TEAM 1 Coding Standards

Table of Contents

1 About the TEAM 1 Coding Standards

The Team 1 Coding Standards were written by Team 1. The purpose is to make the system clean, consistent, and easy to install. It focuses on programs written in JAVA, but many of the rules and principles are useful even if we write in another programming language. The rules often state reasons for writing in a certain way.

The source repository for this document can be found at **https://savannah.gnu.org/projects/gnustandards.**These standards cover the minimum of what is important when writing a document. Likely, the need for additional standards will come up. This release of the Team 1 Coding Standards was last updated December 15,2022.

2 Purpose of Coding Standards and Best Practices

To develop reliable and maintainable applications, we must follow coding standards and best practices. The naming conventions, coding standards and best practices described in this document are compiled from our own experience and by referring to various Microsoft and non Microsoft guidelines.

2.1 Architecture

1. Always having a data layer class which performs all the database related tasks is helpful. This will help us support or migrate to another database back end easily.

2. Usage of try-catch in the data layer to catch all database exceptions. This exception handler should record all exceptions from the database.

3. Separate your application into multiple assemblies. Group all independent utility classes into a separate class library. All your database related files can be in another class library.

2.2 Naming Conventions and Standards

Note :

The terms Pascal Casing and Camel Casing are used throughout this document.

Pascal Casing - First character of all words are Upper Case and other characters are lower case.

Example: BackColor

Camel Casing - First character of all words, except the first word are Upper Case and other characters are lower case.

Example: backColor

1. Used Pascal casing for Class names

```
public class User
{
        ...
}
```

2. Used Camel casing for Method names

```
Public String  created(Driving License)
{
        ...
}
```

3. Used Camel casing for variables and method parameters

```
int dlId;
Public String  created(Driving License)
{
        Loger.info("Driving License saved with ID" +id);
        ...
}
```

4. Used Pascal Casing for interfaces (Example:`DLService`)

5. Used Meaningful, descriptive words to name variables. Do not use abbreviations.

Good:

String name
long mobNo

6. No usage of single character variable names like i, n, s etc. Use names like index, temp

One exception in this case would be variables used for iterations in loops:

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

7. No usage of underscores (_) for local variable names.

8. Do not use variable names that resemble keywords.

9. File name should match with class name.

For example, for the class HelloWorld, the file name should be helloworld.cs (or, helloworld.vb)

10. Used Pascal Case for file names.

2.3 Indentation and Spacing

1. Used TAB for indentation. No usage of SPACES.

2. Comments should be in the same level as the code (use the same level of indentation).

3. Curly braces ({}) should be in the

4. same level as the code outside the braces.

```
if ( ... )
{
        // Do something
        // ...

        return false;
}
```

```
public int loginValidator(String username) {

    if (adminRepo.findByUsername(username) != null) {
        return 1;
    }
    return 0;
}
```

5. Nesting the if-else blocks of code to have clear visibility of function.

```
//create a new dl
@Override
@Transactional(propagation=Propagation.REQUIRED)
public String createDL(DrivingLicense dl)  {
    if(checkIfDLExist(dl.getDlno())!=null ) {
        return "DL already Exists";
    }
    else if(checkIfPhoneNumberExist(dl.getMobNo())!=null){
        return "Phone Number already Exists";
    }
    logger.info("saving new dl");
    DrivingLicense d=dlRepo.save(dl);
    logger.debug("Driving License saved with id ="+d.getId()+" is "+d);
    return "Sucessfully Added DL Information";

}
```

6. Used #region to group related pieces of code together. If you use proper grouping using #region, the page should like this when all definitions are collapsed.

```
using System;

namespace EmployeeManagement
{
    public class Employee
    {
        Private Member Variables

        Private Properties

        Private Methods

        Constructors

        Public Properties

        Public Methods
    }
}
```

7. Kept private member variables, properties and methods in the top of the file and public members in the bottom.

2.4 **Comments**

Good and meaningful comments make code more maintainable. However,

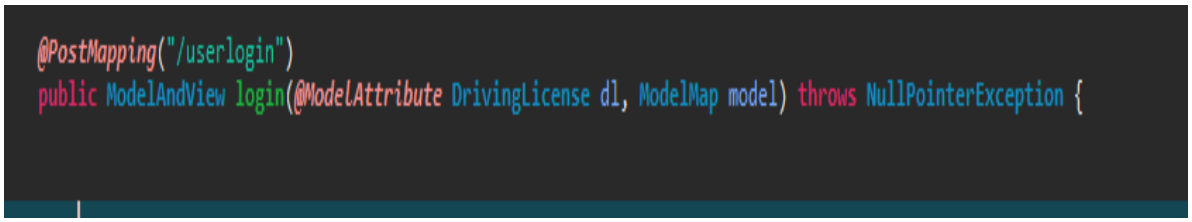
1. Used **//** or **///** for comments. Avoid using **/* … */**

2. Writing comments wherever required. Good readable code required very less comments. All variables and method names were meaningful, that would make the code very readable and will not need many comments.

2.5 Exception Handling

1. In case of exceptions, used a friendly message to display it to the user, but not the actual error with all possible details about the error, including the time it occurred, method and class name etc.

2. Always catch only the specific exception, not generic exception.



3. No need to catch the general exception in all your methods. Leave it open and let the application crash. This will help you find most of the errors during development cycle. You can have an application level (thread level) error handler where you can handle all general exceptions. In case of an 'unexpected general error', this error handler should catch the exception and should log the error in addition to giving a friendly message to the user before closing the application, or allowing the user to 'ignore and proceed'.

2.6 Best Practices for front-end Coding

1. Declaring a DOCTYPE
The DOCTYPE declaration should be in the first line of HTML. Actually, it activates the standard mode in all browser. It is recommended that you use the HTML doctype:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
        <meta charset="UTF-8">

        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
        <title>User</title>
```

2. Closing the tags
Leaving some tags open is simply a bad practice. Only self-closing tags (
,

, ,,, etc) are valid. Normal elements can never have self-closing tags.

3. Explanation of div that are closing

If we view most of the website source at the very bottom of the page an almost endless list of closing tags. Without proper code organization, it can be messy. Using indentation and comment for every div is a good practice.

Example:--<!– #header →

4. Include external CSS inside the HEAD tag
Style sheets can be placed anywhere but the HTML specification recommends that they be placed within the document HEAD tag. The primary benefit is that pages will load faster.

5.Considered placing JavaScript at the bottom
When loading a script, the browser cannot continue until the entire file has been loaded. If we have JavaScript files in order to add functionality, we placed those files at the bottom, just before the closing body tag. This is a good performance practice and the results are quite noticeable.

```
            <script type="text/javascript">
        $('.date').datepicker({
           format: 'dd-mm-yyyy'
         });
    </script>
```

6. Usage of lowercase in tags

It is a good practice to keep markup lower-case. The capitalizing markup will work and will probably not affect how your web pages are rendered, but it does affect code readability. We should keep it simple.

7. Keeping the syntax organized

When pages will grow, managing HTML can be hard. There are some quick rules that can help us to keep our syntax clean and organized. These include the following:

- Indent nested elements
- Use double quotes, not single or completely omitted quotes
- Omit the values on Boolean attributes

8. Using practical ID and classes names and values
We should only give elements an ID attribute if they are unique. Classes can be applied to multiple elements that share the same style properties. It is always preferable to name something ID or class, by the nature of what it is rather than by what it looks like. Class names should be all lowercase and use hyphen delimiters.

```
<form action="/admin" method="post">
  <nav class="navbar fixed-top navbar-dark bg-dark">
    <div class="topnav">
      <a href="/home">Home</a>
      <a href="/about">About</a>
      <a href="/faq">FAQ</a>
      <a href="/aboutus">About Us</a>
      <a href="/home">Logout</a>

    </div>
  </nav>
```

9. Writing CSS using multiple lines and spaces
It is important to place each selector and declaration on a new line. That will make the code easy to read and edit.

10. Avoiding units on zero values

One way to easily cut down on the amount of CSS we write is to remove the unit from any zero value. A zero will always be a zero.

```
position:relative;
top:5%;
width:100%;
text-align: top;
font-size: 60px;
margin: 0;
color: rgb(255, 255, 255);
font-weight: bolder;
font-style:serif;
```

11. Modularizing styles for reuse

CSS is built to allow styles to be reused, specifically with the use of classes. For this reason, styles assigned to a class should be modular and available to share across elements as necessary.

12. Using multiple stylesheets, but be aware of them expanding beyond control

Depending on the complexity of the design and the size of the site, sometimes it's easier to make

13. Checking in cross-browser while developing.

One of the biggest mistake made when developing HTML, CSS, and javascript, was not to test pages on multiple browsers while developing them. Generally, we used to write all code and just view in one of the browsers to see how it was rendered.This becomes the one of best practice to follow.