グラフィックス及び演習・第3回

2024年7月1日

今回の内容

3.1 イベント駆動型プログラミング

グラフィカルユーザーインタフェース (GUI[1]) を持ったアプリケーションプログラムでは、ユーザーがマウスやキーボードなどを使って操作を行います。このユーザーの操作のように、プログラムの実行の流れとは関係なく発生する出来事を、一般に、(プログラム側からの視点で)**イベント**と呼びます。イベントには、ユーザーの行う操作以外にも、ある時点から一定の時間が経過した、ネットワーク経由で何らかの要求を受け取ったなど、いろいろなものが考えられます。

コンソールウィンドウを介した文字列の入出力によるコマンドラインユーザーインタフェース (CLI[2]) では、ユーザーがプログラムの実行のペースに合わせて操作 (文字列の入力) を行いますので、一連の流れの主体はプログラムの方にあって、ユーザーの操作はその一部でしかありませんが、GUI では、ユーザーのペースに合わせて、プログラムがその操作を受理しますので、むしろ、流れを作っているのはユーザーの方で、ユーザーが発生させるイベントが、プログラム中の何からの手順を起動させていると捉えることができます。

このため、GUI のプログラムでは、

> 発生しうる各イベントに対して、それを処理する手順をそれぞれ用意しておく

という考え方でプログラミングを行います。これを一般に、**イベント駆動型** (event driven) プログラミングと呼びます。また、イベント毎に、それを処理するために定められた手続き (メソッドなど) やその仕事を行うオブジェクトを、そのイベントの**イベントハンドラ** (event handler)、あるいはイベントリスナ (event listener) と呼びます。イベント駆動型のプログラミングでは、プログラムが働くための準備[3]を行った後、

1. 何らかのイベントが発生するのを待つ
2. 発生したイベントに対するイベントハンドラを起動する

---

[1] Graphical User Interface

[2] Commnad Line Interface

[3] たとえば、GUI を持つプログラムの場合のウィンドウの作成や、GUI 部品の生成と配置など

という手順を繰り返すことになります。この繰り返しを、一般に、**イベントループ** (event loop) と呼びます。また、イベントが発生した際に、そのイベントに対応するイベントハンドラを調べて、そのイベントハンドラにイベントを送る (たとえば、引数として起動する) ことを**イベントディスパッチ** (event dispatching) と呼びます。

メモ

3.2 イベントディスパッチスレッド

Java (JFC) の GUI のプログラムでは、このイベントループの処理を `main` メソッドは別の実行の流れで行います。Java は、プログラムの異なる部分を (あたかも) 同時に並行して実行する[4]ことが可能となっており、それぞれの実行の流れを**スレッド** (thread) と呼びます。各スレッドは、プログラムの異なる部分をそれぞれ実行していくことができます。

Java プログラムが起動されると、起動したクラスの `main` メソッドを実行するためのスレッド (main スレッド) が生成されます[5]。GUI を持たないプログラムでは、この `main` スレッドの仕事がすべてで、`main` メソッドの仕事が終われば、プログラム全体も終了することになりますが、GUI を持つプログラムの場合、`JFrame` クラス[6]のインスタンスなど、トップレベルのウィンドウが生成される際に、`main` のスレッドとは別に、GUI に関するイベントループの仕事を行うスレッドが生成されます。このスレッドを**イベントディスパッチスレッド** (event dispatch thread) と呼びます。画面の再描画など (前回説明した `paintComponent` メソッドの起動など) の処理も、このスレッドが行います。イベントディスパッチスレッドが残っている場合、`main` のスレッドが終わっても、プログラム全体は終了しません。

メモ

イベントを表すクラス群　Java で扱われる GUI 関連のイベントには、マウスポインタの移動、マウスボタンの押下や解放、キーボードのキーの押下や解放のような低レベルの (単なる物理的な現

[4]複数の CPU を持つシステムでは、複数のスレッドが本当に同時に実行される場合もありますが、そうでない場合でも時分割処理を行って、あたかも同時に実行されているように見せかけることができます。

[5]この他にも、実行時コンパイルを行ったり、メモリ管理を行ったりするものなど、あらかじめいくつかのスレッドが動作しています。

[6]`GameFrame` は、このサブクラスとなっています。

象に近い) イベントもあれば、画面上のボタンがクリックされた、メニュー項目が選択されたなど、低レベルのイベントを解釈することで発生する高レベルの (何らかの操作上の意味を持つ) イベントもあります。

通常、これらのイベントは java.awt.event パッケージや javax.swing.event パッケージで定義されている特定のクラスのオブジェクトとして表現されます。たとえば、画面上のボタンを実現するための部品として Swing に用意されている JButton クラスの場合、そのボタンがクリックされると、java.awt.event.ActionEvent というクラスのインスタンスが生成されます。

ActionEvent クラス — その部品特有のアクションを起こさせる操作の発生

主なクラス変数	
static final int ALT_MASK	getModifiers の戻り値で、Alt キー押されていたことを表すビットマスク
static final int CTRL_MASK	getModifiers の戻り値で、Ctrl キー押されていたことを表すビットマスク
static final int META_MASK	getModifiers の戻り値で、Meta キー押されていたことを表すビットマスク
static final int SHIFT_MASK	getModifiers の戻り値で、Shift キー押されていたことを表すビットマスク
主なインスタンスメソッド	
Object getSource()	イベントが発生したオブジェクト (ボタンなどの部品) を戻す。
int getModifiers()	イベントが発生したときに押されていた修飾キーの状態を戻す。
long getWhen()	イベントが発生した時刻 (協定世界時 1970 年 1 月 1 日 00:00:00 からの経過時間をミリ秒単位で表したもの) を戻す。

メモ

JFC/Swing でよく使用される GUI イベントのクラスには、次の表のようなものがあります。どのクラスのイベントが発生する可能性があるか、また、発生するとしたらどのような時に発生するのかは、GUI の部品によって異なります。

java.awt.event パッケージ

ActionEvent	ボタンのクリック、メニュー項目の選択、テキスト入力フィールドでの Enter キーの押下など、その部品特有のアクションを起こさせる操作の発生
FocusEvent	その部品がキーボードフォーカスを持っているかどうか[7]に関する状態の変更
ItemEvent	一覧の項目などの選択状態の変更
KeyEvent	キーボードのキーの押下や解放

[7] KeyEvent の送り先の部品として選択されているかどうか

MouseEvent	マウスの移動、マウスボタンの押下や解放

javax.swing.event パッケージ

ChangeEvent	部品の状態の変更
ListSelectionEvent	一覧 (JList) や表 (JTable) などの選択範囲の変更

メモ

イベントハンドラの登録　GUI の部品でイベントが発生した際に何らかの処理を行いたい場合は、その処理を行うオブジェクトを生成して、その部品のイベントハンドラとして登録しておきます。イベントハンドラとなることのできるオブジェクトは、特定の名前で、特定の数と型の引数をとり、特定の型の戻り値を返すインスタンスメソッドを持っていなければなりません。

たとえば、JButton クラスの部品で発生する ActionEvent を処理するためには、

```
public void actionPerformed(ActionEvent e)
```

というインスタンスメソッドを持つオブジェクトを生成し、このオブジェクトを、その部品 (JButton のインスタンス) のイベントハンドラとして登録します。登録は、イベントハンドラを引数として、その部品の

```
public void addActionListener(ActionListener l)
```

というインスタンスメソッドを起動することにより行うことができます。このメソッドの引数の型 ActionListener は、後述のインタフェースと呼ばれる型で、イベントハンドラとなれるオブジェクトの種類を表しています。

GUI の部品で ActionEvent が発生すると、その部品に対して登録されているイベントハンドラの actionPerformed メソッドが起動されます。その際、引数として、発生したイベントに関する情報が ActionEvent クラスのインスタンスとして渡されますので、イベントハンドラとなるオブジェクトの actionPerformed メソッドは、引数の内容を調べて必要な仕事を行うように定義しておきます。

メモ

3.3 インタフェース

Java では、オブジェクトが持つべきインスタンスメソッド群を**インタフェース**と呼ばれるもので表現することができます。インタフェースは**インタフェース宣言**と呼ばれる次のような書式で定義します。

```
interface インタフェース名 {
    抽象メソッドなどの宣言[8]
}
```

インタフェース宣言の書式はクラス宣言の書式に似ています。ただし、class というキーワードの代りに interface というキーワードを用います。インタフェース名 の部分では、クラス名の場合と同様に、各単語の先頭のみを英大文字として残りを英小文字とするのが慣習ですので、これに従いましょう。インタフェースは、オブジェクトがどのような仕事をすることが可能かという、言わば、オブジェクトの資格のようなものを表しますので、ActionListener や Cloneable など、そのことが分かるようなインタフェース名がよく選ばれます。インタフェースもクラスと同様に、特定のパッケージに属するものとして扱われます。たとえば、ActionListener の完全限定名は java.awt.event.ActionListener です。

メモ

抽象メソッド　インタフェース名 に続く {} の中には、**抽象メソッド**の宣言を並べます。抽象メソッドの宣言は、次のような書式で、インスタンスメソッド[9]の名前と、その引数の数と型、戻り値の型を宣言したものです。

```
abstract 戻り値の型名 メソッド名(仮引数宣言の列);
```

通常のインスタンスメソッドの宣言との違いは、abstract という修飾子が付くことと、メソッドの本体がなく ; で終わっているところです。インタフェース宣言内で宣言できるメソッドは、すべて抽象メソッドですので、インタフェース宣言内では、abstract という修飾子を省略することもできます。また、インタフェース宣言内で宣言された抽象メソッドは、とくにアクセス修飾子が指定されていない場合でも、すべて public というアクセス修飾子を持つものと見なされます。これ以外のアクセス修飾子を抽象メソッドの宣言で指定することはできません。

---

[8] 抽象メソッドの宣言以外には、final なクラス変数、メンバクラス (このインタフェースの定義のために用いられるクラス)、メンバインタフェース (このインタフェースの定義のために用いられる別のインタフェース) の宣言が現れることができます。

[9] 抽象クラスメソッドというものはありませんので、抽象メソッドとは「抽象インスタンスメソッド」を意味します。

インタフェースの継承　クラス宣言の場合と同様に、インタフェース名 に続いて extends 節を書くことで、すでに宣言されているインタフェースを元にして、新しいインタフェースを宣言することもできます。インタフェースでは、クラスの場合と違って、元となるインタフェース名を「,」で区切って複数書くこともできます。元となるものを**スーパーインタフェース**、拡張してできるものを**サブインタフェース**と呼ぶのもクラスの場合と同様です。

インタフェース宣言の例　たとえば、java.awt.event.ActionListener は次のように宣言されています[10]。

```
public interface ActionListener extends java.util.EventListener {
    public abstract void actionPerformed(ActionEvent e);
}
```

この例では、1 つだけ抽象メソッドが宣言されていますが、複数の抽象メソッドを宣言することもできます。

メモ

3.4　インタフェースの実装

インタフェースは、オブジェクトが持つべきインスタンスメソッドを宣言したものですから、オブジェクトの資格のようなものを表していると考えることができます。しかし Java では、インタフェースで指定されたメソッドを単に持っているだけでは、その資格は認めらません。あるクラスのオブジェクトがあるインタフェースによって定義された資格を得るには、そのクラスを宣言する際に、クラス宣言の本体の直前[11]に implements 節と呼ばれる書式を書いて、そのクラスのオブジェクトがそのインタフェースが指定しているメソッドを持っていることを宣言することが必要です。次の ShuffleButtonHandler.java はそのようなクラスの宣言を行った例です。

ShuffleButtonHandler.java

```
 1 import java.awt.event.*;
 2 import jp.ac.ryukoku.math.cards.Pile;
 3
 4 class ShuffleButtonHandler implements ActionListener {
 5     Pile p;
 6
 7     ShuffleButtonHandler(Pile p) {
 8         this.p = p;
 9     }
10
```

[10] ActionListener の元となっている java.util.EventListener は、すべてのイベントハンドラを 1 つの型としてまとめて扱うためのインタフェースで、そこでは抽象メソッドは特に宣言されていません。

[11] extends 節がある場合は、その後に書きます。

```
11      public void actionPerformed(ActionEvent e) {
12          p.shuffleAsync();
13      }
14 }
```

この宣言では、class ShuffleButtonHandler に続いて、implements ActionListener という implements 節を書くことで、このクラスのオブジェクトが、ActionListener というインタフェースが要求している actionPerformed というインスタンスメソッドを持っていることを宣言しています。インタフェースで宣言される抽象メソッドはすべて public ですから、このプログラムの 11 行目のように、定義するインスタンスメソッドのアクセス修飾子も public である必要があります。このクラスでは、actionPerformed 以外のメソッドは宣言されていませんが、他のメソッドが宣言されていても構いません。

クラスが implements 節を伴って宣言されているとき、そのクラスは implements 節に書かれたインタフェース (とそのスーパーインタフェース) を**実装** (implement) していると言います[12]。たとえば、この ShuffleButtonHandler クラスは ActionListener を実装しています[13]。

このクラスのオブジェクトは Pile クラスのインスタンス (カードの山) をコンストラクタの引数に指定して生成します。生成されたオブジェクトは、ActionListener として働くことができます。このオブジェクトをイベントハンドラとしてボタンなどの部品に登録しておけば、その部品で ActionEvent が発生したときに、イベントループから、このオブジェクトの actionPerformed メソッドが起動されて、そこでコンストラクタの引数に指定されていたカードの山がシャッフルされます。12 行目で起動されている shuffleAsync メソッドは shuffle と同様にデッキをシャッフルするメソッドですが、その作業を shuffleAsync メソッドを起動したスレッドとは別のスレッドで行います。JFC による画面の更新はイベントディスパッチスレッドのイベントループの中で行われますので、もし、12 行目で shuffle を起動してしまうと、シャッフルの作業が終わるまで (イベントループから起動された actionPerformed が終わらないので) 画面の更新が行われず、デッキがシャッフルされている様子が表示されません。シャッフルが完了して初めて、新しいデッキの状態が画面に表示されることになります。

shuffleAsync の場合は、シャッフルの作業は別スレッドで行われ、shuffleAsync の呼び出しは直ちに完了します。このため、actionPerformed も直ちに終了してイベントループに戻りますので、別スレッドで行われているデッキのシャッフルの経過を随時画面に反映させることが可能です。

メモ

[12] メソッド毎に、「ShuffleButtonHandler は、ActionListener の抽象メソッド ActionPerformed を実装している」のように言うこともあります。

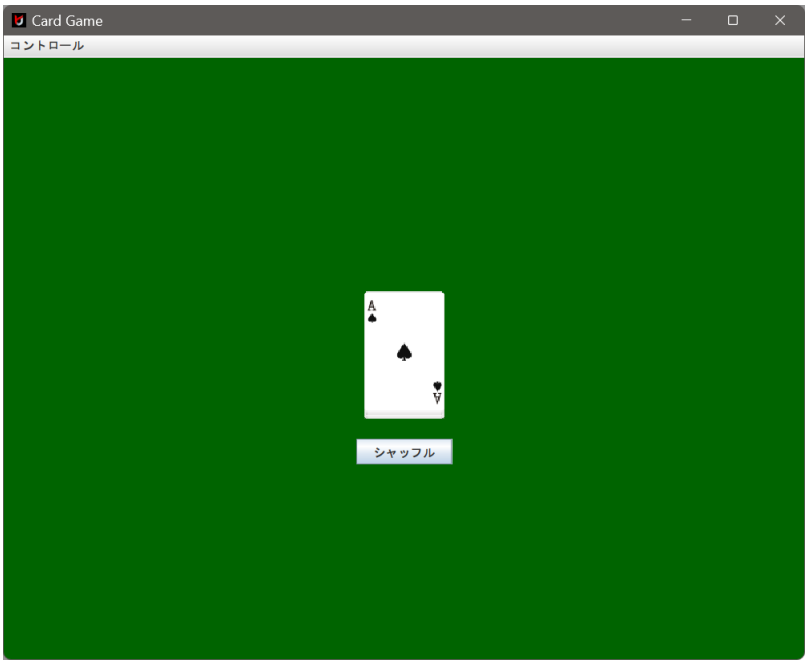
[13] ActionListener インタフェースは EventListener のサブインタフェースですので、ShuffleButtonHandler クラスは EventListener も実装していることになります。

イベントハンドラの使用例　JFC の Swing には JButton という、画面上のボタンを実現するためのクラスが用意されています。次のプログラム G301.java は、表向きにしたデッキ (Deck) とボタン (JButton) をゲーム盤に置き、ボタンを押すとデッキがシャッフルされるようにしたものです。

G301.java

```
 1 import java.awt.event.ActionListener;
 2 import javax.swing.JButton;
 3 import jp.ac.ryukoku.math.cards.*;
 4
 5 class G301 {
 6     public static void main(String[] args) {
 7         GameFrame f = new GameFrame();
 8         Deck d = new Deck();
 9         ActionListener l = new ShuffleButtonHandler(d);
10         JButton b = new JButton("シャッフル");
11         b.addActionListener(l);
12         d.flip();
13         f.add(d);
14         f.add(b, 352, 380);
15     }
16 }
```

JButton のインスタンスは、ボタンがクリックされたときに ActionEvent を発生させます。9 行目で生成した ShuffleButtonHandler クラスのインスタンスを ActionListener として、ボタン (JButton) に登録しています (11 行目)。



メモ

インタフェース型　クラスを宣言すると、そのクラスの名前は型の名前となり、その型は、そのクラスとそのサブクラスのインスタンスすべてと null 参照を含むものとなります。インタフェースも同様に、その名前は型の名前となり、その型は、そのインタフェースを実装しているクラスのインスタンスすべてと null 参照を含むものとなります。たとえば、G301.java の 9 行目で宣言された変数 l の型としてインタフェース ActionListener が使われています。ShuffleButtonHandler はこのインタフェースを実装していますので、そのインスタンスは ActionListener 型の値として扱うことができます。

インタフェースを宣言することでできる型を**インタフェース型**と呼びます。インタフェース型もクラス型と同様に参照型の一種です。

main を持つクラスでの ActionListener の実装　G301.java では、ActionListener を実装したクラスを別に宣言して、そのインスタンスをイベントハンドラとして用いましたが、これまで、main メソッドを宣言するためだけに使っていたクラスをイベントハンドラとして利用することもできます。次の G302.java は、G302 というクラス自身を ShuffleButtonHandler に相当するものとして利用するプログラムです。

G302.java

```
 1 import java.awt.event.*;
 2 import javax.swing.JButton;
 3 import jp.ac.ryukoku.math.cards.*;
 4
 5 class G302 implements ActionListener {
 6     Pile p;
 7
 8     G302(Pile p) {
 9         this.p = p;
10     }
11
12     public void actionPerformed(ActionEvent e) {
13         p.shuffleAsync();
14     }
15
16     public static void main(String[] args) {
17         GameFrame f = new GameFrame();
18         Deck d = new Deck();
19         ActionListener l = new G302(d);
20         JButton b = new JButton("シャッフル");
21         b.addActionListener(l);
22         d.flip();
23         f.add(d);
24         f.add(b, 352, 380);
25     }
26 }
```

メモ

main メソッドを持つクラスのインスタンスを積極的に使用するのであれば、次の G303.java のように、main メソッドの中で行っていた仕事をそのクラスのコンストラクタの中に移動することもできます。

G303.java

```
 1 import java.awt.event.*;
 2 import javax.swing.JButton;
 3 import jp.ac.ryukoku.math.cards.*;
 4
 5 class G303 implements ActionListener {
 6     GameFrame f;
 7     Deck d;
 8     JButton b;
 9
10     G303() {
11         f = new GameFrame();
12         d = new Deck();
13         b = new JButton("シャッフル");
14         b.addActionListener(this);
15         d.flip();
16         f.add(d);
17         f.add(b, 352, 380);
18     }
19
20     public void actionPerformed(ActionEvent e) {
21         d.shuffleAsync();
22     }
23
24     public static void main(String[] args) {
25         new G303();
26     }
27 }
```

このプログラムの場合、main メソッドでの仕事は、単に G303 クラスのインスタンスを生成することだけになってしまっています。

メモ

3.5 GUI 関連処理のイベントディスパッチスレッドへの集中化

ここまで作成したプログラムはすべて、main スレッドを起動したスレッド (main スレッド) で、ウィンドウ (GameFrame) を生成し、そこにデッキやカード、ボタンなどを追加していました。今のところこれで問題なく動作してはいますが、実はこのやり方には不具合を引き起こす可能性が含まれています。

Swing の部品群やそれを動作させるための仕組みは、複数のスレッドから同時に操作されることを想定していません。このため、複数のスレッドが勝手に Swing の部品を操作すると、画面の

表示が乱れたり、ユーザーの操作を正しく認識しないと言った不具合が起ってしまう可能性があります。このため Swing では、部品の生成、配置、操作など、GUI に関わるすべての仕事をイベントディスパッチスレッドで行う約束になっています。

カードゲーム向けのクラスライブラリでは、このようなことが起らないような配慮がされていますので、複数のスレッドから GameFrame や Card、Deck などの部品を操作しても構いませんが、JButton のように Swing の標準的な部品を使用する場合は、この約束に従わなければなりません。Swing のイベントディスパッチスレッドで行いたい仕事がある場合は、

```
public interface Runnable {
    public abstract void run();
}
```

のように宣言された Runnable というインタフェース[14]を実装したクラスのオブジェクトを生成し、run メソッドの中で行いたい仕事を行うようにします。このようなオブジェクトを引数として、javax.swing.SwingUtilities クラスの invokeLater というクラスメソッドを起動しておくと、イベントディスパッチスレッドから、引数に渡されたオブジェクトの run メソッドの起動を行ってくれます[15]。

メモ

次の G304.java というプログラムは、この方法を利用して、G303.java を Swing の約束に従うように書き直したものです。25 行目で invokeLater を使って、イベントディスパッチスレッドから run が起動されるようにしています。ここで、G304 クラスのインスタンスを生成していますが、G304 クラスの (デフォルト) コンストラクタは GUI に関する仕事は全く行いません。

G304.java

```
 1 import java.awt.event.*;
 2 import javax.swing.*;
 3 import jp.ac.ryukoku.math.cards.*;
 4
 5 class G304 implements ActionListener, Runnable {
 6     GameFrame f;
 7     Deck d;
 8     JButton b;
 9
10     public void actionPerformed(ActionEvent e) {
11         d.shuffleAsync();
12     }
13
```

[14] Runnable クラスは java.lang パッケージのインタフェースなので、特に import 宣言を書かなくても、Runnable とだけ書いて、このインタフェースを指定することができます。

[15] invokeLater の名前が示すように、直ちに run メソッドが起動されるのではなく、イベントディスパッチスレッドが処理すべきイベントが溜っている場合は、それらの処理が終わってから run が起動されます。

```
14      public void run() {
15          f = new GameFrame();
16          d = new Deck();
17          b = new JButton("シャッフル");
18          b.addActionListener(this);
19          d.flip();
20          f.add(d);
21          f.add(b, 352, 380);
22      }
23
24      public static void main(String[] args) {
25          SwingUtilities.invokeLater(new G304());
26      }
27  }
```

3.6 CardEvent クラスと CardListener インタフェース

この科目のカードゲーム向けクラスライブラリでは、カードがドラッグされたり、ダブルクリックされたりしたことを、高レベルのイベントとして知ることができます。このイベントのイベントハンドラは、次のようなインタフェースとなっています。

```
public interface CardListener extends java.util.EventListener {
    boolean cardSelected(CardEvent e);
    boolean cardMoved(CardEvent e);
    void cardPicked(CardEvent e);
}
```

Card クラスのインスタンスメソッド

```
public void addCardListener(CardListener l)
```

を使って、カードに対してイベントハンドラを登録しておくと、各メソッドが以下のように起動されます。

cardSelected　カード上でマウスボタンが押されると、まず cardSelected メソッドが起動されます。イベントハンドラは、このイベントに対する処理を行って boolean 型の戻り値を返さなければなりません。true が返されると、その後のドラッグ操作に従って (一時的に) そのカードが移動します。false の場合は、マウスがドラッグされてもカードは移動しません。

cardMoved　マウスによるカードのドラッグが終了する (マウスボタンが放される) と、cardMoved が起動されます。cardMoved メソッドは、この操作に対する処理を行って、やはり boolean 型の戻り値を返します。この戻り値は、そのドラッグ操作をイベントハンドラが承認したかどうかを表します。true が返された場合は、特にそれ以上のことは起りませんが、false の場合、カードは元の位置に戻されます。

cardPicked　カード上でマウスによるダブルクリックが起ると、cardPicked が起動されます。このメソッドには戻り値はありませんので、その操作に対して必要な処理を行うだけとなります。

メモ

CardEvent クラス　これら 3 つのメソッドが起動される際には、起ったイベントに関する情報が CardEvent クラスのインスタンスとして渡されます。CardEvent クラスは MouseEvent クラスのサブクラスとなっており、独自のインスタンスメソッドとして次のようなものが提供されています。

CardEvent クラス — カードに対する操作の発生

主なインスタンスメソッド	
`Card getCard()`	イベントが発生したカードを戻す。
`Pile getPile()`	イベントが発生したカードが属している山 (Pile や Deck) を戻す。どの山にも属していない場合は null を戻す。
`Pile getDest()`	カードの移動先となっている山 (Pile や Deck) を戻す。山への移動でない場合は null を戻す。
`int getOriginalX()`	カードの元の (移動開始前の) x 座標を戻す。
`int getOriginalY()`	カードの元の (移動開始前の) y 座標を戻す。

cardMoved が呼ばれたときのカードの新しい位置は、その引数 e に対して getCard を起動し、その戻り値に対して (Card のインスタンスメソッドである) getX や getY を起動することで取得できます。

メモ

CardEvent の既定の処理　CardEvent のイベントハンドラ (CardListener) が 1 つも登録されていないカードでは、次のような既定の処理が行われます。

- cardSelected では、何もしないで true を戻します。
- cardMoved では特に何もせずに、そのカードが山に属していなければ true を、属している場合は false を戻します。このため、カードが山に属していなければ移動先の位置に留まりますが、山にあったカードは元の位置 (山) に戻ってしまいます。
- cardPicked では、そのカードが山の 1 番上のカードであれば山から取り出し (pickUp)、山に属していない場合はカードを反転します (flip)。それ以外 (山の 2 番目より下のカード) の場合は何もしません。

メモ

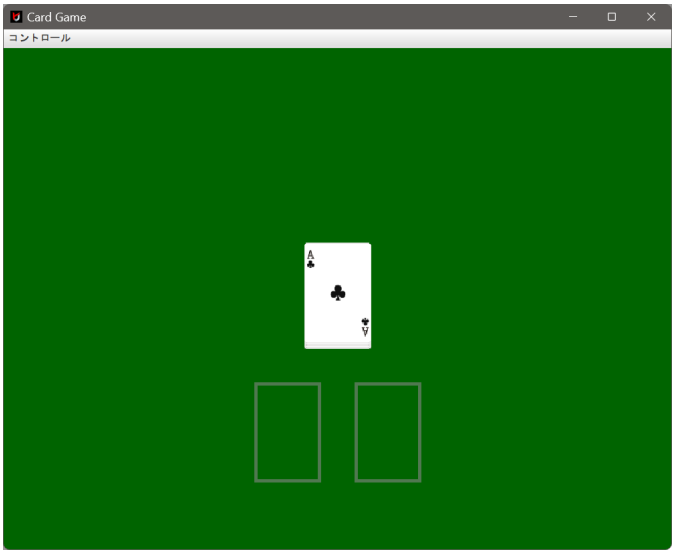
CardListener のプログラムの例　次の G305.java は CardListener を使ったプログラムの例です。G305 クラスのインスタンス自身が CardListener として働きます。また、G304.java と同様に、main メソッドの中では、SwingUtilities.invokeLater(new G305()); のみを実行して、すべての処理をイベントディスパッチスレッドで行うようにしています (48 行目)。

G305.java

```
 1 import javax.swing.SwingUtilities;
 2 import jp.ac.ryukoku.math.cards.*;
 3
 4 class G305 implements CardListener, Runnable {
 5     Pile p1, p2;
 6     Deck d;
 7
 8     public boolean cardSelected(CardEvent e) {
 9         return true;
10     }
11
12     public boolean cardMoved(CardEvent e) {
13         Card card = e.getCard();
14         Pile from = e.getPile();
15         Pile to = e.getDest();
16         if (to == null || from == null || from == to
17                 || from.top() != card) {
18             return false;
19         }
20         Card top = to.top();
21         if (top == null || card.isJoker() || top.isJoker()
22                 || top.isRed() == card.isRed()) {
23             card.moveAsyncTo(to);
24             return true;
25         }
26         return false;
27     }
28
29     public void cardPicked(CardEvent e) {
30     }
31
32     public void run() {
33         GameFrame f = new GameFrame();
34         p1 = new Pile();
35         p2 = new Pile();
36         d = new Deck(2);
37         for (Card c : d.getCards()) {
38             c.addCardListener(this);
39         }
40         d.shuffle();
```

```
41          d.flip();
42          f.add(d);
43          f.add(p1, 300, 400);
44          f.add(p2, 420, 400);
45      }
46
47      public static void main(String[] args) {
48          SwingUtilities.invokeLater(new G305());
49      }
50 }
```

このプログラムは、ゲーム盤上に作成した 3 つの山 (デッキ 1 つと空の山 2 つ) の間で、マウスのドラッグ操作によりカードを移動させることができるようにしたものです。ただし、移動先の山 (あるいはデッキ) は、空であるか、移動するカードと同じ色 (赤あるいは黒) のカードが一番上になければなりません。この条件を満さないカードの移動はできないようになっています。また、ジョーカーは、赤黒どちらのカードとしても見なされるようにしています。



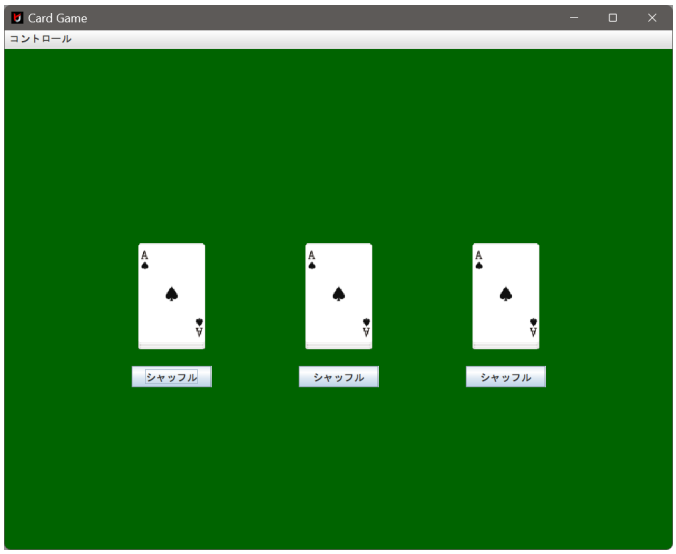
このプログラムの 37 行目で使用している `getCards` は、その山に含まれるすべてカードを配列 (`Card[]` 型) にして戻す `Pile` クラスのインスタンスメソッドです。デッキ中のすべてのカードに対して、`addCardListener` を起動して、`CardEvent` のイベントハンドラ (`G305` クラスのインスタンス) を登録しています (38 行目)。

また、22 行目では、そのカードが赤のカードであるかどうかを `boolean` 型の戻り値で返す `Card` クラスのインスタンスメソッド `isRed` を利用しています。23 行目の `moveAsyncTo` は、別スレッドで `moveTo` と同じ仕事を行うメソッドです。

メモ

3.7 演習問題

1. G304.java を G306.java にコピーし、ゲーム盤に 3 つのデッキと 3 つのボタンを次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しなさい。



イベントハンドラの actionPerformed メソッドには ActionEvent クラスのインスタンスが引数として渡されますが、この引数に対して getSource メソッドを起動して、クリックされたボタンを調べる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. 演習問題 1 の G306.java と同じ動作をするプログラム G307.java を、actionPerformed メソッドが、引数に渡されてきた ActionEvent クラスのオブジェクトを使わずに仕事をするようなものに書き換えなさい。もちろん getSource メソッドも使用しません。G301.java で利用した ShuffleButtonHandler のインスタンスを 3 つ生成し、3 つのボタンにそれぞれ登録するようにします。

3. G305.java を G308.java にコピーし、その動作を次のように変更しなさい。

   - カードをダブルクリックすると表裏が反転する
   - 裏向きのカードはドラッグ操作に反応しない
   - 1 番上のカードが裏向きの山へはカードを移動できない (元の位置に戻ってしまう)

   Card クラスの flip メソッドの代りに、flipAsync というメソッドを起動すると、別スレッドでカードを反転することができます。

3.8 付録: いろいろなイベントとイベントハンドラ

Swing で扱われる主なイベントには次のようなものがあります。これらの詳細については「Java Platform, Standard Edition 8 API 仕様」(`https://docs.oracle.com/javase/jp/8/docs/api/`) を参照してください。

イベントのクラス 　イベントハンドラの 　　インタフェース 　　起動されるメソッド 　イベントハンドラの登録メソッド	イベントの内容

`java.awt.event` パッケージ

`ActionEvent`
　`ActionListener`
　　`actionPeformed`
 	
　`addActionListener`
	ボタンやメニュー項目などがクリックされた、テキスト入力フィールドで Enter キーが押されたなど

`FocusEvent`
　`FocusListener`
　　`FocusGained`
　　`FocusLost`
　`addFocusListener`
	その部品がキーボードフォーカスを取得した
	その部品がキーボードフォーカスを失った

`ItemEvent`
　`ItemListener`
　　`itemtateChnaged`
　`addItemListener`
	項目の選択状態が変更された

`MouseEvent`
　`MouseListener`
　　`mousePressed`
　　`mouseClicked`
　　`mouseReleased`
　　`mouseEntered`
　　`mouseExited`
　`MouseMotionListener`
　　`mouseMoved`
　　`mouseDragged`
　`addMouseListener`
	マウスボタンが押された
	マウスボタンが押されて (移動しないまま) 放された
	マウスボタンが放された
	マウスポインタがその部品の中に入った
	マウスポインタがその部品の中から出た
	
	マウスポインタが移動した
	マウスポインタがボタンを押されたまま移動した (ドラッグされた)

`KeyEvent`
　`KeyListener`
　　`keyPressed`
　　`keyTyped`
　　`keyReleased`
　`addKeyListener`
	キーボードのキーが押された
	特定の文字を表す組み合わせでキーボードのキーが押された
	キーボードのキーが放された

`javax.swing.event` パッケージ

`ChangeEvent`
　`ChangeListener`
　　`stateChanged`
　`addChangeListener`
	部品の状態が変更された

`ListSelectionEvent`
　`ListSelectionListener`
　　`valueChanged`
　`addListSelectionListener`
	一覧や表などの選択範囲が変更された

グラフィックス及び演習・第 3 回・終わり