クラスの継承・多様性

オブジェクト指向の特徴3つ

オブジェクト指向言語（C#）が持っている3つの特徴

・継承

・カプセル化

・多態性(多様性・ポリーモフィズム)

継承

継承（インヘリタンス）

使い方　→　クラス名：継承元

```
public class Sample1 : MonoBehaviour
{
```

継承：クラスのプロパティやメソッドなどを引き継いだクラスを作ること。

　　継承先でも継承元と同じ機能が使える。

　　コードの重複を減らして、再利用性を高める。

　　共通する機能を使いまわしたい時、既存のクラスを利用する。

継承元の呼び方：スーパクラス・基底クラス・ベースクラス・親クラス

継承先の呼び方：サブクラス・派生クラス・子クラス

継承

```
class Parent
{
    public int a = 123;
    public void ShowParent()
    {
        Debug.Log("ShowParent");
    }
}
```

- スーパクラス
- 基底クラス
- ベースクラス
- 親クラス

```
class Child : Parent
{
    public int b = 456;
    public void ShowChild(){Debug.Log("ShowChild");}
}
```

- サブクラス
- 派生クラス
- 子クラス

```
void Start()
{
    Child child = new Child();
    Debug.Log(child.a);
    child.ShowParent();
    Debug.Log(child.b);
    child.ShowChild();
}
```

オーバーライド（再定義）

```
class Parent
{
    public int a = 123;
    public virtual void ShowParent()
    {
        Debug.Log("ShowParent");
    }
}
class Child : Parent
{

    public override void ShowParent() { Debug.Log("ShowChild"); }
}
void Start()
{
    Child child = new Child();
    Debug.Log(child.a);
    child.ShowParent();
}
```

```
// 123
// ShowChild
```

コンストラクタ

```
class Parent
{

    public Parent() { Debug.Log("ShowParent"); }
}
class Child : Parent
{

    public Child() { Debug.Log("ShowChild"); }
}
void Start()
{
    Child child = new Child();
}
```

親クラスのコンストラクタが、

子クラスのコンストラクタより

先に実行される

引数つきコンストラクタ

```
class Parent
{
    public int a;
    public Parent(int a) { Debug.Log("ShowParent " + a); }
}
class Child : Parent
{
    public Child(int a) : base(a)
    { Debug.Log("ShowChild" + a); }
}
void Start()
{
    Child child = new Child(2);
}
```

```
// ShowParent 2
// ShowChild 2
```

引数つきコンストラクタ

```
class Parent
{
    public int a;
    public Parent(int a) { Debug.Log("ShowParent " + a); }
}
class Child : Parent
{
    public Child(int a) : base(a / 2)
    { Debug.Log("ShowChild " + a); }
}
void Start()
{
    Child child = new Child(2);
    // ShowParent 1
    // ShowChild  2
}
```

多態性(多様性・ポリモーフィズム)

多態性（多様性・ポリモーフィズム）

オーバライド：再定義

親クラスで定義したメソッドを子クラスで再定義すること

オーバーロード：多重定義

引数の型もしくは、引数の数さえ違えば、

同じクラス内に同じ名前のメソッドを定義できる

オーバーライド（再定義）

```
abstract class Animal
{
    public abstract string GetCall();//実装は継承先に任せる
    // public virtual string GetCall(){return "";}
}
class Dog : Animal{
    public override string GetCall() { return "ワン"; }
}
class Cat : Animal{
    public override string GetCall() { return "にゃー"; }
}
void Start()
{
    Dog dog = new Dog();
    Cat cat = new Cat();
    Animal animal;
    if (Random.Range(0, 2) == 0){ animal = dog; }
    else{ animal = cat; }
    Debug.Log(animal.GetCall());
}
```

必ず継承して使うクラスの時は
「abstract」 を使う

中身のメソッドなどは

「abstract」もしくは「virtual」どちらでも使用可能

オーバライドしても元のメソッドが使える

```
abstract class Animal//abstractインスタンスを作成できないクラス
{
    public virtual string GetCall()
    {
        Debug.Log("baseクラスのログ!!!!");
        return "baseクラスのログ";
    }
}
class Cat : Animal
{
    public override string GetCall()
    {
        base.GetCall();
        Debug.Log("にゃー！！");
        return "にゃー";
    }
}
```

baseがうまくいっているかの確認

```
void Start()
{
    Cat cat = new Cat();
    // cat.GetCall();//コンソール表示//"baseクラスのログ!!!!" //"にゃー！！" //返り値に"にゃー"
    Animal animal;
    animal = cat;
    animal.GetCall(); ;//コンソール表示//"baseクラスのログ!!!!"//"にゃー！！" //返り値に"にゃー"
    // Debug.Log(cat.GetType());//inheri_base + Cat
    // Debug.Log(animal.GetType());//inheri_base + Cat
}
```

オーバーロード(多重定義)

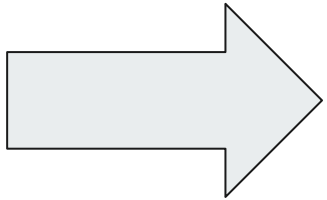
```
public class Mathf
{
    public int Max(int a, int b)
    {
        Debug.Log("int");
        if (a > b) return a;
        else return b;
    }
    public float Max(float a, float b)
    {
        Debug.Log("float");
        if (a > b) return a;
        else return b;
    }
}
void Start()
{
    Mathf mathf = new Mathf();
    Debug.Log(mathf.Max(1, 2));
    Debug.Log(mathf.Max(1.3f, 2.4f));
}
```

intが引数に指定された場合のメソッド

floatが引数に指定された場合のメソッド

同じメソッド名でも「引数の型」もしくは「引数の数」のどちらかが違えば、いくつでもメソッドを定義でき、どのメソッドを使うかは型や引数の数から自動で判断してくれる

クラスのカプセル化（隠蔽）の例え



PCの配線が剥き出しでも動く

でも、ちゃんとカバーを被せて、猫とか親とか子供とかに配線メチャクチャにされたり、壊されたりするのを防ぐ

カバーは、パーツ交換したい時に外せばいい

クラスのカプセル化（隠蔽）の例

作ったクラスにカバーをかける

うっかりプロパティ（メンバー変数）を触って予期せぬ動作をしないように、クラス内でしか使わないメソッドは隠す

そうすればトラブルを防げるんじゃないかという考え方

C#でのカプセル化（隠蔽）

アクセス修飾子を使う

public　　：どこからでも使える
protected　：自身のクラスと継承先でのみ使える
private　　：クラス内で使える

クラス内において基本はprivateだけ使えばいい、
そもそもクラス内であれば何も書かなければprivateとして扱われる

クラスのカプセル化

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class NewBehaviourScript : MonoBehaviour
6  {
7      // Start is called before the first frame update
8      void Start()
9      {
10
11     }
12
13     // Update is called once per frame
14     void Update()
15     {
16
17     }
18 }
```

C#スクリプトを作成した時の
初期画面でも

```
private  void  Start(){

}

private  void  Update(){

}
```

になってる

アクセス修飾子の使い道の考え方例

①クラス内でプロパティ（メンバ変数）やメソッドを使う時、privateで書けないか

②継承先でもアクセスできる必要があるな　→　protected

③他のクラスでもアクセスしたいな　→　public

とはいえ、**基本は「全て」private**

それでもprivate以外を使いたい時に用意されてるものが　→　get・set

publicを使う時の例(プロパティ)

どうしてもpublicを使いたい　→　プロパティ

フィールド(それぞれのインスタンスが持っている変数)をインスタンスの外部から見えないようにしつつ、プロパティ(get・set)を使用し、間接的にフィールドの値を取得したり、変更したりする。

プロパティの基本的な使い方

```
class   Samp{
  型名 d;//バッキングフィールド (プロパティは値を保存できないのでここで保存)
    public   型名   プロパティ名{  //プロパティ名とデータの宣言
        get {   //プロパティから値を取り出す
            return d;
              }
        set{   //プロパティにて値を設定
           d = value;   //設定しようとしている値を「value」に設定できる
              }
    }
}
```

プロパティの基本的な使い方

```
void Start()
{
    Samp samp = new Samp(); //Sampクラスからインスタンス生成
    変数 = samp.プロパティ名; //getが実行される
    samp.プロパティ名 = 値;     //値はvalueとして扱われ、setが実行される

    samp.プロパティ名 = samp.プロパティ名 + 値
    //getが実行され,d + 値が実行されsetが実行される
}
```

例

```
using UnityEngine;
public class Character
{
  private int hp = 300;
  private int maxHp = 300;
  public int HP{
    get{ return hp; }//「HP」だと無限ループするので、わかりやすくするために _hpなどとする
    set{
      hp = value;
      if (hp > maxHp){ hp = maxHp; }//回復なので最大HPを超えるなんてことはありえない
      if (hp <= 0)//hpが0以下になったら「死亡」の処理を動かしたい。hpがマイナスもおかしい
      { hp = 0;
        Dead() ; }
    }
  }
  private void Dead(){ Debug.Log("DEAD"); }
}
```

例

```
public class Example_Property : MonoBehaviour
{
    void Start()
    {
        Character chara = new Character();
        chara.HP -= 100;
        Debug.Log($"HP:{chara.HP}");
        chara.HP += 200;
        Debug.Log($"HP:{chara.HP}");
        chara.HP -= 400;
        Debug.Log($"HP:{chara.HP}");
    }
}
```

なるべくpublicを使わない

UnityでInspectorで値を設定したいとき、

publicにすれば外から設定するUIを作れる（シリアライズ化できる）

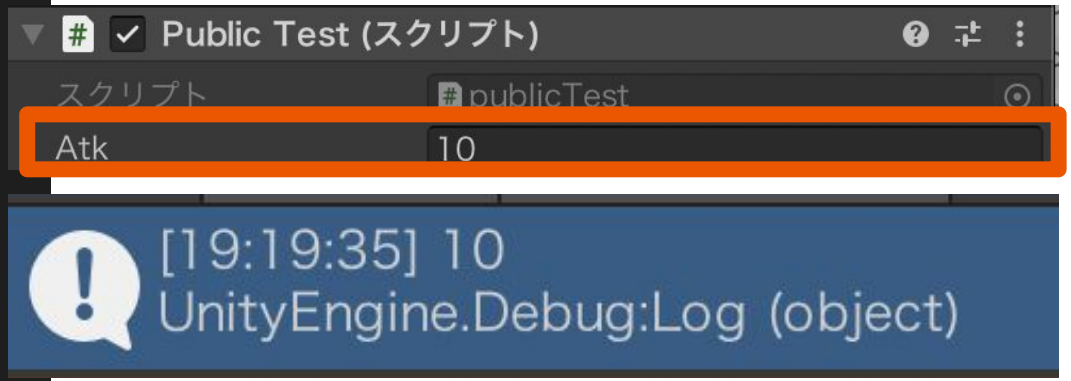
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class publicTest : MonoBehaviour
{
    public int atk;
    void Start()
    {
    }
    void Update()
    {
    }
}
```

なるべくpublicを使わない

public int atk; のようにpublicはなるべく使用しない
しかし、private int atk;だと表示されない。
なので [SerializeField] private int atk;
とすればprivateのままInspectorからの操作ができる

```
public class publicTest : MonoBehaviour
{
    [SerializeField] private int atk;
    void Start()
    {
        Debug.Log(atk);
    }
    void Update()
    {
    }
}
```



なるべくpublicを使わない

他のクラスから取得だけはしたい時、

```
private int atk;   //フィールド・インスタンス変数・メンバ変数
public int Atk => atk;//プロパティから値だけ取得
```

```
public int Atk { get => atk; }
```

```
public int Atk {
        get{
            return atk;
        }
    }
```

とすれば、

「atk」という変数設定はできない(「set」できない)

しかし、「atk」という変数を取得できるようになる(「get」できる

public int IntValue {get; private set;} //自動プロパティ

イメージ　↓↑　簡略化できる

```
private int _IntValue; //フィールドを一つ用意
public int IntValue //プロパティ経由
{
  get =>_IntValue
  private set{_IntValue =value;}
}
```