グラフィックス及び演習・第 6 回

2025 年 7 月 21 日

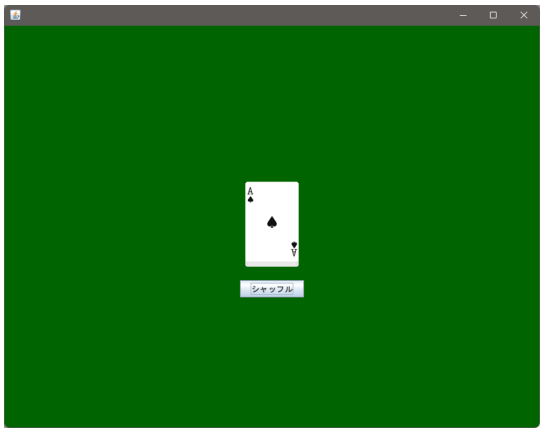
配布資料の内容

6.1 入れ子クラス

Java 言語によるオブジェクト指向のプログラミングでは、オブジェクトに特定の仕事を行わせるために、その設計図としてクラスを定義します。これまで見てきた Java プログラムでは、基本的にソースファイルのトップレベルでのみクラスを宣言していました。Java では、ソースファイルのトップレベル以外でも、いろいろなところでクラス宣言を行って、それを利用することができるようになっています。ソースファイルのトップレベルではなく、他のクラス宣言の内部で宣言されたクラスを**入れ子クラス (nested class)**と呼びます。今回は、この入れ子クラスと呼ばれるいくつかのクラス宣言の形について勉強します。入れ子クラスには、**メンバクラス**、**ローカルクラス**、**匿名クラス**の 3 種類があります。

メモ

例題プログラム　いろいろな入れ子クラスを紹介するための例題として、次の `G601.java` というプログラムについて考えます。このプログラムは、右の図のようにゲーム盤の中央にデッキを表向きに置き、そのすぐ下に追加したボタンをクリックすることで、デッキがシャッフルされるようにしたものです。このプログラムは、第 3 回で紹介した `G301.java` と同じ仕事を行います。



G601.java

```
1 import javax.swing.*;
2 import jp.ac.ryukoku.math.cards.*;
3
```

```
 4 class G601Panel extends GamePanel {
 5     Deck deck = new Deck();
 6     JButton button = new JButton("シャッフル");
 7
 8     G601Panel() {
 9         deck.flip();
10         add(deck);
11         add(button, 352, 380);
12         button.addActionListener(new ShuffleButtonHandler(deck));
13     }
14 }
15
16 class G601 implements Runnable {
17     public void run() {
18         JFrame f = new JFrame();
19         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20         f.add(new G601Panel());
21         f.pack();
22         f.setVisible(true);
23     }
24
25     public static void main(String[] args) {
26         SwingUtilities.invokeLater(new G601());
27     }
28 }
```

このプログラムでは、ボタン (JButton) がクリックされたときに発生する ActionEvent のイベントハンドラとして、次のように宣言された ShuffleButtonHandler クラス[1]のインスタンスを生成して使用しています (12 行目)。

ShuffleButtonHandler.java

```
 1 import java.awt.event.*;
 2 import jp.ac.ryukoku.math.cards.*;
 3
 4 class ShuffleButtonHandler implements ActionListener {
 5     Pile p;
 6
 7     ShuffleButtonHandler(Pile p) {
 8         this.p = p;
 9     }
10
11     public void actionPerformed(ActionEvent e) {
12         p.shuffleAsync();
13     }
14 }
```

メモ

[1]第 3 回の 6 ページのプログラムと同じものです。

6.2 メンバクラス

G601.java の G601Panel クラスでは、全く独立に宣言した ShuffleButtonHandler クラスを使用していますが、このクラスを G601Panel クラスだけが利用するのなら、むしろ G601Panel のクラス宣言の中で宣言する方が自然です。

Java では、クラス宣言の {} の直下に、つまりそのクラスのコンストラクタ、インスタンス変数、インスタンスメソッド、クラス変数、クラスメソッドなどと同じレベルで、別のクラスを宣言することができます。このように宣言されたクラスは、その宣言を含んでいる (外側の) クラスの**メンバクラス**と呼ばれます。

メモ

次のクラス宣言は、ShuffleButtonHandler クラスを、G601Panel クラスのメンバクラスとして宣言したものです。

ShuffleButtonHandler をメンバクラスとした G601Panel の宣言

```
 1 import java.awt.event.*;
 2 import javax.swing.*;
 3 import jp.ac.ryukoku.math.cards.*;
 4
 5 class G601Panel extends GamePanel {
 6     static class ShuffleButtonHandler implements ActionListener {
 7         Pile p;
 8
 9         ShuffleButtonHandler(Pile p) {
10             this.p = p;
11         }
12
13         public void actionPerformed(ActionEvent e) {
14             p.shuffleAsync();
15         }
16     }
17
18     Deck deck = new Deck();
19     JButton button = new JButton("シャッフル");
20
21     G601Panel() {
22         deck.flip();
23         add(deck);
24         add(button, 352, 380);
25         button.addActionListener(new ShuffleButtonHandler(deck));
26     }
27 }
```

ShuffleButtonHandler.java で宣言されていた ShuffleButtonHandler クラスの宣言が、そのままの形で、このプログラムの 6 ～ 16 行目に取り込まれていることに注意してください。新しい

プログラムの 18 行目以降は変っていません。

6 行目の先頭の static は、ここでメンバクラスとして宣言された ShuffleButtonHandler のインスタンスが、G601Panel クラスのインスタンスとは独立に存在することができることを示しています。このように static という修飾子を伴って宣言されたメンバクラスは、通常のクラス宣言で定義されるクラスと特に違いはありません。たとえば、G601Panel の static なメンバクラス ShuffleButtonHandler のインスタンスも、ShuffleButtonHandler.java のトップレベルで宣言された ShuffleButtonHandler クラスのインスタンスも同じように働くことができます。

唯一つの違いは、メンバクラスを、それを含んでいるクラス宣言の外から参照する際には、クラス変数やクラスメソッドを使用するときのように、

[そのメンバクラスを含むクラス名].[メンバクラス名]

という形で指し示さなければならない点です[2]。たとえば、25 行目は G601Panel の宣言の中ですので、単に、ShuffleButtonHandler という単純なクラス名で指し示すことができていますが、もし、この文が G601Panel のクラス宣言の外側に書かれているのなら、

```
button.addActionListener(new G601Panel.ShuffleButtonHandler(deck));
```

のように書かないといけません[3]。

メモ

6.3 内部クラス

ボタンがクリックされた際に、ShuffleButtonHandler クラスのインスタンスがデッキをシャッフルするためには、どのデッキをシャッフルすればよいのか記憶しておく必要があります。このため、ShuffleButtonHandler は、Pile p; と宣言されたインスタンス変数を持っていて、コンストラクタの引数として渡されたデッキを、このインスタンス変数で記憶するようにしています。

G601Panel のメンバクラス ShuffleButtonHandler のインスタンスの場合、G601Panel クラスの特定のインスタンスのために働いていますので、シャッフルする対象となるデッキを記憶する代りに、自分が担当している G601Panel クラスのインスタンスを記憶しておき、次のようなプログラムとすることもできるはずです (変更された行を * 印で示してあります)。

---

[2] メンバクラスを完全限定名で指し示す場合は、この前に、さらに [パッケージ名]. が付くことになります。メンバクラスの宣言には、インスタンスメソッドなどの宣言と同様に、public、protected、private などのアクセス修飾子を付けることもできます。そのメンバクラスが、そのメンバクラスを含んでいるクラスの外側からアクセス可能かどうかは、これらのアクセス修飾子の設定によって変わります。

[3] G601Panel のクラス宣言の内部であっても、こう書いて構いません。

メンバクラス ShuffleButtonHandler の宣言 (その 2)

```
  ⋮
  5 class G601Panel extends GamePanel {
  6     static class ShuffleButtonHandler implements ActionListener {
* 7         G601Panel panel;
  8
* 9         ShuffleButtonHandler(G601Panel panel) {
*10             this.panel = panel;
 11         }
 12
 13         public void actionPerformed(ActionEvent e) {
*14             panel.deck.shuffleAsync();
 15         }
 16     }
 17
 18     Deck deck = new Deck();
 19     JButton button = new JButton("シャッフル");
 20
 21     G601Panel() {
 22         deck.flip();
 23         add(deck);
 24         add(button, 352, 380);
*25         button.addActionListener(new ShuffleButtonHandler(this));
 26     }
 27 }
```

こちらの ShuffleButtonHandler の宣言では、デッキ (Pile) の代わりに G601Panel のインスタンスを覚えるようにしています (7 行目)。これに伴い 25 行目では、デッキの代わりに G601Panel のインスタンス自身をコンストラクタに渡しています。

この例のように、そのクラスのインスタンスのために何らかの仕事をするようなオブジェクトのクラスとして、そのメンバクラスが定義されることがよくあります。その際、メンバクラスのオブジェクトは、自分はどのインスタンスの仕事を担当しているのかを記憶しているのが普通です。

Java では、static という修飾子を付けずにメンバクラスを宣言することで、メンバクラスのインスタンスを、そのメンバクラスを宣言しているクラスのインスタンスに自動的に関係付けることができます。このようなメンバクラス宣言を利用すると、次のプログラムのように、簡潔に G601Panel とそのメンバクラス ShuffleButtonHandler を宣言することができます (変更された行を * 印で示してあります)。

内部クラスとしてのメンバクラス ShuffleButtonHandler の宣言

```
  ⋮
  5 class G601Panel extends GamePanel {
* 6     class ShuffleButtonHandler implements ActionListener {
  7         public void actionPerformed(ActionEvent e) {
* 8             deck.shuffleAsync();
  9         }
 10     }
 11
 12     Deck deck = new Deck();
 13     JButton button = new JButton("シャッフル");
 14
 15     G601Panel() {
 16         deck.flip();
```

```
 17          add(deck);
 18          add(button, 352, 380);
*19          button.addActionListener(new ShuffleButtonHandler());
 20      }
 21 }
```

このプログラムの 6 ～ 10 行目のように、static なしに宣言されたメンバクラスのインスタンスは、そのメンバクラスを宣言している (外側の) クラスのインスタンスに従属した形でしか存在できません。特定のクラスのインスタンスに従属した形でしか存在できないようなオブジェクトのクラスを、Java では一般に**内部クラス (inner class)**と呼びます。

メモ

外側のインスタンス　内部クラスのインスタンスが従属しているオブジェクトのことを、そのインスタンスの**外側のインスタンス (enclosing instance)**と呼びます。内部クラスのインスタンスは、言わば、その上司である「外側のインスタンス」の仕事の一部を担当する部下のようなものです。

内部クラスのインスタンスを生成するには、まず、その上司となる「外側のインスタンス」が存在していなければなりません。内部クラスのインスタンスが生成される際には、明示的にあるいは暗黙の内に、その外側のインスタンスが指定されることになります。このとき、生成される (内部クラスの) インスタンスには、自動的に名前のないインスタンス変数が 1 つ追加され、そこに指定された「外側のインスタンス」が記憶されます。

static でないメンバクラスは内部クラスとなりますので、そのインスタンスは、そのメンバクラスを宣言している (外側の) クラスのインスタンスがなければ生成することができません。その外側のクラスのコンストラクタやインスタンスメソッドの中で、このようなメンバクラスのインスタンスを生成すると、このとき this で表されるオブジェクトを「外側のインスタンス」として、メンバクラスのオブジェクトが生成されます。

先のプログラムの 19 行目では、内部クラス ShuffleButtonHandler のインスタンスを生成していますが、ここは G601Panel のコンストラクタの中ですので、生成されたばかりの G601Panel クラスのインスタンスを「外側のインスタンス」として、内部クラス ShuffleButtonHandler のインスタンスが生成されます。ShuffleButtonHandler のインスタンスは、自動的に、隠れたインスタンス変数で、その「外側のインスタンス」である G601Panel のインスタンスを記憶しますので、メンバクラス ShuffleButtonHandler の宣言からインスタンス変数 panel の宣言がなくなってしまっています。また、そのインスタンス変数を初期化する必要もなくなったため、特にコンストラクタは宣言されていません[4]。

[4] ShuffleButtonHandler クラスはデフォルトコンストラクタのみを持つことになります。

内部クラスでは、そのクラスのインスタンス変数やインスタンスメソッドに加えて、その「外側のインスタンス」のインスタンス変数やインスタンスメソッドに直接アクセスすることが可能です。このため、8 行目では、あたかも deck が ShuffleButtonHandler クラスのインスタンス変数であるかのように、この変数の値を参照しています。

メモ

6.4 ローカルクラス

static でないメンバクラスを宣言する方法以外にも、内部クラスを宣言する方法があります。内部クラスは、コンストラクタやメソッドなどの宣言の本体に現れるブロック {} の中で宣言することもでき、このように宣言されたクラスを**ローカルクラス (local class)**と呼びます[5]。次のプログラムは、ローカルクラスを利用して G601Panel クラスの宣言を書き換えたものです。

ローカルクラスとしての ShuffleButtonHandler の宣言

```
  ⋮
 5 class G601Panel extends GamePanel {
 6     Deck deck = new Deck();
 7     JButton button = new JButton("シャッフル");
 8
 9     G601Panel() {
10         deck.flip();
11         add(deck);
12         add(button, 352, 380);
*13         class ShuffleButtonHandler implements ActionListener {
*14             public void actionPerformed(ActionEvent e) {
*15                 deck.shuffleAsync();
*16             }
*17         }
18         button.addActionListener(new ShuffleButtonHandler());
19     }
20 }
```

メモ

[5]ローカルクラスの宣言では、final でないクラス変数、クラスメソッドの宣言を行うことはできません。

6.5 匿名クラス

static でないメンバクラスや、ローカルクラスとして ShuffleButtonHandler を宣言することで、G601Panel クラスの宣言はかなり簡潔なものとなりましたが、よく注意してみると、このプログラムの中で ShuffleButtonHandler を利用しているのは、

```
button.addActionListener(new ShuffleButtonHandler());
```

の 1 箇所だけであることに気づきます。

このように、たった 1 箇所でのインスタンス生成のためだけに、そのクラスに適当な名前を付けてクラス宣言を行うのは面倒です。Java には、このような場合に便利な、クラス宣言とそのインスタンス生成を同時に行う仕組みが用意されています。このようなインスタンスは、次のような書式のインスタンス生成式で生成することができます。

```
new スーパークラス(インタフェース)名(コンストラクタの引数の列) {
    インスタンスメソッドなどの宣言[6]
}
```

この書式の意味は、適当なクラス名 X を選び、スーパークラス(インタフェース)名 のサブクラスとして、インスタンスメソッドなどの宣言 を持つ内部クラス X を宣言した上で、

```
new X(コンストラクタの引数の列)
```

というインスタンス生成式を評価して、内部クラス X のインスタンスを生成する、というものです。ただし、クラス X には コンストラクタの引数の列 を、そのままスーパークラスのコンストラクタに渡すようなコンストラクタが用意されます。この内部クラス X に相当する名前のないクラスのことを**匿名クラス (anonymous class)**と呼びます。匿名クラスはすべて内部クラスとなることに注意してください。

メモ

匿名クラスを利用すると、G601Panel クラスのプログラムは、次のように書き換えることができます。

匿名クラスを利用した G601Panel クラス

```
⋮
5 class G601Panel extends GamePanel {
6     Deck deck = new Deck();
7     JButton button = new JButton("シャッフル");
8
```

[6] インスタンスメソッドの宣言の他にも、インスタンス変数、final なクラス変数、メンバクラスの宣言をすることができます。コンストラクタ、final でないクラス変数、クラスメソッドの宣言を行うことはできません。

```
 9     G601Panel() {
10         deck.flip();
11         add(deck);
12         add(button, 352, 380);
*13        button.addActionListener(new ActionListener() {
*14            public void actionPerformed(ActionEvent e) {
*15                deck.shuffleAsync();
*16            }
*17        });
18     }
19 }
```

これでかなり簡潔なプログラムとなりました。イベントハンドラを別クラスとせず、G601Panel クラスで直接 ActionListener を実装する方法をとれば、さらにプログラムを短くすることが可能ですが、複数のボタンがあって、それぞれ別のイベントハンドラを登録したいような場合には、その方法を使うことができません。そのような場合には匿名クラスが威力を発揮します。

メモ

6.6 ラムダ式

JButton クラスの addActionListener メソッドは、その引数として、ActionListener を要求します。一方、ActionListner というインタフェースには、ただ一つ、actionPerformed という抽象メソッドが宣言されているだけですので、ActionListner を実装するためには、このメソッドの定義さえできれば十分ということになります。**ラムダ式** と呼ばれる書式の式を使うと、次のプログラムのように、抽象メソッドが 1 つだけ宣言されたインタフェースを実装したクラスのインスタンスを、匿名クラスよりもさらに簡潔に生成することができます。

ラムダ式を利用した G601Panel クラス

```
 ⋮
 5 class G601Panel extends GamePanel {
 6     Deck deck = new Deck();
 7     JButton button = new JButton("シャッフル");
 8
 9     G601Panel() {
10         deck.flip();
11         add(deck);
12         add(button, 352, 380);
*13        button.addActionListener(e -> deck.shuffleAsync());
14     }
15 }
```

このプログラムの 13 行目で、addActionLisntener の引数となっている

```
e -> deck.shuffleAsync()
```

がラムダ式と呼ばれる式です。この式は、

```
new ActionListner () {
    public void actionPerformed(ActionEvent e) {
         deck.shuffleAsync();
    }
}
```

の省略形とみなされます。この例は最も簡略化された書き方ですが、ラムダ式の基本となる書式は

(仮引数宣言の列) -> メソッド定義本体

の形です。この形で、先の例を書き直すと

```
(ActionEvent e) -> { deck.shuffleAsync(); }
```

となります。ラムダ式の 仮引数宣言の列 では、仮引数の型を省略できます。この省略により

```
(e) -> { deck.shuffleAsync(); }
```

と書くことが可能で、さらに仮引数が 1 つの場合は、仮引数を囲む ( ) も省略可能です。また、メソッド定義本体 が { 式 ; } や { return 式 ; } の場合は、式 だけを書けばよいことになっていますので、先の例で見た `e -> deck.shuffleAsync()` という書き方が可能となります。

ローカルクラスや匿名クラス、ラムダ式からのローカル変数へのアクセス　コンストラクタやメソッドなどの中などで宣言されたローカルクラスや匿名クラス、ラムダ式からは、そのコンストラクタやインスタンスメソッド内の final なローカル変数にもアクセスすることができます。たとえば、上のプログラムで、ラムダ式の中からアクセスされている変数 `deck` は `G601Panel` クラスのインスタンス変数ですが、これをコンストラクタの final なローカル変数として宣言し[7]、次のようなプログラムに変更することもできます。

ラムダ式からのローカル変数へのアクセス

```
  ⋮
  5 class G601Panel extends GamePanel {
  6     JButton button = new JButton("シャッフル");
  7
  8     G601Panel() {
* 9         final Deck deck = new Deck();
 10         deck.flip();
 11         add(deck);
 12         add(button, 352, 380);
 13         button.addActionListener(e -> deck.shuffleAsync());
 14     }
 15 }
```

ローカルクラスや匿名クラス、ラムダ式からアクセスできるローカル変数は final な変数でないといけないことに注意しましょう。たとえば、上のプログラムで、ラムダ式の本体が実行されるのは、`P601Panel` クラスのコンストラクタの実行が終了した後であるはずです。この時、コンストラクタのローカル変数 `deck` は、もう存在していません[8]。このため、一般のローカル変数については、

[7]実質的に final であれば、`final` という修飾子は省略することができます。

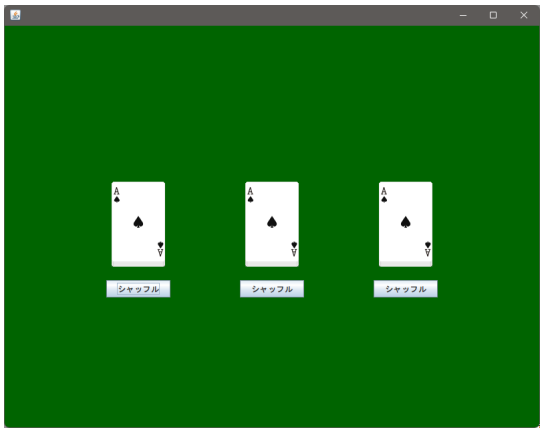
[8]この変数が記憶していた Deck クラスのインスタンスは、まだ存在しています。

ローカルクラス、匿名クラス (で宣言されたインスタンスメソッド) やラムダ式の中から、その変数にアクセスすることは許されません。一方、final なローカル変数の値は変更されませんので、その値のコピーを (匿名クラスなどの) インスタンス内部に記憶しておくことで、あたかもそのローカル変数が引き続き存在しているかのように振る舞わせることができるようになっています。

メモ

6.7 演習問題

1. G601.java を、G602.java にコピーし、ゲーム盤に 3 つのデッキと 3 つのボタンを右の図のように配置し、ボタンをクリックすると、そのボタンの上のデッキをシャッフルできるように改造しなさい。ただし、3 つのボタンのイベントハンドラは、それぞれラムダ式を使って定義しなさい。また、このラムダ式の本体では、インスタンス変数やローカル変数を参照してシャッフルするデッキを決定するようにし、ラムダ式の仮引数 (ActionEvent 型) は参照しないようにしてください。



3 つのデッキの座標は、左から (160, 240)、(360, 240)、(560, 240)、ボタンの座標は、左から (152, 380)、(352, 380)、(552, 380) となっています。

2. 次のように宣言されたクラス G603Panel は、最初は中央に置かれた 1 枚のカード (ジョーカー) を、マウスのボタンが押される度に、マウスポインタの位置に移動していくようなゲーム盤のクラス (GamePanel のサブクラス) となっています。

G603Panel.java

```
import java.awt.event.*;
import jp.ac.ryukoku.math.cards.*;

class G603Panel extends GamePanel implements MouseListener {
    Card card = new Card();
```

```
    G603Panel() {
        add(card);
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent e) {
        card.moveAsyncTo(e.getX() - card.getWidth() / 2,
            e.getY() - card.getHeight() / 2);
    }

    public void mouseReleased(MouseEvent e) { }
    public void mouseClicked(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
}
```

このゲーム盤のクラス (G603Panel) では、自分自身のクラスで MouseListener を実装していますが、この G603Panel クラスに、static でないメンバクラス CardMover の宣言を追加して、その CardMover のインスタンスを MouseListener として使用するように変更しなさい。ソースファイルの名前は G603Panel.java とし、G603Panel クラスのクラス宣言からは mousePressed などのインスタンスメソッドの宣言を削除すること。

MouseListener インタフェースについては、第 3 回の「付録: いろいろなイベントとイベントハンドラ」(第 3 回 17 ページ) を参照してください。MouseEvent クラスのインスタンスメソッド getX や getY は、イベントが発生したときのマウスポインタの (その部品内での) 位置の x 座標や y 座標を戻り値として返します。

また、このプログラムをテストする際には、次の G603.java を利用してください。

G603.java

```
import javax.swing.*;

class G603 implements Runnable {
    public void run() {
        JFrame f = new JFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(new G603Panel());
        f.pack();
        f.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new G603());
    }
}
```

3. 演習問題 2 のテストに利用した G603.java では、G603 クラスで Runnable インタフェースを実装し、main メソッドでは、この G603 クラスのインスタンスを SwingUtilities クラスのクラスメソッド invokeLater の引数としています。G603.java を書き換えて、G603 クラスでは Runnable インタフェースは実装せず、ラムダ式を invokeLater の引数にすることで同じ動作を実現しなさい。

グラフィックス及び演習・第 6 回・終わり