本書内で作成したサンプルのプロジェクトならびに、ゲーム素材、スクリプトは、本書のサポートページよりダウンロード可能です。サポートページ内のリンクをクリックして、ダウンロードページへ進んでください。

▶ **本書サポートページ**

URL https://isbn2.sbcr.jp/06657/

サンプルファイルはzip形式で圧縮されております。ダウンロード後にローカル環境に展開・保存してご使用ください。
サンプルファイルは、以下のようなフォルダ構成になっております。

「Sample_PC」フォルダ

Chapter3 ～ Chapter8に掲載したサンプルゲーム（PC版）のプロジェクトファイル一式が収録されています。

「Sample_Smartphone」フォルダ

Chapter3 ～ Chapter8に掲載したサンプルゲーム（スマートフォン対応版）のプロジェクトファイル一式が収録されています。

「ゲーム素材」フォルダ

Chapter3 ～ Chapter8に掲載したサンプルゲーム用の素材データが収録されています。

「Listデータ」フォルダ

Chapter2 ～ Chapter8に掲載したスクリプト（List）を、テキストデータ形式で収録してあります。

2-1 スクリプトとは

スクリプトとは、ゲーム中のオブジェクトを動かすための台本のようなものです。映画や舞台では役者さんの動き方を台本に書くように、Unityではオブジェクトの動き方をスクリプトに書きます。スクリプトを書き終えたら、オブジェクトに渡す（アタッチする）ことで、オブジェクトをスクリプト通りに動かすことができます。

2-1-1 スクリプトを習得する極意

これから学習を始めるにあたって、スクリプトを習得するための極意を紹介しましょう。スクリプトも日本語や英語と同様に「言語」です。言語を習得する王道が「読んで、書いて、話す」ことであるように、スクリプトもこの3つがとても大切です。そして、マスターするには量をこなしてください。「話す」は冗談みたいですけど、誰かに言葉で説明することで自分の頭にも定着します。是非試してみてくださいね！

Fig.2-1 スクリプトとオブジェクト

スクリプト言語習得の極意

できるだけ多くのスクリプトを読んで、書いて、話す！

2-2 スクリプトを作成しよう

ここから、実際にスクリプトを書いていきます。スクリプトを身につけるために、これから紹介するサンプルを実際に書いて、動作を確認してください。まずは、テスト用のプロジェクトを作成して、スクリプトファイルを準備するところから始めます。

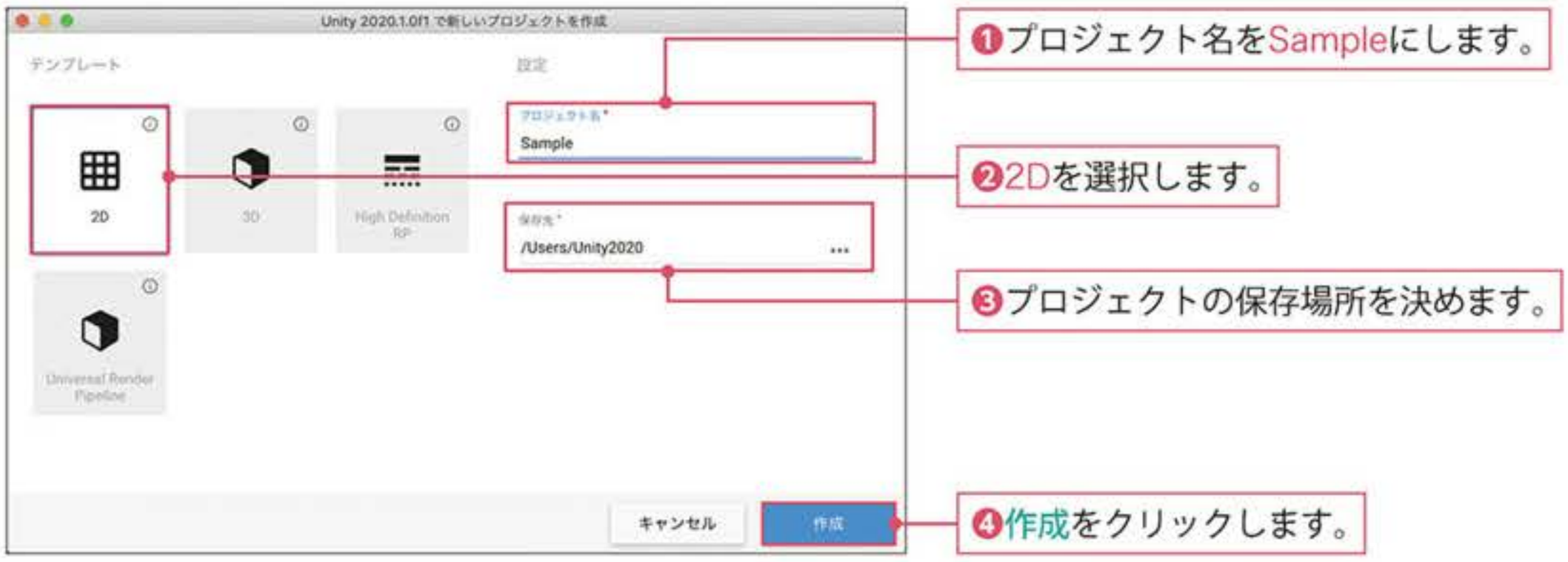
2-2-1 プロジェクトの作成

まずはスクリプトをテストするためのプロジェクトを作成します。Unity Hubを起動して、最初に表示される画面の右上にある新規作成をクリックしてください。Unityエディタを既に起動している場合には、画面上部のツールバーからFile→New Projectを選択します。

Fig.2-2 プロジェクトの作成画面

「新規作成」をクリックすると、プロジェクトの設定画面に進みます。プロジェクト名は「Sample」にします。プロジェクトの保存場所を入力し、テンプレートの項目で「2D」を選択します（ここでは2Dゲームのプロジェクトを作成します）。右下の青色の作成ボタンをクリックすると、指定したフォルダにプロジェクトが作成され、Unityエディタが起動します（Fig.2-3）。

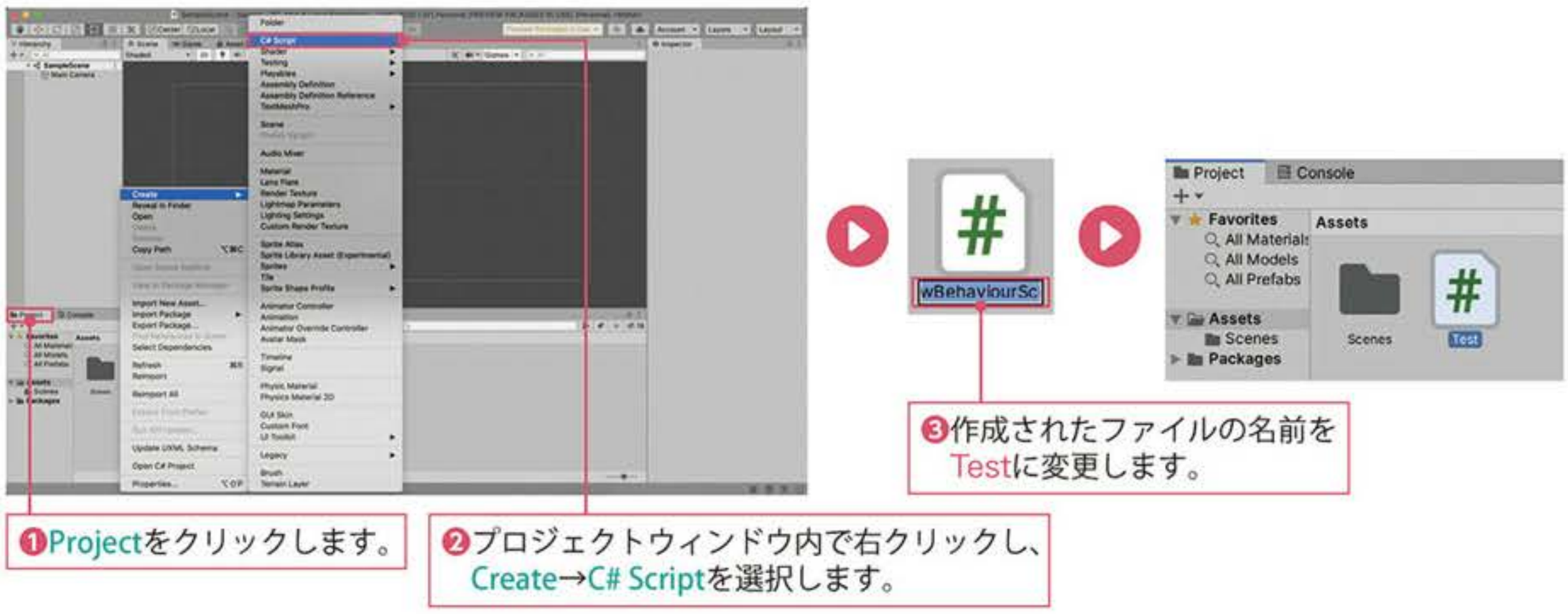
Fig.2-3 プロジェクトの設定画面



2-2-2 スクリプトの作成

Unityエディタが起動したら、Projectタブを選択して、プロジェクトウィンドウ内で右クリックし、表示されるメニューからCreate→C# Scriptを選択します。ファイルが生成された直後はファイル名が編集状態になるので、ファイル名を「Test」に変更して決定します。

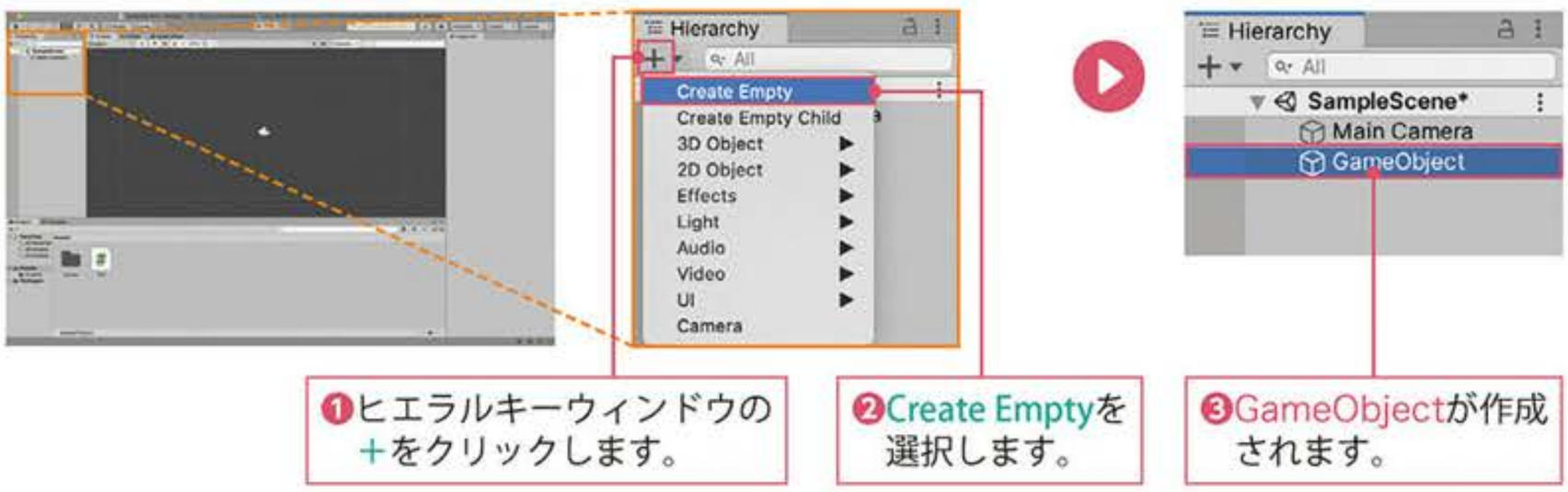
Fig.2-4 スクリプトの作成



作成したスクリプトを動かすためにゲームオブジェクトを追加します（ゲームオブジェクトが必要な理由は、次ページのTipsを参照してください）。Fig.2-5のようにUnityエディタ左上のヒエラルキーウィンドウから＋→Create Emptyを選択してください。ヒエラルキーウィンドウにGameObjectが作成されます。

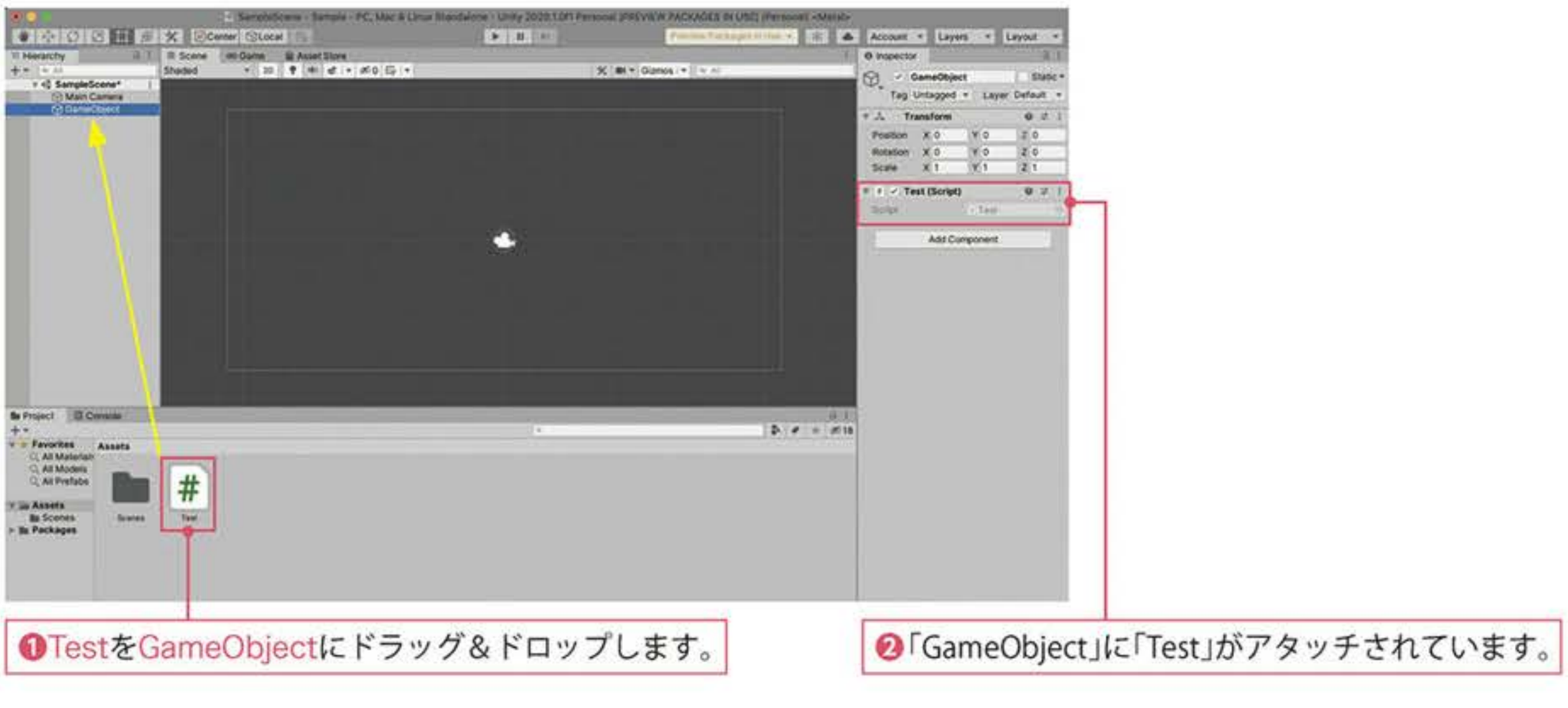
ここで作成したゲームオブジェクトは、中身は空っぽのもので、何も機能を持ちません。このようなゲームオブジェクトを「空のオブジェクト」と呼びます。

Fig.2-5 ゲームオブジェクトの作成



作成した「Test」スクリプトを、ヒエラルキーウィンドウの「GameObject」にドラッグ＆ドロップします。これはアタッチという作業で、**スクリプトをゲームオブジェクトに結び付けることで、スクリプトを実行できるようにしています。**また、アタッチできているかどうかはインスペクターで確認できます。

Fig.2-6 スクリプトのアタッチ



Tips スクリプトを動かすには「アタッチ」が必要

スクリプトを動かすには、何らかのゲームオブジェクトと結び付ける必要があります。例えば、キャラクターを動かすためのスクリプトは、キャラクターにしたいオブジェクトと結び付けることで動作します。また、カメラを操作するスクリプトはカメラオブジェクトと結び付けることで動作します。このように、作成したスクリプトを動かすためには特定のオブジェクトにアタッチする必要があります。

2-3 スクリプトの第一歩

先ほど作成したスクリプトファイルには、既にひな形となるスクリプトが記述されています。まずはスクリプトを開いて、中身を見ることから始めましょう。

2-3-1 スクリプトの概要

まずは、スクリプトがどのようなものかを簡単に見てみましょう。プロジェクトウィンドウの「Test」アイコンをダブルクリックすると、スクリプトを記述するためのVisual Studio※が起動してスクリプトが表示されます。ファイルには自動的に作成されたスクリプトが書いてあります。

Fig.2-7 スクリプトファイルを開く

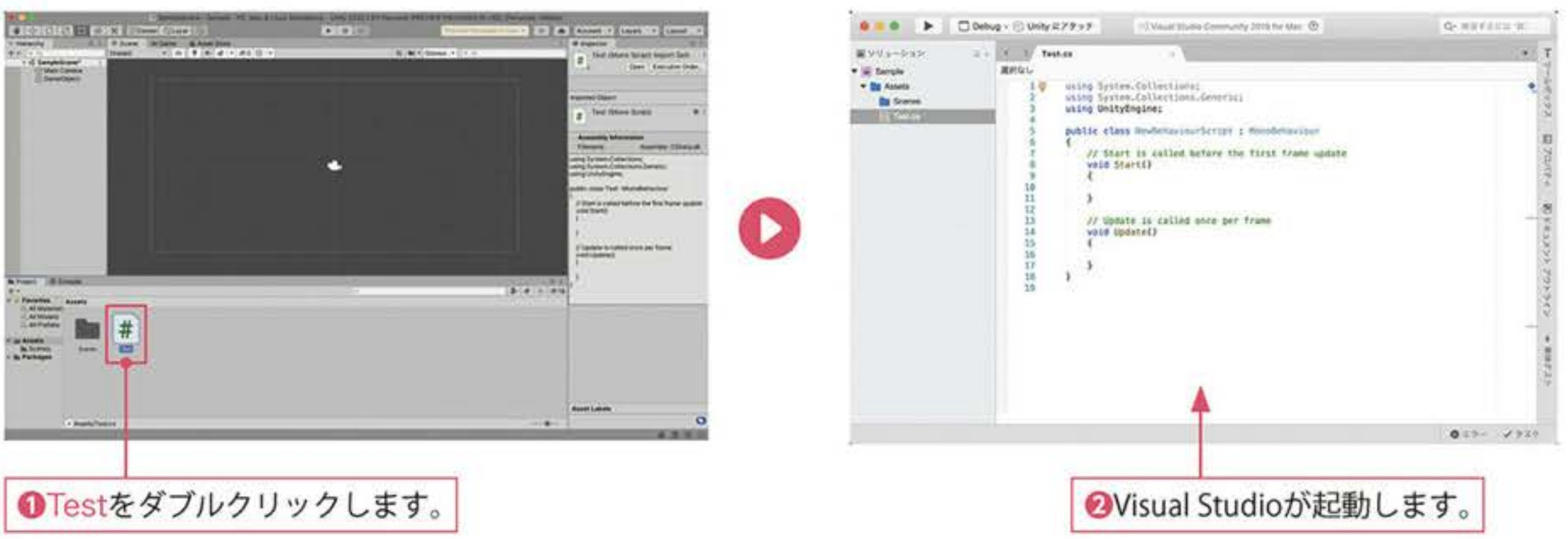
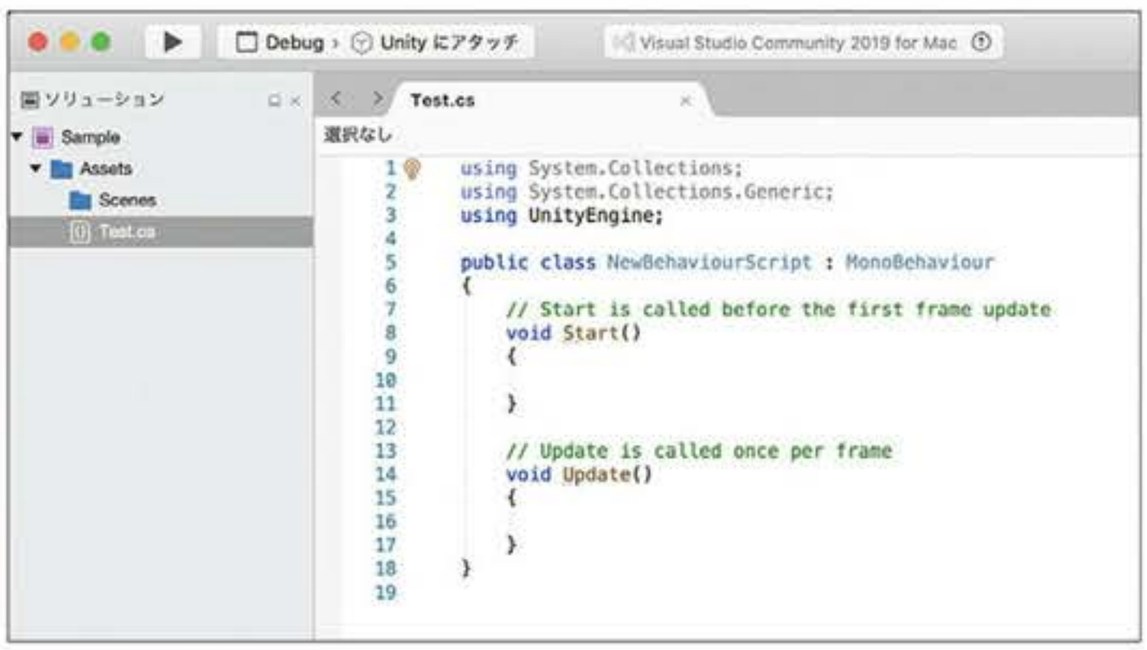


Fig.2-8 スクリプトが表示される



「Test」スクリプトファイルは、List 2-1のようになっています。

List 2-1 自動的に作成されるスクリプト

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Test : 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 }
```

usingとかpublicとかclassとか、謎の単語がたくさん並んでいて暗号のように見えますが、この章を読み終わる頃には理解できるようになるので安心してください！

1行目と2行目のSystem.CollectionsとSystem.Collections.Genericは、データを格納するための型を提供してくれます（型については後ほど解説します）。3行目のUnityEngineは、Unityを動かすために必要な機能を提供してくれます。ここを書き換えることはほとんどないので、軽く見ておくだけにします。

5行目ではクラス名を決めています。C#で書かれたプログラムは、クラスという単位で管理します。ここでは、**「クラス名」＝「スクリプト名」**ということを覚えておけばよいでしょう。クラスについては、後ほどあらためて解説します。

スクリプトを実行した時に処理される内容は、6行目の「{」から、18行目の「}」の間に書きます。「{」と「}」で挟まれた部分をブロックと呼びます。「{」と「}」は必ず対になるようにしてください。{} や () を閉じ忘れると、エラーになります。また、「{」の位置は、次のように現在の行に続けて書くこともできます。

※Visual Studio
Visual StudioはMicrosoft社が提供する統合開発環境（Integrated Development Environment）です。統合開発環境とは、プログラム作成に必要なツール一式（テキストエディタ・デバッグ機能・プロジェクト管理機能など）をひとまとめにしたアプリケーションです。
実行環境によってはVisuall Studioの起動時にMicrosoftアカウントへのサインインを求められることがあります。その際には、アカウントへサインインして、Visual Studioをご利用ください。

```
public class Test : MonoBehaviour {
    ※ここにクラスの処理を書きます。
}
```

7行目と13行目の「//」で始まる行はコメントと言い、「//」の後に記述した文字はスクリプト実行時に無視されます。記述したスクリプトを消したくないけど無効にしたい時や、メモを残したい時などに便利です。「//」は行の途中に入れることもできます。その場合は、「//」から後ろの部分だけが無視されます。

8行目にStart、14行目にUpdateと書かれた部分があり、これらはそれぞれStartメソッド、Updateメソッドと呼ばれています。今はまだ何も処理が書かれていませんが、次のように、ブロック内に処理の内容を書いていきます。

```
void Start()
{
    ※ここに処理の内容を書きます。
}
```

「スクリプトを実行すると、StartやUpdateのブロック内に書かれた処理が実行されます」という風に今は覚えておきましょう。メソッドについては、後ほど詳しく解説します。

フレームと実行タイミング

ゲームの画面は、映画やアニメと同じく1コマ1コマの絵をパラパラ漫画方式で表示しています。この１コマ１コマの絵のことをフレームと呼び、1秒間に表示される枚数は「FPS(Frame Per Second)」という単位で数えます。基本的に映画では1秒間に24フレーム(24FPS)、ゲームでは1秒間に60フレーム(60FPS)の速度で絵を切り替えてアニメーションにします。

ただし、1秒間に60フレーム切り替えると設定されていても、実際にはユーザからの入力による表示内容の変化やシステムにかかる負荷などによって、フレーム間の時間は1/60秒より速くなったり遅くなったりします。実際に前のフレームから何秒たったのかは、Time.deltaTimeという仕組みで知ることができます(Fig.2-9)。このTime.deltaTimeは後ほど本書内でも出てきます。今は詳しいことは知らなくても大丈夫ですので、フレームとTime.deltaTimeのイメージをつかんでおいてください。

Fig.2-9 フレームとは？

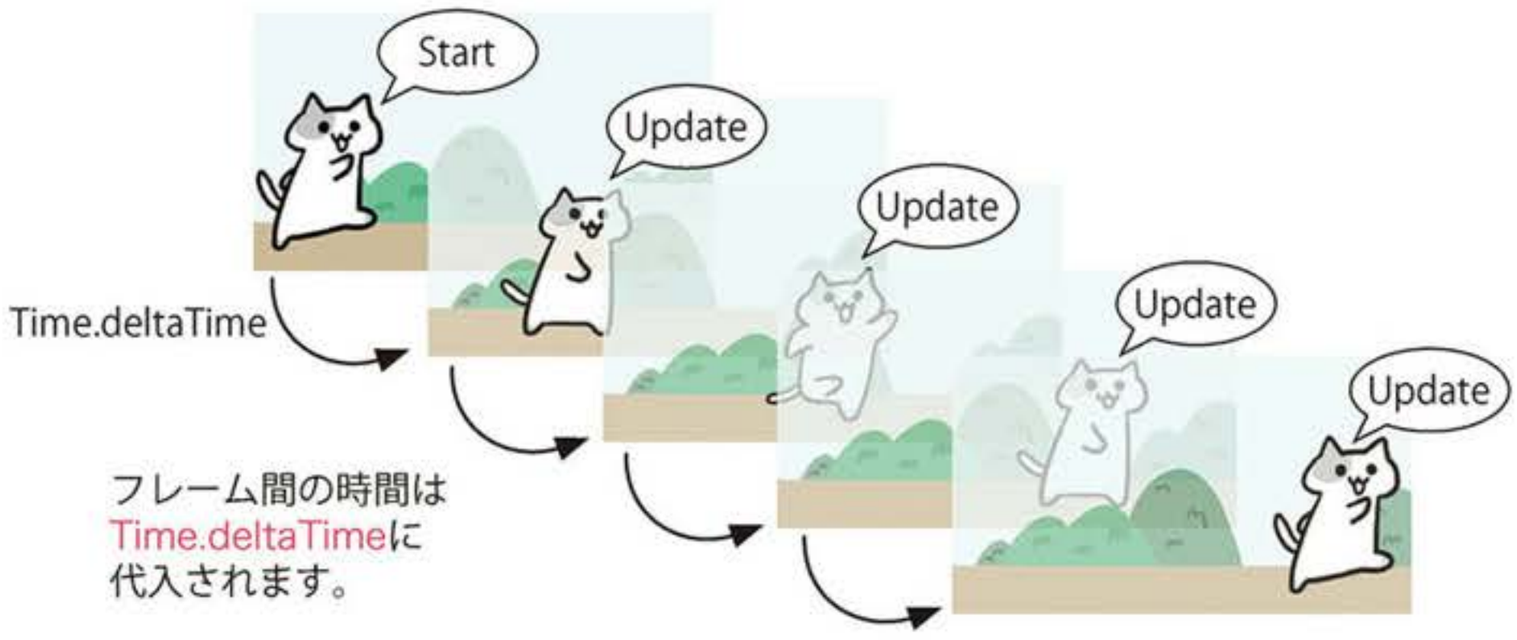
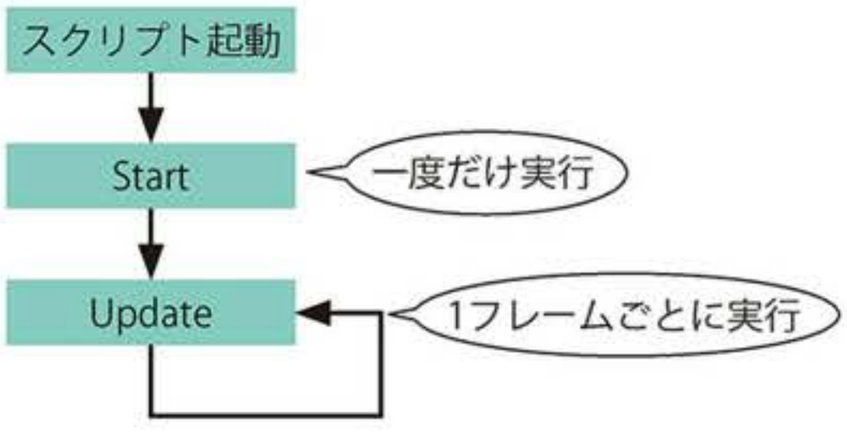


Fig.2-9のイラストに、StartやUpdateという吹き出しが付いていることに注目してください。スクリプトを起動すると、直後にStartメソッドの内容が一度だけ実行されます。続けて、フレームごとにUpdateメソッドが繰り返し実行されます。例えば、キャラクターが右に歩くアニメーションを作る場合には、最初にStartメソッドでキャラクターを表示し、その後はフレームごとにキャラクターを少し右の位置に移動させる、という感じで処理を行っていきます。

```
void Start()
{
    ※キャラクターを表示する処理
}

void Update()
{
    ※現在のキャラクターを少し右に移動する処理
}
```

また、キャラクターの表示だけでなく、当たり判定やキー操作などの処理も、フレームごとに行います。この流れをまとめると、Fig.2-10のようになります。

Fig.2-10 スクリプトの大きな流れ



2-3-2 「Hello, World」を表示する

それでは実際に処理を行うスクリプトを書いてみましょう。

ここでは、「Hello, World」とUnityエディタのコンソールウィンドウ※に表示する処理を作ります。List 2-2のようにスクリプトを記述してください(2章では、スクリプトを起動した直後に一度だけ実行されるStartメソッドを使ってスクリプトの練習をしていきます)。

List 2-2 「Hello, World」を表示するスクリプト

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Test : MonoBehaviour
6  {
7      void Start()
8      {
9          // Hello, Worldをコンソールウィンドウに表示する
10         Debug.Log("Hello, World");
11     }
12 }
```

今回のスクリプトでは、Startメソッドの9行目と10行目に記述を追加しています。9行目はコメントなので実行時に無視されます。また10行目のようにDebug.Log()と書くと、()のなかに書いた文字列をコンソールウィンドウに表示してくれます。

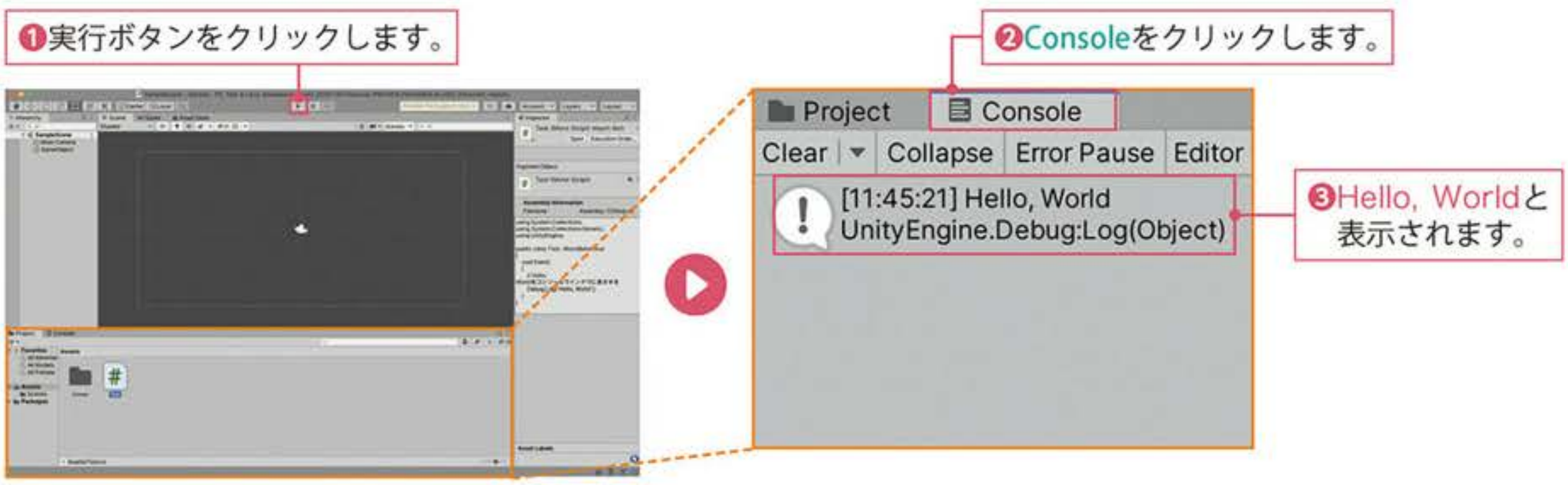
```
Debug.Log("コンソールウィンドウに表示する文字列");
```

ここで「文字列」という言葉が出てきました。文字列とは複数の文字を連ねたものです。スクリプト内に文字列を記述する場合は、文字列の前後を「"」(ダブルクォーテーション)でくくります。もし、「"」でくくらずにDebug.Log(Hello, World);と書いてしまうとエラーになります。「1234」のような数字でも、「"1234"」のように「"」でくくるとスクリプト内では文字列として扱われます。

スクリプトの実行

スクリプトが記述できたら保存してUnityエディタに戻り、シーンビューの上にある実行ボタンをクリックしてください。そのうえで、Unityエディタ下側付近のConsoleタブをクリックして、プロジェクトウィンドウからコンソールウィンドウに切り替えてください。

Fig.2-11 Debug.Logの表示確認



ゲームを実行すると、ヒエラルキーウィンドウに登録されたゲームオブジェクトがアクティブになり、それに結び付けられた(アタッチされた)スクリプトが起動します。スクリプトが起動すると、クラスのなかにあるStartメソッドの処理が、最初に一度だけ実行されます。その後はゲームが終了するまで、1フレームごとにUpdateメソッドの処理が繰り返されます(61ページ)。実行ボタンをもう一度クリックしてゲームを停止します。

コンソールウィンドウを見てみると、ちゃんと「Hello, World」と表示されていますね！ これでスクリプトを書くための第一歩が踏み出せました！ これからたくさんスクリプトを書いて、徐々に慣れていきましょう。

シーンの保存

ここまでの作業内容を保存するために、シーンを保存しておきます。ツールバーからFile→Save Asを選択し、シーン名を「SampleScene」として保存してください。保存できるとUnityエディタのプロジェクトウインドウに、シーンのアイコンが出現します。

≻Tips≺ 「;」とは

`Debug.Log("Hello, World");`の行の最後には「;」(セミコロン)が付いています。小さくて見落としやすいですが、コンピュータにスクリプトの区切りを教える大切な役割があります。これを忘れるとエラーになるので注意してください。

セミコロンを書き忘れた場合でも、「セミコロンを忘れているよ！」というエラーは出ません。書き忘れた次の行にエラーが出ることが多いので、直前の行をチェックしてみてください。

※コンソールウィンドウ
エラーや警告を表示したり、スクリプト内で使用している値を表示したりすることができます。また、スクリプト中の任意のタイミングで指定した文字列を表示できるため、デバッグする際によく使います。

2-8 クラスを作ってみよう

メソッドは処理をまとめたものでしたが、クラスは**メソッドと変数をまとめたもの**です。メソッドと変数をまとめると、どのようなよいことがあるのか？ それを理解していきましょう。

2-8-1 クラスとは

Unityでゲームを作る場合、プレイヤ、敵、武器、アイテムなどの「モノ」ごとに、その動きを定義するスクリプトを作成します。この場合、メソッドのような「処理単位」ではなくて「モノ単位」でスクリプトを作れた方が便利です。

具体的に、プレイヤのスクリプトを作る場合を考えてみましょう。プレイヤにはHPやMPなどのステータス（変数）や、攻撃や防御、魔法などのアクション（メソッド）が必要です。

Fig.2-37 クラスはモノ単位

これらの変数とメソッドをひとまとめにせず、バラバラに実装してしまうと、どの変数とどのメソッドが関連しているのかわからなくなってしまいます。**クラスを使えば関係のある変数とメソッドをひとまとめにできるので、スクリプトを管理しやすくなる**のです。

クラスの書式を単純化したものは下記の通りです。classというキーワードに続けてクラス名を書き、そのなかにクラスで使う変数とメソッドを記述します。クラスで使う変数をメンバ変数、クラスで使うメソッドをメンバメソッドと呼びます。

```
class クラス名
{
    メンバ変数の宣言;
    メンバメソッドの実装;
}
```

作成したクラスはintやstringなどと同様、型のように使えます。つまり、Playerクラスを作ればPlayer型を使うことができるようになります（厳密にはクラスと型は別物です。詳しくはC#の文法書を参照してください）。

`int num;`と書けばint型のnum変数を作れるように、`Player myPlayer;`と書けばPlayer型のmyPlayer変数を作ることができます。この状態ではmyPlayer変数の箱のなかは空っぽです。

int型のnum変数には「2」や「1500」など数値を代入するように、Player型のmyPlayer変数には「プレイヤの実体」を代入します。この「実体」のことをインスタンスと呼びます。インスタンスを生成する例は2-8-3項で説明します。

Fig.2-38 インスタンスの意味


```
int num;   Player myPlayer;
```

myPlayer変数が持つメンバメソッドやメンバ変数を使うには、「myPlayer.メンバメソッド名（またはメンバ変数名）」と記述します。この先「.」でつながった記述がたくさん出てきます。「〇〇.××」を見たら**「〇〇クラスが持つ××メソッド（または変数）を使っている」**と理解しておきましょう。

Fig.2-39 メンバメソッドの使い方


```
myPlayer.Attack();
```

Tips Unityが用意してくれているクラスもある！

クラスは自分で作る以外に、Unityに最初から用意されているものもあります。2-9節で紹介するVectorクラスや、ログの表示などに使うDebugクラスなどです。Unityを使いこなすためには、クラスの概念をしっかりと理解することが大切です。

2-8-2 クラスを作成する

説明ばかりだったので、まだイメージがわきにくいと思います。ここでは具体的にFig.2-39で示したPlayerクラスを実装しながら理解していきましょう。ここまで使用してきた「Test」スクリプト（Test.cs）に、Playerクラスを追加しましょう。Playerクラスは、Testクラスの外側に追加していることに注意してください。

List 2-29 Playerクラスの作成

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Player
6  {
7      private int hp = 100;
8      private int power = 50;
9
10     public void Attack()
11     {
12         Debug.Log(this.power + "のダメージを与えた");
13     }
14
15     public void Damage(int damage)
16     {
17         this.hp -= damage;
18         Debug.Log(damage + "のダメージを受けた");
19     }
20 }
21
22 public class Test : MonoBehaviour
23 {
24     void Start()
25     {
26         Player myPlayer = new Player();
27         myPlayer.Attack();
28         myPlayer.Damage(30);
```

```
29          }
30  }
```

＼出力結果／

```
50のダメージを与えた
30のダメージを受けた
```

5〜20行目が今回作成したPlayerクラスです。ざっとクラスの構成を眺めると、5行目でPlayerクラスを宣言し、7〜8行目でプレイヤのHPと攻撃力を表すメンバ変数を宣言しています。10〜19行目では攻撃するAttackメンバメソッドとダメージを受けるDamageメンバメソッドを作成しています。メソッドの中身は、後ほど詳しく見ていきます。

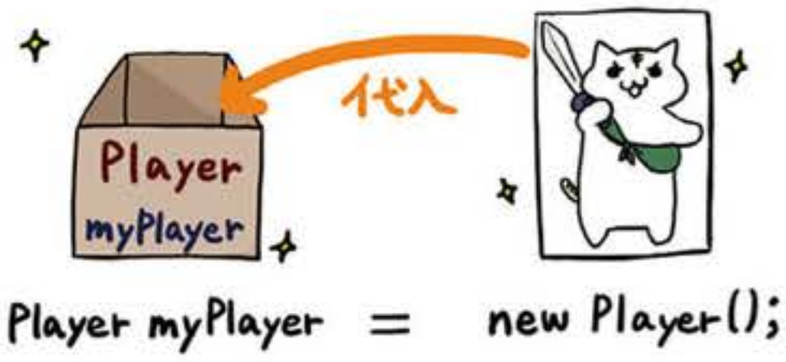
2-8-3 クラスの使い方

次に、作成したPlayerクラスを使う部分を見ていきましょう。List 2-29のStartメソッド内の26〜28行目で、先ほど作ったPlayerクラスのインスタンスを生成して使っています。

まず26行目の左辺で`Player myPlayer`と書いてPlayer型のmyPlayer変数を宣言しています。この段階ではPlayer型の箱を作っただけなので、Player型の実体であるインスタンスを作成して代入する必要があります。

インスタンスを作るため、「new」キーワードに続けて「クラス名()」と書きます（26行目の右辺）。これによりPlayerクラスのインスタンスが作られ、myPlayer変数のなかに代入されます。

Fig.2-40 インスタンスの作成方法



27行目でインスタンスが持つAttackメソッドを`myPlayer.Attack()`として、「変数名.メンバメソッド名()」の書式で呼び出しています。同様に28行目ではDamageメソッドを呼び出しています。

実行ボタンを押して、コンソールウィンドウに結果が表示されることを確認してください。

2-8-4 アクセス修飾子

List 2-29に示したPlayerクラスのメンバ変数とメンバメソッドの先頭には、「public」か「private」というキーワードが付いています。これはアクセス修飾子といって、**「○○.××」という書き方で他のクラスからメンバにアクセスできるかどうか**を表しています。

publicの付いているメンバは他のクラスから呼び出すことができますが、privateの付いているメンバは他のクラスから呼び出すことはできません。Attackメソッドにはpublicが付いているので、`myPlayer.Attack()`のようにAttackメソッドを呼び出すことができます。しかし変数hpにはprivateが付いているので、`myPlayer.hp`と書いてhp変数を呼び出すことはできません。

Fig.2-41 アクセス修飾子

クラス内の変数やメソッドは、すべてpublicを付けてアクセスできるようにしておけばいいんじゃないの？ と思うかもしれません。もちろん、それでもスクリプトは動きますし、ゲームも作れます。それでもprivateがあるのは、他の人に自分が作ったクラスを使ってもらう際、**「privateメソッドは使わないでね、private変数は書き換えないでね」**という意思表示ができるからです。逆に、他の人が作ったクラスを使う時は、publicの付いた変数とメソッドだけを見ればよいのです。

Fig.2-42 クラスはpublicを見て使う

なお、アクセス修飾子を省略するとprivateであると見なされます。したがって、公開したい変数やメソッドにpublic修飾子を付けるだけでも動作としては問題ありません。

アクセス修飾子の一覧をTable 2-3にまとめておきます。

Table 2-3 アクセス修飾子

アクセス修飾子	アクセス可能クラス
public	すべてのクラスからアクセス可能
protected	同じクラスとそのサブクラスからアクセス可能
private	同じクラスからのみアクセス可能

2-8-5 thisキーワード

AttackメソッドやDamageメソッド内で使用しているhpやpowerなどのメンバ変数の前に「this」というキーワードが付いていることに気が付きましたか？ 「this」は自分自身のインスタンスを指すキーワードです。つまり`this.power`は自分自身のインスタンスが持つpower変数（Playerクラスのインスタンスが持つpower変数）を表します。

thisを付けなくても自身のクラスのメンバ変数を使うことはできます。ただ、List 2-29のAttackメソッド内で、次のようにメンバ変数と同じ名前のローカル変数（power）を宣言した場合、powerとだけ書くとローカル変数の値が優先的に使われます。そこで、メンバ変数を使う時は明示的に「this」を付けておくことで、バグを防ぐことができます。

```
public void Attack()
{
    int power = 9999;
    Debug.Log(power + "のダメージを与えた");
    // ローカル変数が優先され、9999のダメージを与えることになる
}
```

2-8節で説明した内容は、プログラミングの世界ではオブジェクト指向と呼ばれるものです。オブジェクト指向の3大要素には「継承」「アクセス修飾子によるカプセル化」「ポリモーフィズム」があります。ここではpublicやprivateなどのアクセス修飾子を使った「カプセル化」を紹介しました。今回は本書を読むために必要な事項だけを説明しましたが、興味のある方は拙著『確かな力が身につくC#「超」入門』などの文法書を参考にしてください。

Tips 継承

Testクラスの宣言部分（List 2-29の22行目）の後ろに付いている「: MonoBehaviour」は継承（けいしょう）と呼ばれ、Unityが事前に用意したMonoBehaviourクラスの機能をTestクラスに取り込むことを宣言しています。

MonoBehaviourクラスは、ゲームオブジェクトを構成するための基本的な機能がメンバ変数・メンバメソッドとして用意されているクラスです。したがってゲームオブジェクトにアタッチするスクリプトは、MonoBehaviourクラス（あるいはMonoBehaviourを元にしたクラス）を継承する必要があります。

Tips Debug.Logはインスタンスなしで使える？

この節ではPlayerクラスのインスタンスを作り、「インスタンス変数名.メンバメソッド名」という形でメンバメソッドを呼び出していました。一方でここまで何度も使っているDebug.Logメソッドは、「クラス名.メンバメソッド名」という形でメンバメソッドを直接呼び出しています。これはDebug.Logがstaticメソッド（インスタンスを作らなくても使うことのできるメソッド）として宣言されているからです。内容が少々高度になるため、最初のうちは「そういう特殊なメソッドもあるんだ」と考えておくのがよいでしょう。

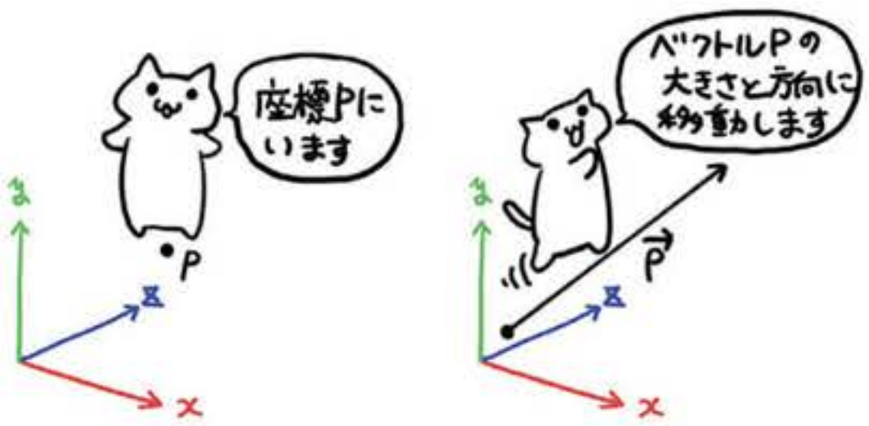
2-9 Vectorクラスを使ってみよう

この章の最後に、これからゲームを作るうえでよく使うVectorクラスを紹介します。Vectorクラスは、キャラクターなどを動かす場合によく使うので、ここでイメージをつかんでおきましょう。

2-9-1 Vectorとは？

3Dゲームを作る場合、空間上のどこにオブジェクトを置くのか、どちらの方向に移動するのか、どちらに力を加えるのかなどを決めるため、**float型の「x, y, z」の3つの値**を扱います。

Fig.2-43 Vector型の使われ方



そこで、これらの値をひとまとめに扱えるようにしたVector3クラス（正確には構造体※と呼ばれるものです）が用意されています。また、2Dゲーム用には、float型の「x, y」の値を持つVector2クラスが用意されています。Vector3クラスを擬似的に書くと、次のような感じになります。

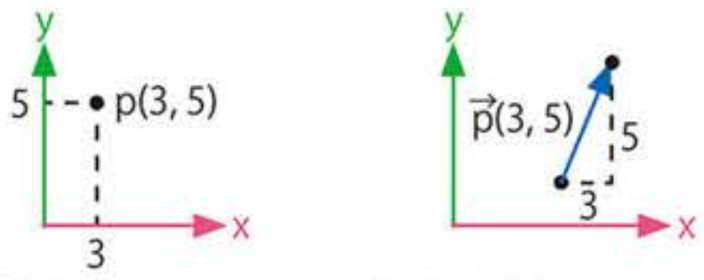
```
class Vector3
{
    public float x;
    public float y;
    public float z;

    // 以下Vector用のメンバメソッドが続く
}
```

※構造体
構造体はクラスと同じく、変数とメソッドをひとまとめにする仕組みです。ただしクラスと比べると使える機能が制限されています。そのかわりに、クラスより高速に動作します。

このように、Vector3クラスにはメンバ変数として「x」「y」「z」、Vector2クラスには「x」「y」が用意されており、座標やベクトルとして使うことができます。

座標として使う場合、「x = 3」「y = 5」とすると(3, 5)の位置にオブジェクトが配置されることを意味します。一方でベクトルとして使う場合は、現在の位置から「x方向に3」「y方向に5」進むことを意味します。

Fig.2-44 座標としてのVector2とベクトルとしてのVector2



座標としてのVector2　ベクトルとしてのVector2

2-9-2 Vectorクラスの使い方

では、Vector2クラスの簡単な使い方のサンプルを見ておきましょう。まずはVector2クラスのメンバ変数に数値を加算するサンプルです。

List 2-30 Vector2クラスの持つメンバ変数に加算する

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Test : MonoBehaviour
6  {
7      void Start()
8      {
9          Vector2 playerPos = new Vector2(3.0f, 4.0f);
10         playerPos.x += 8.0f;
11         playerPos.y += 5.0f;
12         Debug.Log(playerPos);
13     }
14 }
```

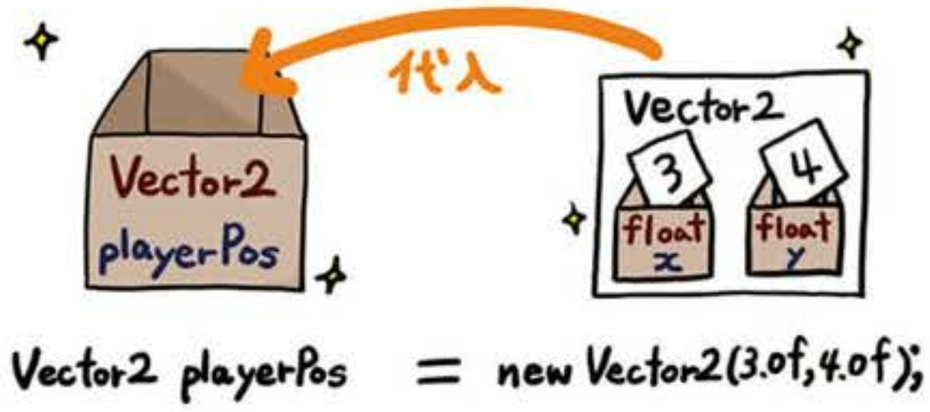
＼出力結果／

```
(11.0, 9.0)
```

9行目の左辺でVector2クラスの変数「playerPos」を宣言しています。これはVector2型の箱を作っただけなので、代入するためのインスタンスを作成する必要があります。そこで、続けて`new Vector2(3.0f, 4.0f)`として、Vector2クラスのインスタンスを作成して代入しています。「new クラス名()」でクラスのインスタンスが作成できるのでしたね（106ページ）。

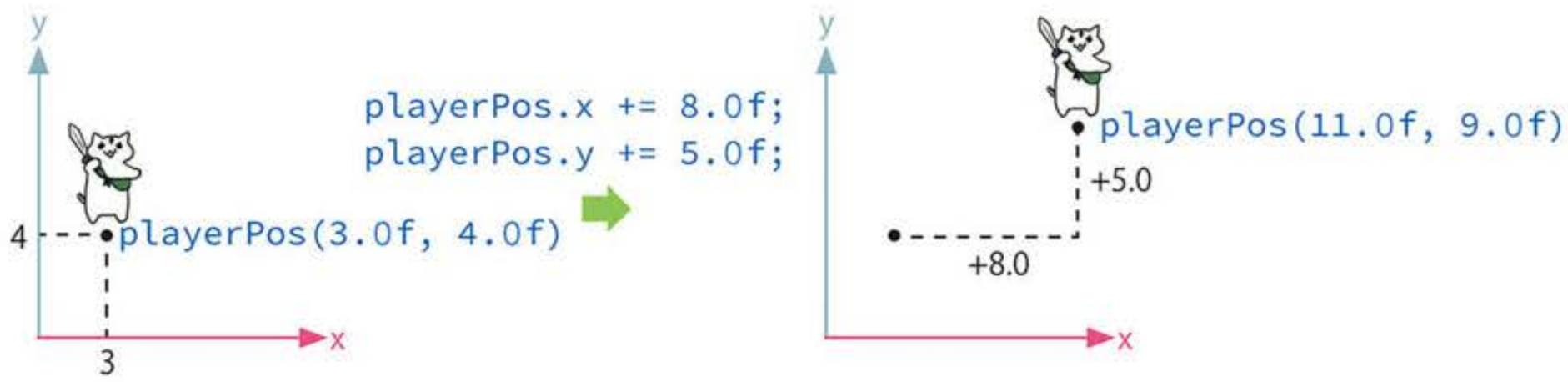
Vector2クラスでは「new」を使ってインスタンスを作る際に、引数を渡すことでメンバ変数を初期化することができます。ここではxを「3.0f」、yを「4.0f」で初期化しています（前述のように、Vector2クラスはメンバ変数としてfloat型の「x」と「y」を持っています）。

Fig.2-45 Vector2のインスタンスの生成



クラスのところで説明した通り、メンバ変数である「x」や「y」の値にアクセスするには「変数名.x」「変数名.y」と記述します（104ページ）。これを利用して、10〜11行目でプレイヤのいるX座標を「8」、Y座標を「5」増加しています。

Fig.2-46 Vector2のメンバ変数への加算



Vector2クラスの変数をゲームブジェクトの座標として使っている場合、値を増加させることで画面上のゲームオブジェクトを正の方向（右あるいは上）に移動できます。逆に値を減少させると、負の方向（左あるいは下）に移動します。

Vector2クラスのメンバ変数（xとy）に対して加算ができることは上記の例で確認できました。次にVector2クラス同士で減算を行うサンプルを紹介します。

List 2-31 Vector2同士の減算

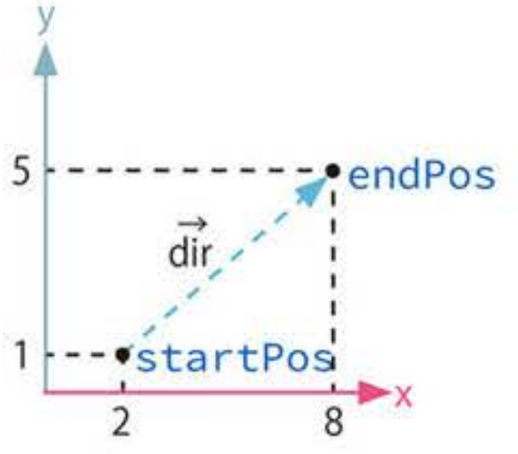
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Test : MonoBehaviour
6  {
7      void Start()
8      {
9          Vector2 startPos = new Vector2(2.0f, 1.0f);
10         Vector2 endPos = new Vector2(8.0f, 5.0f);
11         Vector2 dir = endPos - startPos;
12         Debug.Log(dir);
13
14         float len = dir.magnitude;
15         Debug.Log(len);
16     }
17 }
```

＼出力結果／

```
(6.0, 4.0)
7.211102
```

このサンプルでは「startPos」から「endPos」に向かうベクトル「dir」を求めています。2点の座標からベクトルを求めるため、11行目で「endPos」から「startPos」を減算しています。このように、Vector2同士の減算もできます。

Fig.2-47 Vector2同士の減算



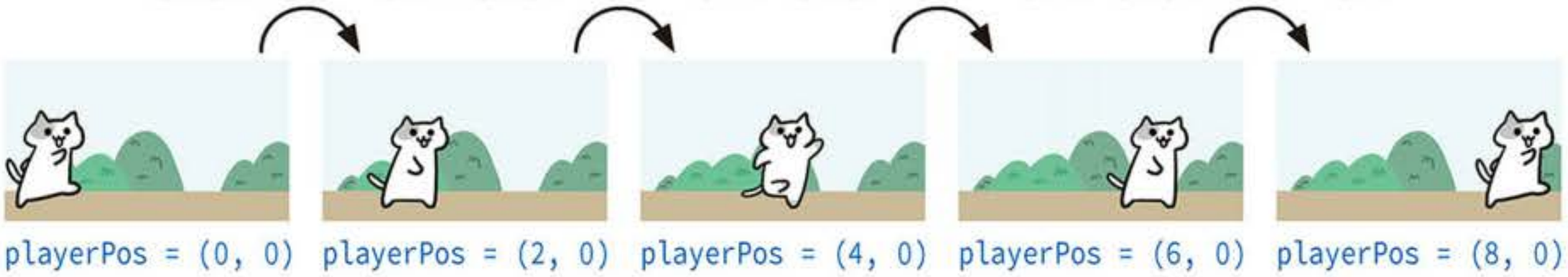
14行目では、「startPos」から「endPos」の間の距離を求めています。この距離は、Fig.2-47のベクトル「dir」の長さに等しいので、Vector2クラスが持つmagnitudeメンバ変数を使い、ベクトル「dir」の長さを求めています。

このようにVectorクラスにはベクトル計算に便利なメンバ変数がいくつか用意されています。一度リファレンスに目を通しておくとよいでしょう。

2-9-3 Vectorクラスの応用

ここまでの例はVectorクラスを座標やベクトルとして扱ってきました。この他にもVectorクラスを加速度や力、移動速度など物理的な数値として使うこともできます。例えばプレイヤの移動速度を、Vector2クラスを使って`Vector2 speed = new Vector2(2.0f, 0.0f);`と書いたとしましょう。このspeedをプレイヤの座標に毎フレーム加算することで、フレームごとにプレイヤをX方向に2ずつ移動させることができます。この具体的な使用例は、3章以降で紹介します。

Fig.2-48 Vector2を速度として使う



3章以降、ゲームを作るためにさまざまなスクリプトを作成します。舞台の脚本家になったつもりで、それぞれのオブジェクトに関して「こんな動きをして欲しい！」と考えながらスクリプトを書いていきましょう。

Tips Visual Studioの挙動

たまにVisual Studioの予測変換機能がきかなくなったり、スクリプト全体に赤い波線が表示されることがあります。その場合は一度、Visual Studioを再起動してみてください。