The OpenTK Style Guide

The OpenTK Maintenance Team
https://opentk.github.io/

Jarl Gullberg
jarl.gullberg@gmail.com

October 15, 2018
Version 1.0.0

DRAFT

Contents

DRAFT

DRAFT

List of Figures

DRAFT

Foreword

This document outlines the style and code conventions used in the OpenTK project. While the codebase is an old, very large chunk of code, all new additions or alterations to the project must follow these guidelines. It is intended as an exhaustive list of all conventions used in OpenTK and its related projects, and may be changed as the conventions evolve or new cases to take into consideration appear.

It's important for any and all contributions to adhere to this set of guidelines in order to maintain a readable, consistent, and maintainable codebase. It's recommended that pull requests submitters take a few minutes and leaf through the document before submitting their pull requests to minimize review times.

Beyond this document, a significant number of rules are implemented using StyleCop.Analyzers. These rules are verified on-the-fly if you're using an editor that supports them, and will be raised as compiler errors if they are detected in the code. Refer to the "stylecop.rules" file for an exhaustive list, and the StyleCop documentation for descriptions of each rule.

The document is divided into several chapters, each dealing with a specific area of the guidelines. See the table of contents for an exhaustive list.

DRAFT

0.1 Rule Types

Within each chapter, a set of rules will be listed. Each rule's definition will be described, and, if applicable, an example will be given. Rules may optionally be accompanied by a C# compiler warning code (either a builtin code, or one provided by an external analyzer such as StyleCop), which refers to the compiler warning or error that will be raised in case of a detected violation.

Each rule falls into one of the following categories.

0.1.1 Must

Rules in this category must be followed without exception. Any pull request, code alteration, or contribution that breaks a rule in this category will be rejected until the issue has been rectified.

0.1.1 EX0001: A rule that must be followed

The description of the rule.

0.1.2 Should

Rules in this category should be adhered to as closely as possible, but may have exceptions or contextual variations. Typically, the rule should be applied as a *must* rule, but violations will not result in an outright rejection. Violations will, however, be subject to close review.

0.1.2 EX0002: A rule that should be followed

The description of the rule.

0.1.3 May

Rules in this category are non-critical, and are typically characterized as general guidelines and not hard rules. They're nice to have, and most likely lead to better or more readable code, but they can be ignored.

0.1.3 An optional rule

The description of the rule.

DRAFT

Rules & Guidelines

DRAFT

1 Project Structure

1.0.1 Files should be organized according to their namespace

Files shall be placed in subdirectories which map to the namespace they are part of.

For example, if the file contains a type with the namespace `OpenTK.Platform.X11`, it shall reside in `OpenTK/Platform/X11`. Subfolders within a major namespace may be used for organizational purposes if creating another namespace for that folder would add verbosity without adding useful separation. For example, `OpenTK/XML` might contain the folders `OpenTK/XML/Interfaces` and `OpenTK/XML/Implementations`. These folders are there for organizational purposes, but would only complicate the namespace tree if the above rule was enforced.

1.0.2 Files without a namespace should be placed correctly

If the file does not contain a type or a namespace, it shall be placed in a folder relevant to its usage or perceived namespace - That is, the following example an XML file which contains localizations would be placed in a folder named ‘Localizations‘.

DRAFT

2 Formatting

2.1 General

This section contains some general rules that apply to all files in the project - text, source code, or data. They deal with generic requirements and restrictions that are language-agnostic, and serve to facilitate easier cross-editor and cross-platform support.

2.1.1 Line width must not exceed 120 columns

Any line that is longer than 120 columns (documentation, code, or otherwise) must be wrapped at the nearest word boundary and continue on the next line.

A "word boundary" is mentioned here. To clarify, a word boundary is any point at which a typical word starts or ends (spaces, commas, punctu-

ation, et cetera), as well as programmatic word boundaries - the end of an identifier, the end of an assignment, and so on. You should wrap as close as possible to the point at which you need to wrap, unless another rule applies.

2.1.2 All files must end with a line feed character

In accordance with the POSIX specification, a line is defined as a string of characters ending with a line feed character. As such, all files must end with a single line feed character.

2.1.3 Files must be saved as UTF-8

Files shall be encoded in 'UTF-8' with no byte order mark.

2.1.4 Lines must end with a line feed character

Lines shall be terminated by a single line feed character. Under no circumstances shall a return carriage character occur in a file, except as a part of a string literal.

DRAFT

2.2 Braces, Parens, and Brackets

2.2.5 Braces must be formatted correctly

OpenTK exclusively uses the Allman brace style. No other brace styles are allowed.

As an example to the above rule, the following style is valid, while the subsequent is not.

```
foreach (var item in list)
{
    // ...
}
```

Listing 1: Correct formatting

```
foreach (var item in list) {
    // ...
}

foreach (var item in list) { // ... }
```

Listing 2: Incorrect formatting

This rule extends to formatting of method calls and various initializers when they need to be split over more than one line. OpenTK uses a similar block-based style when method calls need to be wrapped.

2.2.6 Method calls and initializers must be wrapped correctly

When a method call or initializer must be wrapped, it should be wrapped using block-style formatting, similar to the Allman style for brace layout.

2.2.7 Wrapped method calls must be aligned correctly

If a method call chain is wrapped, all method calls (including the first) must be placed separately on their own lines. If a single method call is wrapped, it must remain on the same line as the object it is invoked on, except when remaining exceeds the maximum line width.

DRAFT

```
var myArray = new[]
{
    1,
    2,
    3,
    4,
    5,
    6
};
```

Listing 3: Wrapping an initializer

```
var linqResults = myCollection
    .Where(x => x > 10)
    .Select
    (
        x =>
            new OtherType(x, x * 2);
    );

var otherResults = myCollection.Select
    (
        x =>
        {
            var myThing = DoThingWith(x);
            var myOtherThing = DoOtherThing(x);

            return new OtherType(x, x * 2);
        }
    );
```

Listing 4: Wrapping a method call

Notice how the call to `Select` is wrapped. Instead of sharing the opening parens with the call, it is placed on a new line. This also extends to method declarations.

2.2.8 Method declarations must be wrapped correctly

When a method declaration must be wrapped, its parameters should be wrapped using block-style formatting, similar to the Allman style for brace layout.

DRAFT

```
public void MyBigMethod
(
    int index,
    MyLongClassNameThatKeepsGoingAndEatsSpace instance,
    ActionType action
)
{
    // ...
}
```

Listing 5: Wrapping a method declaration

In the majority of cases, having methods that do not require excessively long calls is preferable.

2.2.9 Braces must not have additional blank lines

Opening braces may not be preceded or succeeded by a blank line. Closing braces must be succeeded by a blank line, unless the next line contains another closing brace. If that is the case, then it must not be followed by a blank line.

That is, the following example

```
if (myThing)
{
    DoThing();
}

AnotherThing();

// and
public void MyMethod()
{
    if (myThing)
    {
        return;
    }
}
```

Listing 6: Good formatting

is to be preferred over

DRAFT

```
if (myThing)
{

    DoThing();
}

AnotherThing();

// and
public void MyMethod()
{
    if (myThing)
    {

        return;

    }

}
```

Listing 7: Extraneous lines

3 Naming

3.1 Files

3.1.1 Files must be named correctly

The file ending shall be in all lower case, whereas the name of the file shall match the name of the type exactly. A single punctuation mark shall separate the name and file extension. No more than one punctuation mark may be present in the filename.

3.1.2 Filenames should match their contained type

Files should share the name of the type they contain, excluding the namespace. This includes generic types - no extra type information should be annotated in the file name.

In some cases, there may exist both a generic and a non-generic declaration of a type. In this case, the file containing the nongeneric type takes precedence, and the following rule comes into effect.

DRAFT

3.1.3 Names of files containing a generic type may be suffixed

If required, the name of a file that contains a generic type may be suffixed by the names of the generic arguments to the class, held within a set of braces.

As an example, given two type definitions,

```
public interface ISomeInterface { }
public interface ISomeInterface<in TInput, out TOutput> { }
```

Listing 8: Two declarations with the same name

would be placed into `ISomeInterface.cs` and `ISomeInterface{TInput, TOutput}.cs`, respectively.

3.2 Types and Members

While most of these naming rules are covered by StyleCop, they still bear mentioning here - StyleCop is not entirely exhaustive, and some of these rules are not covered.

3.2.4 Identifiers must not contain underscores

Identifiers (type names, variable names, enumeration members, etc) must not contain underscores, except as overridden by another rule.

3.2.5 Types must be named correctly

All types must be named according to `PascalCase`.

3.2.6 Fields must be named correctly

Instance fields must be prefixed with the underscore (_) character, and be named according to `camelCase`. Static fields must *not* be prefixed with an underscore, and must be named according to `PascalCase`.

3.2.7 Methods must be named correctly

All methods (including local methods) must be named according to `PascalCase`.

DRAFT

3.2.8 Properties must be named correctly

All properties must be named according to `PascalCase`.

3.2.9 Locals must be named correctly

All locals (local variables, method parameters, lambda arguments, etc) must be named according to `camelCase`.

3.2.10 Named `ValueTuple` members must be named correctly

Named `ValueTuple` members are, for all intents and purposes, considered public properties on a class. Therefore, they must be named according to `PascalCase`.

3.2.11 Abbreviations must use correct casing

In accordance with the standard library, abbreviations must be fully capitalized if they are two characters or less; in all other cases, they must be PascalCased. For instance, use "ID", but also "Http".

DRAFT

4 Declarations

4.0.1 Declarations must be correctly ordered

Members in a type declaration must be declared in the following order:

- Public fields
- Private fields
- Public properties
- Private properties
- Constructors
- Public methods
- Private methods
- Overridden methods
- Interface implementations

Any static declaration must come last in its corresponding section. Within a section, there is no sorting order. Apply common sense.

4.0.2 Types must be declared within a namespace

Any type declaration must appear inside a namespace.

DRAFT

4.1 Methods

4.1.3 Methods should be private

Whenever possible, methods should be declared private, or public as part of an internal class.

In order to reduce the public API surface of the library beyond what is actually maintained, the set of methods exposed to the end user should be as small as possible.

4.2 Properties

4.2.4 Properties should be auto-properties

Whenever possible, a property's accessors should be automatically implemented.

4.2.5 Auto-properties must be declared on one line

If a property is declared as an auto-property, it must be declared on one line.

That is, the following example

```
public bool IsCorrect { get; set; }
```

Listing 9: One line

is to be preferred over

```
public bool IsCorrect
{
    get;
    set;
}
```

Listing 10: Multiple lines

4.2.6 Properties should be expression-bodied

Whenever possible, a property should be expression-bodied.

That is, the following example

```
public int GetSomeValue => _internalValue * 5;
```

Listing 11: One line

is to be preferred over

DRAFT

```
public bool IsCorrect
{
    get
    {
        return _internalValue * 5;
    }
}
```

Listing 12: Multiple lines

5 Documentation

5.0.1 Private members should be documented

Whenever possible, the internal API surface should be documented.

In order to help the next person along, all members should be documented - not just the members exposed to the end user.

5.0.2 Files must have the correct header

Files shall contain a file header, containing the following text.

DRAFT

```
//
//  <filename>
//
//  Copyright (C) <year of creation> OpenTK
//
//  This software may be modified and distributed under
    the terms
//  of the MIT license. See the LICENSE file for details.
//
```

Listing 13: File header

The file header is configured and checked by StyleCop. The most recent version is included in this guide, and will always accompany this rule.

6 Language Features

6.1 Declarations

6.1.1 `var` should be used whenever possible

Implicitly typed local variables should be used whenever possible, in order to reduce typing and code verbosity.

While `var` is useful, it may sometimes obscure valuable information, and an explicit type might be a better fit.

In general, an explicit type should be used over `var` if

- Knowing the type is explicitly required, or
- Knowing the type is valuable or particularly useful in understanding the surrounding code

In all other cases, however, using implicit typing is preferred.

6.1.2 `out` variables should be inlined

Whenever possible, `out` variables should be inlined into their respective method calls.

That is, the following example

```
MyMethod(out var myThing);
```

Listing 14: Inlined out

is to be preferred over

```
MyType myThing;
MyMethod(out myThing);
```

Listing 15: External out

6.1.3 Object initializers should be used

Whenever possible, object initializers should be used instead of assigning members after the fact.

DRAFT

That is, the following example

```
var myThing = new MyThing
{
    MyProp = true,
    MySecondProp = 42
};
```

Listing 16: Object initializer

is to be preferred over

```
var myThing = new MyThing();
myThing.MyProp = true;
myThing.MySecondProp = 42;
```

Listing 17: Post-construction assignments

6.2 Invocations

6.2.4 Events should be directly invoked

Events shall not have wrapper methods for invoking them unless absolutely required. Instead, null-propagation operators shall be used to invoke them directly.

That is, the following example

```
MyEvent?.Invoke();
```

Listing 18: Direct invocation

is to be preferred over

```
private void OnMyEvent()
{
    MyEvent?.Invoke();

    // or

    if (!(MyEvent is null))
    {
        MyEvent();
    }
}

OnMyEvent();
```

DRAFT

6.3 Null Checking

Null checking has evolved with the advent of C# 7.0, allowing us to use pattern matching to detect null values. In the ongoing move to treat null as a more and more exceptional value, this is a very good thing, and lets us check for null in a faster, safer way.

6.3.5 Null should be treated as an exceptional value

`null` is to be treated as an exceptional value, and should not be a typical return value. Methods should reject null as valid input parameters to the broadest extent possible, and should not return null themselves. In the instances where a method must return or accept null, it should be explicitly documented, and obviously implemented.

6.3.6 Null should be checked using pattern matching

To assist in highlighting `null`'s exceptionality, and to avoid overridden equality operators, null checking shall be performed using pattern matching.

That is, the following example

```
bool result = value is null; // true
bool result = !(value is null); // false
```

is to be preferred over

```
bool result = value == null;
bool result = value != null;
```

DRAFT