```
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: Class names are in mixed uppercase and         ↵
                      lowercase with an initial capital letter. Never use plural ↵
                      in class names.
112
113                 // TypeName: Type definitions, including enumerated types,   ↵
                      use mixed uppercase and lowercase with an initial capital ↵
                      letter.
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 mixed uppercase and ↵
                      lowercase with an initial lowercase letter.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 (1). Use plural for       ↵
                      collections.
121
122                 // _methodParameter: MEthod parameters are formatted the same↵
                       as local variables but they are preceded with an under   ↵
                      slash
123
124                 // MethodName(): Methods are in mixed uppercase and          ↵
                      lowercase. Use plural if the method returns a collection.
125
126                 // CONSTANT_VAR: Named constants are in ALL_CAPS.
127
128                 // STATIC_VAR: Named statics are in ALL_CAPS.
129
130                 // PropertyName: Because properties are indeed methods, we   ↵
                      use the same convention as methods.
131
132                 // OnEventTriggered: Events names are in mixed uppercase and ↵
                      lowecase with an initial capital letter. All events should ↵
                      have a verb or verb phrase.
133
134                 // **** UNITY SPECIFICS ****
135
136                 // 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
137                 // modifying a prefab by error.
138
139                 // varNameTemplate: When a variable is a reference to some   ↵
```

```
                    component in the hierachy with the intention of being
                    copied using Instantiate(varNameTemplate),
140                 // 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!)
141
142                 // cachedComponentName: Use this to cached Unity components
                    like cachedTransform, cachedAnimator, etc.
143
144                 // ************************
145
146                 // NOTES:
147                 // (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
148                 // to differentiate between two vars with the same name is
                    needed
149             }
150
151             // ============================
152             // 1.2 - Class order
153             // ============================
154
155             public void ClassOrder()
156             {
157                 /* Within a class, struct, or interface, elements must be
                    positioned in the following order:
158
159                 1 - Constant Fields
160                 2 - Fields
161                 3 - Constructors
162                 4 - Finalizers (Destructors)
163                 5 - Delegates
164                 6 - Events
165                 7 - Enums
166                 8 - Properties
167                 9 - Indexers
168                 10 - Methods
169                     10.1 - Unity Methods
170                     10.2 - Handlers
171                     10.3 - Class methods
172                 11 - Structs
173                 12 - Classes
174
175                 Within each of these groups order by access:
176
177                 public
178                 internal
179                 protected internal
180                 protected
181                 private
182
183                 Within  each of the access groups, order by static, then non-
                    static:
184
185                 static
186                 non-static
187
188                 Within each of the static/non-static groups of fields, order
```

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

```
246             }
247         }
248
249
250         //                                                                    ↵
              ###########################################################################
251         // 2 - Layout & Style conventions
252         //                                                                    ↵
              ###########################################################################
253
254         private class LayoutConventions
255         {
256             public void Parentheses()
257             {
258                 // DO: Use parentheses to clarify expresions that involve     ↵
                      more than two terms
259
260                 // GOOD:
261                 int result1 = 5 * (6 + 4);
262
263                 // BAD:
264                 int result2 = 5 * 6 + 4;
265             }
266
267             public void CodeBlocks()
268             {
269                 // DO: Add a blank line for the begin brace
270
271                 // GOOD:
272                 if (example != null)
273                 {
274                     Debug.Log(example);
275                 }
276
277                 // BAD:
278                 if (example != null){
279                     Debug.Log(example);
280                 }
281
282                 // BAD:
283                 // This inline method make difficulty to debug because you    ↵
                      can't add a breakpoint inside the if
284                 if (example != null) { Debug.Log(example); }
285
286                 // DO: Use begin-end pairs ALWAYS to designate block          ↵
                      boundaries. This will avoid bugs related with incorrect    ↵
                      modifications
287
288                 // GOOD:
289                 if (example != null)
290                 {
291                     Debug.Log(example);
292                 }
293
294                 // BAD:
295                 if (example != null)
296                     Debug.Log(example);
297
298                 // DO: Use a new line in statements with complex expressions
299
300                 // GOOD:
```

```
301                 char inputChar = ' ';
302                 if ((('0' <= inputChar) && (inputChar <= '9'))
303                     || (('a' <= inputChar && inputChar <= 'z'))
304                     || (('A' <= inputChar && inputChar <= 'Z')))
305                 {
306
307                 }
308             }
309
310             private void IndividualStatements()
311             {
312                 // DO: Break large statements in several lines making obvious
                       the incompleteness of the statement.
313                 // If you have operators, put them at the beggining of the
                      new line so all of them stay aligned
314
315                 // GOOD:
316                 int arrayIndex1 = 0;
317                 int arrayIndex2 = 0;
318                 int[] exampleArray1 = new int[10];
319                 int[] exampleArray2 = new int[10];
320
321                 if (exampleArray1[arrayIndex1] == 0
322                     || exampleArray2[arrayIndex2] == 0
323                     || exampleArray1[arrayIndex1] == END_VALUE
324                     || exampleArray2[arrayIndex2] == END_VALUE)
325                 {
326
327                 }
328
329                 // BAD:
330                 if (exampleArray1[arrayIndex1] == 0 || exampleArray2
                      [arrayIndex2] == 0 || exampleArray1[arrayIndex1] ==
                      END_VALUE || exampleArray2[arrayIndex2] == END_VALUE)
331                 {
332
333                 }
334
335                 // DO: Break methods with lot of parameters in several lines
336                 // TODO: Decide with the team the limit of parameters to use
                      line breaks
337                 Vector3 rayOrigin       = Vector3.zero;
338                 Vector3 direction       = Vector3.up;
339                 RaycastHit raycastHit   = new RaycastHit();
340                 float maxDistance       = 99f;
341                 int layerMask           = LayerMask.GetMask("Default");
342
343                 // GOOD:
344                 bool hit = Physics.Raycast(rayOrigin,
345                     direction,
346                     out raycastHit,
347                     maxDistance,
348                     layerMask);
349
350                 // BAD:
351                 bool hit1 = Physics.Raycast(rayOrigin, direction, out
                      raycastHit, maxDistance, layerMask);
352
353                 // DO: Align right sides of assigment statements that belong
                      to a same code block (1*)
```

```
354                 // TODO: Discuss this with the team to decide if we keep
                      doing this or not
355
356                 // GOOD:
357                 int variableName1               = 0;
358                 int variableNameWithMoreLength  = 0;
359
360                 OnExampleEvent1                 +=
                      LayoutConventions_OnExampleEvent1;
361                 OnExampleEventWithLongerName    +=
                      LayoutConventions_OnExampleEvent2;
362                 OnExampleEvent3                 +=
                      LayoutConventions_OnExampleEvent3;
363
364                 // DO: Use only one data declaration per line
365
366                 // GOOD:
367                 int thisIsAnInt = 0;
368                 int thisIsAnotherInt = 0;
369
370                 // BAD:
371                 string firstName = null, lastName = null;
372
373                 // DO: Order declarations by type
374
375                 // GOOD:
376                 Vector3 velocity        = Vector3.zero;
377                 Vector3 acceleration    = Vector3.zero;
378                 float maxSpeedInKmH     = 120;
379
380                 // BAD:
381                 Vector3 velocity1       = Vector3.zero;
382                 float maxSpeedInKmH1    = 120; // <-- float is in the middle
                      of two vector3
383                 Vector3 acceleration1   = Vector3.zero;
384
385                 // Additional notes
386                 // (1*): Despite knowing it is not a recommended guidline
                      because it increases the cost to maintain the code when you
                       rename variables
387                 // i beleieve that the advantages of improved readibility and
                       the slightly chance of catching some errors make this rule
                       worth it.
388             }
389
390             private void LayoutConventions_OnExampleEvent1()
391             {
392                 throw new System.NotImplementedException();
393             }
394
395             private void LayoutConventions_OnExampleEvent2()
396             {
397                 throw new System.NotImplementedException();
398             }
399
400             private void LayoutConventions_OnExampleEvent3()
401             {
402                 throw new System.NotImplementedException();
403             }
404
```

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

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

```
509             }
510
511             // 10º - Methods
512
513             // 10.1 - Unity Methods
514
515             void Awake()
516             {
517
518             }
519
520             // DO: Provide services in pairs with their opposites
521             void OnEnable()
522             {
523                 OnClicked += ClassConventions_OnClicked;
524             }
525
526             void OnDisable()
527             {
528                 OnClicked -= ClassConventions_OnClicked;
529             }
530
531             // DO NOT: Leave blank Unity methods, delete them.
532
533             // 10.2 - Handlers
534
535             private void ClassConventions_OnClicked(GameObject _target)
536             {
537                 throw new System.NotImplementedException();
538             }
539
540             // 10.3 - Class methods
541
542             public void Open()
543             {
544
545             }
546
547             public void Close()
548             {
549
550             }
551
552             private void PrivateMethod()
553             {
554
555             }
556
557             // 11º - Structs
558
559             public struct NestedStruct
560             {
561                 public int exampleInt;
562             }
563
564             // 12º - Classes
565
566             private class NestedClass
567             {
568                 public int exampleInt = 0;
569             }
570
571             // ============================
572             // 3.2 - Class Names
```

```
573             // ============================
574
575             private void ClassNames()
576             {
577                 // DO: Use Pascal case for class names
578
579                 // DO: Use noun or noun phrase to name a class
580
581                 // DO: Use abbreviations sparingly.
582
583                 // 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.
584                 // For example, ApplicationException is an appropriate name
                      for a class derived from a class named Exception, because
                      ApplicationException is a kind of Exception.
585                 // Use reasonable judgment in applying this rule. For
                      example, Button is an appropriate name for a class derived
                      from Control.
586                 // Although a button is a kind of control, making Control a
                      part of the class name would lengthen the name
                      unnecessarily.
587
588                 // DO: Use the preffix I to naming Interfaces
589
590                 // DO NOT: Use Abstract or Base as prefix of Abstract
                      Classes. You should regular name conventions to define the
                      name of abstract classes.
591                 // GOOD:
592                 // Shape (abstract class) Square (child class), Circle (child
                       class)
593             }
594
595             // ============================
596             // 3.3 - Class Considerations
597             // ============================
598
599             private class ClassConsiderations
600             {
601                 // DO: Provide services in pairs with their opposites (Open &
                       Close, OnEnable & OnDisable, Enter & Exit, etc)
602
603                 // GOOD:
604                 public void Enter()
605                 {
606
607                 }
608
609                 public void Exit()
610                 {
611
612                 }
613
614                 public void Update()
615                 {
616
617                 }
618
619                 // BAD:
620                 public void Enter1()
621                 {
```

```
622
623                 }
624
625                 public void Update1()
626                 {
627
628                 }
629
630                 public void Exit1()
631                 {
632
633                 }
634
635                 // DO: Move unrelated information to another class
636
637                 // DO: Don’t expose member data in public
638
639                 // GOOD:
640                 [SerializeField]
641                 private GameObject explosionPrefab = null;
642
643                 // BAD:
644                 public GameObject deadParticlesPrefab = null;
645
646                 // DO: Minimize accessibility of classes and members (hide as much information as possible)
647
648                 // DO: Keep the number of methods in a class as small as possible. Consider creating new clasess to keep each class small
649
650                 // DO: Avoid deep inheritance trees
651
652                 // DO: Preserve the Law of Demeter which helps to keep low the coupling of the class with other classes:
653                 // Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
654                 // Following this principle implies avoiding this kind of lines:
                    otherClass.otherComponent.whateverThing.transform.position
655             }
656         }
657
658         //############################################################################
659         // 4 - Event Conventions
660         //############################################################################
661
662         private class EventConventions
663         {
664             // DO: Because the events always refer to some action, use verbs to name events.
665             // Use the verb tense to indicate the time the event is raised
666
667             // GOOD:
668             // Clicked, Painting, DroppedDown
669             public Action Click;
670             public Action Clicked;
671             public Action Closed;
672
```

```
673             // BAD:
674             public Action BeforeClose;
675             public Action AfterClose;
676 
677 
678             // DO: Call event handlers with the "EventHandler" suffix
679 
680             // GOOD;
681             public delegate void ClickEventHandler(Button _button);
682         }
683 
684         //                                                                    ↵
              ########################################################################
685         // 5 - Properties Conventions
686         //                                                                    ↵
              ########################################################################
687 
688         private class PropertiesConventions
689         {
690             // DO: Because properties refers to data, we should use noun      ↵
                  phrase or adjective names
691 
692             // GOOD:
693             public System.Object PlayerData {get; private set;}
694 
695             // BAD:
696             public System.Object GetPlayerData {get; private set;}
697 
698             // DO: Use plural if the property refers to a collection
699 
700             // GOOD:
701             public System.Object[] TeamMembers { get; private set; }
702 
703 
704             // DO: Name boolean properties with affirmative phrases or add a  ↵
                  prefix as "Is", "Can" or "Has"
705 
706             // GOOD:
707             public bool HasConnection {get; private set;}
708             public bool CanJump {get; private set;}
709         }
710 
711         //                                                                    ↵
              ########################################################################
712         // 6 - Methods Conventions
713         //                                                                    ↵
              ########################################################################
714 
715         private class MethodsConventions
716         {
717             Example examples = new Example();
718             Example.MethodExamples examples2 = new Example.MethodExamples();
719 
720             // ================================
721             // 6.1 Methods Naming Conventions
722             // ================================
723 
724             // DO: A method name must describe everything the method does
725 
726             // GOOD
```

```
727             void ComputeGameOverScoreAndUploadToServer()
728             {
729
730             }
731
732             // BAD
733             void GameOverScore()
734             {
735
736             }
737
738             // DO NOT: Use meaningless, vague or wishy-washy verbs
739
740             // GOOD:
741             void UploadGameResults()
742             {
743
744             }
745
746             // BAD:
747             void Upload()
748             {
749
750             }
751
752             // DO NOT: Don't use numbers to differentiate method names
753
754             // GOOD:
755             void PlaySound(AudioClip _clip)
756             {
757
758             }
759
760
761             void PlaySound(AudioClip _clip,float _volume)
762             {
763
764             }
765
766             // BAD:
767             void PlaySound1(AudioClip _clip)
768             {
769
770             }
771
772             void PlaySound2(AudioClip _clip, float _volume)
773             {
774
775
776             }
777
778             // DO: Make names of methods as long as neccesary
779
780             // DO: Use verbs for method names
781
782             // DO: If the function returns a value, to name a function use a
                  description of the value (only when it returns a single value)
783
784             // DO: Stablish conventions for common operations
785
786             // BAD:
787
788             int GetId()
789             {
```

```
790                 return 0;
791             }
792
793             int Id()
794             {
795                 return 0;
796             }
797
798             // ================================
799             // 6.2 - method Parameters Conventions
800             // ================================
801
802             private void MethodParametersOrder()
803             {
804                 // DO: Put parameters in input-modify-output-
805                 // in the output parameters, put error/status order last
806
807                 // GOOD
808                 GameObject mainWeapon = null;
809                 GameObject instantiatedPlayer = null;
810                 bool skinConfigurationError = false;
811                 examples2.ConfigurePlayerSkin(true, ref instantiatedPlayer,
                      out mainWeapon, out skinConfigurationError);
812             }
813
814             // DO: method parameters start with a underscore '_'
815
816             // GOOD:
817             void ExampleMethod(int _parameter)
818             {
819
820             }
821
822             // DO: If similar methods use similar parameters, put the similar
                   parameters in a consistent order
823
824             // DO: Use all the parameters, remove unused parameters
825
826             // DO NOT: Don't use method parameters as working variables, use
                  local variables instead
827
828             // BAD:
829             int MathOperationExampleBad(int _value)
830             {
831                 _value *= 5;
832
833                 return _value;
834             }
835
836             // GOOD:
837             int MathOperationExampleGood(int _value)
838             {
839                 int result = _value * 5;
840
841                 return _value;
842             }
843
844             // DO: Limit the number of method's parameters to about seven. If
                   you need more parameters probably you need
845             // a new class/struct to represent that data
846
```

```
847             // EXAMPLE:
848             public void MethodParametersLimit(int value1, int value2, int
                  value3, int value4, int value5, int value6, int value7)
849             {
850
851             }
852
853             // DO: Pass the variables or objects that the method needs to
                  maintain its interface abstraction
854
855             void ParatemersAbstractionExample()
856             {
857                 // GOOD:
858                 examples2.MethodWithGoodParameters(examples.dummyInt,
                      examples.dummyString);
859
860                 // BAD:
861                 examples2.MethodWithBadParameters(examples.DummyObject); //
                      <- this method doesn't need to receive the whole object,
                      only needs the int and string
862             }
863
864             // DO: Use named parameters in lambda expresions and auto-
                  generated delegates
865
866             void LambdaExpressionExample()
867             {
868                 List<int> example = new List<int> ();
869
870                 // GOOD:
871                 example.FindAll(delegate (int _index)
872                 {
873                     return _index > 2;
874                 });
875
876                 // BAD:
877                 example.FindAll(delegate (int _x)
878                 {
879                     return _x > 2;
880                 });
881             }
882
883             // DO: Check input paramaters before assignation. Make sure that
                  the values are reasonable
884             // You can use asserts if you don't want this checks in your
                  release versions in situations where you need performance
885             void MethodParametersCheck(Example _exampleObject)
886             {
887                 if(_exampleObject == null)
888                 {
889                     Debug.LogError("The parameter _exampleObject can't be
                        null");
890                     return;
891                 }
892
893                 Debug.LogFormat("Status: {0}",_exampleObject.Status);
894             }
895
896             // ================================
897             // 6.3 - Methods return value conventions
898             // ================================
```

```
899
900             // DO: Have a single return point in your method, this will help
                  to maintain and debug your code
901
902             // GOOD:
903             public int ExampleMethod1()
904             {
905                 // DO: Initialize the return value at the beggining of the
                      function to a default value
906                 int returnValueExample = 0;
907
908                 bool condition1 = false;
909                 bool condition2 = false;
910
911                 if (condition1)
912                 {
913                     returnValueExample = 1;
914                 }
915                 else if (condition2)
916                 {
917                     returnValueExample = 2;
918                 }
919
920                 return returnValueExample;
921             }
922
923             // BAD:
924             public int ExampleMethod2()
925             {
926                 bool condition1 = false;
927                 bool condition2 = false;
928
929                 if (condition1)
930                 {
931                     return 1;
932                 }
933                 else if (condition2)
934                 {
935                     return 2;
936                 }
937
938                 return 0;
939             }
940
941             // ================================
942             // 6.4 Method considerations
943             // ================================
944
945             private void MethodsConsiderations()
946             {
947                 // DO: methods must have a single purpose
948
949                 // 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
950                 // so there is not an official standard in the industry. Try
                      to avoid large methods because usually, they are
                      consequences of bad programming practices
951                 // but you can always write methods over 200 lines if you
                      need it and they are simple enough to be readable,
                      maintenable and undertestanable.
```

```
952             }
953
954         }
955
956         //                                                                    ↵
              ##########################################################################
957         // 7 - Variable Conventions
958         //                                                                    ↵
              ##########################################################################
959
960         private class VariableConventions
961         {
962             // ================================
963             // 7.1 Variable Declarations
964             // ================================
965
966             private void VariableDeclarations()
967             {
968                 // DO: Initialize all variables
969
970                 // GOOD:
971                 int myInt = 0;
972                 GameObject myGameObject = null;
973
974                 // BAD:
975                 int anotherInt;
976                 GameObject targetGameobject;
977
978
979                 // DO: Initialize each variable close to where it's firts     ↵
                      used to
980                 // preserve the Principle of Proximity: keep related actions  ↵
                      together
981
982                 // GOOD:
983                 string id = example.DummyObject.ID;
984                 Debug.LogFormat("Id: {0}", id);
985
986                 string secondID = example.DummyObject.ID;
987                 Debug.LogFormat("secondID: {0}", secondID);
988
989
990                 // BAD:
991                 string sourceId = example.DummyObject.ID;
992                 string targetId = example.DummyObject.ID;
993
994                 Debug.LogFormat("SourceId: {0}", sourceId);
995                 Debug.LogFormat("TargetId: {0}", targetId);
996
997                 // CONSIDER: declaring and defining each variable where it's  ↵
                      first used:
998
999                 // GOOD:
1000                Example.StatusEnum status1 = example.Status;
1001
1002                // BAD:
1003                Example.StatusEnum status2 = Example.StatusEnum.Undefined;
1004                status2 = example.Status;
1005
1006                // DO: Group related statements
1007
```

```
1008                // GOOD:
1009                var dummyObject = example.DummyObject; // <-- Dummy Object
1010                bool dummyObjectOK = example.DoOperations(dummyObject); //
                      <-- Operate over Dummy Object
1011                var oldDummyObject = example.OldDummyObject; // <-- Old Dummy
                       Object
1012                bool oldDummyObjectOK = example.DoOperations
                      (oldDummyObject); // <-- Operate over Old dummy object
1013
1014
1015                // DO NOT:
1016                var dummyObject1 = example.DummyObject; // <-- Dummy Object
1017                var oldDummyObject1 = example.OldDummyObject; // <-- Old
                      Dummy Object
1018                bool oldDummyObjectOK1 = example.DoOperations
                      (oldDummyObject1); // <-- Operate over Old dummy object
1019                bool dummyObjectOK1 = example.DoOperations(dummyObject1); //
                      <-- Operate over Dummy Object
1020
1021                // DO: Use each variable for one purpose only to improve
                      readibility
1022
1023                // GOOD:
1024                int randomIndex = example.GetRandomInt();
1025                Debug.LogFormat("Random Index: {0}", randomIndex);
1026
1027                int score = example.CalculateScore();
1028                Debug.LogFormat("Score: {0}", score);
1029
1030                // BAD:
1031                int tempValue = example.GetRandomInt();
1032                Debug.LogFormat("Random Index: {0}", tempValue);
1033
1034                tempValue = example.CalculateScore();
1035                Debug.LogFormat("Score: {0}", tempValue);
1036
1037
1038                // DO NOT: Set variables with hidden meanings to avoid
                      "hybrid coupling"
1039
1040                // GOOD:
1041                var status = example.Status;
1042                if (status == Example.StatusEnum.Ok)
1043                {
1044                    int finalScore = example.CalculateScore();
1045                }
1046                else
1047                {
1048                    Debug.LogError("Some error occured. The game doesn't
                       ended well");
1049                }
1050
1051                // BAD:
1052                int gameOverScore = example.CalculateScore();
1053                if (gameOverScore == -1) // Two meanings for gameOverScore:
                      game score & error
1054                {
1055                    Debug.LogError("Some error occured. The game doesn't
                       ended well");
```

```
1056                }
1057
1058
1059                // DO: Make sure all declared variables are used.
1060                // You can see Warnings in the console to remove unused
                      variables.
1061
1062
1063                // DO: Use 'var' keyword in variable declarations except for
                      numeric variables
1064                // - Code maintenance is improved
1065                // - Code readability is improved
1066                // TODO: Discuss this with the team
1067
1068                // GOOD:
1069                var target = example.DummyGameObject;
1070
1071                // BAD:
1072                GameObject source = example.DummyGameObject;
1073
1074                // With numeric values:
1075
1076                // BAD:
1077                var totalPlays = 5;
1078                var totalScore = example.CalculateScore();
1079                var averageScore = totalScore / totalPlays; // <-- we can
                      have errors because we don't know the type of the variables
                       (float, int, etc)
1080
1081
1082                // Note (1): If you are concerned about performance, you can
                      use Assertions (Unity Debug.Assert)
1083            }
1084
1085            // ================================
1086            // 7.2 Variable Names
1087            // ================================
1088
1089            private void VariableNames()
1090            {
1091                // DO: Try to avoid computer related terms and use problem
                      domain terms.
1092
1093                // GOOD:
1094                System.Object playerSaveGame = new System.Object();
1095
1096                // BAD:
1097                System.Object savedData = new System.Object();
1098
1099                // DO NOT: Use too short, too long, hard to type, hard to
                      pronounce variable names
1100
1101                // CONSIDER: Using variables with a name length: Between 8
                      and 20 characters
1102                // ABCDEFGH <-- 8 characters
1103                // ABCDEFGHIJKLMNOPQRST <-- 20 characters
1104
1105                // DO: If the variable has a qualifer like: Total, Sum,
                      Average, Max, Min, Record, String, Pointer, etc
1106                // put it at the end of the name
```

```
1107
1108                // GOOD:
1109                int scoreTotal = 0;
1110                float healthAverage = 0;
1111                Text titleLabel = null;
1112
1113                // CONSIDER: Using more descriptive names for loops indexs to
                       improve readibility
1114
1115                // GOOD:
1116                System.Object[] clamMembers = new System.Object[10];
1117
1118                for (int clanMemberIdx = 0; clanMemberIdx <
                      clamMembers.Length; clanMemberIdx++)
1119                {
1120
1121                }
1122
1123                // BAD:
1124                for (int i = 0; i < clamMembers.Length; i++)
1125                {
1126
1127                }
1128
1129                // DO: Use more detailed index names for nested loops to
                      avoid index corss-talk errors (saying i when you mean j and
                       vice versa)
1130
1131                // GOOD:
1132                int[,] scores = new int[5, 5];
1133                int teamCount = 5;
1134                int eventCount = 5;
1135                for (int teamIndex = 0; teamIndex < teamCount; teamIndex++)
1136                {
1137                    for (int eventIndex = 0; eventIndex < eventCount;
                       eventIndex++)
1138                    {
1139                        scores[teamIndex, eventIndex] = 0;
1140                    }
1141                }
1142
1143                // DO: Give boolean variables names that imply true or false
                      as:
1144                // done, error, found, success, ok
1145
1146                // DO: Use positive boolean variable names:
1147
1148                // GOOD:
1149                bool found = false;
1150
1151                // BAD:
1152                bool notFound = true;
1153
1154
1155                // DO NOT: Use names with similar meanings
1156
1157                // GOOD:
1158                int fileCount = 0;
1159                int fileIndex = 1;
1160
1161                // BAD:
```

```
1162                int fileNumber = 0;
1163                int fileIndex1 = 0;
1164
1165
1166                // DO NOT: Use similar names in variables with different
                      meaning
1167
1168                // GOOD:
1169                Rect screenArea = new Rect();
1170                Vector2 screenResolution = new Vector2();
1171
1172                // BAD:
1173                Rect screenRect = new Rect();
1174                Vector2 screenRes = new Vector2();
1175
1176                // DO: Avoid names that sound similar
1177
1178                // DO NOT: Use numerals in names. If you feel you need
                      numerals probably you need a different data type as an
                      array
1179
1180                // GOOD:
1181                GameObject[] itemSlots = null;
1182
1183                // BAD:
1184                GameObject itemSlot1 = null;
1185                GameObject itemSlot2 = null;
1186            }
1187
1188            // ================================
1189            // 7.3 Numbers
1190            // ================================
1191
1192            private void NumericVariables()
1193            {
1194                // DO: Avoid magic numbers
1195
1196                // GOOD:
1197                float initialSpeed = 10f;
1198                float timeInSeconds = 60f;
1199                float fallSpeed = initialSpeed + Example.GRAVITY_ACCELERATION
                       * timeInSeconds;
1200
1201                // BAD:
1202                float fallSpeed1 = initialSpeed + 9.8f * timeInSeconds;
1203
1204                // DO: Anticipate divide-by-zero errors
1205
1206                // DO: Make data type conversions explicit in mathematical
                      operations.
1207
1208                // GOOD:
1209                float x = 0f;
1210                int i = 0;
1211                float y = x + (float)i;
1212
1213                // BAD:
1214                float w = x + i;
1215
1216                // DO NOT: Use mixed-type comparisons
```

```
1217
1218                // GOOD:
1219                if (i == (int)x)
1220                {
1221
1222                }
1223
1224                // BAD:
1225                if (i == x)
1226                {
1227
1228                }
1229
1230                // DO: Be careful with integer divisions
1231                float result = 10f * (7 / 10);  // <-- will return 0 in C#
1232
1233                // CONSIDER: Integer overflow, you should think about the
                      largest value your expression can assume.
1234
1235                // DO NOT: add/substract on numbers that have greatly
                      different magnitudes with float numbers
1236                float result1 = 1000000.00f + 0.1f; // <--  can have a result
                      of 1,000,000.00
1237
1238                // DO NOT: Use equality comparisons in floating-point numbers
1239
1240                // GOOD:
1241                float maximumSpeed = 0f;
1242                float speed = 0f;
1243                if (maximumSpeed - speed < Example.SPEED_MAX_DELTA)
1244                {
1245
1246                }
1247
1248                // BAD:
1249                if (speed == maximumSpeed)
1250                {
1251
1252                }
1253            }
1254
1255            // ================================
1256            // 7.4 Characters And Strings
1257            // ================================
1258
1259            private void CharacterAndStrings()
1260            {
1261                // DO: Avoid magic strings
1262
1263                // GOOD:
1264                string localizedTitle = example.GetLocalizedText
                      (Example.LOCALIZED_TITLE);
1265
1266                // BAD:
1267                string localizedTitle1 = example.GetLocalizedText
                      ("TITLE_KEY");
1268
1269                // DO: Use string.Format() to compose strings
1270
1271                // GOOD:
1272                int score = 0;
1273                string finalText1 = string.Format("You have won {0} points",
```

```
                    score);
1274
1275                // BAD:
1276                string finalText2 = "You have won " + score + " points";
1277
1278                // **** UNITY SPECIFICS ****
1279
1280                // DO: Use Debug.LogFormat, Debug.LogErrorFormat &
                      Debug.LogWarningFormat to print logs with composes messages
1281
1282                // ************************
1283
1284                // DO: Use string.IsNullOrEmpty to check for empty/null
                      strings
1285
1286                // GOOD:
1287                string userName = "";
1288                if (string.IsNullOrEmpty(userName))
1289                {
1290
1291                }
1292
1293                // BAD:
1294                if (userName == null || userName == "")
1295                {
1296
1297                }
1298
1299                // DO: Consider using the StringBuilder class if you need to
                      work with long strings to reduce the impact on performance
1300
1301                // GOOD:
1302                string text = null;
1303                System.Text.StringBuilder sb = new System.Text.StringBuilder
                      ();
1304                for (int i = 0; i < 100; i++)
1305                {
1306                    sb.AppendLine(i.ToString());
1307                }
1308
1309                // BAD:
1310                for (int i = 0; i < 100; i++)
1311                {
1312                    text += i.ToString();
1313                }
1314            }
1315
1316            // ================================
1317            // 7.5 Booleans
1318            // ================================
1319
1320            private void Booleans()
1321            {
1322                // DO: Use boolean variables to increase readibility and
                      maintenance in logic expressions
1323
1324                // GOOD:
1325                int elementIndex = 0;
1326                int lastElementIndex = 0;
1327
1328                bool finished = ((elementIndex < 0) || (Example.MAX_ELEMENTS
```

```
                    < elementIndex));
1329                bool repeatedEntry = (elementIndex == lastElementIndex);
1330
1331                if (finished || repeatedEntry)
1332                {
1333
1334                }
1335
1336                // BAD:
1337                if ((elementIndex < 0) || (Example.MAX_ELEMENTS <
                      elementIndex) || (elementIndex == lastElementIndex))
1338                {
1339
1340                }
1341            }
1342
1343            // ================================
1344            // 7.6 Enums
1345            // ================================
1346
1347            // DO: Define always the first value in an enum for an "invalid"
                  value and the last value to use it to iterate over all enum
                  values
1348
1349            // GOOD:
1350            public enum GameModes
1351            {
1352                None,
1353                DeathMatch,
1354                TeamDeathMatch,
1355                CaptureTheFlag,
1356                End,
1357            }
1358
1359            // **** UNITY SPECIFICS ****
1360
1361            // 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
1362
1363            // GOOD:
1364            private enum GameModes1
1365            {
1366                None,
1367                DeathMatch,
1368                TeamDeathMatch,
1369                CaptureTheFlag,
1370                NewGameMode, // <-- Added at the end (before Final)
1371                Final,
1372            }
1373
1374            // BAD:
1375            private enum GameModes2
1376            {
1377                None,
1378                DeathMatch,
1379                NewGameMode, // <-- Added in the middle, now the seralizable
                      fields with the value of TeamDeatchMatch will have
                      NewGameMode assigned
1380                TeamDeathMatch,
```

```
1381                CaptureTheFlag,
1382                Final,
1383            }
1384
1385            // ************************
1386
1387
1388            private void UsingEnums()
1389            {
1390                // DO: Try to use the same enum values in external services to avoid name conversions
1391
1392                // GOOD:
1393                string gameModeString = example.GetGameModeFromServer();
1394                GameModes gameModeFromServer = (GameModes)Enum.Parse(typeof(GameModes), gameModeString);
1395                if (gameModeFromServer == GameModes.DeathMatch)
1396                {
1397
1398                }
1399
1400                // BAD:
1401                string gameModeString1 = example.GetGameModeFromServer();
1402
1403                if (gameModeString1 == "DEATH_MATCH")
1404                {
1405                    gameModeFromServer = GameModes.DeathMatch;
1406                }
1407            }
1408
1409            // ================================
1410            // 7.7 Arrays
1411            // ================================
1412
1413
1414            public void Arrays()
1415            {
1416                // DO: Check that array indexes are within the bounds of the array
1417
1418                // GOOD:
1419                GameObject[] exampleArray = new GameObject[5];
1420                int index = 0;
1421                int maxLength = exampleArray.Length;
1422
1423                if (index >= maxLength)
1424                {
1425                    Debug.LogErrorFormat("Array out of range. Index: {0} Max: {1}", index, maxLength);
1426                    index = maxLength;
1427                }
1428
1429                // DO: Use lists if you don't know the size of the array, in other case, use arrays
1430            }
1431
1432            private Example example = new Example();
1433        }
1434
1435        // ##########################################################################
1436        // 8 - Statements Conventions
```

```
1437        //  ########################################################################
1438
1439        private class StatementsConventions
1440        {
1441            // ============================
1442            // 8.1 - Conditionals
1443            // ============================
1444
1445            private Example example = new Example();
1446
1447            private void Conditionals()
1448            {
1449                // DO: Write the nominal path through the code first, then write unusual cases. This improves readability and performance
1450
1451                // DO: Write errors case outside the conditional where you are taking the decissions
1452
1453                // GOOD:
1454                bool error = false;
1455
1456                if(error)
1457                {
1458                    Debug.LogError("Notify of error");
1459                    return;
1460                }
1461
1462                bool condition1 = false;
1463
1464                if (condition1)
1465                {
1466                    // Do something
1467                }
1468
1469                // BAD:
1470                if (error)
1471                {
1472                    Debug.LogError("Notify of error");
1473                }
1474                else
1475                {
1476                    if (condition1)
1477                    {
1478                        // Do something
1479                    }
1480                }
1481
1482                // DO: Put the normal case after the if rather than after the else
1483
1484                // GOOD:
1485                if (example.Status == Example.StatusEnum.Ok)
1486                {
1487                    // Normal case
1488                }
1489                else
1490                {
1491
1492                }
1493
```

```
1494                // BAD:
1495                if (example.Status != Example.StatusEnum.Ok)
1496                {
1497
1498                }
1499                else
1500                {
1501                    // Normal case
1502                }
1503
1504                // CONSIDER: Consider to write the else clause to make clear
                      to other programmers that you have considered that case
1505
1506                // GOOD:
1507                bool validID = !string.IsNullOrEmpty(example.DummyObject.ID);
1508                if (validID)
1509                {
1510                    // Normal case
1511                }
1512                else
1513                {
1514                    // If the id is not valid nothing happens here
1515                }
1516
1517                // BAD:
1518                if (validID)
1519                {
1520                    // Normal case
1521                }
1522
1523                // DO: Simplify complicated tests with boolean functions
                      calls
1524
1525                // GOOD:
1526                bool statusIsOk = (example.Status == Example.StatusEnum.Ok);
1527                if(validID && statusIsOk)
1528                {
1529                    // Do something
1530                }
1531
1532                // BAD:
1533                if (!string.IsNullOrEmpty(example.DummyObject.ID) &&
                      example.Status == Example.StatusEnum.Ok)
1534                {
1535                    // Do somethinf
1536                }
1537
1538                // DO: In several if/else if statements, start always for the
                       most frequent cases first
1539
1540                // GOOD:
1541                char inputChar = ' ';
1542                if (example.IsLetter(inputChar))
1543                {
1544                    // Do something with letter
1545                }
1546                else if(example.IsNumber(inputChar))
1547                {
1548                    // Do something with number
1549                }
1550                else if(example.IsPunctiation(inputChar))
```

```
1551                {
1552                    // Do something with punctuations
1553                }
1554                else
1555                {
1556                    Debug.LogErrorFormat("Unrecognized char: {0}",
                        inputChar);
1557                }
1558            }
1559
1560            // ============================
1561            // 8.2 - Loops
1562            // ============================
1563
1564            public void Loops()
1565            {
1566                List<string> exampleNames = new List<string>();
1567
1568                // DO: Put initialization code directly before the loop
1569
1570                // EXAMPLE:
1571                bool enc    = false;
1572                int index   = 0;
1573                while (!enc)
1574                {
1575                    enc = exampleNames[index] == "Unnamed";
1576                    index++;
1577                }
1578
1579                // **** UNITY SPECIFICS ****
1580                // DO: Prefer for loops instead of foreach loops if possible.
                       For loops ar faster in Unity that foreach loops
1581
1582                // GOOD:
1583                for (int i = 0; i < exampleNames.Count; i++)
1584                {
1585                    Debug.Log(exampleNames[i]);
1586                }
1587
1588                // BAD:
1589                foreach(var name in exampleNames)
1590                {
1591                    Debug.Log(name);
1592                }
1593
1594                // ************************
1595
1596                // DO: When the number of iterations is indefinite, use a
                      whille loop
1597
1598                // DO: Use { and } to enclose statements in a loop always to
                      prevent errors when code is modified.
1599
1600                // GOOD:
1601                for (int i = 0; i < 100; i++)
1602                {
1603                    Debug.Log(i);
1604                }
1605
1606                // BAD:
1607                for (int i = 0; i < 100; i++)
```

```
1608                    Debug.Log(i);
1609
1610                // DO: Keep loop-housekeeping chores at either the beginning or the end of the loop
1611
1612                // DO: Make each loop perform only one function. Loops should be like methods.
1613                // An exception of this rule it is in places where performance is critical
1614
1615                // DO: Make sure the loop ends. This is specially important in some while loops that are potentially dangerous as infinite loops
1616
1617                // GOOD:
1618                float securityTimer = 0f;
1619                while (example.Status != Example.StatusEnum.Ok && securityTimer < 10f)
1620                {
1621                    // Do something
1622
1623                    securityTimer += Time.deltaTime;
1624                }
1625
1626                if(securityTimer >= 10f)
1627                {
1628                    Debug.Log("TimeOut");
1629                }
1630
1631                // BAD:
1632                while (example.Status != Example.StatusEnum.Ok)
1633                {
1634
1635                }
1636
1637                // DO: If you need to use continue in a for loop, do it at the top of the loop
1638
1639                // GOOD:
1640                for(int i = 0; i < 100; i++)
1641                {
1642                    if (example.Status != Example.StatusEnum.Ok)
1643                        continue;
1644                }
1645
1646                // DO: Use meaningful variable names to make nested loops readable
1647
1648                // GOOD:
1649                for(int month = 0; month < 12; month++)
1650                {
1651                    for(int day = 0; day < 31; day++)
1652                    {
1653
1654                    }
1655                }
1656
1657                // BAD:
1658                for (int i = 0; i < 12; i++)
1659                {
1660                    for (int j = 0; j < 31; j++)
```

```
1661                    {
1662
1663                    }
1664                }
1665
1666                // DO: Limit nesting of loop to three levels. Break the loops into methods if you need it to avoid this
1667
1668                // DO: Make long loops specially clear
1669            }
1670        }
1671
1672        //////////////////////////////////////////////////////////////////////////
1673        // 9 - Comments conventions
1674        //////////////////////////////////////////////////////////////////////////
1675
1676        private class CommentsConventions
1677        {
1678            private Example example = new Example();
1679
1680            // 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,
1681            // it is better to improve the code that to write the comment
1682
1683            // DO NOT: Write comments that repeats what the code does
1684
1685            // DO: Write comments that summary the code to help other programmers at reading the code
1686
1687            // 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
1688
1689            // GOOD:
1690            // get current employee information
1691
1692            // BAD:
1693            // get employee object from database query
1694
1695            // DO NOT: Leave big regions of code commented when you commit to developer. Use the version repository to see old code.
1696
1697            // 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
1698            // to think more about the problem you are triying to solve
1699
1700            // DO NOT: Use endline comments (except for data declarations or end of blocks)
1701
1702            // GOOD
1703
1704            /// <summary>
1705            /// Explain what this method does
1706            /// </summary>
1707            private void ExampleMetho1d()
1708            {
1709
```

```
1710            }
1711
1712            // BAD
1713            private void ExampleMethod2() // Explain what this method does
1714            {
1715
1716            }
1717
1718            // DO: Use endline comment to annotate data declarations
1719
1720            // GOOD
1721            private int playerIndex = 0; // Index of the player in the list
                  of players of this lobby
1722
1723            // **** UNITY SPECIFICS ****
1724            // 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
1725            // isn't clear with the name.
1726            // This will replace the regular commenting style
1727            // GOOD
1728            [Tooltip("Use this variable to declare the maximum value of the
                  player health. Useful when implementing penalties")]
1729            [SerializeField]
1730            public float maximumHealthClamp = 1f;
1731            // ********************
1732
1733            // 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.
1734
1735            // GOOD
1736
1737            private void ExampleMethod3()
1738            {
1739                if (example.Status == Example.StatusEnum.Ok)
1740                {
1741                    // JUST IMAGINE A VERY LARGE IF BLOCK
1742                    // ...
1743                    // ...
1744                    // ...
1745                    // ...
1746                    // ...
1747
1748                } // End of STatus == Ok condition
1749            }
1750
1751            // DO: Use comment to justify violations of good programming
                  style
1752
1753            // 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
1754
1755            // DO: Comment units of numeric data (no matter if the unit of
                  the data form part of the variable name)
1756
1757            // GOOD:
1758
1759            public Vector3 spaceshipVelocity = Vector3.zero; // Spaceship
                  velocity vector in meters per second
```

```
1760
1761            // DO: Comment the range of allowable numeric values
1762
1763            // GOOD:
1764
1765            public float normalizedProgress = 0f; // Player normalized progress in this level between 0 and 1
1766
1767            // CONSIDER: Commenting enums values if they are not obvious
1768
1769            // DO: When commenting class members as methods and properties, use the default format in c# (automatically generated if you write ///)
1770
1771            // GOOD:
1772
1773            /// <summary>
1774            /// Summary of the method here
1775            /// </summary>
1776            /// <param name="_param1"> Describe the parameter, describe expected values, units, etc</param>
1777            /// <returns> Describe the return value </returns>
1778            public int Method1(int _param1)
1779            {
1780                return 0;
1781            }
1782
1783            // Because to much comments decrease code readability, we should rely in clean code, small & simple classes instead of commenting everything.
1784            // in order to know what to comment, follow the next guide:
1785
1786            // Fields: because class fields are protected or private, you should only comment fields that aren't clear enough with the field name
1787            // 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
1788            // Private methods/properties: They only should be commented if their purpose isn't clear
1789
1790            // DO: Comment classes describing their design approach, limitations, usage assumptions and so on
1791        }
1792
1793        //////////////////////////////////////////////////////////////////////////
1794        // 10 - Unity Special Conventions
1795        //////////////////////////////////////////////////////////////////////////
1796
1797        private class UnityConventions
1798        {
1799            Example example = new Example();
1800
1801            private MonoBehaviour dummyComponent = null;
1802
1803            // ================================
1804            // 10.1 - Coroutines
1805            // ================================
```

```
1806
1807            // 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()"
1808            // doing this you will avoid memory allocation in each execution of the loop
1809
1810            // Note: It's important to know that  yield return null is not evaluated in the same moment than WaitForEndOfFrame();
1811            // https://answers.unity.com/questions/755196/yield-return-null-vs-yield-return-waitforendoffram.html
1812
1813            // GOOD:
1814            IEnumerator WaitOneFrameGood()
1815            {
1816                while (true)
1817                {
1818                    yield return null;
1819                }
1820            }
1821
1822            // BAD:
1823            IEnumerator WaitOneFrameBad()
1824            {
1825                while (true)
1826                {
1827                    yield return new WaitForEndOfFrame();
1828                }
1829            }
1830
1831            // DO: Be aware of living coroutines when you exit from a scene or a game section
1832
1833            // EXAMPLE:
1834            void Open()
1835            {
1836                example.DummyGameObject.GetComponent<MonoBehaviour>().StartCoroutine(LivingCoroutine());
1837            }
1838
1839            void Close()
1840            {
1841                // NOTE that the LivingCoroutine is using this.example object
1842                this.example = null;
1843            }
1844
1845            IEnumerator LivingCoroutine()
1846            {
1847                // If Close is called this coroutine will keep being called because it was launched in another GameObject (can happen with a singleton for example)
1848                while (true)
1849                {
1850                    // CRASH!! This will crash after Close() was called
1851                    Debug.Log(this.example.DummyGameObject.name);
1852                }
1853            }
1854
1855            // DO: Launch coroutines using method references instead of method names so we can always search for refences
1856
```

```
1857            void LaunchCoroutineExample()
1858            {
1859                // GOOD:
1860                dummyComponent.StartCoroutine(CoroutineExample());
1861
1862                // BAD:
1863                dummyComponent.StartCoroutine("CoroutineExample");
1864            }
1865
1866
1867            IEnumerator CoroutineExample()
1868            {
1869                yield return null;
1870            }
1871
1872            // DO: Stop coroutines using the coroutine reference not the
                  coroutine name
1873
1874            void StopCoroutineExample()
1875            {
1876                var dummyCoroutine = dummyComponent.StartCoroutine
                      (CoroutineExample());
1877
1878                // GOOD:
1879                dummyComponent.StopCoroutine(dummyCoroutine);
1880
1881                // BAD:
1882                dummyComponent.StopCoroutine("CoroutineExample");
1883            }
1884
1885            // DO: If your project rely on a heavy use of coroutines,
                  consider doing a coroutine manager
1886            // Here is one example: https://assetstore.unity.com/packages/
                  tools/coroutine-manager-pro-53120
1887
1888            // 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
1889            // doing this we avoid calling by accident the coroutine method
                  WITHOUT StartCoroutine which causes a silent error in Unity
1890
1891            // GOOD:
1892            public void RequestDataFromServer()
1893            {
1894                dummyComponent.StartCoroutine(RequestDataFromServerCoroutine
                      ());
1895            }
1896
1897            IEnumerator RequestDataFromServerCoroutine()
1898            {
1899                yield return null;
1900            }
1901
1902            // ================================
1903            // 10.2 - Field attributes
1904            // ================================
1905
1906            // CONSIDER: When declaring variables that can be accesed from
                  the Unity inspector, don't forget about Unity attributes.
1907            // This attributes will improve the inspector readibility.
1908
```

```
1909            // [Header("Header")] Useful to group related fields
1910            // [Space()] Useful to add extra space between groups of fields
1911            // [Range(min,max)]  Use this in floats, specially when you want
                  a normalized value between 0 and 1f
1912            // [Tooltip("Tooltip description")] Can replace fields comments,
                  this is also described in section 9 - Comments Conventions
1913        }
1914    }
1915
1916    #region Example class, ignore it
1917    public class Example
1918    {
1919        public const float GRAVITY_ACCELERATION = 9.8f;
1920
1921        public const float SPEED_MAX_DELTA = 0.1f;
1922
1923        public const int MAX_ELEMENTS = 99;
1924
1925        public const string LOCALIZED_TITLE = "TITLE_KEY";
1926
1927        public enum StatusEnum
1928        {
1929            Undefined,
1930            Ok,
1931            Error,
1932            Interrupted
1933        }
1934
1935        public DummyClass DummyObject { get; private set; }
1936
1937        public DummyClass OldDummyObject { get; private set; }
1938
1939        public GameObject DummyGameObject { get; private set; }
1940
1941        public StatusEnum Status { get; private set; }
1942
1943        public int dummyInt;
1944
1945        public string dummyString;
1946
1947        public bool DoOperations(DummyClass _dummyObject)
1948        {
1949            return true;
1950        }
1951
1952        public int GetRandomInt()
1953        {
1954            return 0;
1955        }
1956
1957        public int CalculateScore()
1958        {
1959            return 0;
1960        }
1961
1962        public bool IsLetter(char _char)
1963        {
1964            return true;
1965        }
1966
1967        public bool IsNumber(char _char)
1968        {
```

```
1969            return true;
1970        }
1971        public bool IsPunctiation(char _char)
1972        {
1973            return true;
1974        }
1975
1976        public string GetLocalizedText(string _textKey)
1977        {
1978            return "";
1979        }
1980
1981        public string GetGameModeFromServer()
1982        {
1983            return "";
1984        }
1985
1986        public class DummyClass
1987        {
1988            public string ID { get; private set; }
1989        }
1990
1991        public class MethodExamples
1992        {
1993            // Vague method name
1994            public void ComputeScore()
1995            {
1996
1997            }
1998
1999            // Good method name
2000            public void ComputeGameOverScore()
2001            {
2002
2003            }
2004
2005            // Good method name
2006            public void ComputeGameOverScoreAndUploadToServer()
2007            {
2008
2009            }
2010
2011            // method name without verb
2012            public void Score()
2013            {
2014
2015            }
2016
2017            public void GetScore()
2018            {
2019
2020            }
2021
2022            // Bad examples
2023            public string GetId()
2024            {
2025                return "";
2026            }
2027
2028            public string Id()
2029            {
2030                return "";
2031            }
2032
```

```
2033                public string ID { get; private set; }
2034
2035                // Parameters order
2036
2037                public void ConfigurePlayerSkin(bool _premiumSkins, ref
                      GameObject _instantiatedPlayer, out GameObject _mainWeapon, out
                       bool _error)
2038                {
2039                    _error = false;
2040                    _mainWeapon = new GameObject();
2041                }
2042
2043                // Examples of methods to pass variables to maintain interface
                      abstraction
2044                public void MethodWithGoodParameters(int _intParameter, string
                      _stringParameter)
2045                {
2046
2047                }
2048
2049                public void MethodWithBadParameters(DummyClass _dummyObject)
2050                {
2051
2052                }
2053            }
2054
2055        }
2056        #endregion
2057    }
```