

## Spiceland-Project Coding Standards

---

# Table of Contents

1	About the Spiceland-Project Coding Standards.....
2	Purpose of Coding standards and best practices .....
2.1	Naming Convention and standards.....
2.2	Indentation and spacing.....
2.3	Comments .....
2.4	Exception Handling .....
2.5	Best practices for front-end design.....

## **About the Spiceland Project Coding Standards**

The Spiceland project Coding Standards were written by Team 5. 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.

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 Spiceland project Coding Standards was last updated December 16, 2022.

### **1 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.

## 1.1 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 Products getProduct(int id)
{
    ...
}
```

3. Used Camel casing for variables and method parameters

```
int cartId;
public Products getProduct(int id)
{
    Logger.info("fetching product with ID" +id);
    ...
}
```

4. Used Pascal Casing for interfaces ( Example: **ProductService** )

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

Good:

```
string address  
double price
```

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.

## 1.2 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).

```
// Details of the product by id.  
@Override  
public Products getProduct(int id) {  
    logger.info("Fetching product with id " + id);  
    Products product = productRepository.findById(id);  
    logger.debug("Product with id = " + id + " is " + product);  
    return product;  
}
```

3. Curly braces ( {} ) should be in the same level as the code outside the braces.

```

if ( ... )
{
    // Do something
    // ...
    return false;
}

```

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

```

/*
 * Once the delivery person login, It will navigate to the deliverhome.
 */
@GetMapping("/delivery/deliverhome/{id}")
public String deliverhome(@PathVariable("id") int id, ModelMap model) {
    model.addAttribute("user", service.getuser(id));
    List<Orders> orders = orderService.findAll();
    ArrayList<Integer> customerIds = new ArrayList<>();
    if (orders.size() > 0) {
        for (Orders order : orders) {
            if (customerIds.contains(order.getUser_id())) {
            }
            else {
                customerIds.add(order.getUser_id());
            }
        }
    }
}

```

5. 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
    {
        #region Private Member Variables
        #endregion
        #region Private Properties
        #endregion
        #region Private Methods
        #endregion
        #region Constructors
        #endregion
        #region Public Properties
        #endregion
        #region Public Methods
        #endregion
    }
}

```

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

## 1.3 Comments

Good and meaningful comments make code more maintainable. However,

1. Used `//` or `/* ... */` for comments.
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.

## 1.4 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, but generic exception was in-use to know for the exposure to errors during the development-cycle.

```
@PostMapping("/delivery/getOrders/{userid}/{orderid}/{customerid}")
public String DeleteProduct(Model model, @PathVariable("userid") int userid, @PathVariable("orderid") int id,
    @PathVariable("customerid") int customerid) throws AuthenticationException{
```

3. No need to catch the general exception in all the methods as stated with the above reason. This will help us find most of the errors during development cycle. We can have an application level (thread level) error handler where we 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'.

## 1.5 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:

```
<!doctype html>
```

### 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

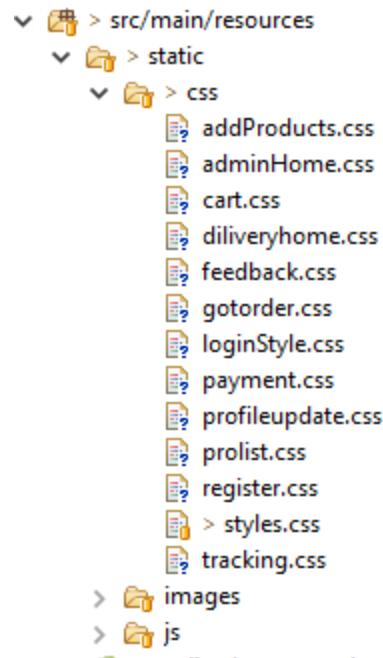
If we view most of the website source at the very bottom of the page an almost endless list of closing tags. Without proper code organization, it can be messy. Using indentation and comment for every div is a good practice.

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
  src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
  integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
  crossorigin="anonymous">
</script>
<script
  src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"
  integrity="sha384-wfSDF2E50Y2D1uUdJ003uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6"
  crossorigin="anonymous">
</script>
</form>
</body>
</html>
```

#### 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.

```
<body class="bg-dark">
<form action="/delivery/deliverhome" method="get">
  <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <div class="container-fluid">
      <a class="navbar-brand" href="#">
```

## 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.

```
display: inline-block;
font-family: -apple-system,sans-serif;
font-size: 1.2rem;
font-weight: 700;
letter-spacing: -.24px;
margin: 0;
outline: none;
padding: 1rem 1.3rem;
quotes: auto;
```

## 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. 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

## 12.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.