

第6章 スレッド

この章では、スレッドという概念を学ぶ。スレッドとは、簡単に言うとプログラムの処理の流れである。スレッドは応用範囲の広い概念である。例えば Java GUI アプリケーションの場合、スレッドを用いるとアニメーションを実現できる。また、スレッドはネットワークプログラミングをする際にも必須の概念である。

GUI アプリケーションの `paintComponent` などのメソッド、あるいは GUI 部品のコールバックメソッド (`mouseClicked`, `keyPressed`, `actionPerformed` など) はイベントによって呼び出される。これらのメソッドが実行されている間は、アプリケーションは他の仕事をすることができない。このため、これらのメソッドはすぐに実行を終える必要がある。アニメーションやゲームのように何らかの動きがあるアプリケーションは、`mouseClicked` や `paintComponent` メソッドにその処理を直接記述することはできない。何らかの方法でアプリケーションのイベント処理と同時に、これらの処理を行わなければならない。

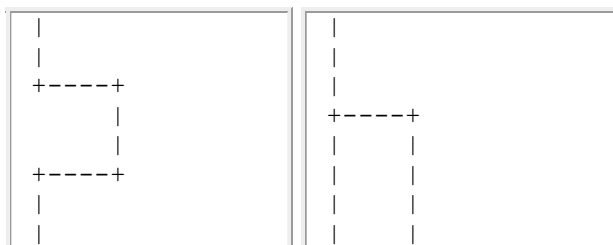
このようなコンピューターの処理を行なう単位を スレッド (thread, もともとの意味は“糸”) という (具体的にいえば、変数の値や、プログラムのどこを実行しているか、どのようなメソッドのどこから呼び出されたかなどの情報 (プログラムカウンタを含む CPU のレジスタ、およびスタックなどの情報) のことである。)。つまり、アニメーションやゲームの GUI アプリケーションはスレッドを複数必要とする (マルチスレッド, multi-thread)。CPU が 1 つしかないコンピューターでは、実際にはスレッドを短い時間で切り替えて実行し、並行に実行されているように見せかける。CPU が複数個あれば、スレッドを別々の CPU に割り当てて同時実行することも可能である。

これまで紹介したプログラムのように、スレッドが一つだけのプログラムは、シングルスレッドと言う。

関数呼出しとスレッドの比較

関数呼出し

スレッド



ここでは、Java で新しいスレッドを生成し実行するための方法を学ぶ。

6.1 スレッドの生成と実行

スレッドを生成するためには、その新しいスレッドが実行するメソッドを指定しなくてはならない。そのメソッドの名前は Java では run という無引数・戻り値なしのメソッドとすることが決まっている。run は、Runnable インタフェースの唯一のメソッドである。

次のような簡単な例では MyRunnable クラスが Runnable インタフェースを実装している。つまり、run というメソッドを持っている。

run メソッドの中身は単純な出力の繰り返して、繰り返しの途中で Thread.sleep というメソッドを呼んで 10 ミリ秒寝る（実行を止める）。

ファイル ThreadTest.java

```
1 class MyRunnable implements Runnable {
2     String name;
3
4     MyRunnable(String n) {
5         name = n;
6     }
7
8     public void run() {
9         int i;
10        for (i = 0; i < 10; i++) {
11            try {
12                Thread.sleep(10); // 10ミリ秒お休み
13            } catch (InterruptedException e) {
14            }
15            System.out.printf("%s: %d, ", name, i);
16        }
17    }
18 }
19
20 void main() {
21     Thread ta = new Thread(new MyRunnable("A"));
22     Thread tb = new Thread(new MyRunnable("B"));
23     Thread tc = new Thread(new MyRunnable("C"));
24     ta.start();
25     tb.start();
26     tc.start();
27 }
```

ThreadTest クラスに main 関数があり、ここでスレッドを生成している。Thread のコンストラクターの引数は Runnable を実装している必要がある。new 演算子で Thread オブジェクトを生成してスレッドを作成（つまり実行を準備）し、この Thread オブジェクトの start メソッドを呼び出して、実際にスレッドの実行を開始する。

下は、このプログラムの実行例である。各スレッドが並行に実行されていることがわかる。

```
A: 0, A: 1, B: 0, A: 2, A: 3, B: 1, A: 4, C: 0, A: 5, B:
2, A: 6, A: 7, B: 3, A: 8, C: 1, A: 9, B: 4, B: 5, B: 6,
C: 2, B: 7, C: 3, B: 8, C: 4, B: 9, C: 5, C: 6, C: 7, C:
8, C: 9,
```

もちろんスレッドが切り替わるタイミングによって、異なる出力になることもある。以下も同じプログラムの実行例である。

```
A: 0, C: 0, B: 0, A: 1, B: 1, C: 1, A: 2, B: 2, C: 2, A:
3, B: 3, C: 3, A: 4, B: 4, C: 4, A: 5, B: 5, C: 5, A: 6,
B: 6, C: 6, A: 7, B: 7, C: 7, A: 8, B: 8, C: 8, A: 9, B:
9, C: 9,
```

6.2 スレッドを利用した GUI アプリケーション

例題 6.2.1 ぐるぐる廻る

単に文字列が円の上を動くだけの簡単なスレッドを利用した GUI アプリケーションである。この程度なら `Timer` クラスを利用しても作成できるが、ここではスレッドを利用する方法を学ぶ。最初の例の `GuruguruTh` は `Timer` クラスのときに紹介した `Guruguru` と同じように動作するが、スレッドを利用したバージョンである。

ファイル `GuruguruTh.java`

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 class GuruguruTh extends JPanel
5     implements Runnable {
6     private int r = 50;
7     private volatile int x = 110, y = 70 ;
8     private double theta = 0; // 角度
9     private volatile Thread thread = null;
10
11     public GuruguruTh() {
12         setPreferredSize(new Dimension(200, 180));
13         JButton startBtn = new JButton("start");
14         startBtn.addActionListener(e -> startThread());
15         JButton stopBtn = new JButton("stop");
16         stopBtn.addActionListener(e -> stopThread());
17         setLayout(new FlowLayout());
18         add(startBtn); add(stopBtn);
19         startThread();
20     }
21
22     private void startThread() {
23         if (thread == null) {
24             thread = new Thread(this);
25             thread.start();
26         }
27     }
28
29     private void stopThread() {
30         thread = null;
31     }
32
33     @Override
34     public void paintComponent(Graphics g) {
35         // スーパークラスの paintComponent を呼び出す
36         super.paintComponent(g);
37         // 全体を背景色で塗りつぶす。
38         g.drawString("Hello, World!", x, y);
```

4 行めの `implements Runnable` に注意する。これで `GuruguruTh` クラスが `run` という名前のメソッドを持っていることを宣言する。`paintComponent` メソッドは座標 (x, y) に “Hello, World!” と表示するだけである。

このクラスは `thread` という `Thread` 型のフィールドを持っている。`thread` の初期値は `null` である。`null` は、[未生成のオブジェクトを表す値](#)（C 言語の `NULL` に相当する）で、`thread` に最初は意味のある値が割り当てられていないことを示す。また、`volatile` /`ˈvɒlət, aɪl`/（揮発性の、うつろいやすい、という意味）という修飾子は、スレッドがフィールドのキャッシュ（局所的なコピー）を作らないように指示する働きがある。これによってスレッドが（他のスレッドによって変更されたかもしれない）フィールドの最新の値を参照することを保証する。

スレッドはこのアプリケーションでは `startThread` メソッド内で生成される。`Thread` のコンストラクターの引数は、この場合は `this` — [つまり `GuruguruTh` クラスのオブジェクト自身](#) である。（実質的には、これは自身の `run` メソッドを指す。）`thread` を生成した後、この `thread` の `start` メソッドを起動してスレッドの実行をスタートしている。

一方 `stopThread` メソッドは、スレッドの実行を止めるためのメソッドである。

次に、`run` メソッドは次のように定義されている。

ファイル `GuruguruTh.java`

```

41 public void run() {
42     Thread thisThread = Thread.currentThread();
43     for (; thread == thisThread; theta += 0.02) {
44         x = 60 + (int)(r * Math.cos(theta));
45         y = 100 - (int)(r * Math.sin(theta));
46         repaint(); // paintComponent を間接的に呼出す
47         try {
48             Thread.sleep(30); // 30 ミリ秒お休み
49         } catch (InterruptedException e) {}
50     }
51 }
52 }
53
54 void main() { /* 省略 */ }
```

`Thread` クラスのクラスメソッド `currentThread` は、このメソッドを実行しているスレッドを返す。この時点では、フィールド `thread` と同じオブジェクトが返るはずである。

実際にスレッドが実行するメソッド（`run` メソッド）のループの条件式 `thread == thisThread` は奇妙に見えるが、`stopThread` メソッドによって、`thread` の値が `null` に変更されると、このループは終了し、スレッド自体も終了する。このような手法でスレッドを停止できるようにしておくのが一般

的である。ループの中では、 x と y の値を計算して再描画 (repaint) すると `Thread.sleep` を呼んで 30 ミリ秒スリープする。`Thread.sleep` は `InterruptedException` という例外を起こす可能性がある（その説明は割愛する）ので周りを `try ~ catch` で囲んでいる。

この程度のアニメーションであれば、`Timer` クラスを使ったほうが簡単に記述できるが、スレッドを使うとループの中など任意の場所で描画できて柔軟なアニメーションが可能になるので、両方の方法を理解しておくといよい。

GUI アプリケーションでスレッドを使うときには、通常

- `run` メソッドを定義する。メソッド内は、通常は `Thread` 型のフィールドの値を `null` に変更することによって、スレッドを停止できるようなループにしておく。
- クラスに `implements Runnable` を付け加える。
- `Thread` 型のフィールド（初期値 `null`）を追加する。
- コンストラクターなどでスレッドを生成する。

のようにする。

Q 6.2.2 `GuruguruTh.java` の `paintComponent` メソッドの定義の中で `super.paintComponent(g);` がなければどのような振る舞いになるか？

Q 6.2.3 `TextAnimation.java` は、文字列が右から左に移動し、左端まで移動すれば、再び右端から現れるアニメーションを表示する。

`TextAnimation.java` を完成させよ。

ファイル `TextAnimation.java`

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 class TextAnimation extends JPanel
5 {
6     private int x = 200;
7     private volatile Thread thread = null;
8
9     public TextAnimation() {
10         setPreferredSize(new Dimension(200, 150));
11         JButton startBtn = new JButton("start");
12         startBtn.addActionListener(e -> startThread());
13         JButton stopBtn = new JButton("stop");
14         stopBtn.addActionListener(e -> stopThread());
15         setLayout(new FlowLayout());
16         add(startBtn); add(stopBtn);
17         startThread();
18     }
19
20     // startThread, stopThread は GuruguruTh と同じ
```

```

21
22 @Override
23 public void paintComponent(Graphics g) {
24     // スーパークラスの paintComponent を呼び出す
25     super.paintComponent(g);
26     // 全体を背景色で塗りつぶす。
27     g.drawString("Hello, World!", x, 100);
28 }
29
30 public void run() {
31     Thread thisThread = Thread.currentThread();
32     while (true) {
33         x -= 5;
34         if (x < 0) { x = 200; }
35
36         try {
37             Thread.sleep(100); // 100 ミリ秒お休み
38         } catch (InterruptedException e) {}
39     }
40 }
41 }
42
43 // main メソッドは省略

```

Q 6.2.4 run メソッドをラムダ式として定義するように、GuruguruTh.java を書き換えよ。

ファイル GuruguruTh.java

```

1 import java.awt.*;
2 import javax.swing.*;
3
4 class GuruguruTh {
5     private int r = 50;
6     private volatile int x = 110, y = 70;
7     private double theta = 0; // 角度
8     private volatile Thread thread = null;
9
10    ... // コンストラクターは元のバージョンと同じ
11
12    private void startThread() {
13        if (thread == null) {
14            thread = new Thread(() {
15                /* 元のバージョンの run メソッドの中身と同じ */
16            });
17            thread.start();
18        }
19    }
20
21    // stopThread, paintComponent, main は元と同じ
22    ...
23 }

```

問 6.2.5 (ビリヤード?)

円が等速で斜めに動いて上下左右の壁にぶつかったとき、入射角と反射角が等しくなるように跳ね返るような GUI アプリケーションを書け。

問 6.2.6 次のプログラムは、どのようなアニメーションを描画するか？予想して確かめよ。（import 部と startThread, stopThread, main メソッドは、GuruguruTh.java とほとんど同じなので省略する。）

ファイル `Mystery.java`

```
4 class Mystery extends JPanel
5     implements Runnable {
6     static final double p = 0.8660254; // sqrt(3) / 2
7     int x = 50, y = 0, z = 0,
8         dx = 0, dy = 1, dz = 0;
9     Thread thread = null;
10
11     public Mystery() {
12         setPreferredSize(new Dimension(220, 220));
13         startThread();
14     }
15
16
17     @Override
18     public void run() {
19         Thread me = Thread.currentThread();
20         while (thread == me) {
21             for (int i = 0; i < 50; i++) {
22                 x += dx; y += dy; z += dz;
23                 repaint();
24                 try {
25                     Thread.sleep(40);
26                 } catch (InterruptedException e) {
27                 }
28             }
29             int tmp = dz; dz = -dx; dx = -dy; dy = -tmp;
30         }
31     }
32
33     @Override
34     public void paintComponent(Graphics g) {
35         super.paintComponent(g);
36         g.setColor(new Color((int) (x * 255.0 / 50.0),
37                               (int) (y * 255.0 / 50.0),
38                               (int) (z * 255.0 / 50.0)));
39         double x0 = 0.0 * x + p * y - p * z,
40                y0 = 1.0 * x - 0.5 * y - 0.5 * z;
41         g.fillOval(100 + (int) (x0 * 2),
42                   100 + (int) (y0 * 2), 20, 20);
43     }
44 }
```

6.3 SwingUtilities.invokeLater

このテキストで紹介している GUI ライブラリーの Swing は、シングルスレッドでアクセスするように設計されており、`paintComponent` や `actionPerformed` のようにイベントから起動されるメソッドのスレッドから GUI オブジェクトの状態を取得・変更しなければいけない、という制限がある。そのため、別に生成したスレッドから Swing のメソッドを呼び出すときは、`SwingUtilities.invokeLater` メソッドを使い、イベントキューに処理を投げ込んでおき、あとで Swing のスレッドに処理してもらおう、という方法を取る。`SwingUtilities.invokeLater` メソッドには、`Runnable` のオブジェ

クトを渡す。下のプログラムでは 39 ~ 40 行めで
SwingUtilities.invokeLater メソッドを使ってスレッドから GUI のラベ
ルの文字列を変更している。

(Timer の ActionListener では invokeLater を使う必要はないので、この
例では Timer を使うほうがかなり簡単になる。)

また、main メソッドのなかで Swing のメソッドを呼び出すときに
SwingUtilities.invokeLater メソッドを使うのも同じ理由である。

ファイル Denko.java

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 class Denko extends JPanel
5     implements Runnable {
6     private JLabel label;
7     private volatile Thread thread = null;
8     private String msg = "0123456789ABCDEF ";
9
10    public Denko() {
11        label = new JLabel(msg);
12        add(label);
13        JButton startBtn = new JButton("start");
14        startBtn.addActionListener(e -> startThread());
15        JButton stopBtn = new JButton("stop");
16        stopBtn.addActionListener(e -> stopThread());
17        setLayout(new FlowLayout());
18        add(startBtn); add(stopBtn);
19        startThread();
20    }
21
22    private void startThread() {
23        if (thread == null) {
24            thread = new Thread(this);
25            thread.start();
26        }
27    }
28
29    private void stopThread() {
30        thread = null;
31    }
32
33    public void run() {
34        Thread thisThread = Thread.currentThread();
35        for (int i = 0; thread == thisThread;
36            i++, i %= msg.length()) {
37            String str = msg.substring(i)
38                + msg.substring(0, i);
39            SwingUtilities.invokeLater(
40                () -> label.setText(str));
41            try {
42                Thread.sleep(100); // 100 ミリ秒お休み
43            } catch (InterruptedException e) {}
44        }
45    }
46 }
47
48 void main() {
49     SwingUtilities.invokeLater(() -> {
```

```

50     JFrame frame = new JFrame("電光掲示板");
51     frame.add(new Denko());
52     frame.pack();
53     frame.setLocationRelativeTo(null);
54     frame.setVisible(true);
55     frame.setDefaultCloseOperation(
56         JFrame.EXIT_ON_CLOSE);
57     });
58 }

```

なお repaint メソッドは、SwingUtilities.invokeLater と同じようにイベントキューを利用するため、repaint() の呼出しは、SwingUtilities.invokeLater の中に入れる必要はない。例えば GuruguruTh クラスも repaint の呼出しの周りで SwingUtilities.invokeLater メソッドを使っていない。

6.4 スレッドを利用したプログラム

例題 6.4.1 ソーティングの視覚化

ファイル BubbleSort1.java (その 1)

```

1  import java.awt.*;
2  import javax.swing.*;
3
4  class BubbleSort1 extends JPanel
5      implements Runnable {
6      private int[] data = new int[12];
7      private final Color[] cs = {
8          Color.RED, Color.ORANGE, Color.GREEN, Color.BLUE};
9      private volatile Thread thread = null;
10     private int i, j;
11
12     public BubbleSort1() {
13         setPreferredSize(new Dimension(320, 250));
14         startThread();
15     }
16
17     private void startThread() {
18         if (thread == null) {
19             thread = new Thread(this);
20             thread.start();
21         }
22     }

```

これはバブルソート (bubble sort) と呼ばれるアルゴリズムを視覚化したものである。二重ループの中で repaint を呼び出しているため、Timer を使って書き直すのは少し難しくなる。

ファイル BubbleSort1.java (その 2)

```

24     @Override
25     public void paintComponent(Graphics g) {
26         int k;
27         super.paintComponent(g);
28         g.setColor(Color.YELLOW);
29         g.fillOval(5, 50 + j * 10, 10, 10);
30         g.setColor(Color.CYAN);

```

```

31     g.fillOval(5, 50 + i * 10, 10, 10);
32     for (k = 0; k < data.length; k++) {
33         g.setColor(cs[k % cs.length]);
34         g.fillRect(20, 50 + k * 10, data[k] * 5, 10);
35     }
36 }

```

paintComponent も棒グラフ（第3章の Graph.java）のときとほとんど同じである。

配列を乱数で初期化するメソッドを用意しておく。

ファイル BubbleSort1.java（その3）

```

38     private void prepareRandomData() {
39         int len = data.length;
40         for (int k = 0; k < len; k++) {
41             // 適当な範囲の乱数
42             data[k] = (int)(Math.random() * len * 4);
43         }
44     }

```

run メソッドの中は単なるバブルソートアルゴリズムだが、データのスワップをした後、再描画して少し止まるようになっている。

ファイル BubbleSort1.java（その4）

```

46     public void run() {
47         while (true) {
48             prepareRandomData();
49             // バブルソートアルゴリズム
50             for (i = 0; i < data.length - 1; i++) {
51                 for (j = data.length - 1; j > i; j--) {
52                     if (data[j - 1] > data[j]) { // スワップする。
53                         int tmp = data[j - 1];
54                         data[j - 1] = data[j];
55                         data[j] = tmp;
56                         repaint();
57                         try { // repaintの後でしばらく止まる
58                             Thread.sleep(500);
59                         } catch (InterruptedException e) {}
60                     }
61                 }
62             }
63             try { // 並び換え完了後に長めに止まる
64                 Thread.sleep(5000);
65             } catch (InterruptedException e) {}
66         }
67     }
68 }
69
70 void main() { /* 省略 */ }

```

問 6.4.2 クイックソート (quick sort) アルゴリズムを例題 6.4.1 の BubbleSort1.java にならって、データのスワップをしたあと、必ず再描画して少し止まるようにアニメーション化せよ。

参考: クイックソート

```

1     private int[] data = ...

```

```

2
3 private void swap(int i, int j) {
4     int tmp = data[i];
5     data[i] = data[j];
6     data[j] = tmp;
7 }
8
9 private void qsort(int left, int right) {
10    if (left >= right) return;
11    int i = left, j = right;
12    int pivot = data[i + (j - i) / 2];
13    while (true) {
14        while (data[i] < pivot) i++;
15        while (pivot < data[j]) j--;
16        if (i >= j) break;
17        swap(i, j);
18        i++; j--;
19    }
20    qsort(left, i - 1);
21    qsort(j + 1, right);
22 }

```

例題 6.4.3 ソーティングの視覚化（その2）

BubbleSort1.java では、スレッドはいわば自分で目覚しを仕掛けて起きていたが、他人（他のスレッド）に起こしてもらうことを期待して寝ることもできる。次のプログラムではボタンを押したときにスレッドが再開されるようになっている。

ファイル BubbleSort2.java（その1）

```

1 import javax.swing.*;
2
3 import java.awt.*;
4 import java.awt.event.*;
5
6 class BubbleSort2 extends JPanel
7     implements Runnable, ActionListener
8 {
9     /* フィールドは、ほとんど BubbleSort1 と同じ */
10
11     /* 追加のフィールド */
12     JButton step;
13     private volatile boolean threadSuspended = true;
14
15     public BubbleSort2() {
16         setPreferredSize(new Dimension(320, 250));
17         step = new JButton("Step");
18         step.addActionListener(this);
19         setLayout(new FlowLayout());
20         add(step);
21         startThread();
22     }
23     /* startTread は BubbleSort1 と同じ */
24 }

```

このクラスは Runnable と ActionListener の2つのインタフェースを実装しているので、implements のあとに2つのインタフェース名を「,」（コンマ）で区切って並べている。

目覚しを仕掛けずに寝る（待つ）には `sleep` の代わりに、以下のような、
_____ メソッドを用いた形を使う。

ファイル `BubbleSort2.java` (その 2)

```
34 public void run() {
35     while(true) {
36         prepareRandomData();
37         // バブルソートアルゴリズム
38         for (i = 0; i < args.length - 1; i++) {
39             for (j = args.length - 1; j > i; j--) {
40                 if (args[j - 1] > args[j]) { // スワップする。
41                     int tmp = args[j - 1];
42                     args[j - 1] = args[j];
43                     args[j] = tmp;
44                 }
45                 repaint();
46                 /* repaint の後で止まる */
47                 try {
48                     synchronized (this) {
49                         while (threadSuspended) {
50                             wait();
51                         }
52                         threadSuspended = true;
53                     }
54                 } catch (InterruptedException e) {}
55             }
56         }
57         step.setText("Restart");
58         try {
59             synchronized (this) {
60                 while (threadSuspended) {
61                     wait();
62                 }
63                 threadSuspended = true;
64             }
65         } catch (InterruptedException e) {}
66         step.setText("Step");
67     }
68 }
```

また `synchronized` というキーワードにも注意して欲しい。 `wait` メソッドと次に説明する `notify` メソッドを使う場合、 `synchronized (this) { ... }` という形で周りを囲む必要がある。 `synchronized` は排他制御を行なうための構文である。排他制御とは、複数のスレッドが同時に同じリソース（例えば変数やオブジェクト）にアクセスすることを防ぐことである。

また、スレッドを起こすには、_____ というメソッドを使う。

ファイル `BubbleSort2.java` (その 3)

```
70 public synchronized
71 void actionPerformed(ActionEvent e) {
72     threadSuspended = false;
73     notify();
74 }
```

```

75  /* paintComponent, main などは BubbleSort1 と同じ */
98  }

```

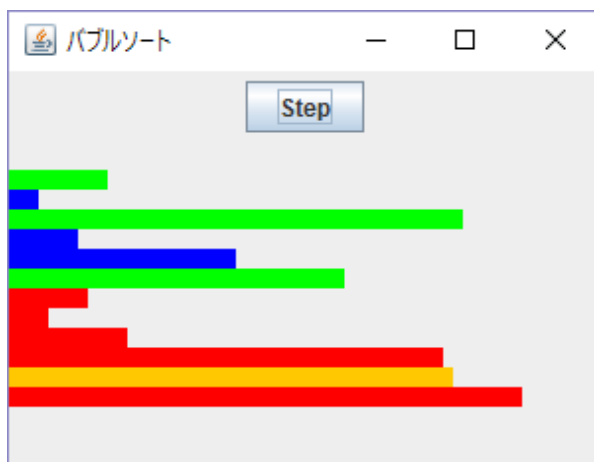
この例のようにメソッドの定義の最初に `synchronized` を修飾子として付け加えると、次のようにメソッドの本体全体を `synchronized (this) { ... }` で囲うのと同じことになる。

```

public void actionPerformed(ActionEvent e) {
    synchronized (this) {
        threadSuspended = false;
        notify();
    }
}

```

この例では、`actionPerformed` の中で `notify` を呼んで、スレッドの実行を再開させている。`threadSuspended` という変数の値を変更しているのは、この `actionPerformed` 以外からも `notify` が（隠れて）呼び出される場合があり、その時にスレッドが間違えて起きないようにするためである。



`BubbleSort2.java` の場合は、少し工夫をすれば、スレッドを使わなくても同じような動作をするプログラムを作ることができる。しかし、クイックソートの場合は、再帰呼出しを使っているので、スレッドを使わずに同じような動作をするプログラムを書くのは難しい。一般的には、このようにアニメーションではないプログラムに対しても、スレッドは有効なテクニックである。

問 6.4.4 クイックソートも例題 6.4.3 の `BubbleSort2.java` と同じようにボタンを押すと 1 ステップ動く（一回のスワップが起こる）ように改造せよ。

挑戦: ただ単に比較ごとに止まるアニメーションにしてもクイックソートはわかりにくいので、過程がわかるように表示を工夫してみよ。

問 6.4.5 （発展）同じデータに対するクイックソートとバブルソートを、比較できるように横に並べて表示するようにせよ。

問 6.4.6 （発展）`BubbleSort2.java` と同じように、ボタンをクリックすると 1 ステップ動くプログラムを、スレッドを使わずに作成せよ。

問 6.4.7 (挑戦) `qsort` メソッドを再帰を使わない形に書き換えて、クイックソートでボタンをクリックすると 1 ステップ動くプログラムを、スレッドを使わずに作成せよ。

6.5 synchronized 文

`synchronized` 文 は次のような形で用いる。

synchronized (式) ブロック

“式”はオブジェクトである（つまり整数などのプリミティブ型ではない）必要がある。`synchronized` 文はこのオブジェクトを“鍵”として、ブロックを排他実行する。つまり、同じ鍵を持つ `synchronized` 文は複数存在している可能性もあって、`synchronized` ブロックを実行している間、他のスレッドに同じ鍵を用いている `synchronized` 文のブロックの実行を待たせる。ブロックの中で `wait` メソッドなどを呼んだ場合は、鍵は一旦返却され、他のスレッドが同じ鍵を用いている `synchronized` 文のブロックを実行することができる。

`synchronized` 文は、途中で中断されると変なことが起こりうる一連の文を実行するときに必要になる。例えば、いくつかのスレッドで共通の変数 `x` を増分するために次のような単純な文:

```
x = x + 1;
```

を実行するような場合も、`synchronized` が必要になる。

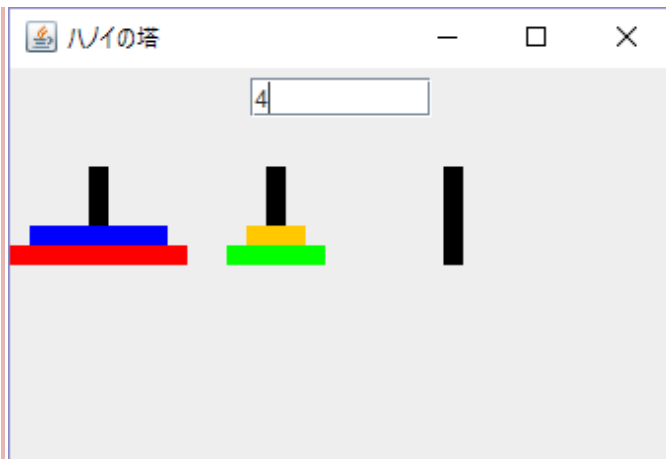
`synchronized`がない場合に起こりうること:

1. 最初 `x = 0` とする。
2. スレッド A が `x + 1` の値 (1) を計算する。
3. ここでスレッドが切り替わる。
4. スレッド B が `x + 1` の値 (1) を計算する。
5. スレッド B が `x` に 1 を代入する。
6. ここでスレッドが切り替わる。
7. スレッド A が `x` に 1 を代入する。

つまり、`x = x + 1;` という文が 2 回実行されたにも関わらず、`x` の値は 1 しか増えていない。これは、`x + 1` の値の計算と `x` への代入の間にスレッドの切り替わりが起こったためである。`synchronized` を使うとこのような事態を避けることができる。いくつかのスレッドで共通の変数をアクセスするときは、大抵このような `synchronized` 文が必要になる。特に、`wait` と `notify` の呼出しにはそのまわりに `synchronized` が必要である。

6.6 問: ハノイの塔

問 6.6.1 ハノイの塔のアルゴリズムをアニメーション化せよ。



(ヒント) ハノイの塔のルール:

3つの棒と直径が1, 2, ..., nのn枚の真中に穴のあいた円盤を用いる。まず、すべての円盤が、小さいものを上に大ききの順に1つの棒にささっている。すべての円盤を別の一つの棒に移動できたら終了である。ただし、

1. 一度に1枚の円盤だけを動かすことができる、
2. 小さな円盤の上に大きな円盤をのせてはいけない、

という制限がある。

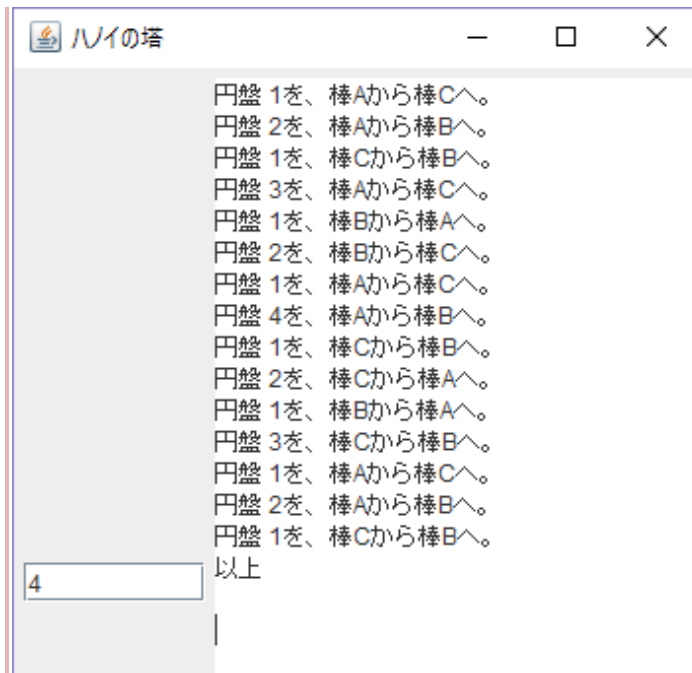
ハノイの塔は再帰法を使って解くことができる。つまり、n-1枚の場合の解き方がわかっているとして、n枚を棒Aから棒Bへ移動する場合:

1. n-1枚の円盤を棒Aから棒Cへ移動する。このやり方はわかっている。
2. 一番下のもっとも大きな1枚を棒Aから棒Bへ移動する。
3. n-1枚の円盤を再び棒Cから棒Bへ移動する。

というように考える。

すると単に output (TextArea のインスタンス) に手順を出力するメソッドの場合は以下になる。

```
1 void hanoi(int n, String a, String b, String c) {
2     if (n > 0) {
3         hanoi(n - 1, a, c, b);
4         output.append("円盤 " + n + "を、"
5                     + a + "から" + b + "へ。\\n");
6         hanoi(n - 1, c, b, a);
7     }
8 }
```



人間がこのような手順を間違えずに実行することは難しいが、コンピュータはまず間違えずに実行してくれる。

キーワード

スレッド、マルチスレッド、Runnable インタフェース、volatile、null、Thread.sleep メソッド、wait メソッド、notify メソッド、synchronized 文