



# スクリプトを書こう



Tips

## 完成データのダウンロード

この章で作成するプロジェクトの完成データは、以下のアドレスからダウンロードできます。

- <https://www.shoeisha.co.jp/book/download/3603/read>

### 3.1

## スクリプトでゲームオブジェクトを操作しよう



ここからは、スクリプトを書いてキャラクターを操作してみましょう。

Unityではプログラムのことをスクリプトといい、スクリプトを書くためにC#というプログラミング言語を使用します。

既存のコンポーネントはインスペクタービューの [Add Component] ボタンで追加できましたが、これから追加するスクリプトは自分で作る

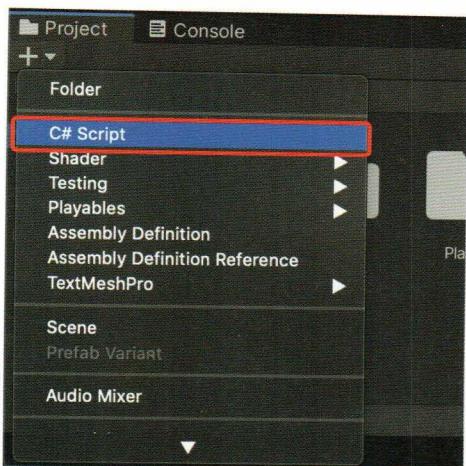
ので、まだどこにも存在していません。ですから、まずスクリプトから作っていきましょう。

そして作ったスクリプトもコンポーネントの一つとして、ゲームオブジェクトにアタッチしていきます。

## スクリプトファイル (PlayerController) を作ろう

それでは、プレイヤーキャラクターを動かす機能を持つスクリプトを作りましょう。

プロジェクトビューの左のリストから Player フォルダーを選択してください。この中に新しくスクリプトを作ります。



プロジェクトビューの左上にある、[+]ボタンをクリックしてください。プルダウンメニューが開きます。ここから「C# Script」を選択します。



プロジェクトビューに「NewBehaviourScript」というC#のスクリプトファイルができます。ファイル名が編集状態になっているので、名前を「PlayerController」に変更してください。これがこのスクリプトのファイル名になります。

スクリプトには、この最初の時点で適切な名前を付けるようになります。あとからスクリプトの名前を変えてしまうと、ファイル名と内部の名前が食い違って少し面倒なことになります。

今回はプレイヤーを操作するためのスクリプトなので、「PlayerController」としました。このようにスクリプトはその機能に合った名前を付けるようにしましょう。



**接頭語と接尾語**

スクリプト名は機能がわかるようするために、「接頭語」と「接尾語」というものが使われことがあります。例えばゲームで操作できるプレイヤーキャラクターには頭に「Player」という単語を付けておく、さらに場合によっては省略して「PLY」や「PL」のように3文字や2文字に書いたりします。名前の頭を見れば何のデータかわかるというわけですね。これが接頭語です。

接尾語は名前の後ろに付ける決まった単語ですね。何かを操作するから「Controller」と付ける、何かを管理するから「Manager」と付けるなどです。

「接頭語」「接尾語」に決まったルールがあるわけではありません。適当な名前を付けてしまうとあとで自分でもわけがわからなくなってしまいます。自分で命名規則を決めておけば、あとから見てもわかりやすくなります。

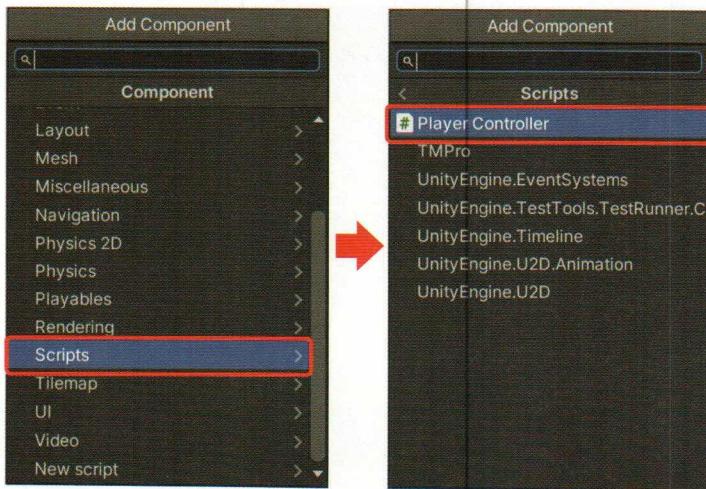
## スクリプトをアタッチする

スクリプトもコンポーネントの一つです。ですから先ほどゲームオブジェクトに Rigidbody 2D や Collider 2D を付けたのと同じように、ゲームオブジェクトにアタッチして使います。

ゲームオブジェクトにスクリプトをアタッチする方法は主に2つあります。

### ◆ Add Component ボタンからアタッチする

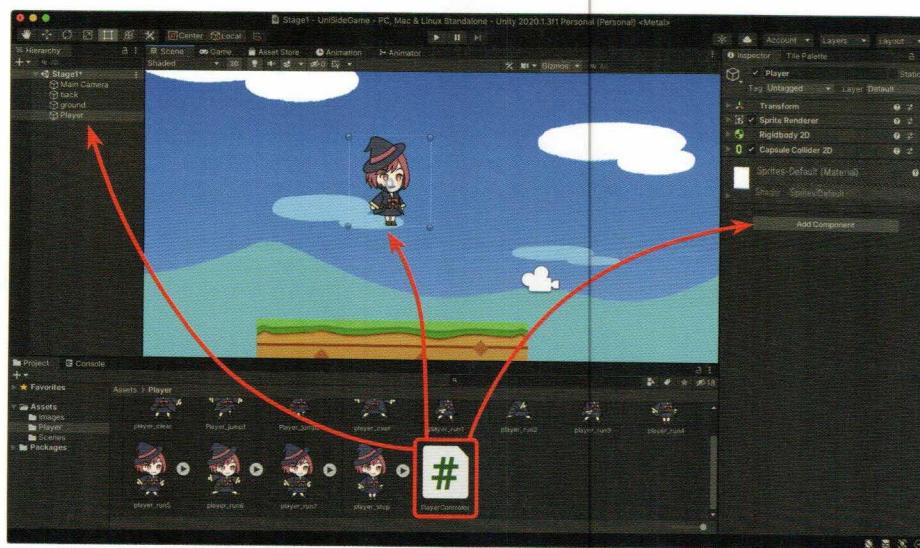
スクリプトをアタッチしたいゲームオブジェクトを選択して、インスペクタービューの [Add Component] ボタンを押してから、「Scripts」を選択してください。「Scripts」はメニューを少しスクロールした下のほうにあります。



すると、作ったスクリプトファイルが見えるはずです。左端にアイコンが付いているのが自分で作ったスクリプトですね。それを選択するとアタッチできます。

## ◆ ドラッグ&ドロップでアタッチする

プロジェクトビューにある、スクリプトアイコンをヒエラルキービューやシーンビューのゲームオブジェクトにドラッグ&ドロップするとアタッチできます。

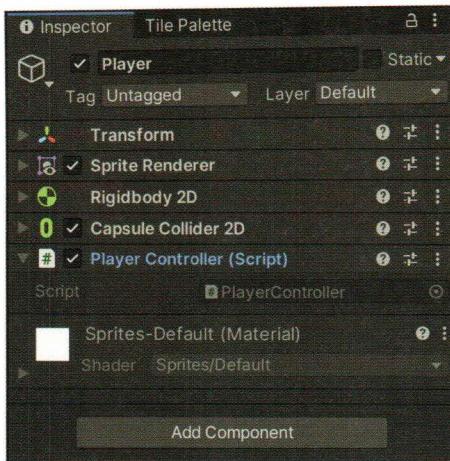


またはゲームオブジェクトを選択した状態で、インスペクタービューの各コンポーネントの隙間にドラッグ&ドロップすることでもアタッチできます。

**Tips**

**目的のゲームオブジェクトに確実にアタッチするには**

ゲームオブジェクトが増えたり重なり合ったりしてくると、意図しないゲームオブジェクトにアタッチしてしまうことがあります。そういった状況の場合はヒエラルキービューかインスペクタービューへのドラッグ&ドロップ、または [Add Component] ボタンからアタッチするのが確実でしょう。



スクリプトがアタッチされたゲームオブジェクトを選択すると、インスペクタービューにスクリプト情報が表示されます。

ところで「Player Controller」と単語の間にスペースが入っているのに気がついたでしょうか。これはUnityのエディターが読みやすくするために自動的に入れてくれているのです。単語の頭を大文字で書き、それ以降を小文字で書くと、このようにUnityが自動的にスペースを入れてくれます。

## スクリプトの中身を見てみよう

それでは、作ったスクリプトファイルの中身を見てみましょう。プロジェクトビューにある、「PlayerController」のC#アイコンをダブルクリックしてください。スクリプトを編集するエディターである、[ビジュアル スタジオ](#)が起動してC#のプログラムコードが表示されます。

新しく作られたスクリプトには以下のような18行のプログラムコードが書かれています。しかし中身はまだ空っぽです。

```

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

public class PlayerController : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

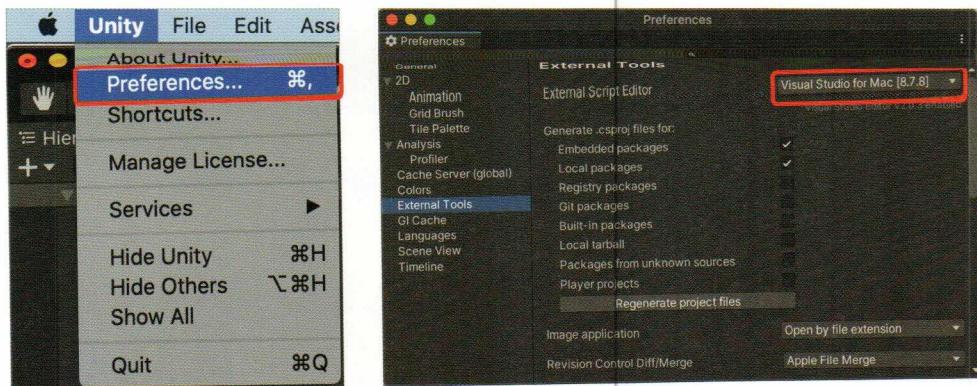
    // Update is called once per frame
    void Update()
    {

    }
}

```

## External Tools を設定しよう

もし、ダブルクリックでVisual Studioが起動しなかったり、スクリプトファイルが開かなかったりした場合は、メニューから〔Preferences…〕を選択して、「Preferences」ウィンドウを開いてください。



その中の「External Tools」タブにある、「External Script Editor」に「Visual Studio」が設定されているかを確認してください。ここはスクリプトを編集するエディターの設定です。新しいバージョンのUnityをインストールした場合など、設定が外れてしまうことがあるため、そのような場合はここで確認しましょう。

## ゲームオブジェクトをキー入力で操作するスクリプトを書こう



PlayerControllerにスクリプトを書いていきましょう。これから作るのはサイドビューのゲームキャラクターなので、まずはパソコンの左右矢印キーを押すことでキャラクターを左右に移動できるようにしてみましょう。スクリプトが、ゲームキャラクターを動かす指示書になります。

以下が更新したスクリプトです。更新部分はハイライトしています。「// Rigidbody2D型の変数」などの「//」から始まって改行までの箇所は書き込まなくても大丈夫です。これが何なのかは後ほど説明します。

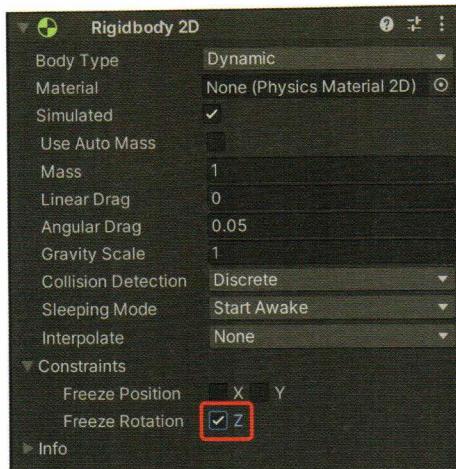
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    Rigidbody2D rbody; // Rigidbody2D型の変数
    float axisH = 0.0f; // 入力

    // Start is called before the first frame update
    void Start()
    {
        // Rigidbody2D を取ってくる
        rbody = this.GetComponent<Rigidbody2D>();
    }

    // Update is called once per frame
    void Update()
    {
        // 水平方向の入力をチェックする
        axisH = Input.GetAxisRaw("Horizontal");
    }

    void FixedUpdate()
    {
        // 速度を更新する
        rbody.velocity = new Vector2(axisH * 3.0f, rbody.velocity.y);
    }
}
```



ここまでできたら、キャラクターにアタッチしている Rigidbody 2D の「Freeze Rotation Z」のチェックボックスをオンにしておいてください。

ゲームオブジェクトの左右に物理的な力が加わった場合、Collider（当たり）として底辺が丸い Capsule Collider 2D を使っているので転がってしまうのですが、このチェックボックスをオンにすることでそれが防止されます。

この状態で一度スクリプトを保存（[ファイル] メニューから [保存] を選択）して、Unity ウィンドウに戻ってゲームを実行してみましょう。

## ゲームを実行しよう



ステータスバーの実行ボタンを押して、ゲームを実行してください。

キーボードの左右キーを押してみてください。キーボードの右キーを押すと右方向に進み、左キーを押すと左方向に移動するようになりました。


**Tips**

**正しく動かない場合は**

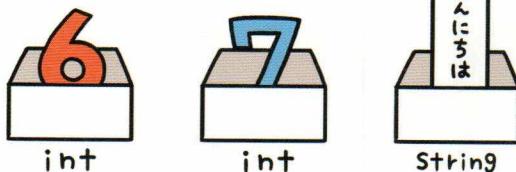
正しく動かない場合は、もう一度打ち込んだスクリプトを確認してみましょう。大文字や小文字の入力間違い、行の最後にセミコロンの抜け、不必要なスペースなどを確認してみましょう。



## 3.2 C# プログラムの基礎を覚えよう

先ほど書いたスクリプトを説明する前に、理解しておいてほしいC# プログラムの基礎を少しだけ解説しておきます。

### 型と変数を知ろう



**変数**は数字や文字などの値を入れることのできる「箱」のようなもので  
す。箱の中身、つまり変数の中身は  
自由に入れ替えることができます。

変数は、プログラムでの計算や判  
断に使う最も基礎的なものです。実  
際に変数の書き方を見てみましょう。

```
int num1;  
int num2;  
num1 = 3;  
num2 = 2;  
int num3 = num1 + num2;
```

上記のプログラムコードの1行目、**int** というのが**型名**、**num1** というのが**変数名**です。変数  
名は自由に決めることができますが、変数を表す肩書きといえる**型**は決まっています。

それから、「=」(イコール) の記号です。算数や数学では「右の値と左の値は同じ」という  
意味で使われますが、ほとんどのプログラム言語では「右の値を左に入れる」という意味で  
使います。結果的には右と左は同じ値になるわけですが、意味合い的には少し違うというこ  
とを覚えておいてください。また行の最後には必ず「;」(セミコロン) が必要です。

ここではひとまず、変数を用意するためには、

```
int num1;  
(型名) (変数名)
```

というように書く、ということを覚えておいてください。



Tips

## 変数の命名ルール

「変数名は自由に決めることができる」と書いていますが、実は少しだけ制限があります。変数名の先頭に数字を使うことはできません。また「＃」「#」「\$」「%」「&」「\*」「<」「>」「:」「;」「〔〕」「〔〕」「+」「-」などのような記号文字も使うことができません。変数名に使える記号は、「\_」(アンダースコア)だけです。

そしてプログラム初心者がハマるポイントとして、アルファベットの大文字と小文字は別の文字として区別される、ということが挙げられます。つまり、num1 (Nが小文字) と Num1 (Nが大文字) は別の名前として扱われるということです。

1

2

3

スクリプトを書こう

### ◆ 主な型

主な型には以下のようなものがあります。

- **int (整数型)**: 整数を表す型名です。整数とは1、2、3、4のようなものの数として数えられる数字のことです
- **float (小数点型)**: 3.14 や 1.4142 のように、小数点が付く数字を表す型名です。プログラムの中で書くときは 3.14f のように最後に「f」を書き足します
- **string (文字列型)**: 文字を表す型名です。C#での文字は" " (ダブルクオーテーション) ではさんで表現します。例えば「こんにちは」をプログラムで文字列にするならば、" こんにちは " と書きます
- **bool (論理値型)**: true (はい) と false (いいえ) で表される型名です。プログラムで分岐を判断する条件判断などで使われます

## 演算子で計算をしよう

プログラムの中では計算を行います。そこでは「足し算」「引き算」「掛け算」「割り算」などの計算を行います。足し算をするときはキーボードの「+」、引き算をするときは「-」を使うということはわかりますが、掛け算や割り算の記号はキーボードにはありませんね。掛け算は「\* (アスタリスク)」、割り算は「/ (スラッシュ)」を計算記号(演算子)として使います。

- 足し算:  $6 + 2$
- 引き算:  $6 - 2$
- 掛け算:  $6 * 2$
- 割り算:  $6 / 2$

実際の使い方は、このあとスクリプトを書きながら覚えていきましょう。

## プログラムにコメントを付けよう

先頭に「//」が書かれている行は「コメント」です。コメントはプログラムに影響を与えない、このプログラムを読む人への注意書きのようなものです。

例えば、64ページのプログラムの10行目には、

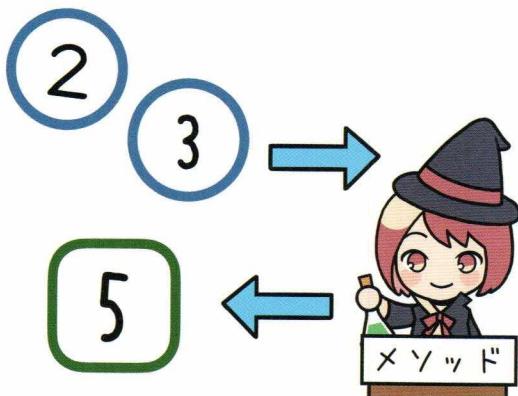
```
// Start is called before the first frame update
```

と書かれています。日本語に訳すと「Startは最初のフレーム更新の前に呼ばれる」という Start メソッドの説明が書かれています。

このように、コメントはメソッドやメソッド内のプログラムの説明に多用されます。コメントにはこのような英語だけでなく、日本語も書くことができます。プログラムの行数が少なければ、プログラムの内容も比較的わかりやすいのですが、行数が増えてプログラムが複雑になってくると、ぱっと見では何が書かれているのかわからなくなってくるものです。

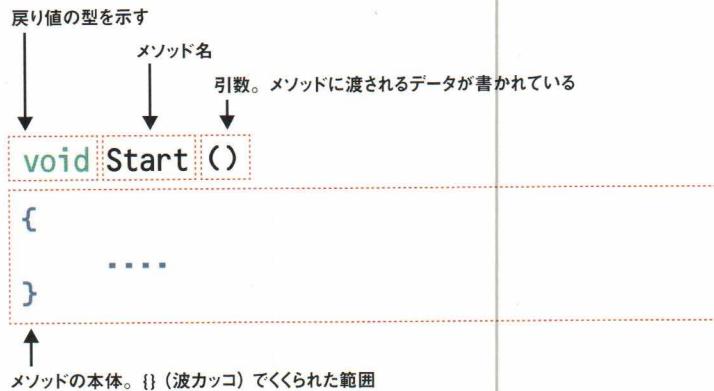
また自分が書いたプログラムであっても、時間がたてば書いた本人でも判別できないということは多々あります。ですから、他の人のため、未来の自分のためにコメントはできるだけ多く、わかりやすく書きましょう。

## メソッド（関数）で処理を行おう



メソッドとは「何かの値を受け取つて、作業（処理）を行い、結果を値として返す」という仕事をします。関数やファンクションなどとも呼ばれますですが、この本ではメソッドという呼び方で統一します。

メソッドにも決まった書き方があります。メソッドに必要なものは、「戻り値」「メソッド名」「引数」「本体」です。先ほど作ったPlayerControllerに書いてあったStartというメソッドを見てみます。



### ◆ 戻り値の型

メソッドが仕事を終えたあと、その結果を返すときの「型名」を指定します。

Start メソッドの戻り値を見ると void となっています。**void** というのは何も値を返さないという意味で、Start メソッドは結果として何も値を返しません。

なお、Start メソッドには後ほど紹介する引数もないで、丸カッコ「( )」の中は空っぽです。このように何も受け取らず、値も返さないメソッドもあります。

### ◆ メソッド名

変数に名前を付けたように、メソッドにも名前を付けます。ここはそのメソッドの名前を指定します。

### ◆ 引数

引数とはこのメソッドが仕事をするために受け取る値です。引数はメソッド名のあとに、「( )」丸カッコでくくられて書かれます。Start メソッドでは引数を受け取らないで、丸カッコの中は空ですが、引数がある場合は、以下のように書きます。

(型 値)

この書き方は変数の宣言と同じですね。引数が2つ以上ある場合は、

(型 値, 型 値)

というように「,」(カンマ) で区切ってつなげて書きます。引数は必要な数だけ書くことができます。

## ◆ ステートメント

引数のあと、「{}」(波カッコ)に囲まれている範囲がメソッドの範囲です。この波カッコに囲まれた範囲を「ステートメント」といいます。このステートメントがプログラム内での1つの区切りになります。この中に必要なプログラムを書いていきます。



**カッコの閉じ忘れに注意！**

スクリプトを書くとき、「{}」(波カッコ)の閉じ忘れに注意してください。「{」で始めたら必ず「}」で終わらなければいけません。同じく「()」(丸カッコ)も「(」で始めたら必ず「)」で終わらなければいけません。

## ◆ 戻り値

メソッドは、その結果を返すことができます。これを「戻り値」と呼びます。

例として、数字を2つ引数に受け取って、数字を1つ返すメソッドを書いてみましょう。以下のようになります。

```
int AddCal(int num1, int num2)
{
    return num1 + num2;
}
```

これは「2つの数字を足し算して、その答えを返す」という仕事をする `AddCal` という名前のメソッドです。`AddCal` というメソッド名は `addition` (足し算) と `calculator` (計算機) それぞれの頭3文字を取った略字です。引数で `num1` と `num2` という整数型の値を2つ受け取って、それを足し算して返しています。

`AddCal` メソッドの3行目にある、

```
return 値;
```

というのが「戻り値を返す」という記述です。戻り値が `void` になっている場合 `return` を書く必要はありませんが、途中で `return;` を書くことで、そこで処理を終了してメソッドを中断することができます。メソッドを使う場合は以下のように書きます。この場合、`answer` という `int` 型の変数に、5が入ることになります。

```
int answer = AddCal(2, 3);
```

ちなみに、メソッドを使うことを「メソッドを呼ぶ」などといいます。この言い方はこれから頻繁に使います。ぜひ覚えておいてください。

## クラスで変数やメソッドをまとめよう

**クラス**とは「特定の仕事を行う変数やメソッドの集まり」と考えてください。Unityのスクリプトは1ファイルごとに1つのクラスとして作られます。`PlayerController`を例にクラスの構成を見てみましょう。



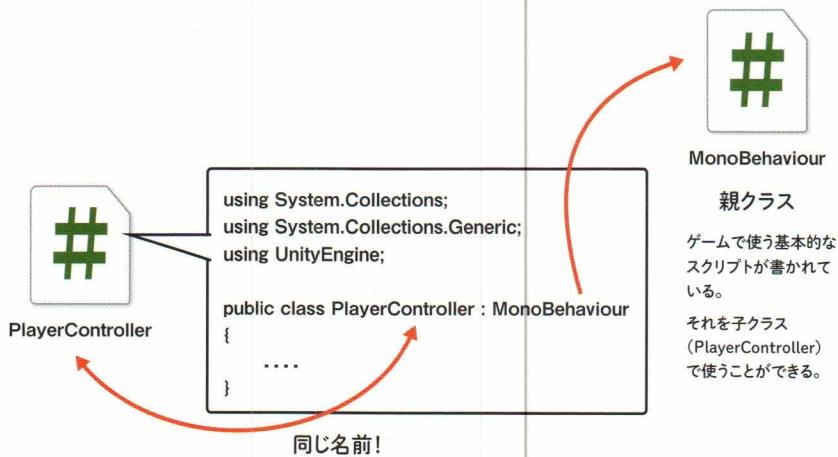
### ◆ `public`

行頭に `public` と書いてある場合は「このクラスをプロジェクトに新しく作成されたスクリプト全体で使えるようにする」という意味になります。`public` が付けられたクラス内の記述は全体に公開され、他のスクリプトからでも使うことができるようになります。ひとまず「Unityでプログラムを書くときは、必ずクラスの先頭に `public` を書く」と覚えておくとよいでしょう。

先ほど説明したメソッドや、クラス内に変数を書いた場合も、先頭に `public` と付けることで外部から参照できるようになります。Unityで `public` はかなり重要なキーワードです。`public` を付けたものはプロジェクト全体に公開して使えるようになるというのは、今後いろいろなところで出てくるので覚えておいてください。

### ◆ `class`

次の `class` という記述は、「これ以降がクラスである」という宣言です。クラス宣言から最後まで、「`{ } (波カッコ)`」に囲まれている範囲がクラスの範囲です。



`class`の次の、`PlayerController`というのはスクリプトを作るときに付けた名前ですね。同時にこのスクリプトのクラス名でもあります。クラス名とスクリプト名は同じになるということを覚えておいてください。

「`:`」(コロン) をはさんで次に書かれた `MonoBehaviour` とは、このクラスの「親クラス」の名前です。クラスには親子関係があり、親クラスで書いたスクリプトを子クラスでそのまま使うことができます。`MonoBehaviour` クラスには Unity でゲームを作るための基本的なスクリプトが書かれています。

### 3.3

## PlayerController スクリプトを見ていく

それでは、64ページで書いた `PlayerController` スクリプトを見てみましょう。このような構成になっています。

```

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

public class PlayerController : MonoBehaviour
{
    Rigidbody2D rbody;           // Rigidbody2D 型の変数
    float axisH = 0.0f;          // 入力
}

```

クラス

変数

```
// Start is called before the first frame update
void Start()
{
    //Rigidbody2D を取ってくる
    rbody = this.GetComponent<Rigidbody2D>();
}

// Update is called once per frame
void Update()
{
    // 水平方向の入力をチェックする
    axisH = Input.GetAxisRaw("Horizontal");
}

void FixedUpdate()
{
    // 速度を更新する
    rbody.velocity = new Vector2(axisH * 3.0f, rbody.velocity.y);
}
```

それぞれが何を意味するのか順番に説明していきましょう。

まず、先頭3行は「`using`」のあとに書いてあるプログラムの機能を使えるようにする」という宣言です。

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

これから書いていくプログラム以外にもすでにたくさんのプログラムが事前に用意されています。「`using ○○○○`」と、このスクリプトで使うということを最初に宣言しておくことで利用可能になります。この行をあとから書き換えることは、ほぼありません。

これ以降はクラスに関する記述です。順番に見ていきましょう。

## クラスに関する記述を見よう

### ◆ 変数定義

まず書かれている `Rigidbody2D` というのは、ゲームオブジェクトを物理法則に従って動かすようにするコンポーネントでしたね。`Rigidbody2D` というのは `Rigidbody2D` クラスとして定義されています。そして、クラスも型として扱います。ここでは、

```
Rigidbody2D rbody;  
(型名)      (変数名)
```

というように、宣言されていることになります。コンポーネントのほぼすべてはクラスとして作られていて、これと同じように型として使用することができます。同じように、float型のaxisH変数はUpdateメソッドの中で入力されたキーの値を保存しておくための変数です。

変数定義以降に書かれているものがメソッドです。PlayerControllerクラスには以下の2つのメソッドが最初から書かれていましたね。この2つはUnityのスクリプトには必ずある特殊なメソッドで、以下のような役割があります。

### ◆ Start メソッド

このクラスがシーンに読み込まれたときに1回だけ呼ばれるメソッドです。クラスのプログラムを動かす前に、何か準備が必要なときはこのStartメソッドで行います。

Startメソッドでは、先ほど定義したrbody変数にRigidbody2Dの値を入れています。これはプログラムの中でRigidbody2Dの持つ機能を使うために、あらかじめ変数にRigidbody2Dを入れて用意しておくためです。

ここでthis.という記述がありますが、これは自分自身、つまりPlayerControllerクラスを指します。「.」(ドット)のあとにはGetComponentメソッドが書かれています。これは「自分が持っているGetComponentメソッドを呼んでいる」とことを表します。C#ではこのように、何かの中にあるものを指し示すときには「.」を使います。しかし、自分(PlayerControllerスクリプト)の中にはどこにも、GetComponentメソッドはありません。これは3行目に書かれている、using UnityEngine;によって使えるようになっているのです。

なお、this.という記述は省略することができます。ここは説明のためにthis.を書きましたが、今後この本では必要ない限り、this.を省いた書き方をします。



### ゲームオブジェクトのコンポーネントを取得する GetComponentメソッド

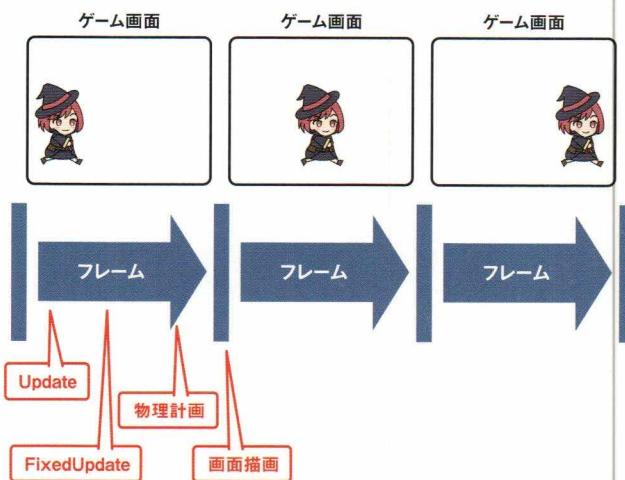
「=」(イコール)の右側にあるのはGetComponentメソッドの呼び出しです。GetComponentメソッドは山カッコでくくって<型名>と指定することで、その型のコンポーネントを取ってきてくれるメソッドです。ここでは「Rigidbody 2Dコンポーネントを取ってこい」と指示しているわけです。この結果、rbody変数にはRigidbody2Dコンポーネントが入ります。

```
GameObject obj = (ゲームオブジェクト).GetComponent<型名>();
```

ここの(ゲームオブジェクト)はゲームオブジェクトであれば何でもかまいません。そのゲームオブジェクトにアタッチされているコンポーネントを何でも取ってくことができます。ここでは「this.」と書いているので「自分自身から」取ってきているわけですが、他のゲームオブジェクトを指定することもできます。

`GetComponent`メソッドはかなり多用するメソッドです。ぜひ覚えておきましょう。

## ◆ Update メソッド



ゲームは、定期的に画面を少しずつ書き換えることにより、画面の映像を動いているように見せています。これが**フレーム**です。

Unityでは、1フレームに1回 `Update` メソッドが呼びられます。画面更新やゲームの操作に必要なことがあれば、この `Update` メソッドで行うこと、それが反映されます。

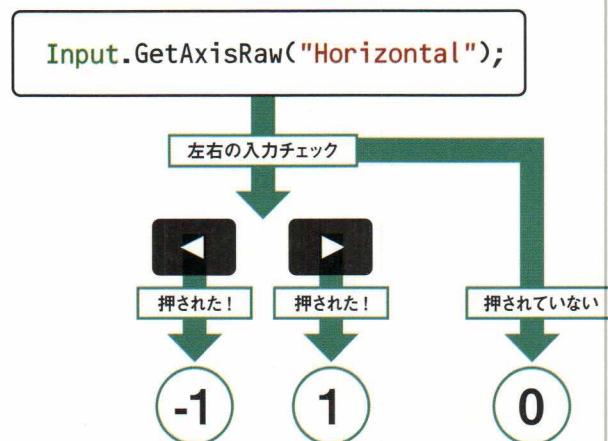
**Update メソッドは同じ間隔で呼ばれるわけではない**

1つ注意しておきたいのは、`Update` メソッドは必ずしも同じ間隔で呼ばれるわけではない、ということです。ゲーム中の処理によっては間隔がずれる可能性があります。このあとに説明する `FixedUpdate` メソッドが一定間隔で呼ばれるメソッドです。

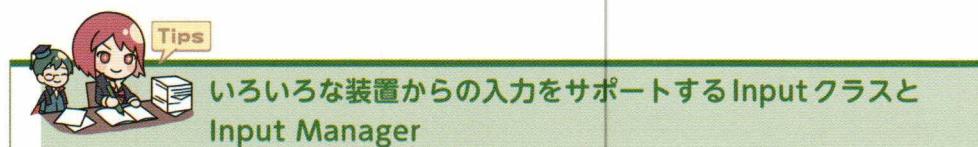
今はとにかく、「1フレームに1回呼ばれるメソッドがあり、そこでゲームに必要なことを処理する」ということを覚えておいてください。

`Update` メソッドに追加した1行では、キーボードの左右キーが押されたかどうかをチェックしています。`Input` というのはいろいろな入力系を管理するクラスで、`Input` クラスも「`using UnityEngine;`」によって使えるようになっています。

`Input` クラスが持つ `GetAxisRaw` メソッドを使うと、入力がチェックできます。ここでは引数に文字列で "Horizontal" と指定しています。Horizontalは英語で「水平」という意味ですが、これが「左と右」という指定を表しています。

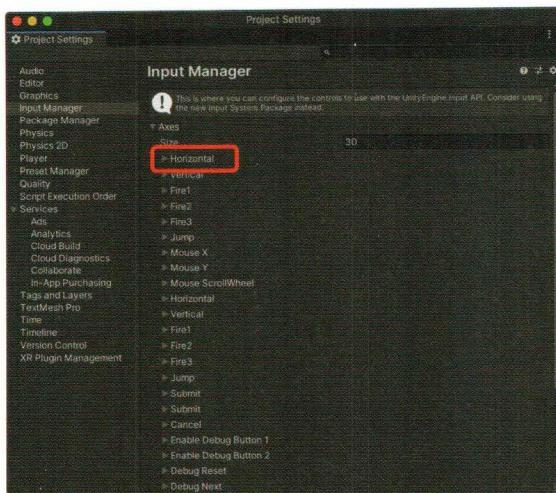


`GetAxisRaw` メソッドは右キーが押されていれば、`1.0f`を、左キーが押されていれば`-1.0f`を返します。何も押されていなければ`0.0f`が返されます。

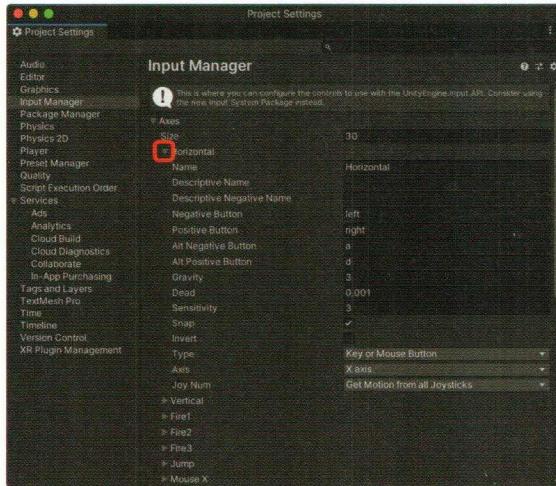


Unityではさまざまな機種向けのゲームが作れます。そして各ゲーム機にはいろいろな入力装置が付いています。パソコンならばマウスやキーボード、専用ゲーム機ならコントローラー、スマホならタッチパネルなどです。`Input` クラスは、それらさまざまな入力装置からの入力をまとめて扱ってくれる便利なクラスです。

例えばここで使っている `GetAxisRaw` メソッドですが、引数に "Horizontal" という文字列を指定することでキーボードの左右キーの他にも `[A]` キーを左、`[D]` キーを右というように対応させてくれます。



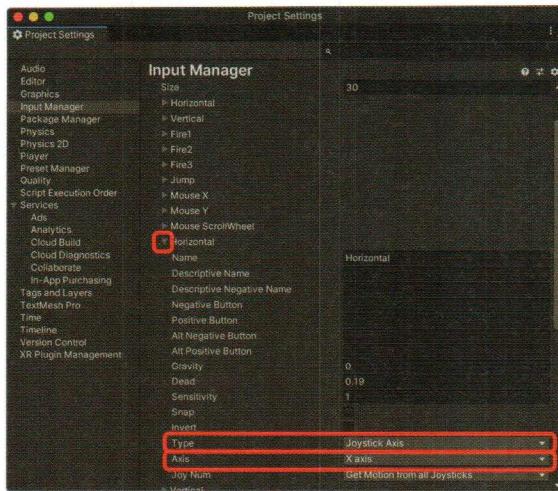
これらInputクラスのメソッドからの入力と対応キーはInput Managerというものが処理してくれています。[Edit]メニューから、[Project Settings…]を選択すると「Project Settings」ウィンドウが開きます。



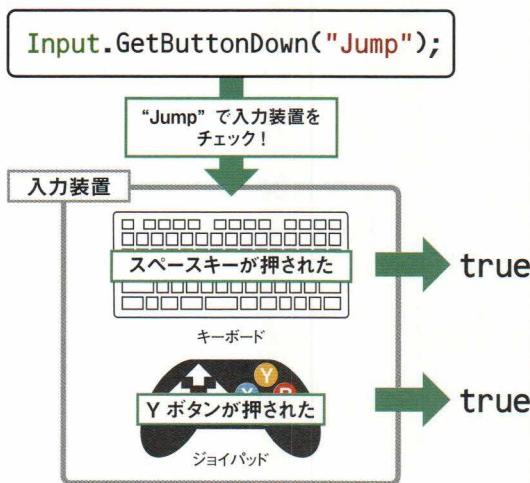
Project Settings ウィンドウの [Input Manager] タブを選択すると、入力設定のリストが表示されます。30個の設定が並んでいますね。一番上の「Horizontal」の左側にある三角ボタンをクリックして、開いてみてください。

「Negative Button」の欄に「left」、「Positive Button」の欄に「right」と書かれています。また「Alt Negative Button」には「a」、「Alt Positive Button」には「d」とあります。Positiveがプラスで右方向、Negativeがマイナスで左方向を表していて、つまり、パソコンの左右キーと [A] キー、[D] キーに対応するという意味です。

「Type」に「Key or Mouse Button」、「Axis」に「X axis」とありますが、これは「キーボードとマウスからの入力をX軸の入力として受け付ける」という設定を行う項目です。



下のほうにもう1つ「Horizontal」という設定があります。開いてみると、「Type」に「Joystick Axis」、「Axis」に「X axis」とあります。



これはゲームコントローラーのアナログスティックをX軸の入力として受け付けるという設定です。つまり、「Horizontal」という名前を引数として `Input` クラスの `GetAxisRaw` を呼べば、パソコンのキーボードとゲームコントローラーに対応してくれるということです。

その他にも「Fire1」(攻撃)、「Jump」(ジャンプ)などのようにゲーム中の抽象的な動作名で入力の設定が作られています。この入力はこのあとで説明する、`GetButtonDown` メソッドで受け取ることができます。

この名前を使うことでスクリプトからは具体的なキーなどを意識せず操作の設定ができます。また、この `Input Manager` の設定を書き換えて対応キーを変更することも可能です。`Input` クラスのメソッドについては別の Tips で説明していますので、そちらを参考してください。

**参照** → 「`Input` クラスのいろいろな入力メソッド」 [111 ページ](#)

## ◆ FixedUpdate メソッド

次に、新しく追加した **FixedUpdate** メソッドを解説しておきましょう。**FixedUpdate** メソッドは毎フレーム、必ず一定の間隔で（Unity の初期設定では 0.02 秒ごと、1 秒間に 50 回）呼ばれるメソッドです。

物理シミュレーションの処理は **FixedUpdate** メソッドのあとに行われます。これは物理挙動の計算は一定間隔で行わないと動きにズレが出るからです。ですから物理系の処理は **FixedUpdate** メソッドを追加して、そこに書くようにします。

**Rigidbody2D** クラス（**body** 変数）の持つ **velocity** 変数はそのゲームオブジェクトの現在の移動速度を表す **Vector2** 型の変数です。この変数に値を入れて、**Rigidbody2D** コンポーネントに速度を操作する処理をしています。**Vector2** についてはこのあとの「座標とベクトル」を参照してください。**x** には 3 を掛け算（「\*」は掛け算）しており、結果として

- 右: 3.0f
- 左: -3.0f
- 押されてない: 0.0f

となり、右の場合は右に「3」の速度で移動、左の場合は左に「3」の速度で移動、押されてない場合は速度「0」、つまり停止する、という結果になります。



### Update メソッドと FixedUpdate メソッドの使い分け

入力系の処理は **Update** メソッドで行い、物理を使った移動などの処理は **FixedUpdate** メソッドで行うようにしましょう。



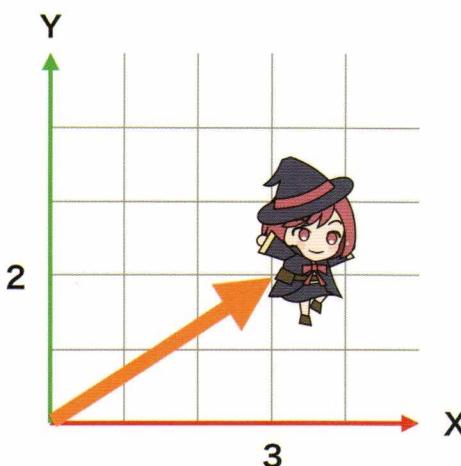
Tips

## 座標とベクトル

Vector2は2次元座標（X軸とY軸）の値を決める型です。そしてこのVector2型はVector2メソッドを使って変数として作ることができます。なお、ここで出てきたnewというキーワードはその右側にあるクラスなどの値を新しく作るためのものです。

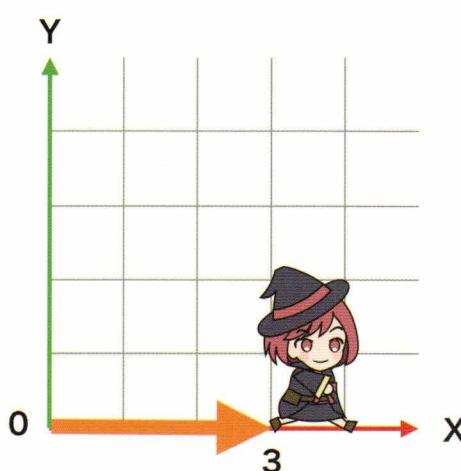
Vector2メソッドは以下のような定義になっています。

```
Vector2 Vector2(float x, float y);  
戻り値 メソッド名 (引数1、引数2)
```



引数にfloat型でxとyの2つの値を取り、戻り値としてVector2型の値を返すメソッドになっています。このVector2型は「ベクトル」という形でX軸とY軸を表しています。

例として、Xが3、Yが2というベクトルがあったとします。それをグラフにすると図のようになります。このときの矢印の向きが物体が動く方向です。この場合だと右斜め上に向かっているということですね。



矢印の長さは速さを表します。XとYの値が大きくなればなるほど矢印の長さは伸びます。つまり「速度が上がる」ということです。

Yの値が0になった場合、矢印はまっすぐ右方向に伸びることになります。つまり右真横に3の速度で進むということになります。また、Xがマイナスだと左方向に、Yがマイナスだと下方向に向かうというわけです。

先ほどの移動速度を作るスクリプトでは、X値は3が掛け算されており、Y値は「rbody.velocity.y」と指定しているように、現在の速度をそのまま使っています。これはつまり縦方向には物理シミュレーションによって、重力で下に引っ張られる力だけがかかるということになります。

## Unityエディターからパラメーター変更をしよう

それではスクリプトをもう少し書き換えてみましょう。以下のようにスクリプトを追記してみてください。変更箇所はハイライトしています。

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

public class PlayerController : MonoBehaviour
{
    Rigidbody2D rbody; // Rigidbody2D型の変数
    float axisH = 0.0f; // 入力
    public float speed = 3.0f; // 移動速度

    // Start is called before the first frame update
    void Start()
    {
        // Rigidbody2Dを取ってくる
        rbody = this.GetComponent<Rigidbody2D>();
    }

    // Update is called once per frame
    void Update()
    {
        // 水平方向の入力をチェックする
        axisH = Input.GetAxisRaw("Horizontal");
        // 向きの調整
        if (axisH > 0.0f)
        {
            // 右移動
            Debug.Log("右移動");
            transform.localScale = new Vector2(1, 1);
        }
        else if (axisH < 0.0f)
        {
            // 左移動
            Debug.Log("左移動");
            transform.localScale = new Vector2(-1, 1); // 左右反転させる
        }
    }
}
```

```
        }

    void FixedUpdate()
    {
        // 速度を更新する
        rbody.velocity = new Vector2(speed * axisH, rbody.velocity.y);
    }
}
```

移動速度の値 `3.0f` を `speed` という変数で定義し、変数の先頭には `public` を付けています。`public` キーワードを付けたものは全体に公開されるのでしたね。

ここで Unity に戻って、キャラクターのゲームオブジェクトを選択し、インスペクタビューの「Player Controller (Script)」を見てください。戻る前に、書いたスクリプトを保存するのを忘れないようにしましょう。



上の図のように `Speed` というテキストボックスが増えています。ここにはスクリプトで設定した数値が入っています。

この値を編集することでスクリプトを触らなくても速度が更新可能になります。つまり、`public` を付けたクラスの変数は Unity エディターから変更可能になるのです。

## キャラクターを反転させよう

`Update` メソッドの中で、キャラクターを左方向に移動させた場合に、向きの反転を行うようにしました。入力されたキーが「左」であればキャラクターを左右反転させています。

入力が「左」だった場合 (`axisH` 変数は `0.0` より小さくなる) に、`Transform` コンポーネントの `localScale` の X を `-1`、Y を `1` に、右だった場合 (`axisH` 変数は `0.0` より大きくなる) には X を `1` に、Y を `1` にしています。

`localScale` は拡大縮小率を設定するパラメーターなのですが、マイナスの値にすることで表示を反転させられるというわけです。

## if 文で条件分岐をしよう

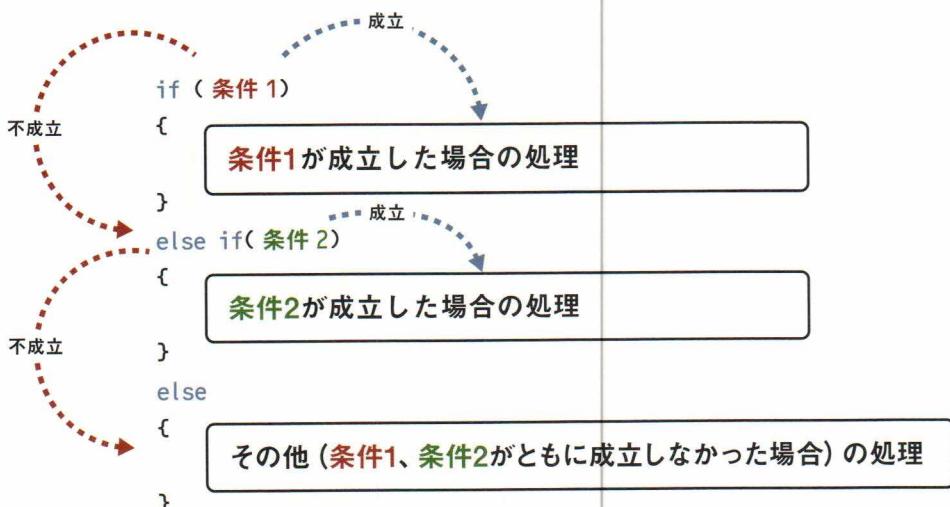
ここで出てくる、「`if { … }`」という書き方は **if 文** という「プログラムの条件分岐」です。

**if 文** は以下のように書きます。「条件」の値が `true` (はい／正しい) か `false` (いいえ／正しくない) で処理が分岐します。

```
if ( 条件 )
{
    条件が成立した場合
}
else
{
    条件が成立しなかった場合
}
```

`if ( 条件 )`が成立した場合は、`if( 条件 ) { … }`の波カッコの中の処理を、成立しなかった場合は、`else { … }`の中の処理を行います。

また、以下のように書くことで、いくつかの条件を連続してチェックすることができます。



ここでは、`axisH`変数が0より大きいか小さいかで判断をしています。この`axisH`変数と`0.0f`の間にある「`>`」は、「比較演算子」といいます。左右の値を判断して、`true`（はい）か`false`（いいえ）で判断をします。

比較演算子には他にも以下のようなものがあります。

- 値1 == 値2 : 値1と値2が同じであれば `true`（はい）
- 値1 != 値2 : 値1と値2が違えば `true`（はい）
- 値1 > 値2 : 値1が値2よりも大きければ `true`（はい）
- 値1 >= 値2 : 値1が値2よりも同じか大きければ `true`（はい）
- 値1 < 値2 : 値1が値2よりも小さければ `true`（はい）
- 値1 <= 値2 : 値1が値2よりも同じか小さければ `true`（はい）

## ゲームを実行しよう



それでは、この状態でゲームを開始してみましょう。プレイヤーキャラクターを左に移動させると、キャラクターの向きも左に向くようになります。

 Tips

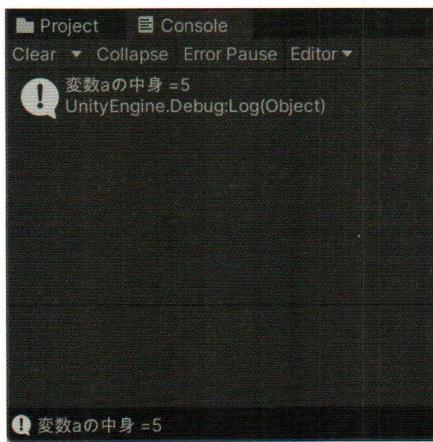
### デバッグログを表示する

メソッド内にある Debug.Log という記述ですが、これは引数の文字列をコンソールに表示するためのメソッドです。

```
Debug.Log(" 左移動 ");
```



プロジェクトビューの上に「Console」というタブがあります。表示をこちらに切り替えてみてください。Debug.Logから出力されたログが表示されています。



このログ表示をうまく使うことで、現在プログラムがどこまで進んで、どこが実行されているのかを知ることができます。

また、このDebug.Logには文字列だけでなく、変数も表示することができます。

```
int a = 5;
Debug.Log("変数 a の中身 =" + a);
```

このログ表示の結果は図のようになります。



### 楽するプログラムは正義

「若いときの苦労は買ってでもせよ」などという「ことわざ」があります。しかしプログラムを作るうえでは、うまい具合に手を抜いて楽をするのがいいプログラマーだと思います。

この場合の「手を抜く」というのは「適当なプログラムを書いてもいいよ」ということではなく、「行数を少なく、短いプログラムを書いていきましょう」ということです。ながーいプログラムを書くよりも少ない行数で同じことができれば、楽ですよね。

プログラムを書いていると、ほとんど同じようなことをいくつもの箇所に書くことがあります。プログラム初心者のうちはコードをコピーして、同じプログラムコードをあちこちに量産しがちです。これは一見楽なように感じますが、プログラムが大きく長くなつて、バグが発生した場合、間違った箇所を見つけるのに非常に苦労します。

同じようなプログラムがある場合は、その仕事をさせるクラスやメソッドを作って、それに仕事をまとめて任せてしまします。そうしておけば、何かあってもその1カ所を見れば不具合を特定しやすくなるのです。

