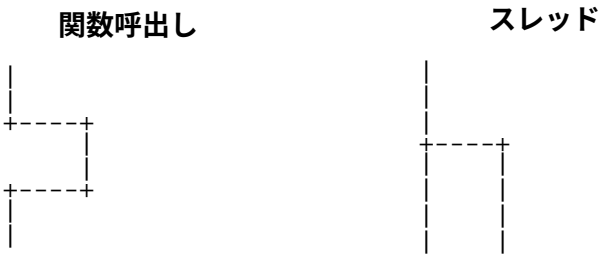
第6章　スレッド

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、これらの処理を行な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 _____（空欄 6.0.1）（thread, もともとの意味は “糸”）という [1]。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（_____（空欄 6.0.2）, mutlithread）。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。



関数呼出とスレッドの比較

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。

[1] 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ーを含む CPU のレジスター、およびスタックなどの情報）のことである。

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で、繰り返しの途中で Thread.sleep というメソッドをを呼んで 10 ミリ秒寝る（実行を止める）。

ファイル ThreadTest.java

```
class MyRunnable implements Runnable {
  String name;
  MyRunnable(String n) {
    name = n;
  }
  public void run() {
    int i;
    for (i = 0; i < 10; i++) {
      try {
        Thread.sleep(10);  // 10 ミリ秒お休み
      } catch (InterruptedException e) {}
      System.out.printf("%s:␣%d,␣", name, i);
    }
  }
}

public class ThreadTest {
  public static void main(String[] args) {
    Thread ta = new Thread(new MyRunnable("A"));
    Thread tb = new Thread(new MyRunnable("B"));
    Thread tc = new Thread(new MyRunnable("C"));
    ta.start(); tb.start(); tc.start(); // スレッド実行開始
  }
}
```

ThreadTest クラスに main 関数があり、ここでスレッドを生成してる。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。（もちろんスレッドが切り替わるタイミングによって、この例と異なる出力になることもある。）

```
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,
```

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。
ファイル Guruguru.java

```
import java.awt.*;
import javax.swing.*;

public class Guruguru extends JPanel implements Runnable {
  private int r = 50, x = 110, y = 70;
  private double theta = 0; // 角度
  private volatile Thread thread = null;

  public Guruguru() {
    setPreferredSize(new Dimension(200, 180));
    JButton startBtn = new JButton("start");
    startBtn.addActionListener(e -> startThread());
    JButton stopBtn = new JButton("stop");
    stopBtn.addActionListener(e -> stopThread());
    setLayout(new FlowLayout());
    add(startBtn); add(stopBtn);
    startThread();
  }

  private void startThread() {
    if (thread == null) {
      thread = new Thread(this);
      thread.start();
    }
  }

  private void stopThread() {
    thread = null;
  }

  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    // スーパークラスの paintComponent を呼び出す
    // 全体を背景色で塗りつぶす。
    g.drawString("Hello,␣World!", x, y);
  }
```

1行めの implements Runnable に注意する。これで Guruguru 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 に対応する）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 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 Guruguru 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。

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。

```
  public void run() {
    Thread thisThread = Thread.currentThread();
    for (; thread == thisThread; theta += 0.02) {
      x = 60 + (int)(r * Math.cos(theta));
      y = 100 - (int)(r * Math.sin(theta));
      repaint();  // paintComponent を間接的に呼出す
      try {
        Thread.sleep(30); // 30 ミリ秒お休み
      } catch (InterruptedException e) {}
    }
  }

   ...  // main メソッドの定義は割愛
}
```

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。

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 TextAnimation.java は、文字列が左から右に移動し、右端まで移動すれば、再び左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

```
import javax.swing.*;
import java.awt.*;

public class TextAnimation extends JPanel
                              ____________________________ {
  private int x = 0;
  private volatile Thread thread = null;

  public TextAnimation() {
    setPreferredSize(new Dimension(200, 150));
    JButton startBtn = new JButton("start");
    startBtn.addActionListener(e -> startThread());
    JButton stopBtn = new JButton("stop");
    stopBtn.addActionListener(e -> stopThread());
    setLayout(new FlowLayout());
    add(startBtn); add(stopBtn);
    startThread();
  }

  // startThread, stopThread は Guruguru と同じ

  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawString("HELLO␣WORLD!", x, 100);
  }

  public void run() {
    Thread myThread = Thread.currentThread();
    while (_____________________________) {
      x += 5;
      if (x > 200) {
        x = 0;
      }

      _____________
      try {
        Thread.sleep(100); // 100 ミリ秒お休み
```

```
      } catch (InterruptedException e) {}
    }
  }

   ...  // main メソッドの定義は割愛
}
```

問 6.2.3 run メソッドをラムダ式として定義するように、Guruguru.java を書き換えよ。
ファイル Guruguru.java

```
import java.awt.*;
import javax.swing.*;

public class Guruguru ____________________________________________ {
  private int r = 50, x = 110, y = 70;
  private double theta = 0; // 角度
  private volatile Thread thread = null;

   ...  // コンストラクターは元のバージョンと同じ

  private void startThread() {
    if (thread == null) {
      thread = new Thread(_______{
        /* 元のバージョンの run メソッドの中身と同じ */
      });
      thread.start();
    }
  }

   ...  // stopThread, paintComponent, main は元のバージョンと同じ
}
```

問 6.2.4 （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時、跳ね返るような GUI アプリケーションを書け。

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。

ファイル Denko.java

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

Runnable はメソッドを一つしか持たないインタフェースなので、27〜31 行目は以下のようにラムダ式を使うこともできる。

```
      SwingUtilities.invokeLater(() -> label.setText(str));
```

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、Guruguru クラスも 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

```
import javax.swing.*;
import java.awt.*;

public class BubbleSort1 extends JPanel implements Runnable {
  private int[] args = new int[12]; // 適当なサイズ
  private final Color[] cs = { Color.RED, Color.ORANGE,
                               Color.GREEN, Color.BLUE };
  private volatile Thread thread = null;
  private int i, j;

  public BubbleSort1() {
    setPreferredSize(new Dimension(300, 200));
    startThread();
  }

  private void startThread() {
    if (thread == null) {
      thread = new Thread(this);
      thread.start();
    }
  }

   ...
```

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。

```
  @Override
  public void paintComponent(Graphics g) {
    int k;
    super.paintComponent(g);
    g.setColor(Color.YELLOW);
    g.fillOval(5, 50 + j * 10, 10, 10);
    g.setColor(Color.CYAN);
    g.fillOval(5, 50 + i * 10, 10, 10);
    for(k = 0; k < args.length; k++) {
      g.setColor(cs[k % cs.length]);
      g.fillRect(20, 50 + k * 10, args[k] * 5, 10);
    }
  }
```

paintComponent も棒グラフ（第 3 章の Graph.java）の時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。

```
        private void prepareRandomData() {
                int len = args.length;
                for (int k = 0; k < len; k++) {
                        args[k] = (int)(Math.random() * len * 4);
// 適当な範囲の乱数
                }
        }
```

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。

```
  public void run() {
    while (true) {
      prepareRandomData();
      // バブルソートアルゴリズム
      for (i = 0; i < args.length - 1; i++) {
        for (j = args.length - 1; j > i; j--) {
          if (args[j - 1] > args[j]) { // スワップする。
            int tmp = args[j - 1];
            args[j - 1] = args[j];
            args[j] = tmp;
            repaint();
            try { // repaint の後でしばらく止まる
              Thread.sleep(500);
            } catch (InterruptedException e) {}
          }
        }
      }
    }
  }
   ...  // main メソッドの定義は割愛
}
```

問 6.4.2 クイックソート （quick sort）アルゴリズムをバブルソートにならって、データのスワップをしたあと、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

```
  static void swap(int[] v, int i, int j) {
    int tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;
  }

  static void qsort(int[] v, int left, int right) {
    if (left >= right) return;
    int i = left, j = right;
    int pivot = v[i + (j - i) / 2];
```

```
    while (true) {
      while (v[i] < pivot) i++;
      while (pivot < v[j]) j--;
      if (i >= j) break;
      swap(v, i, j);
      i++; j--;
    }
    qsort(v, left, i - 1);
    qsort(v, j + 1, right);
  }
```

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時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

```
public class BubbleSort2 extends JPanel
                         implements Runnable, ActionListener {
   ...
  private volatile boolean threadSuspended = true;

  public BubbleSort2() {
    setPreferredSize(new Dimension(300, 200));
    JButton step = new JButton("Step");
    step.addActionListener(this);
    setLayout(new FlowLayout());
    add(step);
    startThread();
  }

  // startThread メソッドは BubbleSort1 と同じ
   ...
}
```

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には sleep の代わりに、以下のような、＿＿＿＿ (空欄 6.4.1) メソッドを用いた形を使う。

```
  public void run() {
    while(true) {
      prepareRandomData();
      // バブルソートアルゴリズム
      for (i = 0; i < args.length - 1; i++) {
```

```
        for (j = args.length - 1; j > i; j--) {
          if (args[j - 1] > args[j]) { // スワップする。
            int tmp = args[j - 1];
            args[j - 1] = args[j];
            args[j] = tmp;
          }
          repaint();
          /* repaintの後で止まる */
          try {
            synchronized (this) {
              while (threadSuspended) {
                wait();
              }
              threadSuspended = true;
            }
          } catch (InterruptedException e) {}
        }
      }
    }
  }
```

また synchronized というキーワードにも注意して欲しい。synchronized は排他制御（後述）を行なうための構文である。

また、スレッドを起こすには、＿＿＿＿＿(空欄 6.4.2) というメソッドを使う。

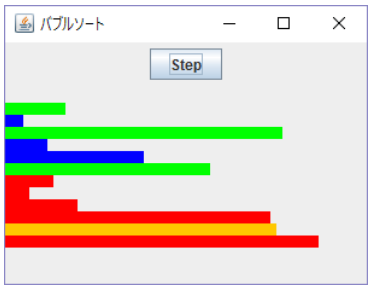
```
  public synchronized void actionPerformed(ActionEvent e) {
     // ボタンが押された時の処理
    threadSuspended = false;
    notify();
  }
```

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();
    }
  }

   ...  // main メソッドの定義は割愛
}
```

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っ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、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（次の問参照）。

問 6.4.4 クイックソートも同じようにボタンを押すと 1 ステップ動く（一回の比較が起こる）ように改造せよ。

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、この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時に必要になる。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。

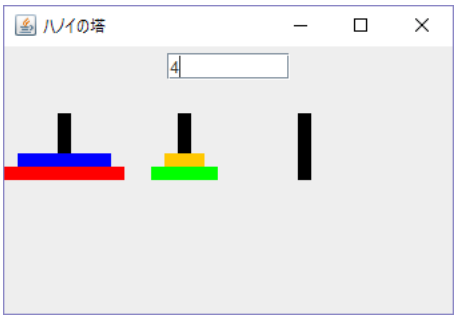
synchronized がない場合に起こりうること:

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時は、大抵このような `synchronized` 文が必要になる。

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さの順に 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。

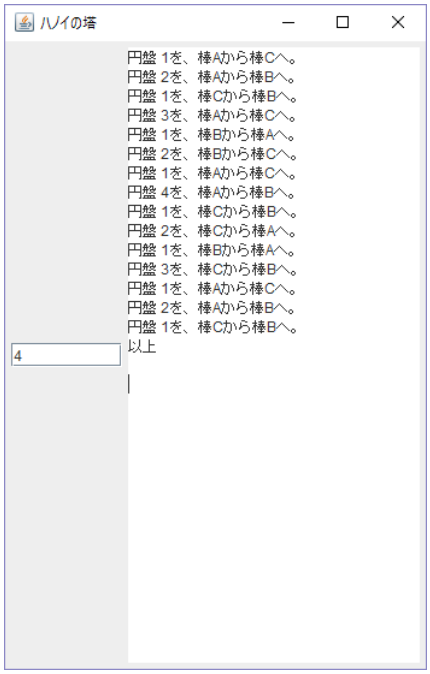
ハノイの塔は再帰法を使って解くことができる。つまり、$n-1$ 枚の場合の解き方がわかっているとして、n 枚を棒 A から棒 B へ移動する場合:

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のようになる。

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



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、コンピューター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