DHX Interactive Coding Standards

C# Guidelines

#### *Prepared by Julian Spillane*

#### *Contributions from Paul Forest Document Version 1.02 January 2014*

Introduction

This coding standards document describes the rules and recommendations for developing applications and libraries using the C# language and .NET specifications, specifically as they apply to the Unity3D development environment. The goal is to define guidelines to enforce consistent style, formatting and nomenclature to increase readability and avoid common mistakes and pitfalls.

# Scope

This document only applies to the C# and .NET as they apply to Unity3D development. For additional DHX coding standards please see the respective language’s document.

Also note that while standards for curly-braces and white-space tend to be very personal and wildly variable, these topics are addressed to ensure greater consistency across our various codebases.

Naming Conventions

Consistency is the key to code maintainability. This is most true when it comes to names for projects, source files and identifiers.

# General Guidelines

1. Always use Pascal Case[1](http://c2.com/cgi/wiki?PascalCase) for naming, with the exception of constants.
2. Do not create declarations of the same type (namespace, class, method, structure, property, field, or parameter) and access modifier (private, public, protected) that vary only by capitalization.
3. Do not use names that begin with a numeral.
4. Do not add numeric suffixes to identifier names
5. Prepend all non-public class members with underscores (e.g. private int \_FrameCounter).
6. Explicitly declare access modifier of every class method, property and member variable.
7. Always choose meaningful and specific names.

Do not use short cryptic names or names based on internal jokes. It should be easy to type a name without looking up how it is spelled.

Exception: Loop variables and variables with a small scope (less than 20 lines) may have short names to save space if the purpose of that variable is obvious.

1. Only use English names.
2. Variables with large scope should be verbosely named, while variables with small scope can have shorter names.

Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables shall be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are i, j, k, m, n and for characters c and d.

1. Do not use Hungarian Notations: **e.g.** **strName** or **iCount**
2. Avoid using abbreviations unless the full name is excessive.
3. Avoid abbreviations longer than 5 characters: **e.g.** OMFGBBQStorage
4. User upper-case for two-letter abbreviations and Pascal Case for longer.
5. Do not use reserved words as names.
6. Avoid naming conflicts with standard library methods.
7. Avoid adding redundant or meaningless prefixes and suffixes to identifiers: **e.g.**

enum GameStateEnum { … }  
struct MyCoolStruct { … }

1. Do not include the parent class / struct name within a property name. **E.g.**: Object.Transform NOT Object.ObjectTransform

# Entity Naming Specifics

|  |  |
| --- | --- |
| Identifier | Naming Convention |
| Project Files | Pascal Case. Try to match project code-name and intended executable name. Abbreviations can be used for brevity.  **Example:**  The Doozers Podmobile => DoozersPodmobile |
| Source File | Pascal Case.  Always match *class* name and file name.  Avoid including more than one *class* per file. Use a descriptive file name when defining multiple *classes*.  **Example:**  Foo.cs => class Foo { … }; |
| Namespace | Pascal Case.  If used, try to match project or library name.  **Example:**  namespace YGGRetrocade { … } |
| Class | Pascal Case.  Use a noun or noun-phrase for class names.  Add an appropriate suffix when sub-classing where possible.  **Examples:**  class Foo {…}; class SpecializedFoo : public Foo {…}; |
| Struct | Pascal Case.  Use a noun or noun-phrase for struct names.  **Example:**  struct Bar {…}; |
| Interfaces | Pascal Case.  Always prefix abstract (interface) classes with capital “I”.  **Example:**  interface IEquatable<T>  {  bool Equals(T obj);  }; |
| Property | Pascal Case.  If used to provide access to a member variable, match the name. Never prefix property names with “Get” or “Set”.  **Example:**  class Foo  {  private string \_Name;  public string Name  {  get { return \_Name; }  set { \_Name = value; }  }  } |
| Generic Class | Always use a single capital letter, such as T or K.  **Example:**  class Bar<T>  {  T Foo();  };  Note: closed-constructed types can be appropriate, but do not over-do generic specialization (i.e.) class Bar<T> : BaseBar<int> {} |
| Method | Pascal Case.  Always try to use a verb or verb-object pair.  **Examples:**  public void Activate() {…}  public void UpdateText(string newText) {…} |
| Member Variable | Pascal Case.  Always prefix non-public variables with ‘\_’. Avoid using public variables where possible. Instead, try and use properties or accessor methods.  **Examples:**  private float \_Counter; |
| Constants | Upper Case with words separated by underscores.  Try to be as descriptive as possible.  **Example:**  const float ACCEL\_DUE\_TO\_GRAVITY = -9.80665f; |
| Enum | Pascal Case (both type and options).  Whenever possible assign values to ensure safe casting.  **Example:**  enum LevelMask  {  FireDungeon = (1<<0),  WaterDungeon = (1<<1)  }; |
| Local Variable | Camel Case.  Avoid using single characters where possible.  Avoid enumerating variable names, e.g.: type1, type2, etc.  **Example:**  int lineCounter = 0; |
| Parameter | Camel Case.  **Example:**  void DoSomething(int timesToLoop); |

Coding Style

Every developer has a preference for a specific coding style and rarely are two ever the same. These differences can lead to inconsistency and controversy between developers (this is the leading cause of compile-time sword fights). Therefore, to help ensure maintainability, consistency and readability, the below sections describe the preferred way to implement C# source code.

# Formatting

## Indentation and Spacing

1. Braces must follow the “Exdented Style”’

The Exdented Bracing Style means that the curly brace pairs are lined up with the surrounding statement. Statements and declarations between the braces are indented relative to the braces. **E.g.:**if( \_Field == true )  
{  
 // do something  
}

1. Always enclose conditional and loop sub-statements in curly braces (excepting empty statements i.e.: while(1) {} ).
2. Always use a Tab & Indentation size of **4**.
3. Place each statement on a line of its own. There is no need to conserve vertical space.
4. Declare all variables separately. This can cause confusion, especially when dealing with unsafe code (i.e. pointers). Avoid: int foo = 0, bar = 1, \*fooBar = null;

## Declarations

1. Sequence declarations within structures in the following order:
   1. Enums, constants, structs and internal classes
   2. Member variables
   3. Properties
   4. Methods
2. Sequence declarations based upon access modifier and visibility:
   1. Public
   2. Protected
   3. Private
3. Always explicitly provide the return type of methods.
4. Declare static members at the top of their respective accessibility blocks.
5. Where possible use Named Functions when specifying *delegates*. If it is short, a one or two-line lambda function or anonymous method is acceptable.
6. Always declare inherited functions with override.

An inherited function is implicitly virtual if it is declared virtual in the base class, but will only be called if the derived class is explicitly instantiated. Declaring a method as override will ensure that the appropriately derived method is always called.

1. Avoid the using directive on 3rd-party libraries. In addition to potentially introducing name collisions, this can cause confusion when parsing code without the prefixed namespace.
2. When dealing with resources that implement IDisposable (such as file handles or device contexts), try to use using blocks where possible. This will guarantee proper disposal of the object even if an exception occurs.  
     
   e.g.  
    using (FileStream configFile = File.OpenRead(localFilePath) )  
   {  
    // do stuff  
   }
3. Only use the keyword struct for member variable encapsulation. Avoid methods where possible.
4. When dealing with objects that notify other objects based on stimuli, consider using *events*. Events allow a class (the publisher) to activate methods in other objects (the subscribers) in response to any given action or occurrence. The syntax is as follows:

// in the publisher class  
public event SomeEventDelegate ImportantEvent;  
  
// in a subscriber class  
publisher.ImportantEvent += EventMethod;

For more information see: <http://msdn.microsoft.com/en-us/library/awbftdfh.aspx>

1. When working on libraries, consider using the *internal* keyword to prevent access of otherwise public member variables from external assemblies. This allows you to encapsulate your assemblies without sacrificing access within your library. For more information, see: <http://msdn.microsoft.com/en-us/library/7c5ka91b.aspx>
2. Declare class data private. Data should only be accessed through accessors, class methods and properties. The only exceptions are members that need to be exposed to the *Property Inspector* in *Unity3D*

## Code Commenting

1. All comments **must** be written in English.
2. Use C++-style comments ‘//’ or ‘///’ but never ‘/\* … \*/’
3. Place all inline comments above the line the comment describes, indented identically.
4. Do not “flowerbox” comment blocks. **e.g.**

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// MY COMMENT YAY!!!!!!!!!!  
// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Use inline comments to explain assumptions, known issues and algorithm insights.
2. Do not use inline comments to explain obvious code. Well-written code is self-documenting.
3. Include comments using Task-List keyword flags to allow comment filtering. **E.g.**

// TODO: write stuff here  
// UNDONE: removed Cthulhu-summoning algorithm  
// HACK: temporary fix to prevent Armageddon

1. Always use #ifdef instead of comments to disable blocks of code.
2. Prefix each class with a comment that describes its purpose.
3. Prefix every method that is non-trivial or an accessor with a comment that describes its purpose.

## File Conventions

1. File names must be treated as case sensitive.
2. C# source files must have the extension “.cs”
3. Always put *using* statements at the very top of the file.
4. Never use absolute paths when referencing files.
5. Each file must start with a copyright notice (see: Appendix A)
6. Each file must contain a revision number.

## Statements

1. All switch statements must have a default label. This is to avoid fall-through cases that can cause headaches.
2. Never use do-while loops. Just don’t. They’re hideous and never necessary.

## Assorted

1. Do not use literal numbers other than 0 and 1. Use constants to make the code consistent and easy to maintain. Magic numbers get you into trouble, more often than not.
2. Use plenty of exception checking and assertions!
3. ~~Use prefix increment/decrement instead of postfix when the value of a variable is not used.~~ Do not use any kind of in-line increment/decrement in an expression unless it is part of the conditional for a loop. **e.g.**

int appleCounter = (++i)\*(q++); // **NEVER DO THIS**

1. Do not ever rely on implicit conversion to bool in conditional statements. Always explicitly check the value.

# **Appendix A**

## C# Unity Template File

// <copyright file="#SCRIPTNAME#.cs" company="DHX Media Ltd.">  
// Copyright (c) 2014 All Right Reserved, http://www.dhxmedia.com/  
//  
// All information contained herein is, and remains the property of   
// DHX Media Ltd. The intellectual and technical concepts contained herein  
// are proprietary to DHX Media Ltd. and may be covered by U.S. and Foreign  
// Patents, patents in process, and are protected by trade secret or   
// copyright law. Dissemination of this information or reproduction of this   
// material is strictly forbidden unless prior written permission is obtained   
// from DHX Media Ltd.  
//  
// </copyright>  
// <author>#AUTHOR\_NAME#</author>  
// <email>#AUTHOR\_EMAIL#</email>  
// <date>YYYY-MM-DD</date>  
// <summary>A description of what is contained in this file</summary>  
  
using UnityEngine;  
using System.Collections;  
using System.Collections.Generic;  
  
public class #SCRIPTNAME# : MonoBehaviour   
{  
 // Unity methods  
 private void Start ()   
 {   
 }

private void Update ()   
 {   
 }  
}