Coding Standards for Application Development

TABLE OF CONTENTS

1. NAMING CONVENTION

1.1. FILE NAMES

- The Java files must have .java extension.

  Name File name: Ccccccccc.Java

  Ccccccccc - Meaningful class name

  First few letters are to be used for describing file functionality.

  Use uppercase for the first letter of the functionality as well as for the first letter.

  - Servlet should be identified by assigning suffix as “Servlet”

    E.g. StaffServlet
  - Action class should be identified by assigning suffix as “Action”

    E.g. StaffAction
  - Action class for operations will have suffix as “Operation”

    E.g. CreateStaffAction
  - Delegates should be identified by assigning suffix as “Delegate”

    E.g. StaffDelegate
  - Delegate Factory should be identified by assigning suffix as “DelegateFactory”

    E.g. StaffDelegateFactory
  - Abstract Class should be identified by assigning prefix as “Abstract”

    E.g. AbstractPolicy
  - Plain Old Java Object (POJO) Class should be identified by assigning suffix as “Entity”

    E.g. PolicyEntity
  - Data Access Object (DAO) Class should be identified by assigning suffix as “DAO”

    E.g. PolicyDAO

1.2. CONSTANTS

- Constants should be written in upper case letters only.
- Underscore can be used to separate meaningful words.
- The names should be self- explanatory.
- Write comments with each constant.

  E.g. int final LOGIN_SUCCESS = 1;

Note: Constant range is fixed for each module. Refer to "Constant Range.doc" for more detail

1.3. METHOD NAMES

For all methods, start the method name with lowercase and the keyword to start with uppercase (Hungarian Notation)

Example: toString();

getValues (...);

getAccountNumber (...) ;

1.4. VARIABLE NAMES

SR. NO.	TYPE	PREFIX (LOWERCASE)	EXAMPLE
1.	Integer Number	i	iCount, iCustomerCount, iSize, iLength, etc.
2.	Double	d	dAmount, dPrice, etc.
3.	Float	f	fAmount, fPrice, etc.
4.	Boolean	b	bAutoCommit, bDistributedLoaded, etc.
5.	Date	date	dateCustExpire, dateCustCreate, dateDeactivate, etc.
6.	String	str	strText, strHelp, strMessage, strUserName, etc.
7.	StringBuffer	sb	sbText, sbMessage, etc.
8.	Vector	vct	vctUserNames, vctCustInfo,, etc.
9.	Hashtable	htb	htbCustInfo, htbRategroupInfo, etc.
10.	Char	ch	chActive, chUnlimitedDay, etc.
11.	Enumeration	enum	enumRows, enumElements, enumServices etc
12.	Iterator	iter	iterServices
13.	Collection	coll	collServices

Figure 1: Variable Naming convention

2. FILE ORGANIZATION

A file consists of sections that should be separated by blank lines and optional comments identifying each section.

2.1. JAVA SOURCE FILES

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, these can be put in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files should have the following ordering

- Package statements
- Import statements
- Beginning Comments
- Class and interface declarations
- Class (static) variables. The order of these variables should be public, protected and then private
- Instance variables. The order of these variables should be public, protected and then private
- Constructors

3. COMMENTS

3.1. GENERAL

Use the javadoc format of /** Comments */. This enables a documentation of the class in HTML format to be carried out by running the javadoc documentation tool provided by JDK.

- javadoc generates documentation only for variables and methods with public, protected or private protected access. Any comments relating to private or default package variables and methods are not translated to the output documentation.

This is in keeping with the object oriented methodology where only what is accessible to other classes should be revealed. Please refer appendix 1for more details on documentation using javadoc

3.2. COMMENTS FOR A CLASS

Following should be incorporated just above the class definition

```
/*
@author:
Functionality description:
        What it is supposed to do. Any algorithms used or books referred.
External Methods Called:
Start Date:
Modification log:
*/
```

3.3. COMMENTS FOR A METHOD

Just above a method explain the following

```
/**
    *DESCRIPTION
    *@param <name of param> description
    *@return description of the return value
*/
```

3.4. BLOCK COMMENTS

A blank line to set it apart from the rest of the code should precede a block comment.

Block comments should be formatted as follows

```
/*
 *  This is a block comment
```

```
*/
```

3.5. END OF LINE COMMENTS

Use the //comment delimiter to put in comments at the end of a line. This should not be used to comment out multiple lines of code.

4. CODING STYLE

4.1. GENERAL

The style followed must be readable. The same style has to be followed through out the class and application.

```
Void myFn (...){
    …
}
```

Whatever the convention is followed, the code must be aligned properly. Use the same tab size for all the files.

4.2. LINE LENGTH

Avoid line longer than 80 characters as readability becomes an issue.

4.3. WRAPPING LINES

When an expression will not fit on a single line, break it according to these general principles

- Break after a comma
- Break before an operator
- Align the new line with the beginning of the expression at the same level on the previous line
- If the above rules lead to confusing code, just indent 8 spaces instead.

E.g.

```
Method1(longExpression1, longExpression2, longExpression3,
LongExpression4, longExpression5);
Var2 = longName1 + longName2 –
longName3 * (longName4 + LongName5)
```

5. DECLARATIONS

5.1. NUMBER PER LINE

One declaration per line is recommended since it encourages commenting.

E.g. String strName ;//Name of Person
String strType; // Type of Participant

5.2. PLACEMENT

Put declarations only at the beginning of blocks. Don't wait to declare variables before their first use; it can lead to confusion

5.3. INITIALIZING VARIABLES

Java initializes the standard data type variables viz. int, char, Boolean, float etc. But all classes that have the Object class to be the parent are initialized to null. So care must be taken to initialize these variables first.

E.g. when an instance of a String, or instance of a Vector or Hash table are used. They must be initialized as follows

String strName= new String();

6. SECURED CODING PRACTICE

6.1. 24ONLINE SECURED CODING PRACTICE

24online follows secured coding practice –

1] Input parameters are encoded such that it doesn't create XSS attack.

For example, if input is taken from GUI form, it should be encoded to prevent XSS attack.

2] Prepared statement are used instead of create statement, whenever it is feasible.

3] Input is not used directly in queries, so as to protect against SQL injection.

For example, data in used queries is encoded in a way that it will not create SQL injection attack.

4] For data manipulation operation, user is authenticated and authorised against csrf by passing csrf token whenever necessary.

5] hosts.allow contains GSS entries only.

6] Any port on server is not open except standard service ones like httpd, SSH, etc.

7] Risks and problems that can be generated are identified before integrating any new API. Resolutions to such problems or risks are known first and applied while using that particular API.

Refer: http://cvedetails.com for risk and problem identification

8] Cookie being used is marked with http only flag.

9] Auto completion of authentication fields is disabled.

10] Authentication details like passwords, etc are passed and stored securely. 11] Software and API version details are not disclosed to end-clients/users.

12] Database access is restricted to 24online server ip address only.

13] The credit card, net banking, or any other financial related details of the user are not stored by 24online.

14] Console user's passwords are used using /etc/shadow.

15] All unnecessary services like FTP, SNMPD, Telnet are disabled.

16] Unwanted and default users are disabled in the system.

17] Access of each content and application is granted according to the privileges.

18] Database access is restricted from anywhere outside the application.

6.2. SECURITY FEATURE OF JAVA

The security manager makes decisions about access to sensitive resources, which is an instance of the *SecurityManager* class. Once a security manager has been set using *System.setSecurityManager*, it is always a security violation to try to set a new security manager. There can be only one security manager in an application. The implementation of the *SecurityManager* class that is part of the core Java library is not very useful because it is an implementation that ***denies*** everything no matter what the source. To implement a security policy, the *SecurityManager*, must be subclassed, overriding the methods to determine resource access according to policies that are defined. But override only methods that are required. By allowing access, there could be many subtle implications which must be investigated. The security manager doesn't have all the information that it needs to do a good job of resource tracking. For instance when the *checkAccess(Threadgroup)* method is called, the security manager doesn't know whether a thread is being created or an entire thread group is being destroyed.

7. GOOD PROGRAMMING PRACTICE

7.1. PROVIDING ACCESS TO INSTANCE AND CLASS VARIABLES

Don't make any instance or class variable public without proper reason.

Often, instance variables don't need to be explicitly set or gotten—often that happens as a side effect of method calls.

E.g.

Case where the class is essentially a data structure, with no behavior. In other words, if you would have used a struct instead of a class (if Java supported struct), then it's appropriate to make the class's instance variables public.

7.2. REFERRING TO CLASS VARIABLES AND METHODS

Avoid using an object to access a class (static) variable or method. Use a class name instead.

For example:

classMethod(); //OK

AClass.classMethod(); //OK

anObject.classMethod(); //AVOID!

7.3. VARIABLE ASSIGNMENTS

- Avoid assigning several variables to the same value in a single statement. It is hard to read.

  Example:

  fooBar.fChar = barFoo.lchar = 'c'; // AVOID!

- Do not use the assignment operator in a place where it can be easily confused with the equalityoperator.

  Example:

  if (c++ = d++) { // AVOID! Java disallows

  ...

  }

  should be written as

  if ((c++ = d++) != 0) {

  ...

  }

- Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler, and besides, it rarely actually helps.

  Example:

  d = (a = b + c) + r; // AVOID!

  should be written as

  a = b + c;

  d = a + r;

8. MISCELLANEOUS

8.1. POINTS TO BE CONSIDERED

SR. NO	POINT	DESCRIPTION
1.	Import of Classes	Import only the classes that are required from a package. Import the entire package only when a large number of classes from that package are going to be used.
2.	Multiple decisions	In Java the switch - case statement can handle only chars, bytes and ints. Under this circumstance it may be required to write nested If statements. Restrict nested If's to three levels by using the fall through logic
3.	Creation of Packages	It is advisable to create packages at the outset of coding classes. This ensures that a full testing of all classes is carried out using the packaging. If done at the end it can cause multiple problems.
4.	Use of Static Class	it is resident in memory, if too many static classes are present it could affect the performance of the system or if multiple instances of the same applet exists it could result in overwriting the contents of the static class as in the case of any global variable.

8.2. USE OF THREADS

Java incorporates threads into its design. But some of the points to take into consideration before using threads.

- The memory requirements of a thread include the memory for its Thread object and perhaps Runnable object.
- On a low level, a thread has memory for its execution stack associated with it.
- A thread may require kernel resources from the underlying operating system
- The more threads, the more the system has to manage
- An application can be paralleled only by the number of available processors

A thread is a resource that should be used carefully. Not only can the use of threads increase the resource requirements of an application, they can also decrease its performance. An application that can be split among very independent threads is much easier to create than one where the threads require much interaction between them. The more threads there are that need to cooperate with one another, the more chances there are for subtle errors and

performance problems. It may very well turn out that a multithreaded application will be spending much of its time synchronizing rather than doing work. In that case, it makes better sense to decrease the threads or do the entire application in a single thread.

8.3. USE OF BROWSER COMPITIBILITY

1) IE 6 doesn't support **window.opener.closed** method. i.e. It always returns true, whether parent window is closed or open. In case of IE6 **window.opener.closed check** will always returns **true**. So beware of such checks.
   REF: http://www.shaftek.org/blog/2005/05/05/browser-differences-in-windowopener-behavior/

2) Firefox doesn't support Synchronous calls for Ajax. So with FireFox you should always use AJAX asynchronous call. In Ajax request.open(method,url,async) method async parameter should be true, so call such method to make asynchronous call.
   REF : http://www.codeguru.com/forum/archive/index.php/t-396169.html

3) Using window.opener.closed sometimes Firefox looks like that it hangs when parent window is closed.
   REF : https://bugzilla.mozilla.org/show_bug.cgi?id=165418