op implementations. (Also, if there is a common default implementation, consider instead writing it as a protected method so that subclass authors can just write a one-line implementation to call the default.)
27.	Use method equals instead of operator == when comparing objects. In particular, do not use == to compare Strings.
28.	Always embed wait statements in while loops that re-wait if the condition being waited for does not hold.
29.	Use notifyAll instead of notify or resume.
30.	Declare and initialize a new local variable rather than reusing (reassigning) an existing one whose value happens to no longer be used at that program point.
31.	Assign null to any reference variable that is no longer being used. (This includes, especially, elements of arrays.)

S.No.	Description
32.	Avoid assignment''(``=") inside if and while conditions.
33.	Document cases where the return value of a called method is ignored.
34.	Ensure that there is ultimately a catch for all unchecked exceptions that can be dealt with.

35.	Embed casts in conditionals. For example:
	
	C cx = null;
	if (x instanceof C)© = (C)x;
	else evasiveAction();

36.	Most computers have only one CPU, so threads must share the CPU with other threads. The execution of multiple threads on a single CPU, in some order, is called scheduling. The Java runtime supports a very simple, deterministic scheduling algorithm known as fixed priority scheduling.
37.	Each Java thread is given a numeric priority between MIN_PRIORITY and MAX_PRIORITY (constants defined in the Thread class). At any given time, when multiple threads are ready to be executed, the thread with the highest priority is chosen for execution. Only when that thread stops, or is suspended for some reason, will a lower priority thread start executing.
38.	Scheduling of the CPU is fully preemptive. If a thread with a higher priority than the currently executing thread needs to execute, the higher priority thread is immediately scheduled.
39.	The Java runtime will not preempt the currently running thread for another thread of the same priority. In other words, the Java runtime does not time-slice. However, the system implementation of threads underlying the Java Thread class may support time-slicing. Do not write code that relies on time-slicing.
40.	When all the runnable threads in the system have the same priority, the scheduler chooses the next thread to run in a simple, non-preemptive, round-robin scheduling order.
41.	The best choice is to prevent deadlock rather than to try and detect it. Deadlock detection is complicated and beyond the scope of this tutorial. The simplest approach to preventing deadlock is to impose ordering on the condition variables.
42.	At any given time, the highest priority thread is running. However, this is not guaranteed. The thread scheduler may choose to run a lower priority thread to avoid starvation. For this reason, use priority only to affect scheduling policy for efficiency purposes. Do not rely on thread priority for algorithm correctness.
43.	Document fragile constructions used solely for the sake of optimization.
44.	For good design minimize the coupling between classes. The general rule of thumb is to be as restrictive as possible when setting the visibility of a method. If method doesn't have to be public then make it protected, and if it doesn't have to be protected then make it private.

–Table 4 - Suggestions

2. ROUTINES

2.1 High Quality Routines

2.1.1 Cohesion

Cohesion refers to how closely the operations in a routine relates to each other. The goal is for the routine to do only one thing but to do it well.

2.1.2 Loose Coupling

Coupling represents the relationships of one routine to other routines. The goal is to have a routine as independent as possible. Any other routine should be able to call this routine.

2.1.3 Defensive Programming

Routines should contain protection against bad data. No assumption should be made that data will be valid. A routine should do the following.

- Check the values of all data input from external source
- Check the values of all routine input parameters
- Decide how to handle bad parameters (depending on the circumstances, you might want to return an error code, return a neutral value, substitute the next piece of valid data and continue as planned, return the same answer as the previous time, use the closest legal value, call an error-processing routine, log a warning message to a file, print an error message, or shut down).
- Use exception handling to deal with unexpected cases.
- Design the routines so that likely changes will not require redesign.

However, too much defensive programming can render the program fat and slow. Use defensive programming but, for productive code, use it only where you feel it is needed.

3. Exception, Testing and Debugging

3.1 Overview

Code should be written to trap run-time exceptions and correct the situation, prompt the user for action, or save data before ending the program. Error handling should be done at both the places, Server side and Client side.

3.2 Exception Handling

When using the try { } catch mechanism *always* provide a default handler for the exception. It can be very difficult to debug code that throws an exception which is simply ignored.
It is good practice to provide an exception handling method that deals with all general and even insignificant exceptions e.g.

```
try {
   ….
   ….
    } catch (Exception e) { handleException(e); }

private void handleException(Exception e)
{
    e.printStackTrace();
```

```
}
```

Following this rule makes it easier to debug code and allows for greater control e.g. it is a relatively simply matter to redirect all exceptions to an error log or to prevent them being visible to the user when the system is delivered.

3.3 Testing

Test with a recognized set of current browsers (for applets) and be aware of recent changes that may impact roll-out with older versions. Use Activator if this is a problem.

3.4 Debug Statements

Standard Debug Methods should be used across all modules in the project to ensure project standards and consistency. The Debug Package will contain debug classes derived from the superclass debug.
All debug code should be enclosed in a debug flag as given below :

```
if (Debug.DEBUG)
{
// debug code
}
```

The compiler will remove the dead code during optimization if DEBUG is defined as a constant of false.

3.5 Considerations

3.5.1 Performance

Consider network traffic when trying to design performance systems. As classes are only loaded as required partitioning into more classes will aid re-use.

3.5.2 Threading

Java makes writing multi-threaded code trivial, however, remember that code needs to 'thread safe'. Always consider the implications of two or more threads executing a piece of code simultaneously even if you are not specifically writing a multithreaded module. JavaBeans, JCO's and Enterprise Beans may be executed on multithreaded servers and cannot therefore be assumed to be single threaded.
Using the 'synchronized' keyword on method declarations will serialize access to the method i.e. it will only allow one thread at a time to execute, the others will be blocked. It should therefore be used sparingly and only where necessary.

3.5.2.1 Inner Classes

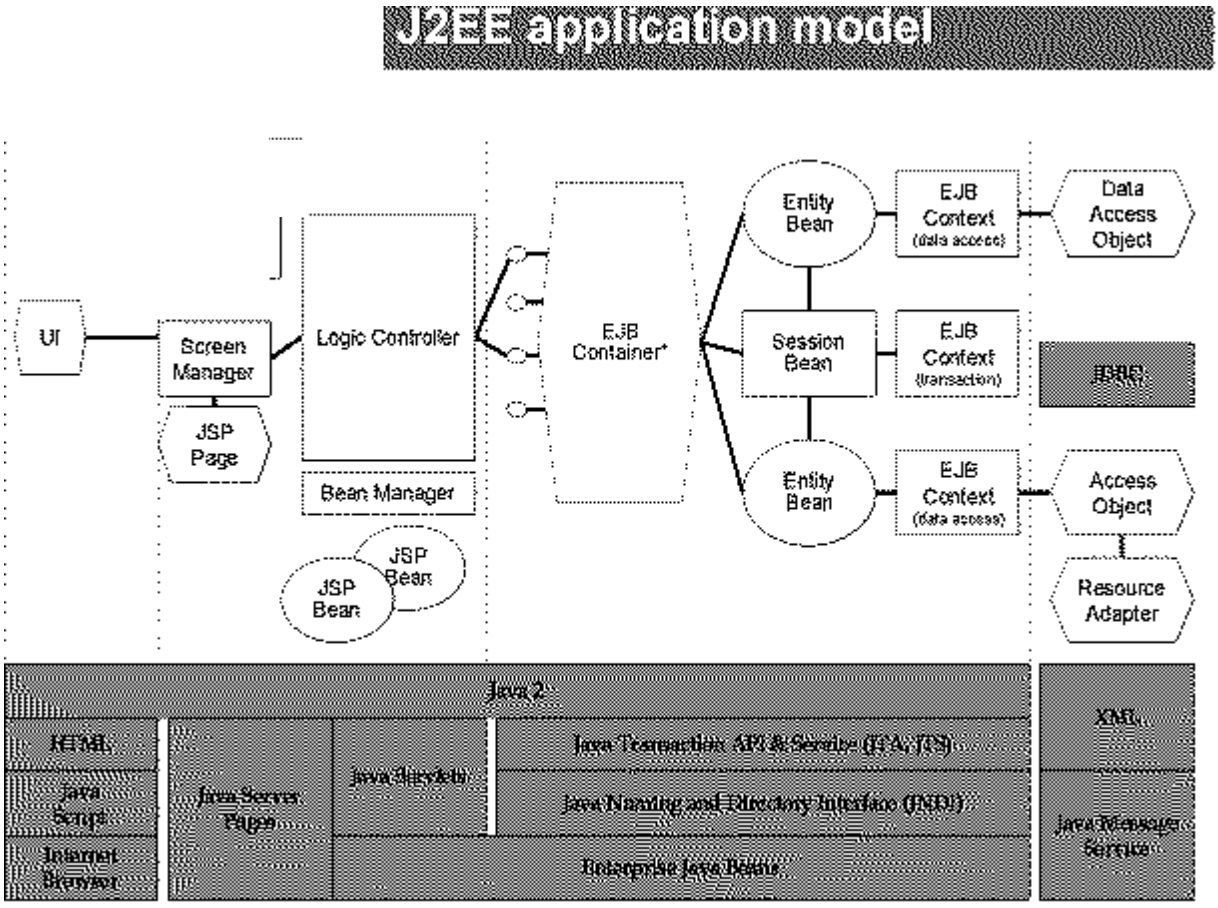
JDK 1.1 introduced inner classes specifically to tidy up the complex event handling mechanism. They are a convenient way of presenting a different interface to a class but can quickly lead to poorly structured code and many extra classes to download.
The preferred method of implementing simple Event Listeners is to implement the Listener interface and to provide the required method to process the event ensuring that the method is commented as belonging to the interface.
More complex listeners are best implemented as conventional classes as this encourages re-use.

3.5.2.2 Abstract Classes

Consider the use of abstract classes instead of interfaces when in a single inheritance tree. Abstract classes allow for the provision of some default implementation whilst still mandating a specific implementation of the class (subclass). Remember, however, that java only supports single inheritance at the implementation level which may limit the use of abstract classes

Enterprise Java Overview

The IM Systems component will utilize the Sun Java™ 2 Platform, Enterprise Edition (J2EE) methodology, or Enterprise Java for short, for much of the functionality. This standards definition document will cover the standard approach that should be used when developing these components for the middle tier of the model. The enterprise Java model is depicted by the following diagram.



EJB Overview

Enterprise Java Beans are a server-side component architecture piece of the Java 2 Platform, Enterprise Edition (J2EE) or Enterprise Java for short. The EJB specification allows the developer to encapsulate business logic in a component framework without worrying about low-level details. Details such as multi-threading, remote invocation, state management, etc are handled by the EJB 'container'.

Because EJB's are written in Java the developer has the advantage of a powerful object-oriented language and the write-once run anywhere mantra of the Java

programming language. With the EJB specification the developer also has the added advantage of being able to move their EJB from platform to platform and EJB-compliant server to EJB-compliant server.

There are two basic types of EJB's. An Entity Bean that represents persistent data that is stored in a database and a Session Bean that provides some service but does not represent persistent data.

3.6 Entity Beans Overview

An entity bean represents persistent data stored in a database. This bean will contain methods to manipulate this entity and attributes that contain data about the entity.

The persistence of the data for an entity bean can either be managed by the 'container', or can be 'bean-managed' in which the developer writes the code for the entity bean to store its data. For the best flexibility, maintainability and interoperability, persistence within entity beans should be 'bean-managed'. For more details on why entity beans should be 'bean-managed' see the section on 'persistence' under Entity Beans Specifics'.

3.7 Session Beans Overview

A session bean is not persistent and exists to serve a single client. The server creates a Session Bean when it receives a request and maintains the Session Bean until the connection is closed. A session bean typically implements business logic or procedural logic.

A session bean can be stateful of stateless. A stateful session bean maintains data between client calls until the client connection is closed and therefore there is a one-to-one mapping between a stateful session bean and a client. A stateless session bean does not maintain data between calls and therefore the same session bean can be called by many clients.

3.8 EJB specific Rules

To implement an enterprise bean, two interfaces and one or two classes need to be defined: Remote interface, Home interface, Bean class, and Primary key (for entity beans only). When documenting the enterprise bean as a whole, its remote interface, home interface, bean class, and so forth, should be referred to by its remote-interface name followed by the word "bean." For example, an enterprise bean that is developed to model a cabin on a ship will be called the "Cabin bean." The remote interface for the Cabin bean would be called the **Cabin** remote interface, the home interface **CabinHome**, the bean class itself **CabinBean**, and the primary key would be called **CabinPK**. Following are descriptions and examples for the different interfaces and classes:

3.8.1 The remote interface

The remote interface for an enterprise bean defines the bean's business methods: the methods a bean presents to the outside world to do its work. Example:

```
import java.rmi.RemoteException;

public interface Cabin extends javax.ejb.EJBObject
{
```

```
      public String getName() throws
RemoteException;
            ...
}
```

3.8.2 The home interface

The home interface defines life-cycle methods and methods for looking up beans. Example:

```
import java.rmi.RemoteException;
import java.ejb.CreateException;
import java.ejb.FinderException;

public interface CabinHome extends
javax.ejb.EJBHome {
      public Cabin Create(int id)
            throws CreateException,
RemoteException;
      ...
}
```

3.8.3 The bean class

Here is an example of an actual bean:

```
import java.ejb.EntityContext;

public class CabinBean implementss
javax.ejb.EntityBean {
      public int id;
      public String name;
      public int deckLevel;
            ...
}
```

3.8.4 The primary key

The primary key is a pointer that helps locate data that describes a unique record or entity in the database; it is used in the *findByPrimaryKey()* method of the home interface to locate the specific entity. Example:

```
Public class CabinPK implements
java.io.Serializable {
      public int id;

      public int hashCode() {
            return id;
      }
            ...
}
```

4. Checklist

	Code Review Checklist	
Module:	Tester:	Version: Date:

Type	Check	Result
Source Code Formatting	Tab Characters not used	
	Structure conforms to standards	
	Comments conform to standards	
–	Layout - {} explicitly included	
Naming Conventions	Package Names conform to standards	
	Class Names conform to standards	
	Method Names conform to standards	
	Variable Names conform to standards	
General Standards	Default Visibility not used	
	Wide Imports not used	
	Class constructors supplied	
	Standard methods and templates used	
	Classes not defined as final unless appropriate	
Exception, Testing and Debugging	Default Handlers provided for exceptions	
	Standard Debug Methods used	
Considerations	Performance	
	Static Declarations not used unless appropriate	
	Code is Threadsafe	
	Inner Classes not used unless appropriate	
	The use of Dynamic Query must be fully justified.	
	An interface class must exists for each content provider class.	
	Interface for content provider must contain definition of each method present in the content provider	
	The content provider must implement it's interface and this must not be commented at the time of compilation.	
	If nothing is entered, on a particular field on the screen it must not be passed as 'space' to the database API.	
	Ensure that if a read/write to the session is being done , then a session must exist prior to that.	
	If a null value is supplemented with a gap or a space , it should be reverted back to " null" while sending it to API.	
Documentation Standards	Documentation standards defined in section 2.3.1 used	
	Abstract Classes used appropriately	

5. ANNEXURE I

Glossary of terms commonly referred to :

Accessor – A method that either modifies or returns the value of an attribute. Also known as an access modifier. See getter and setter.

Attribute – A variable, either a literal data type or another object that describes a class or an instance of a class. Instance attributes describe objects (instances) and static attributes describe classes. Attributes are also referred to as fields, field variables, and properties.

Block – A collection of zero or more statements enclosed in (curly) braces.

Class – A definition, or template, from which objects are instantiated.

Component – An interface widget such as a list, button, or window.

Concurrency strategy – Any class that implements the interface runnable should have it's concurrency strategy fully described. It is also important to document why the particular strategy was chosen over others. A common concurrency strategy will include synchronized objects.

Constructor – A method which performs any necessary initialization when an object is created.

C-style comments – A Java comment format, /* … */, adopted from the C/C++ language that can be used to create multiple-line comments. Commonly used to "document out" unneeded or unwanted lines of code during testing.

Document applicable invariants – An invariant is a set of assertions about an instance or class that must be true at all stable times. Here a stable time is defined as the period before a method is invoked on the object / class and immediately after a method is invoked. Documenting the invariants of a class provides valuable insight to other programmers as to how a class can be used.

Documentation comments – A Java comment format, /** … */, that can be processed by javadoc to provide external documentation for a class file. The main documentation for interfaces, classes, methods, and attributes should be written with documentation comments.

finalize() – A method that is automatically invoked during garbage collection before an object is removed from memory. The purpose of this method is to do any necessary cleanup, such as the closing of open files.

HTML – Hypertext markup language, an industry-standard format for creating web pages.

Inline comments – The use of a line comment to document a line of source code where the comment immediate follows the code on the same line as the code. Single line comments are typically used for this, although C-style comments can also be employed.

Interface – The definition of a common signature, including both methods and attributes, which a class that implements an interface must support. Interfaces promote polymorphism by composition.

javadoc – A utility included in the JDK that processes a Java source code file and produces an external document, in HTML format, describing the contents of the source code file based on the documentation comments in the code file.

Lazy initialization – A technique in which an attribute is initialized in its corresponding getter method the first time that it is needed. Lazy initialization is used when an attribute is

not commonly needed and it either requires a large amount of memory to store or it needs to be read in from permanent storage.

Local variable – A variable that is defined within the scope of a block, often a method. The scope of a local variable is the block in which it is defined.

Method – A piece of executable code that is associated with a class, or the instances of a class. Think of a method as the object-oriented equivalent of a function.

Name hiding – This refers to the practice of using the same, or at least similar, name for an ATTRIBUTE/VARIABLE/ARGUMENT AS FOR ONE OF HIGHER SCOPE. THE MOST COMMON ABUSE OF NAME HIDING IS TO NAME A LOCAL VARIABLE THE SAME AS AN INSTANCE ATTRIBUTE. NAME HIDING SHOULD BE AVOIDED AS IT MAKES YOUR CODE HARDER TO UNDERSTAND AND PRONE TO BUGS.

FEEDBACK FORM

To : SEPG
From :
Project :
Date :
Location :

Please use the space below to give your comments, suggestions, queries or to point out errors. Please use additional sheets, if necessary.

For SEPG use

Assigned to :
Date :
Action taken :