## 命名規則

#### １、メソッド、イベント、クラス名とpublic変数はPascal記法にすること

|  |
| --- |
| public class Player // クラス名は名詞にする。数字と\_を使わないこと {  public bool IsDead; // boolはis,can,has,shouldを先頭にする  public static UnityEvent OnHpZero; // delegate,eventはstaticにし、名前はOnを先頭に、発動条件を表す。  public void SayHello(string name) // 関数名は動詞+名詞の形にする  {  } } |
|  |

#### ２、メンバー変数、ローカル変数と関数引数はCamel記法にすること

|  |
| --- |
| public class Player {  private bool isDead; // private を書き忘れないで  public void AddHp(int addHpNum)  {  int originHp = 0;  } } |

#### ３、ハンガリアン記法をしないこと。ローカル変数はthisを使用すること

|  |
| --- |
| public class Player {  private bool isDead;  public void AttackEnemy(Enemy target)  {  if (target.isDead && !this.isDead) Debug.log("Win");  } } |

#### ４、定数は全部大文字にしないこと

|  |
| --- |
| // Correct public static const string PlayerType = "Homo"; private const string taste = "RedTea";  // Avoid public static const string PLAYERTYPE = "Homo"; |

#### ５、略語できるだけ使わないこと

## コーディング規則

### １、レイアウト規則

#### 1.1、クラス内、宣言の順番は静的変数宣言、public変数、[SerializeField] private変数、private変数

|  |
| --- |
| public class Player {  public static string OfficialName;  public string nickName;  [SerializeField] private bool isHomo;  private int hp;  } |

#### 1.2、#region…#endregionでコードを整理すること

|  |
| --- |
| public sealed class SingletonClass  {  #region singleton pattern  private static SingletonClass Instance;  private SingletonClass() { }  public static SingletonClass GetInstance()  {  if (SingletonClass.Instance == null)  {  SingletonClass.Instance = new SingletonClass();  }  return SingletonClass.Instance;  }  #endregion  } |

#### 1.3、名前が長くて一行に収まらない場合、.(ピリオド)の後で改行を行うこと

|  |
| --- |
| var currentPerformanceCounterCategory = new System.  Diagnostics.PerformanceCounterCategory(); |

#### 1.4、引数が多く演算子が多く、1行が長くなる場合は((括弧)や、(カンマ)の後で改行すること。引数の数が多くなる場合はBuilderパターンの適用も検討しましょう。

|  |
| --- |
| var ipAddr = String.Format(  "{0}.{1}.{2}.{3}",  0x7F, 0x00, 0x00, 0x01);  bool isEmpty = ((data == null)  || (data == DBNull.Value)  || (String.IsNullOrEmpty(data.ToString())));  if (isEmpty) Debug.log("It's Empty"); |

#### 1.5、できるだけUpdate関数内そのままコードを書き込まないで。関数化してください

|  |
| --- |
| // Not good private void Update() {  // ...  // ...Do something  // ... }  // better private void Update() {  HandleInput(); }  private void HandleInput() {  // ...  // ...Do something  // ... } |

### ２、コメント規則

#### 2.1、コード行の末尾ではなく別の行に記述する。先頭大文字、できるだけ英語で、コメント記号(//)と文字の間には空白をひとつ挿入する。名前見れば使い道がわかるものは、コメント要らない

|  |
| --- |
| // Archive player's Hp, notice this value can not be NEGATIVE public uint8 PlayerDeafultHp; |

## イディオム

#### １、ifの中括弧{ }の省略は、一つの式（一つの処理）しかない場合のみできること

|  |
| --- |
| // Good if (isEnemyDead) OnKilledEnemy.Invoke(); else Debug.log("Enemy still Alive");  // Bad if (isEnemyDead) score++, playerHp++;  // Super bad  if (isEnemyDead) score++; playerHp++; |

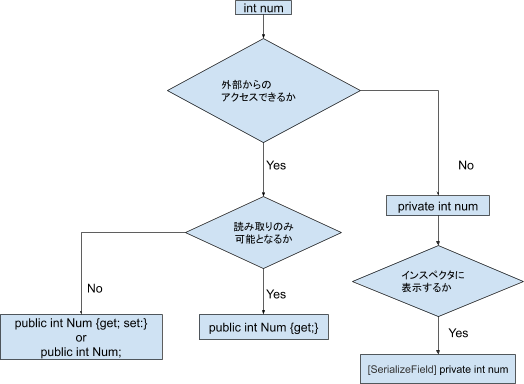
#### ２、ガード節は積極的に使うこと。メソッドの早い段階でのreturnは積極的に行いネストを減らすこと。特にif文の中にまたif文が入っている場合、returnの活用を注意する

|  |
| --- |
| // Bad public void AttackTarget(Enemy target) {  if (this.hp != 0)  {  if (target.Hp != 0)  {  OnAttackedEnemy.Invoke();  target.UnderAttack(this.atkPoint);  }  } }  // Good public void AttackTarget(Enemy target) {  if (this.hp == 0 || target.Hp == 0) return;   OnAttackedEnemy.Invoke();  target.UnderAttack(this.atkPoint); } |

#### ３、遅延処理はInvokeで気楽にしよう。タイマーとコルーチンは面倒くさい

|  |
| --- |
| if (other.tag == "Homo") Invoke("Shout", 3.5f);  private void Shout() {  Debug.log("No More Redtea PLZ"); } |

#### ４、アクセス修飾子について



{get; set;}についてわからないなら、

[【C#】プロパティを活用しよう！使う理由と実装方法を解説](https://www.sejuku.net/blog/102827)これを読んでね

#### ５、初期化処理はAwake関数内で、初期化の同時に外部から情報の読み取りはStart関数内で

|  |
| --- |
| private int num; private bool isTargetDead; [SerializeField] private GameObject target;  private void Awake() {  num = 10; }  private void Start() {  this.isTargetDead = target.isDead; } |

#### ６、他人のために、使えそうなeventを定義してといく。必ずどこかでコールバックするメソッドを解除すること（OnDestroy推奨）

|  |
| --- |
| // Player script using UnityEngine.Event;  private int playerHp; public static UnityEvent OnPlayerDead;  /...どこかの処理 if (playerHp == 0) { 　　　// ↓　if (OnPlayerDead != null) OnPlayerDead.Invoke() の簡略記法 　　　OnPlayerDead?.Invoke(); } |

|  |
| --- |
| // Enemy script private void Start() {  OnPlayerDead += Shout; }  private void Shout() {  Debug.log("やった！"); }  private void OnDestroy() {  OnPlayerDead -= Shout; // 忘れずに、削除して下さい！ } |

このようにしたら、Enemy scriptの担当者は、Player scriptを編集ずに済むのだ

#### ７、変数宣言する時、もし後ろから変数の形判定できれば、varを使って下さい

|  |
| --- |
| **// Not necessary Dictionary<string, List<StringBuilder>> stringToBuilderMap = new Dictionary<string, List<StringBuilder>>(); UnityEngine.AI.NavMeshAgent agent = GetComponent<UnityEngine.AI.NavMeshAgent>();  // better var stringToBuilderMap = new Dictionary<string, List<StringBuilder>>(); var agent = GetComponent<UnityEngine.AI.NavMeshAgent>();** |

#### ８、タスクコメントを使ってください！

<http://baba-s.hatenablog.com/entry/2017/12/05/175600>