C#虎の巻

さて、Unityを最大限活用してプログラミングをするためにはC#という言語を学ばなきゃならない。 この文章は、Unityに関してというより、C#を使えるレベルになるために必要な知識や導入をまとめたものだ。あくまで、非常にコンパクトにまとめているだけなので、普通の参考書などと比べると量に乏しいし、網羅的とは言えない。

実際に教えるときには、何かC#の本を使うことはしないが、何か一冊文法書を持っておくといいかもしれない。 以下はオススメの本だ。

- 独習C# (初心者から上級者まで幅広く使える)
- Effective C#/More Effective C# (中上級者向け)

アルゴリズムとは

アルゴリズムとは、ある問題や処理のための手順のことをいう。(日本語では算法と訳される) 例えば、ある会員制サイトにログインするという処理を見てみるとき、

1. ユーザー名とパスワードを入力する
2. マッチするものがあればログインとして3へ、なければ1へ
3. 会員ページを表示する

というような順序を持つ。Unityで作るゲームも同様にこのようなアルゴリズムを記述することで成り立っている。

さて、世の中に存在するあらゆるアルゴリズムは以下の3つの成分によって実は成り立っている。

- **順序**
- **分岐**
- **繰り返し**

これだけのパターンがあればすべてのアルゴリズムを示すことができる。

例えば、朝起きてから大学に行くまでの時間を例にとって考えてみよう。

順序

朝起きてからご飯を食べるという処理は、ご飯を食べてから朝起きるというような人がいないので順序に分類される。

家の玄関のドアを開けるためには、まず鍵を開けてからドアノブを回して、ドアに力をかけてドアを開け

なければならないので、順序に分類される。

分岐

家を出るとき、朝雨が降っているかどうか判断して降っていたら傘を持っていくだろうし、降っていなかったら傘を持っていかない。

これは条件によって行うことが変わっているので分岐の例である。

繰り返し

朝、歯を磨くとき、歯ブラシを1スライドさせただけで終わる人はいないだろう、歯ブラシをスライドさせることを繰り返して磨き、ある一定の基準を満たしたとき終了するのだ。(3分経ったか、全部の部位を磨いたか、等)

以上のような要素の組み合わせによって世の中のすべてのアルゴリズムは記述できる。

まずはソースコードを見てみる

ちなみに**ソースコード**とは、プログラマーが書くいわゆるプログラムのコードのこと。ソースともいう。

とりあえず、中身を理解する必要はないのだが、これから僕達が書いていくであろうソースコードはどのようなものになっていくのか見てみよう。

```C#
using System.Collections.Generic;

namespace MMF.Bone
{
    // ボーンのソート用クラス
    internal class BoneComparer : IComparer<PMXBone>
    {
        public BoneComparer(int boneCount)
        {
            BoneCount = boneCount;
        }

        public int BoneCount { get; private set; }

        // ボーンの計算順序順に並べ替える
        public int Compare(PMXBone x, PMXBone y)
        {
            /*後であればあるほどスコアが大きくなるように計算する*/
            int xScore = 0;
            int yScore = 0;
            if (x.PhysicsOrder == PhysicsOrder.After)
            {
                if(x.ignoreBone)
                {
                   return 0;
                }
                xScore += BoneCount*BoneCount;
            }
            if (y.PhysicsOrder == PhysicsOrder.After)
            {
                yScore += BoneCount*BoneCount;
            }
            xScore += BoneCount*x.Layer;
            yScore += BoneCount*y.Layer;
            xScore += x.BoneIndex;
            yScore += y.BoneIndex;
            return xScore - yScore;
        }
    }
}
```

プログラミングにまだ触れたことのない人にはよく分からない光景が広がっているかと思うが、**焦る必要は全くない。** C#のコードは大体こんな感じに書かれる。

まずは括弧の対を見つけ出そう

話は変わるが、中学の時、数学では一般的に小括弧、中括弧、大括弧と分けて計算していたけれども、多

分多くの人は高校に入るとすべて小括弧でも問題ないと気付くようになったんじゃないだろうか。

$$y = f[e\{c(ax + b) + d\}]$$

という数式があった時、実は以下のように表現しても問題ないと気がつくだろう。

$$y = f(e(c(ax + b) + d))$$

このように同じ種類の括弧の対が別の対の中に存在することを**ネスト**すると言う。また、特にC#では中括弧のある括弧から対応する括弧までの領域を**スコープ**という。

多くの種類の言語では(特にC#では)、括弧がネストしなければならないような状況は当たり前のように起きる。 まず、第一歩として、数ある括弧がどの括弧と対応しているかを見極められることが重要だ。

例えば、7行めの中括弧は39行めの中括弧で閉じているし、8行めの小括弧は同じく8行めの小括弧で閉じている。 C#の括弧は数学と違って他の種類の括弧とは意味が違うので、厳格に区別されているので注意が必要だ。

練習

- 4行めの括弧はどこで閉じているか?
- 38行めの括弧はどこで始まったか?

[文法]コメント

長い長いプログラムを書いていると、自分が何を書いているのかわからなくなったり、あとで見返したときに目も当てられない状況になる。 そんな状況になるのを避けるために、分かりやすいようにソースコードにはしばしばコメントが書かれる。

コメントは、プログラムには全く影響を与えない。

```C#
// 1行コメント
int a = 0; //左側はコメントじゃない
// int a = 0; コメントの中にプログラムを書いても動かないぞ!

/*好きな範囲をコメントにできる。
もちろん、複数行でも問題はない。*/
```

上の通り、コメントには二つのやり方がある。 `//` から始まる行末までをコメントとする方法と、 `/*` から次の `*/` までをコメントとする方法だ。

いくらプログラムをコメントの中に書いても動かないし、コメントに何を書こうが自由だ。

練習

- 一番最初のソースコードのコメント部分をすべてマークしてみよう。

C#の中括弧のネストの構造

1行めにusingという部分があるが、これについてはまたしばらく後に内容を説明するとして、それ以外の部分のネスト構造について見ていこう。

内容についてはそれぞれ後々説明していくので、まずはどんなものがあるかということをリストアップしていこう。

多くの場合で、一番外側の中括弧は `namespace XXXX` などと書かれている。これは名前空間と呼ばれるが、しばらく後で再登場して解説することになるだろう。 ただし、これが存在しない場合もあるので注意が必要だ。(特にUnityの場合だとしばしば省略される)

さらにその内側(名前空間が省略されている時は一番外)にあるのは、**型**と呼ばれる。

さらに、型は多くの場合中に中括弧のネストがある。これは**メンバ**と呼ばれる。

さらにその下には任意個の**制御文**と呼ばれるものがネストする。

つまり、

(名前空間) ⊃ 型 ⊃ メンバ ⊃ 制御文 ⊃ 制御文 ⊃ … ⊃ 制御文

それぞれが何であるかを解説するのはもちろん一気にはできないのだが、全体の構造としては常にこのような形になることを把握しておいてほしい。今、自分が書いているコードが、制御文の中なのか、メンバの中なのか把握できるようになると、今後混乱しにくく便利だ。

練習

- 一番最初のコードを、名前空間のスコープ、型のスコープ、メンバのスコープ、制御文のスコープに分けてみよう。

メンバの中の文法

UnityでC#スクリプトを追加すると大体こんな感じになる。

```C#
using UnityEngine;
using System.Collections;

public class NewBehaviourScript : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

準備

- 上のコードを前の練習と同じようにスコープで分けてみよう。

これから、しばらくの間はUpdateの中だけを見ていく。Updateの中でいろいろ試してその中で使える文法について確認していく。

また、このUpdateはメンバの中でも**メソッド**と分類される文法である。以下はメンバとして含めることができる文法の種類である。

- メソッド
- フィールド
- プロパティ
- イベント
- コンストラクタ/デストラクタ

それぞれ、英語で言えばSVOやSVOCのように記法が違うのでそれぞれこの後説明していくことになるが、**メソッドならばメンバであるが、メンバならばメソッドとは限らない。**

[機能]Debug.Log

Updateの中にDebug.Log(100)などと書いてみると、実行時に100とコンソールに出る。 プログラム中の計算結果などは、これを使って表示してわかりやすくプログラムすることができる。

[文法]文字列

```C#
Debug.Log("Hello");
```

としてみると、コンソールにHelloと表示される。

用語として、0個以上の文字から構成される文章を**文字列**という。 文字列は `"` (**ダブルクォーテーション**)

で囲わなければならない。

[文法]変数と型名

プログラミング中では様々なデータを保持しておく必要がある。例えば、一般的なRPGで言えば以下のようなものがあるだろう。

- キャラクターのHP
- 主人公の名前
- 現在のマップにおける座標

そして、それぞれ違うデータであることがわかるだろう。このようなデータを**変数**という。(プログラミングの世界では、文字列でも、文字でも変**数**である。)

例えば今の例で言えば上から、

- 非負の整数
- 文字列
- 小数二つ (2次元のマップならば)

このようなデータの種類の違いを**型**という。

多くの型が必要なプログラミング言語は、それ以上分割できない最もシンプルな型、**プリミティブ型**を持つ。

例えば、この例では、非負の整数はそれ以上分割できないプリミティブな型であると言えるし、小数二つは小数一つというプリミティブな型が2つ集まったものであると言える。

確認

- 音楽ゲームの楽譜の一音あたりのタイミングはどのような型で表されるだろうか?

以下はC#で用いる代表的なプリミティブ型である。

型名	表現するデータ
int	整数
float	小数
double	倍精度小数
bool	真偽値
char	文字
string	文字列

*真偽値とは、true/falseしか受け付けない型のことである。

$$Bool = \{true, false\}$$

そのようなデータが欲しい時以下のように記述することでデータが入るメモリを確保することができる。
このような記述を**変数を宣言する**という。

```C#
変数型 変数名;
```

つまり、以下のようにして利用する。

```C#
int hp;
string name;
float[] position; // この型の文法については後々行う
```

例えば、これらの変数は以下のようにしてデータを入れることができる。

```C#
hp = 100;
name = "hello";
```

つまり、以下のようにして入ったデータを確認できる。

```C#
Debug.Log(hp);
Debug.Log(name);
```

[文法]演算子1

概念の整理

一度、プログラミングから立ち戻って、今まで習ってきた数学を少し拡張しよう。

身近にある演算子といえば、以下のようなものを普段扱うだろう。

- ＋ 加算
- － 減算
- × 掛け算
- ÷ 割り算

ところで、演算子とはなんだろうか。

上のどの演算子も以下のような特徴を持つことがわかる。

2つのなんらかの入力としての数を、1つの出力としての数に変換する

例えば、$1+3$という表記は、

1. 入力としては1と3

2. 出力としては4

として、上の関係を満たすことがわかる。だけれど、これは型によっては成立しないかもしれないことに注意したい。

例えば、入力が'a'という文字と3という数字だったとしたら+がどういう結果を返すか定義はできていない。

以後、以下のような記号で便宜上の演算子の性質を示すとしよう。

$$+ : Int^2 \mapsto Int$$

これは、整数2つが+記号によって一つの整数になることを表す。

実は世の中には1つしか入力を受け取らない演算子も存在する。

$$\sqrt{2}$$

という数を見てみれば、2という数しか受け取っていない記号だということがわかる。同様に、-3などの$-$も一つしか記号を受け取っていない演算子だということができるだろう。

例えば、これを表すなら

$$- : Int \mapsto Int$$

と表現できる。

一般的に、1つの入力を受け取る演算子を1項演算子(単項演算子)、2つの入力を受け取る演算子を2項演算子、N個の入力を受け取る演算子をN項演算子という。

実際の文法(算術演算子)

C#に置いて、intやfloatに実際に利用できる演算子は以下のように定義されている。

- $+$ 加算
- $-$ 減算
- $*$ 乗算
- / 徐算
- $\%$ 割り算の余り

例えば以下のように利用できる。

```C#
Debug.Log(8 + 3);
Debug.Log(3 * 10);
```

変数と一緒に用いて以下のようなこともできる。

```C#
int hp = 100;
Debug.Log(hp * 100);
```

また、現実的な算術と同じように、演算子には優先順位がある。乗算と徐算と割り算の余りは、加算と減算よりも先に計算される。

```C#
int hp = 100;
Debug.Log(200 + 5 * hp -100); // 600
```

さらに、()を用いて優先順位をつけることが可能だ。**C#では常に小括弧であって、中括弧や大括弧は違う意味になってしまうので気をつけよう。**

```C#
int hp = 100;
Debug.Log((200+5)*hp - 100); // 20400
```

ところで=もまた演算子である。この計算順序は他の加算、減算、乗算、徐算、割り算の余りのどれよりも低い。 したがって以下のように書いた時、指定した計算でhpを更新することができる。

```C#
int hp = 100;
hp = hp * 200;
```

この例では、$hp = hp$よりも先に、$hp * 200$が評価され、その後代入されるので結果として20000となる。

練習

- int型3つの変数a,b,cがあったとしてその3辺からなる三角形の面積の二乗S^2を出す数式を考えよう。以下はヘロンの公式である。

$$s = \frac{a+b+c}{2}$$
$$S^2 = s(s-a)(s-b)(s-c)$$

概念の整理(関係演算子)、文法

さて、高校までに不等号>や<あるいは、等号=を習っただろう。これらも実は演算子なのである。

先ほど定義した真偽値は要するに、真か偽かという値のみが入る型であった。

至極当たり前であるが以下の式を見てみよう。

$$4 > 1(True)$$

$$4 = 2(False)$$

以上の通り、上の不等号や等号は2つの数字から真か偽かの値に変換する役割を持った演算子なのである。

先ほどの記号を用いれば以下のように記述できるだろう。

$$>: Int^2 \mapsto Bool$$

例えば以下のように利用できる。

```C#
Debug.Log(8 > 3); // True
```

また、前述の算術演算子と併用すれば以下のようなこともできる。

```C#
Debug.Log(3 * 2 < 2 + 3); //False
```

以下はC#における関係演算子のリストである。

- >もしくは< 不等号、見たとおりである
- >=、<= 以上、もしくは以下。方向は見たとおりである。
- == 等しい
- != 等しくない

特に、等しいと等しくないの表記が特殊であることに気をつけよう

実際に関係演算子が重要な役割を発するのはもう少し後、**制御文**を学んだ時なのだが、これをかけるようにしておこう。

概念の整理(論理演算子)、文法

hpが100よりも大きい、かつ500よりも小さい時。という判定をしたいとしよう。

C#では以下のように書くことができる。

```C#
hp > 100 && hp < 500
```

hpが100よりも小さい、または500よりも大きい時という判定をしたいとすれば以下のように書ける。

```C#
hp < 100 || hp >500
```

これらは、trueかfalseかの値を2つ受け取り、一つのtrueもしくはfalseに変換するので、先の記号を用いて以下のように表せる。

かつ、とまたはに関しては二つ記号が連続しなければならないことに気をつけよう。(単体だと全く意味が変わってしまう)

$$|| : Bool^2 \mapsto Bool$$

$$\&\& : Bool^2 \mapsto Bool$$

さらに、真と偽を反転する演算子も存在する。 例えば、「aよりもbの方が大きく、bよりもcの方が小さいときでない」というのを表すなら以下のように表せる。

```C#
!(a < b && b > c)
```

このように!で真か偽か反転できる。

制御文

制御文とは、アルゴリズムの分類として説明した分岐や繰り返しなど、普通は上から下に順番に実行されるプログラムのフローを制御しうるものである。 まずは、分岐を学んでみよう。

分岐1-A (if)

if文は以下のような文法で後続する中括弧の中の処理を実行するかしないか決めることができる。

```C#
if(真か偽)
{
    // 真であった時実行される処理
}
```

例えば、hpが100以上であった時は「HPは100以上です」と表示するならば以下のように表記することができる。

```C#
if(hp >= 100)
{
    Debug.Log("HPは100以上です。");
}
```

分岐1-B (else)

else文はif文に後続してifの中括弧の中に入らなかったものを処理する。

```C#
if(hp >=  100)
{
    Debug.Log("HPは100以上です");
}
else
{
    Debug.Log("HPは100未満です");
}
```

分岐1-C (else if)

さらに発展的な文法として、ifの中には入らなかった中で、特定の条件を満たす際に分岐することができる、if else句がある。

```C#
if(hp >= 100)
{
    Debug.Log("HPは100以上です。");
}
else if(hp >=50)
{
    Debug.Log("HPは50以上、100未満です。");
}
else
{
    Debug.Log("HPは50未満です。");
}
```

また、else if文はifとelseの間に複数個取ることができる。例えば、上記のコードにさらに25以上、50未満を判定したい時は以下のようなコードになる。

```C#
if(hp >= 100)
{
    Debug.Log("HPは100以上です。");
}
else if(hp >=50)
{
    Debug.Log("HPは50以上、100未満です。");
}
else if(hp >= 25)
{
    Debug.Log("HPは25以上、50未満です。");
}
else
{
    Debug.Log("HPは25未満です。");
}
```

練習

- 入力、a,b,cの数字があった時に、a,b,cの辺の長さから三角形が作れる場合は”三角形が作れる"と、作れない場合は"三角形が作れない"と表示するプログラムを作ってみよう。

繰り返し1(while)

もう一つのアルゴリズムの構成要素として、"繰り返し"がある。 この繰り返しは以下のように記述することができる。

```C#
while(真か偽)
{
    // 真の間繰り返したい処理
}
```

例えば、0から99までの数を表示したいなら以下のように記述することができる。

```C#
int index = 0;
while(index < 100)
{
    Debug.Log(index);
    index = index + 1;
}
```

ちなみに、途中で特定の条件の時whileを抜けたい場合は**break**を用いる。

```C#
int index = 0;
while(index < 100)
{
    Debug.Log(index);
    index = index + 1;
    if(index % 3 == 0)
    {
        break;
    }
}
```

上のコードはindexが初めて3の倍数になった時whileを抜ける。つまり、0だけ表示して終わる。

理論上は、今のifやwhileを用いれば存在するすべてのアルゴリズムを組むことができる。

練習

- 0から100の数字まで表示する中で、3の倍数ならFizz,5の倍数ならBuzz、それでもなければその数字、15の倍数ならばFizzBuzzと表示するプログラムを書いてみよう(FizzBuzzプログラムという入門用に有名なプログラムだ)

繰り返し2(for)

実は他にも制御文は存在する。理論的にはifとwhileだけですべてのアルゴリズムを記述することができるが、もっと便利に短く特定の場合にかける場合がある。 特に、指定回数を繰り返すというような場合は**for**を使うと良い。 例えば以下のように書けば、0から99までの数を表示することができる。

```C#
for(int index = 0; index < 100; index = index + 1)
{
    Debug.Log(index);
}
```

今までの他の制御文と比べると複雑に見えるが、forの()の中は;で3つに区切られ、以下のようにそれぞれ意味を持つ。

- 左側 一番最初に実行する処理。　上の例ではindexという変数を0とする。

- 真ん中 真か偽か判定する処理。これが真の時繰り返す。 上の例ではindexが100よりも小さい時。
- 右側 毎回中括弧の終わりまで達した時に実行する処理。上の例ではindexに1を足す。

また、**for文の中でも、whileと同様にbreakを利用することができる。**

変数の使用可能な範囲

スコープの中で宣言した変数は、基本的にはその中でだけ使用することができる。プログラミングにおいてソースコードを記述する際には、変数の使用可能な範囲を意識しよう。

- 例1

```C#
using UnityEngine;
using System.Collections;

public class ClassName : MonoBehaviour {

    // Use this for initialization
    void Start () {
    int x;
    x = 10;
    }

    // Update is called once per frame
    void Update () {
      Debug.log("x =" + x); //Error
    }
}
```

- 例2

```C#
    for(int i = 0; i < 10; i++){
    Debug.Log(i);
    }
    //Debug.Log(i); //Error
```

この `for` 文の中で宣言された `i` はfor文のスコープの中でしか使えない。合わせて覚えておこう。

練習

- 実際にエラーが出ることを確かめよう。

[Tips]宣言の方法

まず下記のソースコードを見てほしい。

```C#
void Start () {
    int x;
    x = 10;
    }
```

前回までは上記のような方法でint型変数xを定めた。これを略して、以下のように宣言することができる。

```C#
void Start () {
    int x = 10;
    }
```

このように、変数の宣言と同時に値を入れることを変数の初期化という。これからは `int x = 10;` のような記述をしていくことにする。

また、 `x = x + 1;` のような形を略して `x += 1;` と表すこともできる。 加える増分が1の場合に限っては `x++;` と記述することもできる。この `x++;` については後ほど説明を加えると思います。

```C#
void Start () {
    int x = 10;
    x += 3;
    Debug.Log("x = "+ x); //x = 13
    }
```

[文法]配列

配列とはプログラミングにおけるデータ構造の一つ。複数の値をひとまとめにして扱いたい場合がある。前回学んだ、C#で用いる代表的なプリミティブ型でも配列を使用できる。変数の型と[]を使って、任意の配列を作成することができる。

型	表現するデータ
int[]	整数の配列
float[]	小数の配列
double[]	倍精度小数の配列
bool[]	真偽値の配列
char[]	文字の配列
string[]	文字列の配列

ではまず配列の宣言の仕方を学ぼう。

```C#
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {
    float[] array = new float[3]; //３つのfloat値が入る配列
    array[0] = 1.2; //配列は0からはじまる
    array[1] = 2.3;
    array[2] = -0.5;
    }
}
```

[注意]C#の配列は0番目から数えられる。つまり上記の例では0番目に1.2、1番目に2.3、3番目に-0.5が入ることになる。

- 例

```C#
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {
    float[] array = new float[20]; //20のfloat値が入る配列
    for(int i = 0; i < array.length; i++){
        array[i] = i * 2; //0,2,4,...
    }
    }
}
```

このように `for` 文を使って、配列に値を代入することができる。よく使うのでこれも合わせて覚えておこう。

練習

- 25個の要素を入れることのできる配列を宣言しよう。
- 作った配列に100以下の素数を順番に入れるアルゴリズムを考えてみよう。

素数:正の約数が 1 と自分自身のみである自然数で、1 でない数のこと。

ジャグ配列

配列の中に配列を入れることもできる。これをC#ではジャグ配列という。

- 例1

```C#
int[][] jaggedArray = new int[5][];
for(int i=0; i<jaggedArray.Length; i++ ){
    table[i] = new int[3];
}
```

5個の配列に３個の要素を持つことのできる配列を入れた。

- 例2

```C#
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

ジャグ配列の要素には、上記のように大きさの異なる配列を入れることもできる。

多次元配列

活動では今のところとばします。後日資料を用意しておきますのでしばらくお待ち下さい。知りたい人は、暇なときに調べてもいいです。

[文法]メソッド

プログラミングでも数学の関数のようなものを定義することができる。その関数のようなものをC#ではメソッドという。ソースコードを見やすくしたり、ソースコードの再利用性を高めたりすることができるので、使えると便利である。では、以下で使い方を見ていこう。

```C#
using UnityEngine;
using System.Collections;

public class ClassName : MonoBehaviour {

    // Use this for initialization
    void Start () {
    int a = 10;
    int b = 20;
    Debug.log(plus(a, b)); //30
    }

    public static int plus(int x, int y){ //メソッド
      //plusの部分には自由に名前をつけられる
      return x + y;
    }
}
```

これは、二つのint型の数字を受け取って、足したint型の値を返すメソッドの例である。数学の関数もこのように定義できる。プログラミングでは、メソッドが受け取る値の事を**引数(ひきすう)**といい、返ってくる結果を**返り値 or 戻り値**という。

結果は `return value;` で値を返す。返り値の型はメソッドを定義するときに、 `返り値の型　メソッドの名前(引数){メソッドの内容}` で書く。
メソッドは前回行った

(名前空間) ⊃ 型 ⊃ メンバ ⊃ 制御文 ⊃ 制御文 ⊃ … ⊃ 制御文

のメンバの部分にあたる。 メソッドならばメンバである。

Unityの `Start()` や `Update()` もメソッドの一つだ。

```C#
using UnityEngine;
using System.Collections;

public class ClassName : MonoBehaviour {
    // Use this for initialization
    void Start () {
    float x = 10f;
    flaot y = 20f;
    float z = 5f;
    Debug.log(x + (x + y) * x - (3 * x) * 3);
    Debug.log(y + (y + z) * y - (3 * y) * 3);
    Debug.log(z + (z + x) * z - (3 * y) * 3);

    }
}
```

これは、単に同じ計算を値を変えて行っているだけだが、以下のようにこの計算をするという同じ動作を取り出して、まとめることだ出来る。以下のような方法を用いることがある。

```C#
using UnityEngine;
using System.Collections;

public class ClassName : MonoBehaviour {
    // Use this for initialization
    void Start () {
    float x = 10f;
    float y = 20f;
    float z = 5f;
    Debug.log(mathodName(x, y));
    Debug.log(methodName(y, z));
    Debug.log(methodName(z, x));
    }

    public static float methodName(float a, float b){
      return a + (a + b) * a - (3.0 * a) * 3.0;
    }
}
```

まとめた場合、式を扱う部分が一箇所だけになっている。つまり、動作の内容（この場合は計算式）が変わっても変更を反映することが容易になる。

練習 実際にメソッドを一つ作ってみよう。

$$f(x) = 3x + 1 \quad (x : float型)$$

を表すメソッドを実装してみよう。実装できたら、実際にメソッドを `Start()` の中で使ってみよう。メソッドの名前は自由に決めて良いです。

クラス

一度、今まで学んだ文法を、最初に記述した中括弧のスコープ別に考えてみよう。

スコープ	今まで学んだ	これから学ぶ
名前空間	なし	名前空間
型	なし	クラス、構造体、インターフェース、列挙体、デリゲート
メンバ	メソッド	コンストラクタ、デストラクタ、フィールド、プロパティ、イベント
制御文	if、while、for、switch	foreach、try-catch、using
(メンバ内)	変数、演算子、new等	-

さて、今までC#に元々用意されているプリミティブ型やその配列を使えるようにはなったが、自分で型を作ることができると、様々なプログラムが楽になることがある。

[文法]フィールド

例えば、あるゲームのモンスターは以下のような値を持つ。

- string 名前
- int hp
- int 攻撃力

もしもこれを自分で型を作らずにモンスター3体がプログラムの中に登場したら記述が以下のようになってしまうだろう。

```C#
string name1 = "スライム", name2="メタルスライム",name3="はぐれスライム";
int hp1 = 30, hp2 = 1000, hp3 = 100;
int ap1 = 10, ap2 = 5, ap3 = 30;
```

どれがどれだかわからなくなるだろう。そういう時、モンスターごとに値を管理できたとしたら便利だ。

例えば以下のように記述することで新たにMonster型を作成することができる。このように宣言された場合、 `Monsterクラス` と呼び示すことが多い。

```C#
public class Monster
{
    public string name;

    public int hp;

    public int ap;
}
```

利用するには以下のようにすれば良い。

```C#
Monster m1 = new Monster();
m1.name="スライム";
m1.hp = 30;
m1.ap = 10;
```

上を見ればわかる通り、Monsterという名の**型**を追加したに過ぎず、今までの変数の宣言となんら変わらない変数の確保方法であることに注意したい。

以下はクラス型の宣言の仕方である。

```C#
(型アクセス修飾子) class クラス名
{
    (メンバアクセス修飾子) 変数型 変数名; // フィールド
}
```

また、上記の例で言えば、hpやnameやapなど型直下の変数のことを**フィールド**という。

型アクセス修飾子、メンバアクセス修飾子については後々扱うが、現時点ではpublicとしておいて問題はないだろう。

また、C#では基本的に**クラス名は大文字から始まり、単語の区切りで大文字にすることが多い**例えば、ボスのモンスターならば `BossMonster` などの名前が適切だろう。

もちろん、これはただの名前であり、自由でつけられるが他の人が読みやすいように名前には一定の規則がつけられてることが多い。これを変数の**命名規則**という。

あるクラス内のフィールドにアクセスするには、クラスの変数に `.` をつけてアクセスすることができる。

先の例のように、Monster型の変数m1の中のnameフィールドにアクセスするには、 `m1.name` とすれば良い。

[文法]コンストラクタ

さて、以下の先ほどのソースコードを見てみると、まだまだ冗長である。

```C#
Monster m1 = new Monster();
m1.name="スライム";
m1.hp = 30;
m1.ap = 10;
```

例えば、name,hp,apがMonster型の変数を作成するときに代入されることが前提なら、以下のように記述するとさらに便利になる。

```C#
public class Monster
{
    public string name;

    public int hp;

    public int ap;

    public Monster(string _name,int _hp,int _ap)
    {
        this.name = _name;
        this.hp = _hp;
        this.ap = _ap;
    }
}
```

これを使うときには以下のような記述で先ほどと同じような変数m1を用意することができる。

```C#
Monster m1 = new Monster("スライム",30,10);
```

まず、新しいMonster型のプログラムを見てみると9行めからMonsterという名前のようなメソッドのようなものができている。 **ただし、メソッドとは違い、返り値型が記述されていないことに気をつけよう**

つまり、コンストラクタは以下のような文法で定義できる。

```C#
(型アクセス修飾子) class クラス名
{
    (メンバアクセス修飾子) クラス名(引数型1 引数名1,引数型2 引数名2,....,引数型N 引数名N)
    {
        // コンストラクタの中身
    }
}
```

このようなメンバを**コンストラクタ**と言い、そのクラスがnewされるときに呼ばれることになる。

`this` はこの型の変数自体を指す。例えば、このコンストラクタの中のthisはそれを呼び出してnewしている部分のm1と等しい。

コンストラクタはメソッドと同様に**オーバーロードすることができる**

例えば、以下のようにすると、名前しか指定しない場合は、自動でhpとapを0にするコンストラクタも同時に定義できる。

```C#
public class Monster
{
    public string name;

    public int hp;

    public int ap;

    public Monster(string _name,int _hp,int _ap)
    {
        this.name = _name;
        this.hp = _hp;
        this.ap = _ap;
    }

    public Monster(string _name)
    {
        this.name = _name;
        this.hp = this.ap = 0;
    }
}
```

上の例では、2番目の正規表現の内容を以下のように書くことでさらに省略できる。

```C#
public class Monster
{
    public string name;

    public int hp;

    public int ap;

    public Monster(string _name,int _hp,int _ap)
    {
        this.name = _name;
        this.hp = _hp;
        this.ap = _ap;
    }

    public Monster(string _name)
    {
        this(_name,0,0);
    }
}
```

コンストラクタ内の `this()` は別のコンストラクタの呼び出しを示す。

[文法]メソッド(その2)

例えば、このモンスターはhpが1減ると、apが1ずつ増えるようなモンスターだったとしよう。

このモンスターにダメージを与えるという処理は以下のように書くことができるだろう。

```C#
int damage; //何らかの値
Monster m1 = new Mosnter("スライム",30,10);
m1.hp = m1.hp - damage;
m1.ap = m1.ap + damage;
```

しかし、プログラムのあらゆるところで、ダメージを与えるという処理が増えていくと、時に攻撃力をプラスするという処理を忘れてしまうかもしれない。

ここで、一つ考え方を抽象化しよう。

今までの考え方

- モンスターにダメージを与えるときはhpからdamageを引く
- モンスターにダメージを与えるときはapにdamageを足す。

抽象化された考え方

- モンスターがダメージを受けるという処理にdamageを引数として与える。

とにかく、以下のように書いてみよう。

```C#
public class Monster
{
    public string name;

    public int hp;

    public int ap;

    public Monster(string _name,int _hp,int _ap)
    {
        this.name = _name;
        this.hp = _hp;
        this.ap = _ap;
    }

    public void Damage(int damage)
    {
        this.hp = this.hp - damage;
        this.ap = this.ap + damage;
    }
}
```

その上で、以下のようにこの場合の例は記述できる。

```C#
int damage; //何らかの値
Monster m1 = new Mosnter("スライム",30,10);
m1.Damage(damage);
```

こうすれば、仮に別の人がMonsterを作ったとしても、自分はその振る舞いについて理解せずともダメージを与えることができるようになる。 これがメソッドの本質である。

[文法]静的フィールド

例えば、 `Cat` クラスを作ったとしよう。それぞれの猫は独立した名前 `name` をもつ。 同時に、猫の足の本数が入っているフィールド、 `legCount` があったとしよう。 今までの知識から `Cat` クラスを作るとするならば、以下のようになるだろう。

```C#
public class Cat
{
    public string name;

    public int legCount = 4;

    public Cat(string name)
    {
        this.name = name;
    }
}
```

例えば使うときには以下のようにする。

```C#
Cat mike = new Cat("ミケ");
Cat shure = new Cat("シュレディンガー");
Debug.Log(mike.legCount);
Debug.Log(shure.legCount);
```

しかし、ここで当たり前のことだが、基本的には**猫の足の本数は4本**である。 この状況では、 `Cat.legCount` と記述できた方が分かりやすいだろう。

そういう時は、以下のようにCatを書き換える。

```C#
public class Cat
{
    public string name;

    public static int legCount = 4;

    public Cat(string name)
    {
        this.name = name;
    }
}
```

こうすることで、legCountフィールドは `Cat.legCount` によりアクセス可能となる。このようなフィールドを静的フィールドやstaticフィールドと呼ぶ。

[文法]静的メソッド

今度は、 `Cat` クラスに、その泣き声を出力するメソッドを作ったとしよう。おそらく以下のようになるだろう。

```C#
public class Cat
{
    public string name;

    public void Say(){
        Debug.Log("にゃー");
    }

    public Cat(string name)
    {
        this.name = name;
    }
}
```

そして使う時は以下のようになる。

```C#
Cat mike = new Cat("ミケ");
Cat shure = new Cat("シュレディンガー");
mike.Say();
sure.Say();
```

しかし、これはフィールドの件と同様に、**すべての猫は「にゃー」と鳴く**ので、`Cat.Say()`と言えた方が適切である。

そういう時は同様にstaticをつける。

```C#
public class Cat
{
    public string name;

    public static void Say(){
        Debug.Log("にゃー");
    }

    public Cat(string name)
    {
        this.name = name;
    }
}
```

これを**静的メソッド**や**staticメソッド**という。静的メソッドには以下のような制約がある。

staticでないメンバを呼び出すことはできない。

これは、staticの仕組みから考えれば至極当然のことである。例えば、以下のようなコードが成立したとしよう。

実際には以下のコードはエラーとなる

```C#
public class Human
{
    public string name;

    public Human(string name)
    {
        this.name = name;
    }

    public static void SayName()
    {
        Debug.Log(this.name);
    }
}
```

```C#
Human taro = new Human("太郎");
Human ziro = new Human("次郎");
Human.SayName();
```

これが成立したとき、staticとは人間全体に共有で使えることを前提としているのに、nameに関してはそれぞれの人間に依存する。 そのため、SayName()の中のthis.nameが誰を指しているか不明瞭である。

よってこのような使い方ができない。

しかし、逆にstaticでないメソッドからstaticなフィールドやstaticなメソッドを呼ぶことは可能である。

参照と値

メモリとアドレス

今まで、変数というものを何も意識せず、ただ単にデータが入る箱としてだけ見てきたが、実際にメモリの中でどう格納されているか考えてみよう。

今までメソッドを作る中で、別のメソッドの中に同じ変数名があったとしても問題がないことはわかったはずだ。 メモリの中で変数が混ざらないようにするには、一意の名前を振り分けておく必要がある。

そこで、コンピューターのメモリは1byteごとに連番で数字がつけられている。

その上で、「aという変数は2100byte目に格納されている」というような形で保持されているのだ。

この例はメモリを大きなマンションに例えてみると分かりやすいかもしれない。

4byte利用するint型なAさんの場合

あるソースコードの中で、int A;というような表現があった時、コンピューターは、自分のメモリの中から

4byte連続で空いている部分を見つけ出す。

例えるなら、マンションの中から4部屋連続で空いている部分を見つけ出すわけだ。その上で、Aさんが利用する部屋番号のうち一番若い者をAの所在地として記憶する。Aさんが必要なのは4byteなので、Aさんが占める部屋はその部屋番号+3までだということがわかる。

部屋番号の長さ

例えば、このマンションの部屋番号が10進数で4桁までだったとしよう。 その時は、0号室から9999号室までしかないから、1つ1byteだと10000byteしか確保できないことになる。

実際のコンピューターにおける部屋番号(**メモリアドレス**)は2進数で管理されていて、現代ではほとんどが32bitか、64bitの長さのメモリアドレスが使われている。

Windows64bitだとか、Windows32bitだとか聞いたことないだろうか。これは、そのOSが使うメモリアドレスの長さを表しているのである。

ここからは完全に余談なのだが、32bit、64bitで使える最大のメモリの大きさを考えてみよう。

$$2^{32}byte = 2^{22}KB = 2^{12}MB = 2^{2}GB = 4GB$$

一方、64bitあると、以下のようになる。

$$2^{64}byte = 2^{54}KB = 2^{44}MB = 2^{34}GB = 2^{24}TB = 2^{14}PB = 2^{4}EB = 16EB$$

となって、64bitOSでは現代では使い切れないほどのメモリを搭載することが理論的には可能になっている。

そして、32bitOSから64bitOSに移行が進んだ背景として、PCのメモリが大容量化したりアプリケーションが大きなメモリを要求するようになってきて、最大値が4GBという制限に限界がきたからである。

メソッドの値渡しと参照渡し

int型変数の入れ替えメソッド

あるプログラムの中で、二つの変数を入れ替えることが頻繁に起きたとしよう。そんな時、前に出てきた**メソッド**の文法を用いて、以下のように記述すれば二つの変数を入れ替えるメソッド `Swap` を作成することができると思うかもしれない。

```C#
void Swap(int a,int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

実験

- 実際に上のプログラムを記述し、3行めにブレークポイントを設置し、6行めまで順次実行し入れ替わるか試そう。
- Swap関数を実際に呼んで、与えた変数が入れ替わったか試そう。

int型の配列の入れ替えメソッド

今度は、2つの配列の中身を入れ替えることが頻繁に起きたとしよう。そんな時、以下の2つは配列の入れ替えを行うためのメソッド `SwapArray1` と `SwapArray2` である。両方とも入れ替えそうに見えるがどうだろうか。

```C#
void SwapArray1(int[] a, int[] b)
{
   int[] temp = a;
   a = b;
   b = temp;
}

void SwapArray2(int[] a,int[] b)
{
   int[] temp = new int[a.length];
   for(int i = 0; i < a.length; i++)
   {
      temp[i] = a[i];
      a[i] = b[i];
      b[i] = temp[i];
   }
}
```

実験

- それぞれ前の練習のように入れ替わるか試してみよう

メモリで何が起こっているか

例えば、以下のコードを考えてみよう。

```C#
void Update()
{
   int a1 = 10;
   int b1 = 20;
   Swap(a1,b1);
}
```

1. 3行めでコンピューターはa1という名前でメモリ上のどこかに4byteのメモリ領域を確保する。
2. 4行めでコンピューターはb1という名前でメモリ上のどこかに4byteのメモリ領域を確保する。

3. Swapに渡す時、Swapに入る瞬間にa,bという名前でそれぞれ4byteのメモリ領域が確保され、a1,b1の**中身が**コピーされる。
4. Swapの3行めでtempという名前でさらに4byteのメモリ領域を確保する。
5. aとbの内容が入れ替わる
6. Swapを通った後のa1,b1は何も変わっていない。

ここからがややこしいことなのだが、実は3番めのように中身がコピーされるのは渡そうとしている変数型が**値型**と呼ばれるものであった時のみである。

一方で、SwapArray2について見てみよう。

```C#
void Update()
{
    int[] a1 = new int[1000000];
    int[] b1 = new int[1000000];
    // a1,b1をここで何らかの初期化をしたとする。
    SwapArray2(a1,b1);
}
```

もしも、Swapのようなことが起きたら実は都合が悪いことになる。 渡そうとしているa1,b1はものすごく大きいデータであったとしよう。この場合、4000000byte = 4MBのメモリを一つの変数あたり使うことになる。

もし、 上のSwapArray2ぐらいであったらいいのだが、呼び出したメソッドがさらに別のメソッドを、その中身がさらに別のメソッドを配列を引数に呼び出していたとしたら、ものすごい量のメモリを使ってしまうことになる。

例えば、この例で仮に `int[]` が値型であったとして、SwapArray2を呼び出すとしたらこの呼び出しまで全体で必要な理論メモリ容量は20MBになる。

だが、実際にはそうはならない。

ここで以下のようなソースコードを考えたとしよう。

```C#
int[] a;
a = new int[100];
int b = a[22];
```

それぞれの行でどれだけのメモリが実際に使われるだろうか?

ここで、おそらく1行目がどうなるのか答えるのが難しいだろう。

そこで、とりあえず2行目について考えてみる。2行目に達した時、コンピューターは以下のように振る舞う。

1. メモリの中から連続して4 * 100byte分連続して空いている部分を見つけ確保する。
2. その先頭の要素のメモリアドレスをaに入れる。

つまり、**aに入ってるのは実際にint型の複数の要素ではなく、複数の要素がある場所へのメモリアドレスである**ということだ。

aが実際にはメモリアドレスだったとしたら理論上の容量はOSによって異なる。32bitOSならば32bit = 4byte、64bitOSならば64bit = 8byteが1行めで確保される事になる。

さらに3行目では以下のような事がコンピューターの中で起こる。

1. メモリ中で連続して4byte分空いているところを見つけ確保する。(変数b)
2. aに入っているメモリアドレス + 22 * 4byte の部分から始まる4byte分を読み取りbにコピーする。

ここで、SwapArray2の例に戻ろう。

```C#
void Update()
{
    int[] a1 = new int[100];
    int[] a2 = new int[100];
    SwapArray2(a1,b1);
}
```

これを実行する環境が64bitOS環境だったとしよう。

1. 3行目でメモリアドレスを入れる8byte分のa1と、int100個分である400byteが確保される。
2. 4行目でメモリアドレスを入れる8byte分のb1と、int100子分である400byteが確保される。
3. SwapArray2に渡す時、メモリアドレスが入るそれぞれ8byteのaとbが確保され、a1とb1の中の**メモリアドレス**がコピーされる。
4. SwapArray2の中の10行めから16行めまでの間で、aとbのメモリアドレスが指し示す先の内容を入れ替えている。
5. aとbのメモリアドレスの指し示す先とa1,b1が指し示す先は一緒だったからこれで変更されている。

特に、このように**メモリアドレス**で管理される型を**参照型**という。今まで出てきた中で参照型は配列だけなのであるが、 今後たくさん出てくるので注意しよう。

練習

- なぜSwapArray1は動作しなかったか?