オブジェクト指向及び演習・第6回

2024年5月27日

今回の内容

6.1 コンストラクタの多重定義 6–1
6.2 同じクラスの他のコンストラクタの起動 6–2
6.3 デフォルトコンストラクタ 6–3
6.4 クラス変数とクラスメソッド 6–3
6.5 パッケージ 6–7
6.6 インポート宣言 6–8
6.7 演習問題 6–11

6.1 コンストラクタの多重定義

前回の Hand.java には、

```
12      Hand(int x, int y) {
13          this.x = x;
14          this.y = y;
15      }
```

のようなコンストラクタが宣言されていて、左端の手札の位置を指定して Hand クラスのインスタンスを生成できるようになっていました。しかし、2番目以降の手札の位置は、この x と y の値の他に

```
 7      int deltaX = 100;      // 隣り合った手札の x 座標の差
```

と宣言されたインスタンス変数の値で決まります。この deltaX の値を変えたい場合は、Hand クラスを使用しているプログラムの側で、

```
Hand h = new Hand(160, 400);
h.deltaX = 25;
```

のように、生成したインスタンスの deltaX の値を、後で書き換えないとなりません。これは、ちょっと不便ですし、また不自然なので[1]、deltaX の値も指定して、Hand のインスタンスを生成できるようにしたくなります。

前回、簡単に紹介したように、Java では、引数の数やデータ型の違いで区別できる場合は、複数のコンストラクタや (同名の) メソッドを複数定義することができます。これを、**多重定義**、あるいは**オーバーロード (overloading)**と呼びます。

手札の位置 x、y と、deltaX の値を指定して、Hand のインスタンスを初期化する3引数のコンストラクタを追加して

```
        Hand(int x, int y) {
            this.x = x;
            this.y = y;
        }
```

[1] そもそも、カプセル化の観点からは、deltaX を、Hand クラスの外部からアクセスできないようにすべきです。

```
        Hand(int x, int y, int deltaX) {
            this.x = x;
            this.y = y;
            this.deltaX = deltaX;
        }
```

のようにコンストラクタを多重定義することができます。new Hand(160, 400) のようにインスタンス生成式を書けば、最初のコンストラクタ宣言が、new Hand(160, 400, 25) のように書けば、2番目のコンストラクタ宣言が使われてインスタンスの初期化が行われます。

メモ

6.2 同じクラスの他のコンストラクタの起動

この2つのコンストラクタが行う仕事を見てみると、2番目のコンストラクタの仕事の一部は最初のコンストラクタの仕事に含まれています。Java では、コンストラクタ本体の先頭に限って、同じクラスの他のコンストラクタを起動して、初期化の仕事の一部を任せることもできます。この時のコンストラクタの宣言は、

```
クラス名 ( 仮引数宣言の列 ) {
    this( 他のコンストラクタへ渡す引数の列 );
    ⋮
}
```

という書式になります。この書式に現れている this は、初期化の対象となっているインスタンスを表す this と同じキーワードですが、こちらの場合は this(···) のように、あたかも this というメソッドのメソッド起動式の形で現れていることに注意してください。ただし、この this(···); という文の実行は、コンストラクタ本体の先頭で1回だけしか許されません。

この書式を使うと、Hand クラスの2番目のコンストラクタの宣言は、

```
    Hand(int x, int y, int deltaX) {
        this(x, y);
        this.deltaX = deltaX;
    }
```

のように書くことができます。また、逆に、2番目の宣言はそのままにしておいて、最初のコンストラクタを

```
    Hand(int x, int y) {
        this(x, y, 100);
    }
```

と宣言することも可能です。

6.3　デフォルトコンストラクタ

Java のクラスには、少なくとも 1 つのコンストラクタが必要です。ただし、クラス宣言の中でコンストラクタを全く宣言していない場合は、自動的に

```
public [クラス名]() {}
```

というコンストラクタ宣言があるものと見なされます[2]。このコンストラクタを**デフォルトコンストラクタ**と呼びます。

たとえば、Hand クラスに全くコンストラクタが宣言されていない場合でも、new Hand() というインスタンス生成式で Hand のインスタンスを生成することは可能となります[3]。

デフォルトコンストラクタは、1 つもコンストラクタが宣言されていない場合のみ補われますので、Hand クラスのようにコンストラクタを 1 つ以上宣言した場合は、new Hand() という形のインスタンス生成式を使うことはできません。もし、そのようなインスタンス生成式を使いたい場合は、自分で引数なしのコンストラクタを宣言する必要があります。

6.4　クラス変数とクラスメソッド

クラス宣言とはオブジェクトの設計図であると考えることができ、インスタンス変数、コンストラクタ、インスタンスメソッドの宣言をすることで、そのクラスのインスタンスが、どのような状態を持っていて、どのような振る舞いをするのかを定義していました。

これがクラス宣言の主要な役割ですが、クラス宣言の中には、そのクラスのインスタンスの状態 (変数) や振る舞い (メソッド) ではなく、そのクラスそのものに関連するプログラム全体の状態や振る舞いを宣言することもできます。そのクラスに関連するプログラム全体の状態は**クラス変数**[4]の値として、そのクラスに関連するプログラム全体の振る舞いは**クラスメソッド**[5]として定義されます。

たとえば、次のプログラムは、前回の演習問題で作成した Hand クラスの宣言に、さらにクラス変数 CAPACITY とクラスメソッド createHands の宣言を追加したものです。

---

[2] 先頭の public は、プログラムのどの部分からでも (このコンストラクタを) 起動できることを示すアクセス修飾子です。

[3] この場合、デフォルトコンストラクタは何の仕事もしませんので、インスタンス変数の x や y の値は 0 となります。

[4] Java の言語仕様書では**静的フィールド**と呼んでいます。

[5] Java の言語仕様書では**静的メソッド**と呼んでいます。

Hand.java

```
 1  import jp.ac.ryukoku.math.cards.*;
 2
 3  class Hand {
 4      /* このクラスのクラス変数 */
 5      static final int CAPACITY = 5;
 6
 7      /* このクラスのインスタンス変数の宣言 */
 8      int x, y;                           // 左端の手札の位置
 9      int deltaX;                         // 隣の手札との x 座標の差
10      int numCards;                       // 手札の枚数
11      Card[] cards = new Card[CAPACITY];  // 手札の配列
12
13      /* このクラスのコンストラクタの宣言*/
14      Hand(int x, int y) {
15          this.x = x;
16          this.y = y;
17      }
18
19      Hand(int x, int y, int deltaX) {
20          this(x, y);
21          this.deltaX = deltaX;
22      }
23
24      /* このクラスのクラスメソッド createHands の宣言 */
25      static Hand[] createHands(int num, int x, int y,
26              int dx, int shift) {
27          Hand[] hands = new Hand[num];
28          for (int i = 0; i < num; i++) {
29              hands[i] = new Hand(x, y, dx);
30              x += shift;
31          }
32          return hands;
33      }
34
35      /* このクラスのインスタンスメソッド add の宣言 */
36      void add(Card c) {
37          if (numCards < CAPACITY) {
38              c.moveTo(x + numCards * deltaX, y);
39              cards[numCards++] = c;
40          }
41      }
 ⋮
```

メモ

クラス変数

クラス変数は、そのクラスに関連してプログラムが記憶しておかなければならない値の記憶場所で、C 言語の大域変数 (グローバル変数) を、クラスごとに分類したものと考えることができます。Hand クラスでは、手札の最大の枚数を CAPACITY という名前のクラス変数で記憶しています。

クラス変数は、Hand.java の 5 行目のように、static というキーワードを使って、

static 型名 変数名の列 ;

という書式で宣言します[6]。インスタンス変数は、そのクラスのインスタンスごとに作られますが、クラス変数は起動したプログラムの中でただ 1 つ用意されます。インスタンス変数は、そのクラスのインスタンスが存在しない限り、その変数も存在しませんが、クラス変数は、そのクラスのインスタンスの存否に関わらず、プログラムの中に常に 1 つ存在しています。

プログラム中でクラス変数にアクセスする場合は、

クラス名.変数名

の形式で行います。クラス変数とインスタンス変数の名前が重なることは許されません。

同じクラスのクラス宣言の中でアクセスする場合は、同名の局所 (ローカル) 変数などと名前が重なっていなければ、クラス名. を省略することもできます[7]。

Hand.java には、final な (値を書き換えられない) クラス変数として、CAPACITY が宣言されています。クラス変数の名前のつけ方に関する慣習は、インスタンス変数や局所 (ローカル) 変数、メソッドやコンストラクタの仮引数と同じですが、final なクラス変数に関しては、すべて大文字にして単語の切れ目には _ (下線) を挟むようにします[8]。

メモ

クラスメソッド

クラスメソッドは、そのクラスに関連した何らかの手続きです。C 言語の関数を、クラスごとに分類したものと考えることができます。Hand クラスでは、数組の手札をまとめて生成し、それらを Hand 型を要素とする配列に格納し、その配列を戻り値として返すクラスメソッド createHands が宣言されています。

[6] 型名 あるいは static の前に public や private というアクセス修飾子や、final や volatile などの修飾子を書くこともできます。

[7] Hand.java の 11 行目や 37 行目では、この形でクラス変数 CAPACITY にアクセスしています。

[8] final なクラス変数は、C 言語での #define でマクロ定義された定数と同じ役割で使われます。

クラスメソッドの宣言も、Hand.java の 25 行目のように、static というキーワードを使って、

static 戻り値の型名 メソッド名 (仮引数宣言の列) メソッド本体

という書式で行います[9]。引数の数や型で区別できれば、クラスメソッドとインスタンスメソッドの名前が重なっても構いません[10]。

プログラム中でクラスメソッドを起動する場合は、

クラス名.クラスメソッド名(引数の列)

という形のメソッド起動式で行います[11]。同じクラス宣言の中で起動する場合は、クラス名. を省略することもできます。

次の P601.java は、今回修正した Hand クラスを使って、3 組の手札を配置し、シャッフルしたデッキから、それぞれにカードを配るプログラムです。

P601.java

```
 1 import jp.ac.ryukoku.math.cards.*;
 2
 3 class P601 {
 4     public static void main(String[] args) {
 5         GameFrame f = new GameFrame();
 6         Deck d = new Deck();
 7         f.add(d, 600, 100);
 8         d.shuffle();
 9         Hand[] hands = Hand.createHands(3, 70, 400, 25, 240);
10         for (int i = 0; i < Hand.CAPACITY; i++) {
11             for (Hand h : hands) {
12                 h.draw(d);
13             }
14         }
15     }
16 }
```

9 行目で、Hand クラスのクラスメソッド createHands を起動して、手札を 3 組生成しています。3 組の手札のうち、最も左の手札の位置が (70, 400)、同じ組の隣り合った手札の x 座標の差が 25、隣り合った 2 組の手札の x 座標のずれが 240 です。また、10 行目では Hand クラスのクラス変数 CAPACITY にアクセスしています。11 行目では、第 4 回の「付録: 配列のための for 文」で紹介した形の for 文が使われています。

メモ

[9] 戻り値の型名 あるいは static の前に、public や private などの修飾子を書くこともできます。

[10] つまり、多重定義されたあるメソッドにインスタンスメソッドとクラスメソッドが混在していても構いません。

[11] クラス名 の代わりに、オブジェクトを表す式 を書くこともできます。この場合、オブジェクトを表す式 の型に合わせてクラスが選ばれますが、こちらの書式はインスタンスメソッドの起動と紛らわしいのでほとんど使いません。

6.5 パッケージ

Java プログラミングでは、既存のクラスライブラリの中に用意されているクラスと、自分で定義したクラスとをうまく組み合わせてプログラム全体を実現します。Java の標準的な実行環境に含まれているクラスライブラリを含めて、世の中にはたくさんのクラスライブラリが存在しますが、これらは独立した多くのプログラマによって作成されたものですので、どうしても同じ名前のクラスが複数存在する状況 (名前の衝突) が起きてしまいます。

この問題を解決するため、Java には**パッケージ**と呼ばれるクラスを分類するための仕組みが用意されています[12]。パッケージは、ファイルを整理して置くためのディレクトリのようなものです。同じ名前のファイルでも、異なるディレクトリに置いておけば区別ができるのと同じように、同じ名前のクラスでも、異なるパッケージのクラスは別のクラスとして扱うことができます。また、ディレクトリの場合と同様に、パッケージは階層構造を持つことができ、パッケージの中にパッケージを置くことができます。

完全限定名

たとえば、この科目で使っているクラスライブラリには、`GameFrame` や `Card`、`Deck` などのクラスが含まれていますが、これらのクラスはすべて、`jp` という名前のパッケージに含まれる `ac` という名前の (サブ) パッケージに含まれる `ryukoku` という名前の (サブ) パッケージに含まれる `math` という名前の (サブ) パッケージに含まれる `cards` という名前の (サブ) パッケージに含まれるクラスとなっています。Java では、この「というパッケージに含まれる」を `.` (ピリオド) で表して、これらのクラスを、たとえば

```
jp.ac.ryukoku.math.cards.GameFrame
```

のように書いて指定します。`jp.ac.ryukoku.math.cards` がパッケージ名で[13]、`GameFrame` はそのパッケージに含まれる (単純な) クラス名です。

[パッケージ名].[単純なクラス名]

の書式を、クラスの**完全限定名**と呼びます。

メモ

[12] クラス以外に、「インタフェース」と呼ばれるものもパッケージの仕組みを用いて分類されます。インタフェースについては、この科目に引き続いて開講される「グラフィックス及び演習」という科目で勉強しますが、今回紹介するパッケージやインポート宣言に関する話は、クラスだけではなくこのインタフェースと呼ばれるものについても通用します。

[13] パッケージ名が衝突しないように、そのパッケージを作成した組織のドメイン名を逆順に並べたものをパッケージ名とする慣習があります。

パッケージ宣言

クラスを宣言する際には、そのクラスがどのパッケージのクラスなのかを、ソースプログラムの冒頭で、**パッケージ宣言**と呼ばれる次の書式で指定します。

package パッケージ名 ;

このソースファイルで宣言されたクラスは、すべて パッケージ名 で指定したパッケージのクラスとなります。

無名パッケージ

もし、ソースファイルがパッケージ宣言で始まっていない場合は、そのソースファイルで宣言されたクラスは**無名パッケージ**(unnamed package) と呼ばれる仮のパッケージのクラスとして扱われます。これまで作成してきた Java のソースプログラムは、どれもパッケージ宣言を書いていませんでした[14]ので、`Hand` クラスや `P601` クラスは、この無名パッケージのクラスとなります。

標準的な Java の環境では、`java` コマンドや `javac` コマンドを実行しているカレントディレクトリに置かれたクラスファイル群で無名パッケージが構成されます。無名パッケージはテスト用のパッケージですので、本格的なプログラムを作成する場合には、必ずパッケージ宣言を行った上でクラス等を宣言します。

メモ

6.6 インポート宣言

宣言しようとしているクラスと同じパッケージのクラスについては単純なクラス名だけで指定することができます[15]。たとえば、`P601.java` で使用している `Hand` は `P601` と同じ無名パッケージのクラスですから、9 行目の

```
Hand[] hands = Hand.createHands(3, 70, 400, 25, 240);
```

のように、`Hand` という単純なクラス名を用いて指定できます。一方、たとえば、`GameFrame` クラスは `P601` とは異なるパッケージのクラスですから、5 行目の

```
GameFrame f = new GameFrame();
```

[14] 一方、`GameFrame` クラスや `Card` クラスのソースファイルの冒頭には `package jp.ac.ryukoku.math.cards;` というパッケージ宣言が書かれています。

[15] 同じパッケージのクラス以外にも、`java.lang` というパッケージのクラスを単純な名前で指定することができます。`java.lang` パッケージには、Java プログラミングに欠かすことのできないごく基本的なクラスが含まれています。`java.lang` パッケージのいくつかのクラスについては第 8 回で勉強する予定です。

は、本来なら完全限定名を用いて

```
jp.ac.ryukoku.math.cards.GameFrame f
    = new jp.ac.ryukoku.math.cards.GameFrame();
```

と書かなければなりません。しかし、さすがにこれは大変ですので、Java では、`P601.java` の 1 行目の

```
import jp.ac.ryukoku.math.cards.*;
```

のように、**インポート宣言**と呼ばれる宣言を行うことで、パッケージ名を省略できるようにする仕組みが用意されています。

パッケージ名を省略するためのインポート宣言には次の 2 通りの書式があり[16]、ソースファイルの冒頭で (パッケージ宣言の後に) いくつでもインポート宣言を書くことができます。

```
import パッケージ名.単純なクラス名;
import パッケージ名.*;
```

1 つ目の書式の宣言では、パッケージ名.単純なクラス名 で指定されたクラスが、2 つ目の書式では、パッケージ名 のすべてのクラスが、単純なクラス名で指定できるようになります[17]。

メモ

クラス変数やクラスメソッドのインポート

次の 2 つのいずれかの書式のインポート宣言を用いると、完全限定名で指定したクラスのクラス変数やクラスメソッドを、その変数名やメソッド名だけでアクセスしたり起動したりすることもできます。

```
import static クラスの完全限定名.クラス変数名またはクラスメソッド名;
import static クラスの完全限定名.*;
```

1 つ目の書式では、クラスの完全限定名.クラス変数名またはクラスメソッド名 で指定されたクラス変数やクラスメソッドが、2 つ目の書式では、クラスの完全限定名 で指定されたにクラスの

[16]クラスを単純なクラス名で指定するためのインポート宣言には、この他にも、入れ子のクラス (クラス宣言の中で宣言されたクラス) を指定するための書式もあります。

[17]異なるパッケージの同名のクラスを 1 つ目の書式で複数インポートすることはできません。また、1 つ目の書式で指定されたクラスが、2 つ目の書式で指定される別のパッケージにも存在している場合は、1 つ目の書式が優先されます。1 つ目の書式では指定されないクラスがソースファイル中で使われていて、そのクラスが 2 つ目の書式で指定された異なる複数のパッケージに見つかる場合は、ソースファイルのコンパイルが失敗します。

すべてのクラス変数やクラスメソッドが、変数名やメソッド名のみでアクセスしたり起動したりできるようになります[18]。

次の P602.java は、このようなインポート宣言を利用したプログラムの例です。このプログラムでは、2 〜 3 行目の インポート宣言により、Suit クラスのクラス変数 SPADES (21 行目) と Rank クラスのクラスメソッド rankOf (22 行目) を、クラス名を指定せずに使っています[19]。

P602.java

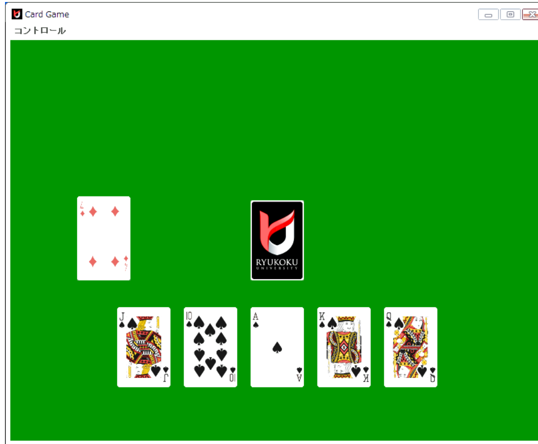
```
 1 import jp.ac.ryukoku.math.cards.*;
 2 import static jp.ac.ryukoku.math.cards.Suit.*;
 3 import static jp.ac.ryukoku.math.cards.Rank.*;
 4
 5 class P602 {
 6     public static void main(String[] args) {
 7         GameFrame f = new GameFrame();
 8         Deck d = new Deck(1);
 9         Pile p = new Pile();
10         f.add(d);
11         f.add(p, 100, 240);
12         d.shuffle();
13         Hand h = new Hand(160, 400);
14         do {
15             while (h.count() < Hand.CAPACITY) {
16                 h.draw(d);
17             }
18             int i = 0;
19             while (i < h.count()) {
20                 Card c = h.get(i);
21                 if (c.suit != SPADES || (!c.isPictureCard()
22                         && c.rank != rankOf(10))) {
23                     h.discard(i, p);
24                     continue;
25                 }
26                 i++;
27             }
28         } while (h.count() < Hand.CAPACITY);
29     }
30 }
```

このプログラムは、ジョーカー 1 枚を含むデッキから 5 枚のカードを手札に加えた後、不要な手札を何枚か捨てては、デッキからカードを引いて手札を 5 枚にする、ということを繰り返し、最終的には、次の図のように、手札をスペードの 10、ジャック、クイーン、キング、エースにします。

メモ

[18]同名のクラス変数やクラスメソッドが異なるクラスに存在する場合の扱いは、通常のインポート宣言で、同名のクラスが異なるパッケージに存在する場合と同様です。

[19]Hand は無名パッケージのクラスですので、インポート宣言を使って、15 行目や 28 行目の Hand.CAPACITY の Hand. を省略することはできません。



static 付きのインポート宣言は、どのクラスのクラス変数やクラスメソッドを使っているのかを分かり難くしてしまいますので、安易にこれを使うことは推奨されません。P602.java の場合、Suit.SPADES や Rank.rankOf(···) の、たった 2 個所が簡単になるだけですので、本来は static 付きのインポート宣言を使うべきではありません。

6.7　演習問題

1. P601.java で実行されている

```
10      for (int i = 0; i < Hand.CAPACITY; i++) {
11          for (Hand h : hands) {
12              h.draw(d);
13          }
14      }
```

という for 文に相当する仕事を行う、次のようなクラスメソッド fillHands の定義を Hand クラスに追加しなさい。

static void fillHands(Hand[] hands, Deck d)	配列 hands に含まれる手札の組にデッキ d から 1 枚ずつカードを配って、配列中のすべての手札の組を 5 枚にする。ただし、デッキが空になったらカードを配るのを止める。

このように Hand クラスを変更できたら、P601.java の 10 行目から始まる for 文を

```
Hand.fillHands(hands, d);
```

に書き換えて、P601.java が同じように動作するかテストしてみましょう。

2. P602.java で実行されている

```
15      while (h.count() < Hand.CAPACITY) {
16          h.draw(d);
17      }
```

という while 文に相当する仕事を行う、次のようなインスタンスメソッド fill の定義を Hand クラスに追加しなさい。

void fill(Deck d)	手札が5枚になるまで、デッキ d からカードを引く。ただし、デッキが空になったらカードを引くのを止める。

このように Hand クラスを変更できたら、P602.java の 15 行目から始まる while 文を

```
h.fill(d);
```

に書き換えて、P602.java が同じように動作するかテストしてみましょう。

3. import 宣言を使用しないように P602.java を書き換えたプログラム P603.java を作成しなさい。

オブジェクト指向及び演習・第6回・終わり