オブジェクト指向とは

プログラムコードは、**データ(変数など)**と**手続き(関数など)**に分類できる。規模が小さいプログラムの場合は、図 1 規模が小さいプログラムのイメージのように、一つのプログラムに全てのデータと手続きをまとめても構わない。

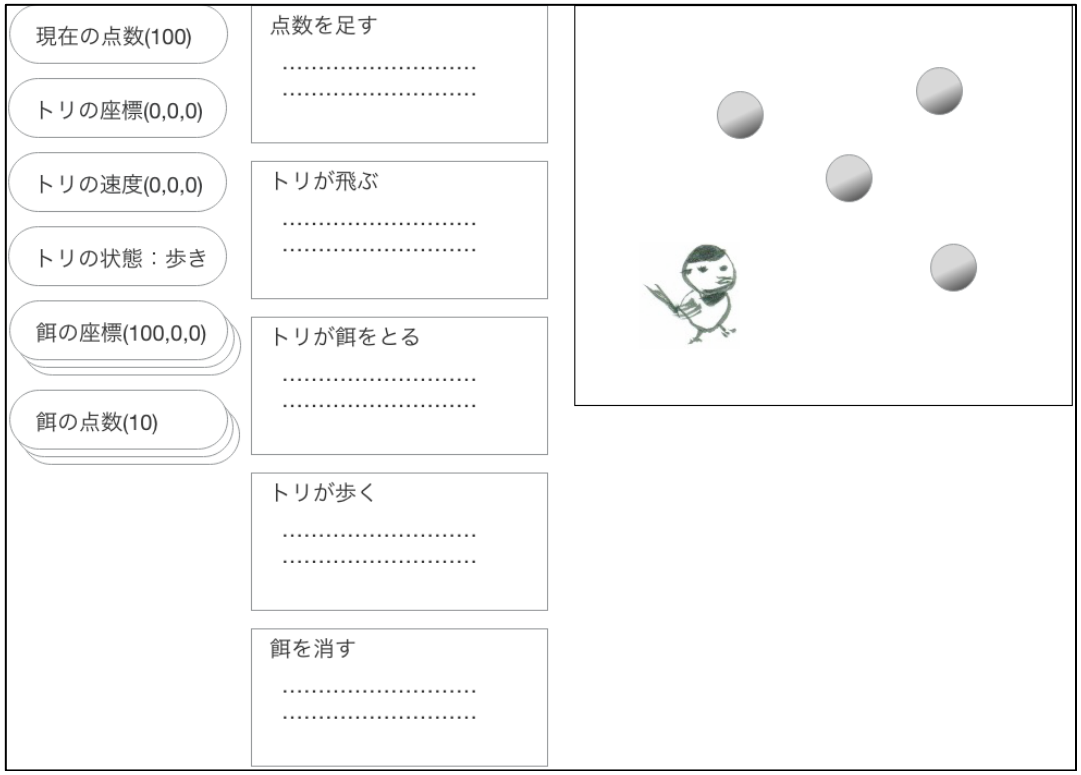

図 1 規模が小さいプログラムのイメージ

しかし、規模が大きくなり、扱うものが増えてくると、データや手続きの数が膨大になり、このやり方では限界が出てくる。

◆問題点

- ◆ どこに変数や関数があるのか分からない
- ◆ 変数名や関数名の管理が大変
- ◆ 再利用が難しい
- ◆ etc...

これらの問題を解決するために考案されたのが**オブジェクト指向プログラミング**である。**オブジェクト（もの）**を定義して、そのオブジェクトに関連するデータと手続きをまとめて実装することで、プログラムの肥大化を防ぎ、データや手続きが実装されている場所を推測しやすくするのである(図 2)。

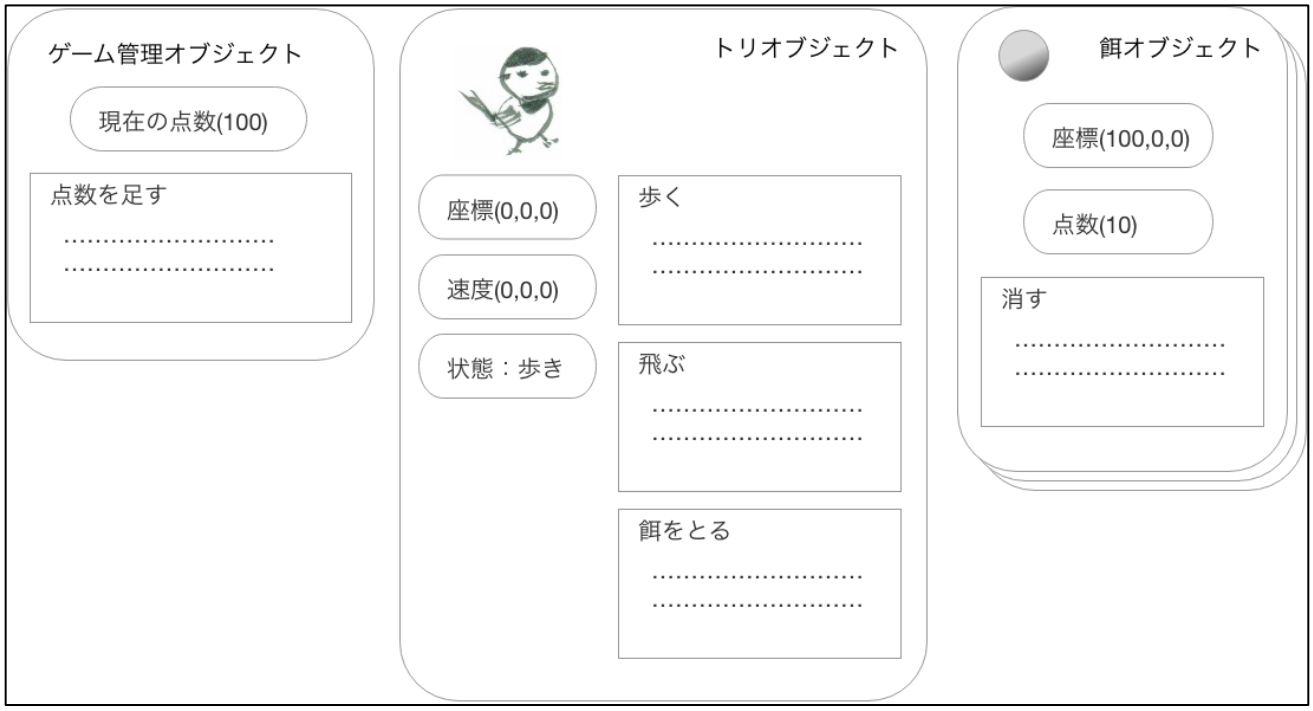

図 2 オブジェクト指向プログラミングのイメージ

オブジェクトごとに整理することで、データや手続きを分かり易くグループに分けることができる。また、オブジェクト指向言語には、プログラムの再利用性を高める**継承**機能や、複雑さを減らすための**ポリモーフィズム**という機能を持っている。

オブジェクト指向プログラミングのメリットには以下のようなものが挙げられる。

- ✓ データや機能の所在が分かりやすくなる
- ✓ ファイルを適度に分けて、プログラムの見通しが良くなる
- ✓ 内部データや手続きを隠して、プログラムをシンプルにできる
- ✓ オブジェクトを拡張して再利用できる
- ✓ 同じ親を持つオブジェクト同士で、記録する配列や命令の呼び出しを共有できる

一方、慣れないと不便に感じることもある。

- ✓ オブジェクト指向プログラミングについて新たに勉強する必要がある
- ✓ プログラムが分割されるので、一箇所を見るだけでは全体像が分からない
- ✓ 再利用しないプログラムでは、手間が増えることに対する恩恵が少ない

オブジェクト指向の考え方を使って設計すれば、C 言語などのオブジェクト指向言語ではないものでも同様の実装は可能である。しっかりと理解をして、様々なところで応用できるようにしよう。

習作の概要

オブジェクト指向プログラミングを学習するために、**人**、**鳥**、**魚**の自己紹介と、行動を表現するプログラムを作成する(図 3)。

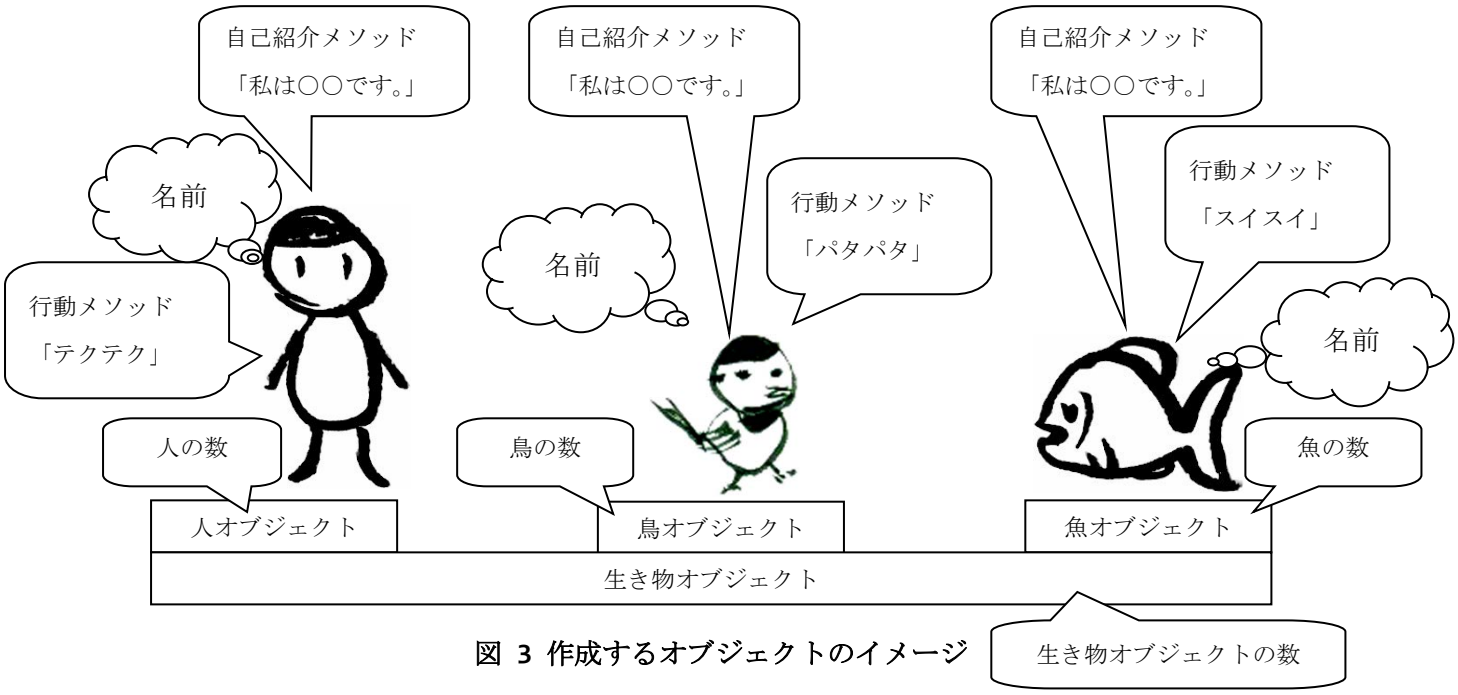

図 3 作成するオブジェクトのイメージ

Visual C#で図 4 のようなフォームを作成して、機能を実装する。

- テキストボックス txtIndex
  - 操作するキャラをインデックスで指定
- テキストボックス textBox1
  - 登録されているキャラリスト
  - 行頭の数字がインデックス
  - ボタンを押した結果も、ここに表示
- ボタン buttonHello
  - txtIndex のキャラの挨拶を表示
- ボタン buttonAction
  - txtIndex のキャラの行動を表示
- ボタン buttonNumber
  - 現在のキャラの数を表示

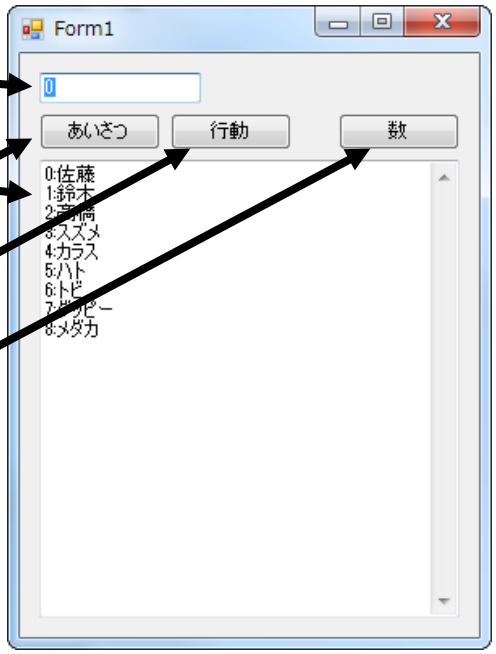

図 4 作成するプログラムのフォーム

オブジェクトを使わずにベタに実装した例を以下で公開している。プロジェクトをダウンロードして、開いてみよう。

- プログラムコード
  - https://gist.github.com/tanakaedu/a55477056822950e6ab59f16efc83dbb
- プロジェクトの ZIP ファイル
  - https://github.com/tanakaedu/object-practice-2016/archive/no-object.zip

この程度の規模であればこの手法でも問題はない。一箇所にプログラムがまとまっているので、初心者には分かり易いプログラムだろう。しかし、規模が大きくなり、変数や関数、キャラの種類が増えると、実装されている場所を見つけるのが大変になったり、ボタンの中の switch 文が膨大になってメンテナンスが難しくなったりすることが予想される。また、多人数で開発する時に、同じファイルを編集するのはコンフリクトの元になる。

オブジェクト指向言語向けに設計し直すことで、これらの問題がどのように解決するかを確認してみよう。

オブジェクトの設計

問題の一つは、**変数や関数が、何のものなのかが分からない**ことである。オブジェクト指向では、その「**何**」にあたるものを**オブジェクト**として定義して、そこにデータ（プロパティー）や手続き（メソッド）を属させて整理する。オブジェクトでは、変数にあたるものを**プロパティー**、関数にあたるものを**メソッド**と呼ぶので覚えておこう。

今回登場するのは、**人オブジェクト**、**鳥オブジェクト**、**魚オブジェクト**である。これに加えて、プログラム全体を管理するための **Form1 オブジェクト**を引き続き利用する。オブジェクトを導入することで、変数と関数がどのようになるかを図 5 に示す。

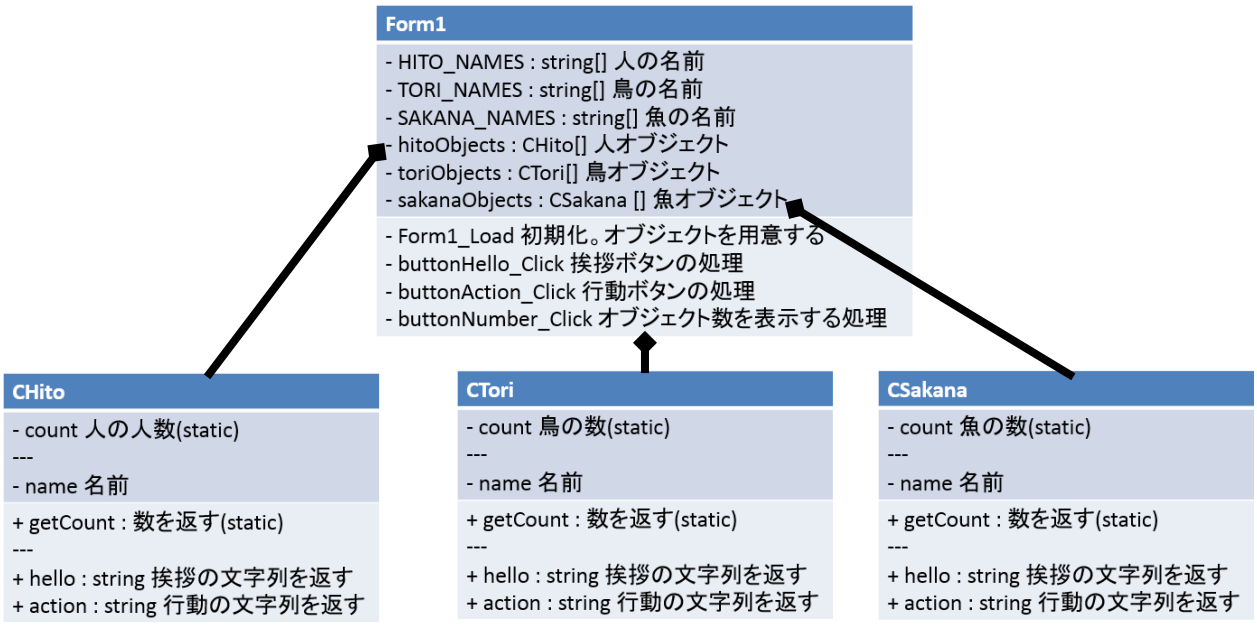

図 5 オブジェクト設計　その 1

プログラムが分割されて、プロパティーやメソッドが何に関連するのかが明確になった。まずは、この状態で実装してみよう。

オブジェクト第１段階～クラスの作成

オブジェクト指向プログラムとしては中途半端な状態だが、理解を深めるためにこの段階でプログラム化してみよう。Visual C#でクラスを作成して、図 5 で設計したプログラムに改造する。

Form1 を整理

クラスの実装に先立って Form1.cs を整理する。コードの下の方から、以下のコードを削除する。

1. 118～121 行目　　キャラ数を表示するコードを削除
2. 98～113 行目　　行動ボタンの処理を削除
3. 89～93 行目　　挨拶ボタンの処理を削除
4. 51～84 行目　　フォーム起動時の処理を削除
5. 22～43 行目　　不要な列挙定義や、プロパティーを削除

実行すると、テキストボックスへの表示や、ボタンの機能がなくなる。ここにクラスを追加して機能を復活させていく。以下の URL で削除済みのコードを確認できる。

https://gist.github.com/tanakaedu/f843c5744c770ed466824a62018a8821

クラスとは

クラスは、C# や Java などでオブジェクトを定義するために利用するものである [1]。「設計図」とか「遺伝子」のようなものと説明される場合が多い。C#や Java などでは図 6 のように記述する。

```
class クラス名 {
  // クラスの持つデータ（変数や定数）や、手続き（関数）を記述する。
}
```

図 6 C#でのクラスの宣言

Visual C#で、ツールバーに並んでいるものもクラスである。 .NET Framework によって様々なクラスが提供され、簡単に高度なプログラムを構築できるようになっている。

[1] 古い JavaScript など、クラスを使わずに直接オブジェクトを定義するような言語もある

人クラスを作成

人クラスを作成してみよう。先に紹介したクラスなしの Visual C# プロジェクトから、作業を開始する。

1. [ソリューションエクスプローラー]からプロジェクト名(1)の部分を右クリックして、[追加](2)、[クラス](3)を選択(図 7)

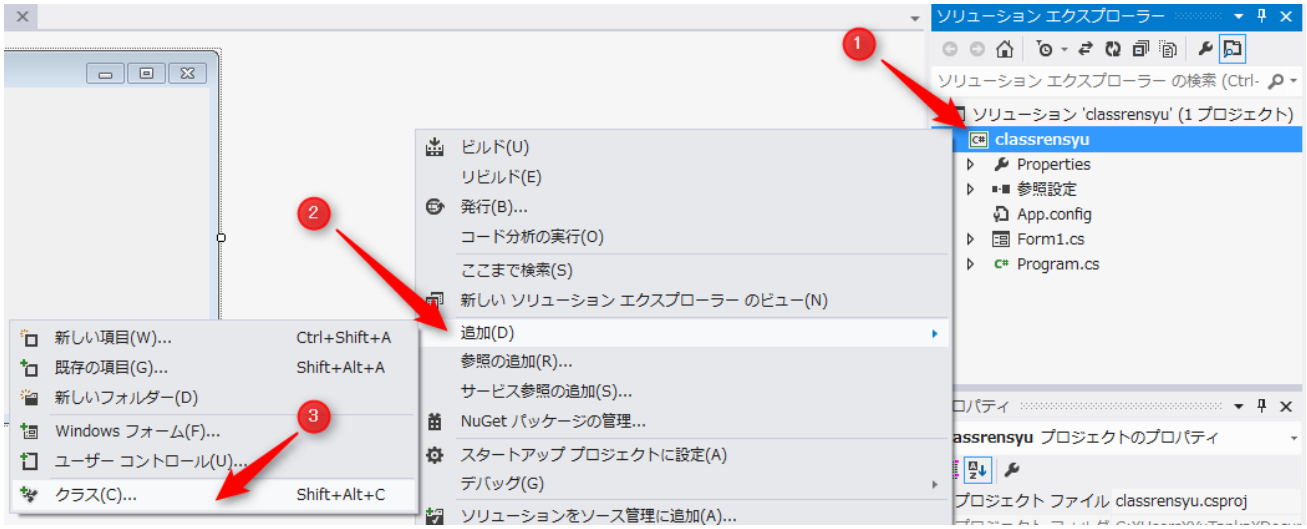

図 7 クラスの作成

2. 「CHito」などの名前で保存
3. Visual Studio Express 2012 では、図 8 のようなコードが自動で作成される

```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
4: using System.Text;
5: using System.Threading.Tasks;
6:
7: namespace object_practice_2016
8: {
9:     class CHito
10:    {
11:    }
12: }
```

Visual Studio で利用する機能を参照するための using 文

ネームスペース名
プロジェクト名が設定される

クラス名

図 8 Visual Studio Express 2012 で自動生成されるクラス

実際のクラスの宣言は、9～11 行目の 3 行で、他の行は削除してもクラスの機能は損なわれない。クラス宣言以外のコードは、以下のようなものである。

1～5 行目　　Visual Studio でよく利用するクラスを簡単に呼び出せるようにするための宣言。Visual Studio 以外の環境でコードを利用したい場合は、不要な宣言は削除する

7,8,12 行目　作成する CHito クラスを、object_practice_2016 というネームスペースに属させるための宣言。ネームスペースについては後述する

プロパティー(変数)とメソッド(関数)の定義

class の宣言に続く { と } に囲まれた範囲に変数定義をすればプロパティーに、関数を定義すればメソッドになる。図 5 で示した人オブジェクトをそのまま実装したのが図 9 である。冒頭 1～6 行目の using は削除しても問題ないので省略している。今後の例でも 1～6 行目は省略する。

```
 7: namespace object_practice_2016
 8: {
 9:     class CHito
10:     {
11:         /** 人数*/
12:         private int count = 0;
13:         /** 自分の名前*/
14:         private string name = “”;
15:
16:         /** コンストラクター*/
17:         public CHito(string nm)
18:         {
19:             name = nm;
20:             count++;
21:         }
22:
23:         /** 人数を返す*/
24:         public int getCount()
25:         {
26:             return count;
27:         }
28:
29:         /** 挨拶*/
30:         public string hello()
31:         {
32:             return "私は" + name + "です。";
33:         }
34:
35:         /** 行動*/
36:         public string action()
37:         {
38:             return "テクテク";
39:         }
40:     }
41: }
```

図 9 CHito.cs バージョン 1

名前やあいさつなどの機能が、CHito クラスに定義された。これにより、各プロパティーとメソッドが何のためのものかが明確になったことに加えて、 Form1 から削除することができる。このクラスを Form1 から使うようにしよう。

インスタンスとは

クラスが設計図や DNA と表現されるのに対して、それらから実際に生成した製品にあたるものを**インスタンス**という。C# では、図 10 のような形でインスタンスを生成する。

```
new クラス名([引数,・・・])
```

図 10 インスタンスの生成

生成したインスタンスは、クラス名で型を指定した変数に代入することができる。これらは通常の変数と同じように利用することができ、配列にすることもできる。図 11 は CHito クラスからインスタンスを１つ生成して、変数 ins に代入して、挨拶のメッセージを生成する例である。

```
CHito ins = new CHito(“佐藤”);      // インスタンスの生成
string mes = ins.hello();           // 挨拶を実行
```

図 11 CHito のインスタンスの利用例

CHito を利用してみよう。

Form1 で CHito を利用する

CHito を入れておくための配列を定義する。Form1.cs の 22 行目あたりに、図 12 の定義を追加する。

```
        /** 人オブジェクト*/
        private CHito[] hitoObjects;
```

図 12 CHito 配列の定義

Form が読み込まれた時に、オブジェクトを生成する。30 行目付近の Form1_Load メソッドに、図 13 のコードを追加する。

```
        private void Form1_Load(object sender, EventArgs e)
        {
            hitoObjects = new CHito[HITO_NAMES.Length];
            for (int i = 0; i < HITO_NAMES.Length; i++)
            {
                hitoObjects[i] = new CHito(HITO_NAMES[i]);
                // 名前リストを表示
                textBox1.AppendText("" + i + ":" + HITO_NAMES[i] + "¥r¥n");
            }
        }
```

図 13 CHito の生成

実行すると、人の名前が表示されるようになる。ボタンはまだ反応しない。

人インスタンスが生成されたので、動作の実装ができる。41 行目付近以降の 3 つのボタンの処理をまとめて実装する(図 14)。

```
        private void buttonHello_Click(object sender, EventArgs e)
        {
            // 選択されているインデックスを取得
            int index = int.Parse(txtIndex.Text);

            // 挨拶
            textBox1.AppendText(hitoObjects[index].hello() + "¥r¥n");
        }

        private void buttonAction_Click(object sender, EventArgs e)
        {
            // 選択されているインデックスを取得
            int index = int.Parse(txtIndex.Text);

            // 行動
            textBox1.AppendText(hitoObjects[index].action() + "¥r¥n");
        }

        private void buttonNumber_Click(object sender, EventArgs e)
        {
            // 選択されているインデックスを取得
            int index = int.Parse(txtIndex.Text);

            // 人数を表示する
            textBox1.AppendText("人は" + hitoObjects[index].getCount() + "人¥r¥n");
        }
```

図 14 ボタンから CHito を呼び出す

実行してボタンを押してみよう。「あいさつ」と「行動」は正しく実行される。

オブジェクトを使わなかった時(図 15)と、使った時(図 16)の「あいさつ」のコードを比べてみる。

```
// 選択されているインデックスを取得
int index = int.Parse(txtIndex.Text);

// 挨拶
textBox1.AppendText("私は" + names[index] + "です。¥r¥n");
```

図 15 オブジェクト未使用

```
// 選択されているインデックスを取得
int index = int.Parse(txtIndex.Text);

// 挨拶
textBox1.AppendText(hitoObjects[index].hello() + "¥r¥n");
```

図 16 オブジェクトを使用

違うのは最後の行の()の中である。図 15 では、 Form1 に自己紹介の文面が入っている。一方の図 16 は、人がどのような挨拶を返すかは書かれていない。

挨拶の内容を書き換えることを考えてみよう。前者では、 Form1 の中から該当箇所を探し出して、コードを変更することになる。後者は Form1 の変更は不要である。 CHito の hello() メソッドで挨拶することは分かっているので、CHito の hello() メソッドを書き換えればよいことは明白である。

チームで開発する場合、前者のような形だと担当分けが難しい。後者なら Form1 と CHito に担当者を割りあてればよいので、チーム開発に向いたスタイルであると言える。

プログラムを分割する際は、分割したコードの詳しい実装を知らなくても使えるようにするべきである。そのような状態を**疎結合**(そけつごう)といい、よいプログラミングスタイルとされているので覚えておこう。

問題点

「あいさつ」と「行動」ボタンは正しく動作した。しかし、「数」ボタンは以下の点でうまく動いていない。

- 人の数が 1 になっている
- 生き物全体の数が表示されていない

次は、人の数を正しく数えることができるようにするためのオブジェクト指向言語の機能を紹介する。

オブジェクト第２段階～static の利用

人の数は CHito クラス自身に数えさせたい。 count という変数を用意して、コンストラクターが呼ばれる時にそれに 1 を加えているのだが、これだとうまくいかない。図 17 を見てほしい。

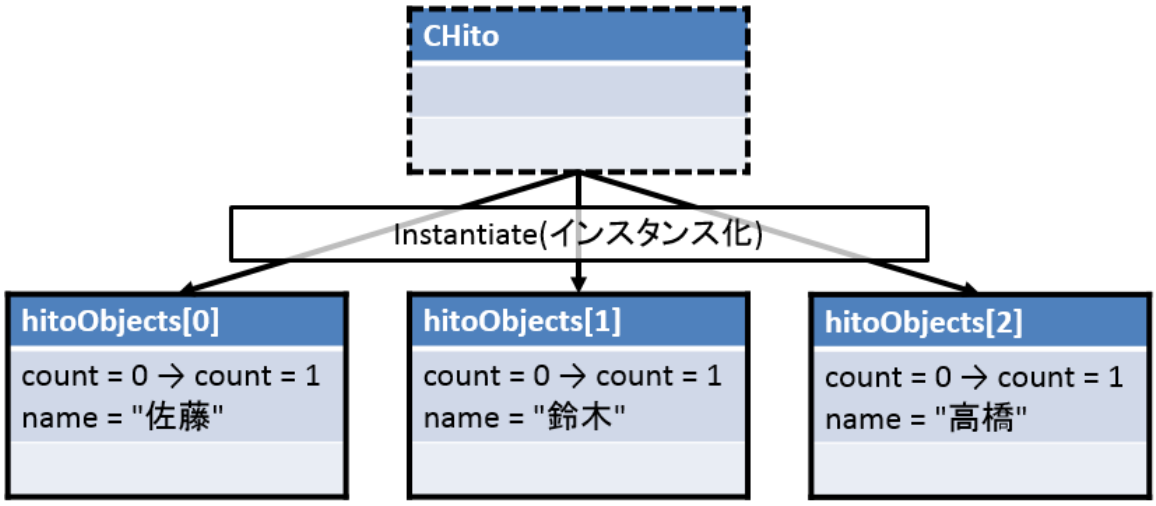

図 17 インスタンス化とプロパティー

CHito クラスから 3 つのインスタンスを生成している。生成されたインスタンスは、それぞれ別の存在なのでプロパティーには別の値が記録される。DNA が同じ双子が、別々の経験を積むのと同じである。

name プロパティーのような普通のプロパティーには必要な性質だが、count に関しては、クラスに１つだけあればよい。そこで利用するのが **static** である。宣言の時に static をつけると、クラス自体がプロパティーやメソッドを持つようになる（図 18）。

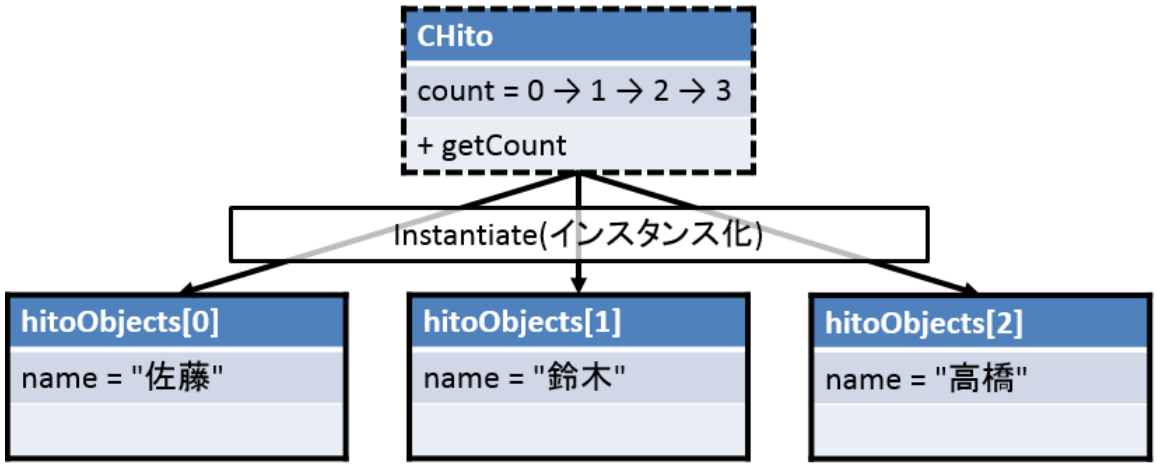

図 18 static の利用

これで、人数を数えることができる。まずは CHito に変更を加える。
12 行目付近の count の宣言に static を追加する。

```
        private static int count = 0;
```

図 19 count に static を追加

24 行目付近の getCount() メソッドの宣言にも static を追加する。

```
        public static int getCount()
```

図 20 getCount()に static を追加

以上で、 count が**クラスプロパティー**、getCount()が**クラスメソッド**になり、**クラスから直接呼び出せる**ようになった。このままだと getCount() の呼び出しでエラーが発生するので、 Form1 の呼び出し部分を変更する。60 行目付近の buttonNumber_Click() メソッドを以下のように修正する。

```
        private void buttonNumber_Click(object sender, EventArgs e)
        {
            // 人数を表示する
            textBox1.AppendText("人は" + CHito.getCount() + "人¥r¥n");
        }
```

図 21 getCount()の呼び出しを変更

実行すると、「数」ボタンの機能が正しく動くことが確認できる。

getCount() がクラスメソッドになったことで、 hitoObjects[] からではなく、 CHito クラスから直接呼び出すようになった。インデックスが不要になったので、その行を削除している。

static で宣言されたクラスプロパティーやクラスメソッドは、インスタンスを生成する必要がないので簡単に機能を提供できる。算術演算クラス(Math)など new をしなくても利用できるクラスでは、この性質を利用している。

以上で、人オブジェクトの機能の実装は完了した。次は、鳥や魚のオブジェクトの実装について考えてみよう。

オブジェクト第３段階～継承

人オブジェクトと同じように鳥オブジェクトを実装した例を図 22 に示す。

```
 7: namespace object_practice_2016
 8: {
 9:     class CTori
10:     {
11:         /** 人の人数*/
12:         private static int count = 0;
13:         /** 自分の名前*/
14:         private string name = "";
15:
16:         /** コンストラクター*/
17:         public CTori(string nm)
18:         {
19:             name = nm;
20:             count++;
21:         }
22:
23:         /** 人数を返す*/
24:         public static int getCount()
25:         {
26:             return count;
27:         }
28:
29:         /** 挨拶*/
30:         public string hello()
31:         {
32:             return "私は" + name + "です。";
33:         }
34:
35:         /** 行動*/
36:         public string action()
37:         {
38:             return "パタパタ";
39:         }
40:     }
41: }
```

図 22 そのまま作成した鳥クラス

このままでも動くのだが、プログラミングスタイルとして良くない点がある。以下の言葉を覚えておこう。

One Fact in One Place

～１つのものは、１つの場所に～

同じコードやデータを繰り返し書くな、という意味である。

人クラスと鳥クラスを比較すると、同じプロパティーやメソッドが見つかる。このようなプログラムは、コピー&ペーストで手軽に作れるので初心者が作りがちである。しかしこれだと、仕様の変更が発生した時にコピー&ペーストした全ての場所を探して同じ修正を繰り返すという不毛な作業が発生する。それに加えて、変更し忘れがあればそこで仕様の不整合やバグが発生することになる。このようによくないコーディング方法を**アンチパターン(anti pattern)**という。

継承とは

人クラスと鳥クラス、ついでに魚クラスも並べて観察すると、共通しているプロパティーやメソッドが見つかるだろう。その反対に、違う部分はどこだろう。

今回の場合、**action()**メソッドが各クラスで違い、それ以外は同じである。

こういう時に利用するのが**継承**である。継承とは、あるオブジェクトの機能を拡張して、新しいオブジェクトを作る機能である。今回の例では、図 23 のようにクラスを設計し直すことができる。

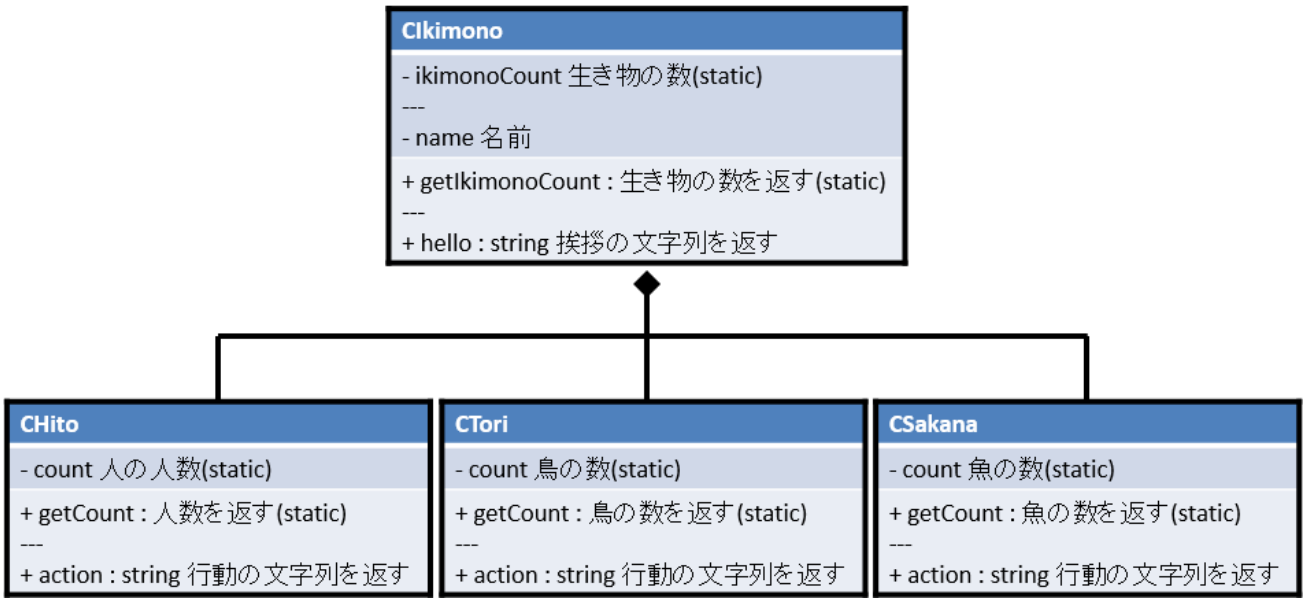

図 23 クラスの継承

新しく **CIkimono** クラスを作成して、共通するプロパティーやメソッドをこのクラスに移動する。CHito 、 CTori、 CSakana クラスには CIkimono を継承させる。継承元となる CIkimono のようなクラスを**親クラス**や**スーパークラス**、**基底クラス(base class)**などという。継承した CHito、 CTori、CSakana のようなクラスを**子クラス**や**サブクラス**、**派生クラス**などという。

生き物クラスの作成

親クラスである CIkimono クラスを作成する。親クラスは普通のクラスとして作成すればよい。

1. ソリューションエクスプローラーから、プロジェクト名を右クリックして、[追加]>[クラス]を選択
2. 名前を CIkimono として作成
3. 図 24 のコードを実装

```
namespace object_practice_2016
{
    class CIkimono
    {
        /** 生き物の数*/
        private static int ikimonoCount = 0;
        /** 名前*/
        private string name = "";

        /** 生き物を数える*/
        public CIkimono(string nm)
        {
            name = nm;        // 名前を記録
            ikimonoCount++;   // 生き物を数える
        }

        /** 生き物の数を返す*/
        public static int getIkimonoCount()
        {
            return ikimonoCount;
        }

        /** 挨拶*/
        public string hello()
        {
            return "私は" + name + "です。";
        }
    }
}
```

図 24 生き物クラス

人クラスを子クラスに再構築

C# でクラスを継承するには、クラス宣言の後ろに **:** (コロン)記号を書き、続けて親クラス名を書く。CIkimono クラスを継承した CHito クラスを宣言するには図 25 のように書く。

```
class CHito : CIkimono
```

図 25 クラスの継承

子クラスに書き換えた人クラスを図 26 に示す。

```
namespace object_practice_2016
{
    class CHito : CIkimono
    {
        /** 人の人数*/
        private static int count = 0;

        /** コンストラクター*/
        public CHito(string nm) : base(nm)
        {
            count++;
        }

        /** 人数を返す*/
        public static int getCount()
        {
            return count;
        }

        /** 行動*/
        public string action()
        {
            return "テクテク";
        }
    }
}
```

図 26 子クラスにした人クラス

これで問題なく動くようになる。name プロパティーと hello() メソッドが無いが、親クラスが持っているので、エラーにならずに実行できるのである。

人クラスの変更点

図 27 で変更点を確認してみよう。

```
...  ...  @@ -6,17 +6,14 @@
 6    6
 7    7   namespace object_practice_2016
 8    8   {
 9        -      class CHito
      9   +      class CHito : CIkimono
10   10          {
11   11              /** 人の人数*/
12   12              private static int count = 0;
13       -          /** 自分の名前*/
14       -          private string name = "";
15   13
16   14              /** コンストラクター*/
17       -          public CHito(string nm)
      15  +          public CHito(string nm) : base(nm)
18   16              {
19       -              name = nm;
20   17                  count++;
21   18              }
22   19
```

図 27 人クラスの変更点

- 9 行目　　**: CIkimono** が追加された。これにより、CIkimono クラスが提供する機能はそのまま CHito は使えるようになった。
- 13～14 行目 name プロパティーは CIkimono クラスに実装されたので不要になったので削除
- 17 行目　　コンストラクターの宣言に **: base(nm)** が追加。これは、親クラスのコンストラクターを呼び出す方法で、名前を渡している。
- 19 行目　　name の制御は CIkimono クラスに移動したので削除

追加されたコードも少しあるが、基本的にはコードが減っていることが確認できただろう。1 つのクラスでは些細な変化でも、子クラスが増えればこの差は大きくなる。

鳥クラスと魚クラスの実装

それでは、鳥クラスと魚クラスをまとめて実装する。これまでと同様の手順で CTori という名前の鳥クラスを作成して図 28 のコードを入力する。CHito クラスをコピーペーストして、違うところを変更すると楽である。

```
 7: namespace object_practice_2016
 8:{
 9:    class CTori : CIkimono
10:    {
11:        /** 鳥の数*/
12:        private static int count = 0;
13:
14:        /** コンストラクター*/
15:        public CTori(string nm)
16:            : base(nm)
17:        {
18:            count++;
19:        }
20:
21:        /** 鳥の数を返す*/
22:        public static int getCount()
23:        {
24:            return count;
25:        }
26:
27:        /** 行動*/
28:        public string action()
29:        {
30:            return "パタパタ";
31:        }
32:    }
33: }
```

図 28 鳥クラス

同様に CSakana クラスを作成して、図 29 のコードを実装する。

```
namespace object_practice_2016
{
    class CSakana : CIkimono
    {
        /** 魚の数*/
        private static int count = 0;

        /** コンストラクター*/
        public CSakana(string nm)
            : base(nm)
        {
            count++;
        }

        /** 魚の数を返す*/
        public static int getCount()
        {
            return count;
        }

        /** 行動*/
        public string action()
        {
            return "スイスイ";
        }
    }
}
```

図 29 魚クラス

Form1 を対応させる

オブジェクトが揃ったので、Form1 に鳥と魚クラスを利用するコードを追加する。ついでに生き物数を表示して当初の仕様通りに動くようにしよう。以下、追加点を示す。

◆ 24 行目付近に、図 30 の宣言を追加

```
/** 鳥オブジェクト*/
private CTori[] toriObjects;
/** 魚オブジェクト*/
private CSakana[] sakanaObjects;
```

図 30 インスタンス変数の定義

◆ 43 行目付近の Form1_Load()メソッド内、人の設定後に図 31 のコードを追加

```
toriObjects = new CTori[TORI_NAMES.Length];
for (int i = 0; i < TORI_NAMES.Length; i++)
{
    toriObjects[i] = new CTori(TORI_NAMES[i]);
    // 名前リストを表示
    textBox1.AppendText("" + i + ":" + TORI_NAMES[i] + "¥r¥n");
}

sakanaObjects = new CSakana[SAKANA_NAMES.Length];
for (int i = 0; i < SAKANA_NAMES.Length; i++)
{
    sakanaObjects[i] = new CSakana(SAKANA_NAMES[i]);
    // 名前リストを表示
    textBox1.AppendText("" + i + ":" + SAKANA_NAMES[i] + "¥r¥n");
}
```

図 31 鳥と魚の生成

◆ 83 行目付近の buttonNumber_Click()メソッドを、図 32 のようにコードを追加

```
private void buttonNumber_Click(object sender, EventArgs e)
{
    // 人数を表示する
    textBox1.AppendText("人は" + CHito.getCount() + "人¥r¥n");
    // 鳥の数を表示する
    textBox1.AppendText("鳥は" + CTori.getCount() + "羽¥r¥n");
    // 魚の数を表示する
    textBox1.AppendText("魚は" + CSakana.getCount() + "匹¥r¥n");
    // 生き物数を表示する
    textBox1.AppendText("生き物は" + CIkimono.getIkimonoCount() + "体。¥r¥n");
}
```

図 32 オブジェクト数を表示

以上で実行すると、起動時に鳥や魚が一覧に追加される。インデックスが間違えているのは、後で直す。[数]ボタンを押すと、当初の仕様の内容が表示される。あとは「あいさつ」ボタンと「行動」ボタンに鳥と魚を実装すれば完成である。

オブジェクト第４段階～ポリモーフィズム

オブジェクト指向を使わないプログラムでは、types 配列にキャラの種類を記録して、switch 文で種類を判別して処理を呼び分けていた(図 33)。

```
// 選択されているインデックスを取得
int index = int.Parse(txtIndex.Text);

// 行動
switch (types[index])
{
    case IKIMONO_TYPE.IKIMONO_HITO:
        textBox1.AppendText("テクテク¥r¥n");
        break;
    case IKIMONO_TYPE.IKIMONO_TORI:
        textBox1.AppendText("パタパタ¥r¥n");
        break;
    case IKIMONO_TYPE.IKIMONO_SAKANA:
        textBox1.AppendText("スイスイ¥r¥n");
        break;
}
```

図 33 当初の行動処理

このようなコードは、シーンを分けたりする時などにも利用してきた。しかし、以下の点が心配である。

- switch 文で処理を呼び分ける方法は、規模が大きくなった時に面倒になりそう
- オブジェクト化した時に、キャラの配列が hitoObjects、 toriObjects、 sakanaObjects に分割してしまったので、 types のようにインデックスで簡単にアクセスができない

この両方をまとめて解決する手法が**ポリモーフィズム(多態性)**である。

ポリモーフィズムは、同じ親クラス(あるいはインターフェース)を持つクラスの間で使える機能である。**インスタンス変数を共有**したり、**メソッドを自動的に呼び分け**たりすることができる。

インスタンス変数の共有

CHito、 CTori、 CSakana のインスタンスはいずれも、親クラスである CIkimono クラス型の変数に代入することができる。

この性質を使って、各クラスのインスタンスを ikimonos 配列にまとめてみよう。

◆ 21～27 行目付近　　人、鳥、魚クラスの配列定義を削除して、以下の定義だけにする(図 34)

```
/** 生き物オブジェクト*/
private CIkimono[] ikimonos;
```

図 34 生き物オブジェクトの定義

◆ 30 行目付近　　Form1_Load()メソッドを以下のように改造する()

```
private void Form1_Load(object sender, EventArgs e)
{
    ikimonos = new CIkimono[
        HITO_NAMES.Length+TORI_NAMES.Length+SAKANA_NAMES.Length];
    int index = 0;
    for (int i = 0; i < HITO_NAMES.Length; i++, index++)
    {
        ikimonos[i] = new CHito(HITO_NAMES[i]);
        // 名前リストを表示
        textBox1.AppendText("" + i + ":" + HITO_NAMES[i] + "¥r¥n");
    }

    for (int i = 0; i < TORI_NAMES.Length; i++,index++)
    {
        ikimonos[index] = new CTori(TORI_NAMES[i]);
```

```
            // 名前リストを表示
            textBox1.AppendText("" + index + ":" + TORI_NAMES[i] + "¥r¥n");
        }

        for (int i = 0; i < SAKANA_NAMES.Length; i++,index++)
        {
            ikimonos[index] = new CSakana(SAKANA_NAMES[i]);
            // 名前リストを表示
            textBox1.AppendText("" + index + ":" + SAKANA_NAMES[i] + "¥r¥n");
        }
    }
```

図 35 初期化を CIkimono に対応

◆ 61 行目付近　　buttonHello_Click()メソッド内の挨拶の呼び出しを図 36 のように ikimonos から呼び出すように変更

```
    private void buttonHello_Click(object sender, EventArgs e)
    {
        // 選択されているインデックスを取得
        int index = int.Parse(txtIndex.Text);

        // 挨拶
        textBox1.AppendText(ikimonos[index].hello() + "¥r¥n");
    }
```

図 36 hello の呼び出しを変更

◆ 72 行目付近　　buttonAction_Click()メソッド内の行動呼び出しを図 37 の通り、変更(エラーが発生するので、この段階ではコメントアウトしておく)

```
    private void buttonAction_Click(object sender, EventArgs e)
    {
        // 選択されているインデックスを取得
        int index = int.Parse(txtIndex.Text);

        // 行動
        // textBox1.AppendText(ikimonos[index].action() + "¥r¥n");
    }
```

図 37 action の呼び出しを変更

以上でインスタンスの配列を CIkimono 型に変更できた。実行して、配列を CIkimono に統一してもエラーが発生せずに実行できることを確認してみよう。また、インデックスを 0、3、8 などに変更しながら「あいさつ」ボタンを押して、全ての生き物があいさつできることを確認しよう(図 38)。

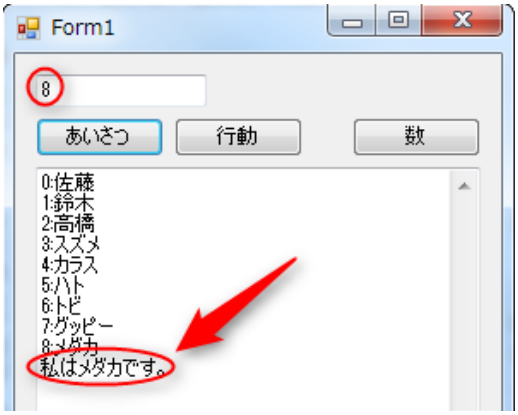

図 38 インデックス 8 での実行例

子クラスのメソッドの呼び分け～abstract～

あとはエラーが発生している action() メソッド呼び出しの解決である。まずは 72 行目付近の action() の呼び出しのコメントを図 39 のように外す。

```
            // 行動
            textBox1.AppendText(ikimonos[index].action() + "¥r¥n");
```

図 39 action のコメントを外す

'CIkimono'に'action'の定義が含まれておらず・・・というようなエラーが発生するようになる。

hello() メソッドは CIkimono クラスのメソッドなので ikimonos 配列から呼び出せた。しかし、action() メソッドは子クラスのメソッドなので、 ikimonos には存在しない。

そこで、**子クラスに必ず実装するから呼び出していいよ**、という定義を親クラスに実装する。そのような実装予定のメソッドを**抽象メソッド(abstract method)**といい、 **abstract** キーワードをメソッド定義の場所に書く。親クラスに実装はないので、中身は書かなくてよい。

CIkimono.cs を開いて、クラス定義の中に以下の定義を追加する(図 40)。

```
        /** 行動の抽象メソッド*/
        public abstract string action();
```

図 40 抽象メソッド action を定義

まだエラーが発生する。抽象メソッドを持つクラスは実装が足りないのでインスタンスを生成することができない。そのことをクラス定義に abstract キーワードを追加して、**抽象クラス**にすることで明示する必要がある。

CIkimono クラスの 9 行目付近のクラス宣言に、図 41 のように abstract を追加する。

```
abstract class CIkimono
```

図 41 抽象クラス

最後にもう一つやることがある。子クラスを実装する時に、うっかり、親クラスで抽象宣言されたメソッドと知らずに、別の目的のメソッドを実装してしまう可能性がある。そのようなミスを防ぐために、抽象メソッドを実装する際には、**override** キーワードを付けるルールになっている。

CHito, CTori, CSakana の全ての action() メソッドに、図 42 のように override を追加する。

```
public override string action()
```

図 42 抽象メソッドを override する

ポリモーフィズムの活用により、図 33 で十数行あったコードが、図 39 の 1 行で済むようになった。この方法ならキャラの種類が増えてもコードは増えない。キャラの他に、シーンの呼び分けなど様々な場所で活用できる便利な機能である。

以上で、オブジェクト化は完了である。

カプセル化

すでに利用しているが、クラスや変数、メソッドには、**private, protected, public** という公開レベルを指定するキーワードが利用できるので、紹介しておく。

- private
  - 自分しか見ることができない
  - 省略すると private になる
- protected
  - 継承の時に利用する
  - 親クラスで protected 宣言したプロパティーやメソッドは、親クラス自身と、継承先の子クラスのみで利用できる
- public
  - どこからでもアクセスできるプロパティーやメソッドになる
  - オブジェクトが、外部に提供する定数やメソッドにつける
  - プロパティーを公開したい場合は、アクセサ―(get; set;)の利用を検討すること

オブジェクト内でしか利用しないプロパティーやメソッドは隠した方がよい。機能が少ない方が、利用する側は理解しやすくなる。必要な機能だけ公開して、それ以外のものを隠すことを**カプセル化**という。

原則として、まずは private で実装をして、問題が発生したら protected → public と公開レベルを上げていくのがよい。

ネームスペースとは

公開レベルによって、クラス内にプロパティーやメソッドを囲い込むことができる。しかし、プロジェクトの規模が大きくなってくると、クラス名が衝突する可能性が出てくる。そこで利用するのがネームスペースである。クラス宣言を { } で囲んだ直前にあった図 43 がネームスペースの宣言である。

```
namespace object_practice_2016
```

図 43 ネームスペースの宣言

例えば、object_practice_2016 というネームスペースで定義された CHito クラスを利用したい場合、以下のいずれかの方法でアクセスする必要がある。

1. 同じネームスペースで定義されたクラスからは、自由にアクセスできる(例は全てこれに該当)
2. 違うネームスペースのクラスで利用したい場合は、ファイルの冒頭に図 44 のような using 文を追加する

```
using object_practice_2016;
```

図 44 using

3. using を使わない場合は、図 45 のようにネームスペースを明示的に指定する

```
object_practice_2016.CHito.getCount();
```

図 45 using を利用しないアクセス

これにより、より規模が大きいプロジェクトでの名前の衝突を避けるのである。Visual C# のコードにも、 Unity のコードにも、自動的に冒頭に using の行があるのは、それぞれの環境でよく利用するクラスを簡単に使えるようにするためだったのである。

クラス定義の時に、namespace は省略することもできる。その場合は、using を使わなくても直接利用することができる。小さくて、再利用性が低いプロジェクトでは、敢えて省略することもある。

まとめ

オブジェクト指向について、重要な点は以上である。冒頭で挙げた不満点が以下の通り解決した。

- どこに変数や関数があるのか分からない
  - 関連するクラスに定義されている
- 変数名や関数名の管理が大変
  - 変数名や関数名はクラスで気を配ればよい
- 再利用が難しい
  - CIkimono を継承することで、次々に新しいクラスが作れる
  - switch がなくなったので、種類が増えても大丈夫

まだ、いくつか不満点がある。解決策を検討してみよう。

- 配列の数を最初に定義する必要がある
  - List クラスを調べる
- 各クラスの count と getCount() が冗長
  - CHito、CTori、CSakana からそれらを削除して、CIkimono クラスにグループを管理する機能を実装するとよいだろう

今回作成した CIkimono、 CHito、 CTori、 CSakana クラスは、簡単に Unity で動かすことができる。以下、ヒントを挙げるので試してみよう。

- 上記 4 クラスを Unity の Project ビューにドラッグ＆ドロップして追加する
- 4 つのクラスの 1～6 行目の using を削除する
- 1 つ新しいクラスを作成して、 Form1 の変わりに以下のことをする
  - Form1 に定義されている定数やプロパティーを追加する
  - インデックスボックス代わりに public int index; を定義。Inspector で値を設定
  - Start()メソッドで、 Form1_Load()の処理を行う
  - textBox1.AppendText()を、 Debug.Log()に変更
  - Update()メソッド内で、マウスのクリックなどで、hello()や action()を呼ぶ

ここで実装した完成プロジェクトをダウンロードする URL を以下に示す。

https://github.com/tanakaedu/object-practice-2016/archive/fix.zip