Coding Standards and Best Practices

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 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.

At Triangle Factory we use this standard.

Naming Conventions

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

Use Pascal Casing for Class names

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

Use Pascal Casing for Method names

```
public void SayHello(string name)
{
        ...
}
```

Use Camel Casing for local variables and method parameters

```
int totalCount = 0;

void SayHello(string name)
{
        string fullMessage = $"Hello {name}";
}
```

Do not use underscores for local variable names. All private member variables use Camel Casing and must be prefixed with an underscore so that they can be identified from other local variables. `protected` and `public` variables use Pascal Casing.

```
private string _address;
protected string Name;
public int Number;
```

Use the prefix `I` with Pascal Casing for interfaces.

```
public class HelloWorld : IEntity
{
        ...
}
```

Do not use Hungarian notation for name variables. In earlier days most of the programmers liked it - having the data type as a prefix for the variable name.
However, this is not recommended. Usage of data type should not be used as this requires refactoring to change a variable's type. All variables should use camel casing and should be independent of their data type.

DO

```
string name;
int age;
int[] numbers;
```

DON'T

```
string sName;
int nAge;
int[] iNumbersArr;
```

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

DO

```
string address;
int salary;
```

DON'T

```
string nam;
string addr;
int sal;
```

Do not use single character variable names like `t`, `n`, `s` etc. Use names like `index`, `temp`. One exception in this case would be variables used for iterations in loops (`i`, `kvp`). Another exception could be variables in mathematical code, like `x`, `y` and `z`.

DO

```
int temp;
int xPosition = Vector3.X;
for (int i = 0; i < _numbers.Count; ++i)
{
  // ...
}
```

DON'T

```
string s;
int x;
Object o;
```

Do not use variable names that resemble keywords.

DON'T

```
string object;
object @object;
```

Prefix boolean variables, properties and methods with `Is`, `Can` or similar prefixes.

```
private bool isFinished;
private bool canBeBought;

public bool IsValid(int number)
{
        ...
}
```

File names should match with the (main) class name. Use Pascal Case for file names.

HelloWorld.cs

Constants are written in Pascal Casing.

```
public const int MaximumNumber = 100;
public const long InvalidPlayerId = -1587L;
```

Static variables use the same rules as normal variables.

```
private static string _address;
protected static string Name;
public static int Number;
```

A method name should tell what it does. Do not use misleading names. Don't overdo this either.

DO

```
public void SaveUserData(UserData userData)
{
        // Save the user data to xml.
}
```

DON'T

```
public void Save(UserData userData)
{
        // Save the user data to xml.
}

// This might be a bit too much.
// It requires refactoring if we ever decide to work with json files
instead of xml files.
public void SaveUserDataToXml(UserData userData)
{
        // Save the user data to xml.
}
```

Indentation, Spacing and Comments

Use **TABS** for indentation. Do not use **SPACES**. Define the tab size as 4. Make sure to specify this in your IDE settings for **ALL** languages (yes, also for xml, php, …)!

```
if (isValid)
{
< TAB > // No spaces!
}
```

Curly braces {} should start on a new line and the opening and closing braces should have the same indentation. Code inside the braces has one extra indentation. The only exception is a really small body; then you can put it all on one line.

DO

```
public bool IsValid(int number)
{
        ...
}

for (int i = 0; i < _numbers.Count; ++i)
{
        ...
}

Dictionary<string, string> getArgs = new Dictionary<string, string>
{
        {"access_token", AccessToken}                // This is ok.
};
```

DON'T

```
public bool IsValid(int number) {
        ...
}

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

Always use curly braces for control structures, even if they have a body consisting of a single line. This avoids errors if you have to add lines to the body later on.

DO

```
int number = 5;
if ((number % 2) == 0)
{
        Debug.Log("Even");
}
else
{
        Debug.Log("Uneven");
}
```

DON'T

```
int number = 5;
if ((number % 2) == 0)
        Debug.Log("Even");
else
        Debug.Log("Uneven");
```

The only exception to this rule is a return statement.

```
if (listContainer == null) return;
```

Use regions to group related pieces of code together. Always use the following regions, but feel free to omit any that are empty. You may also add additional regions within these regions.

- Editor Fields: contains all the fields that are exposed to the unity inspector.
- Fields: contains all the private fields of the class/struct.
- Properties
- Events
- Life Cycle: Contains constructors, Unity's Start and Awake, other kind of initialization methods, destructors, Dispose and Unity's OnDestroy.
- Methods

There should always be one and only one single blank line between each method inside a class. There should be one blank line before and after a region declaration. The blank line can be ommitted between fields for grouping, except when the fields have a summary.

DO

```
#region Fields

private int _number1;
private int _number2;

#endregion

#region Methods

public void Method1()
{
        ...
}

public void Method2()
{
        ...
}

#endregion
```

DON'T

```
#region Fields
private int _number1;
private int _number2;

#endregion

#region Methods



public void Method1()
{
        ...
}
public void Method2()
{
        ...
}

#endregion
```

Use a single space before and after each operator.

DO

```
if (showResult)
{
        for (int i = 0; i < 10; ++i)
        {
                ...
        }
}
```

DON'T

```
if ( showResult )
{
        for ( int i = 0 ; i < 10 ; ++i )
        {
                ...
        }
}

if(showResult)
{
        for(int i=0;i<10;++i)
        {
                ...
        }
}
```

Try to keep lines shorter than 80 characters. A little bit more is OK, but avoid lines of 200 characters.

Split the conditional operator over multiple lines unless it's a really simple one.

DO

```
return fullUrl.EndsWith("/")
        ? (fullUrl.Substring(0, fullUrl.Length - 1))
        : fullUrl;
```

DON'T

```
return fullUrl.EndsWith("/") ? (fullUrl.Substring(0, fullUrl.Length -
1)) : fullUrl;
```

Adding summaries for classes, fields, properties and methods **is not required** in your game code but is allowed when extra information is needed.
Instead, make sure your method names are self explaining and when required, place comments inside the method to annotate behaviour. Use the double slash notation `// comment`. Avoid using `/* comment */`.

```
public class PlayerMovement : MonoBehaviour
{
  #region Methods

  public void HandlePlayerInput()
  {
    // read the player input
    ...

    // apply the offset from the headset
    ...
  }

  #endregion
}
```

Adding summaries for classes, fields, properties and methods **is required** in library code.
Library code is code that will be reused across other projects.
When you write documentation, use the standard C# documentation rules.
For Coroutines, you can omit the `returns` tag.

```
/// <summary>
/// A class with handy methods to do math
/// </summary>
public static class MathHelpers
{
  #region Methods

  /// <summary>
  /// Returns the absolute value of the given number.
  /// </summary>
  /// <param name="number">The number where we want the absolute value
from.</param>
  /// <returns>The absolute value of the given number.</returns>
  public static float GetAbsoluteValue(float number)
  {
        return Mathf.Abs(number);
  }

  #endregion
}
```

Events and Callbacks

Use events. Use an `Action` to define your event signature. Use the `Invoke` method with Null-conditional operator `?.`, this way you will not get a null reference exception when there are no subscribers. This operator is thread-safe.

```
public class Player
{
    public event Action<Player, int> TookDamage;

        // In your code where the player took damage.
        TookDamage?.Invoke(this, damage);
}
```

Add an arguments class when you have more then 3 arguments to pass with the event. Also use an arguments struct when the arguments are not clear.

DO

```
public class EnemyManager
{
  public struct EnemyDestroyedEventArgs
  {
    public Enemy DestroyedEnemy;
    public Player Killer;
    public int Damage;
  }

  public event Action<EnemyManager, EnemyDestroyedEventArgs>
EnemyDestroyed;
}
```

DON'T

```
public event Action<EnemyManager, Enemy, Player, int> EnemyDestroyed;
```

Correctly subscribe to events. The first parameter is the object that fired the event (don't use object, use the current type). Use the `On` prefix for subscriber/callback methods.

```
private static void OnLanguageLoaded(Localization localization,
LanguageLoadedEventArgs languageLoadedEventArgs)
{
        if (languageLoadedEventArgs.Success)
        {
                Debug.Log($"'{languageLoadedEventArgs.LanguageCode}'
loaded successfully.");
        }
        else
        {
                Debug.LogWarning($"'{languageLoadedEventArgs.
LanguageCode}' didn't load successfully.");
        }
}
```

Avoid settings UnityEvents from the inspector. Set it from the code as early as possible.

```
public class ExampleClass : MonoBehaviour
{
    private UnityEvent _myEvent;

    protected void Awake()
    {
        if (_myEvent== null)
        {
          _myEvent = new UnityEvent();
        }
        _myEvent.AddListener(OnPing);
    }
}
```

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. The same goes for other UI elements.

```
_reloadLevelButton.OnClickEvent += OnReloadLevelButtonClick;

// Don’t programmatically call this function.

private void OnReloadLevelButtonClick (Button button,
ButtonClickEventArgs eventArgs)
{
        ReloadLevel();
}

// You can call this function from anywhere in the code.
private void ReloadLevel()
{
        // Reload the level.
}
```

Condition Checking

Prefer to check on `false` conditions and return early from functions instead of enclosing the body of your function with a lot of `if` conditions.

DO

```
public bool CanBuyItem(Player player, Item item)
{
        if (player.Credits < item.Cost) return false;
        if (player.Level < item.UnlockLevel) return false;

        return true;
}
```

DON'T

```
public bool CanBuyItem(Player player, Item item)
{
        if (player.Credits >= item.Cost && player.Level >= item.
UnlockLevel)
        {
                return true;
        }

        return false;
}
```

If there are lots of checks that can cause an early return from a function, it might be even better to define an `enum` and return one of the `enum` values so the calling code has a better idea why it was rejected.

```
public CanBuyItemResult CanBuyItem(Player player, Item item)
{
        if (player.Credits < item.Cost)
        {
                return CanBuyItemResult.NotEnoughCredits;
        }

        if (player.Level < item.UnlockLevel)
        {
                return CanBuyItemResult.TooLowLevel;
        }

        return CanBuyItemResult.Success;
}
```

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.

DO

```
if (memberType == MemberTypes.Registered)
{
        // Registered user ...
}
else if(memberType == MemberTypes.Guest)
{
        // Guest user ...
}
else
{
        // Un expected user type.
        throw new NotImplementedException($"Unexpected value
'{memberType.ToString()'.");

        // If we introduce a new user type in future, we can easily
find the problem here.
}
```

DON'T

```
if (memberType == MemberTypes.Registered)
{
        //Registered user ...
}
else
{
        // Guest user ...
        // If we introduce another user type in future, this code will
fail and will not be noticed.
}
```

When applicable, use
`string.Compare(A, B, CultureInfo.InvariantCulture` or
`string.Compare(A, B, CultureInfo.InvariantCultureIgnoreCase)`.
This is faster than doing a normal string compare and is good enough for 99% of all use cases in games.
Adding `IgnoreCase` will give you less problems when dealing with user input or editor tools.

Use `String.IsNullOrEmpty` instead of checking on `null` and `""`.

DO

```
if (string.Compare(name, "john", CultureInfo.
InvariantCultureIgnoreCase))
{
  ...
}

if (string.IsNullOrEmpty(name))
{
  ...
}
```

DON'T

```
if (name == "john")
{
  ...
}

if (name == null || name == string.Empty)
{
  ...
}
```

Don't included `== true` in boolean checks, but do include `== false` instead of using an exclamation mark. It's a lot easier to miss an exclamation mark then to miss `== false` in a condition.

DO

```
if (dictionary.ContainsKey(key) == false)
{
        ...
}
```

DON'T

```
if (!dictionary.ContainsKey(key))
{
        ...
}
```

Performance

Put all log calls in an #if DEBUG. This way your debug log will not slow down release builds.

```
private void ErrorReceived(string error)
{
#if DEBUG
  Debug.Log($"Error received: {error}");
#endif
}
```

Use string interpolation when creating strings with arguments.

```
string myString = $"The name is {_name}";
```

Use `StringBuilder` class instead of `string` when you have to manipulate string objects in a loop. Each time you append a string, it is actually discarding the old string object and recreating a new object (GC), which is a relatively expensive operation.

DO

```
Dictionary<string, string>> args = ...

StringBuilder sb = new StringBuilder();
foreach (KeyValuePair<string, string> kvp in args)
{
        sb.AppendLine($"{kvp.Key} {kvp.Value}");
}
string s= sb.ToString();
```

DON'T

```
string s = new string();

foreach (KeyValuePair<string, string> kvp in args)
{
        s += ($"{kvp.Key} {kvp.Value}{Environment.NewLine}");
}
```

Do not use `foreach` when possible, use a for loop. The mono implementation generates a huge amount of garbage.

DO

```
private void Method()
{
  for (int i = 0; i < _list.Count; ++i)
  {
    ...
  }
  foreach(KeyValuePair<string, string> kvp in _dictionary)
  {
    ...
  }
}
```

DON'T

```
private void Method()
{
  foreach (int element in _list)
  {
    // ...
  }
}
```

Do not use `LINQ`, `LINQ` uses `foreach` for every extension method. Use our extension methods that have the for-loop implementation. if you really need to do a lot of `LINQ` like transformations (unlikely), consider looking at LinqBenchmarks to figure out a good `LINQ` library for your use case. Make sure you have a good understanding of how LINQ works before you do this. Consult with your seniors if in doubt.

Always cache the list element when used more then once in the loop.

DO

```
private void Method()
{
  for (int i = 0; i < _list.Count; ++i)
  {
    int element = _list[i];
    Calculation1(element);
    Calculation2(element);
  }
}
```

DON'T

```
private void Method()
{
  for (int i = 0; i < _list.Count; ++i)
  {
    Calculation1(_list[i]);
    Calculation2(_list[i]);
  }
}
```

Use unity `NonAlloc` methods whenever possible (`RaycastNonAlloc`, `SphereCastNonAlloc`, ...).
In general avoid the methods that return an array because it's most likely a copy.

DO

```
private void Method()
{
  for (int i = 0; i < Input.touchCount; ++i)
  {
    Touch touch = Input.GetTouch(i);
  }
}
```

DON'T

```
private void Method()
{
  for (int i = 0; i < Input.touches.Count; ++i)
  {
    // ...
  }
}
```

Try to reuse arrays and lists instead of recreating them.

DO

```
private static RaycastHit [] _collissions = new RaycastHit[128];

private void CheckCollision()
{
  int count = Physics.SphereCastNonAlloc(origin, radius, direction,
_collisions);

  for (int i = 0; i < count; ++i)
  {
    var collision = _collisions[i];
  }
}
```

DON'T

```
private void CheckCollision()
{
  RaycastHit [] _collissions = new RaycastHit[128];
  Physics.SphereCastNonAlloc(origin, radius, direction, _collisions);

  for (int i = 0; i < _collisions.Length; ++i)
  {
    var collision = _collisions[i];
    if (collision == null)
      continue;
  }
}
```

Always cache animator parameters, shader properties ids, layer mask ids.

```
protected static readonly int AnimationHash = Animator.StringToHash
("Walk");

private void PlayAnimation()
{
  _animator.SetFloat(AnimationHash, 0);
}
```

Always turn off the `RaycastTarget` toggle on any graphic component when not needed.

Put UI that is updated very often in an extra canvas, each update of an element causes a redraw of the entire canvas it is located in. Do not overdo this, because to much canvases adds extra overhead.

Always fill in the input camera field in a world canvas.

Cache `GetComponents`, never use it in an update.

```
public void Awake()
{
  _myScript = GetComponent<myScript>();
}
```

Prefer `TryGetComponent` over `GetComponent` as `GetComponent` allocates in the editor, which will impact your profiling results when profiling in editor.

Avoid `Find` and `FindObjectOfType`, these are very heavy for performance.

Consider pooling instead of destroying and recreating objects.

Cache `Camera.main`, it uses `FindObjectWithTag`. This is fixed as of Unity version 2019.2, but still do it anyway.

Good practices

Do not use the keyword var. Except when the creation is on the same line as the declaration or in a `foreach` for dictionaries. It needs to be clear what the type is without needing to do research into the code.

DO

```
var class = new MyClass();

var myDictionary = new Dictionary<string, Vector3>();
foreach (var kvp in myDictionary) { ... }
```

DON'T

```
var x = GetMyValue();
```

Editor fields should always be `private` and have the `[SerializeField]` property. If you want to access the field in other classes, make a property for it. This way when a variable is giving errors you know the problem is in the inspector.

DO

```
#region Editor Fields

[SerializeField]
private UILabel _nameLabel;

#endregion
```

DON'T

```
#region Editor Fields

public UILabel NameLabel;

#endregion
```

Avoid writing very long methods. If a method becomes to long, you must consider refactoring into separate methods.

DO

```
public void OnGUI()
{
        DrawUserRegion();
        DrawAddressRegion();
        DrawPictureRegion();
}
```

DON'T

```
public void OnGUI()
{
        // 300 lines of GUI code.
}
```

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

DO

```
string name;
int age;
object contactInfo;
```

DON'T

```
String name;
Int16 age;
Object contactInfo;
```

Don’t use hardcoded numbers. Use constants. Declare constants in the top of the file in the fields region or inside the method and use them. However, using constants isn’t always the best practice either. You should load constants from a config file, database or `ScriptableObject` so that you can change them later. Only declare them as constants if you are sure their value will never need to be changed or when it makes sense.

When working with id’s, declare an invalid id that is a negative number.

```
public const long InvalidUserId = -2341L;
```

Don’t over use member variables. Only declare a new one if you really need it. Avoid member variables with double information. For example: if only one object in a list can be active at a single time, store a member variable that contains which object in the list is active. Don’t give each individual object a member variable `_isActive` to indicate whether that object is active or not. This can lead to inconsistencies and hard to find bugs.

DO

```
List<UiPanel> _uiPanels;
UiPanel _activePanel;
```

DON’T

```
public class UiPanel
{
        private bool _isActive;
}
```

Use enums. Don’t use numbers or strings to indicate discrete values.

DO

```
public enum MailType
{
        Html,
        PlainText
}

public void SendMail(string message, MailType mailType)
{
        switch (mailType)
        {
                case MailType.Html:
                        // Do something.
                        break;

                case MailType.PlainText:
                        // Do something.
```

```
                        break;

                default:
                        // Do something.
                        break;
        }
}
```

DON'T

```
public void SendMail(string message, string mailType)
{
        switch (mailType)
        {
                case "Html":
                        // Do something.
                        break;

                case "PlainText":
                        // Do something.
                        break;

                default:
                        // Do something.
                        break;
        }
}
```

Don't make member variables `public` or `protected`. Keep them private and expose `public` or `protected` Properties. Use the member variable in the class where it is defined, unless the setter contains extra code.

DO

```
private int _userId = InvalidUserId;

public int Userid
{
  get { return _userId; }
  set
  {
    _userId = value;
    // Add extra logic that needs to happen when the user id is set,
here.
  }
}
```

DON'T

```
public int UserId;
```

An exception are data structs. Public fields are allowed when a struct only has some fields to hold data.

DO

```
public struct PlayerData
{
        public string Id;
        public string Name;
}
```

DON'T

```
public struct Player
{
        public string Id;
        public string Name;

        public void ChangeName(string NewName)
        {
                Name = NewName;
        }
}
```

Use automatic properties where possible.

DO

```
public bool HighScoresLoaded { get; private set; }
```

DON'T

```
private bool _highScoresLoaded;

public bool HighScoresLoaded
{
```

```
        get { return highScoresLoaded; }
        private set { highScoresLoaded = value; }
}
```

Use namespaces.

```
namespace ProjectName.Folder.Structure
{
  ...
}
```

Make all unity methods `protected`. This way you will receive a warning when inheriting the class and using the same unity method

```
protected void Start()
{
  ...
}
```

Never hardcode a path or drive name in code. Get the application path programmatically and use relative paths. Never assume that your code will run from drive `C:`. Some users may run it from a network path or from `D:`.

```
string relativePath = "Content/Data/Languages/en.xml";
string fullPath = Application.DataPath + relativePath;
```

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.

DO

```
public struct WallPostArguments
{
        public string Message;
        public string Link;
        public string Picture;
        public string Name;
        public string Caption;
        public string Description;
}

public bool MakeWallPostToProfile(long userId, WallPostArguments
wallPostArguments)
{
        ...
}
```

DON'T

```
public bool MakeWallPostToProfile(long userId, string message, string
link, string picture, string name, string caption, string description)
{
        // ...
}
```

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.` Or use error codes/numbers.

When displaying error messages, in addition to telling what is wrong, the message should also tell what the user should 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.`

Show a short and friendly message to the user. But log the actual error with all possible information. This will help a lot in diagnosing problems.

Do not have more than one class in a single file. Some exceptions are: comparer classes, event argument classes, …

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. Even better: whenever possible use the `using(...)` keyword as it does all of this automatically.

```
using(FileStream stream = ...)
{
        stream.Read(...);

        stream.Close();
}
```

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.

Always use `/` in paths and urls, never `\`. Be consistent about this, so that when you have to do a find (eg to get a filename out of a full path), you know what to look for.

When concatinating parts of a path, use `Path.Combine`, this will make sure you don't end up with double directory separators like `/Content //Data/Languages/en.xml`

```
string hostname = "www.triangle-factory.be";
string relativePath = "Content/Data/Languages/en.xml";

DownloadFile(relativePath);
public File DownloadFile(string relativePath)
{
  string fullUrl = hostname;
  if (relativePath.StartsWith('/') == false)
  {
        fullUrl += "/";
  }
```

```
  fullUrl += relativePath;

  // Download file
  ...
}
```

Use the conditional operator and the coalescing operator. Don't use the coalescing operator for Unity types! They will not work for `GameObject`. Since the `null` check is not an actual `null` on the object but on an underlying C++ pointer.

```
private string _email = string.Empty;

public string Email
{
        get { return _email; }
        set { _email = value ?? string.Empty; }        // This way
_email will never be null.
}
```

Always initialize variables. If possible, initialize them at the same time as their declaration. The only exceptions are member variables that have to be initialized to their default value (0, null or false).

DO

```
private bool _isValid;
private bool _isActive = true;

private int _counter;
private int _counter2 = 1;

private string _name = string.Empty;

private Vector3 _position = Vector3.zero;
private Quaternion _rotation = Quaternion.identity;

private List<string> _names;
private List<string> _names2 = new List<string>();
```

DON'T

```
private bool _isValid = false;

private int _counter = 0;

private string _name = null;
```

```
  private string _name = "";
  private string _name;

  private Vector3 _position;

  private Quaternion _rotation;

  private List<string> _names = null;
```