IT-632-Software Engineering

Cafeteria Management System (Cashless Canteen)

Coding Conventions Version 1.0

Team-2

Instructor – Professor Asim Banerjee

16th November, 2013

DA-IICT, Gandhinagar

Document Revision History:

Version	Primary Author(s)	Description	Reviewer(s)	Date
1.0	Irfan Wadia	Coding Convention	Ankush Gupta	16th november

Table of Contents

1. Introduction

1.1 Purpose

The goal of these guidelines is to create uniform coding habits among software personnel in the project teams so that reading, checking, and maintaining code written by different persons becomes easier. The intent of these standards is to define a natural style and consistency, yet leave to the authors of the project source code, the freedom to practice their craft without unnecessary burden.

When a project adheres to common standards many good things happen:

• Programmers can go into any code and figure out what's going on, so maintainability, readability, and reusability are increased.

• New people can get up to speed quickly.

• People new to a language are spared the need to develop a personal style and defend it to death.

• People new to a language are spared making the same mistakes over and over again, so reliability is increased.

• Idiosyncratic styles and college-learned behaviors are replaced with an emphasis on business concerns – high productivity, maintainability, shared authorship, etc.

• People make fewer mistakes in consistent environments.

1.2 Scope

This document describes general software coding standards for code written in any text based programming language. Each language specific coding standard will be written to expand on these concepts with specific examples, and define additional guidelines unique to that language.

1.3 Coding Standard Documents

Each project shall adopt a set of coding standards consisting of three parts:

• General Coding Standard, described in this document.

• Language specific coding standards for each language used. These language shall supplement, rather than override, the General Coding standards as much as possible.

• Project Coding Standards. These standards shall be based on the coding standards in this document and on the coding standards for the given language(s). The project coding standards should supplement, r a t h e r than override, the General Coding standards and the language coding standards. Where conflicts between documents exist, the project standard shall be considered correct.

1.4 Other Related Project Documents

The 'Software Life Cycle' and 'Configuration Management' policy standards define a set of documents required for each project, along with a process for coordinating and maintaining them. Documents referred to in this report include:

• High level Design Document(HLDD)

• Low Level Design Document(LLDD)

• Configuration Management (CM)

1.5 Terms Used In Document

• The term 'program unit' means a single function, procedure, subroutine or, in the case of various languages, an include file, a package, a task, a Pascal unit, etc.

• A 'function' is a program unit whose primary purpose is to return a value.

• A 'procedure' is a program unit which does not return a value (except via output parameters).

• A 'subroutine' is any function or procedure.

• An 'identifier' is the generic term referring to a name for any constant, variable, or program unit.

• A 'module' is a collection of 'program units' that work on a common domain.

2. FILE and MODULE GUIDELINES

This section lists commonly used file suffixes and names.

2.1 File Suffixes

We will use the following file suffixes:

File Type	Suffix
Java files	.java
Java bytecode	.class
HTML files	.html
CSS files	.css
Javascript files	.js
SQL files	.sql
XML Config Files	.xml

3. File Organization

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

Files longer than 2000 lines are cumbersome and should be avoided.

3.1 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, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files have the following ordering:

• Beginning comments (see “Beginning Comments” on page 4)

• Package and Import statements; for example:

```
import java.applet.Applet;

import java.awt.*;

import java.net.*;
```

• Class and interface declarations

3.1.1 Beginning Comments

All source files should begin with a c-style comment that lists the programmer(s), the date, a copyright notice, and also a brief description of the purpose of the program. For example:

/*

* *Classname*

*

* *Version info*

*

* *Copyright notice*

*/

3.1.2 Package and Import Statements

The first non-comment line of most Java source files is a `package` statement. After that, `import` statements can follow. For example:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```

3.1.3 Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear. See "Java Source File Example" on page 19 for an example that includes comments.

	Part of Class/Interface Declaration	**Notes**
1	Class/interface documentation comment (`/**...*/`)	See "Documentation Comments" on page 9 for information on what should be in this comment.
2	`class` or `interface` statement	
3	Class/interface implementation comment (`/*...*/`), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
4	Class (`static`) variables	First the `public` class variables, then the `protected`, and then the `private`
5	Instance variables	First `public`, then `protected`, and then `private`.
6	Constructors	
7	Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

4 - Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

4.1 Line Length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length—generally no more than 70 characters.

4.2 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.

• Prefer higher-level breaks to lower-level breaks.

• 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 or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

```
function(longExpression1, longExpression2, longExpression3,

longExpression4, longExpression5);

var = function1(longExpression1,

function2(longExpression2,

longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)

+ 4 * longname6; // PREFER

longName1 = longName2 * (longName3 + longName4

- longName5) + 4 * longname6; // AVOID
```

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION

someMethod(int anArg, Object anotherArg, String yetAnotherArg,

Object andStillAnother) {

...

}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS

private static synchronized horkingLongMethodName(int anArg,

Object anotherArg, String yetAnotherArg,

Object andStillAnother) {

...

}
```

Line wrapping for `if` statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION

if ((condition1 && condition2)

|| (condition3 && condition4)

||!(condition5 && condition6)) { //BAD WRAPS

doSomethingAboutIt(); //MAKE THIS LINE EASY TO MISS

}

//USE THIS INDENTATION INSTEAD

if ((condition1 && condition2)

|| (condition3 && condition4)

||!(condition5 && condition6)) {

doSomethingAboutIt();

}

//OR USE THIS

if ((condition1 && condition2) || (condition3 && condition4)
```

```
||!(condition5 && condition6)) {

doSomethingAboutIt();

}
```

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta

: gamma;

alpha = (aLongBooleanExpression)

? beta

: gamma;
```

5 - Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by `/*...*/`, and `//`. Documentation comments (known as "doc comments") are Java-only, and are delimited by `/**...*/`. Doc comments can be extracted to HTML files using the javadoc tool.

Implementation comments are mean for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective. to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or nonobvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters.

Comments should never include special characters such as form-feed and backspace.

5.1 Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing and end-of-line.

5.1.1 Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments should be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code. Block comments have an asterisk "*****" at the beginning of each line except the first.

```
/*
* Here is a block comment.
*/
```

Block comments can start with `/*-`, which is recognized by **indent**(1) as the beginning of a block comment that should not reformatted. Example:

```
/*
* Here is a block comment with some very special
* formatting that I want indent(1) to ignore.
*
* one
* two
* three
*/
```

Note: If you don't use **indent**(1), you don't have to use `/*-` in your code or make any other concessions to the possibility that someone else might run **indent**(1) on your code. See also "Documentation Comments" on page 9.

5.1.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format (see section 5.1.1). A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java code (also see "Documentation Comments" on page 9):

```
if (condition) {

/* Handle the condition. */

...

}
```

5.1.3 Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting. Avoid the assembly language style of commenting every line of executable code with a trailing comment.

Here's an example of a trailing comment in Java code (also see "Documentation Comments" on page 9):

```
if (a == 2) {

return TRUE; /* special case */

} else {

return isprime(a); /* works only for odd a */

}
```

5.1.4 End-Of-Line Comments

The `//` comment delimiter begins a comment that continues to the newline. It can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

```
if (foo > 1) {

// Do a double-flip.

...

}

else

return false; // Explain why here.

//if (bar > 1) {

//
// // Do a triple-flip.
// ...
//}
//else
// return false;
```

5.2 Documentation Comments

Note: See "Java Source File Example" on page 19 for examples of the comment formats described here.

For further details, see "How to Write Doc Comments for Javadoc" which includes information on the doc comment tags (`@return`, `@param`, `@see`):

```
http://java.sun.com/products/jdk/javadoc/writingdoccomments.html
```

For further details about doc comments and javadoc, see the javadoc home page at:

```
http://java.sun.com/products/jdk/javadoc/
```

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters `/**...*/`, with one comment per API. This comment should appear just before the declaration:

```
/**
* The Example class provides ...
*/
class Example { ...
```

Notice that classes and interfaces are not indented, while their members are. The first line of doc comment (`/**`) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter.

If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment (see section 5.1.1) or single-line (see section 5.1.2) comment immediately *after* the declaration. For example, details about the implementation of a class should go in in such an implementation block comment *following* the class statement, not in the class doc comment.

Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration *after* the comment.

6 - Declarations

6.1 Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level
```

```
int size; // size of table
```

is preferred over

```
int level, size;
```

In absolutely no case should variables and functions be declared on the same line. Example:

```
long dbaddr, getDbaddr(); // WRONG!
```

Do not put different types on the same line. Example:

```
int foo, fooarray[]; //WRONG!
```

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

```
int level; // indentation level

int size; // size of table

Object currentEntry; // currently selected table entry
```

6.2 Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void MyMethod() {
       int int1; // beginning of method block
       if (condition) {
              int int2; // beginning of "if" block
              ...
       }
}
```

The one exception to the rule is indexes of `for` loops, which in Java can be declared in the `for` statement:

```
for (int i = 0; i < maxLoops; i++) { ...
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;

...

func() {

	if (condition) {

		int count; // AVOID!

		...

	}

	...

}
```

6.3 Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

6.4 Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

• No space between a method name and the parenthesis "(" starting its parameter list

• Open brace "{" appears at the end of the same line as the declaration statement

• Closing brace "}" starts a line by itself indented to match its corresponding openingstatement, except when it is a null statement the "}" should appear immediately after the

"{"

```
class Sample extends Object {

	int ivar1;

	int ivar2;

	Sample(int i, int j) {

	ivar1 = i;

	ivar2 = j;

}

int emptyMethod() {}

	...

}
```

• Methods are separated by a blank line

7 - Statements

7.1 Simple Statements

Each line should contain at most one statement. Example:

```
argv++; argc--; // AVOID!
```

Do not use the comma operator to group multiple statements unless it is for an obvious reason.

Example:

```
if (err) {
        Format.print(System.out, "error"), exit(1); //VERY WRONG!
}
```

7.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`". See the following sections for examples.

• The enclosed statements should be indented one more level than the compound statement.

• The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

• Braces are used around all statements, even singletons, when they are part of a control structure, such as a `if-else` or `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

7.3 return Statements

A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

7.4 if, if-else, if-else-if-else Statements

The `if-else` class of statements should have the following form:

```
if (condition) {
        statements;
}
if (condition) {
```

```
        statements;
} else {
        statements;
}
if (condition) {
        statements;
} else if (condition) {
        statements;
} else if (condition) {
        statements;
}
```

Note: `if` statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!

statement;
```

7.5 for Statements

A `for` statement should have the following form:

```
for (initialization; condition; update) {

statements;

}
```

An empty `for` statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a `for` statement, avoid the complexity of using more than three variables. If needed, use separate statements before the `for` loop (for the initialization clause) or at the end of the loop (for the update clause).

7.6 while Statements

A `while` statement should have the following form:

```
while (condition) {

        statements;

}
```

An empty `while` statement should have the following form:

```
while (condition);
```

7.7 do-while Statements

A `do-while` statement should have the following form:

```
do {

       statements;

} while (condition);
```

7.8 switch Statements

A `switch` statement should have the following form:

```
switch (condition) {
       case ABC:
              statements;
              /* falls through */
       case DEF:
              statements;
              break;
       case XYZ:
              statements;
              break;
       default:
              statements;
              break;
}
```

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /* falls through */ comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

7.9 try-catch Statements

A `try-catch` statement should have the following format:

```
try {

       statements;

} catch (ExceptionClass e) {

       statements;

}
```

8 - Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier—for example, whether it's a constant, package, or class—which can be helpful in understanding the code.

The conventions given in this section are high level. Further conventions are given at (*to be determined*).

Identifier Type	Rules for Naming	Examples
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	`class Raster;` `class ImageSprite;`
Interfaces	Interface names should be capitalized like class names.	`interface RasterDelegate;` `interface Storing;`
Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	`run();` `runFast();` `getBackground();`
Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic— that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters.	`int i;` `char *cp;` `float myWidth;`
Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	`int MIN_WIDTH = 4;` `int MAX_WIDTH = 999;` `int GET_THE_CPU = 1;`

9. Java Source File Example

The following example shows how to format a Java source file containing a single public class.

Interfaces are formatted similarly. For more information, see "Class and Interface

Declarations" on page 4 and "Documentation Comments" on page 9

```
/*
* %W% %E% Firstname Lastname
*
* Copyright (c) 1993-1996 Sun Microsystems, Inc. All Rights Reserved.
*
* This software is the confidential and proprietary information of Sun
* Microsystems, Inc. ("Confidential Information"). You shall not
* disclose such Confidential Information and shall use it only in
* accordance with the terms of the license agreement you entered into
* with Sun.
*
* SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
* THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
* TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
* PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
* ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
* DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
*/
package java.blah;
import java.blah.blahdy.BlahBlah;
/**
* Class description goes here.
*
* @version 1.10 04 Oct 1996
* @author Firstname Lastname
*/
public class Blah extends SomeClass {
/* A class implementation comment can go here. */
/** classVar1 documentation comment */
      public static int classVar1;
/**
* classVar2 documentation comment that happens to be
* more than one line long
*/
      private static Object classVar2;
/** instanceVar1 documentation comment */
      public Object instanceVar1;
/** instanceVar2 documentation comment */
      protected int instanceVar2;
/** instanceVar3 documentation comment */
      private Object[] instanceVar3;

/**
* ...method Blah documentation comment...
*/
      public Blah() {
      // ...implementation goes here...
      }
/**
* ...method doSomething documentation comment...
*/
      public void doSomething() {
            // ...implementation goes here...
      }
/**
* ...method doSomethingElse documentation comment...
* @param someParam description
*/
      public void doSomethingElse(Object someParam) {
```

```
            // ...implementation goes here...
        }

}
```

10 Apache Struts 2

Apache Struts 2 is an elegant, extensible framework for creating enterprise-ready Java web applications. The framework is designed to streamline the full development cycle, from building, to deploying, to maintaining applications over time.

10.1 Configuration files

From a Struts developer point of view, the one required configuration file used by the framework is `web.xml`. From here, you have full control over how Struts configures both itself and your application. By default, Struts will load a set of internal configuration files to configure itself, then another set to configure your application, however it is possible to build an entire Struts application without writing a single configuration file other than `web.xml`.

The table lists the files that you can use to configure the framework for your application. Some configuration files can be reloaded dynamically. Dynamic reloading makes interactive development possible.

File	Optional	Location (relative to webapp)	Purpose
web.xml	no	/WEB-INF/	Web deployment descriptor to include all necessary framework components
struts.xml	yes	/WEB-INF/classes/	Main configuration, contains result/view types, action mappings, interceptors, and so forth
struts.properties	yes	/WEB-INF/classes/	Framework properties
struts-default.xml	yes	/WEB-INF/lib/struts2-core.jar	Default configuration provided by Struts
struts-default.vm	yes	/WEB-INF/classes/	Default macros referenced by `velocity.properties`
struts-plugin.xml	yes	At the root of a plugin JAR	Optional configuration files for Plugins in the same format as `struts.xml`.
velocity.properties	yes	/WEB-INF/classes/	Override the default Velocity configuration

10.1.1 web.xml

The web.xml web application descriptor file represents the core of the Java web application, so it is appropriate that it is also part of the core of the Struts framework. In the web.xml file, Struts defines its FilterDispatcher, the Servlet Filter class that initializes the Struts framework and

handles all requests. This filter can contain initialization parameters that affect what, if any, additional configuration files are loaded and how the framework should behave.

In addition to the FilterDispatcher, Struts also provides an ActionContextCleanUp class that handles special cleanup tasks when other filters, such as those used by Sitemesh, need access to an initialized Struts framework.

Key Initialization Parameters

config - a comma-delimited list of XML configuration files to load.

actionPackages - a comma-delimited list of Java packages to scan for Actions.

configProviders - a comma-delimited list of Java classes that implement the ConfigurationProvider interface that should be used for building the Configuration.

loggerFactory - The class name of the LoggerFactory implementation.

* - any other parameters are treated as framework constants.

Configuring web.xml for the framework is a matter of adding a filter and filter-mapping.

```
<web-app id="WebApp_9" version="2.4"
        xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
   <filter>
      <filter-name>struts2</filter-name>
      <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
      <init-param>
         <param-name>actionPackages</param-name>
         <param-value>com.mycompany.myapp.actions</param-value>
      </init-param>
   </filter>
```

```
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <!-- ... -->
</web-app>
```

10.1.2. Struts2.xml

The core configuration file for the framework is the default (`struts.xml`) file and should reside on the classpath of the webapp (generally `/WEB-INF/classes`).

- The default file may include other configuration files as needed.
- A `struts-plugin.xml` file can be placed in a JAR and automatically plugged into an application, so that modules can be self-contained and automatically configured.
    - In the case of Freemarker and Velocity modules, the templates can also be loaded from the classpath, so the entire module can be plugged in as a single JAR.

Struts2.xml example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
        <constant name="struts.devMode" value="true" />
        <package name="basicstruts2" extends="struts-default">
        <!-- setup the default-stack exception mapping interceptor
                so that any exceptions not caught by this application
                will be logged and then handled by the global exception
                mapping -->
                <interceptors>
```

```
            <interceptor-stack name="appDefault">
             <interceptor-ref name="defaultStack">
                  <param name="exception.logEnabled">true</param>
                  <param name="exception.logLevel">ERROR</param>
                  </interceptor-ref>
          </interceptor-stack>
            </interceptors>
            <default-interceptor-ref name="appDefault" />

      <global-results>
              <result name="error">/error.jsp</result>
              <result name="securityerror">/securityerror.jsp</result>
          </global-results>
          <global-exception-mappings>
              <exception-mapping
exception="org.apache.struts.register.exceptions.exceptions.SecurityBreachException"
result="securityerror" />
              <exception-mapping exception="java.lang.Exception" result="error" />
              </global-exception-mappings>
               <action name="causesecurityexception"
class="org.apache.struts.register.action.Register" method="throwSecurityException">
              <result>/register.jsp</result>
      </action>
      <action name="causeexception" class="org.apache.struts.register.action.Register"
method="throwException">
              <result>/register.jsp</result>
```

```
    </action>
    <action name="causenullpointerexception"
class="org.apache.struts.register.action.Register" method="throwNullPointerException">
        <result>/register.jsp</result>
    </action>
    <!-- If no class attribute is specified the framework will assume success and
    render the result index.jsp -->
    <!-- If no name value for the result node is specified the success value is the default -->
        <action name="index">
            <result>/index.jsp</result>
        </action>

        <!-- If the URL is hello.action the call the execute method of class HelloWorldAction.
        If the result returned by the execute method is success render the HelloWorld.jsp -->
        <action name="hello" class="org.apache.struts.helloworld.action.HelloWorldAction"
method="execute">
            <result name="success">/HelloWorld.jsp</result>
        </action>
  <action name="register" class="org.apache.struts.register.action.Register" method="execute">
    <result name="success">/thankyou.jsp</result>
  </action>
    </package>
</struts>
```

10. 2 Struts 2 ValueStack

Now lets understand how the UI tags work. In Struts 2 *ValueStack* is the place where the data associated with processing of the request is stored. So all the form properties will be stored on

the*ValueStack*. The *name* attribute of the UI tag is the one which links the property on the *ValueStack*.

<s:textfield name="userName" label="User Name" value="Eswar"/>

10.3 Textfield and Password Tags

Now lets see each UI tag in detail. The *textfiled* tag is used to create a textfield and *password* tag is used to create a password field. These tags are simple and uses only the common attributes discussed before.

<s:textfield name="userName" label="User Name" />

<s:password name="password" label="Password" />

10.4 Submit Tag

The *submit* tag is used to create the Submit button

<s:submit />

11 HTML/CSS

11.1 HTML

11.1.1 IDs and Classes

IDs and classes should contain lowercase letters and words should be separated with a hyphen. e.g.:

<td class="alt-row"></td>

As a rule, use classes for everything.

11.1.2 Default Skeleton

The following markup should be used as the base skeleton for most HTML documents. Variations are allowed, but the IDs should be preserved if possible:

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

```
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="description" content="">


    <title></title>


  </head>
  <body class="page-name">
    <div class="wrapper">
      <div class="header">
      </div> <!-- /.header -->


      <ul class="nav">
      </ul>


      <div class="content">
      </div> <!-- /.content -->


      <div class="footer">
      </div> <!-- /.footer -->
    </div> <!-- /.wrapper
  </body>
</html>
```

11.1.3 Best Practices

Use clear and precise names for IDs and classes. Choose names according to semantic meaning rather than presentation. Avoid using unnecessary classes and wrapper elements.

Validate HTML. A validator can often help a developer track down styling or scripting bugs. Valid HTML also increases the likelihood that a page will be displayed correctly in current and future browsers.

11.2 CSS

11.2.1 Style Block

Each style block should adhere to the following format:

```
selector { property: value; }

selector-2 {

  property: value;

  property: value;

}
```

Each selector should: be on it's own line be followed by a space end in an opening brace on the same line be closed with a brace on a new line without indentation.

Each property-value pair should: be on its own line be indented with two spaces contain a space after the colon Stylesheet Organization

11.2.2 Document Head

```
/*********************************************************************************

    Rain CSS Coding Standards outlined at http://standards.mediarain.com/html-css


    Color Reference:

        red: #FF0000

        blue: #0000FF

        gray: #333333


*********************************************************************************/
```

Each main stylesheet should contain a document head. The head should contain a reference to this coding standards document (http://standards.mediarain.com/html-css). This reference will help any new or outside developers be aware of the coding guidelines they should follow.

Optionally, the document head may contain a color reference. A color reference is an index of colors commonly used in the design. This reference makes it easy to identify specific color values without a color picker. It is also useful when a color is updated, since a quick find/replace can be completed.

A description, table of contents, or other meta data may be included in this head as deemed appropriate.

11.2.3 Sections

CSS style blocks should be grouped by section and ordered according to the markup in the HTML document. Common sections include the following:

Global

Type

Forms

Header

Navigation

Content

Footer

Page- or layout-specific sections may be added. Sections should be denoted with comments which use the following format:

```
/* ************************************

 *

 * @Section: Global

 *

************************************ */
```

Two empty lines should precede each section comment. The '@' symbol is used to quickly traverse the stylesheet using a text editor's "Find Next" feature.

11.2.4 Best Practices

Use a global reset. A global reset helps create more consistent presentation across browsers.

Use sprites for all rollover/active states. CSS sprites prevent unwanted image flicker on rollover. CSS sprites also reduce the total number of HTTP requests.

Use as few browser-specific styles as possible. If needed, browser-specific stylesheets or page classes (http://paulirish.com/2008/conditional-stylesheets-vs-css-hacks-answer-neither/) should be used instead of putting CSS hacks in the main stylesheet.

11.2.5 Validate CSS.

File Naming and Organization

The following example file structure should be used for static resources (framework-specific conventions take precedence):

/css

    /reset.css

    /main.css

    /ie.css

/img

    /btn-submit.png

    /btn-submit-SOURCE.psd

    /es

        /btn-submit.png

/media

/js

/files

Filenames

Filenames should contain lowercase letters and words should be separated with a hyphen. The hyphen word separator (as opposed to an underscore or camelcase) is considered good practice for SEO reasons.

11.2.6 CSS Stylesheets

CSS stylesheets should be contained in a directory named 'css'. This directory will often contain a global reset stylesheet, named reset.css and a main stylesheet, appropriately named main.css. Any Internet Explorer-specific stylesheets should be named ie.css or ie7.css, etc. Other stylesheets may be added depending on the size and breadth of the website.

Images

Images should be contained in a directory named 'img'. In general, images should not be divided into subdirectories.

Images of common types (buttons, icons, etc.) should contain a prefix with the type abbreviation. e.g. btn-submit.png

Source files (Photoshop, Fireworks, etc.) for sprite images should always be saved with the web version. Each source filename should match the corresponding web version and contain the suffix "-SOURCE".

Locale-specific images should be contained in a subdirectory named after the locale abbreviation. e.g. /img/es/

Media Files

Flash, Silverlight, or other media files should be contained in a directory named 'media'.

Javascript Files

Javascript files should be contained in a directory named 'js'.

Other Files

All other downloadable documents, such as PDFs or Word docs, should be contained in a directory named 'files'.

• • •