```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5  using UnityEngine.UI;
6
7  #pragma warning disable
8
9  // Author: Adrian Mesa Pachon
10 // ContactInfo: https://www.linkedin.com/in/adrianmesa/
11
12 // -- Summary:
13 // This file is organized in different classes, each class is a collection of guidelines with examples.
14 // Some guidelines are more concrete and other are general ideas that improve code quality.
15
16 // The file has a top-down organization from more general to more concrete things
17
18 // 1 - General conventions
19 //  1.1 - Naming Conventions
20 //  1.2 - Class order
21 //  1.3 - Valid Opposites
22 //  1.4 - Project Abrevations
23
24 // 2 - Layout and Style
25
26 // 3 - Classes conventions
27 //  3.1 - Class Order
28 //  3.2 - Class Names
29 //  3.3 - Class Considerations
30
31 // 4 - Events conventions
32
33 // 5 - Properties conventions
34
35 // 6 - Methods/Properties conventions
36 //  6.1 Methods Naming Conventions
37 //  6.2 - Methods Parameters Conventions
38 //  6.3 - Methods return value conventions
39 //  6.4 - Methods considerations
40
41
42 // 7 - Variables conventions
43 //  7.1 Variable Declarations
44 //  7.2 Variable Names
45 //  7.3 Numbers
46 //  7.4 Characters And Strings
47 //  7.5 Booleans
48 //  7.6 Enums
49 //  7.7 Arrays
50
51 // 8 - Statements Conventions
52 //  8.1 - Conditionals
53 //  8.2 - Loops
54
55 // 9 - Comments conventions
56
```

```
57 // 10 - Unity Special Conventions
58 //  10.1 - Coroutines
59 //  10.2 - Field attributes
60
61 // -- Why i need it?
62 // As stated in "Code Complete Second Editions" the purpose of this convention is the following:
63 // https://www.amazon.com/Code-Complete-Practical-Handbook-Construction/dp/0735619670
64
65 // ■ They let you take more for granted.By making one global decision rather than
66 // many local ones, you can concentrate on the more important characteristics of
67 // the code.
68
69 // ■ They help you transfer knowledge across projects.Similarities in names give you
70 // an easier and more confident understanding of what unfamiliar variables are
71 // supposed to do.
72
73 // ■ They help you learn code more quickly on a new project.Rather than learning
74 // that Anita’s code looks like this, Julia’s like that, and Kristin’s like something
75 // else, you can work with a more consistent set of code.
76
77 // ■ They reduce name proliferation. Without naming conventions, you can easily
78 // call the same thing by two different names.For example, you might call total
79 // points both pointTotal and totalPoints.This might not be confusing to you when
80 // you write the code, but it can be enormously confusing to a new programmer
81 // who reads it later.
82
83 // -- How to use it:
84 // The idea is to include this file and the example file in your Unity project
85 // so you can open it quickly in order to check anything.
86 // Discuss with your team this guide and modify it so that everyone feels happy with the result before starting to code
87
88 // Because this guide has been designed to work with Unity projects, i have included some specific conventions.
89 // Feel free to share this file with your team-mates
90
91 // -- Expected feedback
92 // Any kind of feedback is welcome, i would like to improve this guide and make it a standard
93
94
95 namespace CodeConventions
96 {
97     public class CodeConventionForUnity
98     {
99         //
```

```
            ########################################################################
100         // 1 - General Conventions
101         //                                                                      ↵
            ########################################################################
102
103         public class GeneralConventions
104         {
105             // ============================
106             // 1.1 - Naming Conventions
107             // ============================
108
109             public void NamingConvention()
110             {
111                 // ClassName: PascalCase. Never use plural in class names.
112
113                 // TypeName: Type definitions, including enumerated types in ↵
                      PascalCase.
114
115                 // EnumeratedTypes: In addition to the rule above, enumerated↵
                       types are always stated in the plural form.
116
117                 // localVariable: Local variables are in camelCase.The name  ↵
                      should be independent
118                 // of the underlying data type and should refer to whatever  ↵
                      the variable represents. Use plural for collections.
119
120                 // classVariable: Member variables that are available to     ↵
                      multiple methods within a class are in camelCase(1). Use  ↵
                      plural for collections.
121
122                 // methodParameter: Use camelCase, use plural for           ↵
                      collections.
123
124                 // MethodName(): Methods are PascalCase. Use plural if the  ↵
                      method returns a collection.
125
126                 // CONSTANT_VAR: Named constants are in ALL_CAPS.
127
128                 // static readonly STATIC_VAR: Use ALL_CAPS. We use this    ↵
                      readoonly static var as const so we use the same naming   ↵
                      rules.
129
130                 // static varName: For a regular static var we use the same ↵
                      style as classVariable: camelCase.
131
132                 // PropertyName: Because properties are indeed methods, we  ↵
                      use the same convention as methods. This will help to     ↵
                      understand that calling a
133                 // property can have a cost.
134
135                 // OnEventTrigger: Events names are in mixed uppercase and  ↵
                      lowecase with an initial capital letter. All events should ↵
                      have a verb or verb phrase.
136
137                 // **** UNITY SPECIFICS ****
138
139                 // varNamePrefab: When a variable is a reference to a prefab,↵
                       we concat the prefab keyword at the end of the variable  ↵
                      name. We do this is to avoid
140                 // modifying a prefab by error.
```

```
141
142                 // varNameTemplate: When a variable is a reference to some component in the hierachy with the intention of being copied using Instantiate(varNameTemplate),
143                 // we add Template keyword at the end, with this we can see quickly if we are doing an invalid operation against a template (like destroying it!)
144
145                 // cachedComponentName: Use this to cached Unity components like cachedTransform, cachedAnimator, etc.
146
147                 // ************************
148
149                 // NOTES:
150                 // (1) We don't need to differentiate between local vars and class vars because we usually has different names. You can always can use this.varName
151                 // to differentiate between two vars with the same name is needed
152             }
153
154             // ============================
155             // 1.2 - Class order
156             // ============================
157
158             public void ClassOrder()
159             {
160                 /* Within a class, struct, or interface, elements must be positioned in the following order:
161
162                 1 - Constant Fields
163                 2 - Fields
164                 3 - Delegate declarations
165                 4 - Events
166                 5 - Enums
167                 6 - Constructors
168                 7 - Finalizers (Destructors)
169                 8 - Properties
170                 9 - Indexers
171                 10 - Methods
172                     10.1 - Unity Methods
173                     10.2 - Handlers
174                     10.3 - Class methods
175                 11 - Structs
176                 12 - Classes
177
178                 Within each of these groups order by access:
179
180                 public
181                 internal
182                 protected internal
183                 protected
184                 private
185
186                 Within  each of the access groups, order by static, then non-static:
187
188                 static
189                 non-static
```

```
190
191                 Within each of the static/non-static groups of fields, order ↵
                      by readonly, then non-readonly:
192                 readonly
193                 non-readonly
194
195                 Unrolled example:
196
197                 public static methods
198                 public methods
199                 internal static methods
200                 internal methods
201                 protected internal static methods
202                 protected internal methods
203                 protected static methods
204                 protected methods
205                 private static methods
206                 private methods
207
208                 */
209             }
210
211             // ============================
212             // 1.3 - Valid Opposites
213             // ============================
214
215             public void ValidOpposites()
216             {
217                 // To help consistency (and readability) you should use this ↵
                      kind of opposites precisely
218
219                 //add - remove
220                 //increment - decrement
221                 //open - close
222                 //begin - end
223                 //insert - delete
224                 //show - hide
225
226                 //create - destroy
227                 //lock- unlock
228                 //source - target
229                 //first - last
230                 //min - max
231                 //start - stop
232                 //current - next - previous
233                 //up - down
234                 //get - set
235                 //old - new
236             }
237
238             // ============================
239             // 1.4 - Project Abrevations
240             // ============================
241
242             public void ProjectAbrevations()
243             {
244                 // Put here your project abreviations
245
246                 // Idx for Index
247                 // SO for ScriptableObject: buildingSO, levelRewardsSO
```

```
248                 // SZ for classes that are serializable in Unity: buildingSZ, levelRewardsSZ
249             }
250         }
251
252
253         //#########################################################################
254         // 2 - Layout & Style conventions
255         //#########################################################################
256
257         private class LayoutConventions
258         {
259             public void Parentheses()
260             {
261                 // DO: Use parentheses to make clear expresions that involve more than two terms
262
263                 // GOOD:
264                 int result1 = (5 * 6) + 4;
265
266                 // BAD:
267                 int result2 = 5 * 6 + 4;
268             }
269
270             public void CodeBlocks()
271             {
272                 // DO: Add a blank line for the begin brace
273
274                 // GOOD:
275                 if (example != null)
276                 {
277                     Debug.Log(example);
278                 }
279
280                 // BAD:
281                 if (example != null){
282                     Debug.Log(example);
283                 }
284
285                 // BAD:
286                 // This inline method make difficulty to debug because you can't add a breakpoint inside the if
287                 if (example != null) { Debug.Log(example); }
288
289                 // DO: Use begin-end pairs ALWAYS to designate block boundaries. This will avoid bugs related with incorrect modifications
290
291                 // GOOD:
292                 if (example != null)
293                 {
294                     Debug.Log(example);
295                 }
296
297                 // BAD:
298                 if (example != null)
299                     Debug.Log(example);
300
301                 // DO: Use a new line in statements with complex expressions
```

```
302
303                 // GOOD:
304                 char inputChar = ' ';
305                 if (('0' <= inputChar && inputChar <= '9')
306                     || ('a' <= inputChar && inputChar <= 'z')
307                     || ('A' <= inputChar && inputChar <= 'Z'))
308                 {
309
310                 }
311             }
312
313             private void IndividualStatements()
314             {
315                 // DO: Break large statements in several lines making obvious
                       the incompleteness of the statement.
316                 // If you have operators, put them at the beggining of the
                      new line so all of them stay aligned
317
318                 // GOOD:
319                 int arrayIndex1 = 0;
320                 int arrayIndex2 = 0;
321                 int[] exampleArray1 = new int[10];
322                 int[] exampleArray2 = new int[10];
323
324                 if (exampleArray1[arrayIndex1] == 0
325                     || exampleArray2[arrayIndex2] == 0
326                     || exampleArray1[arrayIndex1] == END_VALUE
327                     || exampleArray2[arrayIndex2] == END_VALUE)
328                 {
329
330                 }
331
332                 // BAD:
333                 if (exampleArray1[arrayIndex1] == 0 || exampleArray2
                      [arrayIndex2] == 0 || exampleArray1[arrayIndex1] ==
                      END_VALUE || exampleArray2[arrayIndex2] == END_VALUE)
334                 {
335
336                 }
337
338                 // DO: Break methods with lot of parameters in several lines
339                 // TODO: Decide with the team the limit of parameters to use
                      line breaks
340                 Vector3 rayOrigin       = Vector3.zero;
341                 Vector3 direction       = Vector3.up;
342                 RaycastHit raycastHit   = new RaycastHit();
343                 float maxDistance       = 99f;
344                 int layerMask           = LayerMask.GetMask("Default");
345
346                 // GOOD:
347                 bool hit = Physics.Raycast(rayOrigin,
348                     direction,
349                     out raycastHit,
350                     maxDistance,
351                     layerMask);
352
353                 // BAD:
354                 bool hit1 = Physics.Raycast(rayOrigin, direction, out
                      raycastHit, maxDistance, layerMask);
355
```

```
356                 // DO: Align right sides of assigment statements that belong
                      to a same code block (1*)
357                 // TODO: Discuss this with the team to decide if we keep
                      doing this or not
358
359                 // GOOD:
360                 int variableName1               = 0;
361                 int variableNameWithMoreLength  = 0;
362
363                 OnExampleEvent1                 +=
                      LayoutConventions_OnExampleEvent1;
364                 OnExampleEventWithLongerName    +=
                      LayoutConventions_OnExampleEvent2;
365                 OnExampleEvent3                 +=
                      LayoutConventions_OnExampleEvent3;
366
367                 // DO: Use only one data declaration per line
368
369                 // GOOD:
370                 int thisIsAnInt = 0;
371                 int thisIsAnotherInt = 0;
372
373                 // BAD:
374                 string firstName = null, lastName = null;
375
376                 // DO: Order declarations by type
377
378                 // GOOD:
379                 Vector3 velocity        = Vector3.zero;
380                 Vector3 acceleration    = Vector3.zero;
381                 float maxSpeedInKmH     = 120;
382
383                 // BAD:
384                 Vector3 velocity1       = Vector3.zero;
385                 float maxSpeedInKmH1    = 120; // <-- float is in the middle
                      of two vector3
386                 Vector3 acceleration1   = Vector3.zero;
387
388                 // Additional notes
389                 // (1*): Despite knowing it is not a recommended guidline
                      because it increases the cost to maintain the code when you
                       rename variables
390                 // i beleieve that the advantages of improved readibility and
                       the slightly chance of catching some errors make this rule
                       worth it.
391             }
392
393             private void LayoutConventions_OnExampleEvent1()
394             {
395                 throw new System.NotImplementedException();
396             }
397
398             private void LayoutConventions_OnExampleEvent2()
399             {
400                 throw new System.NotImplementedException();
401             }
402
403             private void LayoutConventions_OnExampleEvent3()
404             {
```

```
405                 throw new System.NotImplementedException();
406             }
407
408             #region For examples
409             private const int END_VALUE = 99;
410
411             private delegate void ExampleHandlerEvent();
412             private event ExampleHandlerEvent OnExampleEvent1;
413             private event ExampleHandlerEvent OnExampleEventWithLongerName;
414             private event ExampleHandlerEvent OnExampleEvent3;
415
416             private Example example = new Example();
417             private Example.MethodExamples methodsExamples = new
                  Example.MethodExamples();
418             #endregion
419         }
420
421         //
              ###########################################################################
422         // 3 - Class Conventions
423         //
              ###########################################################################
424
425         private class ClassConventions
426         {
427             // ============================
428             // 3.1 - Class Order
429             // ============================
430
431             // DO: Use the same order of class members in every class
432
433             // 1º - Constatns
434             public const string CONSTANT_FIELD = "All constants go first";
435
436
437             private const string CONSTANT_FIELD_PRIVATE = "Private goes
                  always after public :P";
438
439             // 2º - Fields
440
441             public static readonly int[] STATIC_WITH_READONLY_TABLE = new int
                  [] {1,1,2,3,5,8}; // Note that we are using ALL_CAPS because
                  the static var is readonly (is used as a const)
442
443             public static string staticString = "";
444
445             // NOTE: As you already know ;), is not a good practice to use
                  public fields, you must use accesor methods (get/set)
446             public int examplePublicField = 0;
447
448
449             protected int protectedField = 0;
450
451
452             private static string HIDDEN_STATIC_STRING = "";
453
454
455             private int[] privateArray = new int[] { 0, 1, 2, 3, 4 };
456
457
458             // **** UNITY SPECIFICS ****
```

```
459             // If you need a field to the Unity inspector and avoid the violation of the encapsulation principle we can use [SerializeField] attribute:
460             [SerializeField]
461             private int thisIntMustBeSerializable = 0;
462
463             // These should always be private and marked with the [SerialiseField] attribute. If needed, these can be exposed
464             // to other scripts via public properties. Avoid changing inspector fields via script, as this can lead to unexpected behavior.
465             // Inspector fields should not be used to provide debug information to developers.
466             // ************************
467
468             // 3º - Delegates
469             public delegate void ButtonClickHandler(GameObject _target);
470             public delegate void WindowClosedHandler(MonoBehaviour _window);
471
472             // 4º - Events
473             public event ButtonClickHandler OnClicked;
474
475             // 5º - Enums
476             public enum ExampleEnum
477             {
478                 None = 0,
479                 Enum1 = 1,
480                 Enum2 = 2,
481             }
482
483             // 6º - Constructors
484
485             // **** UNITY SPECIFICS ****
486             // If your class is inheriting from Monobehaviour, you should know that the default constructor (constructor without parameters)
487             // is used by the Unity serializer and you can't acces Unity objects from this constructor.
488             // DO NOT: Do not use a constructor in classes that inherit from MonoBehaviour. Instead use Awake for internal initialization, and start for external initialization
489             // ************************
490
491             public ClassConventions()
492             {
493
494             }
495
496             // 7º - Destructors
497             ~ClassConventions()
498             {
499
500             }
501
502
503             // 8º - Properties
504             public int ExampleProperty { get; private set; }
505
506             // 9º - Indexers
507             public int this[int _index]
```

```
508             {
509                 get { return privateArray[_index]; }
510                 set { privateArray[_index] = value; }
511             }
512
513             // 10º - Methods
514
515             // 10.1 - Unity Methods
516
517             void Awake()
518             {
519
520             }
521
522             // DO: Provide services in pairs with their opposites
523             void OnEnable()
524             {
525                 OnClicked += ClassConventions_OnClicked;
526             }
527
528             void OnDisable()
529             {
530                 OnClicked -= ClassConventions_OnClicked;
531             }
532
533             // DO NOT: Leave blank Unity methods, delete them.
534
535             // 10.2 - Handlers
536
537             private void ClassConventions_OnClicked(GameObject _target)
538             {
539                 throw new System.NotImplementedException();
540             }
541
542             // 10.3 - Class methods
543
544             public void Open()
545             {
546
547             }
548
549             public void Close()
550             {
551
552             }
553
554             private void PrivateMethod()
555             {
556
557             }
558
559             // 11º - Structs
560
561             public struct NestedStruct
562             {
563                 public int exampleInt = 0;
564             }
565
566             // 12º - Classes
567
568             private class NestedClass
569             {
570                 public int exampleInt = 0;
571             }
```

```
572
573             // ============================
574             // 3.2 - Class Names
575             // ============================
576
577             private void ClassNames()
578             {
579                 // DO: Use Pascal case for class names
580
581                 // DO: Use noun or noun phrase to name a class
582
583                 // DO: Use abbreviations sparingly.
584
585                 // CONSIDER: Using a compound word to name a derived class.
                      The second part of the derived class's name should be the
                      name of the base class.
586                 // For example, ApplicationException is an appropriate name
                      for a class derived from a class named Exception, because
                      ApplicationException is a kind of Exception.
587                 // Use reasonable judgment in applying this rule. For
                      example, Button is an appropriate name for a class derived
                      from Control.
588                 // Although a button is a kind of control, making Control a
                      part of the class name would lengthen the name
                      unnecessarily.
589
590                 // DO: Use the preffix I to naming Interfaces
591
592                 // DO NOT: Use Abstract or Base as prefix of Abstract
                      Classes. You should regular name conventions to define the
                      name of abstract classes.
593                 // GOOD:
594                 // Shape (abstract class) Square (child class), Circle (child
                       class)
595             }
596
597             // ============================
598             // 3.3 - Class Considerations
599             // ============================
600
601             private class ClassConsiderations
602             {
603                 // DO: Provide services in pairs with their opposites (Open &
                       Close, OnEnable & OnDisable, Enter & Exit, etc)
604
605                 // GOOD:
606                 public void Enter()
607                 {
608
609                 }
610
611                 public void Exit()
612                 {
613
614                 }
615
616                 public void Update()
617                 {
618
619                 }
620
```

```
621                 // BAD:
622                 public void Enter1()
623                 {
624
625                 }
626
627                 public void Update1()
628                 {
629
630                 }
631
632                 public void Exit1()
633                 {
634
635                 }
636
637                 // DO: Move unrelated information to another class
638
639                 // DO: Don’t expose member data in public
640
641                 // GOOD:
642                 [SerializeField]
643                 private GameObject explosionPrefab = null;
644
645                 // BAD:
646                 public GameObject deadParticlesPrefab = null;
647
648                 // DO: Minimize accessibility of classes and members (hide as much information as possible)
649
650                 // DO: Keep the number of methods in a class as small as possible. Consider creating new clasess to keep each class small
651
652                 // DO: Avoid deep inheritance trees
653
654                 // DO: Preserve the Law of Demeter which helps to keep low the coupling of the class with other classes:
655                 // Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
656                 // Following this principle implies avoiding this kind of lines:
                      otherClass.otherComponent.whateverThing.transform.position
657             }
658         }
659
660         //////////////////////////////////////////////////////////////////////////
661         // 4 - Event Conventions
662         //////////////////////////////////////////////////////////////////////////
663
664         private class EventConventions
665         {
666             // DO: Because the events always refer to some action, use verbs to name events.
667             // Use the verb tense to indicate the time the event is raised
668
669             // GOOD:
670             // Clicked, Painting, DroppedDown
671             public event Action Click;
```

```
672             public event Action Clicked;
673             public event Action Closed;
674 
675             // BAD:
676             public event Action BeforeClose;
677             public event Action AfterClose;
678 
679 
680             // DO: Call event handlers with the "EventHandler" suffix
681 
682             // GOOD:
683             public delegate void ClickEventHandler(Button _button);
684 
685             // CONSIDER: When the handler of an event have recognozible parameters, you can use Action, when this
686             // paramater aren't clear, declare a delegate with the name of the parameters
687 
688             // Example:
689             // GOOD:
690             public event Action<Example.StatusEnum> StatusChanged; // There is no doubt with the parameter of this event
691 
692             // BAD:
693             public event Action<int, int, int> ScoreChanged; // In this case, the parameters aren't clear
694 
695             // GOOD
696             public delegate void ScoreChangeHandler(int newScore, int previousScore, int bestScore);
697         }
698 
699         //#########################################################################
700         // 5 - Properties Conventions
701         //#########################################################################
702 
703         private class PropertiesConventions
704         {
705             // DO: Because properties refers to data, we should use noun phrase or adjective names
706 
707             // GOOD:
708             public System.Object PlayerData {get; private set;}
709 
710             // BAD:
711             public System.Object GetPlayerData {get; private set;}
712 
713             // DO: Use plural if the property refers to a collection
714 
715             // GOOD:
716             public System.Object[] TeamMembers { get; private set; }
717 
718 
719             // DO: Name boolean properties with affirmative phrases or add a prefix as "Is", "Can" or "Has"
720 
721             // GOOD:
722             public bool HasConnection {get; private set;}
```

```
723             public bool CanJump {get; private set;}
724         }
725
726         //                                                                    ↵
              ###########################################################################
727         // 6 - Methods Conventions
728         //                                                                    ↵
              ###########################################################################
729
730         private class MethodsConventions
731         {
732             Example examples = new Example();
733             Example.MethodExamples examples2 = new Example.MethodExamples();
734
735             // ================================
736             // 6.1 Methods Naming Conventions
737             // ================================
738
739             // DO: A method name must describe everything the method does
740
741             // GOOD
742             void ComputeGameOverScoreAndUploadToServer()
743             {
744
745             }
746
747             // BAD
748             void GameOverScore()
749             {
750
751             }
752
753             // DO NOT: Use meaningless, vague or wishy-washy verbs
754
755             // GOOD:
756             void UploadGameResults()
757             {
758
759             }
760
761             // BAD:
762             void Upload()
763             {
764
765             }
766
767             // DO NOT: Don't use numbers to differentiate method names unless ↵
                   those numbers have a logical meaning in that context.
768
769             // GOOD:
770             void PlaySound(AudioClip _clip)
771             {
772
773             }
774
775
776             void PlaySound(AudioClip _clip,float _volume)
777             {
778
779             }
780
781             // BAD:
782             void PlaySound1(AudioClip _clip)
```

```
783             {
784
785             }
786
787             void PlaySound2(AudioClip _clip, float _volume)
788             {
789
790
791             }
792
793             // DO: Make names of methods as long as neccesary. If the method
                  name get too long, consider this as a signal to split the
                  method in two methods.
794
795             // DO: Use verbs for method names
796
797             // DO: If the function returns a value, to name a function use a
                  description of the value (only when it returns a single value)
798
799             // DO: Stablish conventions for common operations
800
801             // BAD:
802
803             // You are using two different method names to get an Id, choose
                  one form and stick to it.
804
805             int GetId()
806             {
807                 return 0;
808             }
809
810             int Id()
811             {
812                 return 0;
813             }
814
815             // ================================
816             // 6.2 - method Parameters Conventions
817             // ================================
818
819             private void MethodParametersOrder()
820             {
821                 // DO: Put parameters in input-modify-output-
822                 // in the output parameters, put error/status order last
823
824                 // GOOD
825                 GameObject mainWeapon = null;
826                 GameObject instantiatedPlayer = null;
827                 bool skinConfigurationError = false;
828                 examples2.ConfigurePlayerSkin(true, ref instantiatedPlayer,
                      out mainWeapon, out skinConfigurationError);
829             }
830
831             // DO: method parameters start with a underscore '_'
832
833             // GOOD:
834             void ExampleMethod(int _parameter)
835             {
836
837             }
838
839             // DO: If similar methods use similar parameters, put the similar
```

```
                parameters in a consistent order
840
841             // DO: Use all the parameters, remove unused parameters. If you
                  need to keep compatibility for legacy code, make use of
                  [Obsolete] attribute
842
843             // DO NOT: Don't use method parameters as working variables, use
                  local variables instead
844
845             // BAD:
846             int MathOperationExampleBad(int _value)
847             {
848                 _value *= 5;
849
850                 return _value;
851             }
852
853             // GOOD:
854             int MathOperationExampleGood(int _value)
855             {
856                 int result = _value * 5;
857
858                 return _value;
859             }
860
861             // DO: Limit the number of method's parameters to about seven. If
                   you need more parameters probably you need
862             // a new class/struct to represent that data
863
864             // EXAMPLE:
865             public void MethodParametersLimit(int value1, int value2, int
                  value3, int value4, int value5, int value6, int value7)
866             {
867
868             }
869
870             // DO: Pass the variables or objects that the method needs to
                  maintain its interface abstraction. Note that this can be
                  subtle
871             // and in some cases, passing the entire object as parameter is
                  OK if the two clases are coupled.
872
873             void ParatemersAbstractionExample()
874             {
875                 // GOOD:
876                 examples2.MethodWithGoodParameters(examples.dummyInt,
                      examples.dummyString);
877
878                 // BAD:
879                 examples2.MethodWithBadParameters(examples.DummyObject); //
                      <- this method doesn't need to know anything about
                      DummyObject, only needs the int and string
880             }
881
882             // DO: Follow the regular naming guidelines in lambda expresions
                  and auto-generated delegates
883
884             void LambdaExpressionExample()
885             {
886                 List<int> example = new List<int> ();
887
```

```
888                 // GOOD:
889                 example.FindAll(delegate (int _index)
890                 {
891                     return _index > 2;
892                 });
893 
894                 // BAD:
895                 example.FindAll(delegate (int _x)
896                 {
897                     return _x > 2;
898                 });
899             }
900 
901             // DO: Check input paramaters before assignment. Make sure that the values are reasonable
902             // You can use asserts if you don't want this checks in your release versions in scenarios where performance is the priority
903             void MethodParametersCheck(Example _exampleObject)
904             {
905                 if(_exampleObject == null)
906                 {
907                     Debug.LogError("The parameter _exampleObject can't be null");
908                     return;
909                 }
910 
911                 Debug.LogFormat("Status: {0}",_exampleObject.Status);
912             }
913 
914             // ================================
915             // 6.3 - Methods return value conventions
916             // ================================
917 
918             // DO: Have a single return point in your method, this will help to maintain and debug your code.
919             // An exception of this rule is when doing error handling at the beggining of a method
920 
921             // GOOD:
922             public int ExampleMethod1()
923             {
924                 // DO: Initialize the return value at the beggining of the function to a default value
925                 int returnValueExample = 0;
926 
927                 bool condition1 = false;
928                 bool condition2 = false;
929 
930                 if (condition1)
931                 {
932                     returnValueExample = 1;
933                 }
934                 else if (condition2)
935                 {
936                     returnValueExample = 2;
937                 }
938 
939                 return returnValueExample;
940             }
941 
942             // BAD:
```

```
943             public int ExampleMethod2()
944             {
945                 bool condition1 = false;
946                 bool condition2 = false;
947 
948                 if (condition1)
949                 {
950                     return 1;
951                 }
952                 else if (condition2)
953                 {
954                     return 2;
955                 }
956 
957                 return 0;
958             }
959 
960             // ================================
961             // 6.4 Method considerations
962             // ================================
963 
964             private void MethodsConsiderations()
965             {
966                 // DO: methods must have a single purpose
967 
968                 // DO: Try to avoid methods over 200 lines of code (comments
                      and blank lines are excluded). There are a lot of studies
                      with different results
969                 // so there is not an official standard in the industry. Try
                      to avoid large methods because usually, they are
                      consequences of bad programming practices
970                 // but you can always write methods over 200 lines if you
                      need it and they are simple enough to be readable,
                      maintenable and undertestanable.
971             }
972 
973         }
974 
975         //
            ###########################################################################
976         // 7 - Variable Conventions
977         //
            ###########################################################################
978 
979         private class VariableConventions
980         {
981             // ================================
982             // 7.1 Variable Declarations
983             // ================================
984 
985             private void VariableDeclarations()
986             {
987                 // DO: Initialize all variables
988 
989                 // GOOD:
990                 int myInt = 0;
991                 GameObject myGameObject = null;
992 
993                 // BAD:
994                 int anotherInt;
995                 GameObject targetGameobject;
```

```
996
997
998                 // DO: Initialize each variable close to where it's firts used to
999                 // preserve the Principle of Proximity: keep related actions together
1000
1001                // GOOD:
1002                string id = example.DummyObject.ID;
1003                Debug.LogFormat("Id: {0}", id);
1004
1005                string secondID = example.DummyObject.ID;
1006                Debug.LogFormat("secondID: {0}", secondID);
1007
1008
1009                // BAD:
1010                string sourceId = example.DummyObject.ID;
1011                string targetId = example.DummyObject.ID;
1012
1013                Debug.LogFormat("SourceId: {0}", sourceId);
1014                Debug.LogFormat("TargetId: {0}", targetId);
1015
1016                // CONSIDER: declaring and defining each variable where it's first used:
1017
1018                // GOOD:
1019                Example.StatusEnum status1 = example.Status;
1020
1021                // BAD:
1022                Example.StatusEnum status2 = Example.StatusEnum.Undefined;
1023                status2 = example.Status;
1024
1025                // DO: Group related statements
1026
1027                // GOOD:
1028                var dummyObject = example.DummyObject; // <-- Dummy Object
1029                bool dummyObjectOK = example.DoOperations(dummyObject); // <-- Operate over Dummy Object
1030                var oldDummyObject = example.OldDummyObject; // <-- Old Dummy Object
1031                bool oldDummyObjectOK = example.DoOperations (oldDummyObject); // <-- Operate over Old dummy object
1032
1033
1034                // DO NOT:
1035                var dummyObject1 = example.DummyObject; // <-- Dummy Object
1036                var oldDummyObject1 = example.OldDummyObject; // <-- Old Dummy Object
1037                bool oldDummyObjectOK1 = example.DoOperations (oldDummyObject1); // <-- Operate over Old dummy object
1038                bool dummyObjectOK1 = example.DoOperations(dummyObject1); // <-- Operate over Dummy Object
1039
1040                // DO: Use each variable for one purpose only to improve readibility
1041
1042                // GOOD:
1043                int randomIndex = example.GetRandomInt();
1044                Debug.LogFormat("Random Index: {0}", randomIndex);
```

```
1045
1046                int score = example.CalculateScore();
1047                Debug.LogFormat("Score: {0}", score);
1048
1049                // BAD:
1050                int tempValue = example.GetRandomInt();
1051                Debug.LogFormat("Random Index: {0}", tempValue);
1052
1053                tempValue = example.CalculateScore();
1054                Debug.LogFormat("Score: {0}", tempValue);
1055
1056
1057                // DO NOT: Set variables with hidden meanings to avoid
                      "hybrid coupling"
1058
1059                // GOOD:
1060                var status = example.Status;
1061                if (status == Example.StatusEnum.Ok)
1062                {
1063                    int finalScore = example.CalculateScore();
1064                }
1065                else
1066                {
1067                    Debug.LogError("Some error occured. The game doesn't
                      ended well");
1068                }
1069
1070                // BAD:
1071                int gameOverScore = example.CalculateScore();
1072                if (gameOverScore == -1) // Two meanings for gameOverScore:
                      game score & error
1073                {
1074                    Debug.LogError("Some error occured. The game doesn't
                      ended well");
1075                }
1076
1077
1078                // DO: Make sure all declared variables are used.
1079                // You can see Warnings in the console to remove unused
                      variables.
1080
1081
1082                // DO: Use 'var' keyword in variable declarations where the
                      type is obvious or isn't relevant. For example, with
                      numeric variables
1083                // you CAN'T user var because it is important to know if you
                      are working with ints or floats
1084                // - Code maintenance is improved
1085                // - Code readability is improved
1086                // TODO: Discuss this with the team
1087
1088                // GOOD:
1089                var target = example.DummyGameObject;
1090
1091                // BAD:
1092                GameObject source = example.DummyGameObject;
1093
1094                // With numeric values:
1095
1096                // BAD:
```

```
1097                var totalPlays = 5;
1098                var totalScore = example.CalculateScore();
1099                var averageScore = totalScore / totalPlays; // <-- we can
                      have errors because we don't know the type of the variables
                       (float, int, etc)
1100
1101
1102                // Note (1): If you are concerned about performance, you can
                      use Assertions (Unity Debug.Assert)
1103            }
1104
1105            // ================================
1106            // 7.2 Variable Names
1107            // ================================
1108
1109            private void VariableNames()
1110            {
1111                // DO: Try to avoid computer related terms and use problem
                      domain terms.
1112
1113                // GOOD:
1114                System.Object playerSaveGame = new System.Object();
1115
1116                // BAD:
1117                System.Object savedData = new System.Object();
1118
1119                // DO NOT: Use too short, too long, hard to type, hard to
                      pronounce variable names
1120
1121                // CONSIDER: Using variables with a name length: Between 8
                      and 20 characters
1122                // ABCDEFGH <-- 8 characters
1123                // ABCDEFGHIJKLMNOPQRST <-- 20 characters
1124
1125                // DO: If the variable has a qualifer like: Total, Sum,
                      Average, Max, Min, Record, String, Pointer, etc
1126                // put it at the end of the name
1127
1128                // GOOD:
1129                int scoreTotal = 0;
1130                float healthAverage = 0;
1131                Text titleLabel = null;
1132
1133                // DO: Use more detailed index names for nested loops to
                      avoid index corss-talk errors (saying i when you mean j and
                       vice versa)
1134
1135                // GOOD:
1136                int[,] scores = new int[5, 5];
1137                int teamCount = 5;
1138                int eventCount = 5;
1139                for (int teamIndex = 0; teamIndex < teamCount; teamIndex++)
1140                {
1141                    for (int eventIndex = 0; eventIndex < eventCount;
                      eventIndex++)
1142                    {
1143                        scores[teamIndex, eventIndex] = 0;
1144                    }
1145                }
1146
```

```
1147                // DO: Give boolean variables names that imply true or false as:
1148                // done, error, found, success, ok
1149
1150                // DO: Use positive boolean variable names:
1151
1152                // GOOD:
1153                bool found = false;
1154
1155                // BAD:
1156                bool notFound = true;
1157
1158
1159                // DO NOT: Use names with ambiguous meanings
1160
1161                // GOOD:
1162                {
1163                    int fileCount = 0;
1164                    int fileIndex = 1;
1165                }
1166
1167                // BAD:
1168                {
1169                    int fileNumber = 0;
1170                    int fileIndex = 0;
1171                }
1172
1173
1174                // DO NOT: Use similar names in variables with different meaning
1175
1176                // GOOD:
1177                Rect screenArea = new Rect();
1178                Vector2 screenResolution = new Vector2();
1179
1180                // BAD:
1181                Rect screenRect = new Rect();
1182                Vector2 screenRes = new Vector2();
1183
1184                // DO: Avoid names that sound similar
1185
1186                // DO NOT: Use numerals in names. If you feel you need numerals probably you need a different data type as an array
1187
1188                // GOOD:
1189                GameObject[] itemSlots = null;
1190
1191                // BAD:
1192                GameObject itemSlot1 = null;
1193                GameObject itemSlot2 = null;
1194            }
1195
1196            // ================================
1197            // 7.3 Numbers
1198            // ================================
1199
1200            private void NumericVariables()
1201            {
1202                // DO: Avoid magic numbers
1203
```

```
1204                // GOOD:
1205                float initialSpeed = 10f;
1206                float timeInSeconds = 60f;
1207                float fallSpeed = initialSpeed + Example.GRAVITY_ACCELERATION
                       * timeInSeconds;
1208
1209                // BAD:
1210                float fallSpeed1 = initialSpeed + 9.8f * timeInSeconds;
1211
1212                // DO: Anticipate divide-by-zero errors
1213
1214                // DO: Make data type conversions explicit in mathematical
                      operations.
1215
1216                // GOOD:
1217                float x = 0f;
1218                int i = 0;
1219                float y = x + (float)i;
1220
1221                // BAD:
1222                float w = x + i;
1223
1224                // DO NOT: Use mixed-type comparisons
1225
1226                // GOOD:
1227                if (i == (int)x)
1228                {
1229
1230                }
1231
1232                // BAD:
1233                if (i == x)
1234                {
1235
1236                }
1237
1238                // DO: Be careful with integer divisions
1239                float result = 10f * (7 / 10);  // <-- will return 0 in C#
1240
1241                // CONSIDER: Integer overflow, you should think about the
                      largest value your expression can assume.
1242
1243                // DO NOT: add/substract on numbers that have greatly
                      different magnitudes with float numbers
1244                float result1 = 1000000.00f + 0.1f; // <--  can have a result
                       of 1,000,000.00
1245
1246                // CONSIDER: The presence of accuracy errors in comparisons
                      with floating-point numbers
1247
1248                // GOOD:
1249                float maximumSpeed = 0f;
1250                float speed = 0f;
1251                if (maximumSpeed - speed < Example.SPEED_MAX_DELTA)
1252                {
1253
1254                }
1255
1256                // BAD:
1257                if (speed == maximumSpeed)
1258                {
```

```
1259
1260                }
1261            }
1262
1263            // ================================
1264            // 7.4 Characters And Strings
1265            // ================================
1266
1267            private void CharacterAndStrings()
1268            {
1269                // DO: Avoid magic strings
1270
1271                // GOOD:
1272                string localizedTitle = example.GetLocalizedText
                      (Example.LOCALIZED_TITLE);
1273
1274                // BAD:
1275                string localizedTitle1 = example.GetLocalizedText
                      ("TITLE_KEY");
1276
1277                // DO: Use string.Format() to compose strings
1278
1279                // GOOD:
1280                int score = 0;
1281                string finalText1 = string.Format("You have won {0} points",
                      score);
1282
1283                // BAD:
1284                string finalText2 = "You have won " + score + " points";
1285
1286                // **** UNITY SPECIFICS ****
1287
1288                // DO: Use Debug.LogFormat, Debug.LogErrorFormat &
                      Debug.LogWarningFormat to print logs with composes messages
1289
1290                // ************************
1291
1292                // DO: Use string.IsNullOrEmpty to check for empty/null
                      strings
1293
1294                // GOOD:
1295                string userName = "";
1296                if (string.IsNullOrEmpty(userName))
1297                {
1298
1299                }
1300
1301                // BAD:
1302                if (userName == null || userName == "")
1303                {
1304
1305                }
1306
1307                // DO: Consider using the StringBuilder class if you need to
                      work with long strings to reduce the impact on performance
1308
1309                // GOOD:
1310                string text = null;
1311                System.Text.StringBuilder sb = new System.Text.StringBuilder
                      ();
1312                for (int i = 0; i < 100; i++)
```

```
1313                {
1314                    sb.AppendLine(i.ToString());
1315                }
1316
1317                // BAD:
1318                for (int i = 0; i < 100; i++)
1319                {
1320                    text += i.ToString();
1321                }
1322            }
1323
1324            // ================================
1325            // 7.5 Booleans
1326            // ================================
1327
1328            private void Booleans()
1329            {
1330                // DO: Use boolean variables to increase readibility and
                      maintenance in logic expressions
1331
1332                // GOOD:
1333                int elementIndex = 0;
1334                int lastElementIndex = 0;
1335
1336                bool finished = ((elementIndex < 0) || (Example.MAX_ELEMENTS
                      < elementIndex));
1337                bool repeatedEntry = (elementIndex == lastElementIndex);
1338
1339                if (finished || repeatedEntry)
1340                {
1341
1342                }
1343
1344                // BAD:
1345                if ((elementIndex < 0) || (Example.MAX_ELEMENTS <
                      elementIndex) || (elementIndex == lastElementIndex))
1346                {
1347
1348                }
1349            }
1350
1351            // ================================
1352            // 7.6 Enums
1353            // ================================
1354
1355            // CONSIDER: Defining  the first value in an enum for an
                  "invalid" value if it has sense with the purpose of the enum.
1356            // CONSIDER: Defining the last value of an enum if you are going
                  to use it in iterations
1357
1358            // GOOD:
1359            public enum GameModes
1360            {
1361                None,
1362                DeathMatch,
1363                TeamDeathMatch,
1364                CaptureTheFlag,
1365                End,
1366            }
1367
1368            // **** UNITY SPECIFICS ****
1369
```

```
1370            // DO NOT: Add new enum values in the middle of the enum, this
                  will change the value of a serializable field in the Unity
                  Inspector
1371
1372            // GOOD:
1373            private enum GameModes1
1374            {
1375                None,
1376                DeathMatch,
1377                TeamDeathMatch,
1378                CaptureTheFlag,
1379                NewGameMode, // <-- Added at the end (before Final)
1380                Final,
1381            }
1382
1383            // BAD:
1384            private enum GameModes2
1385            {
1386                None,
1387                DeathMatch,
1388                NewGameMode, // <-- Added in the middle, now the seralizable
                      fields with the value of TeamDeatchMatch will have
                      NewGameMode assigned
1389                TeamDeathMatch,
1390                CaptureTheFlag,
1391                Final,
1392            }
1393
1394            // *************************
1395
1396
1397            private void UsingEnums()
1398            {
1399                // DO: Try to use the same enum values in external services
                      to avoid name conversions
1400
1401                // GOOD:
1402                string gameModeString = example.GetGameModeFromServer();
1403                GameModes gameModeFromServer = (GameModes)Enum.Parse(typeof
                      (GameModes), gameModeString);
1404                if (gameModeFromServer == GameModes.DeathMatch)
1405                {
1406
1407                }
1408
1409                // BAD:
1410                string gameModeString1 = example.GetGameModeFromServer();
1411
1412                if (gameModeString1 == "DEATH_MATCH")
1413                {
1414                    gameModeFromServer = GameModes.DeathMatch;
1415                }
1416            }
1417
1418            // ================================
1419            // 7.7 Arrays
1420            // ================================
1421
1422
1423            public void Arrays()
1424            {
```

```
1425                // DO: Check that array indexes are within the bounds of the
                      array
1426
1427                // GOOD:
1428                {
1429                    GameObject[] exampleArray = new GameObject[5];
1430                    int index = 0;
1431                    // index = NextIndex();
1432                    int maxLength = exampleArray.Length;
1433
1434                    if (index >= maxLength)
1435                    {
1436                        Debug.LogErrorFormat("Array out of range. Index: {0}
                       Max: {1}", index, maxLength);
1437                        index = maxLength - 1;
1438                    }
1439
1440                    var myGameObject = exampleArray[index];
1441                }
1442
1443                // BAD:
1444                {
1445                    GameObject[] exampleArray = new GameObject[5];
1446                    int index = 0;
1447                    // index = NextIndex();
1448                    int maxLength = exampleArray.Length;
1449
1450                    var myGameObject = exampleArray[index];
1451                }
1452
1453                // DO: Use lists if you don't know the size of the array, in
                      other case, use arrays
1454            }
1455
1456            private Example example = new Example();
1457        }
1458
1459        //
          ############################################################################
1460        // 8 - Statements Conventions
1461        //
          ############################################################################
1462
1463        private class StatementsConventions
1464        {
1465            // ============================
1466            // 8.1 - Conditionals
1467            // ============================
1468
1469            private Example example = new Example();
1470
1471            private void Conditionals()
1472            {
1473                // DO: Write the nominal path through the code first, then
                      write unusual cases. This improves readability and
                      performance
1474
1475                // DO: Write errors case outside the conditional where you
                      are taking the decissions
1476
1477                // GOOD:
```

```
1478                bool error = false;
1479
1480                if(error)
1481                {
1482                    Debug.LogError("Notify of error");
1483                    return;
1484                }
1485
1486                bool condition1 = false;
1487
1488                if (condition1)
1489                {
1490                    // Do something
1491                }
1492
1493                // BAD:
1494                if (error)
1495                {
1496                    Debug.LogError("Notify of error");
1497                }
1498                else
1499                {
1500                    if (condition1)
1501                    {
1502                        // Do something
1503                    }
1504                }
1505
1506                // DO: Put the normal case after the if rather than after the ↵
                       else
1507
1508                // GOOD:
1509                if (example.Status == Example.StatusEnum.Ok)
1510                {
1511                    // Normal case
1512                }
1513                else
1514                {
1515
1516                }
1517
1518                // BAD:
1519                if (example.Status != Example.StatusEnum.Ok)
1520                {
1521
1522                }
1523                else
1524                {
1525                    // Normal case
1526                }
1527
1528                // CONSIDER: Consider to write the else clause to make clear ↵
                     to other programmers that you have considered that case
1529
1530                // GOOD:
1531                bool validID = !string.IsNullOrEmpty(example.DummyObject.ID);
1532                if (validID)
1533                {
1534                    // Normal case
1535                }
1536                else
1537                {
```

```
1538                    // If the id is not valid nothing happens here
1539                }
1540
1541                // BAD:
1542                if (validID)
1543                {
1544                    // Normal case
1545                }
1546
1547                // DO: Simplify complicated tests with boolean functions
                      calls
1548
1549                // GOOD:
1550                bool statusIsOk = (example.Status == Example.StatusEnum.Ok);
1551                if(validID && statusIsOk)
1552                {
1553                    // Do something
1554                }
1555
1556                // BAD:
1557                if (!string.IsNullOrEmpty(example.DummyObject.ID) &&
                      example.Status == Example.StatusEnum.Ok)
1558                {
1559                    // Do somethinf
1560                }
1561
1562                // DO: In several if/else if statements, start always for the
                      most frequent cases first
1563
1564                // GOOD:
1565                char inputChar = ' ';
1566                if (example.IsLetter(inputChar))
1567                {
1568                    // Do something with letter
1569                }
1570                else if(example.IsNumber(inputChar))
1571                {
1572                    // Do something with number
1573                }
1574                else if(example.IsPunctiation(inputChar))
1575                {
1576                    // Do something with punctuations
1577                }
1578                else
1579                {
1580                    Debug.LogErrorFormat("Unrecognized char: {0}",
                       inputChar);
1581                }
1582            }
1583
1584            // ============================
1585            // 8.2 - Loops
1586            // ============================
1587
1588            public void Loops()
1589            {
1590                List<string> exampleNames = new List<string>();
1591
1592                // DO: Put initialization code directly before the loop
1593
1594                // EXAMPLE:
```

```
1595                bool enc    = false;
1596                int index   = 0;
1597                while (!enc)
1598                {
1599                    enc = exampleNames[index] == "Unnamed";
1600                    index++;
1601                }
1602
1603                // **** UNITY SPECIFICS ****
1604                // DO: Prefer for loops instead of foreach loops if possible.
                       For loops ar faster in Unity that foreach loops
1605
1606                // GOOD:
1607                for (int i = 0; i < exampleNames.Count; i++)
1608                {
1609                    Debug.Log(exampleNames[i]);
1610                }
1611
1612                // BAD:
1613                foreach(var name in exampleNames)
1614                {
1615                    Debug.Log(name);
1616                }
1617
1618                // ************************
1619
1620                // DO: When the number of iterations is indefinite, use a
                     while loop
1621
1622                // DO: Use { and } to enclose statements in a loop always to
                     prevent errors when code is modified.
1623
1624                // GOOD:
1625                for (int i = 0; i < 100; i++)
1626                {
1627                    Debug.Log(i);
1628                }
1629
1630                // BAD:
1631                for (int i = 0; i < 100; i++)
1632                    Debug.Log(i);
1633
1634                // DO: Keep loop-housekeeping chores at either the beginning
                     or the end of the loop
1635
1636                // DO: Make each loop perform only one function. Loops should
                       be like methods.
1637                // An exception of this rule it is in places where
                     performance is critical
1638
1639                // DO: Make sure that a loop ends. This is specially
                     important in some while loops that are potentially
                     dangerous as infinite loops.
1640                // Add maximum iterations counters or timers to avoid this
                     problem.
1641
1642                // GOOD:
1643                float securityTimer = 0f;
1644                while (example.Status != Example.StatusEnum.Ok &&
                     securityTimer < 10f)
```

```
1645                {
1646                    // Do something
1647
1648                    securityTimer += Time.deltaTime;
1649                }
1650
1651                if(securityTimer >= 10f)
1652                {
1653                    Debug.Log("TimeOut");
1654                }
1655
1656                // BAD:
1657                while (example.Status != Example.StatusEnum.Ok)
1658                {
1659
1660                }
1661
1662                // DO: If you need to use continue in a for loop, do it at the top of the loop
1663
1664                // GOOD:
1665                for(int i = 0; i < 100; i++)
1666                {
1667                    if (example.Status != Example.StatusEnum.Ok)
1668                        continue;
1669                }
1670
1671                // DO: Use meaningful variable names to make nested loops readable
1672                // Note: this is already seen in the section: 7.2 Variable Names
1673
1674                // GOOD:
1675                for (int month = 0; month < 12; month++)
1676                {
1677                    for(int day = 0; day < 31; day++)
1678                    {
1679
1680                    }
1681                }
1682
1683                // BAD:
1684                for (int i = 0; i < 12; i++)
1685                {
1686                    for (int j = 0; j < 31; j++)
1687                    {
1688
1689                    }
1690                }
1691
1692                // DO: Limit nesting of loop to three levels. Break the loops into methods if you need it to avoid this
1693            }
1694        }
1695
1696        //############################################################################
1697        // 9 - Comments conventions
1698        //############################################################################
1699
1700        private class CommentsConventions
```

```
1701        {
1702            private Example example = new Example();
1703
1704            // DO: Write code that is enough clear to be a "self-documenting"
                   code. If you have to write a comment just because the code is
                  so complicated,
1705            // it is better to improve the code that to write the comment
1706
1707            // DO NOT: Write comments that repeats what the code does
1708
1709            // DO: Write comments that summary the code to help other
                  programmers at reading the code
1710
1711            // DO: Write comments to describe the code's intent. Use always
                  vocabulary in the domain of the problem instead of triying to
                  describe the solution
1712
1713            // GOOD:
1714            // get current employee information
1715
1716            // BAD:
1717            // get employee object from database query
1718
1719            // DO NOT: Leave big regions of code commented when you commit to
                   developer. Use the version repository to see old code.
1720
1721            // DO NOT: Leave comments until the end. You need to integrate
                  commenting into your development style. This will help others
                  in code reviews and also will help you
1722            // to think more about the problem you are triying to solve
1723
1724            // DO: Use XML comments because the will be showed in the
                  Intellisense.
1725            // DO NOT: Use endline comments.
1726
1727            // GOOD
1728
1729            /// <summary>
1730            /// Explain what this method does
1731            /// </summary>
1732            private void ExampleMetho1d()
1733            {
1734
1735            }
1736
1737            // BAD
1738            private void ExampleMethod2() // Explain what this method does
1739            {
1740            }
1741
1742
1743            // BAD
1744            // Explain what this method does
1745            private void ExampleMethod3()
1746            {
1747
1748            }
1749
1750
1751            // **** UNITY SPECIFICS ****
1752            // CONSIDER: Using Unity Tooltip atribute instead of comments to
```

```
                  describe a variable if this variable is going to be modified in
                   the editor and its meaning
1753              // isn't clear with the name.
1754              // This will replace the regular commenting style
1755              // GOOD
1756              [Tooltip("Use this variable to declare the maximum value of the
                  player health. Useful when implementing penalties")]
1757              [SerializeField]
1758              public float maximumHealthClamp = 1f;
1759              // ********************
1760
1761              // CONSIDER: Using endline comments to mark end of blocks in
                  complex if/else nested blocks. By default, you should use your
                  IDE matching brackets tool.
1762
1763              // GOOD
1764
1765              private void ExampleMethod3()
1766              {
1767                  if (example.Status == Example.StatusEnum.Ok)
1768                  {
1769                      // JUST IMAGINE A VERY LARGE IF BLOCK
1770                      // ...
1771                      // ...
1772                      // ...
1773                      // ...
1774                      // ...
1775
1776                  } // End of STatus == Ok condition
1777              }
1778
1779              // DO: Use comment to justify violations of good programming
                  style
1780
1781              // DO: Use comments to explain optimizations or to explain why
                  you have used a more complicated approach to solve a problem
                  instead of a more straighforward one
1782
1783              // DO: Comment units of numeric data (no matter if the unit of
                  the data form part of the variable name)
1784
1785              // GOOD:
1786
1787              public Vector3 spaceshipVelocity = Vector3.zero; // Spaceship
                  velocity vector in meters per second
1788
1789              // DO: Comment the range of allowable numeric values
1790
1791              // GOOD:
1792
1793              public float normalizedProgress = 0f; // Player normalized
                  progress in this level between 0 and 1
1794
1795              // CONSIDER: Commenting enums values if they are not obvious
1796
1797              // DO: When commenting class members as methods and properties,
                  use the default format in c# (automatically generated if you
                  write ///)
1798
1799              // GOOD:
1800
```

```
1801            /// <summary>
1802            /// Summary of the method here
1803            /// </summary>
1804            /// <param name="_param1"> Describe the parameter, describe
                  expected values, units, etc</param>
1805            /// <returns> Describe the return value </returns>
1806            public int Method1(int _param1)
1807            {
1808                return 0;
1809            }
1810
1811            // Because to much comments decrease code readability, we should
                  rely in clean code, small & simple classes instead of
                  commenting everything.
1812            // in order to know what to comment, follow the next guide:
1813
1814            // Fields: because class fields are protected or private, you
                  should only comment fields that aren't clear enough with the
                  field name
1815            // Public Methods/Properties: THey should be commented. Be aware
                  of methods with more than two lines of comment, they can be a
                  synthom of a design problem
1816            // Private methods/properties: They only should be commented if
                  their purpose isn't clear
1817
1818            // DO: Comment classes describing their design approach,
                  limitations, usage assumptions and so on
1819        }
1820
1821        //
              ###########################################################################
1822        // 10 - Unity Special Conventions
1823        //
              ###########################################################################
1824
1825        private class UnityConventions
1826        {
1827            Example example = new Example();
1828
1829            private MonoBehaviour dummyComponent = null;
1830
1831            // ================================
1832            // 10.1 - Coroutines
1833            // ================================
1834
1835            // DO: If you have to wait for one frame, and you are not working
                     with graphic stuff, use "yield return null" instead of "yield
                  return new WaitForEndOfFrame()"
1836            // doing this you will avoid memory allocation in each execution
                  of the loop
1837
1838            // Note: It's important to know that  yield return null is not
                  evaluated in the same moment than WaitForEndOfFrame();
1839            // https://answers.unity.com/questions/755196/yield-return-null-
                  vs-yield-return-waitforendoffram.html
1840
1841            // GOOD:
1842            IEnumerator WaitOneFrameGood()
1843            {
```

```
1844                while (true)
1845                {
1846                    yield return null;
1847                }
1848            }
1849
1850            // BAD:
1851            IEnumerator WaitOneFrameBad()
1852            {
1853                while (true)
1854                {
1855                    yield return new WaitForEndOfFrame();
1856                }
1857            }
1858
1859            // DO: Be aware of living coroutines when you exit from a scene
                  or a game section
1860
1861            // EXAMPLE:
1862            void Open()
1863            {
1864                example.DummyGameObject.GetComponent<MonoBehaviour>
                      ().StartCoroutine(LivingCoroutine());
1865            }
1866
1867            void Close()
1868            {
1869                // NOTE that the LivingCoroutine is using this.example object
1870                this.example = null;
1871            }
1872
1873            IEnumerator LivingCoroutine()
1874            {
1875                // If Close is called this coroutine will keep being called
                      because it was launched in another GameObject (can happen
                      with a singleton for example)
1876                while (true)
1877                {
1878                    // CRASH!! This will crash after Close() was called
1879                    Debug.Log(this.example.DummyGameObject.name);
1880                }
1881            }
1882
1883            // DO: Launch coroutines using method references instead of
                  method names so we can always search for refences
1884
1885            void LaunchCoroutineExample()
1886            {
1887                // GOOD:
1888                dummyComponent.StartCoroutine(CoroutineExample());
1889
1890                // BAD:
1891                dummyComponent.StartCoroutine("CoroutineExample");
1892            }
1893
1894
1895            IEnumerator CoroutineExample()
1896            {
1897                yield return null;
1898            }
1899
1900            // DO: Stop coroutines using the coroutine reference not the
```

```
                coroutine name
1901
1902            void StopCoroutineExample()
1903            {
1904                var dummyCoroutine = dummyComponent.StartCoroutine
                      (CoroutineExample());
1905
1906                // GOOD:
1907                dummyComponent.StopCoroutine(dummyCoroutine);
1908
1909                // BAD:
1910                dummyComponent.StopCoroutine("CoroutineExample");
1911            }
1912
1913            // DO: If your project rely on a heavy use of coroutines,
                  consider doing a coroutine manager
1914            // Here is one example: https://assetstore.unity.com/packages/
                  tools/coroutine-manager-pro-53120
1915
1916            // DO: If you have to start a coroutine frome more than one place
                   or from other class, create a private method to start the
                  coroutine
1917            // doing this we avoid calling by accident the coroutine method
                  WITHOUT StartCoroutine which causes a silent error in Unity
1918
1919            // GOOD:
1920            public void RequestDataFromServer()
1921            {
1922                dummyComponent.StartCoroutine(RequestDataFromServerCoroutine
                      ());
1923            }
1924
1925            IEnumerator RequestDataFromServerCoroutine()
1926            {
1927                yield return null;
1928            }
1929
1930            // ================================
1931            // 10.2 - Field attributes
1932            // ================================
1933
1934            // CONSIDER: When declaring variables that can be accesed from
                  the Unity inspector, don't forget about Unity attributes.
1935            // This attributes will improve the inspector readibility.
1936
1937            // [Header("Header")] Useful to group related fields
1938            // [Space()] Useful to add extra space between groups of fields
1939            // [Range(min,max)]  Use this in floats, specially when you want
                  a normalized value between 0 and 1f
1940            // [Tooltip("Tooltip description")] Can replace fields comments,
                  this is also described in section 9 - Comments Conventions
1941        }
1942    }
1943
1944    #region Example class, ignore it
1945    public class Example
1946    {
1947        public const float GRAVITY_ACCELERATION = 9.8f;
1948
1949        public const float SPEED_MAX_DELTA = 0.1f;
1950
```

```
1951        public const int MAX_ELEMENTS = 99;
1952
1953        public const string LOCALIZED_TITLE = "TITLE_KEY";
1954
1955        public enum StatusEnum
1956        {
1957            Undefined,
1958            Ok,
1959            Error,
1960            Interrupted
1961        }
1962
1963        public DummyClass DummyObject { get; private set; }
1964
1965        public DummyClass OldDummyObject { get; private set; }
1966
1967        public GameObject DummyGameObject { get; private set; }
1968
1969        public StatusEnum Status { get; private set; }
1970
1971        public int dummyInt;
1972
1973        public string dummyString;
1974
1975        public bool DoOperations(DummyClass _dummyObject)
1976        {
1977            return true;
1978        }
1979
1980        public int GetRandomInt()
1981        {
1982            return 0;
1983        }
1984
1985        public int CalculateScore()
1986        {
1987            return 0;
1988        }
1989
1990        public bool IsLetter(char _char)
1991        {
1992            return true;
1993        }
1994
1995        public bool IsNumber(char _char)
1996        {
1997            return true;
1998        }
1999        public bool IsPunctiation(char _char)
2000        {
2001            return true;
2002        }
2003
2004        public string GetLocalizedText(string _textKey)
2005        {
2006            return "";
2007        }
2008
2009        public string GetGameModeFromServer()
2010        {
2011            return "";
2012        }
```

```
2013
2014        public class DummyClass
2015        {
2016            public string ID { get; private set; }
2017        }
2018
2019        public class MethodExamples
2020        {
2021            // Vague method name
2022            public void ComputeScore()
2023            {
2024
2025            }
2026
2027            // Good method name
2028            public void ComputeGameOverScore()
2029            {
2030
2031            }
2032
2033            // Good method name
2034            public void ComputeGameOverScoreAndUploadToServer()
2035            {
2036
2037            }
2038
2039            // method name without verb
2040            public void Score()
2041            {
2042
2043            }
2044
2045            public void GetScore()
2046            {
2047
2048            }
2049
2050            // Bad examples
2051            public string GetId()
2052            {
2053                return "";
2054            }
2055
2056            public string Id()
2057            {
2058                return "";
2059            }
2060
2061            public string ID { get; private set; }
2062
2063            // Parameters order
2064
2065            public void ConfigurePlayerSkin(bool _premiumSkins, ref
                  GameObject _instantiatedPlayer, out GameObject _mainWeapon, out
                   bool _error)
2066            {
2067                _error = false;
2068                _mainWeapon = new GameObject();
2069            }
2070
2071            // Examples of methods to pass variables to maintain interface
                  abstraction
2072            public void MethodWithGoodParameters(int _intParameter, string
```

```
                  _stringParameter)
2073              {
2074
2075              }
2076
2077              public void MethodWithBadParameters(DummyClass _dummyObject)
2078              {
2079
2080              }
2081          }
2082
2083      }
2084      #endregion
2085  }
```