STANDARD CONDIFICTION UNITY

Unity uses C# as a base of its scripts, that’s mean that unity doesn’t have its own standard codification; therefore we are going to talk about C# standard codification; however Unity implement a mix of the two standard codification of C# that are pascalcase and camelcase; the next chart shows the conventions of C# is used in unity

|  |  |
| --- | --- |
| TYPE | C# |
| class | PascalCase |
| constant | PascalCase |
| method | PascalCase |
| Namespace/pakage | PascalCase |
| propeties | PascalCase |
| parameter | camelCase |
| Local var | camelCase |
| interphase | PascalCase |

This is a code in unity(example):

using UnityEngine;

using System.Collections;

public class DemoScript : MonoBehaviour {

public Light myLight;

void Update () {

if ([Input.GetKeyDown](http://docs.unity3d.com/Documentation/ScriptReference/Input.GetKeyDown.html" \t "_blank) ("space")) {

MyFunction ();

}

}

void MyFunction () {

myLight.enabled = !myLight.enabled;

}

}

C # Code Conventions

The coding conventions have the following objectives:

They create a consistent appearance in the code, so that readers can focus on the content, not the design.

They allow readers to understand the code more quickly by making assumptions based on previous experience.

They make it easy to copy, change and maintain the code.

They show the recommended procedures of C #.

**Naming conventions**

For short examples that do not include using directives, use namespace qualifications. If you know that a namespace is imported into a project by default, it is not necessary to complete the names of that namespace. Full names can be split after a period (.) If they are too long for a single line, as shown in the following example.

var currentPerformanceCounterCategory = new System.Diagnostics.

PerformanceCounterCategory();

It is not necessary to change the names of objects that were created with the tools of the Visual Studio designer to match other guidelines.

**Design conventions**

A good design uses a format that highlights the structure of the code and makes the code easier to read. Microsoft samples and samples comply with the following conventions:

Use the default Code Editor Settings (automatic indent, 4 character indents, tabs saved as spaces).

Write only one instruction per line.

Write only one statement per line.

If the continuation lines are not automatically indented, do so with a tabulation (four spaces).

Add at least one blank line between the method and property definitions.

Use parentheses so that the clauses of an expression are obvious, as shown in the following code.

if ((val1 > val2) && (val1 > val3))

{

// Take appropriate action.

}

**Comments Conventions**

Place the comment on a separate line, not at the end of a line of code.

Start the comment text with a capital letter.

End the comment text with a period.

Insert a space between the comment delimiter (//) and the comment text, as shown in the following example.

// The following declaration creates a query. It does not run

// the query.

**Language conventions**

The following sections describe the practices that the C # team follows to prepare samples and code samples.

**String (Data type)**

Use string interpolation to concatenate short strings, as shown in the following code.

string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";

To append strings in loops, especially when working with large amounts of text, use a StringBuilder object.

var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";

var manyPhrases = new StringBuilder();

for (var i = 0; i < 10000; i++)

{

manyPhrases.Append(phrase);

}

//Console.WriteLine("tra" + manyPhrases);

**Local variables with implicit assignment of types**

Use implicit types for local variables when the type of the variable is obvious from the right side of the assignment, or when the exact type is not important.

// When the type of a variable is clear from the context, use var

// in the declaration.

var var1 = "This is clearly a string.";

var var2 = 27;

var var3 = Convert.ToInt32(Console.ReadLine());

Do not use var when the type is not evident from the right side of the assignment.

// When the type of a variable is not clear from the context, use an

// explicit type.

int var4 = ExampleClass.ResultSoFar();

Do not trust the variable name to specify the type of the variable. It may not be correct.

// Naming the following variable inputInt is misleading.

// It is a string.

var inputInt = Console.ReadLine();

Console.WriteLine(inputInt);

Avoid using var instead of dynamic.

Use implicit types to determine the type of the loop variable in for and foreach loops.

In the following example, implicit types are used in a for statement.

var syllable = "ha";

var laugh = "";

for (var i = 0; i < 10; i++)

{

laugh += syllable;

Console.WriteLine(laugh);

}

**Unsigned data type**

In general, use int instead of unsigned types. The use of int is common throughout C #, and it is easier to interact with other libraries when int is used.

**Matrices**

Use concise syntax to initialize the arrays in the declaration line.

// Preferred syntax. Note that you cannot use var here instead of string[].

string[] vowels1 = { "a", "e", "i", "o", "u" };

// If you use explicit instantiation, you can use var.

var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one at a time.

var vowels3 = new string[5];

vowels3[0] = "a";

vowels3[1] = "e";

// And so on.

**Delegates**

Use concise syntax to create instances of a delegate type.

// First, in class Program, define the delegate type and a method that

// has a matching signature.

// Define the type.

public delegate void Del(string message);

// Define a method that has a matching signature.

public static void DelMethod(string str)

{

Console.WriteLine("DelMethod argument: {0}", str);

}

// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed syntax.

Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.

Del exampleDel1 = new Del(DelMethod);

**Try-catch and using instructions in exception handling**

Use a try-catch statement in most cases of exception handling.

static string GetValueFromArray(string[] array, int index)

{

try

{

return array[index];

}

catch (System.IndexOutOfRangeException ex)

{

Console.WriteLine("Index is out of range: {0}", index);

throw;

}

}

Simplify the code by using the C # using statement. If you have a try-finally statement in which the only code in the finally block is a call to the Dispose method, use a using statement instead.

// This try-finally statement only calls Dispose in the finally block.

Font font1 = new Font("Arial", 10.0f);

try

{

byte charset = font1.GdiCharSet;

}

finally

{

if (font1 != null)

{

((IDisposable)font1).Dispose();

}

}

// You can do the same thing with a using statement.

using (Font font2 = new Font("Arial", 10.0f))

{

byte charset = font2.GdiCharSet;

}

**New (Operator)**

Use the concise form of object instance creation with implicit types, as shown in the following statement.

var instance1 = new ExampleClass();

The previous line is equivalent to the following statement.

ExampleClass instance2 = new ExampleClass();

Use object initializers to simplify the creation of objects.

// Object initializer.

var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,

Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.

var instance4 = new ExampleClass();

instance4.Name = "Desktop";

instance4.ID = 37414;

instance4.Location = "Redmond";

instance4.Age = 2.3;

**Event control**

If you are defining an event handler that you do not need to remove later, use a lambda expression.

public Form2()

{

// You can use a lambda expression to define an event handler.

this.Click += (s, e) =>

{

MessageBox.Show(

((MouseEventArgs)e).Location.ToString());

};

}

**Static members**

Call static members with the class name ClassName.StaticMember. This practice makes the code more readable by clarifying static access. Do not qualify a static member defined in a base class with the name of a derived class. While the code is compiled, its readability is confusing, and may be interrupted in the future if a static member with the same name is added to the derived class.

**LINQ Queries**

Use descriptive names for the query variables. In the following example, seattleCustomers is used for customers located in Seattle.

var seattleCustomers = from cust in customers

where cust.City == "Seattle"

select cust.Name;

Use aliases to ensure that the property names of anonymous types are correctly written with uppercase or lowercase, using the Pascal script.

var localDistributors =

from customer in customers

join distributor in distributors on customer.City equals distributor.City

select new { Customer = customer, Distributor = distributor };

Change the name of the properties when they may be ambiguous in the result. For example, if the query returns a customer name and a distributor identifier, instead of leaving it as a Name and ID in the result, change its name to clarify that Name is the name of a customer and ID is the identifier of a distributor.

var localDistributors2 =

from cust in customers

join dist in distributors on cust.City equals dist.City

select new { CustomerName = cust.Name, DistributorID = dist.ID };

Use implicit types in the declaration of query variables and interval variables.

var seattleCustomers = from cust in customers

where cust.City == "Seattle"

select cust.Name;

Align the query clauses under the from clause, as shown in the previous examples.

Use where clauses before other query clauses to ensure that subsequent query clauses operate on a reduced and filtered data set.

var seattleCustomers2 = from cust in customers

where cust.City == "Seattle"

orderby cust.Name

select cust;

Use several from clauses instead of a join clause to gain access to internal collections. For example, a collection of Student objects could each contain a set of test results. When the following query is executed, it returns each result greater than 90, in addition to the last name of the student who received the score.

// Use a compound from to access the inner sequence within each element.

var scoreQuery = from student in students

from score in student.Scores

where score > 90

select new { Last = student.LastName, score };