



TEKNOWLEDGE
Fusion of Knowledge & Technology

Coding Standards & Best Practices

For TeKnowledge Software
Engineering

v1.1 | 2022-10-28

PPathum Bandara Premaratne

TeKnowledge Shared Services (Pvt) Ltd.

Table of Contents

1. Introduction.....	3
2. Purpose of Coding Standards and Best Practices	3
3. Naming Conventions and Standards.....	4
4. Indentation and Spacing	8
5. Good Programming Practices.....	11
6. Architecture.....	18
7. ASP.NET.....	19
8. Comments	19
9. Exception Handling	20
10. Cascading Style Sheets (CSS)	22
11. JavaScript.....	30

Document Info

Author	PAthum Bandara Premaratne
Date	2022-10-28
Version	1
Revision	1.1

Revision History

Author	Date	Reason	Revision
PAthum Bandara Premaratne	2022/10/28	Replacing the company name	1.1

1. Introduction

Anybody can write code. With a few months of programming experience, you can write 'working applications'. Making it work is easy, but doing it the right way requires more work, than just making it work.

Believe it, majority of the programmers write 'working code', but not 'good code'. Writing 'good code' is an art, you must learn and practice it.

Everyone may have different definitions for the term 'good code'. In my definition, the following are the characteristics of good code.

- Reliable
- Maintainable
- Efficient

Most of the developers are inclined towards writing code for higher performance, compromising reliability and maintainability. But considering the long-term ROI (Return On Investment), efficiency and performance comes below reliability and maintainability. If your code is not reliable and maintainable, you (and your company) will be spending lot of time to identify issues, trying to understand code etc. throughout the life of your application.

2. Purpose of Coding Standards and Best Practices

To develop reliable and maintainable applications, you 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.

There are several standards exists in the programming industry. None of them are wrong or bad and you may follow any of them. What is more important is, selecting one standard approach and ensuring that everyone is following it.

3. 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. Use Pascal casing for Class names

```
public class HelloWorld
{
    ...
}
```

2. Use Pascal casing for Method names

```
void SayHello(string Name)
{
    ...
}
```

3. Use Pascal casing for variables and method parameters

```
int TotalCount = 0;

void SayHello(string Name)
{
    string FullMessage = String.Format("Hello {0}" , Name);
    ...
}
```

4. Use the prefix "I" with Camel Casing for interfaces (Example: **IEntity**)
5. Do not use Hungarian notation to name variables except for member variables.

In earlier days most of the programmers liked it - having the data type as a prefix for the variable name and using m_ as prefix for member variables.

E.g.:

```
string m_sName;  
int nAge;
```

However, in .NET coding standards, this is not recommended. Usage of data type and m_ to represent member variables should not be used.

Some programmers still prefer to use the prefix **m_** to represent member variables, since there is no other easy way for them to identify a member variable. :D

```
//Use as below  
string _Name = String.Empty;
```

6. Use Meaningful, descriptive words to name variables. Do not use shorten names.

Good:

```
string Address  
int Salary
```

Not Good:

```
string Nam  
string addr  
int Sal
```

7. Do not use 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++ )  
{  
    ...  
}
```

If the variable is used only as a counter for iteration and is not used anywhere else in the loop, many people still like to use a single char variable (**i**) instead of inventing a different suitable name.

8. Do not use underscores (**_**) for local variable names to separate words.
9. All member variables must be prefixed with underscore (**_**) so that they can be identified from other local variables.
10. Do not use variable names that resemble keywords.

11. Prefix **boolean** variables, properties and methods with “**Is**” or similar prefixes.

E.g.: **private bool _IsFinished**

12. Namespace names should follow the standard pattern

<CompanyName>.<ProductName>.<Level1Module>.<Level2Module>

E.g.: **TKSE_PostOffice.WMS.Business, TKSE_PostOffice.Leonis.Data.Sync**

13. Use appropriate prefix for the UI elements so that you can identify them from the rest of the variables.

Use appropriate prefix for each of the ui element. A brief list is given below. Since .NET has given several controls, you may have to arrive at a complete list of standard prefixes for each of the controls (including third party controls) you are using.

Control	Prefix
Label	Lbl
TextBox	Txt
DataGrid	Grd
Button	Cmd
ImageButton	Cmd
Hyperlink	Lnk
DropDownList	Drp
ListBox	Lsb
DataList	Lst
Repeater	Rpt
Checkbox	Chk
CheckBoxList	Chl
RadioButton	Rdo

RadioButtonList	Rdl
Image	Img
Panel	Pnl
PlaceHolder	Ph
Table	Tbl
Validators	Val

14. File name should match with class name.

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

15. Use Pascal Case for file names.

E.g.: [User.cs](#), [CustomerAddress.cs](#)

4. Indentation and Spacing

1. Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.
2. Comments should be in the same level as the code (use the same level of indentation).

Good:

```
// Format a Message and display
string FullMessage = String.Format("Hello {0}", Name); // or $"Hello
{Name}" in C# 6+
DateTime CurrentTime = DateTime.Now;
string Message = String.Format("{0}, the time is : {1}", FullMessage,
CurrentTime.ToShortTimeString());
MessageBox.Show ( Message );
```

Not Good:

```
// Format a Message and display
string FullMessage = "Hello " + name;
DateTime CurrentTime = DateTime.Now;
string Message = String.Format("{0}, the time is : {1}", FullMessage,
CurrentTime.ToShortTimeString());
MessageBox.Show ( Message );
```

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

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

4. Use one blank line to separate logical groups of code.

Good:

```
bool SayHello ( string Name )
{
    string FullMessage = String.Format("Hello ", Name);
    DateTime CurrentTime = DateTime.Now;

    string Message = String.Format("{0}, the time is : {1}", FullMessage,
CurrentTime.ToShortTimeString());

    MessageBox.Show ( Message );

    if ( ... )
    {
```

```

        // Do something
        return false;
    }

    return true;
}

```

Not Good:

```

bool SayHello (string Name)
{
    string FullMessage = "Hello " + name;
    DateTime CurrentTime = DateTime.Now;
    string Message = FullMessage + ", the time is : " +
    CurrentTime.ToShortTimeString();
    MessageBox.Show ( Message );
    if ( ... )
    {
        // Do something
        // ...
        return false;
    }
    return true;
}

```

5. There should be one and only one single blank line between each method inside the class.
6. The curly braces should be on a separate line and not in the same line as **if**, **for** etc.

Good:

```

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

```

Not Good:

```

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

```

7. Use a single space before and after each operator.

Good:

```

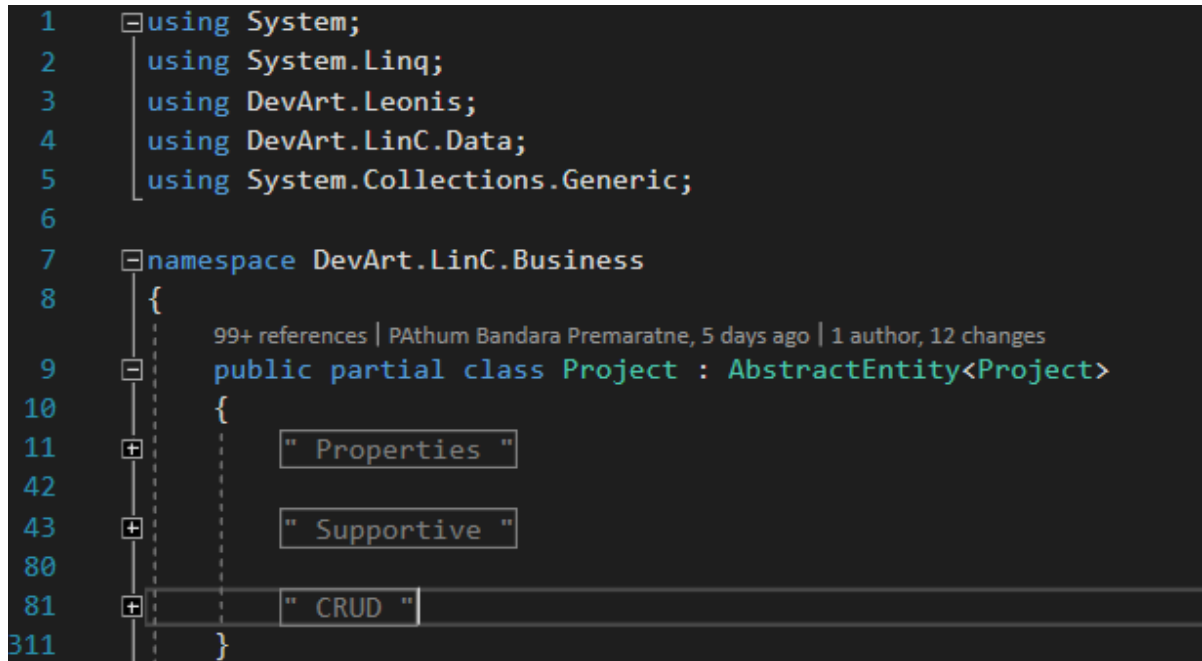
if (ShowResult == true)
    for (int I = 0; I < 10; I++)
        //Do something

```

Not Good:

```
if( showResult==true )  
    for(int i=0;i<10;i++)  
        //Do something
```

8. Use `#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.



```
1  using System;  
2  using System.Linq;  
3  using DevArt.Leonis;  
4  using DevArt.LinC.Data;  
5  using System.Collections.Generic;  
6  
7  namespace DevArt.LinC.Business  
8  {  
9      99+ references | P Athum Bandara Premaratne, 5 days ago | 1 author, 12 changes  
10     public partial class Project : AbstractEntity<Project>  
11     {  
12         " Properties "  
42  
43         " Supportive "  
80  
81         " CRUD "  
311     }  
}
```

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

5. Good Programming Practices

1. Avoid writing very long methods. A method should typically have 1~25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.
2. Method name should tell what it does. Do not use mis-leading names. If the method name is obvious, there is no need of documentation explaining what the method does.

Good:

```
void SavePhoneNo (string PhoneNo)
{
    // Save the phone number.
}
```

Not Good:

```
// This method will save the phone number.
void SaveDetails (string PhoneNo)
{
    // Save the phone number.
}
```

3. A method should do only 'one job'. Do not combine more than one job in a single method, even if those jobs are very small.

Good:

```
// Save the Address.
SaveAddress ( Address );

// Send an Email to the supervisor to inform that the Address is updated.
SendEmail ( Address, Email );

void SaveAddress ( string Address )
{
    // Save the Address.
}

void SendEmail ( string Address, string Email )
{
    // Send an Email to inform the supervisor that the Address is
    changed.
}
```

Not Good:

```
// Save Address and send an Email to the supervisor to inform that
// the Address is updated.
```

```

SaveAddress ( Address, Email );

void SaveAddress ( string Address, string Email )
{
    // Job 1.
    // Save the Address.
    // ...

    // Job 2.
    // Send an Email to inform the supervisor that the Address is changed.
    // ...
}

```

4. Use the C# or VB.NET specific types (aliases), rather than the types defined in System namespace.

```

int Age; //(not Int16)
string Name; // (not String)
object ContactInfo; //(not Object)

```

Some developers prefer to use types in Common Type System than language specific aliases.

5. Always watch for unexpected values. For example, if you are using a parameter with 2 possible values, never assume that if one is not matching then the only possibility is the other value.

Good:

```

If ( MemberType == MemberTypes.Registered )
    // Registered user... do something...
else if ( MemberType == MemberTypes.Guest )
    // Guest user... do something...
else
{
    // Un expected user type. Throw an exception
    throw new ApplicationException ("Un expected value " +
    MemberType.ToString() + ".")
    // If we introduce a new user type in future, we can easily
    find the problem here.
}

```

Not Good:

```

If ( MemberType == MemberTypes.Registered )
    // Registered user... do something...
else
{

```

```
// Guest user... do something...
```

```
// If we introduce another user type in future, this code will fail and
will not be noticed.
}
```

6. Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code.

However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.

7. Do not hardcode strings. Use resource files.
8. Convert strings to lowercase or upper case before comparing. This will ensure the string will match even if the string being compared has a different case.

```
if ( Name.ToLower() == "john" )
{
    //...
}
```

9. Use String.Empty instead of ""

Good:

```
If ( Name == String.Empty )
{
    // do something
}
```

Not Good:

```
If ( Name == "" )
{
    // do something
}
```

10. Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.
11. Use **enum** wherever required. Do not use numbers or strings to indicate discrete values.

Good:

```
enum MailType : int
```

```

{
    Html, PlainText, Attachment
}

void SendMail (string Message, MailType MailType)
{
    switch ( MailType)
    {
        case MailType.Html:
            // Do something
            break;
        case MailType.PlainText:
            // Do something
            break;
        case MailType.Attachment:
            // Do something
            break;
        default:
            // Do something
            break;
    }
}

```

Not Good:

```

void SendMail (string Message, string MailType)
{
    switch ( MailType)
    {
        case "Html":
            // Do something
            break;
        case "PlainText":
            // Do something
            break;
        case "Attachment":
            // Do something
            break;
        default:
            // Do something
            break;
    }
}

```

12. Do not make the member variables public or protected. Keep them private and expose public/protected Properties.

13. The event handler should not contain the code to perform the required action. Rather call another method from the event handler.
14. Do not programmatically click a button to execute the same action you have written in the button click event. Rather, call the same method which is called by the button click event handler.
15. Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.
16. Never assume that your code will run from drive "C:". You may never know; some users may run it from network or from a "Z:".
17. In the application start up, do some kind of "self-check" and ensure all required files and dependencies are available in the expected locations. Check for database connection in startup, if required. Give a friendly Message to the user in case of any problems.
18. If the required configuration file is not found, application should be able to create one with default values.
19. If a wrong value found in the configuration file, application should throw an error or give a Message and also should tell the user what are the correct values.
20. Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."
21. When displaying error messages, in addition to telling what is wrong, the Message should also tell what should the user do to solve the problem. Instead of Message like "Failed to update database.", suggest what should the user do: "Failed to update database. Please make sure the login id and password are correct."
22. Show short and friendly Message to the user. But log the actual error with all possible information. This will help a lot in diagnosing problems.
23. Do not have more than one class in a single file.
24. Have your own templates for each of the file types in Visual Studio. You can include your company name, copy right information etc. in the template. You can view or edit the Visual Studio file templates in the folder <C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\ItemTemplatesCache\CSharp\1033>. (This folder has the templates for C#, but you can easily find the corresponding folders or any other language)
25. Avoid having very large files. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Split them logically into two or more classes.

26. Avoid public methods and properties, unless they really need to be accessed from outside the class. Use “internal” if they are accessed only within the same assembly.
27. Avoid passing too many parameters to a method. If you have more than 4~5 parameters, it is a good candidate to define a class or structure.
28. If you have a method returning a collection, return an empty collection instead of null, if you have no data to return. For example, if you have a method returning an **ArrayList**, always return a valid **ArrayList**. If you have no items to return, then return a valid **ArrayList** with 0 items. This will make it easy for the calling application to just check for the “**Count**” rather than doing an additional check for “**null**”.
29. Use the **AssemblyInfo** file to fill information like version number, description, company name, copyright notice etc.
30. Logically organize all your files within appropriate folders. Use 2 level folder hierarchies. You can have up to 10 folders in the root folder and each folder can have up to 5 sub folders. If you have too many folders than cannot be accommodated with the above mentioned 2 level hierarchy, you may need re factoring into multiple assemblies.
31. Make sure you have a good logging class which can be configured to log errors, warning or traces. If you configure to log errors, it should only log errors. But if you configure to log traces, it should record all (errors, warnings and trace). Your log class should be written such a way that in future you can change it easily to log to Windows Event Log, SQL Server, or Email to administrator or to a File etc. without any change in any other part of the application. Use the log class extensively throughout the code to record errors, warning and even trace messages that can help you trouble shoot a problem.
32. If you are opening database connections, sockets, file stream etc., always close them in the **finally** block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the **finally** block.
33. Declare variables as close as possible to where it is first used. Use one variable declaration per line.
34. Use **StringBuilder** class instead of **String** when you have to manipulate string objects in a loop. The **String** object works in weird way in .NET. Each time you append a string, it is actually discarding the old string object and recreating a new object, which is a relatively expensive operation.

Consider the following example:

```
public string ComposeMessage (string[] Lines)
{
    string Message = String.Empty;
```

```
        for (int I = 0; I < Lines.Length; I++)  
            Message += Lines [I];  
  
        return Message;  
    }
```

In the above example, it may look like we are just appending to the string object 'message'. But what is happening in reality is, the string object is discarded in each iteration and recreated and appending the line to it.

If your loop has several iterations, then it is a good idea to use StringBuilder class instead of String object.

See the example where the String object is replaced with StringBuilder.

```
public string ComposeMessage (string[] Lines)  
{  
    StringBuilder Message = new StringBuilder();  
  
    for (int Counter = 0; Counter < Lines.Length; Counter++)  
        Message.Append( lines[Counter] );  
  
    return Message.ToString();  
}
```

6. Architecture

1. Always use multi-layer (N-Tier) architecture.
2. Always do layering. (Separate your application into logical layers. E.g. DAL, Business, UI etc.)
3. Never access database from the UI pages. Always have a data layer class which performs all the database related tasks. This will help you support or migrate to another database back end easily.
4. Use try-catch in your data layer to catch all database exceptions. This exception handler should record all exceptions from the database. The details recorded should include the name of the command being executed, stored proc name, parameters, connection string used etc. After recording the exception, it could be re thrown so that another layer in the application can catch it and take appropriate action.
5. 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.

7. ASP.NET

1. Do not use session variables throughout the code. Use session variables only within the classes and expose methods to access the value stored in the session variables. A class can access the session using [System.Web.HttpContext.Current.Session](#)
2. Do not store large objects in session. Storing large objects in session may consume lot of server memory depending on the number of users.
3. Always use style sheet to control the look and feel of the pages. Never specify font name and font size in any of the pages. Use appropriate style class. This will help you to change the UI of your application easily in future. Also, if you like to support customizing the UI for each customer, it is just a matter of developing another style sheet for them

8. Comments

Good and meaningful comments make code more maintainable. However,

1. Do not write comments for every line of code and every variable declared.
2. Use `//` or `///` for comments. Avoid using `/* ... */`
3. Write comments wherever required. But good readable code will require very less comments. If all variables and method names are meaningful, that would make the code very readable and will not need many comments.
4. Do not write comments if the code is easily understandable without comment. The drawback of having lot of comments is, if you change the code and forget to change the comment, it will lead to more confusion.
5. Fewer lines of comments will make the code more elegant. But if the code is not clean/readable and there are less comments, that is worse.
6. If you have to use some complex or weird logic for any reason, document it very well with sufficient comments.
7. If you initialize a numeric variable to a special number other than 0, -1 etc, document the reason for choosing that value.
8. The bottom line is, write clean, readable code such a way that it doesn't need any comments to understand.
9. Perform spelling check on comments and also make sure proper grammar and punctuation is used.

9. Exception Handling

1. Never do a 'catch exception and do nothing'. If you hide an exception, you will never know if the exception happened or not. Lot of developers uses this handy method to ignore non-significant errors. You should always try to avoid exceptions by checking all the error conditions programmatically. In any case, catching an exception and doing nothing is not allowed. In the worst case, you should log the exception and proceed.
2. In case of exceptions, give a friendly Message to the user, but log the actual error with all possible details about the error, including the time it occurred, method and class name etc.
3. Always catch only the specific exception, not generic exception.

Good:

```
void ReadFromFile ( string FileName )
{
    try
    {
        // read from file.
    }
    catch (FileNotFoundException Ex)
    {
        // log error.
        // re-throw exception depending on your case.
        throw;
    }
}
```

Not Good:

```
void ReadFromFile (string FileName)
{
    try
    {
        // read from file.
    }
    catch (Exception ex)
    {
        // Catching general exception is BAD. We will never know
        whether
        // it was a file error or some other error. Here you are hiding
        an
        // exception & no one will ever know that an exception
        happened.
        return "";
    }
}
```

4. 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'.
5. When you re throw an exception, use the `throw` statement without specifying the original exception. This way, the original call stack is preserved.

Good:

```
catch
{
    // do whatever you want to handle the exception
    throw;
}
```

Not Good:

```
catch (Exception ex)
{
    // do whatever you want to handle the exception
    throw ex;
}
```

6. Do not write try-catch in all your methods. Use it only if there is a possibility that a specific exception may occur and it cannot be prevented by any other means. For example, if you want to insert a record if it does not already exist in database, you should try to select record using the key. Some developers try to insert a record without checking if it already exists. If an exception occurs, they will assume that the record already exists. This is strictly not allowed. You should always explicitly check for errors rather than waiting for exceptions to occur. On the other hand, you should always use exception handlers while you communicate with external systems like network, hardware devices etc. Such systems are subject to failure anytime and error checking is not usually reliable. In those cases, you should use exception handlers and try to recover from error.
7. Do not write very large try-catch blocks. If required, write separate try-catch for each task you perform and enclose only the specific piece of code inside the try-catch. This will help you find which piece of code generated the exception and you can give specific error Message to the user.
8. Write your own custom exception classes if required in your application. Do not derive your custom exceptions from the base class **System Exception**. Instead, inherit from **Application Exception**.

10. Cascading Style Sheets (CSS)

1. Structure

- Use tabs, not spaces, to indent each property.
- Add two blank lines between sections and one blank line between blocks in a section.
- Each selector should be on its own line, ending in either a comma or an opening curly brace. Property-value pairs should be on their own line, with one tab of indentation and an ending semicolon. The closing brace should be flush left, using the same level of indentation as the opening selector.

Correct:

```
#Selector1,  
#Selector2,  
#Selector3  
{  
    background: #fff;  
    color: #000;  
}
```

Incorrect:

```
#selector-1, #selector-2, #selector-3 {  
    background: #fff;  
    color: #000;  
}  
  
#selector-1 { background: #fff; color: #000; }
```

2. Selectors

- Use human readable selectors that describe what element(s) they style.
- Attribute selectors should use double quotes around values.
- Refrain from using over-qualified selectors, div.container can simply be stated as .container

Correct:

```
#CommentForm
{
    margin: 1em 0;
}

input[type="text"]
{
    line-height: 1.1;
}
```

Incorrect:

```
#commentForm { /* Avoid camelcase. */
    margin: 0;
}

#comment_form { /* Avoid underscores. */
    margin: 0;
}

div#comment_form { /* Avoid over-qualification. */
    margin: 0;
}

#cl-xr { /* What is a cl-xr?! Use a better name. */
    margin: 0;
}

input[type=text] { /* Should be [type="text"] */
    line-height: 110% /* Also doubly incorrect */
}
```

3. Properties

- Properties should be followed by a colon and a space.
- All properties and values should be lowercase, except for font names and vendor-specific properties.
- Use hex code for colors, or rgba() if opacity is needed. Avoid RGB format and uppercase and shorten values when possible: #fff instead of #FFFFFF.
- Use shorthand (except when overriding styles) for background, border, font, list-style, margin, and padding values as much as possible.

Correct:

```
#selector-1
{
```



```
background: #fff;
display: block;
margin: 0;
margin-left: 20px;
}
```

Incorrect:

```
#selector-1 {
    background: #FFFFFF;
    display: BLOCK;
    margin-left: 20PX;
    margin: 0;
}
```

4. **Property Ordering #Property Ordering**

- Display
- Positioning
- Box model
- Colors and Typography
- Other

Top/Right/Bottom/Left (TRBL/trouble) should be the order for any relevant properties (e.g. margin), much as the order goes in values. Corner specifiers (e.g. border-radius—) should be top-left, top-right, bottom-right, bottom-left. This is derived from how shorthand values would be ordered.

Example:

```
#Overlay
{
    position: absolute;
    z-index: 1;
    padding: 10px;
    background: #fff;
    color: #777;
}
```

Another method that is often used, is to order properties alphabetically, with or without certain exceptions.

Example:

```
#Overlay
{
    background: #fff;
    color: #777;
    padding: 10px;
    position: absolute;
    z-index: 1;
}
```

5. Vendor Prefixes

Example:

```
.SampleOutput
{
    -webkit-box-shadow: inset 0 0 1px 1px #eee;
    -moz-box-shadow: inset 0 0 1px 1px #eee;
    box-shadow: inset 0 0 1px 1px #eee;
}
```

6. Values

- Space before the value, after the colon.
- Do not pad parentheses with spaces.
- Always end in a semicolon.
- Use double quotes rather than single quotes, and only when needed, such as when a font name has a space or for the values of the content property.
- Font weights should be defined using numeric values (e.g. 400 instead of normal, 700 rather than bold).
- 0 values should not have units unless necessary, such as with transition-duration.
- Line height should also be unit-less, unless necessary to be defined as a specific pixel value.
- Use a leading zero for decimal values, including in rgba().
- Multiple comma-separated values for one property should be separated by either a space or a newline. For better readability newlines should be used for lengthier multi-part values such as those for shorthand properties like box-shadow and text-shadow, including before the first value. Values should then be indented one level in from the property.
- Lists of values within a value, like within rgba(), should be separated by a space.

Correct:

```
.Class
{
    /* Correct usage of quotes */
    background-image: url(images/bg.png);
    font-family: "Helvetica Neue", sans-serif;
    font-weight: 700;
}

.Class
```

```

{
    /* Correct usage of zero values */
    font-family: Georgia, serif;
    line-height: 1.4;
    text-shadow: 0 -1px 0 rgba(0, 0, 0, 0.5), 0 1px 0
    #fff;
}

.Class
{
    /* Correct usage of short and lengthier multi-part
    values */
    font-family: Consolas, Monaco, monospace;
    transition-property: opacity, background, color;
    box-shadow: 0 0 0 1px #5b9dd9, 0 0 2px 1px rgba(30,
    140, 190, 0.8);
}

```

Incorrect:

```

.class
{
    /* Avoid missing space and semicolon */
    background:#fff
}

.class
{
    /* Avoid adding a unit on a zero value */
    margin: 0px 0px 20px 0px;
}

.class {
    font-family: Times New Roman, serif; /* Quote font
    names when required */
    font-weight: bold; /* Avoid named font weights */
    line-height: 1.4em; /* Avoid adding a unit for line
    height */
}

```

```
.class { /* Incorrect usage of multi-part values */  
    text-shadow: 0 1px 0 rgba(0, 0, 0, 0.5),  
                0 1px 0 #fff;  
    box-shadow: 0 1px 0 rgba(0, 0,  
                        0, 0.5),  
                0 1px 0 rgba(0,0,0,0.5);  
}
```

7. Media Queries

- It is generally advisable to keep media queries grouped by media at the bottom of the stylesheet.
- Rule sets for media queries should be indented one level in.

Example:

```
@media all and (max-width: 699px) and (min-width: 520px)  
{  
    /* Your selectors */  
}
```

8. Commenting

- Comment, and comment liberally. If there are concerns about file size, utilize minified files and the SCRIPT_DEBUG constant. Long comments should manually break the line length at 80 characters.
- A table of contents should be utilized for longer stylesheets, especially those that are highly sectioned. Using an index number (1.0, 1.1, 2.0, etc.) aids in searching and jumping to a location.

For sections and subsections:

```

55  /*****
56      Bootstrap Overrides
57  *****/
58  .container...
63
64  #FormContent, #FormContent::after, #FormContent::before...
```

For inline:

```

144  /*--- HTML Controls ---*/
145  a...
151
152  a:link, a:visited...
```

9. Best Practices

- If you are attempting to fix an issue, attempt to remove code before adding more.
- Magic Numbers are unlucky. These are numbers that are used as quick fixes on a one-off basis.
Example: **.Box** { margin-top: 37px }.
- DOM will change over time, target the element you want to use as opposed to “finding it” through its parents.
Example: Use **.Highlight** on the element as opposed to **.Highlight a** (where the selector is on the parent)
- Know when to use the height property. It should be used when you are including outside elements (such as images). Otherwise use line-height for more flexibility.
- Do not restate default property & value combinations (for instance display: block; on block-level elements).

11. JavaScript

1. Spacing

- Use spaces liberally throughout your code.
- These rules encourage liberal spacing for improved developer readability. The minification process creates a file that is optimized for browsers to read and process.
- Indentation with tabs.
- No whitespace at the end of line or on blank lines.
- Lines should usually be no longer than 80 characters and should not exceed 100 (counting tabs as 4 spaces). This is a “soft” rule, but long lines generally indicate unreadable or disorganized code.
- if/else/for/while/try blocks should always use braces, and always go on multiple lines.
- Unary special-character operators (e.g., ++, --) must not have space next to their operand.
- Any , and ; must not have preceding space.
- Any ; used as a statement terminator must be at the end of the line.
- Any : after a property name in an object definition must not have preceding space.
- The ? and : in a ternary conditional must have space on both sides.
- No filler spaces in empty constructs (e.g., {}, [], fn()).
- There should be a new line at the end of each file.
- Any ! negation operator should have a following space.*
- All function bodies are indented by one tab, even if the entire file is wrapped in a closure.*
- Spaces may align code within documentation blocks or within a line, but only tabs should be used at the start of a line.*

2. Objects

- Object declarations can be made on a single line if they are short (remember the line length guidelines). When an object declaration is too long to fit on one line, there must be one property per line. Property names only need to be quoted if they are reserved words or contain special characters:
- Objects and arrays can be declared on a single line if they are short (remember the line length guidelines). When an object or array is too long to fit on one line, each member must be placed on its own line and each line ended by a comma.
- Do not use **class** keyword to define classes even with ECMA2015(ECMA6).
 - Use **function** keyword instead
 - **class** keyword and the implementation will get translated into **function** keyword implementation in the background.
 - Using **class** keyword will only add an additional overhead to the compiler.

3. Arrays and Function Calls

- Don't include extra spaces around elements and arguments:

```

Let Array = [Value1, Value2];
Foo(Arg);
Foo('String', Object);

Foo(Options, Object[Property]);
Foo(Node, 'Property', 2);
Prop = Object[ 'Default'];
FirstArrayElement = Arr[0];

```

Examples of Good Spacing

```

if (Condition)
  DoSomething('Hello');
else if (OtherCondition )
  DoOtherThing(
    {
      Key : Key,
      Value : Value
    });
else
  DoSomethingElse(true);

```

4. Semicolons

- Use them. Never rely on Automatic Semicolon Insertion (ASI).

5. Indentation and Line Breaks

- Indentation and line breaks add readability to complex statements.
- Tabs should be used for indentation. Even if the entire file is contained in a closure (i.e., an immediately invoked function), the contents of that function should be indented by one tab:

```

(function(Arg1)
{
  DoSomething();

  function DoAnotherThing()
  {
    // Expressions indented
  }
});

```

6. Blocks and Curly Braces

if, else, for, while, and try blocks should always use braces, and always go on multiple lines. The opening brace should be on the same line as the function

definition, the conditional, or the loop. The closing brace should be on the line directly following the last statement of the block.

```
let Val1, Val2, Val3;

if (DoJob1())
{
    // Expressions
}
else if ((Val1 && Val2) || Val3)
{
    // Expressions
}
else
{
    // Expressions
}
```

7. Multi-line Statements

When a statement is too long to fit on one line, line breaks must occur after an operator.

```
// Bad
let Html = '<p>The sum of ' + Val1 + ' and ' + Val2 + ' plus ' + Val3
          + ' is ' + (Val1 + Val2 + Val3) + '</p>';
```

```
// Good
let Html = '<p>The sum of ' + Val1 + ' and ' + Val2 + ' plus ' + Val3 +
          ' is ' + (Val1 + Val2 + Val3) + '</p>';
```

Lines should be broken into logical groups if it improves readability, such as splitting each expression of a ternary operator onto its own line, even if both will fit on a single line.

```
// Acceptable
let Thing = (true === ConditionalStatement()) ? 'thing 1' : 'thing 2';
```

```
// Better
var Thing = GetFirstCondition(Foo) && GetSecondCondition(Bar) ?
    Stich(Foo, Bar) :
    Foo;
```

When a conditional is too long to fit on one line, each operand of a logical operator in the Boolean expression must appear on its own line, indented one extra level from the opening and closing parentheses.

```
if (GetFirstCondition() &&
    GetSecondCondition() &&
```

```

    GetThirdCondition()
  {
    Execute();
  }

```

8. Chained Method Calls

When a chain of method calls is too long to fit on one line, there must be one call per line, with the first call on a separate line from the object the methods are called on. If the method changes the context, an extra level of indentation must be used.

Elements

```

    .addClass('foo')
    .children()
    .html('hello')
    .end()
    .appendTo('body');

```

9. Variables Assignments

- Declaring Variables with **const** and **let**
 - For code written using ES2015 or newer, **const** and **let** should always be used in place of **var**. A declaration should use **const** unless its value will be reassigned, in which case **let** is appropriate.
 - Unlike **var**, it is not necessary to declare all variables at the top of a function. Instead, they are to be declared at the point at which they are first used.
- Declaring Variables With **var**
 - Each function should begin with a single comma-delimited **var** statement that declares any local variables necessary. If a function does not declare a variable using **var**, that variable can leak into an outer scope (which is frequently the global scope, a worst-case scenario), and can unwittingly refer to and modify that data.
 - Assignments within the **var** statement should be listed on individual lines, while declarations can be grouped on a single line. Any additional lines should be indented with an additional tab. Objects and functions that occupy more than a handful of lines should be assigned outside of the **var** statement, to avoid over-indentation.

```

// Good
var Age, Gender, Height,
    // Indent subsequent lines by one tab
    Name = 'TKSE_PostOffice';

```

```

// Bad
var Foo = true;
var Bar = false;
var Age;

```

```
var Gender;  
var Height;
```

- Global Variables
 - Since core JavaScript files are sometimes used within plugins, existing global variables should not be removed.
 - All global variables used within a file should be documented at the top of that file. Multiple global variables can be comma-separated.

10. Common Libraries

Include common libraries before including project specific java script files

11. Naming Conventions

- Use Pascal casing for Class, Function, Variable, Parameter & File names.

```
//Class  
function Planet(Index, Name)  
{  
    this.Name = Name;  
    this.Index = Index;  
  
    this.GetName()  
    {  
        return this.Name;  
    };  
}  
  
//Method  
function CalculateAge(DateOfBirth)  
{  
    //logic  
}
```

- **Do not** use Hungarian notation to name variables except for member variables. In earlier days most of the programmers liked it - having the data type as a prefix for the variable name and using m_ as prefix for member variables.

E.g.: `var m_sName; let nAge;`

- Use Meaningful, descriptive words to name variables. Do not use abbreviations.

Good:

```
var Address;  
let Salary;
```

Not Good:

```
var Name;  
let Sal;
```

Do not use single character variable names like i, n, s etc. Use names like Index, TempOne exception in this case would be variables used for iterations in loops:

Not Good:

```
for (int Counter = 0; Counter < MaxCount; Counter++)  
{  
    ...  
}
```

Not Good:

```
for (int I = 0; I < Count; I++)  
{  
    ...  
}
```

If the variable is used only as a counter for iteration and is not used anywhere else in the loop, many people still like to use a single char variable (i) instead of inventing a different suitable name.

- Do not use underscores (_) for local variable names to separate words.
- All member/global variables must be prefixed with underscore (_) so that they can be identified from other local variables.
- Use **var** for global variables and **let** for local variables.
- Do not use variable names that resemble keywords.
- Prefix **boolean** variables, properties and methods with "Is" or similar prefixes.
let _IsFinished
- Use appropriate prefix for the UI elements so that you can identify them from the rest of the variables. Use appropriate prefix for each of the UI element. A brief list is given below.

Control	Prefix
Label	Lbl
TextBox/Textarea	Txt
DataGrid	Grd

Button	Cmd
ImageButton	Cmd
Hyperlink	Lnk
Select	Drp
Checkbox	Chk
RadioButton	Rdo
Image	Img
Div	Div
Span	Spn
Table	Tbl
FileUpload	Fup

12. File name should match with class name. For example, for the class **HelloWorld**, the file name should be **HelloWorld.js**.

13. Use Pascal Case for file names.

14. Strict equality checks (===) must be used in favor of abstract equality checks (==).

15. Type Checks

These are the preferred ways of checking the type of an object:

String: `typeof(Entity) === 'string'`

Number: `typeof(Entity) === 'number'`

Boolean: `typeof(Entity) === 'boolean'`

Object: `typeof(Entity) === 'Entity' or _.isObject(Entity)`

Plain Object: `jQuery.isPlainObject(Entity)`

Function: `_.isFunction(Entity) or jQuery.isFunction(Entity)`

Array: `_.isArray(Entity)` or `jQuery.isArray(Entity)`

Element: `object.nodeType` or `_.isElement(Entity)`

null: `Entity === null`

null or undefined: `Entity == null`

undefined:

- Global Variables: `typeof(Entity) === undefined`

- Local Variables: `Entity === undefined`

- Properties: `object.prop === undefined`

- Any of the above: `_.isUndefined(object)`

CONFIDENTIAL

16. Comments

- Comments come before the code to which they refer, and should always be preceded by a blank line. Capitalize the first letter of the comment, and include a period at the end when writing full sentences. There must be a single space between the comment token (//) and the comment text.

```
//SomeStatement();
```

```
// Explanation of something complex on the next line  
DoSomething();
```

- Inline comments are allowed as an exception when used to annotate special arguments in formal parameter lists:

```
function Sample(Param1, Param2, Param3, /* INTERNAL */ Param4)  
{  
    // Logic  
}
```

17. Strings

Use single-quotes for string literals:

```
Sentence = 'Strings should be contained in single quotes';
```

When a string contains single quotes, they need to be escaped with a backslash (\):

```
// Escape single quotes within strings:  
'Note the backslash before the \'single quotes\' in strings.';
```

Use **String.Empty** instead of "" in your code. Include **TKSE_PostOffice.Extender.js**.

18. Switch Statements

- The usage of switch statements is generally discouraged, but can be useful when there are a large number of cases – especially when multiple cases can be handled by the same block, or fall-through logic (the default case) can be leveraged.
- When using switch statements, use a break for each case other than default. When allowing statements to “fall through,” note that explicitly.
- Indent case statements one tab within the switch.

```
switch (Event.keyCode)  
{  
    // ENTER and SPACE both trigger Method1()  
    case $.ui.keyCode.ENTER:  
    case $.ui.keyCode.SPACE:  
        Method1 ();  
        break;
```

```
        case $.ui.keyCode.ESCAPE:
            Method2();
            break;

        default:
            Method3();
    }
```

- It is not recommended to return a value from within a switch statement: use the case blocks to set values, then return those values at the end.

```
function GetKeyCode(Event)
{
    var Result;

    switch (Event.keyCode)
    {
        case $.ui.keyCode.ENTER:
        case $.ui.keyCode.SPACE:
            Result = 'commit';
            break;

        case $.ui.keyCode.ESCAPE:
            Result = 'exit';
            break;

        default:
            Result = 'default';
    }

    return Result;
}
```


19. Event Handling

A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element. To execute code when a user clicks on an element, add JavaScript code to an HTML event attribute:

```
<a id="LnkSubmit" onclick="LnkSubmit_Click">Click on this text!</a>

<script type="text/javascript">
    function LnkSubmit_Click(Sender)
    {
        ...
    }
</script>
```

Event handling method name should be according to the following format.

ElementName_Event(Sender);

20. Best Practices

- Arrays

- Creating arrays in JavaScript should be done using the shorthand [] constructor rather than the **new Array()** notation.

```
let NewArray = [];
```

- You can initialize an array during construction:

```
var myArray = [ 1, 'WordPress', 2, 'Blog' ];
```

- In JavaScript, associative arrays are defined as objects.

- Objects

- There are many ways to create objects in JavaScript. Object literal notation, {}, is both the most performant, and also the easiest to read.

```
let Entity = {};
```

- Object literal notation should be used unless the object requires a specific prototype; in which case the object should be created by calling a constructor function with new.

```
var Entity = new ConstructorMethod();
```

- Object properties should be accessed via dot notation, unless the key is a variable or a string that would not be a valid identifier:

```
Prop = Entity.propertyName;
Prop = Entity[Index];
```

```
Prop = Entity['Key/Name'];
```

- Iteration
 - **Never use jQuery to iterate over raw data or Vanilla JavaScript objects.**
The only time jQuery should be used for iteration is when iterating over a collection of jQuery objects.
 - When iterating over a large collection using a for loop, it is recommended to store the loop's max value as a variable rather than re-computing the maximum every time:

```
// Good & Efficient
let Counter, Max;

for (Counter = 0, Max = Array.Length; Counter < Max; Counter++)
{
    // Logic
}

// Bad & Potentially Inefficient:
for (let I = 0; I < Array.Length; I++) // Array.Length gets called every
time
{
    // Logic
}
```

- Underscore.js Collection Functions
 - Learn and understand Underscore's collection and array methods. These functions, including **_.each**, **_.map**, and **_.reduce**, allow for efficient, readable transformations of large data sets.
 - Underscore also permits jQuery-style chaining with regular JavaScript objects:

```
let Entity =
{
    Property1 : 'Value 1',
    Property2 : 'Value 2'
};

let Array = _.chain(Entity)
    .keys()
    .map(function(Key)
    {
        return Key + ' comes ' + Entity [Key];
    })
    .value(); // Exit the chain
```