2024 Capstone Coding Standards

This is the list of standards that are expected to be followed when doing project work for the 2024 Capstone Friday morning class.

Summaries

Summaries simply provide a summary of what something is, what it does, and the dates of when it was created and changed. Using three forward slashes (///) Visual Studio should autocomplete a summary comment for you. Summaries should be above **CLASSES**, **METHODS**, **STORED PROCEDURES, DATABASE TABLES** and **XAML DOCUMENTS**. (For XAML the ///<Summary> commenting style will be replaced with HTML commenting style. This will also be included for .CSHTML files.) The comment should be formatted like this:

```
/// <summary>
/// Creator: FirstName LastName
/// Created: MM/DD/YYYY
/// Summary: Summary
/// Last Updated By: FirstName LastName
/// Last Updated: MM/DD/YYY
/// What Was Changed: Whatever was changed in the most recent version
/// </summary>
```

The initial summary and change sections should explain exactly what the class or method accomplishes, or what was changed. If the object was just created, you must still fill out the updated sections with your name, and write “Initial Creation” in the “What Was Changed” section. Also update the summary to include the new functionality of the object. Here is an example of a summary on a simple method:

```
/// <summary>
/// Creator: Mitchell Stirmel
/// Created: 1/18/2024
/// Summary: This method returns the number input in the method signature.
/// Last Updated By: Mitchell Stirmel
/// Last Updated: 01/18/2024
/// What Was Changed: Initial Creation
/// </summary>
0 references
public int ReturnsANumber(int number)
{
    return number;
}
```

Here is this example, but after a change has been made to the method:

```
/// <summary>
/// Creator: Mitchell Stirmel
/// Created: 1/18/2024
/// Summary: This method returns the absolute value of the
/// number input in the method signature.
/// Last Updated By: Mitchell Stirmel
/// Last Updated: 01/18/2024
/// What Was Changed: The method now returns the absolute value
/// of the input number instead of the input number.
/// </summary>
0 references
public int ReturnsANumber(int number)
{
    return Math.Abs(number);
}
```

Interfaces

Interfaces should start with an “I”, and follow a capitalized format. This means all words in the name have the first letter capitalized, and only words should exist in the name. The name must explain what the interface is. Underscores should not be included in the method name entirely. For example, an interface for segway logic should look like this:

```
0 references
public interface ISegwayLogic
{
}
```

Classes

Classes should follow a capitalized format. This means all words in the name have the first letter capitalized, and only words should exist in the name. The name must explain what the class is.Underscores should not be included in the method name entirely. For example, a class for segway logic should look like this:

```
public class SegwayLogic
{

}
```

Methods

Methods must follow a capitalized format. This means all words in the name have the first letter capitalized, and only words should exist in the name. The name must explain what the method does. Variables in the method signature must be camelCased. Underscores should not be included in the method name entirely. Here is an example of a method that adds two numbers together:

```
0 references
public int AddTwoNumbers(int num1, int numTwo)
{
    return num1 + numTwo;
}
```

Constructors

The name must explain what the constructor does. Variables in the constructor signature must be camelCased. Here is an example of a constructor

```
0 references
public SegwayLogic(int index, string name)
{

}
```
:

Class Level Variables

Class level variables must be camelCased, and start with an underscore. These variables must be defined in the beginning of the class, before any constructors or methods in the class. Here is an example:

```
1 reference
public class SegwayLogic
{
    private int _index = 0;

    private string _segwayName = string.Empty;

    0 references
    public string _weGetThis { get; set; }
```

Method Level Variables

Method level variables must be camelCased, and should **NOT** start with an underscore. Here is an example:

```
0 references
public int ReturnStaticNum()
{
    int staticNum = 1;
    return staticNum;
}
```

Logic Statements

ALWAYS use brackets, even if not necessary, such as single line if statements.

```
if (false)
    return staticNum; // Not correct

if (false)
{
    return staticNum; // Correct
}
```

ALWAYS use double ampersands or double pipes for AND and OR statements (&&, ||).

ALWAYS do Object.Equals for object comparisons instead of normal logic statements. This includes strings, as strings are not a primitive type.

```
if (str == "123")
{

}

if (str.Equals("123"))
{

}
```

Commenting

Commenting does **NOT** need to be done where the code is safely assumed to be **KNOWN** by the class. Examples of this would include information that was taught in previous classes. Here is an example:

```
SqlConnection conn = new SqlConnection(_configuration["ConnectionStrings:Database"]);
var cmd = new SqlCommand("sp_deactivate_comment", conn);
cmd.CommandType = CommandType.StoredProcedure;

cmd.Parameters.Add("@CommentID", SqlDbType.Int);

cmd.Parameters["@CommentID"].Value = commentId;
```

Code like this was necessary in order to pass .NET II for a data driven application. This would not need to be commented.

Commenting **MUST** be done wherever there is code or logic that is safely assumed to **NOT** be known by anyone else in the class. This would include any code that was not taught in class by Jim, or code that is not common. An example is asynchronous operations.

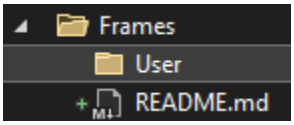
Databasing

For database scripts, we will use brackets around the variables, as well as adding the print ‘’ print ‘*** ***’ for tables and stored procedures. Constraints will be added at the end of the code. This is an example of what a proper database code block would look like.

```
/*
<Summary>
Creator:            Liam Easton
Created:            01/19/2024
Summary:            Creation for Location table
Last Updated By:    Liam Easton
Last Updated:       01/19/2024
What Was Changed:   Initial Creation
<Summary>
*/
print '' print '*** creating Location Table ***'
GO
CREATE TABLE [dbo].[Location]
(
    [LocationID]      [int] IDENTITY(100000,1)   NOT NULL,
    [LocationName]       [nvarchar](50)              NOT NULL,
    [Address]          [nvarchar](50)             NOT NULL,
    [City]               [nvarchar](100)           NOT NULL,
    [State]            [nvarchar](100)           NOT NULL,
    [ZipCode]          [int]                          NOT NULL
    CONSTRAINT [pk_LocationID] PRIMARY KEY ([LocationID]),
)
GO
```

Framing

Framing will be done a specific way. To prevent people from working on the same XAML file, but also prevent people from making multiple windows, each person will create a frame for their story, if need be. Frames will be put in the Frames folder, under the specific folder for that specific frame type. For example, if the story requires a frame that deals with users, it will go in a “Users” folder. Here is an example.



Here is also an example of XAML commenting. Note we include the <summary> aspects.

```
<!--
    <Summary>
    Creator: Mitchell Stirmel
    Created: 01/26/2024
    Summary: This is the main page that will be used to store the individal frames.
    To load a frame into this window, use the frame named FrameLoad here.
    Last Updated By: Mitchell Stirmel
    Last Updated: 01/26/2024
    What Was Changed: Initial Creation was done by autogeneration.
    Added grid row and column definitions and created the frame space to be used.
    <Summary>
-->

<Window x:Class="WpfPresentation.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```