オブジェクト指向及び演習・第 7 回

2023 年 5 月 22 日

今回の内容

7.1 サブクラスとスーパークラス

Hand クラスの add メソッドや draw メソッドは、右端にカードを追加していきますので、手札に加えた順に、カードが左から右へ並ぶことになります[1]。これはこれでよい場合もありますが、トランプを使ったゲームの場合、手札が決まった順[2]に整列されている方がよい場合もあります。

add や draw を起動すると、新しいカードを手札が整列した状態になるような位置に挿入してくれるようなクラスがあると便利です。Hand クラスをそのような動作をするように変更することも考えられますが、すでに Hand クラスを使っているプログラムがある場合は、そのプログラムが期待している Hand クラスの挙動を変えてしまうことになりますので、そういう訳にも行きません。また、Hand クラスのように、自分が作成したクラスならよいのですが、これが既存のクラスライブラリの一部になっている場合など、そもそも変更することができないこともあります。このため、Hand とは別の新しいクラスとして定義する必要が出てきます。

しかし、インスタンス変数や、コンストラクタ、add や draw 以外のインスタンスメソッドの宣言は同じですから、このような新しいクラスを一から定義するのは非効率です。そこで、多くのオブジェクト指向言語には、既存のクラスの一部分を修正して、新しいクラスを定義する仕組みが用意されています。

次のプログラム SortedHand.java は、Java のこのような仕組みを利用して、Hand クラスを元に作成した SortedHand というクラスの宣言です。このクラスのインスタンスメソッド add や draw は、カードのランクが弱い順に手札を並べます[3]。

SortedHand.java

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

[1] deltaX の値が負の場合は、右から左になります。

[2] たとえば、カードのスートやランクの強さの順です。

[3] カードのランクの強さは、弱いものから順に、2、3、4、···、10、ジャック、クイーン、キング、エース、ジョーカーです。

```
 2
 3 /* SortedHand クラスのクラス宣言 (Hand のサブクラス) */
 4 class SortedHand extends Hand {
 5     /* コンストラクタの宣言 (コンストラクタは継承されない) */
 6     SortedHand(int x, int y, int deltaX) {
 7         super(x, y, deltaX);  // スーパークラスのコンストラクタの起動
 8     }
 9
10     SortedHand(int x, int y) {
11         super(x, y);          // スーパークラスのコンストラクタの起動
12     }
13
14     /* 新しいクラスメソッド priority の宣言 */
15     static int priority(Card c) {
16         if (c.isJoker()) {
17             return c.getNumber();
18         }
19         return (c.rank.getNumber() + 11) % 13 * 4
20                     + (4 - c.suit.getNumber());
21     }
22
23     /* インスタンスメソッド add の再定義 */
24     void add(Card c) {
25         if (numCards >= CAPACITY) {
26             return;
27         }
28         super.add(c);         // Handクラスの add の起動
29         int i, last = numCards - 1;
30         /* 挿入位置の決定 */
31         for (i = 0; i < last; i++) {
32             if (priority(c) < priority(cards[i])) {
33                 break;
34             }
35         }
36         if (i == last) {
37             return;
38         }
39         /* 挿入位置以降を右へずらす */
40         c.moveTo(x + i * deltaX, y - 130);
41         for (int j = last; i < j; j--) {
42             cards[j] = cards[j - 1];
43             cards[j].moveTo(x + j * deltaX, y);
44         }
45         cards[i] = c;
46         c.moveTo(x + i * deltaX, y);
47     }
48 }
```

メモ

すでに定義されているクラスを元に、新しいクラスを宣言するには、

```
class 新しく定義するクラス名 extends 元になるクラス名 {
    ⋮
}
```

のように、extends というキーワードを使って宣言します[4]。この extends 元になるクラス名 の部分を extends **節**と呼びます。また、元となったクラスを新しく定義されるクラスの (直接の)**スーパークラス**と呼び、新しく定義されるクラスを元となったクラスの (直接の)**サブクラス**と呼びます。たとえば、Hand は SortedHand の直接のスーパークラスであり、SortedHand は Hand の直接のサブクラスです。

あるクラス C を元にして定義されたクラス D があるとき、そのクラス D を元にして、さらに新しいクラス E を定義することもできます。このとき、C は D の、D は E のそれぞれ直接のスーパークラスですが、C は E の (直接ではない) スーパークラスとなります。サブクラスに関しても同様で、E は C の (直接ではない) サブクラスとなります。

メモ

7.2 継承

クラス宣言中のインスタンス変数、クラス変数、インスタンスメソッド、クラスメソッドの宣言は、そのクラスのサブクラスに引き継がれます[5]。このことを**継承**と呼びます。たとえば、Hand クラスで宣言されている x、y、deltaX、numCards、cards というインスタンス変数は、SortedHand クラスに継承されますので、SortedHand クラスの各インスタンスもこれらの変数を持つことになります。また、draw や count、get、discard というインスタンスメソッドも同様で、SortedHand の各インスタンスはこれらの仕事を同じ定義にしたがって行います[6]。

CAPACITY というクラス変数や createHands というクラスメソッドについても同様です。それぞれ、SortedHand.CAPACITY や SortedHand.createHands(···) のようにアクセスしたり起動したりすることができます。クラス変数については、サブクラス向けに同名の変数が別に用意されるのではなく、スーパークラスのクラス変数が、サブクラスのクラス変数としてもアクセスできるだ

[4] クラス自身が final という修飾子付きで宣言されているクラスは 元になるクラス名 となることができません。

[5] ただし、アクセス修飾子の設定 (ない場合も含む) によって、スーパークラスでの宣言がサブクラスからは見えない場合は継承されません。たとえば、スーパークラスで private というアクセス修飾子付きで宣言されているものはサブクラスへ継承されません。アクセス修飾子については「付録:アクセス修飾子」を参照してください。

[6] add については、SortedHand のクラス宣言中に、同じメソッド名で同じ数と型の引数を持つメソッドが宣言されているため継承されません。

けですので注意が必要です。つまり、SortedHand.CAPACITY と Hand.CAPACITY は、どちらも同じ 1 つの変数を意味します。

メモ

7.3 サブクラスでの変数やメソッドの宣言

SortedHand.java の 15 行目からは、priority という新しいクラスメソッドが、24 行目からは、Hand でも定義されていた add というインスタンスメソッドが宣言されています。この add の宣言では、引数として渡されたカードを手札がランクの強さの順になるような位置に追加しています。この際、ランクの強さの計算に、クラスメソッド priority が使用されています。

サブクラスの宣言内で、インスタンス変数、インスタンスメソッド、クラス変数、クラスメソッドを宣言すると、これらの宣言が、スーパークラスから継承された宣言に追加されます。変数は名前で、メソッドは名前と引数の数や型で区別されますが、これらがスーパークラスから継承しているものと重なる場合は注意が必要です[7]。

メモ

7.4 インスタンスメソッドの再定義 (オーバーライド)

SortedHand クラスで宣言されている add メソッドのように、引数の数や型が等しい同名のインスタンスメソッドがスーパークラスに宣言されている場合は、スーパークラスでの宣言は継承されずに、サブクラスの宣言で上書きされます。これをインスタンスメソッドの**再定義**、あるいは **オーバーライド (overriding)** と呼びます。

インスタンスメソッドの再定義 (オーバーライド) はオブジェクト指向のプログラミングの重要な仕組み[8]の 1 つで、同じ数と型の引数を伴って同じ名前のインスタンスメソッドを起動しても、ど

[7] Java では、ある変数とあるメソッドが同じ名前を持っていても、これら 2 つは区別されますので、継承した変数と同じ名前を持つメソッドを宣言したり、継承したメソッドと同じ名前を持つ変数を宣言することは (プログラムが分かり難くなってしまいますが) 可能です。

[8] スーパークラスから継承しているインスタンス変数やクラス変数と同名の変数を宣言したり、スーパークラスから継承しているクラスメソッドと同じ名前で同じ引数の数や型をもつメソッドを宣言したりした場合とは異なる効果があります。これらの場合については、「付録:インスタンス変数やクラス変数、クラスメソッドの隠蔽」を参照してください。

のオブジェクトに対して起動したのかによって、使われるメソッド定義が変わるという仕組み (後述の多態性) を作ります。

7.5 super によるスーパークラスでの定義の利用

インスタンスメソッドの再定義を行う場合、新しい定義の中で、古い (スーパークラスでの) 定義のメソッドを起動したくなることがよくあります。サブクラスの宣言中で、上書きされる前のメソッドの定義を起動する場合は、SortedHand.java の 28 行目のように、this の代りに super というキーワードを使い、

super.〔インスタンスメソッド名〕(〔引数の列〕)

という形のメソッド起動式を使います。

メモ

7.6 サブクラスのコンストラクタ

SortedHand.java の 6 行目からは、このクラスのコンストラクタを宣言しています。コンストラクタはスーパークラスからサブクラスへ継承されませんので、サブクラスには独自のコンストラクタを宣言する必要があります[9]。とは言っても、サブクラスのインスタンスは、スーパークラスのインスタンスを拡張したものとなりますから、スーパークラスのコンストラクタで行っていた初期化作業は、サブクラスのインスタンスについてもやはり必要です。そこで、Java では、サブクラスのコンストラクタが作業を始める前に、必ずスーパークラスのコンストラクタを起動して、拡張される前の部分についての初期化を行う決まりになっています。

スーパークラスのコンストラクタは、サブクラスのコンストラクタ本体の先頭で、super というキーワードを用いて、

```
〔クラス名〕(〔仮引数宣言の列〕) {
    super(〔スーパークラスのコンストラクタへ渡す引数の列〕);
     ⋮
}
```

のように起動します。これは、this というキーワードを使って同じクラスの別のコンストラクタを起動するときと同じ形です。

Java では、コンストラクタ本体の先頭の文が、super(···); の形でも this(···); の形でもない場合は、そこに

[9] コンストラクタを 1 つも宣言しないと、デフォルトコンストラクタが補われます。

```
super();
```

という文が補われます[10]ので、結局、どのコンストラクタが起動された場合でも、まずスーパークラスのコンストラクタが起動されてから、その後、(サブクラスの) コンストラクタでの作業が行われることになります[11]。

メモ

7.7 サブクラスの利用

次のプログラム P701.java は、SortedHand クラスを利用して、シャッフルしたデッキから、1 枚ずつカードを引いて手札に加え、手札を 5 枚にするプログラムです。

P701.java

```
 1 import jp.ac.ryukoku.math.cards.*;
 2
 3 class P701 {
 4     public static void main(String[] args) {
 5         GameFrame f = new GameFrame();
 6         Deck d = new Deck();
 7         f.add(d);
 8         d.shuffle();
 9         Hand h = new SortedHand(160, 400);
10         for (int i = 0; i < Hand.CAPACITY; i++) {
11             h.draw(d);
12         }
13     }
14 }
```

Hand のサブクラスである SortedHand のインスタンスは、Hand のインスタンスの持つインスタンス変数やインスタンスメソッドを (一部は再定義されていますが) すべて持っています。このため、SortedHand のインスタンスは、Hand のインスタンスとしても仕事をすることが可能です。一般に、サブクラスのインスタンスは、スーパークラスのインスタンスとしても働くことができるため、プログラム中で、サブクラスのインスタンスをスーパークラスのインスタンスであるかのように扱うことができます。たとえば、P701.java の 9 行目では、new SortedHand(160, 400) というイ

[10] この場合、直接のスーパークラスが、引数のないコンストラクタの宣言 (前回説明したデフォルトコンストラクタでも構いません) を持っていなければなりません。また、super(); は、デフォルトコンストラクタに対しても補われますので、デフォルトコンストラクタを省略なしに書くと、public クラス名 { super(); } となります。

[11] 前回紹介した this(···); による別のコンストラクタの起動が、コンストラクタ本体の先頭にしか許されないのはこのためです。途中で this(···); を実行すると、別のコンストラクタからスーパークラスのコンストラクタが起動されます。this(···); を実行したコンストラクタの先頭で、スーパークラスのコンストラクタを起動すると重複してしまいますし、起動しなければ、this(···); に至るまでの部分では、スーパークラスのコンストラクタによる初期化が完了していない状態で、サブクラスのコンストラクタが作業を始めてしまいます。

ンスタンス生成式で生成した SortedHand クラスのインスタンスを、Hand 型の変数 h に代入し、それ以降では、このインスタンスを Hand クラスのインスタンスであるかのように扱っています。

メモ

7.8 動的ディスパッチと多態性

インスタンスメソッドが起動されるときに選ばれるメソッドの定義は、その式が評価される時、起動の対象となっているオブジェクトがどのクラスのインスタンスであるかによって決定されます[12]。このため、実際にプログラムが実行されてみないと、どのメソッド宣言が使われるかは決まりません。たとえば、

```
Hand h;
```

のように宣言された変数 h を使って、

```
h.add(new Card());
```

のようなメソッド起動式を書いたとすると、選択される add の定義は、Hand クラスで宣言されているものとは限りません。このメソッド起動式が評価される時点で、変数 h に代入されているオブジェクトが Hand クラスであれば Hand クラスの add の定義が、SortedHand クラスであれば SortedHand クラスの add の定義が選択されます。

このように、メソッドの定義が実行時に選択されることを、一般に、**動的ディスパッチ (dynamic dispatch)** と呼びます[13]。また、オブジェクトに対して同じ指示をしても (同じメソッド起動式でも)、そのオブジェクト (のクラス) ごとに独自の振る舞いが行われることを、オブジェクト指向プログラミングでは**多態性 (polymorphism)** と呼びます[14]。

P701.java の 11 行目では、draw ソッドが起動されていますが、このメソッドは SortedHand クラスでは宣言されていませんので、スーパークラスである Hand から継承した定義が使われます。つまり、Hand クラスの draw の宣言に従ってメソッド本体が実行されます。その宣言には、たとえば

```
void draw(Deck d) {
    if (d.count() > 0 && numCards < CAPACITY) {
        Card c = d.pickUp();
```

[12] このただ 1 つの例外は、super を使ってインスタンスメソッドを起動する場合です。この場合は、起動の対象となっているオブジェクトのクラスに関わらず、そのメソッド起動式が書かれたクラスの直接のスーパークラスの定義が使われます。

[13] 厳密に言うと異なる意味ですが、動的束縛 (dynamic binding) あるいは、遅延束縛 (late binding) という言葉を使うこともよくあります。

[14] Polymorphism の訳語として、多態性の他にも、多相性や多様性という言葉を使うこともあります。

```
            this.add(c);
            c.faceUp();
        }
    }
```

のように、add メソッドのメソッド起動式が含まれています[15]。この draw は Hand クラスで宣言されているインスタンスメソッドですから、このメソッド起動式の this は Hand 型の式であり、Hand クラスで宣言されている add メソッドの定義が使われそうに見えますが、そうではありません。このメソッド起動式が評価される時の this は SortedHand クラスのインスタンスを指していますから、SortedHand クラスで宣言された新しい add の定義が用いられます。

このように、サブクラスでインスタンスメソッドを再定義することによって、スーパークラスから継承したメソッドの挙動が影響を受けることがあります。SortedHand クラスでは、これが有効に働いていますが、一般に、サブクラスでインスタンスメソッドを再定義する場合には十分な注意が必要となります。また、逆に、スーパークラスの側から見ると、各メソッドを定義する際には、そこで利用しているインスタンスメソッドが (サブクラスで) 再定義されることを想定しておかないといけないことになります。

メモ

7.9 演習問題

1. Hand クラスに次のような 2 つのコンストラクタを追加しなさい。その際、元からあるコンストラクタで生成されるインスタンスの挙動は変わらないようにしなさい。

コンストラクタ	
`Hand(int x, int y, int deltaX, boolean open)`	open が true であれば、Hand(x, y, deltaX) と同じ。false の場合は、draw を起動したときに、カードを表向きにしないでそのまま手札に追加するようなインスタンスを生成する。
`Hand(int x, int y, boolean open)`	open が true であれば、Hand(x, y) と同じ。false の場合は、draw を起動したときに、カードを表向きにしないでそのまま手札に追加するようなインスタンスを生成する。

また、SortedHand クラスにも同様のコンストラクタ (2 つ) を追加しなさい。

[15] 前回の演習問題で、そのように draw を定義しました。

コンストラクタ	
`SortedHand(int x, int y, int deltaX, boolean open)`	同じ仮引数を持つ Hand クラスのコンストラクタと同様。
`SortedHand(int x, int y, boolean open)`	同じ仮引数を持つ Hand クラスのコンストラクタと同様。

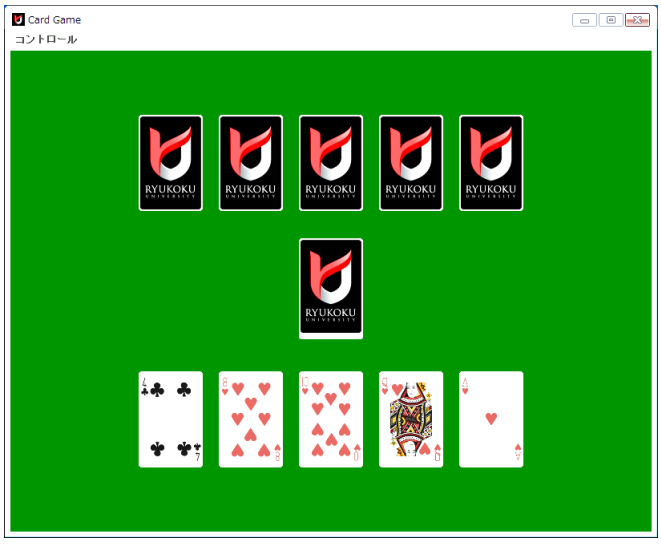
Hand クラスに boolean 型のインスタンス変数を追加し、上の 2 つのコンストラクタに渡された引数 open の値を、そのインスタンス変数で記憶しておくようにしましょう。インスタンスメソッド draw の定義は、このインスタンス変数の値に応じて、デッキから引いたカードを表向きに手札に加えるか、そのまま加えるかを切り替えるように修正します。SortedHand クラスについては、表のようなコンストラクタを 2 つ追加するだけで済むはずです。

このように Hand クラスや SortedHand クラスを変更できたら、次の P702.java を実行してみましょう。

P702.java

```
 1 import jp.ac.ryukoku.math.cards.*;
 2
 3 class P702 {
 4     public static void main(String[] args) {
 5         GameFrame f = new GameFrame();
 6         Deck d = new Deck();
 7         f.add(d);
 8         d.shuffle();
 9         Hand[] hands = {
10             new SortedHand(160, 400),
11             new SortedHand(160, 80, false)
12         };
13         Hand.fillHands(hands, d);
14     }
15 }
```

このプログラムは、表向きと裏向きと 2 組の手札 (SortedHand クラスのインスタンス) を生成して、シャッフルされたデッキからカードを配り、次の図のような状態にします。



2. 次の表にあるようなコンストラクタとインスタンスメソッドをもつクラス MovableHand を、SortedHand クラスのサブクラスとして定義しなさい。

MovableHand クラス — 移動可能な手札の組 (SortedHand を拡張)

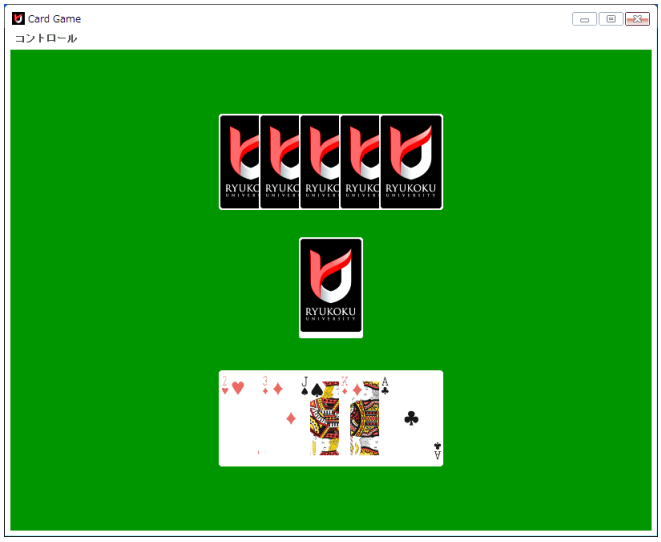
コンストラクタ	
`MovableHand(int x, int y)`	最初 (添字 0) の手札が (x, y) の位置に、また、隣り合った手札の x 座標の差が deltaX となるような手札の組。draw を起動すると表向きに手札に追加する。
`MovableHand(int x, int y, boolean open)`	最初 (添字 0) の手札が (x, y) の位置に、また、隣り合った手札の x 座標の差が deltaX となるような手札の組。open が true なら、draw は、カードを表向きにして手札に追加するが、false ならそのままの向きで追加する。
インスタンスメソッド (SortedHand のメソッドを継承)	
`void relocate(int x, int y, int deltaX)`	この手札の組におけるカードの配置方法を、最初 (添字 0) の手札が (x, y) の位置となり、隣り合った手札の x 座標の差が deltaX となるようなものに変更する。すでにある手札は、新しい配置方法に基づいて配置し直す。

注意: relocate の後に、discard や draw などをした場合、新しい配置方法に基づいてカードが配置されないといけません。

このようなクラス MovableHand が定義できたら、次のプログラム P703.java を実行してみましょう。このプログラムは、P702.java と同じ動作をした後、最後に 2 組の手札を配置し直して、図のような状態にします。

P703.java

```
 1 import jp.ac.ryukoku.math.cards.*;
 2
 3 class P703 {
 4     public static void main(String[] args) {
 5         GameFrame f = new GameFrame();
 6         Deck d = new Deck();
 7         f.add(d);
 8         d.shuffle();
 9         MovableHand[] hands = {
10             new MovableHand(160, 400),
11             new MovableHand(160, 80, false)
12         };
13         Hand.fillHands(hands, d);
14         for (MovableHand h : hands) {
15             h.relocate(h.x + 100, h.y, 50);
16         }
17     }
18 }
```



この実行例のように、より左のカードがより下側になるように手札のカードを重ねるためには、左のカードから順に、`raise` メソッドを起動しましょう。`raise` メソッドを使うと、カードの重なり順を最上位に変更することができます。

オブジェクト指向及び演習・第 7 回・終わり

7.10 付録: インスタンス変数、クラス変数、クラスメソッドの隠蔽

スーパークラスから継承したインスタンスメソッドをサブクラスで再定義すると、継承した定義は上書きされてしまいますが、同様なことを、インスタンス変数、クラス変数、クラスメソッドに対して行うと、その効果はインスタンスメソッドの場合とはかなり異なります。

スーパークラスから継承したインスタンス変数と同じ名前のインスタンス変数を (その直接の) サブクラスで宣言すると、継承したものとは別に、同じ名前の変数が用意されます。継承した方の変数は依然として存在しますが、サブクラスで宣言した同名の変数で隠されてしまいアクセスできなくなってしまいます。このことを (インスタンス変数の)**隠蔽 (hiding)** と呼びます。

隠蔽されてしまったインスタンス変数は、この (隠蔽を起こしたサブ) クラスの宣言以外からはアクセスできません。このサブクラスのコンストラクタやインスタンスメソッドの宣言の中に限り、`super.`インスタンス変数名 という書式で、`this` が持っているはずの、この隠されたインスタンス変数にアクセスすることが許されます。

スーパークラスから継承したクラス変数と同じ名前の変数を宣言したり、継承したクラスメソッドと同じ名前で同じ引数の数や型をもつクラスメソッドを宣言した場合も同様で、これを (クラス変数やクラスメソッドの) 隠蔽と呼びます[16]。これらの場合、スーパークラス名.クラス変数名 や スーパークラス名.クラスメソッド名(···) のように書くことで、いつでも隠されてしまったクラス変数にアクセスしたり、隠されてしまったクラスメソッドを起動したりすることが可能です。

7.11 付録: アクセス修飾子

クラス宣言の中に書かれる、コンストラクタ、インスタンス変数、クラス変数、インスタンスメソッド、クラスメソッドなどの宣言には、`public`、`protected`、`private` の 3 通りの修飾子を付けることができます。これらは**アクセス修飾子**と呼ばれ、プログラム中で、その宣言にアクセスすることのできる範囲を設定します。宣言にアクセスできる範囲は、アクセス修飾子を書かない場合を含めて、次の 4 通りに設定されます。範囲の広い順に、`public`、`protected`、アクセス修飾子なし、`private` です[17]。

`public`　このアクセス修飾子を付けて宣言されたコンストラクタや変数、メソッドなどは、プログラムのどこからでもアクセスできます[18]。同じクラス宣言の中からはもちろん、異なるパッケージに属するクラス宣言の中など、どのクラス宣言の中からでもアクセスできます。

`protected`　このアクセス修飾子が付けられた場合、その宣言を含むクラスと同じパッケージに属するクラスと、その宣言を含むクラスのサブクラスのクラス宣言の中でアクセスできます。サブクラスの場合は、異なるパッケージに属していても構いません。

---

[16] 継承したクラスメソッドをインスタンスメソッドとして再定義したり、継承したインスタンスメソッドをクラスメソッドで隠蔽したりすることはできません。

[17] `protected` の方が「アクセス修飾子なし」よりアクセスできる範囲が広い (制限が緩い) ことに注意してください。

[18] `public` は、ソースファイルのトップレベルに書かれるクラス宣言自身にも付けることができ、そのクラス名をプログラムのどこでも使用できることを示します。

アクセス修飾子なし　宣言にアクセス修飾子が付けられていない場合は、その宣言を含むクラスと同じパッケージ内のクラス宣言の中でのみアクセス可能となります[19]。

`private`　このアクセス修飾子が付けられて宣言されたコンストラクタや変数、メソッドは、その宣言を囲んでいるクラス宣言の中でのみアクセスできます。

[19] クラス宣言自身がアクセス修飾子なしで宣言されている場合、そのクラス名は同じパッケージに属するクラスのクラス宣言の中だけで使用することができます。