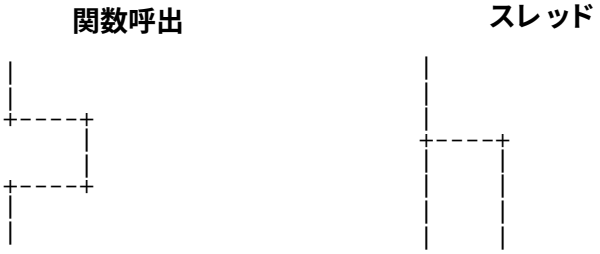
第6章　スレッド

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 アプレットの場合、スレッドを用いるとアプレットに動きを与えることができる。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。

アプレットの `init`, `start`, `paint` 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）はブラウザから呼び出される。これらのメソッドが実行されている間は、ブラウザ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レットは、`start` や `paint` 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、ブラウザの他の処理と同時に、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の処理を行なう単位をスレッド（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アプレットはスレッドを複数必要とする（マルチスレッド（空欄 6.0.2）, mutli-thread）。CPU が 1 つしかない普通のコンピュータでは、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。

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

[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。

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アプレット

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アプレットである。
ファイル Guruguru.java

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

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

  @Override
  public void paint(Graphics g) {
    super.paint(g);
    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。paint 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。

スレッドはアプレットの start メソッド内で生成される。Thread のコンストラクタの引数は、この場合は this — つまりアプレット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。

アプレットの stop メソッドでは、スレッドの実行を止めている。このように、通常アプレットの start と stop メソッドにスレッドの実行開始と停止処理を書いておく。すると、アプレットのあるページから他のページに移動した時にスレッドの実行が停止され、戻ってきた時にスレッドが実行再開される。

```
  @Override
  public void start() {
    if (thread == null) {    // 念のためチェック
      thread = new Thread(this);
      thread.start();
    }
  }

  @Override
```

```
  public void stop() {
    thread = null;
  }
```

これはスレッドを使うアプレットの start と stop メソッドの典型的な定義である。

```
  public void run() {
    Thread thisThread = Thread.currentThread();
    for (; thread == thisThread; theta+=0.02) {
      x = 60+(int)(r*Math.cos(theta));
      y = 70-(int)(r*Math.sin(theta));
      repaint();

      try {
        Thread.sleep(30);  // 30 ミリ秒お休み
      } catch (InterruptedException e) {}
    }
  }
```

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、stop メソッドによって、thread の値が変更されると、このループは終了し、スレッド自体も終了する。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 ミリ秒寝る。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）ので周りを try〜catch で囲んでいる。

アプレットでスレッドを使うときには、通常

- run メソッドを定義する。（通常は永久ループにする。）
- 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。
- start メソッドと stop メソッドは上の例の通りにする。

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 JApplet
```

```
                          implements Runnable {
  int x = 0;
  Thread thread = null;

  @Override
  public void start() {
    if (thread == null) {    // 念のためチェック
      thread = new Thread(this);
      thread.start();
    }
  }

  @Override
  public void stop() {
    thread = null;
  }

  @Override
  public void paint(Graphics g) {
    super.paint(g);
    g.drawString("HELLO␣WORLD!", x, 55);
  }

  public void run() {
    Thread thisThread = Thread.currentThread();
    while (thread == thisThread) {
      x += 5;
      if (x>200) {
        x = 0;
      }
      repaint();
      try {
        Thread.sleep(30); // 30 ミリ秒お休み
      } catch (InterruptedException e) {}
    }
  }
}
```

問 6.2.3 run メソッドを匿名クラスに定義するように、Guruguru.javaを書き換えよ。

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、跳ね返るようなアプレットを書け。

問 6.2.5 （発展）（ちらつき防止）
Guruguru.javaでは文字が少しちらついて見える。これは、描画のたびに一旦画面を背景色で塗りつぶしているためである。ちらつきを防止するための手法である“ダブルバッファリング”について調べ、実装せよ。java.awt.Componentクラス

（JApplet などのスーパークラス）の createVolatileImage メソッドを調べて使用せよ。

6.3　スレッドを利用したプログラム

例題 6.3.1 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 JApplet implements Runnable {
  int[] args = { 10, 3, 46, 7, 23, 34, 8, 12, 4, 45, 44, 52};
  Color[] cs = { Color.RED, Color.ORANGE, Color.GREEN, Color.BLUE};
  Thread thread = null;

   ...
```

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。start, stop は Guruguru.java と全く同じなので省略してある。

```
  @Override
  public void paint(Graphics g) {
    int i;

    super.paint(g);
    for(i=0; i<args.length; i++) {
      g.setColor(cs[args[i]%cs.length]);
      g.fillRect(0, i*10, args[i]*5, 10);
    }
  }
```

paint も棒グラフ（第 3 章の Graph.java）の時とほとんど同じである。

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() {
    int i, j;
    Thread thisThread = Thread.currentThread();

    for (i=0; i<args.length-1; i++) {
      for (j=args.length-1; thread == thisThread && 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の後でしばらく止まる */
```

```
        try {
          Thread.sleep(500);
        } catch (InterruptedException e) {}
      }
    }
  }
}
```

問 6.3.2 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視覚化せよ。
参考: クイックソート

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

  void qsort(int[] v, int left, int right) {
    int i, last;

    if (left >= right) return;
    swap(v, left, (left+right)/2);
    last = left;
    for (i=left+1; i <= right; i++) {
      if (v[i] < v[left]) {
        swap(v, last+1, i);
        last++;
      }
    }
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
  }
```

例題 6.3.3 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 JApplet
                         implements Runnable, ActionListener {
  ...
  private boolean threadSuspended = true;

  @Override
```

```
  public void init() {
    JButton step = new JButton("Step");
    step.addActionListener(this);
    setLayout(new FlowLayout());
    add(step);
  }
   ...
}
```

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 , 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 wait(空欄 6.3.1) メソッドを用いた形を使う。

```
  public void run() {
    int i, j;

    for (i=0; i<args.length-1; i++) {
      for (j=args.length-1; j>i; j--) {
         ...
        repaint();
        /* repaintの後で止まる */
        try {
          synchronized(this) {
            while (threadSuspended) {
              wait();
            }
            threadSuspended=true;
          }
        } catch (InterruptedException e) {}
      }
    }
    thread=null;
  }
```

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、notify(空欄 6.3.2) 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();
  }
}
```

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 6.3.4 クイックソートも同じようにボタンを押すと 1 ステップ動くように改造せよ。

6.4 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。

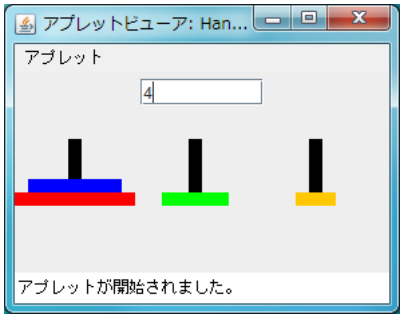
synchronized がない場合に起こりうること:

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。

6.5　問: ハノイの塔

問 6.5.1 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。

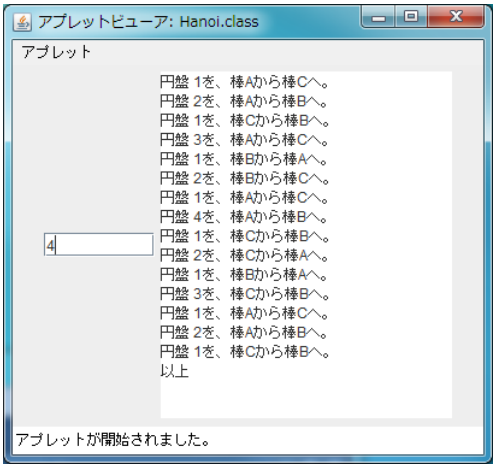
ハノイの塔は再帰法を使って解くことができる。つまり、$n-1$ 枚の場合の解き方がわかっているとして、n 枚を棒 A から棒 B へ移動する場合:

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==1) {
    output.append("円盤␣1を、"+a+"から"+b+"へ。\n");
  } else {
    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, null, Thread.sleep メソッド, wait メソッド, notify メソッド, synchronized