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。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（buttonClicked, 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 _____（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（_____, 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。

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するための方法を学ぶ。

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 とすることが決まっている。run は、Runnable インター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 インター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繰り返しで、ループの途中で Thread.sleep というメソッドをを呼んで 30 ミリ秒実行を止めている（寝る）。

try〜catch〜は _____と呼ばれる形である（後述）。Thread.sleep メソッドを呼び出す時には、この try〜catch〜が必要である。

[1] 具体的にいえば、プログラムカウンタを含む CPU のレジスタ、およびスタックの情報などのことである。

ファイル ThreadTest.java

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

public class ThreadTest {
  public static void main(String[] args) {
    Thread t1 = new Thread(new MyRunnable("A"));
    Thread t2 = new Thread(new MyRunnable("B"));
    Thread t3 = new Thread(new MyRunnable("C"));
    t1.start(); // スレッド実行開始
    t2.start();
    t3.start();
  }
}
```

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。このスレッドの start メソッドを呼び出して、スレッドの実行をスタートする。

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,
```

例題 6.0.1 ぐるぐる廻る

アプレットでスレッドを利用した例である。単に文字列が円の上を動くだけの簡単なアプレットである。

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;

  ...

  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。

また、このクラスが *thread* という *Thread* 型のインスタンス変数を持っていることに注意する。*thread* の初期値は *null* である。*null* は、＿＿＿＿＿＿＿＿＿＿（*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* ― ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿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。

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。

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

  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) {}
  }
}
```

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。

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 型のメンバ thread（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。

問 6.0.2 円が左右に移動し、左か右の壁にぶつかった時は跳ね返るようなアプレットを書け。さらに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 6.0.3 （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。ちらつきを防止するための手法である “ダブルバッファ” について調べ、実装せよ。

try〜catch〜文は

try ブロック$_1$ **catch** (例外型 変数) ブロック$_2$

という形で用いる。

ブロック$_1$ の中で**例外**（エラーと考えて良い）と呼ばれる状況が起こった時、catch の後ろの例外型がその例外の型と一致するか調べ、一致すればその後のブロックを実行する。一致するものがなければ、さらに外側の catch を探す。それでもなければプログラムを終了する。

（“**catch** (例外型 変数) ブロック” という形が複数続いても良い。その場合は最初にマッチするブロックが実行される。また、最後に “**finally** ブロック” という形がつく場合もある。その場合、ブロックは必ず実行される。）

例えば、0 による除算を行なうと ArithmeticException という種類の例外が発生する。次のようなプログラムを実行すると、

ファイル TryCatchTest.java

```
public class TryCatchTest {
  public static void main(String[] args) {
    int i;
    try {
      for (i=-5; i<5; i++) {
        System.out.println(10/i);
      }
    } catch (ArithmeticException e) {
      System.out.println("エラー: "+e.toString());
    }
  }
}
```

出力は次のようになる。

```
-2
-2
-3
-5
-10
エラー: java.lang.ArithmeticException: / by zero
```

i が 0 になった地点で例外が発生し、catch の後のブロックが実行される。
なお、例外を発生させるには throw 文を用いる。

throw 式;

この “式” は例外型（Exception あるいはそのサブクラス）のオブジェクトでなければならない。
次の例は、コマンドライン引数として渡された数字の積を計算するプログラムである。途中で 0 が出てきた場合は、わざと例外を発生させて、残りのかけ算の処理を行なわないようにしている。

ファイル TryCatchTest2.java

```
public class TryCatchTest2 {
  public static void main(String[] args) {
    int i, m=1;
    try {
      for (i=0; i<args.length; i++) {
        int a = Integer.parseInt(args[i]);
        if (a==0) throw new Exception("zero");
        m *= a;
      }
    }  catch (Exception e) {
      m = 0;
    }
    System.out.println("答は " + m + "です。");
  }
}
```

例えば “java TryCatchTest2 1 2 0 3 4 5 6” というコマンドライン引数で実行させると、3 番目の引数の 0 を呼んだ時点で、例外を発生させるため、残りの引数の 3, 4, 5, 6 は無視される。

例題 6.0.4 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。

```
  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 6.0.5 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。

例題 6.0.6 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 {
  ...

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

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 _____メソッドを用いた形を使う。

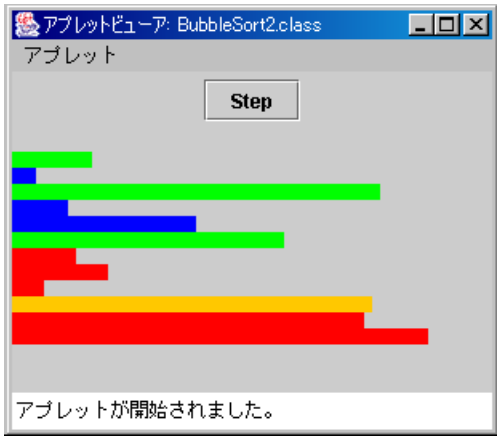
```
  public void run() {
    int i, j;

    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。*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（後述）を行なうための構文である。また、下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、メソッドの本体全体を *synchronized(this)* { ... }で囲うのと同じことになる。）

また、スレッドを起こすには、_____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）。

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。

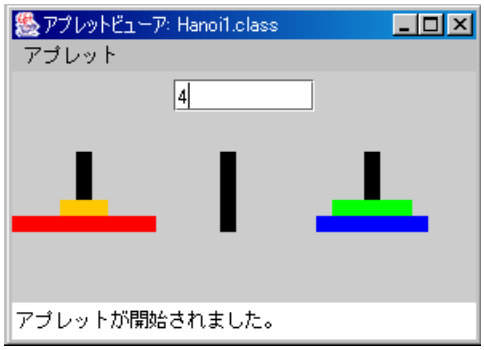
synchronized がない場合に起こりうること:

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。

問 6.0.7 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。

問 6.0.8 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、

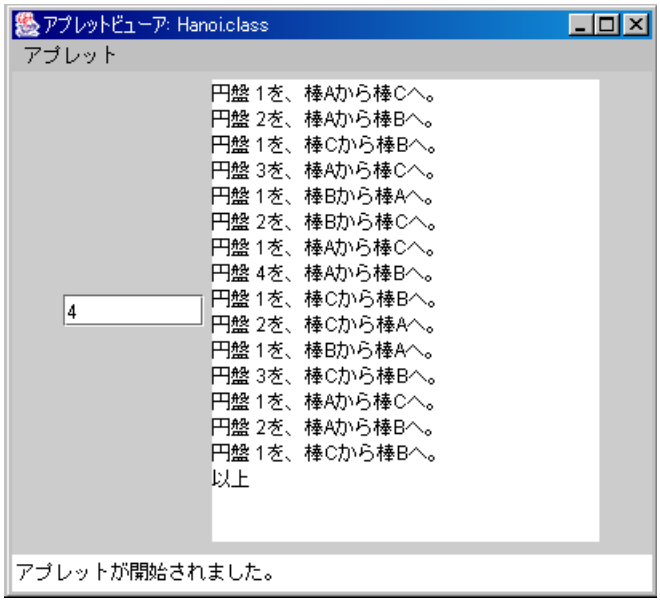
1. 一度に *1* 枚の円盤だけを動かすことができる、
2. 小さな円盤の上に大きな円盤をのせてはいけない、

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 インターフェース, null, Thread.sleep メソッド, 例外処理, wait メソッド, notify メソッド, synchronized