オブジェクト指向及び演習・第 3 回

2024 年 4 月 29 日

今回の内容

3.1 インスタンス変数

生成されたオブジェクトは、その種類 (クラス) に応じていろいろな仕事を行うことになりますが、その仕事を行うためには、それなりの情報をそのオブジェクト自身が記憶しておかなければなりません。たとえば、`Card` クラスのそれぞれのインスタンスは、少なくとも次のような情報を記憶していなければ、その仕事を行うことができないはずです。

1. 自分のスートとランク
2. 自分の位置 (x 座標と y 座標)
3. 自分が表向きか裏向きか
4. 自分の表向きの画像と裏向きの画像

この他にも、自分が使われているゲーム盤はどれか、どこかの山 (`Deck` や `Pile` のクラスのインスタンス) に属しているのかいないのか、属しているならどの山か、などを記憶しておく必要があるかも知れません。

Java では、このような個々のオブジェクトに関する情報を、それぞれのオブジェクトに設けられた**インスタンス変数**[1]と呼ばれる変数で記憶します。1 つのオブジェクトが複数のインスタンス変数を持つのが普通ですので、インスタンス変数は、通常の変数やクラス変数と同様に名前を付けて区別し、それぞれが特定のデータ型を持ちます。つまり、`int` 型や `double` 型のデータを記憶するインスタンス変数もあれば、あるクラス (たとえば `Suit` や `Pile`) のインスタンスを記憶するインスタンス変数もあります[2]。そのオブジェクトがどのような名前でどのようなデータ型のインスタンス変数を持つかは、そのオブジェクトのクラスによって変わってきます[3]。

`Card` クラスのインスタンスは、前回の付録で紹介したように、それぞれ

---

[1] **インスタンスフィールド**と呼ぶこともあります。

[2] 個々のインスタンスが持つ変数なので「インスタンス変数」と呼ぶのであって、あるクラスのインスタンスを記憶するから「インスタンス変数」と呼ぶのではないと言うことに注意してください。たとえば、`int` 型のインスタンス変数は (何かのクラスのインスタンスではなく) `int` 型のデータ (整数値) を記憶します。

[3] 第 5 回で勉強する「クラス宣言」(オブジェクトの設計図に相当するもの) の中で宣言します。

```
Suit suit;    // このカードのスート
Rank rank;    // このカードのランク
```

という 2 つのインスタンス変数を持っています。

メモ

3.2 インスタンス変数への代入と参照

プログラム中で、インスタンス変数へのアクセス (参照や代入) を行うには次のような書式を用います。

オブジェクトを表す式 . インスタンス変数名

インスタンス変数へのアクセスを行うようなプログラムの例を見てみましょう。

P301.java

```
 1 import jp.ac.ryukoku.math.cards.*;
 2
 3 class P301 {
 4     public static void main(String[] args) {
 5         GameFrame f = new GameFrame();
 6         Deck d = new Deck();
 7         Pile reds = new Pile();       // ハートとダイヤを集める山
 8         Pile blacks = new Pile();     // スペードとクラブを集める山
 9         f.add(d, 100, 100);
10         f.add(reds, 210, 400);
11         f.add(blacks, 510, 400);
12         d.shuffle();
13         while (d.count() > 0) {      // デッキが空になるまで繰り返し
14             Card c = d.pickUp();
15             c.flip();
16             /* カードのインスタンス変数 suit を調べる */
17             if (c.suit == Suit.HEARTS || c.suit == Suit.DIAMONDS) {
18                 c.moveTo(reds);
19             } else {
20                 c.moveTo(blacks);
21             }
22         }
23     }
24 }
```

このプログラム P301.java は、デッキから引いたカードのスートを調べて、ハートとダイヤを集める山とスペードとクラブを集める山の二つにカードを振り分けるものです。17 行目の c.suit という式で、Card クラスのインスタンスが持つ suit という名前のインスタンス変数の値を調べています[4]。Card クラスのインスタンス変数 suit は、そのカードのスートを表す Suit クラスの

[4] このプログラムの 13 行目では、Deck クラスのインスタンスメソッド count を起動して、デッキに残っているカードの枚数を調べています。

インスタンスを記憶しています。

メモ

3.3 カプセル化

通常の変数と同様に、インスタンス変数に対する代入を行うことも可能ですが、この科目で使用している Card クラスに関しては、そのインスタンス変数 suit と rank への代入は禁止されています[5]。また、Card クラスのインスタンス変数には、これら 2 つ以外にも

```
int x, y;                // このカードの(左上角の)座標
int width, height;       // このカードの幅と高さ
```

などいくつかありますが、これらのインスタンス変数は隠されていて、参照も代入もできません。

オブジェクトが行うことのできる仕事の手順は、インスタンスメソッドの定義[6]として、クラス宣言[7]の中に記述されています。インスタンス変数は、そのクラスのオブジェクトが仕事をするための記憶場所ですので、インスタンス変数にアクセスするのは、これらのインスタンスメソッドの定義の中からというのが基本となります。それ以外の部分、たとえば、単にそのクラスを利用しているプログラムからは、インスタンス変数を見えないようにしておくのが普通です[8]。

たとえば、Card クラスのインスタンス変数である x や y を、Card クラスを利用しているだけのプログラムが勝手に書き換えてしまうと、Card クラスのインスタンスは自分の仕事を正常に行うことができなくなってしまいます。なぜなら、これらの変数を書き換えても、これらの変数が記憶している int 型の値が変わるだけで、ゲーム盤上に表示されているカードの画像の場所が変わるわけでありませんから、これらの変数の値と、ゲーム盤上に実際に表示されているカードの位置と間で矛盾が生じてしまうからです。

もし、Card クラスを利用しているプログラマが Card クラス内部の仕組みをよく知っているのなら、これらの変数を勝手に書き換えるとともに、カード画像の表示位置の変更など、一緒にやらないといけない仕事を自分で行って、矛盾が生じないようにすることも可能かも知れません。しかし、そのようなプログラムは、その後、Card クラス内部の仕組みが変更されてしまうと動かなく

---

[5] Java では、final というキーワードを使って、(一旦初期化したら) 書き換えることのできない変数を宣言することができます。このような変数を「final 変数」と呼びます。Suit クラスのクラス変数 SPADES、HEARTS、DIAMONDS、CLUMBS や、Rank クラスのクラス変数 ACE、DEUCE、··· QUEEN、KING も final 変数です。

[6] インスタンスメソッドの定義については、第 5 回で勉強することになります。ここでは、C 言語における関数の定義のようなものと考えてください。

[7] 前回、クラス宣言は、そのクラスのオブジェクトの設計図に相当するということ説明しました。

[8] Java では、インスタンス変数やクラス変数を (クラス宣言の中に) 宣言する際に、**アクセス修飾子**と呼ばれる public、protected、private などのキーワードを使って、その変数が見えるプログラムの範囲 (スコープ) を設定することができます。

なってしまう可能性が出てきます。

Card クラスのプログラムを書いたプログラマと、その Card クラスを利用しているプログラマは、お互いのことを知らない別人であるのが普通ですし、非常に多くの人がそのクラスを利用するかも知れませんので、Card クラスを改良する度に、それを利用しているすべてのプログラムを修正するのは実質的に不可能です。結局、高速化や機能強化、あるいは不具合の修正のために、Card クラスの仕組みを変更することが不可能になってしまいます。

このような問題を防ぐため、Card クラスでは、これらの変数の存在とともに、それに関連する内部の仕組みを隠しておいて、代りに、次の 3 つのインスタンスメソッドを提供しています。

```
void moveTo(int x, int y);   // このカードを (x, y) へ移動する
int getX();                  // このカードの x 座標を返す
int getY();                  // このカードの y 座標を返す
```

実際の moveTo メソッドの定義では、インスタンス変数 x、y の値を変更するとともに、ゲーム盤上に表示されているカードの画像を新しい位置に移動するという仕事を行っていますが、このやり方の詳細を、Card クラスを利用しているプログラムが意識する必要はありません。後に、Card クラス内部の仕組みが変更されたとしても、それに合わせて、これら 3 つのインスタンスメソッドが行う仕事の手順も変更されているはずですので、Card クラスを利用しているプログラムはそのまま問題なく動作してくれるはずです。

このように、プログラム (クラス宣言) の独立性を保つために、オブジェクトが仕事を行う仕組みの詳細を隠すことを、一般に**カプセル化 (encapsulation)**[9]と呼びます。カプセル化は、オブジェクト指向プログラミングに付随する重要な考え方の 1 つです。通常、オブジェクト指向プログラミングでは、カプセル化を行って、クラスのインスタンスが持つインスタンス変数を、そのクラスを利用するだけのプログラムからは見えないようにします[10]。

メモ

3.4 オブジェクトの状態

ここまで説明したように、オブジェクトは、自分の仕事を行うために必要な情報を、自分自身のインスタンス変数に記憶しています。インスタンス変数はカプセル化によって隠されている場合もありますが、隠れているものも含めて、1 つのオブジェクトが持つすべてのインスタンス変数の値の組のことを、そのオブジェクトの**状態**と呼びます。

[9]オブジェクト指向の考え方に限らず、プログラムの独立性を保つために、その実現方法の詳細を隠すことを、一般に**情報隠蔽 (information hiding)**と呼びます。

[10]Card クラスでは、suit と rank に関してはあえて隠していません。これら 2 つの変数の値は、Card のインスタンスが生成されたときに決定されて、それ以降は変更されることはありませんので、Card クラスのインスタンスのカプセル化の障害とならないからです。

同じオブジェクトに同じように仕事を依頼 (インスタンスメソッドを起動) しても、そのときのオブジェクトの振る舞いは、そのときのオブジェクトの状態によって変って来ることに注意してください。たとえば `Card` クラスのインスタンスは、自分が表向きなのか裏向きなのかに関する情報をそのインスタンス変数に記憶していますが、`flip` というインスタンスメソッドを呼び出した時の振る舞いは、その値によって変ってきます。そのとき、表向きなら裏向きになりますし、裏向きなら表向きになります。

メモ

3.5 オブジェクトとメモリ

個々のオブジェクトはインスタンス変数を持っていますので、プログラムの実行途中に (インスタンス生成式が評価されて) オブジェクトが生成されると、その状態を記憶するために必要なメモリが Java 仮想機械 (JVM) によって割り当てられます[11]。Java プログラムがオブジェクトを生成し続けると、いずれはメモリが足りなくなって、新しいオブジェクトを生成できなくなってしまいますので、何らかの方法で不要なオブジェクトを消滅させて、そのオブジェクトのために使用されているメモリを解放する必要があります。

Java では、オブジェクトの消滅をプログラマが明示的に指示する必要はありません。JVM には、使用される見込みが完全になくなったオブジェクトを自動的に発見し、そのオブジェクトを消滅させてメモリを解放する仕組み[12]が用意されていますので、プログラマは、要らなくなったオブジェクトを変数などで記憶し続けないということだけを注意すれば済むようになっています。

メモ

3.6 値としてのオブジェクト

これまで見てきたように、オブジェクトは、変数に記憶したり、メソッドやコンストラクタに引数として渡したり、メソッドの戻り値として返されたりします。Java では、`int` 型や `double` 型の値

[11] オブジェクトの状態を記憶するために Java 仮想機械 (JVM) が使用するメモリ領域は、**ヒープ (heap)** と呼ばれています (ヒープソートの「ヒープ」と同じ英単語ですが特に関係はありません)。`java` コマンドの `-Xmx` というコマンドライン引数で、JVM が使用するヒープ領域の最大の大きさを指定することができます。

[12] これを**ごみ集め (garbage collection)**と呼びます。

(データ) と同じように、オブジェクトも値 (データ) として取り扱うことができます。

ただし、その値の扱われ方は int 型や double 型などとはかなり異なるものとなっています。たとえば、

```
int x1 = 123;
int x2 = 456;
```

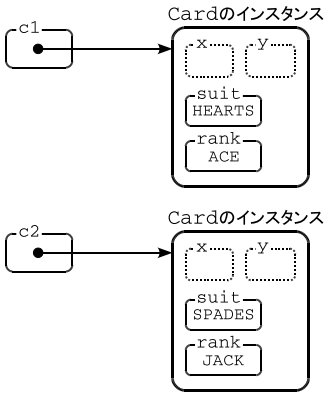
というプログラムを実行すると、123 や 456 という int 型の値は、それぞれ、変数 x1 と x2 の中に記憶され、図のような状態になります。続いて、

```
x1 = x2;
```

を実行すると、x2 に記憶されていた 456 が y にコピーされて、変数 x1 の中にも 456 という値が記憶されることになります。

次にオブジェクトの場合を見てみましょう。たとえば、

```
Card c1 = new Card(Suit.HEARTS, Rank.ACE);
Card c2 = new Card(Suit.SPADES, Rank.JACK);
```



というプログラムを実行すると、それぞれ、ハートのエースとスペードのジャックを表す Card クラスのインスタンスが 2 つ生成されますが、int 型の場合と違って、変数 c1 と c2 には、それら 2 つのオブジェクト自身が記憶されるのではなく、右の図[13]のように、それぞれのオブジェクトを指し示すためリンクのようなもの (図中の矢印) が記憶されます。このリンクのことを Java では**参照**と呼びます。

メモ

先に説明したように、オブジェクトが生成されるときには、その状態を記憶するために一定の大きさのメモリ領域が割り当てられますが、「参照」は、そのメモリ領域のアドレスのようなものと考えて構いません[14]。

先ほどのプログラムに続いて、

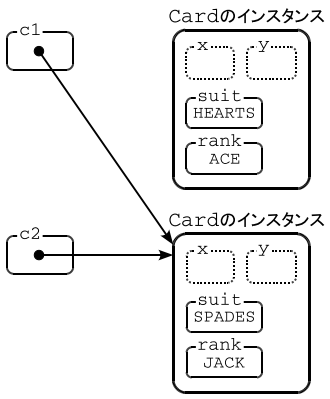
```
c1 = c2;
```

[13]図中の x、y、suit、rank は、Card カードクラスのインスタンスが持つインスタンス変数の一部を表していますが、suit と rank は、それぞれ Suit クラスと Rank クラスのインスタンスを記憶する変数ですので、図中に記されている HEARTS や ACE などは、ハートやエースを表すオブジェクトへの参照 (リンク) がそこに記憶されていることを意味しています。

[14]C 言語の場合で考えると、オブジェクトは malloc 関数でメモリを割り当てられた構造体のようなもので、そのオブジェクトへの「参照」とは、その構造体へのポインタ型の値 (アドレス) のようなものです。

を実行したすると、変数 c2 から c1 へ (スペードのジャックへの) 参照がコピーされ、その結果は右の図のようになります。スペードのジャックのオブジェクトが複製される (同じカードが 2 枚になる) わけではないことに注意してください。

変数 c1 と c2 は同じカード (スペードのジャック) への参照を記憶することになりますので、この後のプログラムでは、c1.flip(); を実行しても、c2.flip(); を実行しても、1 つの同じカードが反転することになります。



メモ

3.7 同じオブジェクトであることの判定

P301.java の 17 行目では、デッキから取り出したカードのスートを判定していましたが、そこでは、Card のインスタンス変数 suit の値と、Suit クラスのクラス変数 HEARTS や DIAMONDS の値とを == という演算子で比較していました。これらの変数 suit、HEARTS、DIAMONDS には、Suit クラスのインスタンスがそれぞれ記憶されていますが、== 演算子は、int 型などの数値が等しいかどうかを判定できるのと同じように、2 つのオブジェクトが同一のオブジェクトであるかどうかを判定できます。ただし、ここでの「同一のオブジェクトである」の意味は、(2 つの参照が)「1 つの同じオブジェクトを指している」ということです。たとえば、ハートのエースのカードを

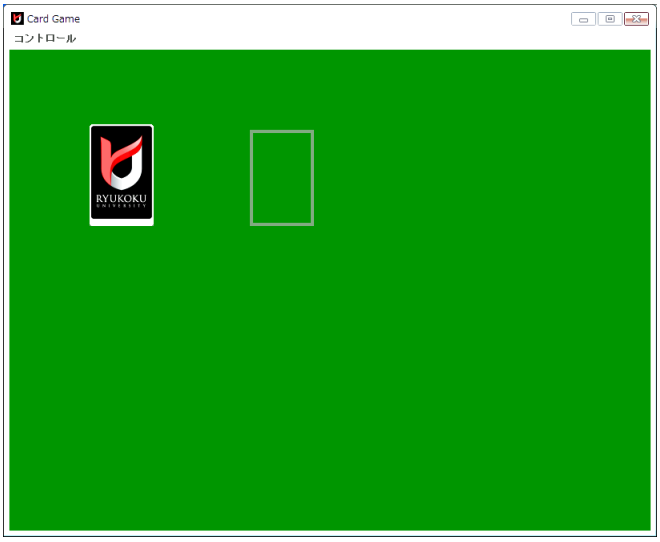
```
Card c1 = new Card(Suit.HEARTS, Rank.ACE);
Card c2 = new Card(Suit.HEARTS, Rank.ACE);
```

のように 2 枚生成して、変数 c1 と c2 にそれぞれ代入したとすると、c1 == c2 は成り立ちません。c1 == c2 が成り立つのは、c1 と c2 が、同じ 1 つのオブジェクト (への参照) を記憶している場合のみです。

メモ

3.8 演習問題

1. 次のようなプログラム `P302.java` を作成しなさい。このプログラムでは、まず、次の図のように、ゲーム盤の (100, 100) の位置にジョーカーを含まないデッキを、(300, 100) に空の山 (`Pile` クラスのインスタンス) を置き、デッキはシャッフルします。

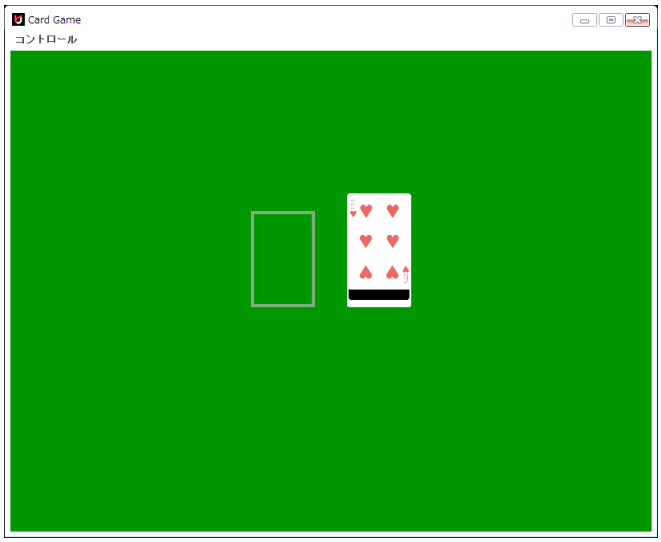


デッキの 1 番上のカードから順に 1 枚ずつ引いて表向きにし、スペードの 10、J、Q、K、A のいずれかであれば、ゲーム盤の下部に移動し左から右へ並べていきます。これらのいずれでもなければ、右隣の山へ裏向きにして追加します。最終的なゲーム盤の状態は、たとえば次の図のようになります。

ゲーム盤の下部に残す 5 枚のカードの x 座標は、デッキから引いた順に 100、200、300、400、500 です。y 座標は 5 枚とも 400 です。

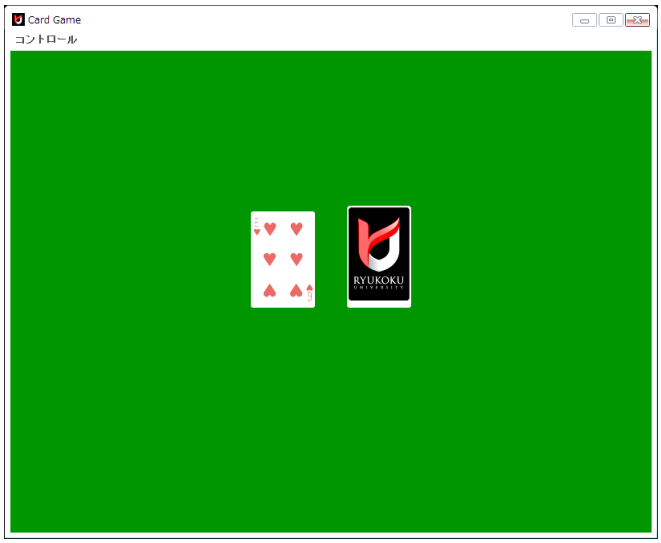
2. 次のようなプログラム `P303.java` を作成しなさい。このプログラムでは、まず、ゲーム盤の (420, 200) の位置にジョーカーを含まないデッキを、(300, 200) に空の山を置き、デッキは

シャッフルします。その後、デッキの１番上のカードを引いて表を上にし、次の図のような状態にします。



ここで３秒間待って、この１番上のカードを伏せてデッキに戻し、元の状態にします。その後、デッキを再びシャッフルし、デッキの１番上から１枚ずつカードを引いて、表を上にして左隣の山へ１枚ずつ移動していきます。最初にめくったカードが移動したところでこの作業を終わります。

たとえば、上の図のように、最初にめくったカードがハートの６であれば、最終的なゲーム盤の状態は次の図のようになります。



デッキの１番上のカードをめくった後、３秒間待つためには、Card クラスのインスタンスメソッド

```
void pause(int msec);
```

を、3000 を引数にして起動してください。pause は、何もしないまま、引数で指定された時間が経過するのを待つメソッドです。引数に指定する待ち時間の単位はミリ秒 (ms) となっています。

また、めくったカードを伏せて、デッキをもう一度シャッフルする際には、めくったカードがデッキの一部になっていないといけないことに注意してください。pickUp メソッドでデッキから取り出したカードを元のデッキに戻すためには、Deck クラスのインスタンスメソッド add を使うことができます[15]。

オブジェクト指向及び演習・第3回・終わり

[15]前回の付録を参照してください。