# Contributing

We are keen for developers to contribute to open source projects to keep them great!

Whilst every effort is made to provide features that will assist a wide range of development cases, it is inevitable that we will not cater for all situations.

Therefore, if you feel that your custom solution is generic and would assist other developers then we would love to review your contribution.

There are however, a few guidelines that we need contributors to follow so that we can have a chance of keeping on top things.

## Getting Started

\* Make sure you have a GitHub account.

\* Create a new issue on the GitHub repository, providing one does not already exist.

\* Clearly describe the issue including steps to reproduce when it is a bug (fill out the issue template).

\* Make sure you fill in the earliest version that you know has the issue.

\* Fork the repository on GitHub.

## Making Changes

\* Create a topic branch from where you want to base your work.

\* If you're fixing a bug then target the `master` branch.

\* If you're creating a new feature then target a release branch.

\* Name your branch with the type of issue you are fixing; `feat`, `chore`, `docs`.

\* Please avoid working directly on your master branch.

\* Make sure you set the `Asset Serialization -> Mode` parameter in `Main Menu -> Edit -> Project Settings -> Editor` to `Force Text`.

\* Make commits of logical units.

\* Make sure your commit messages are in the proper format.

Following the above method will ensure that all bug fixes are pushed to the `master` branch while all new features will be pushed to the relevant next release branch. This means that patch releases are much easier to do as the `master` branch will only contain bug fixes so will be used to fork into new patch releases. Then master will be re-based into the relevant next release branch so the next release also contains the updated bug fixes in the previous patch release.

## Coding Conventions

To ensure all code is consistent and readable, we adhere to the default coding conventions utilized in Visual Studio. The easiest first step is to auto format the code within Visual Studio with the key combination of `Ctrl + k` then `Ctrl + d` which will ensure the code is correctly formatted and indented.

Spaces should be used instead of tabs for better readability across a number of devices (e.g. spaces look better on GitHub source view.)

In regards to naming conventions we also adhere to the [standard .NET Framework naming convention system](https://msdn.microsoft.com/en-gb/library/x2dbyw72(v=vs.71).aspx) and use American spellings to denote classes, methods, fields, etc.

> \*\*Incorrect:\*\*

```

public Color fontColour;

```

> \*\*Correct:\*\*

```

public Color fontColor;

```

All core classes should be within the relevant `VRTK` namespace. Any required `using` lines should be within the namespace block.

> \*\*Incorrect:\*\*

```

using System;

namespace X.Y.Z

{

public class MyClass

{

}

}

```

> \*\*Correct:\*\*

```

namespace VRTK.X.Y.Z

{

using System;

public class MyClass

{

}

}

```

Names of classes should always be a noun describing what the component's role is.

> \*\*Incorrect:\*\*

```

public class UpdateThing;

```

> \*\*Correct:\*\*

```

public class ThingUpdater;

```

Names of methods should always be a verb describing the action that the method will carry out. Methods should also do a specific task which the name clearly highlights. Any use of the word `And`/`Or` in a method name highlights the potential issue that a method is doing too much.

> \*\*Incorrect:\*\*

```

public virtual void ContainerPosition();

public virtual void ContainerPositionAndRotation();

public virtual void ContainerPositionOrRotation();

public virtual void Setup();

```

> \*\*Correct:\*\*

```

public virtual void UpdateContainerPosition();

public virtual void UpdateContainerRotation();

public virtual void SetUpContainer();

```

The only deviation from the MSDN conventions are fields should start with lowercase -> `public Type myField` as this is inline with the Unity software naming convention.

Names of fields and properties should always be an adjective with a clear understanding of what the variable is describing. It is also not necessary to repeat the type name in the variable name as it should be implicitly understood from the naming of the variable.

> \*\*Incorrect:\*\*

```

public GameObject go;

public GameObject jointGameObject;

```

> \*\*Correct:\*\*

```

public GameObject jointContainer;

```

Inline variable declarations should also follow the same naming rules and be clear and concise to their purpose. They should not be abbreviated which can cause confusion whilst reading the code.

> \*\*Incorrect:\*\*

```

for (int i = 0; i < 10; i++) {}

for (int x = 0; x < 10; x++)

{

for (int y = 0; y < 10; y++) {}

}

```

> \*\*Correct:\*\*

```

for (int index = 0; index < 10; index++) {}

for (int containerIndex = 0; containerIndex < 10; containerIndex++)

{

for (int clipIndex = 0; clipIndex < 10; clipIndex++) {}

}

```

Class methods and parameters should always denote their accessibility level using the `public` `protected` `private` keywords. Methods should also always be defined as virtual in concrete classes to allow overriding. Marking as `protected` is favored until there is good reason to make it `private`. This ensures the type is open for extension where possible.

> \*\*Incorrect:\*\*

```

void MyMethod()

```

> \*\*Correct:\*\*

```

protected virtual void MyMethod()

```

The order elements are defined in a class should be as follows:

\* nested classes.

\* public enums.

\* serialized (public or protected/private + `[SerializeField]`) fields.

\* public events.

\* public properties.

\* protected enums.

\* protected fields.

\* protected properties.

\* private fields.

\* private properties.

\* public MonoBehaviour message methods.

\* public methods.

\* protected MonoBehaviour message methods.

\* protected methods.

\* private MonoBehaviour message methods.

\* private methods.

It is acceptable to have multiple classes defined in the same file as long as the subsequent defined classes are only used by the main class defined in the same file. However, don't nest types as it makes it harder to find and reference them.

Where possible, the structure of the code should also flow with the accessibility level of the method or parameters. So all `public` parameters and methods should be defined first, followed by `protected` parameters and methods with `private` parameters and methods being defined last.

Blocks of code such as conditional statements and loops should contain the block of code in braces `{ }` even if it is just one line.

> \*\*Incorrect:\*\*

```

if (this == that) { do; }

```

> \*\*Correct:\*\*

```

if (this == that)

{

do;

}

```

The exceptions to this are:

\* When defining an empty Method.

\* When defining Property get/set fields that only require one line.

> \*\*Example:\*\*

```

public virtual void EmptyMethod() {}

public Type MyType

{

get { return \_backingType; }

set

{

\_backingType = value;

ConfigureMyType(\_backingType);

}

}

```

Any method or variable references should have the most simplified name as possible, which means no additional references should be added where it's not necessary.

> `this.transform.rotation` \*is simplified to\* `transform.rotation`

> `GameObject.FindObjectsOfType` \*is simplified to\* `FindObjectsOfType`

All MonoBehaviour inherited classes that implement a MonoBehaviour [Message](https://docs.unity3d.com/ScriptReference/MonoBehaviour.html) method must at least be `protected virtual` to allow any further inherited class to override and extend the methods.

When defining arrays, it is appropriate to instantiate the array with the `System.Array.Empty<T>()` notation instead of doing `new Array[0]`.

All events should be UnityEvents and not C# delegates as this will automatically provide inspector helpers for attaching event listeners.

When the actual UnityEvent is defined in code it should also be instantiated so it is not null at runtime.

`public MyEventClass MyEventAction = new MyEventClass();`

Public fields that are collections should favor the `System.Collections.Generic.List` data type over a simple `array` as the `List` offers helper operators for mutating the contents of the list (e.g. `Add` `Remove` `Clear`). This makes it easier when accessing the collection via another script when the contents of the collection requires changing.

## Documentation

All core scripts, abstractions, controls and prefabs should contain inline code documentation adhering to the [.NET Framework XML documentation comments convention](https://msdn.microsoft.com/en-us/library/b2s063f7.aspx)

All classes, methods and UnityEvents should be documented using the XML comments and contain a 1 line `<summary>` with any additional lines included in `<remarks>`.

Public serialized parameters that appear in the inspector also require XML comments and an additional `[Tooltip("")]` which is displayed in the Unity Editor inspector panel. All parameter attributes should appear in the same attribute block and comma separated with the `Tooltip` attribute first followed by the remaining attributes in an order that makes most sense when read as a sentence. The order of documentation and parameter attributes should be:

\* XML documentation.

\* Parameter attributes.

> \*\*Example:\*\*

```

/// <summary>

/// Description here.

/// </summary>

[Tooltip("Description here."), InternalSetting, SerializeField]

protected Type myType;

```

If a `[Header]` attribute is to be used to group fields in the Unity Inspector, then the attribute must precede the `Tooltip` attribute in the comma separated list and not be on it's own line above the XML summary as the MSDN specification dictates that attributes must annotate members. It is also advised to wrap `Header` sections using the `#region` directive to denote the block associated. The following shows a valid and invalid example:

> \*\*Incorrect:\*\*

```

[Header("My Header")]

/// <summary>

/// Description here.

/// </summary>

[Tooltip("Description here."), InternalSetting, SerializeField]

protected Type myType;

```

> \*\*Correct:\*\*

```

#region My Header

/// <summary>

/// Description here.

/// </summary>

[Header("My Header"), Tooltip("Description here."), InternalSetting, SerializeField]

protected Type myType;

#endregion

```

Whenever an inherited field, method, etc. is overridden then the documentation should avoid repeating itself from the base class and instead should use the `/// <inheritdoc />` notation instead. Despite not being needed, being explicit about inheriting the documentation simplifies checking contributions for proper documentation.

Any references to other classes, methods or parameters should also use the `<see>` notation within the documentation.

## Commit Messages

All pull request commit messages are automatically checked using [GitCop](http://gitcop.com) - this will inform you if there are any issues with your commit message and give you an opportunity to rectify any issues.

The commit message lines should never exceed 72 characters and should be entered in the following format:

> \*\*Example:\*\*

```

<type>(<scope>): <subject>

<BLANK LINE>

<body>

<BLANK LINE>

```

### Type

The type must be one of the following:

\* feat: A new feature.

\* fix: A bug fix.

\* docs: Documentation only changes.

\* refactor: A code change that neither fixes a bug or adds a feature.

\* perf: A code change that improves performance.

\* test: Adding missing tests.

\* chore: Changes to the build process or auxiliary tools or libraries such as documentation generation.

### Scope

The scope could be anything specifying the place of the commit change, such as, `Controller`, `Interaction`, `Locomotion`, etc...

### Subject

The subject contains succinct description of the change:

\* use the imperative, present tense: "change" not "changed" nor "changes".

\* don't capitalize first letter, unless naming something, such as `Bootstrap`.

\* no dot (.) at the end of the subject line.

### Body

Just as in the subject, use the imperative, present tense: "change" not "changed" nor "changes" The body should include the motivation for the change and contrast this with previous behaviour. References to previous commit hashes is actively encouraged if they are relevant.

> \*\*Incorrect commit summary:\*\*

```

Added feature to improve teleportation

```

> \*\*Incorrect commit summary:\*\*

```

feat(Teleport): Add feature

```

> \*\*Incorrect commit summary:\*\*

```

feat(my-teleport-feature): my feature.

```

> \*\*Correct commit summary:\*\*

```

feat(Teleport): add fade camera option on teleport

```

## Submitting Changes

\* Push your changes to your topic branch in your repository.

\* Submit a pull request to this repository.

\* The core team will aim to look at the pull request as soon as possible and provide feedback where required.

## Copyright Notices

Do not include any copyright notices in any files committed to the repository. As the author of the commit you will continue to retain the copyright for the code committed but do so under the license stated in the repository as outlined in the [GitHub Terms Of Service](https://help.github.com/en/articles/github-terms-of-service#6-contributions-under-repository-license).