# グラフィックス及び演習・第 7 回

2024 年 7 月 29 日

今回の内容



## 7.1 スレッド

第 3 回の「イベントディスパッチスレッド」の節では、Java が、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ことが可能で、それぞれの実行の流れを**スレッド** (thread) と呼ぶことを説明しました。

Java プログラムが起動されると、起動したクラスのクラスメソッド `main` を実行するために、main スレッドと呼ばれるスレッドが生成されます。基本的には、`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。GUI を持つプログラムの場合は、この `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イベントディスパッチスレッドと呼ばれるスレッドが生成され、ウィンドウの内容の描画や、ユーザーの操作によって発生する各種のイベントの処理 (イベントハンドラの起動) を行います。このスレッドが終了しないと GUI プログラムは終了しません。

メモ

## 7.2 Thread クラス

Java では、スレッドを `java.lang.Thread` クラスのオブジェクトとして表現し、このオブジェクトを介してスレッドを操作します。

`Thread` クラス — スレッドを表す

| | |
|---|---|
| 主なコンストラクタ | |
| `Thread()` | 特に何もしない (空の `run` メソッドを持つ) スレッド |
| `Thread(Runnable r)` | `r.run()` を実行するスレッド |
| 主なクラスメソッド | |
| `static Thread currentThread()` | このメソッドを起動したスレッド (現在のスレッド) を戻す |
| `static void sleep(long ms)` | このメソッドを起動したスレッドを `ms` ミリ秒間停止する (その途中で割り込みを受けると、`InterruptedException` がスローされる) |
| 主なインスタンスメソッド | |
| `void interrupt()` | このスレッドに対して割り込みを掛ける |

| | |
|---|---|
| `boolean isAlive()` | このスレッドが実行中かどうかを戻す |
| `void join()` | このスレッドの実行が終了するのを待つ (待っている間に割り込みを受けると、`InterruptedException` がスローされる) |
| `void join(long ms)` | このスレッドの実行が終了するのを最大 `ms` ミリ秒待つ (待っている間に割り込みを受けると、`InterruptedException` がスローされる) |
| `void run()` | このインスタンスが `Runnable` オブジェクトを指定して生成されたものなら、そのオブジェクトの `run` メソッドを起動する。そうでないのなら何もしない |
| `void start()` | このスレッドの実行を開始する |

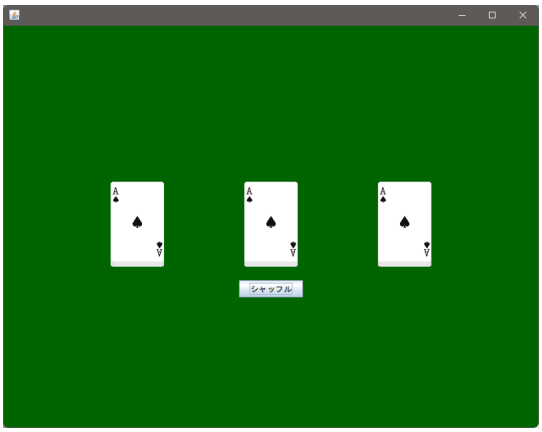
**スレッドの生成**　Java プログラム中で新しいスレッドを生成するためには、`Thread` クラスのインスタンスを生成し、そのインスタンスの `start` メソッドを起動します。`Thread` クラスのインスタンスの生成には、次の 2 通りの方法があり、`start` メソッドでスレッドの実行が開始されたときに起動されるメソッドが異なります。

**方法 1.** `Thread` のサブクラスを定義して、そのインスタンスを生成する— サブクラスでは `run` メソッドを再定義します。生成されたインスタンスに対して `start` メソッドを起動すると、新たに生成された実行の流れは、この再定義された `run` メソッドを実行します。

**方法 2.** `Runnable` を実装したオブジェクトをコンストラクタの引数として、`Thread` のインスタンスを生成する— この場合、`start` メソッドの起動で新たに生成される実行の流れは、コンストラクタの引数となった (`Runnable` 型の) オブジェクトの持つ `run` メソッドを実行します。

メモ

**スレッド生成を行うプログラム例**　スレッド生成を行うプログラムの例を紹介します。次のプログラム `G701.java` は、右の図のように、3 つの表向きのデッキと、1 つのボタンをゲーム盤に置き、ボタンがクリックされると、左から順に 3 つのデッキを、すべてシャッフルするようにしたものです。




このプログラムでは「方法 1」を使ってスレッドを生成しています。ゲーム盤のクラスである `G701Panel` のメンバクラスとして、`Thread` クラスのサブクラス `G701Shuffle` を宣言し (25 ～ 31 行目)、このクラスのインスタンスを生成して `start` しています (21 ～ 22 行目)。

G701.java

```
 1 import javax.swing.*;
 2 import jp.ac.ryukoku.math.cards.*;
 3
 4 class G701Panel extends GamePanel {
 5     Deck[] decks = new Deck[3];
 6     JButton button = new JButton("シャッフル");
 7
 8     G701Panel() {
 9         int x = 160;
10         for (int i = 0; i < decks.length; i++) {
11             decks[i] = new Deck();
12             decks[i].flip();
13             add(decks[i], x, 240);
14             x += 200;
15         }
16         add(button, 352, 380);
17         button.addActionListener(e -> shuffleAll());
18     }
19
20     void shuffleAll() {
21         Thread th = new G701Shuffle();
22         th.start();
23     }
24
25     class G701Shuffle extends Thread {
26         public void run() {
27             for (Deck d : decks) {
28                 d.shuffle();
29             }
30         }
31     }
32 }
33
34 class G701 implements Runnable {
35     public void run() {
36         JFrame f = new JFrame();
37         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
38         f.add(new G701Panel());
39         f.pack();
40         f.setVisible(true);
41     }
42
43     public static void main(String[] args) {
44         SwingUtilities.invokeLater(new G701());
45     }
46 }
```

ボタンがクリックされると、イベントディスパッチスレッドは、そのボタンに ActionListener として登録されたイベントハンドラ (17 行目のラムダ式) を起動し、そこから G701Panel クラスのインスタンスメソッド shuffleAll (20 〜 23 行目) を呼び出します。ここで、Thread のサブクラス G701Shuffle のインスタンスが生成されて (21 行目)、start の起動により、新しいスレッドの実行が開始されます (22 行目)。

新しいスレッドの実行を開始したら、その終了を待たずに、start の起動は直ちに終了しますので、shuffleAll の呼び出し (とラムダ式の実行) が完了して、イベントディスパッチスレッドによ

る実行の流れは、イベントループへ戻ることになります。

一方、start の起動により実行が開始された新しいスレッドは、G701Shuffle の run メソッド (26 ～ 30 行目) を起動します。この run メソッドでは、3 つのデッキを左から順にシャッフルしていますが、この実行の流れと並行して、イベントディスパッチスレッドも実行されていますので、そこでゲーム盤の再描画が行われ、デッキがシャッフルされていく様子がゲーム盤に表示されることになります。

もし、新たなスレッドを生成せずに、単に shuffleAll メソッドを

```
void shuffleAll() {
    for (Deck d : decks) {
        d.shuffle();
    }
}
```

のように宣言すると、確かに 3 つのデッキはシャッフルされますが、その作業はイベントディスパッチスレッドで行われることになりますので、シャッフルが完了するまではイベントループへは戻らず、ゲーム盤の表示の更新も行われません。シャッフルが完了して初めて (完了後の様子に) 表示が切り替わることになります[1]。

メモ

**方法2 によるスレッドの生成**　G701.java と同じ動作は、「方法 2」を用いてスレッドを生成して実現することもできます。G701Shuffle を Thread のサブクラスとして宣言する代りに、次のように Runnable を実装したクラスとして宣言し、

```
*25     class G701Shuffle implements Runnable {
 26         public void run() {
 27             for (Deck d : decks) {
 28                 d.shuffle();
 29             }
 30         }
 31     }
```

21 行目は

```
*21         Thread th = new Thread(new G701Shuffle());
```

と書き換えます (変ったのは 21 行目と 25 行目だけです)。

この例を見て分かるように、新たなクラスを宣言してスレッド生成を行うのなら、方法 1 の方が簡潔に書けますが、すでに Runnable を実装したクラスがあって、そのクラスを利用する場合には、

[1] d.shuffle(); の部分を、d.shuffleAsync(); と変更すれば、シャッフルの様子が画面に表示されますが、3 つのデッキが (ほぼ) 同時にシャッフルされてしまいます。shuffleAsync は、デッキのシャッフルの開始し、その終了を待たずに呼び出し元へリターンするメソッドです。このため shuffleAll も直ちにリターンできるようにはなるものの、デッキを 1 つずつシャッフルすることができなくなります。

方法2が便利です[2]。また、後述のように、方法2ならラムダ式を利用することができます。

メモ

**匿名クラスやラムダ式を利用したスレッドの生成**　方法1では匿名クラスを、方法2では匿名クラスやラムダ式を利用すると、プログラムがより簡潔になります。たとえば、方法1の G701.java では、Thread のサブクラス G701Shuffle を、G701Panel のメンバクラスとして宣言していましたが、メンバクラスの宣言の代りに匿名クラスを利用して、shuffleAll メソッドを

```
void shuffleAll() {
    Thread th = new Thread() {
        public void run() {
            for (Deck d : decks) {
                d.shuffle();
            }
        }
    };
    th.start();
}
```

のように宣言することができます。また、方法2なら、ラムダ式を使って、

```
void shuffleAll() {
    Thread th = new Thread(e -> {
        for (Deck d : decks) {
            d.shuffle();
        }
    });
    th.start();
}
```

とすることができます。いずれの場合も、変数 th は特に必要ありませんので、それぞれ、

```
void shuffleAll() {
    new Thread() {
        ⋮
    }.start();
}
```

や

```
void shuffleAll() {
    new Thread(e -> {
        ⋮
    }).start();
}
```

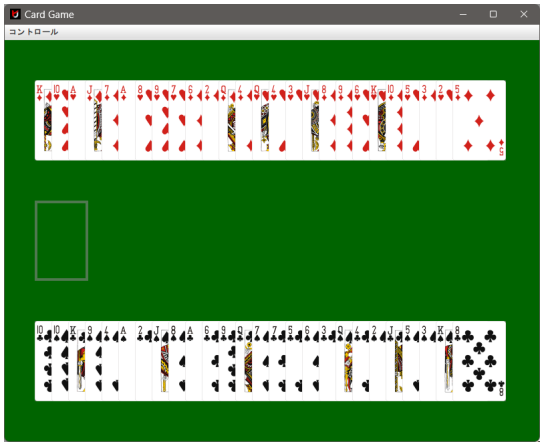
で十分です。

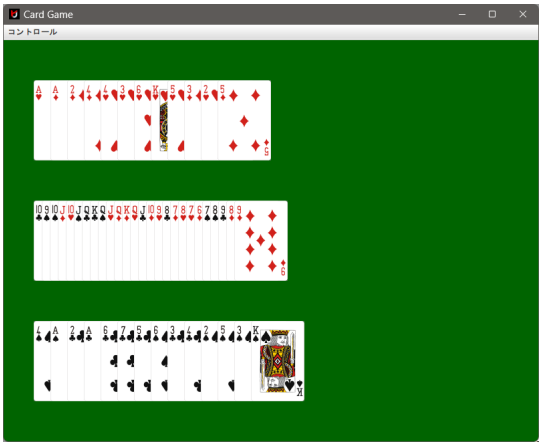
[2] Runnable インタフェースは Thread の生成以外にも、いろいろなところで利用されています。

## 7.3　マルチスレッドと排他制御

G701.java のような、複数のスレッドを利用したプログラミングを、一般に**マルチスレッドプログラミング (multithreaded programming)** と呼びます。マルチスレッドプログラミングでは、並行して実行される複数のスレッドが、オブジェクトの状態に、それぞれ勝手にアクセスすると、プログラムの動作に思わぬ不具合が生じることがあります。

次のプログラム G702.java は、シャッフルした表向きのデッキから、赤のカード (ハートとダイヤ) と黒のカード (スペードとクラブ) を、上下 2 つの山 (Pile) に分類した後 (下図左)、中段の (最初は空の) 山に、その 2 つの山からカードを移動していくプログラムです (下図右)。ただし、中段の山へ移動できるカードは、その 1 番上のカードとランクが 1 つ違いのカードだけです。中段の山が空のとき (最初だけ) は、どのカードでもそこに移動することができます。




不完全な G702.java

```
 1 import javax.swing.*;
 2 import jp.ac.ryukoku.math.cards.*;
 3
 4 class G702Target extends Pile {
 5     G702Target() {
 6         super(12, 0);
 7     }
 8
 9     boolean tryToAdd(Pile from, Card c) {
10         int num = c.rank.getNumber();
11         Card t = top();
12         if (t == null || t.rank.getNumber() == num - 1
13                 || t.rank.getNumber() == num + 1) {
14             from.remove(c);
15             c.moveTo(this);
16             return true;
17         }
18         return false;
19     }
20 }
21
22 class G702 extends GameFrame implements Runnable {
23     G702Target target;
24
25     public void run() {
26         Pile reds = new Pile(25, 0);
```

```
27          Pile blacks = new Pile(25, 0);
28          target = new G702Target();
29          Deck d = new Deck();
30          d.shuffle();
31          d.flip();
32          while (!d.isEmpty()) {
33              Card c = d.pick();
34              c.moveTo(c.isRed() ? reds : blacks);
35          }
36          add(reds, 45, 60);
37          add(blacks, 45, 420);
38          add(target, 45, 240);
39          startJob(reds);
40          startJob(blacks);
41      }
42
43      void startJob(final Pile from) {
44          new Thread(() -> {
45              while (!from.isEmpty()) {
46                  for (Card c : from.getCards()) {
47                      if (target.tryToAdd(from, c)) {
48                          break;
49                      }
50                  }
51              }
52          }).start();
53      }
54
55      public static void main(String[] args) {
56          SwingUtilities.invokeLater(new G702());
57      }
58 }
```

このプログラムの 39 行目と 40 行目では、startJob というメソッド (43 〜 53 行目) を起動し、新たなスレッドを生成しています。生成された 2 つのスレッドは、それぞれ、赤の山 (reds) から中段の山 (target) へのカードの移動の作業と、黒の山 (blacks) から中段の山 (target) へのカードの移動の作業を行います。

メモ

中段の山は、G702Target クラス (Pile のサブクラス) のインスタンスとして実現されていて、そのクラスには、tryToAdd というインスタンスメソッドが宣言されています (9 〜 19 行目) が、このメソッドは、引数として渡されたカードのランクが山の 1 番上のカードと 1 つ違いであるか、あるいは山が空であるときのみ、そのカードを山に追加して true を戻り値として返します。追加できない場合は false を返します。

生成された 2 つのスレッドは、それぞれ、赤あるいは黒の山に残っているカードに対し、順に、そ

のカードを引数として tryToAdd を起動することで、カードを中段の山に移動していきます (45 ～ 51 行目)。これがうまく行けば、中段の山には (たとえば) 次のようにカードが並ぶはずです。

しかし、実際には

のように並んでしまうことがあります。

それぞれ、赤の山と黒の山を担当している 2 つのスレッドは、tryToAdd を起動し、中段の山の状況を調べて (可能なら) カードを移動しますので、たとえば、1 番上のカードのランクが 5 である場合、一方のスレッドが赤の山から 6 のカードを、他方のスレッドは黒の山から 4 のカードを移動しようとしてしまうと、中段の山には、5、6、4、あるいは、5、4、6、の順にカードが並んでしまうことになります。

メモ

tryToAdd メソッドでは、(中段の) 山の 1 番上のカードのチェックして、もし許されるなら、その山へカードを移動するという作業を行っていますが、この一連の作業を分割できないひとまとまりの操作として扱うようにし、

あるスレッドがこの操作を行っている間、他のスレッドはこの操作を行えない

ようにすることができれば、このような不具合を解消できるはずです。

プログラムの特定の部分が、複数のスレッドによって同時に実行されないようにすることを**排他制御**あるいは**相互排他 (mutual exclusion)**と呼びます[3]。また、プログラムの中で、複数のスレッドによって同時並行的に実行されることが許されない部分を**クリティカルセクション (critical section)** と呼びます。G702.java の例では、tryToAdd メソッドの本体の部分をクリティカルセクションとして、この部分が複数のスレッドによって同時に実行されないように排他制御を行う必要があると考えられます。

[3] 一般には、計算機の同一の資源へ複数のスレッドが同時にアクセスできないようにすることを排他制御と呼びます。多くの場合、プログラムの特定部分の実行について排他制御を行うことで、いろいろな資源に関する排他制御を実現します。

メモ

**モニタ**　Java のすべてのオブジェクトには、排他制御を行うための道具として**モニタ (monitor)** と呼ばれる仕組み[4]が、それぞれ 1 つ付属しています。Java の各オブジェクトに備え付けられているモニタは、そのオブジェクトが使用中であることの印として働きます。使用中である印をつけることを**ロック (lock)**する、あるいは**ロックを設定**すると言い、使用中の印を消すことを**アンロック (unlock)**する、あるいは**ロックを解除** すると言います[5]。

1 つのオブジェクト (のモニタ) を複数のスレッドがロックすることはできません。すでに他のスレッドがロックしているオブジェクトをロックしようとしたスレッドは、そのロックが解除されるまで待たされ、ロックが解除されてはじめて、そのスレッドでのロックが可能になります[6]。

メモ

**synchronized 文と synchronized 修飾子**　モニタを利用すると、クリティカルセクションの排他制御ができるようになります。クリティカルセクションに入る前に、特定のオブジェクト (のモニタ) のロックを獲得し、その実行が終わった時点で、そのロックを解除するようにすれば、クリティカルセクションが複数のスレッドによって同時に実行されるのを防ぐことができます。

Java でこれを行うためには、次のような書式の synchronized 文と呼ばれる構文を用いることができます。

synchronized （オブジェクトを表す式） 文

この synchronized 文が実行されるときには、まず、オブジェクトを表す式 が示すオブジェクトをロックしようとします。他のスレッドがすでにそのオブジェクトをロックしている場合は、この synchronized 文の実行は、そこで一旦停止し、ロックが解除されるのを待ちます。オブジェクト

[4]モニタは、マルチスレッドプログラミングで排他制御を行うための一般的な概念で、Java の各オブジェクトに備え付けられているモニタはその一例です。

[5]「オブジェクト ··· のモニタを (アン) ロックする」と言うこともあれば、単に、「オブジェクト ··· を (アン) ロックする」と言うこともあります。

[6]自分自身のスレッドがロックしているオブジェクトは、何重にでもロックすることができます。そのスレッドが行ったロックと同じ回数だけ (そのスレッドが) アンロックを行った時にはじめて、他のスレッドがそのオブジェクトをロックできるようになります。

のロックに成功すると、文 の部分が実行されます。文 の実行が完了すると、獲得したロックが解除されます。

メモ

G702.java の不具合を解決するためには、たとえば、tryToAdd メソッドの宣言を、次のように変更します (* を付けた行が変更されています)。

```
  9     boolean tryToAdd(Pile from, Card c) {
*10         synchronized (this) {
 11             int num = c.rank.getNumber();
 12             Card t = top();
 13             if (t == null || t.rank.getNumber() == num - 1
 14                     || t.rank.getNumber() == num + 1) {
 15                 from.remove(c);
 16                 c.moveTo(this);
 17                 return true;
 18             }
 19             return false;
*20         }
 21     }
```

あるスレッドで、このように変更されたインスタンスメソッド tryToAdd が起動されると、10 行目から始まる synchronized 文で、起動の対象となってる G702Target クラスのインスタンス (移動先の山) のモニタがロックされますので、このスレッドが synchronized 文の実行を終了するまでは、他のスレッドは、この部分の実行を開始することができません。

この例のように、あるインスタンスメソッドの本体の実行すべてを、そのインスタンスメソッドの起動の対象となっているオブジェクトをロックして行いたい場合は、synchronized 文を使う代りに、

```
* 9     synchronized boolean tryToAdd(Pile from, Card c) {
 10         int num = c.rank.getNumber();
 11         Card t = top();
 12         if (t == null || t.rank.getNumber() == num - 1
 13                 || t.rank.getNumber() == num + 1) {
 14             from.remove(c);
 15             c.moveTo(this);
 16             return true;
 17         }
 18         return false;
 19     }
```

のように、synchronized という修飾子を付けてインスタンスメソッドを宣言することもできま

す[7]。synchronized 修飾子を伴って宣言されたインスタンスメソッドが起動される際には、まず、起動の対象となっているオブジェクトのロックが行われ、ロックが獲得されてから、メソッド宣言の本体が実行されます。メソッド本体の実行が終わって呼び出し元にリターンする時にロックが解除されます。

メモ

**モニタのロックと排他制御**　モニタのロックは、あくまでそのオブジェクトに使用中の印を付けるだけですので、他のスレッドにその印を無視されてしまうと意味はありません。たとえ、あるスレッドがオブジェクトをロック中であっても、他のスレッドは、その同じオブジェクトのインスタンス変数にアクセスしたり、(synchronized 修飾子を持たない) インスタンスメソッドを起動したりすることが可能です。モニタのロックを利用した排他制御を機能させるには、クリティカルセクションを実行する可能性のあるすべてのスレッドで、同一のオブジェクトのモニタをロックしてから、その実行を行うようにプログラムを書いておく必要があります。

また、オブジェクトを排他的に使用するための印としては、必ずしもそのオブジェクトのモニタを使用する必要はありません。特定のオブジェクトのモニタのロックを獲得するようにプログラムを書いておきさえすれば、たとえロック対象のオブジェクトが別のオブジェクトであっても排他制御は可能となります。たとえば、G702Target クラスの場合、

```
class G702Target extends Pile {
    Object lock = new Object();          // 排他制御専用のオブジェクト

    G702Target() {
        super(12, 0);
    }
```

のように、インスタンス生成時に排他制御用のオブジェクトを一緒に作って (インスタンス変数に記憶して) おき、tryToAdd では、this をロックする代りに、

```
    boolean tryToAdd(Pile from, Card c) {
        synchronized (lock) {            // this でなく lock をロックする
            int num = c.rank.getNumber();
             :
            return false;
        }
    }
```

のように、このオブジェクトをロックするようにしても問題なく排他制御を行うことができます。

[7]インスタンスメソッドだけではなく、クラスメソッドに対しても synchronized 修飾子を付けることができます。その場合、クラス毎に 1 つ用意されているモニタがロックされてから、クラスメソッドの本体が実行されます。

メモ

**デッドロック**　マルチスレッドプログラミングで排他制御を行う際に十分注意しないといけない点として、**デッドロック (dead lock)**と呼ばれる現象があります。1 つのスレッドが獲得しなければならないロックが複数あって、そのようなスレッドが複数並行して実行されていると、あるオブジェクトをロック中のスレッドが、別のスレッドがロック中のオブジェクトをロックしようとして待ち状態になっているが、その別のスレッドは、その待ち状態なっているスレッドがロックしているオブジェクトをロックしようとして、やはり待ち状態になってしまっているということが起ります。このような状態をデッドロックと呼び、デッドロックが発生すると、それに巻き込まれているスレッドは、どれも永遠に待ち状態から抜け出すことができなくなってしまいます。

ロックしなければならないオブジェクトが複数ある場合は、それらを決まった順番でロックするようにプログラムを書いたり、複数のロックを統合して 1 つのロックで表現したりすることで、1 つのスレッドがすでに獲得しているロックと、これから獲得しなければならないロックの関係に循環が生じないようにすることが必要となります。

メモ

## 7.4　演習問題

1. 付録の `FreeCellPanel.java` は、第 5 回で紹介したフリーセルを行うためのゲーム盤のクラスを次のように機能強化したものです。(b) 〜 (d) の機能は、新たなスレッドを生成することで実現しています。

   (a) ホームセルに移動できるカードをダブルクリックすると、そのカードをホームセルに移動するようにした (第 5 回の演習問題 2)。

   (b) どこにも移動できないカードをダブルクリックすると、ホームセルに移動可能なカードを探し、そのようなカードがあるかぎり、すべてホームセルに移動するようにした。

   (c) ゲームが開始されたとき、全部のカードを一度にカスケードに配るのではなく、1 枚ずつ配るようにした。

(d) 経過時間を計測して、0.1 秒単位でゲーム盤の左上に表示するようにした。また、すべてのカードがホームセルに移動できたら経過時間の更新を止めるようにした。

(c) と (d) が実現されるように、付録のプログラムの空欄 (1) を埋め、新たなスレッドを生成して、そのスレッドで `FreeCellPanel` クラスの `run` メソッドを起動するようにしなさい。

2. 付録の `FreeCellPanel` クラスの宣言では、(b) を実現するための補助的なメソッドとして、`finish` (とさらにその補助として `moveHome`) が宣言されています。付録の (2) の空欄を埋めて、新たなスレッドを生成し、そのスレッドからこの `finish` メソッドを起動するようにしなさい。

## 7.5 付録: 新しい FreeCellPanel.java

第 5 回の FreeCellPanel.java から変更された部分には * が記してあります。86 行目と 87 行目の volatile は、この変数の値を参照するときは、常に最新の値を参照しなければならないということを示す修飾子です。異なるスレッドが変数に代入した値は、必ずしも代入してすぐさま他のスレッドから参照可能となるわけではありません。Java では、スレッド毎に変数のキャッシュコピーを持つことが許されており、オブジェクトのロックが行われる際に、その同期がとられます。変数を volatile と宣言しておけば、特にロックを行わなくても、スレッド間で共有された変数でも最新の値にアクセスできるようになります。

FreeCellPanel.java

```
*  1 import java.awt.*;
   2 import jp.ac.ryukoku.math.cards.*;
   3
   4 abstract class Cell extends Pile {
   5     Cell() {
   6     }
   7
   8     Cell(double dx, double dy) {
   9         super(dx, dy);
  10     }
  11
  12     /* 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 */
  13     abstract boolean movableFrom(Card card, int free);
  14
  15     /* from の card 以下の列がここへ移動可能かどうかを戻す */
  16     abstract boolean movableTo(Cell from, Card card, int free);
  17 }
  18
  19 class Free extends Cell {
  20     boolean movableFrom(Card card, int free) {
  21         return true;
  22     }
  23
  24     boolean movableTo(Cell from, Card card, int free) {
  25         return isEmpty() && from != null && from.top() == card;
  26     }
  27 }
  28
  29 class Home extends Cell {
  30     boolean movableFrom(Card card, int free) {
  31         return false;
  32     }
  33
  34     boolean movableTo(Cell from, Card card, int free) {
  35         if (from == null || from.top() != card) {
  36             return false;
  37         }
  38         if (isEmpty()) {
  39             return card.rank == Rank.ACE;
  40         }
  41         Card top = top();
  42         return card.suit == top.suit
  43                 && card.rank.getNumber() == top.rank.getNumber() + 1;
  44     }
  45 }
  46
  47 class Cascade extends Cell {
  48     Cascade() {
  49         super(0.0, 30.0);
  50     }
```

```
  51
  52      boolean movableFrom(Card card, int free) {
  53          Card prev = null;
  54          for (Card c : getCards()) {
  55              if (prev != null) {
  56                  if (free-- <= 0) {
  57                      return false;
  58                  }
  59                  if (c.isRed() == prev.isRed() || c.rank.getNumber()
  60                          != prev.rank.getNumber() - 1) {
  61                      return false;
  62                  }
  63                  prev = c;
  64              } else if (c == card) {
  65                  prev = c;
  66              }
  67          }
  68          return true;
  69      }
  70
  71      boolean movableTo(Cell from, Card card, int free) {
  72          if (isEmpty()) {
  73              /* 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 */
  74              return from.movableFrom(card, free - 1);
  75          }
  76          Card top = top();
  77          return card.isRed() != top.isRed()
  78                  && card.rank.getNumber() == top.rank.getNumber() - 1;
  79      }
  80  }
  81
  82  public class FreeCellPanel extends GamePanel
* 83          implements CardListener, Runnable {
  84      Cell[] freeCells = new Cell[4];
  85      Cell[] homeCells = new Cell[4];
  86      Cell[] cascades = new Cell[8];
* 87      volatile long time;
* 88      volatile Thread thread;
  89
  90      public FreeCellPanel() {
  91          for (int i = 0; i < freeCells.length; i++) {
  92              freeCells[i] = new Free();
  93              add(freeCells[i], 30 + i * 90, 30);
  94          }
  95          for (int i = 0; i < homeCells.length; i++) {
  96              homeCells[i] = new Home();
  97              add(homeCells[i], 420 + i * 90, 30);
  98          }
  99          for (int i = 0; i < cascades.length; i++) {
 100              cascades[i] = new Cascade();
 101              add(cascades[i], 40 + i * 90, 180);
 102          }
 103      }
 104
*105      /* ゲーム盤を再描画する */
*106      protected void paintComponent(Graphics g) {
*107          super.paintComponent(g);
*108          if (time > 0) {
*109              g.setColor(Color.WHITE);
*110              g.drawString(String.format("経過時間 %.1f 秒",
*111                      time / 1000.0), 30, 20);
*112          }
*113      }
*114
```

```
 115     /* 新しいゲームをスタートさせる */
 116     public void start() {
*117         if (thread != null) {
*118             Thread oldThread = thread;
*119             thread = null;
*120             try {
*121                 oldThread.join();
*122             } catch (InterruptedException e) {
*123             }
*124         }
*125         thread = [          (1)          ]
*126         thread.start();
*127     }
*128
*129     /* カスケードにカードを配った後、経過時間の更新を続ける */
*130     public void run() {
 131         reset();
 132         Deck d = new Deck();
 133         for (Card c : d.getCards()) {
 134             c.setSticky(true);
 135         }
 136         d.shuffle();
 137         d.flip();
 138         add(d, 20, 620);
 139         int i = 0;
 140         Card[] cards = d.getCards();
 141         while (!d.isEmpty()) {
 142             d.pick();
 143         }
 144         d.remove();
 145         for (Card c : cards) {
*146             if (thread == null) {
*147                 break;
*148             }
 149             c.addCardListener(this);
*150             c.setSpeed(5000);
*151             c.moveTo(cascades[i++ % cascades.length]);
*152             c.setSpeed(Card.DEFAULT_SPEED);
 153         }
*154         try {
*155             long startTime = System.currentTimeMillis();
*156             while (thread != null && countFree() < 12) {
*157                 Thread.sleep(100);
*158                 time = System.currentTimeMillis() - startTime;
*159                 repaint();
*160             }
*161         } catch (InterruptedException e) {
*162         }
 163     }
 164
 165     /* ゲームの状態をリセットする */
 166     public void reset() {
 167         for (Pile p : freeCells) {
 168             p.clear();
 169         }
 170         for (Pile p : homeCells) {
 171             p.clear();
 172         }
 173         for (Pile p : cascades) {
 174             p.clear();
 175         }
 176     }
 177
```

```
*178     /* from の1番上のカードをホームセルに移動する
*179      移動できたら true を、できなかったら false を返す */
*180     private boolean moveHome(Cell from) {
*181         Card card = from.top();
*182         if (card != null) {
*183             for (Cell to : homeCells) {
*184                 if (to.movableTo(from, card, 0)) {
*185                     card.moveTo(to);
*186                     return true;
*187                 }
*188             }
*189         }
*190         return false;
*191     }
*192
*193     /* ホームセルに移動できるカードをすべて移動する */
*194     public void finish() {
*195         boolean moved;
*196         do {
*197             moved = false;
*198             for (Cell c : freeCells) {
*199                 moved |= moveHome(c);
*200             }
*201             for (Cell c : cascades) {
*202                 moved |= moveHome(c);
*203             }
*204         } while (moved);
*205     }
*206
 207     /* カードの一時退避場所の数を数えて戻す */
*208     public synchronized int countFree() {
 209         int free = 0;
 210         for (Pile p : freeCells) {
 211             if (p.isEmpty()) {
 212                 free++;
 213             }
 214         }
 215         for (Pile p : cascades) {
 216             if (p.isEmpty()) {
 217                 free++;
 218             }
 219         }
 220         return free;
 221     }
 222
 223     /* カードが選択されたときに起動される */
 224     public boolean cardSelected(CardEvent e) {
 225         Cell cell = (Cell) e.getPile();
 226         cell.raiseAsync();
 227         return cell.movableFrom(e.getCard(), countFree());
 228     }
 229
 230     /* カードのドラッグが終了するときに起動される */
 231     public boolean cardMoved(CardEvent e) {
 232         Card card = e.getCard();
 233         Cell from = (Cell) e.getPile();
 234         Cell to = (Cell) e.getDest();
 235         if (from == null || to == null) {
 236             return false;
 237         }
 238         if (to.movableTo(from, card, countFree())) {
 239             from.moveCardsAsyncTo(card, to);
 240             return true;
 241         }
 242         return false;
```

```
 243     }
 244
 245     /* カードがダブルクリックされたときに起動される */
 246     public void cardPicked(CardEvent e) {
*247         Card card = e.getCard();
*248         Cell from = (Cell) e.getPile();
*249         if (from == null) {
*250             return;
*251         }
*252         if (!from.movableFrom(card, 0)) {
*253                 (2)
*254             return;
*255         }
*256         for (Cell to : homeCells) {
*257             if (to.movableTo(from, card, 0)) {
*258                 card.moveAsyncTo(to);
*259                 return;
*260             }
*261         }
 262     }
 263 }
```

グラフィックス及び演習・第7回・終わり