グラフィックス及び演習・第 5 回

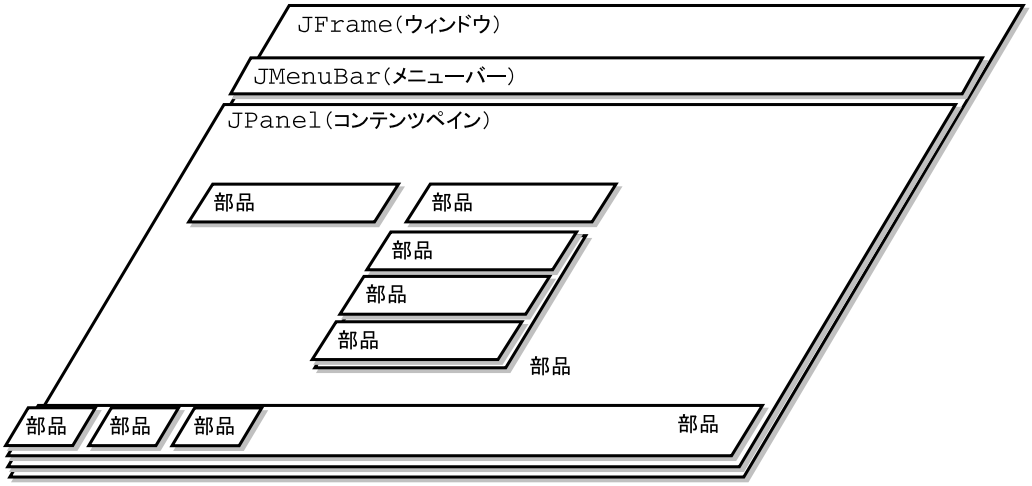
2024 年 7 月 15 日

今回の内容

5.1 GUI アプリケーションの構成 . . . . . 5 – 1
5.2 レイアウトマネージャ . . . . . 5 – 5
5.3 メニューバー . . . . . 5 – 8
5.4 フリーセル . . . . . 5 – 10
5.5 演習問題 . . . . . 5 – 11
5.6 付録: `FreeCellPanel.java` . . . . . 5 – 13

5.1 GUI アプリケーションの構成

JFC/Swing を用いた一般的な GUI アプリケーションは、次の図のように `javax.swing.JFrame` クラスのインスタンスを最上位の入れ物 (ウィンドウ) として、その内部に、メニューバーとコンテンツペイン[1]と呼ばれる GUI 部品を配置することで構成されます。メニューバーはなくても構いませんが、コンテンツペインは必ず存在しなければなりません。



Swing では、ボタンや、ラベル、メニューなどの GUI 部品は、どれも `javax.swing.JComponent` のサブクラスとなっていて、すべて、メニューバーあるいはコンテンツペインに配置します。多くの部品は、その内部に、さらにいくつかの部品を持っていますので、全体としては、`JFrame` のインスタンス (最上位のウィンドウ) を根とする階層構造 (木構造) となります。1 つの部品 (オブジェクト) が、この階層構造の複数の場所に現れることはありません。

メモ

[1]元の英語は content panes なので「コンテントペイン」と呼ぶこともあります。

JFrame クラス　Swing の JFrame クラスは、GUI を持つ Java アプリケーションの最上位のウィンドウを実現するためのクラスで、以下のようなコンストラクタやメソッドを持っています。

JFrame クラス — 最上位のウィンドウ

主なコンストラクタ	
JFrame()	タイトル文字列のないウィンドウ
JFrame(String title)	title をタイトル文字列とするウィンドウ
主なクラス変数[2]	
static final int DO_NOTHING_ON_CLOSE	ウィンドウを閉じようとする操作に対して何もしない
static final int HIDE_ON_CLOSE	ウィンドウを閉じようとする操作でウィンドウを隠す
static final int DISPOSE_ON_CLOSE	ウィンドウを閉じようとする操作でウィンドウを破棄する
static final int EXIT_ON_CLOSE	ウィンドウを閉じようとする操作でアプリケーションを終了する
主なインスタンスメソッド	
void setVisible(boolean b)	ウィンドウの表示/非表示の状態を b にする
void setDefaultCloseOperation (int operation)	このウィンドウが閉じられようとした時に operation で示される処理を行うように設定する
void setIconImage (Image image)	このウィンドウのアイコン画像を image にする
void setJMenuBar (JMenuBar bar)	このウィンドウのメニューバーを bar にする
void setContentPane (Container[3] pane)	このウィンドウのコンテンツペインを pane にする
Container getContentPane()	このウィンドウのコンテンツペインを戻す
void pack()	このウィンドウを最適な大きさに調整し、その内部のレイアウトを行う
void addWindowListener (WindowListener[4] listener)	このウィンドウで発生する WindowEvent[5] のイベントハンドラに listener を追加する

JFrame のインスタンスを生成すると、自動的に JPanel クラスのインスタンスが生成されて、そのコンテンツペインとなります。メニューバーが必要な場合は、別途 JMenuBar クラスのインスタンスを生成して、JFrame に追加しなければなりません。

生成された JFrame は、そのままでは画面には表示されません。ウィンドウを画面に表示するには、生成したインスタンスに対して setVisiable(true) を起動します。

インスタンスメソッド setDefaultCloseOperation は、タイトルバーのウィンドウを閉じるためのボタンがクリックされた時など、このウィンドウが閉じられようとしたときの処理を設定します。4 通りの処理の中から 1 つを選択することが可能で、JFrame のクラス変数として宣言されているいずれかの整数値で設定します。既定の処理は HIDE_ON_CLOSE です。単に、アプリケーショ

[2] ここに挙げた 4 つのクラス変数は、JFrame が実装している WindowConstants という (スーパー) インタフェースで宣言されたものです (EXIT_ON_CLOSE については、JFrame でも、同じ値のクラス変数として宣言されています)。

[5] java.awt.event.WindowEvent クラス

[3] java.awt.Container クラス (JComponent のスーパークラス)

[4] java.awt.events.WindowListener インタフェース

ンプログラムを終了する場合は、EXIT_ON_CLOSE を指定します。4 通り以外の特別な処理を行いたい場合は、addWindowEventListener メソッドで、WindowListener を登録しておけば、ウィンドウが初めて表示された、アクティブになった、閉じられようとした、閉じられた、最小化 (アイコン化) された、通常の大きさになった、などのイベントが発生したことを知ることができます。

メモ

次の G501.java は、JFrame のインスタンスを生成して、それをそのまま画面に表示するプログラムです。5 〜 6 行目を、直接 main の中で実行しても同じですが、このプログラムは Swing の約束に従って、イベントディスパッチスレッドの中で JFrame を生成するようにしています。

G501.java

```
 1 import javax.swing.*;
 2
 3 public class G501 implements Runnable  {
 4     public void run() {
 5         JFrame f = new JFrame();
 6         f.setVisible(true);
 7     }
 8
 9     public static void main(String[] args) {
10         SwingUtilities.invokeLater(new G501());
11     }
12 }
```

コンテンツペインには何の部品も配置していませんので、画面には、右の図のようにタイトルバーのみが表示されます。タイトルバーのボタンでウィンドウを閉じようとすると、ウィンドウは消えてしまいますがプログラムは動作したままです。このボタンで、アプリケーションプログラムを終了させたい場合は、5 行目と 6 行目の間に

```
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

の 1 行を加えます。

JFrame のコンテンツペインに GUI 部品を 追加するには、getContentPane メソッドを起動してコンテンツペイン (JFrame が自動的に生成した JPanel クラスのインスタンス) を取得し、そのオブジェクトに対して add メソッドを起動して部品を追加します。たとえば、カードゲーム向けライブラリの Card クラスのインスタンスを追加したい場合は

```
java.awt.Container p = f.getContentPane();
p.add(new Card());
```

のようにします。

java.awt.Container クラスは、Swing の前身である AWT において、内部に別の部品を含む

ような GUI 部品のスーパークラスです。このクラスのインスタンスメソッド add の引数となる GUI 部品は java.awt.Component クラス、あるいはそのサブクラスでなければなりません。この Component は AWT のすべての GUI 部品のスーパークラスとなっています。Swing の GUI 部品は、Component のサブクラスである javax.swing.JComponent[6]のサブクラスとして実現されていますので、この add メソッドで Swing の部品を追加することができます。

しかし、コンテンツペインに部品を追加しただけでは、JFrame の大きさは変わりませんので、コンテンツペインの内容に合わせてコンテンツペイン (JPanel) やウィンドウ (JFrame) の大きさを調整する必要があります。これを行うには、次のように JFrame クラスのインスタンスメソッド pack を起動します。

```
f.pack();
```

この pack の起動を行わないと、コンテンツペインは幅も高さも 0 の状態のままとなりますので、setVisible(true) しても、画面に表示されるのは (G501.java と同じように) ウィンドウの装飾 (タイトルバー) のみとなってしまいます[7]。

メモ

次のプログラム G502.java は、以上の手順を踏むように run メソッドを書き換えたものです。コンテンツペインに Card のインスタンスを追加しています。

G502.java

```
 1 import java.awt.*;
 2 import javax.swing.*;
 3 import jp.ac.ryukoku.math.cards.*;
 4
 5 public class G502 implements Runnable {
 6     public void run() {
 7         JFrame f = new JFrame();
 8         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 9         Container p = f.getContentPane();
10         p.add(new Card());
11         f.pack();
12         f.setVisible(true);
13     }
14
15     public static void main(String[] args) {
16         SwingUtilities.invokeLater(new G502());
17     }
18 }
```

[6]実際は、Component のサブクラスが Container で、そのサブクラスが JComponent となっています

[7]ウィンドウを最大化すれば、コンテンツペインの内容が表示されます

この G502.java を起動すると、右の図のようなウィンドウが表示されます。コンテンツペインの高さは、カードの高さ (120 ピクセル) になりますが、幅は、ウィンドウのタイトルバーの最小の幅まで引き延ばされています。また、カードの幅 (本来は 80 ピクセル) も、コンテンツペインに合わせて引き延ばされています。ウィンドウの大きさをユーザーが変更すると、それに合わせてコンテンツペインやカードの大きさが変更されます。

メモ

5.2 レイアウトマネージャ

JFrame のコンテンツペインとなっていた JPanel など、いくつかの部品を内部に含む (部品を置く台紙として働く) GUI 部品のクラスには、その部品の大きさに合わせて、その内部に置かれた (add された) 部品の大きさや位置を調整するオブジェクトを登録しておくことができます。

JFC では、このような仕事を行うオブジェクトを**レイアウトマネージャー**と呼び、その資格が java.awt.LayoutManager というインタフェースとして宣言されています。G502.java で、コンテンツペインに追加したカードの大きさが変わったのは、このコンテンツペインに登録されていたレイアウトマネージャによって、カードの大きさが調整されたためです。

レイアウトマネージャの登録　JPanel など、java.awt.Container のサブクラスの部品には、

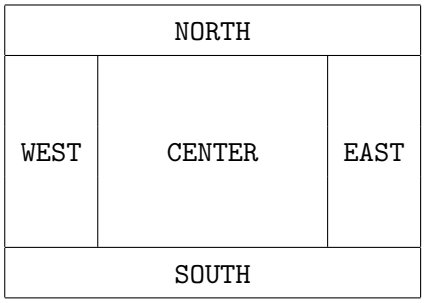
```
public void setLayout(LayoutManager manager);
```

というインスタンスメソッドが用意されており、これを起動することで、レイアウトマネージャを登録することができます。

BorderLayout クラス　最上位のウィンドウとなる JFrame のインスタンスを生成すると、自動的に JPanel のインスタンスが生成されて、それがコンテンツペインとなりますが、このコンテンツペインには、java.awt.BorderLayout クラスのオブジェクトがレイアウトマネージャーとして登録されます[8]。

BorderLayout のインスタンスは、自分がレイアウトを管理している部品 (JPanel のインスタンスなど) の占める区画を、次のような 5 つの部分に分けて、それぞれ最大 1 個の部品を配置しようとします。

[8] 当然、BorderLayout クラスは LayoutManager インタフェースを実装しています



全体の大きさが決まっている場合、BorderLayout では、次のように各部分の部品の大きさを設定します。

1. NORTH や SOUTH に配置された部品は、区画全体の幅まで水平方向に引き延ばされます。これらの部品の高さは、それぞれの部品の自然な高さとなります。
2. WEST や EAST に配置された部品は、それぞれの自然な幅となります。また、NORTH と SOUTH の間を埋めるように垂直方向に引き延ばされます。
3. CENTER に配置された部品は、残りの空間を埋めるように、上下左右に引き延ばされます。

BorderLayout が登録された (JPanel などの) 部品に、1 引数の add メソッドを使って部品を追加すると、その部品は CENTER の位置に配置されます。たとえば、G501.java の 11 行目で add したカードは、CENTER の部分に配置されています。他の 4 つの部分に部品を追加したい場合は、

```
void add(Component comp, Object const)
```

という 2 引数の add メソッドを使うことができます[9]。このメソッドの第 2 引数には、レイアウトマネージャーがその部品の配置を行う際に使用する情報を指定します。BorderLayout では、次の表にあるような 5 つのクラス変数が宣言されていますので、そのいずれかを指定することで、5 つの部分のどこに部品を配置するかを指定することができます。

BorderLayout クラス — 中央とその周りに部品を配置するレイアウトマネージャ

主なコンストラクタ	
BorderLayout()	5 つの部分の間を空けないレイアウトを行う
BorderLayout(int h, int v)	5 つの部分の間に、水平方向では h ピクセル、垂直方向では v ピクセルの間隔を空けてレイアウトを行う
主なクラス変数	
static final String NORTH	NORTH へ配置
static final String WEST	WEST へ配置
static final String CENTER	CENTER へ配置
static final String EAST	EAST へ配置
static final String SOUTH	SOUTH へ配置

[9] この 2 引数の add メソッドや、先に紹介した 1 引数の add メソッドは、同じものが JFrame クラスにも用意されています。JFrame クラスのインスタンスに対してこれらの add メソッドを起動すると、(特に事前にそうしないように設定しておかない限り) JFrame のコンテンツペインとなっているオブジェクトに対して、同じ add メソッドを同じ引数で起動してくれます。このため、G502.java の 11 行目の p.add(new Card()); は、f.add(new Card()); と書いても同じ動作となります。

たとえば、G502.java の run メソッドを

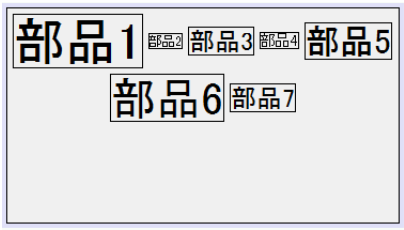
```
public void run() {
    JFrame f = new JFrame();
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container p = f.getContentPane();
    p.add(new Card(), BorderLayout.CENTER);
    p.add(new Card(), BorderLayout.NORTH);
    p.add(new Card(), BorderLayout.WEST);
    p.add(new Card(), BorderLayout.SOUTH);
    p.add(new Card(), BorderLayout.EAST);
    f.pack();
    f.setVisible(true);
}
```

のように書き換えると、コンテンツペインの内容は右の図のように配置されます。NORTH や SOUTH の高さは Card の自然な高さである 120 ピクセルに、WEST や EST の幅は Card の自然な幅である 80 ピクセルになっています。

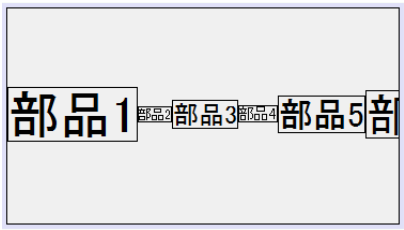
メモ

その他のレイアウトマネージャー　JFC には他にも、以下のような様々なレイアウトマネージャーが用意されています。

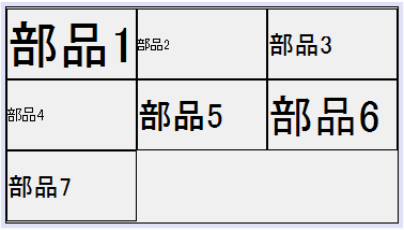
FlowLayout — 部品を左から右へ 1 列に並べます。入りきれなくたら 1 行下にずらして、また左から右へ並べます。設定を変えれば、右から左へ並べることもできます。それぞれの部品の大きさは自然な大きさのままで変更しません。右の図では、各行の部品を中央寄せにしていますが、左に寄せたり、右に寄せたりすることもできます。



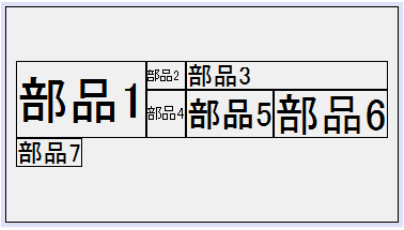
BoxLayout — 部品を左から右へ 1 列に並べます。全体の大きさが大きすぎたり小さすぎたりする場合には、それぞれの部品に設定された最小サイズと最大サイズの範囲で、各部品の大きさを調整しようとします。右端からはみ出てしまった部品は見えなくなります。設定を変えれば、上から下へ部品を 1 列に並べることもできます。



GridLayout — 部品を等間隔の格子の中に配置します。それぞれの部品は格子の間隔一杯に引き延ばされ、すべて同じ大きさとなります[10]。



GridBagLayout — 部品を格子状に区切られた領域の指定された範囲に配置します。指定された範囲を占めるように、水平方向、垂直方向に引き延ばすこともできますし(右の図はその例)、その部品の自然な大きさのままにしておくこともできます。

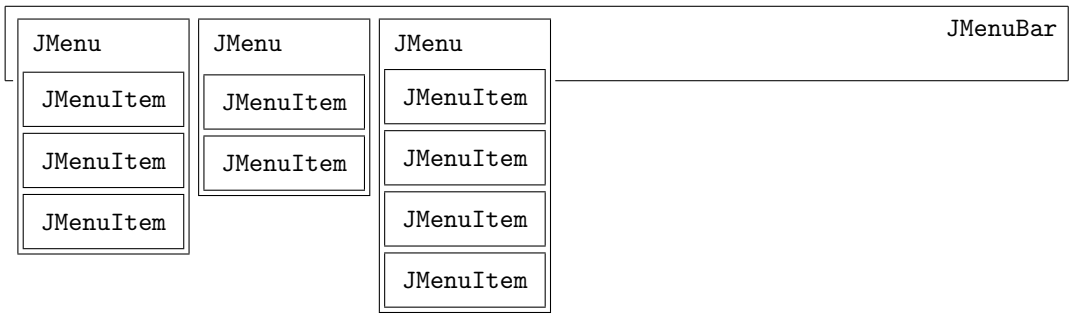


これらの他にも、部品を紙芝居のように重ねておいて、適宜その順番を変える CardLayout や、より詳細な配置を行うことのできる GroupLayout などのレイアウトマネージャーが用意されています。

メモ

5.3 メニューバー

アプリケーションのウィンドウにメニューバーを追加するためには、JMenuBar クラスのインスタンスを生成し、JFrame クラスのインスタンスメソッド setJMenuBar の引数に指定します (2 ページの表参照)。下図のように、メニューバーに並ぶメニューは、それぞれ JMenu クラスのインスタンスで表現します。また、各メニュー中の選択項目は JMenuItem クラスのインスタンスで表現します。JMenuItem は JMenu の add メソッドで JMenu に、JMenu は JMenuBar の add メソッドで JMenuBar に、それぞれ追加することができます。



[10]「オブジェクト指向及び演習」第 1 回の P103TicTacToe.java では、9 個の JButton を生成して GridLayout で格子状に配置していました。

ユーザーによってメニュー項目 (JMenuItem) の 1 つが選択されると、その JMenuItem で ActionEvent が発生します。このイベントハンドラを登録しておくことで、メニュー項目が選択されたときの処理を行うことができます。次のプログラム G503.java は、G502.java にメニューバーを追加し、カードの裏返しと、アプリケーションの終了を行うメニュー項目を含むメニューを追加したものです。

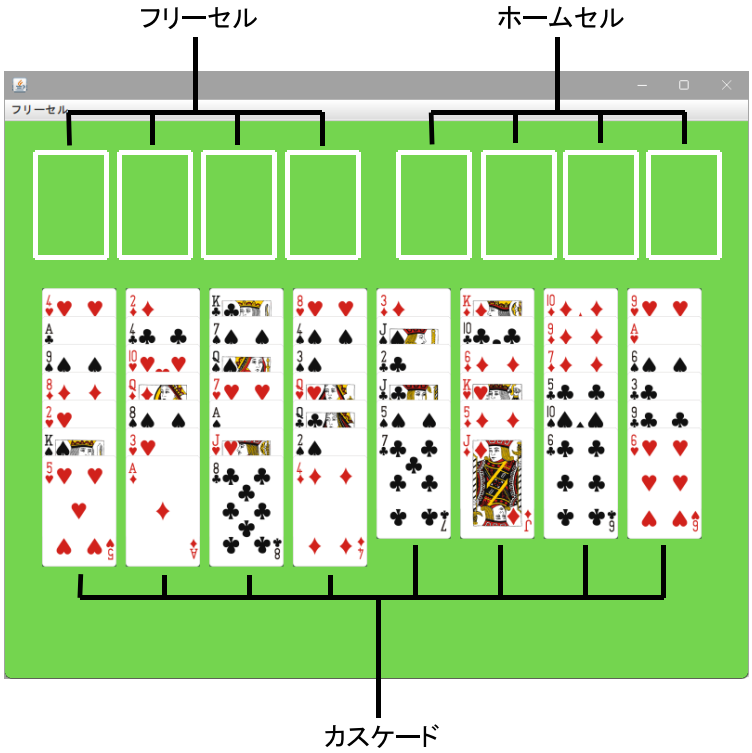
G503.java

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

public class G503 implements Runnable, ActionListener {
    JMenuItem flip = new JMenuItem("裏返す");
    JMenuItem quit = new JMenuItem("終了");
    Card card = new Card();

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == flip) {
            card.flipAsync();
        }
        else {
            System.exit(0);
        }
    }

    public void run() {
        JFrame f = new JFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        flip.addActionListener(this); // JMeniItem に actionListener を登録
        quit.addActionListener(this); // JMenuItem に actionListener を登録
        JMenuBar menuBar = new JMenuBar();      // メニューバーを作成
        JMenu menu = new JMenu("メニュー");      // メニューを作成
        menu.add(flip);                         // メニュー項目を追加
        menu.add(quit);                         // メニュー項目を追加
        menuBar.add(menu);            // メニューをメニューバーに追加
        f.setJMenuBar(menuBar);       // メニューバーを JFrame に追加
        f.getContentPane().add(card);
        f.pack();
        f.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new G503());
    }
}
```

JMenuItem が選択された際に発生する ActionEvent のイベントハンドラ actionPerfmored から参照できるように、各 JMenuItem と Card のインスタンスを記憶する変数を (run メソッドのローカル変数ではなく) G503 クラスのインスタンス変数としていることに注意してください。

5.4　フリーセル

GUI を持つアプリケーションの例として、フリーセルと呼ばれるトランプを使った一人遊びのプログラムを紹介します。フリーセルでは、まず、ジョーカーを含まない一組のデッキをシャッフルし、デッキ中のカードを (次の図のように) カスケードと呼ばれる 8 つの列に表向きに配ります。



プレイヤーの目標は、すべてのカードをゲーム盤の右上のホームセルと呼ばれる 4 つの山へ移動することです。ただし、カードの移動の仕方には以下のような制限があります。

1. 1 度に移動できるカードは 1 枚だけです。
2. 1 つのホームセルには、同じスートのカードを、エースから始めて、2、3、· · ·、10、ジャック、クイーン、キングの順にしか置けません。
3. 1 つのフリーセルには、1 枚のカードしか置けません。
4. 他のカードの下に (一部が) 隠されているカードは移動できません。
5. ホームセルに置かれたカードは移動できません。
6. カスケードには、最後の (他のカードに隠れていない) カードと異なる色で、ランクの表す数が 1 つ小さいカードだけが追加できます。ただし、空になっているカスケードにはどのカードでも置けます。

この制限にあるように、本来、1 度に移動できるカードは 1 枚だけですが、たとえば、空いているフリーセル (や空のカスケード) が 3 個以上あって、あるカスケードの末尾が、赤の 7、黒の 6、赤の 5、黒の 4 のようになっていると、この 4 枚のカードを、空いているフリーセル (や空のカスケード) を

利用して、別のカスケードの黒の 8 のカードにつなげることが可能です。通常、フリーセルのプログラムでは、このような移動も許すようになっています。

メモ

付録の FreeCellPanel.java というプログラムは、カードゲーム向けライブラリに用意されている GamePanel クラスのサブクラスとして、フリーセルを行うことのできるゲーム盤のクラス FreeCellPanel を宣言したものです。GamePanel クラスは JPanel クラスのサブクラスですから、次のプログラム G504.java のように、この FreeCell クラスのインスタンスを JFrame のコンテンツペインに追加し、FreeCellPanel のインスタンスメソッド start を起動するようにすれば、フリーセルを遊べるプログラムとなります。

G504.java

```
 1 import java.awt.*;
 2 import javax.swing.*;
 3
 4 public class G504 implements Runnable {
 5     public void run() {
 6         JFrame f = new JFrame();
 7         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 8         Container p = f.getContentPane();
 9         FreeCellPanel freeCell = new FreeCellPanel();
10         p.add(freeCell);
11         f.pack();
12         f.setVisible(true);
13         freeCell.start();
14     }
15
16     public static void main(String[] args) {
17         SwingUtilities.invokeLater(new G504());
18     }
19 }
```

5.5 演習問題

1. 右の図のように「新しいゲーム」と「終了」の 2 つのメニュー項目を含む「フリーセル」というメニュー (とメニューバー) を持つフリーセルのプログラム G505.java を作成しなさい。「新しいゲーム」は FreeCellPanel の start メソッドを起動し、「終了」は System クラスのクラスメソッド exit を起動します。G504.java では、プログラムを起動すると、即座にカードの配布 (start の起動) が行われていましたが、G505.java では、「新しいゲーム」と

いうメニュー項目を選択しないと、カードの配布は行われません。

2. 付録の `FreeCellPanel.java` の `cardPicked` の定義を変更して、ホームセルに移動可能なカードがダブルクリックされると、そのカードをホームセルのいずれかに移動するようにしなさい。ただし、空のホームセルに移動する場合は、最も左にある空きセルに移動するようにしなさい。

5.6 付録: FreeCellPanel.java

FreeCellPanel.java は、フリーセルを行うためのゲーム盤のクラス FreeCellPanel と、そこで使用する Cell、Home、Free、Cascade の 4 つのクラス宣言を行ったプログラムです。

抽象クラス　ホームセルを表すためのクラス Home、フリーセルを表すためのクラス Free、カスケードを表すためのクラス Cascade の 3 つのクラスが宣言されていますが、この 3 つの共通のスーパークラスとして、Cell が宣言されています。Cell は Pile のサブクラスですが、movableFrom と movableTo という 2 つの抽象メソッド[11]の宣言を含んでいます。

抽象メソッドの宣言を含むクラスは**抽象クラス**と呼ばれ、abstract という修飾子を付して宣言します。一般のクラス宣言はオブジェクトの設計図と考えることができますが、抽象クラスは、その一部が未完成の設計図と考えることができ[12]、そのサブクラスで抽象メソッドの実現方法を追加してはじめてオブジェクトの設計図として働けるようになります。

クラスライブラリの機能　FreeCellPanel.java はカードゲーム向けクラスライブラリを利用して作成されていますが、これまでには紹介していない以下のような機能が使用されています。

- Pile のコンストラクタ Pile(double dx, double dy) — 山のカードを、1 枚当たり (dx, dy) だけ位置をずらして配置するような山にする。引数のないコンストラクタ Pile() の起動は Pile(0.0, -0.15) の起動と等価となる。(8 行目)
- Card のインスタンスメソッド void setSticky(boolean s) — そのカードをドラッグしたときに、そのカードの上に乗っている (同じ山の) カードも一緒に移動するかどうかを設定する。s が false なら、そのカードは単独で、s が true なら、そのカードの上に乗っているカードも一緒に移動する。(107 行目)
- Pile のインスタンスメソッド void remove() — その山をゲーム盤から取り除く。(117 行目)
- Card のインスタンスメソッド void moveAsyncTo(Pile p) — そのカードを山 p へ非同期的に移動する。非同期的な移動では、その移動を開始したら、移動が完了する前にメソッド呼び出しから戻る。(120 行目)
- Pile のインスタンスメソッド void moveCardsAsync(Card c, Pile p) — この山のカード c と、それより上にあるすべてのカードを、山 p へ非同期的に移動する。(168 行目)

FreeCellPanel.java

```
 1 import jp.ac.ryukoku.math.cards.*;
 2
 3 abstract class Cell extends Pile {
 4     Cell() {
 5     }
 6
 7     Cell(double dx, double dy) {
 8         super(dx, dy);
```

[11] 第 3 回で説明しました

[12] 未完成の設計図ですので、抽象クラスのインスタンスを生成することはできません。抽象クラスが含んでいる抽象メソッドの具体的な定義を与えたサブクラスを宣言して、そのクラスのインスタンスを生成することになります。

```
 9     }
10
11     /* card 以下のカードがfree個の空きで移動可能かどうかを戻す */
12     abstract boolean movableFrom(Card card, int free);
13
14     /* from の card 以下の列がここへ移動可能かどうかを戻す */
15     abstract boolean movableTo(Cell from, Card card, int free);
16 }
17
18 class Free extends Cell {
19     boolean movableFrom(Card card, int free) {
20         return true;
21     }
22
23     boolean movableTo(Cell from, Card card, int free) {
24         return isEmpty() && from != null && from.top() == card;
25     }
26 }
27
28 class Home extends Cell {
29     boolean movableFrom(Card card, int free) {
30         return false;
31     }
32
33     boolean movableTo(Cell from, Card card, int free) {
34         if (from == null || from.top() != card) {
35             return false;
36         }
37         if (isEmpty()) {
38             return card.rank == Rank.ACE;
39         }
40         Card top = top();
41         return card.suit == top.suit
42                 && card.rank.getNumber() == top.rank.getNumber() + 1;
43     }
44 }
45
46 class Cascade extends Cell {
47     Cascade() {
48         super(0.0, 30.0);
49     }
50
51     boolean movableFrom(Card card, int free) {
52         Card prev = null;
53         for (Card c : getCards()) {
54             if (prev != null) {
55                 if (free-- <= 0) {
56                     return false;
57                 }
58                 if (c.isRed() == prev.isRed() || c.rank.getNumber()
59                         != prev.rank.getNumber() - 1) {
60                     return false;
61                 }
62                 prev = c;
63             } else if (c == card) {
64                 prev = c;
65             }
66         }
67         return true;
68     }
69
70     boolean movableTo(Cell from, Card card, int free) {
71         if (isEmpty()) {
72             /* 空のカスケードへの移動はfreeが1つ減るので再チェック */
73             return from.movableFrom(card, free - 1);
```

```
 74         }
 75         Card top = top();
 76         return card.isRed() != top.isRed()
 77                 && card.rank.getNumber() == top.rank.getNumber() - 1;
 78     }
 79 }
 80
 81 public class FreeCellPanel extends GamePanel
 82         implements CardListener {
 83     Cell[] freeCells = new Cell[4];
 84     Cell[] homeCells = new Cell[4];
 85     Cell[] cascades = new Cell[8];
 86
 87     public FreeCellPanel() {
 88         for (int i = 0; i < freeCells.length; i++) {
 89             freeCells[i] = new Free();
 90             add(freeCells[i], 30 + i * 90, 30);
 91         }
 92         for (int i = 0; i < homeCells.length; i++) {
 93             homeCells[i] = new Home();
 94             add(homeCells[i], 420 + i * 90, 30);
 95         }
 96         for (int i = 0; i < cascades.length; i++) {
 97             cascades[i] = new Cascade();
 98             add(cascades[i], 40 + i * 90, 180);
 99         }
100     }
101
102     /* 新しいゲームをスタートさせる */
103     public void start() {
104         reset();
105         Deck d = new Deck();
106         for (Card c : d.getCards()) {
107             c.setSticky(true);
108         }
109         d.shuffle();
110         d.flip();
111         add(d, 20, 620);
112         int i = 0;
113         Card[] cards = d.getCards();
114         while (!d.isEmpty()) {
115             d.pick();
116         }
117         d.remove();
118         for (Card c : cards) {
119             c.addCardListener(this);
120             c.moveAsyncTo(cascades[i++ % cascades.length]);
121         }
122     }
123
124     /* ゲームの状態をリセットする */
125     public void reset() {
126         for (Pile p : freeCells) {
127             p.clear();
128         }
129         for (Pile p : homeCells) {
130             p.clear();
131         }
132         for (Pile p : cascades) {
133             p.clear();
134         }
135     }
136
137     /* カードの一時退避場所の数を数えて戻す */
```

```
138     public int countFree() {
139         int free = 0;
140         for (Pile p : freeCells) {
141             if (p.isEmpty()) {
142                 free++;
143             }
144         }
145         for (Pile p : cascades) {
146             if (p.isEmpty()) {
147                 free++;
148             }
149         }
150         return free;
151     }
152
153     /* カードが選択されたときに起動される */
154     public boolean cardSelected(CardEvent e) {
155         Cell cell = (Cell) e.getPile();
156         cell.raiseAsync();
157         return cell.movableFrom(e.getCard(), countFree());
158     }
159
160     /* カードのドラッグが終了するときに起動される */
161     public boolean cardMoved(CardEvent e) {
162         Card card = e.getCard();
163         Cell from = (Cell) e.getPile();
164         Cell to = (Cell) e.getDest();
165         if (from == null || to == null) {
166             return false;
167         }
168         if (to.movableTo(from, card, countFree())) {
169             from.moveCardsAsyncTo(card, to);
170             return true;
171         }
172         return false;
173     }
174
175     /* カードがダブルクリックされたときに起動される */
176     public void cardPicked(CardEvent e) {
177     }
178 }
```

グラフィックス及び演習・第5回・終わり