

主題科目:情報の仕組み
Java プログラミング

佐賀大学学術情報処理センター
只木進一

平成 17 年 1 月 13 日

第1章 この講義の目的

HTML (Hyper Text Markup Language) 文書の相互リンク (link) で構成される WWW(World Wide Web) は、企業や大学の広報から個人の私的日記にまで利用される情報公開、情報収集の基本的な技術になっています。HTML 文書の中には、アクセスする際に送られたデータに応じてその都度 (動的、dynamical と呼ぶ)、内容を変化させているものがあります。動的にページを生成する技術にはさまざまなものがありますが、そのような技術の一つが、Java を使ったプログラムです。本講義では、Java を使ったプログラムの初歩を学びます。

Java が動的ページの生成に使われている理由にはいくつか重要なものがあります。その第一が、Java が特定のコンピュータ環境に依存したものであることです。動的ページを表示している WWW ブラウザが Windows でも Linux でも、Java を動作させる環境である JRE (Java Runtime Environment) があれば動作します。また、理由の第二は、容易に GUI (Graphical User Interface) や動画が作成できることです。

また、Java はオブジェクト指向 (Object Oriented) プログラミング言語の一つと呼ばれています。オブジェクト指向は、プログラミングの考え方として新しいものの一つです。現実の世界で起こる現象を、対象をデータの集まりとして捕らえ、それらの動作としてプログラムを書こうという考えが基本になっています。Java は他のオブジェクト指向プログラミング言語、例えば C++ と共通の側面を多く有しています。Java を学ぶことで、他のプログラミング言語の習得が容易になるでしょう。

本講義では、Java の開発環境の一つである Sun One Studio を使います。

第2章 簡単なプログラム

2.1 はじめて Sun One Studio を使う場合

最初に、この講義で使うディレクトリを作成しましょう。例えば~/javasrc というディレクトリを使うこととにして、作成します。

```
mkdir javasrc
```

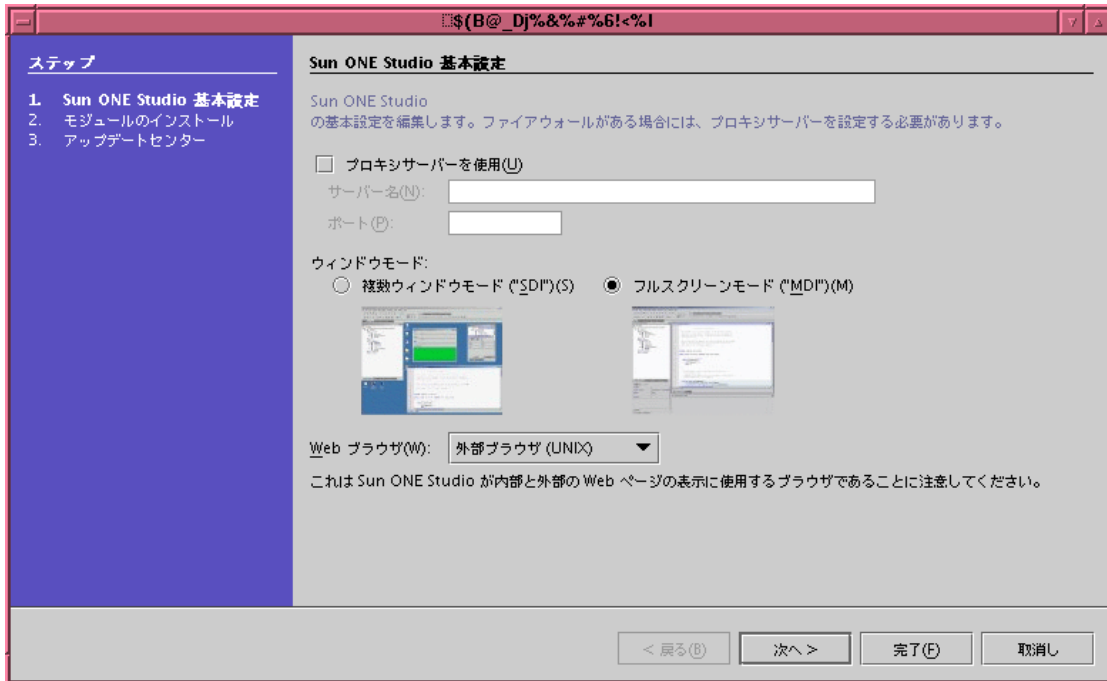
このディレクトリの下に、各プログラムを作っていくことにします。

次に、Sun One Studio を起動します。起動コマンドは

```
runide.sh
```

です。始めて Sun One Studio を使う場合、いくつかの設定が必要です。起動直後に現れるのは「設定をインポートする必要がありますか?」と書かれた画面です。これは、以前のバージョンの Sun One Studio を使って来た人のためのものですから、「いいえ、設定のインポートをスキップします」を選んで、「完了」を押します。

図 2.1.1: 設定画面



次に、Sun One Studio の「見え方」を設定します。図 2.1.1 のように設定します。設定が終わったら一旦終了しましょう。「ファイル」メニューから「終了」を選びます。

上のように作業すると、ホームディレクトリに約 40MB のファイルが生成されます。佐賀大学学術情報処理センター演習室の環境では、基本設定を配布しています。runide.sh を実行する前に、

```
cd
rm -rf ffjuser40ee
tar zxvf /usr/local/s1studio/ffjuser40ee.tar.gz
```

と実行して、環境の初期化を行ってください。

2.2 Hello World

Sun One Studio では、一つのプログラムは一つの「プロジェクト」として管理します。各プロジェクトは、別のディレクトリに置きます。

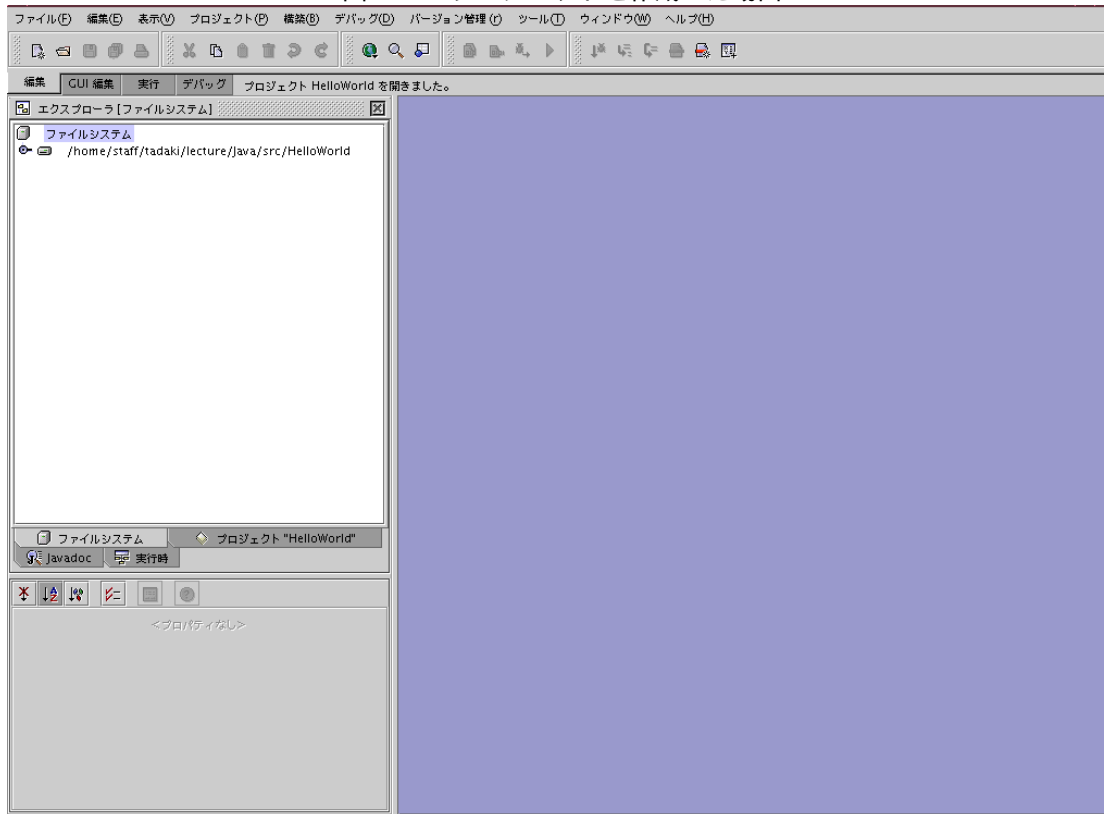
最初のプロジェクト「Hello World」を置くためのディレクトリを上記の~/javasrc の下に作りましょう。

```
cd ~/javasrc
mkdir HelloWorld
```

次に、Sun One Studio を起動します。新しいプロジェクトを作成するために、メニューの「プロジェクト」から「プロジェクトマネージャー」を選びます。プロジェクト一覧で右の「新規」ボタンを押し、新しいプロジェクト名として「Hello World」とします。

次に、メニューの「ファイル」から「ファイルシステムのマウント」を選びます。「テンプレート」として「ローカルディレクトリ」を選び、上記の~/javasrc/HelloWorldを選択します。すると図 2.2.1 のような画面となります。

図 2.2.1: プロジェクトを作成した場面



次の Java のソースファイルを作成します。まず、左側のタブを「GUI 編集」に切替えます。左側の「プロジェクト "HelloWorld"」のところで、マウスの右ボタンをクリックし、「新規を作成」を選びます。この講義ではアプレットを作成しますので、「GUI フォーム」の中の「JApplet」を選び、「次へ」を押します。名前として「HelloWorld」と入力し、「完了」ボタンを押します。

「フォームエディタ」の方は、ボタンなどを配置するのに使いますが、しばらくは使いません。「ソースエディタ」を使います。Program 2.2.1 のようにプログラムを作成しましょう。

Program 2.2.1 HelloWorld.java

```
/*
 * HelloWorld.java
 *
 * Created on 2004/08/07, 14:18
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class HelloWorld extends javax.swing.JApplet {
    Font font = new Font("TimesRoman",Font.PLAIN,40);
    /** Creates new form HelloWorld */
    public HelloWorld() {
        initComponents();
    }

    public void paint(Graphics g){
        g.setFont(font);
        g.setColor(Color.red);
        g.drawString("Hello World",0,50);
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    private void initComponents() { //GEN-BEGIN:initComponents

    } //GEN-END:initComponents

    // Variables declaration - do not modify //GEN-BEGIN:variables
    // End of variables declaration //GEN-END:variables
}
}
```

編集が終わったら、ソースコードをコンパイル (compile) しましょう。コンパイルすることで、人間が読める文字列からコンピュータが実行できる言葉へ翻訳及び必要なライブラリ (library) との結合が行われます。メニューの「プロジェクト」から「プロジェクトのコンパイル」を選択することでコンパイルが実行されます。左側の一覧の「HelloWorld」を右クリックし、「コンパイル」を選ぶことでもコンパイルができます。~/javasrc/HelloWorldディレクトリにファイルHelloWorld.classができていることを確認してください。

左側の一覧の「HelloWorld」を右クリックし、「実行」を選ぶことで、プログラムを実行することができます。

次に、このアプレットHelloWorld.classを呼び出すHTMLファイルを編集しましょう。再び、左側の「プロジェクト”HelloWorld”」のところで、マウスの右ボタンをクリックし、「新規を作成」

を選びます。今度は「その他」の中の「HTML ファイル」を選び、「次へ」を押します。名前として「HelloWorld」として「完了」を押します。

HTML ファイルのソースが表示されます。<BODY>タグのすぐしたに、アプレットを読み出す部分を作成しましょう。

Program 2.2.2 HelloWorld.htm

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

```
<HTML>
  <HEAD>
    <TITLE></TITLE>
  </HEAD>
  <BODY>
    <applet code="HelloWorld.class" width="350" height="100">
    </applet>
  </BODY>
</HTML>
```

2.3 コードの概要

実世界のモノには、属性や状態といったデータと、操作や動作の部分があります。プログラムの中で、属性や状態をもったデータの塊を操作するという考え方でプログラムを作成する方法をオブジェクト指向プログラミング (Object Oriented Programming) と呼びます。また、そうした属性や状態をもったデータの具体的な塊をオブジェクト (object) と呼び、オブジェクトの類形をクラス (class) と呼びます。

つまり、クラスは、属性や状態といったデータと、それらを操作する方法が組になった「変数」型です。その「変数」型の具体化されたものがオブジェクトです。

では、少しだけ、java のプログラムを見てみましょう。このプログラムは、HelloWorld というクラスだけで構成されています。

```
public class HelloWorld extends javax.swing.JApplet
```

キーワード `public` は、他のプログラムから利用することを許すことを表しています。また、最後の部分にある `extends javax.swing.JApplet` は、既に定義されているクラス `javax.swing.JApplet` の拡張として定義することを意味しています。

クラスの操作をメソッド (method) と呼びます。このプログラムでは、`paint` というメソッドが定義されています。

```
public void paint(Graphics g)
```

キーワード `public` は他から呼び出すことができるメソッドであることを示しています。メソッドは、数学の関数と同じように、変数を与えて何か処理を行い、結果を返します。メソッド (関数) 名の前に置かれるキーワードは、結果の型を表しますが、`void` は結果を返さない関数であることを

示しています。なお、メソッド `paint` は、アプレットが呼び出される際に、描画を行う特別なメソッドです。

メソッド `paint` の引数も、クラス `Graphics` に属するオブジェクト `g` です。クラス `Graphics` のメソッド `setFont` と `setColor` を使って、フォントと色が指定され、`drawString` で文字列が表示されています。数値 `0` と `50` は、文字列を描く座標です。座標系が、左上を原点とし、右に `X` 方向正、下に `Y` 方向正をとる座標系であることに注意が必要です。

このクラスと同じ名前をもつメソッド `HelloWorld` は、特別なメソッドでコンストラクタ (constructor) と呼ばれます。これが、クラスの初期化を行う、つまり最も最初に行われるメソッドです。今回のプログラムでは、何も行っていません。

第3章 簡単な計算

3.1 基本的プログラミング

ここでは、Java プログラミングのもっとも基本的な部分について説明していきます。

表 3.1: Java で使える原始型

int	整数型	32bit 符号付き整数 − 2^{31} から $2^{31} - 1$ までの整数
long	整数型	64bit 符号付き整数
short	整数型	16bit 符号付き整数
byte	整数型	8bit 符号付き整数
char	文字型	16bitUnicode 文字 16bit 符号無し整数と同じ
boolean	論理	true または false
float	浮動小数型	32bit 符号付き浮動小数
double	浮動小数型	64bit 符号付き浮動小数

Program3.1.1 は、paint メソッドの中で、簡単な計算を行い、その結果を表示するプログラムです。数値を保存するための三つの変数 a、b 及び c が定義されています。キーワード int は変数の型を表します。

プログラム中で値を保持する変数は、コンピュータのメモリ上に割り当てられます。コンピュータの中ではデータは全て 0 と 1 からなる二進数で表現されています。そこで、変数の型によって、その変数にどのような値を割り当てるかを定義する必要があります。Java で使える基本の型 (原始型) は表 3.1 の 8 種類です。Java では、全ての変数は、型を定義しなくてはなりません。

プログラム中の記号 '=' は、等号ではなく、代入を表します。右辺の計算の結果を左辺に代入します。代入の両辺の型が同じようにするのが原則です。左辺の型のほうが右辺よりも大きい (例えば、整数型の値を浮動小数点型へ代入する) 場合には、型の変換 (キャスト、cast) が自動的に行われます。しかし、逆の場合には注意が必要です。

数値同士の演算は、通常四則演算 (+、−、*、/) が定義されています。整数同士の除算においては、結果が整数に切り捨てられることに注意が必要です。また、整数同士の除算の余りを計算する演算 (%) も定義されています。

Program 3.1.1 SimpleArithmetic.java

```
/*
 * SimpleArithmetic.java
 *
 * Created on 2004/08/01, 23:54
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class SimpleArithmetic extends javax.swing.JApplet {
    private Font f = new Font("TimesRoman",Font.PLAIN,40);

    /** Creates new form SimpleArithmetic */
    public SimpleArithmetic() {
        initComponents();
    }

    public void paint(Graphics g) {
        g.setFont(f);
        int a=8;
        int b=5;
        int c;
        c=a+b;
        String s1=String.valueOf(a)+"+"+String.valueOf(b)+"="+String.valueOf(c);
        g.drawString(s1,0,50);
        c=a-b;
        String s2=String.valueOf(a)+"-"+String.valueOf(b)+"="+String.valueOf(c);
        g.drawString(s2,0,100);
    }

    private void initComponents() {

    }
}
```

3.2 文字と文字列

一文字を保持する変数の型を文字型 (char) と呼びます。Java では、一つの文字を表す文字型変数と、文字の列からなる文字列は、厳密に区別されます。文字定数はシングルクォーテーション'で区切ります。

```
char c='a';
```

文字定数には、通常のアルファベットや数値のほか、いくつかの特殊文字が定義されています。Java では、文字 UTF-8 と呼ばれるコード体系でデータを保持しています。そのため、複数の言語が混じった文字列を作ること可能です。

一方、文字列型 (String) は原始型ではなく、クラスであり、代入にはダブルクォーテーション"で区切ります。文字列を連結するには、演算+を使うことができます。Java の文字列型は、一旦

表 3.2: 特殊文字

\n	改行
\t	タブ
\b	バックスペース
\r	キャリッジリターン
\f	フォームフィード
\\	バックシュラッシュ
\'	シングルクォート
\"	ダブルクォート
\ddd	8進定数

生成されると変更できないという特徴があります。

文字列の連結は '+' で行います。Program3.1.1 では、計算の結果を一つの文字列として先に生成し、文字列を描くメソッド `drawString` で表示しています。数値を文字列に変換するには

`String.valueOf(数値)`

で行います。これは、クラス `String` が持っているメソッド `valueOf` を使うことを意味しています。

3.3 簡略演算子

Java では、C/C++ と同様に、演算を簡潔に記述するために簡略演算子が用意されています。慣れると非常に便利な記法です。

表 3.3: 簡略演算子

表記	意味
<code>x++</code>	<code>x=x+1</code>
<code>x--</code>	<code>x=x-1</code>
<code>x+=y</code>	<code>x=x+y</code>
<code>x-=y</code>	<code>x=x-y</code>
<code>x*=y</code>	<code>x=x*y</code>
<code>x/=y</code>	<code>x=x/y</code>
<code>x%=y</code>	<code>x=x%y</code>

第4章 配列と繰り返し

4.1 配列

多くのプログラミング言語には、表のように同じ型のデータを大量に保持する配列 (array) というデータ構造があります。また表の各列に同じ計算を繰り返すことに対応するような、繰り返し (loop) を行うプログラム制御構造を持っています。ここでは、データの平均と分散を計算する簡単なプログラムを例に、配列と繰り返しの使い方を学びます。

N 個のデータ $\{x_i\}$ を考えます。その平均 $\langle x \rangle$ と分散 (二乗誤差) $\langle \Delta x^2 \rangle$ は、以下のように求められます。

$$\langle x \rangle = \frac{1}{N} \sum_{i=0}^{N-1} x_i \quad (4.1.1)$$

$$\langle \Delta x^2 \rangle = \frac{1}{N} \sum_{i=0}^{N-1} (x_i - \langle x \rangle)^2 = \frac{1}{N} \sum_{i=0}^{N-1} x_i^2 - \langle x \rangle^2 \quad (4.1.2)$$

ここで和の計算を次のように見直してみます。始めの n 個の和を

$$S_{n-1} = \sum_{i=0}^{n-1} x_i \quad (4.1.3)$$

と表すことにします。これを使うと

$$S_n = S_{n-1} + x_n \quad (4.1.4)$$

$$S_0 = x_0 \quad (4.1.5)$$

と表すことができます。つまり、直前までの和 S_{n-1} に新しい項 x_n を加えるという同じ操作を繰り返すことで和を計算することができるのが分かります。

プログラミングで扱う処理の中には、上記のように同じ操作を繰り返し行うものが多数できます。数列 x_i のように、インデクス (i) で値を指定できるデータの集まりを配列 (array) と呼びます。プログラミング言語の変数には型があります。配列は同じ型のデータの集まりであり、整数値をとるインデクスの値で何番目かを指定します。

Java では、配列の宣言を

型 変数名 [] ;

または

型 [] 変数名 ;

で行います。例えば、整数型の配列 `data` を宣言するには

```
int[] data;
```

とします。他のプログラミング言語と異なり、Java では配列もオブジェクトとなります。以下のように大きさ `n` を指定して、オブジェクトを生成することもできます。

```
int[] data = new int[n];
```

キーワード `new` は、新しいオブジェクトを生成することを表しています。この場合、配列の初期値として `0` が入ります。

もう一つの配列オブジェクトの生成方法は、データの内容を指定する方法です。

```
int[] data = {3,5,4,78,6,34,2,3};
```

右辺のデータの数に応じた大きさの配列が自動的に生成されます。配列の大きさは

```
data.length
```

で得ることができます。

```
int[] data = {3,5,4,78,6,34,2,3};
```

```
int size = data.length;
```

配列要素の読み書きには、何番目の要素かを指定します。 `i` 番目の要素は `data[i]` と指定します。インデックス `i` が `0` から始まる点に注意が必要です。

4.2 繰り返し

次に、配列を使って同じ操作を繰り返すことを考えましょう。繰り返し操作に対応するプログラムは

```
for(初期化; 条件; 再設定){
    繰り返す内容
}
```

のように書きます。繰り返し (loop) に入る際に「初期化」を行った後、「条件」が満たされている限りブロック (`{}`と)で囲まれた部分が繰り返されます。一回繰り返されるごとにインデックスの「再設定」が行われます。例にある

```
for(int i=0;i<data_num;i++){
    data[i] = (int)(100*Math.random());
}
```

の場合、変数 `i` が `0` に初期化された後、`i<data_num` である限り、`i` を `1` 増やしながら、配列 `data` に乱数を設定して行きます。変数 `i` は、このループの中だけで有効な変数です。

ここで、`Math.random()` は数学関数を持っているクラス `Math` に含まれる乱数生成関数 `random()` です。この関数は `0` から `1` までの一様乱数を生成します。右辺最初の `(int)` はキャストと呼ばれ、強制的に整数型に変換することを意味します。浮動小数点型から整数型へのキャストでは切捨てが行われます。つまり、`data` には `0` から `99` までの整数が保存されます。

4.3 その他の繰り返し

前節で扱った `for` を使った繰り返しでは、繰り返し回数 `i` を使うのが基本的な使い方です。繰り返しの中には、何かの条件を満たすまで繰り返すという、もっと一般的なものがあります。

繰り返しの一般的な方法の第一は `while` ループを記述するものです。

```
while(条件){  
    繰り返し内容  
}
```

のように使います。条件が満たされる限り、ブロック内部が実行されます。条件が満たされなくなった途端に、このブロックが飛ばされて、次のプログラムへ移行します。

第二の方法は、`do` と `while` を使った記法です。

```
do{  
    繰り返し内容  
}while(条件)
```

`while` を使った記法と似ていますが、ブロックは最低一回は実行され、その後に再度実行すべきかが判断されます。

いずれの記法でも、終了条件が何時までも満足されずに無限ループとなるようなことが無いようように注意が必要です。

演習 4.1 Program4.1.1 の `for` ループを `while` を使って書き換えなさい。

Program 4.1.1 ArrayAndLoop.java

```
/*
 * ArrayAndLoop.java
 *
 * Created on 2004/08/09, 11:42
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class ArrayAndLoop extends javax.swing.JApplet {
    private Font f=new Font("TimesRoman",Font.PLAIN,40);
    private final int data_num=10;
    private final int max=100;
    private int data[];

    /** Creates new form ArrayAndLoop */
    public ArrayAndLoop() {
        initComponents();
        data=new int[data_num];
        for(int i=0;i<data_num;i++){
            data[i] = (int)(100*Math.random());
        }
    }

    public void paint(Graphics g){
        g.setFont(f);
        double a=0.;
        double d=0.;
        for(int i=0;i<data.length;i++){
            a += data[i];
            d += data[i]*data[i];
        }
        a /= (double)data_num;
        d = d/data_num - a*a;
        g.drawString("Average : "+String.valueOf(a),0,50);
        g.drawString("RMS      : "+String.valueOf(d).substring(0,7),0,100);
    }

    private void initComponents() { //GEN-BEGIN: initComponents
    } //GEN-END: initComponents
}
```

第5章 条件分岐

5.1 条件分岐とは

これまでのプログラムの例では、実行は上から下へ向かってプログラムの各行が実行されていきました。実際の作業を考えると、条件に応じて実行される部分を変える必要がある場合があります。条件に応じてプログラムの実行を分けることを条件分岐と言います。

ここでは、でたらめに並んだデータを大きい(小さい)順に並べ直す (sort する) 問題を例に条件分岐の方法を学びます。

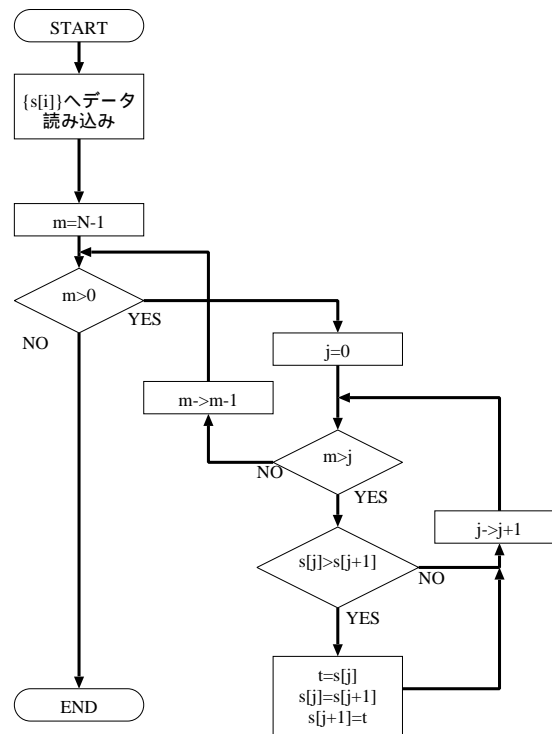


図 5.1.1: 泡立ち法

5.2 泡立ち法による並べ替え

データの並び替えには様々な方法が知られています。ここではもっとも簡単な泡立ち法 (bubble sort) を扱います。

操作手順を図示すると、整理や理解が容易になります。プログラムの際に手順を図示する方法の一つがフローチャート (流れ図)(flow chart) を描く方法です。泡立ち法のフローチャートを図 5.1.1 に示します。

まず、データが配列 $s[N]$ に保存されているとしましょう。データの総数は N 個です。まず、配列の先頭から、ある場所 i と $i+1$ とに保存されている値を比較し、 $s[i]$ が $s[i+1]$ より大きい場合に、順番を入れ換えることにします。この操作を配列の終わりまで一旦行くと、配列の中でもっとも大きい要素が、配列の一番後ろに移動します。

同じ操作をもう一度繰り返すと、今度は配列の中で二番目に大きい要素が配列の後ろから二番目の位置に移動します。以下、同じ操作を N 回繰り返せば、配列の要素を小さい順に並べかえることができます。

さて、ここで、一つ注意が必要です。一回目の操作で一番大きい要素が一番後ろに既に移動しています。従って、二回目の操作で、 $s[N-2]$ と $s[N-1]$ を比較する操作は不要です。従って、最初は、 N 個の要素全てを比較しなければなりません。次は $N-1$ 個、その次は $N-2$ 個と、次第に比較する要素数を減らし、効率良く並べ替えることが可能です。

このような操作によって、配列の内容は変化の様子を図 5.2.1 に示します。

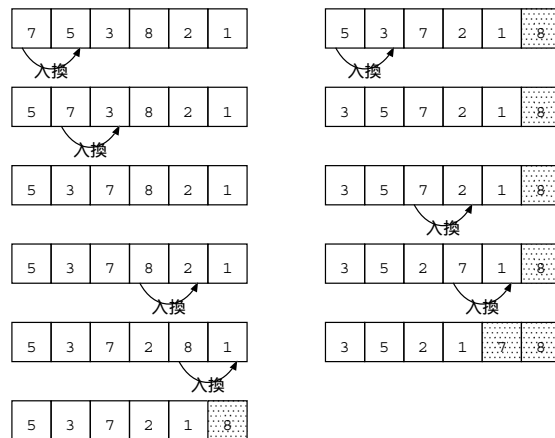


図 5.2.1: 泡立ち法によって配列の内容が変化の様子

演習 5.1 要素の比較がデータ数 N に対して、何回行われるか答えなさい。

Program 5.3.1 Condition.java

```
/*
 * Condition.java
 *
 * Created on 2004/09/13, 14:19
 */
import java.awt.*;
import java.lang.Math;
/**
 *
 * @author tadaki
 */
public class Condition extends javax.swing.JApplet {
    Font f = new Font("TimesRoman",Font.BOLD,20);
    final int data_num=10;
    int[] data=new int[data_num];

    /** Creates new form Condition */
    public Condition() {
        initComponents();
    }
    public void paint(Graphics g){
        g.setFont(f);
        initarray(data);
        String input=mkstr(data);
        int status=bubble(data);
        String output=mkstr(data);
        g.drawString("Input : "+input,0,20);
        g.drawString("Output: "+output,0,40);
    }

    private void initarray(int d[]){//でたらめな値を配列に保存
        for(int i=0;i<d.length;i++){
            d[i] = (int)(100*Math.random());
        }
    }
}
```

5.3 泡立ち法の Java プログラム

プログラム 5.3.1 に、泡立ち法のプログラムを示します。幾つか新しい新しい事柄が出て来ますので、順に説明していきましょう。

まず最初に泡立ち法を行っているメソッドを見ましょう。

```
public int bubble(int d[]){//泡立ち法
```

という部分に注目してください。この中に

```
    if(d[i] > d[i+1]){
        m++;
        int c=d[i];
```

Program 5.3.2 Condition.java(続き)

```

public int bubble(int d[]){//泡立ち法
    int m=0;
    for(int j=d.length-1; j>=1; j--){
        for(int i=0;i<j; i++){
            if(d[i] > d[i+1]){
                m++;
                int c=d[i];
                d[i]=d[i+1];
                d[i+1]=c;
            }
        }
    }
    return m;
}

private String mkstr(int d[]){
    StringBuffer str=new StringBuffer();
    str.append(String.valueOf(data[0]));
    for(int i=1;i<d.length;i++){
        str.append(",");
        str.append(String.valueOf(d[i]));
    }
    return str.toString();
}

private void initComponents() { //GEN-BEGIN: initComponents

    //GEN-END: initComponents
}

```

```

        d[i]=d[i+1];
        d[i+1]=c;
    }

```

という部分があります。ここが、配列の中で順序が小さい順でない場合に、順序を入れ換える部分です。

条件分岐の一般形は

```

if(条件){
    操作 1
}

```

です。条件が満たされると操作 1 が実行され、条件が満たされない場合には、操作 1 は実行されず、次に移動します。また、

```

if(条件){
    操作 1
} else {

```

```

操作 2
}

```

のような形式を使うことも出来ます。この場合、条件を満たす場合には操作 1 を満たさない場合には操作 2 を実行します。

条件を記述する際には、表 5.1 にある関係演算子を使います。また、論理演算子 `&&`(論理積)、`||`(論理和) 及び `!`(否定) も組み合わせて使うことができます。

表 5.1: 関係演算子

演算子	対応する数学記号	演算子	対応する数学記号
<code>></code>	$>$	<code><</code>	$>$
<code>>=</code>	\geq	<code><=</code>	\leq
<code>==</code>	$=$	<code>!=</code>	\neq

もう一度泡立ち法の中心部分を見てください。条件

```
d[i] > d[i+1]
```

が成り立つ場合に、二つの配列要素を入れ換えます。二つの要素の中身を直接に入れ換えることはできないので、`d[i]` の値を別の変数 `c` に一旦入れておき、`d[i]` に `d[i+1]` の値を入れた後、`c` に置いていた値を `d[i+1]` へ保存しています。

演習 5.2 プログラム 5.3.1 において、配列要素の比較が何回行われたかを表示できるようにプログラムを変更しなさい。また演習 5.1 で予想した結果を比較しなさい。

5.4 多数の条件への分岐

`if` を使った条件分岐は、ある条件を満たすか否かの二つに場合分けをすることができました。更に場合分けをしたい場合には、`if` ブロックの中で更に `if` で分岐します。

しかし、多数の条件への分岐を `if` で行うとプログラムが非常に読みにくくなる場合があります。そういう際に利用できるのが `switch` 文です。

各条件に相当する場合に整数の番号 k が付けられているとしましょう。`switch` 文は、それらの各番号に対応した処理を行うことができます。

```

switch(k){
  case 0:
    実行文 0;
    .....
    break;
  case 1:
    実行文 1;

```

```

.....
break;

....

default:
    実行文;
}

```

$k = 0$ の場合には実行文 0 が、 $k = 1$ の場合には、実行文 1 が実行されます。このように、各場合ごとの動作を case で区別しながら記述します。どの場合とも合致しない場合には、default に記述された部分が実行されます。

break は、プログラムブロックからの脱出を表す命令です。例えば $k = 0$ の場合の処理が終わったら、switch ブロックから脱出します。もしも $k = 0$ の場合の最後の break を忘れると、 $k = 0$ の場合の処理が終わった後、続けて $k = 1$ の場合の処理が行われます。

switch 文の実際の例題は、次章で示します。

5.5 メソッド入門

今回のプログラムでは、一つのクラスの中に複数のメソッド (method) が使われています。プログラムを書く際には、長い一つのメソッドではなく、小さなメソッドの集合としてプログラムを書き、一つのメソッドに一つの機能を持たせるようにします。そのようにすることで、プログラムの内容を整理して分かりやすくするとともに、再利用を容易にします。

再び bubble というメソッドに注目しましょう。

```
public int bubble(int d[]){//泡立ち法
```

最初のキーワード public は、このメソッドが外部から呼ばれることを許可しています。

次の int は戻り値の型を表しています。メソッドは、数学関数と同様に変数を与えると内部で操作を行い、値を返します。この int は、操作の結果が整数型で戻って来ることを表しています。今回の場合、実際に配列の要素の入れ換えが起こった回数を結果として返します。値を返さない場合には型を void とします。

この関数の引数 (arguments) は一つで、データの入っている配列です。メソッド paint から呼び出す際の変数名と異なることに注意してください。このように、変数名の有効範囲は、その変数が現れたプログラムブロック (括弧{と})で囲まれた部分) 内部だけです。

関数内部の操作の結果は return 文で行います。

第6章 簡単な作図

6.1 アプレットの動作の基本

いよいよアプレット (applet) 上に絵を描く準備に入りましょう。まず、アプレットでデフォルト (default) で定義されている幾つかのメソッド (method) について説明します。これらのメソッドは

```
public void
```

として定義されています。

処理	説明	メソッド名	引数
初期化	アプレットが最初にクライアントにロードされる際におきる。	init	なし
開始	アプレットが初期化された後におきる。別のページに移動した後に再び戻った場合も、開始動作が実行される。	start	なし
停止	アプレットを含むページから別のページに移った際に実行される。	stop	なし
破棄	アプレットやブラウザの終了時に実行される。	destroy	なし
描画	アプレット上に表示する際に実行される。	paint	Graphics 型

アプレットは、通常の Java のプログラムとは異なり、アプレットビューアー (AppletViewer) や Web ブラウザの中で起動されます。従って、基本的なグラフィック環境はアプレットビューアー (AppletViewer) や Web ブラウザが用意してくれます。また、Web ブラウザ内で起動される場合、利用者が次のページに移動した際に適切な動作を行わなければなりませんし、またアプレットのいるページに戻った際に、再起動が円滑でなければなりません。

そこで、アプレットが始めて起動した時の操作を `init` というメソッドに記述し、始めてであっても、二度目以降であっても表示される際の最初の処理を `start` に記述することで、処理を分割するようになっています。

また、Web ブラウザを起動したままで他のページに移動した際の動作を `stop` で、Web ブラウザが停止して本当にアプレットを停止する際の処理を `destroy` で記述します。

また、通常のプログラムであれば起動時にオプションを設定することができます。しかし、アプレットは Web ページの中に埋め込まれているためそのようなことができません。そこで、Web ページの中からオプションを渡す方法があります。そのことについても、後述します。

6.2 簡単な作図の基礎

いよいよアプレットに簡単な図を描いてみましょう。まず、アプレット上の座標について整理しておきます。アプレットの大きさは Web ページで定義されています。その左上を原点として、右方向に X 座標、下方向に Y 座標が定義されています。通常の座標とは Y 方向の向きが逆であることに注意してください。座標の単位はピクセル (pixel) で整数で指定します。

基本的な描画メソッドを見ていきます。全て Graphics クラスのメソッドとして定義されています。

直線を引く:直線を引くメソッドは `drawLine` です。始点と終点の X 座標及び Y 座標の 4 個の整数を指定します。

矩形を描く:矩形を描くメソッドは 3 種類あります。単純に矩形を描くには `drawRect` を用い、その中を塗りつぶすには `fillRect` を用います。左上隅の X 座標及び Y 座標及び幅と高さの 4 個の整数を指定します。

第二の種類は、角がまるまった矩形を描くものです。`drawRoundRect` と `fillRoundRect` がそれです。左上隅の X 座標及び Y 座標及び幅と高さの 4 個の整数を指定します。

残りの一つの種類は図形が浮き上がったたり沈んだりして見える効果があるものです。`draw3DRect` と `fill3DRect` がそれです。左上隅の X 座標及び Y 座標及び幅と高さの 4 個の整数を指定します。

多角形:多角形は `drawPolygon` と `fillPolygon` で描きます。X 座標の組を表す配列、Y 座標の組を表す配列、及び頂点の数の 3 個の変数を指定します。

また、多角形はポリゴンのオブジェクトとしても定義することができます。

```
int x[]={10,20,50,15};
int y[]={10,10,30,50};
int n=x.length;
Polygon poly = new Polygon(x,y,n);
```

このようにして作った多角形には `poly.addPoint(10,10)` のように、頂点を追加していくことができます。

円と楕円:円は長軸と単軸が等しい楕円として定義されます。`drawOval` と `fillOval` で描くことができます。楕円を取り囲む矩形の左上隅の X 座標及び Y 座標及び幅と高さの 4 個の整数を指定します。

弧:弧を描くには `drawArc` と `fillArc` を使います。その弧に外接する矩形を考えます。その矩形の左上隅の座標、幅と高さ、弧を描く始まりの角度とそこからの相対角度の 6 個の値を指定します。角度は 3 時の位置を 0 度として、反時計回りに 360 度までとします。

次に色を考えましょう。アプレット自身には背景色 (background color) と前景色 (foreground color) という属性があります。文字や図形を描くときに使われる色が前景色、アプレットの地の色

が背景色です。それぞれを設定するメソッドは `setForeground` と `setBackground` です。引数は `Color` オブジェクトです。

図形を描く直前に色を指定し、それ以降の描画の色を指定することもできます。Graphics オブジェクトのメソッド `Graphics.setColor` に `Color` オブジェクトを渡します。例えば

```
g.setColor(Color.green)
```

では、緑色が指定されます。

`Graphics.getColor()` を用いることで、現在設定されている色を調べることもできます。

文字列の表示は、今までも使ってきましたが、基本的性質をまとめておきましょう。文字列を表示するにはフォントが必要です。そのために、`Font` オブジェクトを生成しておきます。

```
Font f = new Font("TimesRoman", Font.BOLD, 24);
```

最初の引数は、フォントファミリーを、二番目はスタイルを、最後は大きさ (pixel) を示します。

次に `Font` オブジェクトを `Graphics` オブジェクトに登録します。

```
g.setFont(f);
```

これで文字列を描く準備が整いました。あとは、`drawString` メソッドで文字列を実際に描きます。

6.3 簡単な図形を描くプログラム

Program 6.3.1 SimpleGraphics.htm

```
<HTML>
<HEAD>
<TITLE>Simple Graphics</TITLE>
</HEAD>
<BODY>
<APPLET code="SimpleGraphics.class" width=150 height=150>
<PARAM NAME=shape VALUE="polygon">
</APPLET>
</BODY>
</HTML>
```

プログラム 6.3.2 は、アプレット上に簡単な作図をするプログラムです。このプログラムでは、Web からどの図形を描くかの情報を得て、作図をしています。

アプレットを呼び出している HTML ファイル (6.3.1) を見てみましょう。<APPLET>と</APPLET>に挟まれて

```
<PARAM NAME=shape VALUE="polygon">
```

という行があります。このタグによって変数名 `shape` の値として `polygon` を設定しています。

これに対応してプログラム 6.3.2 の中の `init()` メソッドにおいて

```
s=getParameter("shape");
```

Program 6.3.2 SimpleGraphics.java

```

/*
 * SimpleGraphics.java
 *
 * Created on 2004/09/13, 14:57
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class SimpleGraphics extends javax.swing.JApplet {

    private int flag;
    private Font f = new Font("TimesRoman",Font.PLAIN,20);
    private String s="NULL";
    static public final int SQUARE=1;
    static public final int THREEDSQUARE=2;
    static public final int CIRCLE=3;
    static public final int POLYGON=4;

    /** Creates new form SimpleGraphics */
    public SimpleGraphics() {
        initComponents();
    }

    public void init(){
        s=getParameter("shape");
        if(s.equals("square"))flag=SQUARE;
        if(s.equals("3Dsquare"))flag=THREEDSQUARE;
        if(s.equals("circle"))flag=CIRCLE;
        if(s.equals("polygon"))flag=POLYGON;
        setBackground(Color.yellow);
    }
}

```

を使って、HTML ファイルから変数名 `shape` に設定されている値を変数 `s` へ読み込んでいます。その後、文字列を比較するメソッド `equals` を使い

```
if(s.equals("square"))flag=SQUARE;
```

のように、形に応じ整数変数 `flag` の値を指定しています。

整数変数 `flag` に設定される `SQUARE` などの値は、クラスの先頭部分で

```
static public final int SQUARE=1;
```

のように定義されています。キーワード `final` は定数、つまり一度設定されると変更されない値であることを示しています。キーワード `static` は、この変数がクラスに共通な変数であることを示しています。このような `static` で宣言される変数やメソッドは、そのクラスのオブジェクトが生成されなくても、使うことができます。

実際の作図が行われる `paint()` メソッドの中では、前章で紹介して `switch` を使って、整数変数 `flag` の値に応じて図形が作図されています。

Program 6.3.3 SimpleGraphics.java(続き)

```
public void paint(Graphics g){
    g.setFont(f);
    g.setColor(Color.black);
    g.drawString(s+":"+String.valueOf(flag),10,140);

    switch(flag){
    case SQUARE://正方形
        g.setColor(Color.green);
        g.fillRect(10,10,50,50);
        break;
    case THREEDSQUARE://3次元正方形
        g.setColor(Color.red);
        g.fill3DRect(10,10,50,50,true);
        break;
    case CIRCLE://円
        g.setColor(Color.pink);
        g.fillOval(10,10,50,50);
        break;
    case POLYGON://多角形
        int x[]={10,35,24,60,48,50};
        int y[]={15,3,35,90,90,50};
        int np=x.length;
        g.setColor(Color.orange);
        g.fillPolygon(x,y,np);
        break;
    default:
        break;
    }
}

private void initComponents() { //GEN-BEGIN: initComponents

} //GEN-END: initComponents
}
```

演習 6.1 複数の図形を組み合わせた絵を描くアプレットを作成しなさい。

6.4 メソッドを調べる

一般に、プログラミングをするためには、その言語そのものについて知るだけでなく、基本となるライブラリについても知らなくてはなりません。Java の場合は、ライブラリはパッケージと呼ばれています。アプレットのうち基本となる部分は `java.awt`、`JApplet` などの先進的な部分は `javax.swing` というパッケージに含まれています。

こうしたパッケージをある程度使いこなさないと Java のプログラミングはできません。だからといって、そのパッケージのクラス名とメソッド名を全て知ることは困難です。

本講義で使っている Sun One Studio では、各クラスの持つメソッドを示す機能があります。プログラムを記述している最中に、クラス名あるいはそのオブジェクト名の後ろにピリオドを打つ

と、メソッドの一覧が表示されます。

また、標準的なパッケージのマニュアルがオンラインで提供されています。

<http://java.sun.com/j2se/1.4/ja/docs/ja/api/index.html>

第7章 簡単な動画

7.1 描画と再描画

いよいよ、動くホームページの入口に来ました。アプレット上に動画を表示することを考えましょう。ここで言う動画とは、既に存在している動画ファイル (mpeg など) を表示することではなく、静止画を連続して表示することで、動きを表示することを指します。

最初に、文字列が次々に変化する例題を見ましょう。ここでは、現在の時刻が表示されるアプレットを作成します。対応する HTML ファイルをプログラム 7.1.1 に示します。

Program 7.1.1 Clock.html

```
<HTML>
<HEAD>
<TITLE></TITLE>
<meta http-equiv="content-type" content="text/html; charset=EUC-JP">
</HEAD>
<BODY>
<P>ただ今の時刻</P>
<APPLET code=Clock.class width=500 height=50>
</APPLET>
</BODY>
</HTML>
```

動画を作成する基本となるのは、再描画 (repaint() メソッド) によって、描画の操作 (paint) を繰り返し呼び出すことで行います。

このように描画を繰り返す場合、スレッド (thread) というのをを使ってプログラムをする場合が多くあります。スレッドとは、「道筋」という意味があり、コンピュータの用語では処理の流れを表します。前章までのプログラムは、処理の道筋が一本しかなく (シングルスレッドと云う)、前の処理が終わると次の処理に移るようになっています。

Java はマルチスレッド (multithread) を簡単に作成できるプログラミング言語です。マルチスレッドとは、複数の処理が並行して行われるものを指します。もちろん、CPU が一つしかないシステムでは、一度に行われる処理は一つですが、処理は細かい時間に分割されて進み、各スレッドは独立に進むことができます。

マルチスレッドで時刻表示のプログラムを作るには、初期化処理で単に新しいスレッドを生成し、そのスレッドの中で、再描画を行うようにします。

Program 7.1.2 Clock.java

```
/*
 * Clock.java
 *
 * Created on 2004/11/17, 17:51
 */
import java.awt.*;
import java.util.Date;
/**
 * 現在の時刻を文字列で表示するアプレット
 * @author tadaki
 */
public class Clock extends javax.swing.JApplet
    implements Runnable //スレッドを使うことを指示
{
    private Font f = new Font("TimesRoman", Font.BOLD, 24);
    private Date theDate;
    private Thread runner;

    /** Creates new form Clock */
    public Clock() {
        initComponents();
    }

    public void init() { //アプレットの前景色と背景色を設定
        getContentPane().setBackground(Color.cyan); //背景色
        setForeground(Color.red); // 前景色
    }

    public void start() {
        if (runner == null) { //起動時にスレッドを生成する
            runner = new Thread(this);
            runner.start();
        }
    }

    public void stop() {
        if (runner != null) { //終了時にスレッドを破棄する
            runner = null;
        }
    }

    public void run() { //スレッドで実行すること
        while(true) {
            theDate = new Date();
            repaint(); //再描画
            //1000 ミリ秒待ち、その間に割り込みが発生したら何も行わない
            try {Thread.sleep(1000);}
            catch (InterruptedException e){}
        }
    }
}
```

Program 7.1.3 Clock.java の続き

```
public void paint(Graphics g) {
    super.paint(g);
    g.setFont(f);
    g.drawString(theDate.toString(),10,40);// 現在の日付時刻を表示
}
private void initComponents() { //GEN-BEGIN:initComponents

} //GEN-END:initComponents

// Variables declaration - do not modify //GEN-BEGIN:variables
// End of variables declaration //GEN-END:variables

}
```

スレッドを使った時刻表示アプレットをプログラム 7.1.2 に示します。スレッドを使うクラスの宣言には

```
implements Runnable
```

というキーワード (interface と呼びます) を付けます。スレッドは Thread というクラスのオブジェクトとして生成されます。

ここでは、runner という名のスレッドが宣言され、start() メソッドでは、単にスレッドの生成と開始が、stop() メソッドではスレッドの停止が行われています。

スレッドの動作は run() というメソッドで記述されます。このメソッドの中で、再描画を repaint() を使って行った後、

```
try {Thread.sleep(1000);}
catch (InterruptedException e){}
```

を使って、1000 ミリ秒停止します。

実際の描画は paint で行われます。最初の super.paint は、今作成しているプログラムのクラス Clock の親のクラス JApplet のメソッド paint を表しています。この操作で、背景色で一旦全てが塗りつぶされます。その後でフォントが設定されて、時刻を表す文字列が表示されます。

現在の時刻は、java.util.Date というクラスを使って見ることができます。

```
theDate = new Date();
```

によって、現在の時刻がオブジェクト theDate に保存されます。メソッド

```
theDate.toString()
```

は、保持している時刻を文字列に変換します。

7.2 動く正方形

もう一つ、簡単な動画プログラムを考えましょう。プログラム 7.2.1 は、正方形が時刻とともに、伸び縮みしながら右へ移動するアプレットです。

まず、時刻を保持する変数を t とします。スレッドの動作 `run` の中で、時刻は 1 ずつ増加しますが、500 回繰り返すと 0 に戻るようになっています。

```
t++; t = t%500;
```

ここで `t=t%500` は、 t の値を 500 で割った余りを t に代入することを表しています。

各頂点の座標が配列 `sqr` に保存されています。メソッド `setsqr` で座標が計算されています。時刻 t とともに、左下の頂点の位置が

```
x=10+t;
```

で移動してきます。同時に高さが

```
h=2*(t%50);
```

で大きくなります。

演習 7.1 三角形が伸び縮みしながら並行移動するアプレットを作成しなさい。

Program 7.2.1 MovingSquare.java

```
/*
 * MovingSquare.java
 *
 * Created on 2004/11/17, 17:13
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class MovingSquare extends javax.swing.JApplet
implements Runnable{
    private Thread runner;
    private int t=0;
    private int sqr[]={10,10,50,50};

    /** Creates new form MovingSquare */
    public MovingSquare() {
        initComponents();
//前景色と背景色を設定
        getContentPane().setBackground(Color.cyan);
        setForeground(Color.red);
    }

    public void start() {
        if (runner == null) {//スレッド起動
            runner = new Thread(this); runner.start();
        }
    }

    public void stop() {
        if (runner != null) {//スレッド停止
            runner = null;
        }
    }

    public void run() {
        while(true) {
            t++; t = t%500;
            repaint();
            try {Thread.sleep(100);}
            catch (InterruptedException e){}
        }
    }
}
```

Program 7.2.2 MovingSquare.java の続き

```
public void paint(Graphics g) {
    super.paint(g);
    g.setColor(getForeground());
    setsqr();
    g.fillRect(sqr[0],sqr[1],sqr[2],sqr[3]);
}

private void setsqr() { //位置の再計算
    int x=10+t;
    int h=2*(t%50);
    if(h>50)h=100-h;
    int y=35-h/2;
    sqr[0]=x; sqr[1]=y; sqr[3]=h;
}

private void initComponents() { //GEN-BEGIN:initComponents

} //GEN-END:initComponents

// 変数宣言 - 編集不可 //GEN-BEGIN:variables
// 変数宣言の終わり //GEN-END:variables

}
```

第8章 クラスを作る

8.1 クラスとは

Program 8.1.1 MySquare.java

```
/*
 * MySquare.java
 *
 * Created on 2004/11/30, 16:00
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class MySquare {
    private int x,y;
    private int w=50,h;
    private Color back,fore;

    /** Creates a new instance of MySquare */
    MySquare(Color b,Color f){
        back=b; fore=f;
    }

    public void set(int t){
        x=10+t;
        h=2*(t%50);
        if(h>50)h=100-h;
        y=35-h/2;
    }

    public void clean(Graphics g){
        g.setColor(back);//背景色で部分的に塗り潰す
        g.fillRect(x,y,w,h);
    }

    public void draw(Graphics g){
        g.setColor(fore);//全景色で部分的に塗り潰す
        g.fillRect(x,y,w,h);
    }
}
```

先週のプログラム `MovingSquare.java` を思い出しましょう。 `paint` メソッドの中で、直前に描いた部分を背景色で塗りつぶし、新しい矩形の情報を計算し、前景色で塗りつぶしました。 `paint` メソッドの中で、矩形の情報が直接使われ、 `fillRect` が直接呼ばれていました。

Program 8.1.2 `MovingSquare.java`

```
/*
 * MovingSquare.java
 *
 * Created on 2004/11/30, 15:59
 */
import java.awt.*;

/**
 *
 * @author tadaki
 */
public class MovingSquare extends javax.swing.JApplet implements Runnable{
    private Thread runner;
    private int t=0;
    private MySquare sqr;

    /** Initializes the applet MovingSquare */
    public void init() {
        initComponents();
        //背景色の設定
        getContentPane().setBackground(Color.cyan);
        getContentPane().setForeground(Color.red);
        //矩形の定義
        sqr = new MySquare(
            getContentPane().getBackground(),
            getContentPane().getForeground());
    }

    public void start() {
        if (runner == null) { //スレッド起動
            runner = new Thread(this); runner.start();
        }
    }

    public void stop() {
        if (runner != null) { //スレッド停止
            runner = null;
        }
    }
}
```

プログラムを書く場合、データの塊ごとあるいは機能ごとにプログラムを分けることで、読みやすく開発コストの小さなプログラムを書くことができます。そこで、この部分を次のような方向に書き換えることを考えましょう。つまり、矩形に対応するモノをひとかたまりに考えます。メインのプログラムは、このモノに対して、「消せ」、「再設定せよ」、「描画せよ」と命令するだけにするのです。メインのプログラムの方からは、矩形の情報がどのように保持されているかは知らなくて

良いようにします。

このようにすることで、メインのプログラムを描かれる図形から切り離し、他の図形を描くことにも使えるようにすることができます。

Program 8.1.3 MovingSquare.java 続き

```
public void run() {
    while(true) {
        t++; t = t%500;
        repaint();
        try {Thread.sleep(100);}
        catch (InterruptedException e){}
    }
}

public void paint(Graphics g) {
    super.paint(g);
    sqr.clean(g);
    sqr.set(t);
    sqr.draw(g);
}

/** This method is called from within the init() method to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
private void initComponents() { //GEN-BEGIN:initComponents

} //GEN-END:initComponents

// Variables declaration - do not modify //GEN-BEGIN:variables
// End of variables declaration //GEN-END:variables
}
```

このように対象の動作を中心にプログラムを作り上げることをオブジェクト指向プログラミング (Object Oriented Programming) と呼びます。対象をオブジェクト (object)、オブジェクトを抽象化した型に相当するものをクラス (class) と呼びます。逆にいうと、類型をクラスと呼び、それが実体化されたものをオブジェクトと呼びます。

つまり、矩形に対応するクラスを定義し、そのクラスに属するオブジェクトに「消せ」、「再設定せよ」、「描画せよ」と命令するだけにします。

矩形のクラスをプログラム 8.1.1 に示します。矩形を表示するための情報として、左上隅の座標、幅、高さ、そして背景色と前景色を持っています。

クラス名と同じ名前のメソッドはコンストラクタ (Constructor) と呼ばれる特殊なメソッドで、new などを使って初期化する際に利用されます。

メインプログラム 8.1.2 からは、クラス MySquare のオブジェクトが定義され、メインプログラ

△のコンストラクタで背景色と前景色が指定されます。

```
sqr = new MySquare(  
    getContentPane().getBackground(),  
    getContentPane().getForeground());
```

その後、paint メソッド内では、矩形クラスのメソッド clean、set 及び draw が呼ばれているだけで、中で何をやっているかをメインプログラムが知る必要はありません。

Program 8.1.4 MovingSquare.html

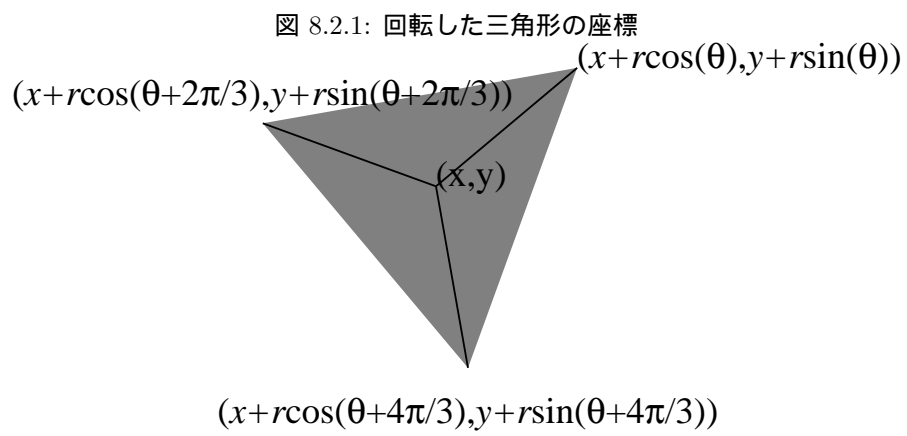
```
<HTML>  
<HEAD>  
  <TITLE>Applet HTML Page</TITLE>  
</HEAD>  
<BODY>  
  
<H3>Applet HTML Page</H3>  
  
<APPLET code="MovingSquare.class" width=350 height=200>  
</APPLET>  
  
<HR>  
</BODY>  
</HTML>
```

プログラム 8.1.4 は、矩形のクラスを使った Applet を呼ぶ HTML ファイルです。ここでは、矩形のクラスを呼ぶ部分は必要ありません。矩形のクラス MySquare.class が同じディレクトリにあれば十分です。

8.2 ころがる三角形

応用として矩形ではなく三角形を転がしながら描くことを考えましょう。プログラム 8.2.1 は、前節のプログラムの矩形に対応した三角形のクラスです。三つの頂点の座標は、図 8.2.1 のように指定することで一つの点からの相対座標として指定することができます。

演習 8.1 三角形が回転しながら移動するアプレットを作成しなさい。



Program 8.2.1 MyTriangle.java

```
/*
 * MyTriangle.java
 *
 * Created on 2004/11/30, 17:21
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class MyTriangle {
    private int x;
    private final int y=150;
    private final int r=100;
    private int xx[]={0,0,0};
    private int yy[]={0,0,0};
    private Color back,fore;

    /** Creates a new instance of MyTriangle */
    public MyTriangle(Color back,Color fore) {
        this.back=back; this.fore=fore;
        //this は自分を表す
    }

    public void set(int t){
        x=10+t;
        double s=0.05*t;
        xx[0]=(int)(x+r*Math.cos(s));
        xx[1]=(int)(x+r*Math.cos(s+2*Math.PI/3));
        xx[2]=(int)(x+r*Math.cos(s+4*Math.PI/3));
        yy[0]=(int)(y+r*Math.sin(s));
        yy[1]=(int)(y+r*Math.sin(s+2*Math.PI/3));
        yy[2]=(int)(y+r*Math.sin(s+4*Math.PI/3));
    }

    public void clean(Graphics g){
        g.setColor(back);
        g.fillPolygon(xx,yy,3);
    }

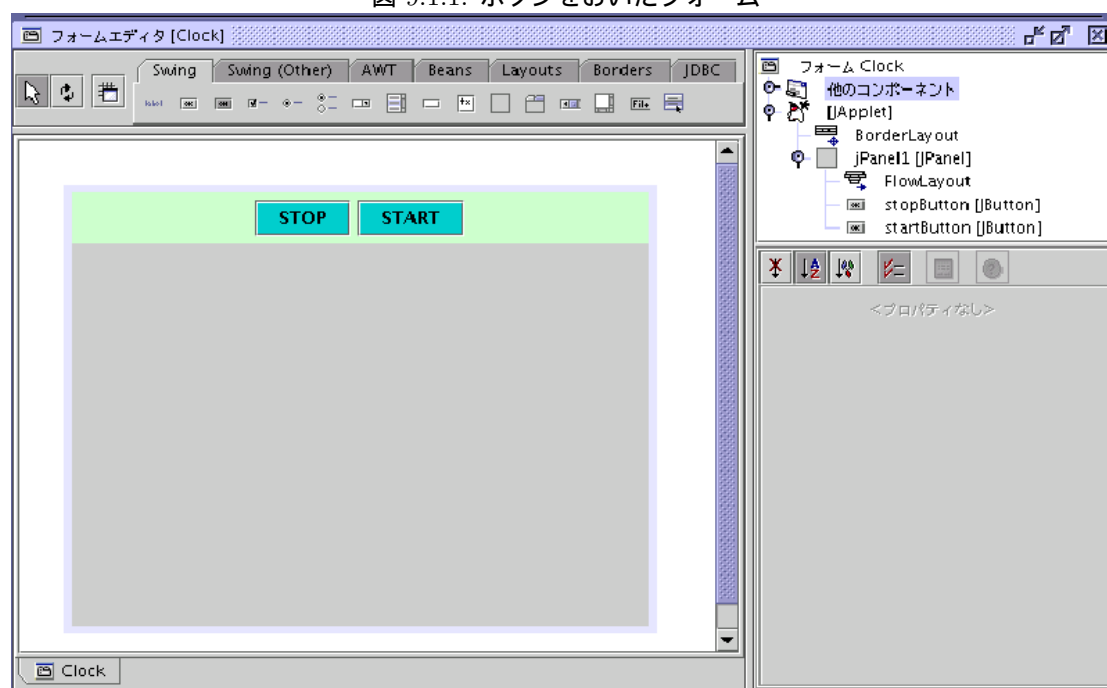
    public void draw(Graphics g){
        g.setColor(fore);
        g.fillPolygon(xx,yy,3);
    }
}
```

第9章 グラフィカルユーザインターフェイス

9.1 ボタンを作る

利用者からの様々な情報を、マウスなどの操作によって得るためのインターフェイスをグラフィカルユーザインターフェイス (GUI, Graphical User Interface) と呼びます。Sun One Studio は、そうした GUI 構築を支援する機能を有しています。ここでは、以前に作成した時計のプログラムにスタートとストップのボタンを作ってみましょう。

図 9.1.1: ボタンをおいたフォーム



まず、以前の時計のプロジェクトを開きます。ソースファイルの「フォームエディタ」を開きます。ここにボタンをおくパネルを作りましょう。「フォームエディタ」内からパネル (JPanel) を選んでフォームの中に置きましょう。右側にそれぞれの部品の「プロパティ」があります。そこで、このパネルの「レイアウト」を「North」としておきます。プロパティで色などを指定してもよいでしょう。

次にこのパネル (JButton) の中にボタンを置きます。ボタンの「プロパティ」の「text」を「STOP」

Program 9.1.1 Clock.java

```

/*
 * Clock.java
 *
 * Created on 2004/11/17, 17:51
 */
import java.awt.*;
import java.util.Date;
/**
 * 現在の時刻を文字列で表示するアプレット
 * @author tadaki
 */
public class Clock extends javax.swing.JApplet
    implements Runnable //スレッドを使うことを指示
{
    private Font f = new Font("TimesRoman", Font.BOLD, 24);
    private Date theDate;
    private Thread runner;
    private boolean id=true;

    /** Creates new form Clock */
    public Clock() {
        initComponents();
    }

    public void init() { //アプレットの前景色と背景色を設定
        getContentPane().setBackground(Color.cyan); //背景色
        setForeground(Color.red); // 前景色
    }
}

```

と書き換えましょう。またボタンの名前を「stopButton」として置きましょう。同様に、もう一つのボタンを作り「text」を「START」に、名前を「startButton」としましょう。

パネルの中にボタンを二つ置いたときの「フォームエディタ」の様子を図 9.1.1 に示します。

ボタンを押した際の動作を記述する前に、時計を表示するかどうかを決める変数を定義しておきます。

```
private boolean id=true;
```

つぎにこの変数を見て時間を更新するように、run メソッドを変更しておきましょう。

```

public void run() { //スレッドで実行すること
    while(true) {
        if(id){
            theDate = new Date();
        }
        repaint(); //再描画

        //1000 ミリ秒待ち、その間に割り込みが発生したら何も行わない
    }
}

```

Program 9.1.2 Clock.java の続き

```
public void start() {
    if (runner == null) { //起動時にスレッドを生成する
        runner = new Thread(this);
        runner.start();
    }
}

public void stop() {
    if (runner != null) { //終了時にスレッドを破棄する
        runner = null;
    }
}

public void run() { //スレッドで実行すること
    while(true) {
        if(id){
            theDate = new Date();
        }
        repaint(); //再描画

        //1000 ミリ秒待ち、その間に割り込みが発生したら何も行わない
        try {Thread.sleep(1000);}
        catch (InterruptedException e){}
    }
}

public void paint(Graphics g) {
    super.paint(g);
    g.setFont(f);
    if(theDate!=null)
        g.drawString(theDate.toString(),10,40); // 現在の日付時刻を表示
}

    try {Thread.sleep(1000);}
    catch (InterruptedException e){}
}
}
```

次にそれぞれのボタンを押したときの動作を記述しましょう。「ストップ」ボタンを押した時には、時計を止めることにします。まず、「ストップ」ボタンをダブルクリックします。すると、「ソースエディタ」の相当する箇所に移動します。ここで変数 `id` を偽にします。

```
private void stopButtonActionPerformed(java.awt.event.ActionEvent evt) {
    id=false;
}
```

つまり、「ストップ」ボタンを押すと、アプレット内の `stop` メソッドが呼ばれ、スレッドが停止します。

Program 9.1.3 Clock.java の続き

```

private void initComponents() {
    jPanel1 = new javax.swing.JPanel();
    stopButton = new javax.swing.JButton();
    startButton = new javax.swing.JButton();

    jPanel1.setBackground(new java.awt.Color(204, 255, 204));
    stopButton.setBackground(new java.awt.Color(0, 204, 204));
    stopButton.setText("STOP");
    stopButton.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            stopButtonActionPerformed(evt);
        }
    });

    jPanel1.add(stopButton);

    startButton.setBackground(new java.awt.Color(0, 204, 204));
    startButton.setText("START");
    startButton.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            startButtonActionPerformed(evt);
        }
    });

    jPanel1.add(startButton);

    getContentPane().add(jPanel1, java.awt.BorderLayout.NORTH);
}

```

同様に「スタート」ボタンを押した際には、id を真にするように記述しましょう。

```

private void startButtonActionPerformed(java.awt.event.ActionEvent evt) {
    id=true;
}

```

全体のプログラムをプログラム 9.1.1 と 9.1.2 に示します。

演習 9.1 作成したアプレットでは、時刻の文字がボタンと重なってしまいました。文字の位置を変えて、重ならないように変更しなさい。

9.2 一覧から選ぶ

つぎに、一覧から項目を選ぶ GUI を作りましょう。一覧は JComboBox というパーツで作ります。今回のプログラムでは、色の一覧を作りましょう。

Program 9.1.4 Clock.java の続き

```

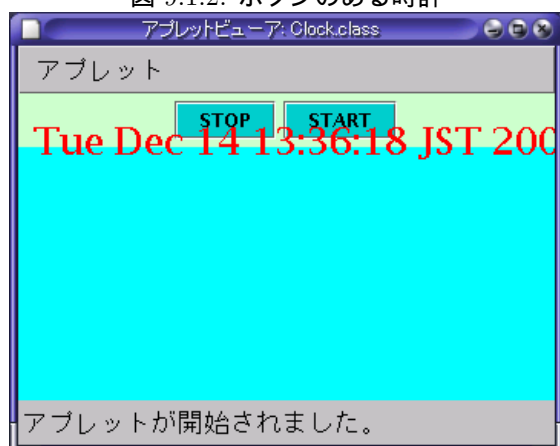
private void startButtonActionPerformed(java.awt.event.ActionEvent evt) {
    id=true;
}

private void stopButtonActionPerformed(java.awt.event.ActionEvent evt) {
    id=false;
}

// 変数宣言 - 編集不可
private javax.swing.JButton stopButton;
private javax.swing.JButton startButton;
private javax.swing.JPanel jPanel1;
// 変数宣言の終わり
}

```

図 9.1.2: ボタンのある時計



まず、色とその名前を保持する小さなクラス `ColorList` を作ります (プログラム 9.2.1)。内部に色を表す変数 `color` とその名前 `colorname` を持っています。メソッド `toString` は、色の名前を返します。メソッド `getColor` は保持されている色を返します。

このプログラムを先に作成しておきましょう。テンプレートはただの `Class` を使います。

つぎに、メインの部分を作りましょう。まず、アプレット `ColorChoice` を作りましょう。続いて、一覧を表示する `JComboBox` をフォームに置きます。名前を、`colors` とつけて置きましょう。レイアウトは `North` とします。次に、色を表示する `JPanel` をフォームに置きます。名前を `panel` とつけます。レイアウトは `Center` とします。

プログラムの先頭に設定できる色の一覧 `clist` を記述します。次に、その色を一覧 `color` に設定します。これをコンストラクタ内に記述します。

```

public ColorChoice() {
    initComponents();
}

```

Program 9.2.1 ColorList.java

```

/*
 * ColorList.java
 *
 * Created on 2004/12/14, 11:14
 */

/**
 *
 * @author tadaki
 */
public class ColorList {
    private java.awt.Color color;
    private String colorname;
    /** Creates a new instance of ColorList */
    public ColorList(java.awt.Color color, String colorname) {
        this.color=color; this.colorname=colorname;
    }

    /** 色の名前を返す関数
     * @return 色の名前
     */
    public String toString(){return colorname;}

    /** 色を返す関数
     * @return 色
     */
    public java.awt.Color getColor(){return color;}
}

```

```

        for(int i=0;i<clist.length;i++){
            colors.addItem(clist[i]);
        }
    }
}

```

更に、フォーム内の一覧 colors をダブルクリックします。すると、メソッド colorsActionPerformed が作成されます。その中に以下のように記述します。

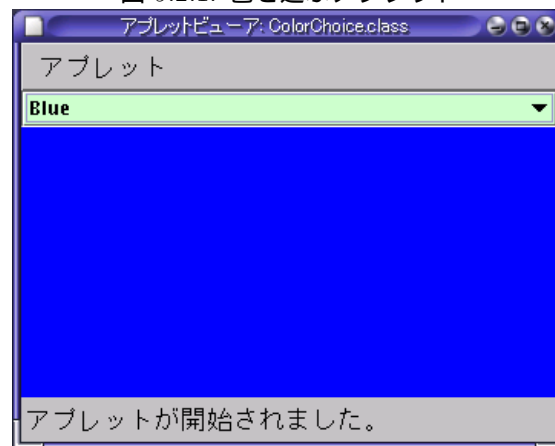
```

/** 色の名前を選択すると、その色をパネルに設定する*/
private void colorsActionPerformed(java.awt.event.ActionEvent evt)
{
    //GEN-FIRST:event_colorsActionPerformed
        ColorList color=(ColorList)colors.getSelectedItemAt();
        panel.setBackground(color.getColor());
    //GEN-LAST:event_colorsActionPerformed
}

```

ここでやっていることは、一覧 color から、選択されたモノを返し (getSelectedItem())、ColorList へリダイレクト (型変換) の後、その色をパネルに設定しています。

図 9.2.1: 色を選ぶアプレット



演習 9.2 一覧とパネルの色や配置を変更しなさい。

演習 9.3 終了ボタンを追加しなさい。

Program 9.2.2 ColorChoice.java

```
/*
 * ColorChoice.java
 *
 * Created on 2004/12/14, 11:14
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class ColorChoice extends javax.swing.JApplet {
    ColorList clist[]={
        new ColorList(Color.white,"White"),
        new ColorList(Color.blue,"Blue"),
        new ColorList(Color.cyan,"Cyan"),
        new ColorList(Color.gray,"Gray"),
        new ColorList(Color.green,"Green"),
        new ColorList(Color.magenta,"Magenta"),
        new ColorList(Color.orange,"Orange"),
        new ColorList(Color.pink,"Pink"),
        new ColorList(Color.red,"Red"),
        new ColorList(Color.yellow,"Yellow")
    };

    /** Creates new form ColorChoice */
    public ColorChoice() {
        initComponents();
        for(int i=0;i<clist.length;i++){
            colors.addItem(clist[i]);
        }
    }
}
```

Program 9.2.3 ColorChoice.java 続き

```
/** This method is called from within the constructor to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
private void initComponents() { //GEN-BEGIN:initComponents
    colors = new javax.swing.JComboBox();
    panel = new javax.swing.JPanel();

    colors.setBackground(new java.awt.Color(204, 255, 204));
    colors.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            colorsActionPerformed(evt);
        }
    });

    getContentPane().add(colors, java.awt.BorderLayout.NORTH);

    getContentPane().add(panel, java.awt.BorderLayout.CENTER);

} //GEN-END:initComponents

/** 色の名前を選択すると、その色をパネルに設定する*/
private void colorsActionPerformed(java.awt.event.ActionEvent evt)
{ //GEN-FIRST:event_colorsActionPerformed
    ColorList color=(ColorList)colors.getSelectedItem();
    panel.setBackground(color.getColor());
} //GEN-LAST:event_colorsActionPerformed

// Variables declaration - do not modify //GEN-BEGIN:variables
private javax.swing.JComboBox colors;
private javax.swing.JPanel panel;
// End of variables declaration //GEN-END:variables
}
```

第10章 マウスの動きを使う

10.1 マウスイベント

Program 10.1.1 DrawSquare.java

```
/*
 * DrawSquare.java
 *
 * Created on 2004/12/14, 18:00
 */
import java.awt.*;
import java.awt.event.*;

/**
 *
 * @author tadaki
 */
public class DrawSquare extends javax.swing.JApplet
    implements MouseListener, MouseMotionListener
    //マウスの動作を拾うためのインターフェイスの定義
{
    private final int MAXSQR=100;
    //四角形
    private MySquare sqr[]= new MySquare[MAXSQR];
    private MySquare tmp;
    private int currentsq=0;

    /** Initializes the applet DrawSquare */
    public DrawSquare() {
        initComponents();
        //マウスの動作を拾うインターフェイスの登録
        addMouseListener(this);
        addMouseMotionListener(this);
    }
}
\end{Program}
//色の設定
getContentPane().setForeground(Color.blue);
getContentPane().setBackground(Color.yellow);
}
```

ここまで作成してきたプログラムは、スレッドに分かれるところはありませんでしたが、作成したプログラムの範囲内で動いているように見えました。しかし、よく考えると、ブラウザが別のページに移った際にアプレットが停止するなど、プログラムの外部から割り込んで来るものがありました。

Program 10.1.2 DrawSquare.java つづき

```

public void paint(Graphics g){
    super.paint(g);
    Color c=getContentPane().getForeground();
    for(int i=0;i<currentsq; i++){//既に確定した四角形の描画
        sqr[i].draw(g,c);
    }
    if(tmp!=null) { //現在作成中の四角形
        tmp.draw(g,Color.red);
    }
}

/** This method is called from within the init() method to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
private void initComponents() { //GEN-BEGIN: initComponents

} //GEN-END: initComponents

```

Java アプレットでは、アプレット内のマウスの動きやキーボード操作をイベント (event) として認識し、割り込みをかけることができるようになっています。このようにイベントが起きると、それに応じてプログラムを動作させることをイベント駆動 (event-driven) といいます。ここまで見てきたプログラムでも、他のウィンドウの下からアプレットが現れた際に画面を再描画するというかたちでイベント駆動が行われていました。

本章では、マウスの動きを追って、矩形を描くプログラムを作成しましょう。

まず、マウスイベントを拾うために、クラスにインターフェイスを追加します。クラス定義部分の

```
implements MouseListener, MouseMotionListener
```

がその部分です。MouseListener がマウスボタンの操作を、MouseMotionListener がマウスの移動などの操作を調べるためのインターフェイスです。さらに、コンストラクタ DrawSquare() においてインターフェイスを追加します。

```
addMouseListener(this);
addMouseMotionListener(this);
```

インターフェイスはクラスによく似たものですが、クラスとは異なりメソッドの名前だけが定義されています。使用する際には、そのインターフェイスで定義されているすべてのメソッドの動作を定義する必要があります。使用している統合環境では、インターフェイスを定義したところで、自動的にすべてのメソッドが追加されます。

今回のプログラムにおいても MouseListener と `MouseListener` に対応するマウスの動きに対応する関数を定義していきます。表 10.1 にある 7 種類の動作全てを記述しなければなりません。必要の無い動作は、何もしないように定義します。これらの関数は全て

Program 10.1.3 DrawSquare.java つづき (2)

```

public void mouseDragged(MouseEvent e) { // ボタンを押しながら移動した時
    tmp.set(e.getX(), e.getY());
    repaint();
}

public void mousePressed(MouseEvent e) { // ボタンを押した時
    tmp=new MySquare(e.getX(), e.getY());
}

public void mouseReleased(MouseEvent e) { // B ボタンを離した時
    if(currentsq < MAXSQR){
        tmp.set(e.getX(), e.getY());
        sqr[currentsq]=new MySquare(tmp);
        tmp=null;
        currentsq++;
    }
    repaint();
}

// 使わないマウスの動作も定義しなくてはならない
public void mouseMoved(MouseEvent e) {}
public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}

// Variables declaration - do not modify//GEN-BEGIN:variables
// End of variables declaration//GEN-END:variables
}

```

```
public void
```

として定義します。また、引数は `MouseEvent` 型です。

プログラム 10.1.1 では、マウスを押して (`mousePressed`) 新しい矩形の描画を開始し、マウスを押しながら移動して (`mouseDragged`)、マウスを放す (`mouseReleased`) と一つの矩形が確定するようになっています。

マウスボタンを押して移動中には、一時的な矩形 `tmp` として描画していますが、マウスボタンを放すと矩形データを保持する配列 `sqr` に登録されます。

マウスが押された位置は `MouseEvent` クラスのメソッド `getX()` と `getY()` を使って得ます。どのボタンを押されたかの区別はできません。

各マウスイベントの処理が終ると `repaint()` で描画されます。描画 (`paint()`) の際には、まず確定している矩形 (配列 `sqr` に保存されている) を前景色で描いた後、現在定義中の矩形 (`tmp` に保存されている) を赤で描画します。

表 10.1: マウスの動き

関数名	内容
mouseClicked	この部分にマウスがクリックされると呼び出される
mouseEntered	この部分にマウスが入ると呼び出される
mouseExited	この部分からマウスが出ると呼び出される
mousePressed	この部分でマウスボタンが押されると呼び出される
mouseReleased	この部分でマウスボタンが放されると呼び出される
mouseDragged	この部分でマウスボタンを押しながら動かすと呼び出される
mouseMoved	この部分でマウスボタンを押さずに動かすと呼び出される

10.2 矩形のクラス

今回も、矩形のクラスを使用します。前回と少し異なる点を説明します。矩形クラス `MySquare` は、左上隅と右下隅の二つの点の位置情報を `Point` クラスとして保持しています。右下隅の座標が指定されると、幅 `w` と高さ `h` を計算します。

メソッド

```
MySquare(final MySquare s)
```

は、他の矩形をコピーするのに用います。コピー元の左上隅と右下隅の二つの点の位置情報をコピーするとともに、幅 `w` と高さ `h` を計算します。

メソッド `isInside` は今回使いません。次回に使う際に説明します。

演習 10.1 マウスを押すことで中心を決め、押しながら移動して半径を設定し、マウスを放すと円が描かれるアプレットを作成しなさい。

Program 10.2.1 MySquare.java

```
/*
 * MySquare.java
 *
 * Created on 2004/12/14, 17:51
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class MySquare {
    private Point start,end;//左上隅と右下隅の点
    public int w,h;//幅と高さ

    /** Creates a new instance of MySquare */
    public MySquare(int x,int y) {
        start = new Point(x,y);
    }

    /** すでに存在している矩形をコピーする
     * @param s コピー元の矩形*/
    public MySquare(final MySquare s){
        start=new Point(s.start);
        end = new Point(s.end);
        w = end.x - start.x;
        h = end.y - start.y;
    }

    /** 右下隅の設定*/
    public void set(int x,int y){
        if(end==null){
            end = new Point(x,y);
        } else {
            end.x = x; end.y=y;
        }
        w = end.x - start.x;
        h = end.y - start.y;
    }
}
```

Program 10.2.2 MySquare.java つづき

```
public void draw(Graphics g,Color c){
    g.setColor(c);
    g.drawRect(start.x,start.y,w,h);
}

public boolean isInside(Point p){
//点 p が四角形の内部か外部かを判定
    boolean id=false;
    if(start.x <= p.x && p.x <= end.x){
        if(start.y <= p.y && p.y <= end.y){
            id=true;
        }
    }
    return id;
}
}
```

第11章 キーボードからの入力を使う

11.1 キーボードイベント

Program 11.1.1 DrawSquare.java

```
/*
 * DrawSquare.java
 *
 * Created on 2004/12/14, 18:00
 */
import java.awt.*;
import java.awt.event.*;

/**
 *
 * @author tadaki
 */
public class DrawSquare extends javax.swing.JApplet
    implements MouseListener, MouseMotionListener,
        //マウスの動作を拾うためのインターフェイスの定義
        KeyListener
        //キーボードイベントを拾うためのインターフェイス
{

    //矩形のリスト
    private java.util.Vector sqrList=null;
    private MySquare tmp;
    private Point present=new Point(0,0);
    private int selected=-1;//選択された矩形の番号

    /** Initializes the applet DrawSquare */
    public DrawSquare() {
        initComponents();
        //マウスの動作を拾うインターフェイスの登録
        addMouseListener(this);
        addMouseMotionListener(this);
        addKeyListener(this);
        setFocusable(true);
        //色の設定
        getContentPane().setForeground(Color.black);
        getContentPane().setBackground(Color.yellow);
        sqrList=new java.util.Vector();
    }
}
```

Program 11.1.2 DrawSquare.java つづき

```

public void paint(Graphics g){
    super.paint(g);
    Color c=getContentPane().getForeground();
    if(sqrList==null)return;
    int maxsqr=sqrList.size();//保存されている矩形の数
    for(int i=0;i<maxsqr;i++){//既に確定した四角形の描画
        Color cc=c;
        if(i==selected)cc=Color.red;
        ((MySquare)sqrList.get(i)).draw(g,cc);
    }
    if(tmp!=null) {//現在作成中の四角形
        tmp.draw(g,Color.blue);
    }
    g.setColor(c);
    String coor="("+String.valueOf(present.x)+","
        +String.valueOf(present.y)+")";
    g.drawString(coor,10,10);
}

/** This method is called from within the init() method to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
private void initComponents() {//GEN-BEGIN:initComponents

} //GEN-END:initComponents

```

今回は、前回のプログラムに改良を加えながら、キーボードからの入力も使って矩形を操作するプログラムを作りましょう。

キーボード操作も、マウス操作と同様にイベントとして処理されます。マウスの場合と同様に `KeyListener` というインターフェイスを implements した後、

```
addKeyListener(this);
```

をつかって、インターフェイスを追加します。

また、X ウィンドウや Windows のような GUI システムでは、キーボードからの入力をどのウィンドウが受けるかの制御が行われています。これを `focus` と呼びます。作成するアプレットが `focus` を受け取るように設定します。

```
setFocusable(true);
```

キーボードイベントを扱う関数は 3 種類です。マウスと同様に、全て定義しなくてははいけません。引数は全て `KeyEvent` 型です。

プログラム 11.1.1 では、キー `x` が押されると、選択されている矩形が消去されるようになっています。押されたキーは `KeyEvent.getKeyChar()` で調べることができます。

Program 11.1.3 DrawSquare.java つづき (2)

```
public void mouseDragged(MouseEvent e) { // ボタンを押しながら移動した時
    tmp.set(e.getX(), e.getY());
    repaint();
}

public void mousePressed(MouseEvent e) { // ボタンを押した時
    tmp=new MySquare(e.getX(), e.getY());
}

public void mouseReleased(MouseEvent e) { // ボタンを離した時
    tmp.set(e.getX(), e.getY());
    if(tmp.w!=0 || tmp.h!=0){ // 大きさゼロの矩形は破棄する
        sqrList.add(tmp);
    }
    tmp=null;
    repaint();
}

// 使わないマウスの動作も定義しなくてはならない
public void mouseMoved(MouseEvent e) {
    present.setLocation(e.getX(), e.getY());
    repaint();
}

public void mouseClicked(MouseEvent e) {
    present.setLocation(e.getX(), e.getY());
    if(sqrList==null) return;
    int maxsqr=sqrList.size(); // 保存されている矩形の数
    for(int i=0; i<maxsqr; i++){ // マウスが入っている矩形を探す
        if(((MySquare)sqrList.get(i)).isInside(present))
            selected=i;
    }
    repaint();
}

public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
```

11.2 マウスイベント処理の改良

今回のプログラムでは、マウスの取り扱いを少し追加しました。

まず、`mouseMoved` を使って、マウスが動くたびに、その位置を `present` という位置に保存しています。この位置は、`paint` メソッドの中で、アプレットの左上に表示されています。

前回のプログラムでは、マウスを単にクリックする、つまり押した直後に放すと、大きさがゼロの矩形が定義されていました。今回のプログラムでは、`mouseReleases` の中で、大きさゼロの矩形を破棄するように改良しました。

更に、マウスをクリックすると、その位置が既に描画した矩形の内部である場合、その矩形を選択することができます。選択された矩形は赤で表示されます。

Program 11.1.4 DrawSquare.java つづき (3)

```

public void keyPressed(KeyEvent e) {
    if(e.getKeyChar()=='x'){
        if(selected>=0){
            sqrList.remove(selected);
            selected=-1;
        }
    }
    repaint();
}

public void keyReleased(KeyEvent e) {}

public void keyTyped(KeyEvent e) {}

// Variables declaration - do not modify//GEN-BEGIN:variables
// End of variables declaration//GEN-END:variables
}

```

表 11.1: キーボードイベント

関数名	内容
keyPressed	この部分でキーが押されると呼び出される
keyReleases	この部分でキーが放されると呼び出される
keyTyped	この部分でキーがタイプされると呼び出される

11.3 矩形をリストに保存する

前回のプログラムでは、矩形は配列に保存されていました。配列を使う場合、その大きさを指定しなくてはなりません。今回は、リストに保存し、プログラム作成時に矩形の数の上限を設けないことにします。

リスト (list) は、要素が一列に並んだものです。Java には標準でリストを作るクラス `java.util.Vector` があります。今回はこれを用います。

```
private java.util.Vector sqrList=null;
```

`java.util.Vector` に保存できるものは、`Object` です。`Object` は Java のすべてのクラスに対応した最上位のクラスです。`Vector` への要素の追加は `add` メソッドで行います。

`Vector` から要素を得るには `get` メソッドを使います。戻り値が `Object` であることに注意します。今回は戻り値を `MySquare` へキャストすることで、矩形の処理を行っています。

```
MySquare ms=(MySquare)sqrList.get(int)
```

`Vector` から要素を削除するには、`remove` を使います。

第12章 再帰的関数

12.1 Sierpinski Gasket

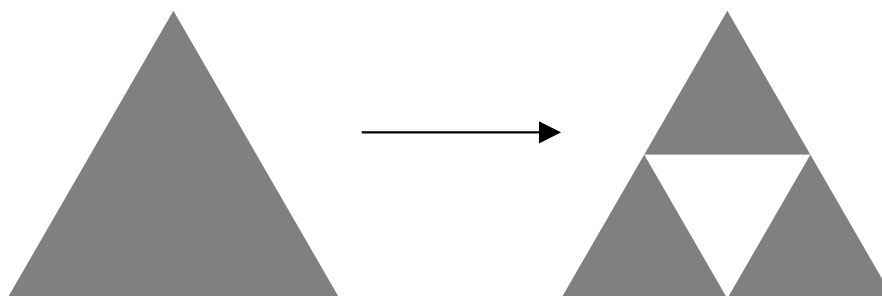


図 12.1.1: Sierpinski Gasket の一回の操作

Java アプレットを使った作図の最後に、フラクタル (fractal) 図形を描くプログラムを作りましょう。フラクタル図形とは、雪の結晶、海岸線の形、葉脈などのように、見る大きさを変えてみても同じような構造が繰り返し現れるような図形です。自然界には、様々なところにフラクタル図形を見出すことができます。自然の様々なところでフラクタル図形が現れることと、その生成機構の説明は、20 世紀の終わりから、様々な研究分野で注目されています。

Sierpinski Gasket は、簡単なフラクタル図形の一つです。それは以下のようにして定義されています。まず、一辺の長さが L である、正三角形を考えます。それを適当な色で塗りつぶしましょう。そこから、各辺の中点を結んでできる中心の三角形をくり抜きます。残った 3 個の一辺の長さ $L/2$ の各三角形について、上と同様に、中点を結ぶ三角形をくり抜く操作を行います。これを無限回繰り返した図形を Sierpinski Gasket と呼びます。毎回、普通の意味での面積が $3/4$ になりますから、無限回の操作で面積はゼロになってしまいます。しかし、残っている図形があります。つまり、通常の中が塗りつぶされているのではなく、すかすかに穴のあいた変な図形ができあがります。

任意の自然数 n に対して、上記の図形から一辺の長さ $2^{-n}L$ の三角形を取り出します。すると、真中に一辺の長さ $2^{-n-1}L$ の穴があいています。また、3 つの一辺の長さ $2^{-n-1}L$ にもそれぞれ一辺の長さ $2^{-n-2}L$ の穴があります。このように、見る長さを変えてみても、同じ構造が繰り返し現れます。これをスケール不変性 (scale invariance) と呼びます。

このような同じことの繰り返しをプログラムする方法の一つが再帰的関数の利用です。簡単な場合として階乗を考えましょう。自然数 n の階乗は

$$n! = n \times (n - 1) \times \cdots \times 2 \times 1$$

Program 12.1.1 Sierpinski.java

```

/*
 * Sierpinski.java
 *
 * Created on 2005/01/11, 13:25
 */
import java.awt.*;
import java.awt.event.*;
/**
 *
 * @author tadaki
 */
public class Sierpinski extends javax.swing.JPanel{
    private Image image=null;
    private int t=0;
    private double l=250.;
    private java.awt.geom.Point2D.Double points[];
    private Dimension imageSize;
    /** Creates new form Sierpinski */
    public Sierpinski() {
        initComponents();
        imageSize=new Dimension(300,300);
    }

    public void init_image(){
        //イメージの初期化
        image=this.createImage(imageSize.width,imageSize.height);
        Graphics g=image.getGraphics();
        g.setColor(getBackground());
        g.fillRect(0,0,imageSize.width,imageSize.height);
        //一番外側の正三角形
        points = new java.awt.geom.Point2D.Double[3];
        points[0]=new java.awt.geom.Point2D.Double(20.,20.);
        points[1]=new java.awt.geom.Point2D.Double(20+1,20.);
        points[2]=new java.awt.geom.Point2D.Double(20+1/2.,20.+0.5*1*Math.sqrt(3.));
    }
}

```

のようにも書けますが、

$$n! = n \times (n - 1)!$$

のように、階乗自身を使って定義することができます。ただし、

$$0! = 1$$

という停止条件を付けなければなりません。

このように関数が、その関数自身で定義されている場合に、再帰的関数と呼びます。再帰的関数を定義する場合には、必ず停止条件が必要であることに注意します。停止条件が無い場合、プログラムは暴走し、コンピュータの資源を使い果たします。

12.2 Sierpinski クラス

Sierpinski Gasket を描くクラスのプログラムをプログラム 12.1.1 に示します。これまでのプログラムでは、`paint` メソッドの中で、作図を直接行ってきました。この方法では、ウィンドウの重ね順が変更になった場合など、再描画が必要になる度に作図を行われていました。今回のプログラムでは、表示すべき図を `image` として、生成し、`paint` 内では、単にその `image` を表示するだけにして、作図が一度だけ行われるようにしています。

図の初期化は `init_image` で行われます。 `image` が背景色で塗られた後、Sierpinski Gasket の一番外側の正三角形が定義されます。この段階では、作図は行われていません。

実際の作図は `update_state` で行われます。繰り返しの最大値は `t` です。実際の再帰関数は `update_state_sub` に記述されています。引数は、外側の正三角形の頂点と、繰り返し回数です。繰り返し回数が最大値未満ならば、内部の三つの三角形に対して、繰り返し回数を一つ増やして再帰呼び出しが行われます。繰り返し回数が最大値と一致すると、現在の正三角形を塗りつぶします。正三角形を描くメソッドは `drawTriangle` です。

メインのアプレットのプログラムをプログラム 12.2.1 に示します。繰り返しの上限が `JComboBox` を使った一覧から選択できるようになっています。繰り返しの上限を選択するたびに、新しい図形が描かれます。

Program 12.1.2 Sierpinski.java 続き

```
public void update_state(){
    init_image();
    update_state_sub(points,0);
    repaint();
}

private void update_state_sub(java.awt.geom.Point2D.Double p[],int n){
    java.awt.geom.Point2D.Double pp[];
    pp=new java.awt.geom.Point2D.Double[3];
    if(n<t){
        //内部の3個の正三角形を再帰的に呼ぶ
        pp[0]=new java.awt.geom.Point2D.Double(p[0].x,p[0].y);
        pp[1]=new java.awt.geom.Point2D.Double((p[0].x+p[1].x)/2,
                                                (p[0].y+p[1].y)/2);
        pp[2]=new java.awt.geom.Point2D.Double((p[0].x+p[2].x)/2,
                                                (p[0].y+p[2].y)/2);
        update_state_sub(pp,n+1);

        pp[0]=new java.awt.geom.Point2D.Double((p[0].x+p[1].x)/2,
                                                (p[0].y+p[1].y)/2);
        pp[1]=new java.awt.geom.Point2D.Double(p[1].x,p[1].y);
        pp[2]=new java.awt.geom.Point2D.Double((p[1].x+p[2].x)/2,
                                                (p[1].y+p[2].y)/2);
        update_state_sub(pp,n+1);

        pp[0]=new java.awt.geom.Point2D.Double((p[0].x+p[2].x)/2,
                                                (p[0].y+p[2].y)/2);
        pp[1]=new java.awt.geom.Point2D.Double((p[1].x+p[2].x)/2,
                                                (p[1].y+p[2].y)/2);
        pp[2]=new java.awt.geom.Point2D.Double(p[2].x,p[2].y);
        update_state_sub(pp,n+1);
    } else {
        //実際に三角形を描く
        Graphics g=image.getGraphics();
        drawTriangle(getForeground(),(java.awt.Graphics2D)g,p);
    }
}
```

Program 12.1.3 Sierpinski.java 続き

```
public void setInit(int t){this.t=t;update_state();}

public void paint(Graphics g){
    super.paint(g);
    if(image==null)return;
    g.drawImage(image,0,0,imageSize.width,imageSize.height,this);
}

private void drawTriangle(Color cl,java.awt.Graphics2D g,
java.awt.geom.Point2D.Double p[]){
    g.setColor(cl);
    java.awt.Polygon polygon=new java.awt.Polygon();
    for(int i=0;i<p.length;i++){
        polygon.addPoint((int)p[i].x,(int)p[i].y);
    }
    g.fillPolygon(polygon);
}

/** This method is called from within the constructor to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
private void initComponents() { //GEN-BEGIN: initComponents

    setLayout(new java.awt.BorderLayout());

    setBackground(new java.awt.Color(255, 255, 204));
    setForeground(new java.awt.Color(255, 0, 0));
} //GEN-END: initComponents

// Variables declaration - do not modify //GEN-BEGIN: variables
// End of variables declaration //GEN-END: variables
}
```

Program 12.2.1 SierpinskiGasket.java

```
/*
 * SierpinskiGasket.java
 *
 * Created on 2005/01/11, 13:23
 */

/**
 *
 * @author tadaki
 */
public class SierpinskiGasket extends javax.swing.JApplet {
    private Sierpinski sierpinski=null;
    /** Initializes the applet SierpinskiGasket */
    public void init() {
        initComponents();
        sierpinski=new Sierpinski();
        sierpinski.setPreferredSize(new java.awt.Dimension(300,300));
        sierpinski.setVisible(true);
        getContentPane().add(sierpinski,java.awt.BorderLayout.CENTER);
        for(int i=0;i<10;i++)
            repeat.addItem(String.valueOf(i));
        sierpinski.setInit(0);
    }
}
```

Program 12.2.2 SierpinskiGasket.java 続き

```
/** This method is called from within the init() method to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
private void initComponents() { //GEN-BEGIN: initComponents
    buttons = new javax.swing.JPanel();
    repeat = new javax.swing.JComboBox();

    buttons.setBackground(new java.awt.Color(204, 255, 204));
    repeat.setBackground(new java.awt.Color(0, 204, 204));
    repeat.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            repeatActionPerformed(evt);
        }
    });

    buttons.add(repeat);

    getContentPane().add(buttons, java.awt.BorderLayout.NORTH);

} //GEN-END: initComponents

private void repeatActionPerformed(java.awt.event.ActionEvent evt) {
//GEN-FIRST:event_repeatActionPerformed
    int t=repeat.getSelectedIndex();
    sierpinski.setInit(t);
    repaint();
} //GEN-LAST:event_repeatActionPerformed

// Variables declaration - do not modify //GEN-BEGIN:variables
private javax.swing.JPanel buttons;
private javax.swing.JComboBox repeat;
// End of variables declaration //GEN-END:variables
}
```

第13章 再帰的関数:その2

13.1 木を描く

もう一つ、簡単な再帰的操作で出て来るフラクタル図形を見ましょう。木の形は、幹から枝が分かれ、枝が再び枝に分かれ、という枝分かれを繰り返して出来ています。それをモデル化しましょう。

ある長さ l の枝が水平線から角度 θ で立っていたとしましょう。その先端から、長さ fl ($f < 1$) で角度がそれぞれ $\theta \pm \sigma$ の枝が生えることにします。これを繰り返します (図 13.1.1)。

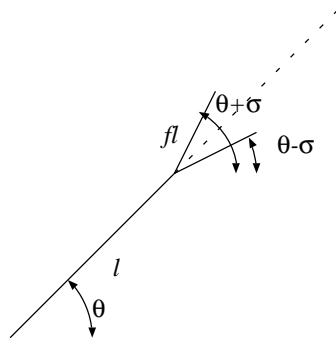


図 13.1.1: 枝の生成規則

木を描くクラス `Tree` をプログラム 13.1.1 に示します。Sierpinski Gasket の場合と同様に `image` を作って、`paint` メソッド内では `image` を表示するだけにしています。

初期化のメソッド `init_image` 内では、木の根 `root` から生えた最初の枝の先端 `top` までの座標が定義されています。座標が保持されているのは、`java.awt.geom.Point2D.Double` というクラスです。

メソッド `update_sub` で再帰的に枝が二本に分かれていきます。引数は繰り返し回数と、枝の根の座標 `r` と先端の座標 `t` です。根と先端の座標から、その長さ `l` と角度 `angle` が計算されます。それを元に次の二本の枝の長さや角度を計算し、再帰的にメソッド `update_sub` を呼び出します。実際に線を描く際には、Y 軸の上下が逆であることに注意します。

Program 13.1.1 Tree.java

```
/*
 * Tree.java
 *
 * Created on 2005/01/11, 15:57
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class Tree extends javax.swing.JPanel {
    private int n=0;
    private final int nmax=15;
    private Image image=null;
    private Dimension imageSize;
    double sigma=0.5;
    double f=0.6;
    private java.awt.geom.Point2D.Double root,top;

    /** Creates new form Tree */
    public Tree() {
        initComponents();
        imageSize=new Dimension(300,300);
        root=new java.awt.geom.Point2D.Double(150,20);
        top =new java.awt.geom.Point2D.Double(150,120);
    }

    private void init_image(){
        //イメージの初期化
        image=this.createImage(imageSize.width,imageSize.height);
        Graphics2D g=(Graphics2D)image.getGraphics();
        g.setColor(getBackground());
        g.fillRect(0,0,imageSize.width,imageSize.height);
    }

    public void update_state(){
        if(n==0)init_image();
        update_sub(0,root,top);
        n++;
        n=n%nmax;
    }
}
```

Program 13.1.2 Tree.java 続き

```

private void update_sub(int tt,java.awt.geom.Point2D.Double r,
                        java.awt.geom.Point2D.Double t){
    //再帰的に枝を伸ばす
    if(tt<n){
        double l=r.distance(t);
        double angle=Math.PI/2.;
        if(Math.abs(t.x-r.x)>0){
            angle=Math.atan((t.y-r.y)/(t.x-r.x));
            if((t.x-r.x<0))angle+=Math.PI;
        }
        java.awt.geom.Point2D.Double np=
            new java.awt.geom.Point2D.Double(
                t.x+f*l*Math.cos(angle-sigma),
                t.y+f*l*Math.sin(angle-sigma)
            );
        update_sub(tt+1,t,np);
        java.awt.geom.Point2D.Double np2=
            new java.awt.geom.Point2D.Double(
                t.x+f*l*Math.cos(angle+sigma),
                t.y+f*l*Math.sin(angle+sigma)
            );
        update_sub(tt+1,t,np2);
    }
    //すでに生成した枝を描く
    Graphics2D g=(Graphics2D)image.getGraphics();
    g.translate(0,imageSize.height);
    java.awt.geom.Line2D.Double line=
        new java.awt.geom.Line2D.Double(r.x,-r.y,t.x,-t.y);
    g.draw(line);
}

public void paint(Graphics g){
    super.paint(g);
    if(image==null)return;
    g.drawImage(image,0,0,imageSize.width,imageSize.height,this);
}

/** This method is called from within the constructor to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
private void initComponents() { //GEN-BEGIN: initComponents

    setLayout(new java.awt.BorderLayout());

} //GEN-END: initComponents

// Variables declaration - do not modify //GEN-BEGIN: variables
// End of variables declaration //GEN-END: variables
}

```

Program 13.1.3 DrawTree.java

```
/*
 * DrawTree.java
 *
 * Created on 2005/01/11, 15:49
 */

/**
 *
 * @author tadaki
 */
public class DrawTree extends javax.swing.JApplet
implements java.awt.event.MouseListener{
    private Tree tree;
    /** Initializes the applet DrawTree */
    public void init() {
        initComponents();
        tree=new Tree();
        tree.setPreferredSize(new java.awt.Dimension(300,300));
        tree.setVisible(true);
        getContentPane().add(tree,java.awt.BorderLayout.CENTER);
        addMouseListener(this);
    }

    /** This method is called from within the init() method to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    private void initComponents() { //GEN-BEGIN:initComponents

        setBackground(new java.awt.Color(204, 255, 255));
        setForeground(new java.awt.Color(0, 0, 255));
    } //GEN-END:initComponents

    public void mouseClicked(java.awt.event.MouseEvent e) {
        tree.update_state();
        repaint();
    }

    public void mouseEntered(java.awt.event.MouseEvent e) {}

    public void mouseExited(java.awt.event.MouseEvent e) {}

    public void mousePressed(java.awt.event.MouseEvent e) {}

    public void mouseReleased(java.awt.event.MouseEvent e) {}

    // Variables declaration - do not modify //GEN-BEGIN:variables
    // End of variables declaration //GEN-END:variables
}
```

第14章 階層化されたグラフィカルユーザーインターフェイス

14.1 スライダー

Program 14.1.1 SliderSample.java

```
/*
 * SliderSample.java
 *
 * Created on 2005/01/12, 12:38
 */

/**
 *
 * @author tadaki
 */
public class SliderSample extends javax.swing.JApplet
implements javax.swing.event.ChangeListener{

    /** Initializes the applet SliderSample */
    public void init() {
        initComponents();
        //slider の変換のイベントを登録する
        slider.addChangeListener(this);
    }

    private void initComponents() { //GEN-BEGIN:initComponents
        label = new javax.swing.JLabel();
        slider = new javax.swing.JSlider();

        label.setBackground(new java.awt.Color(204, 255, 204));
        label.setOpaque(true);
        getContentPane().add(label, java.awt.BorderLayout.CENTER);

        slider.setBackground(new java.awt.Color(204, 255, 255));
        slider.setMaximum(255);
        getContentPane().add(slider, java.awt.BorderLayout.SOUTH);

    } //GEN-END:initComponents
```

GUI (Graphical User Interface) の一つとして、スライダーを考えましょう。これは、マウスでスライドバーを操作することで、数値を入力するインターフェイスです。例を図 14.1.1 に示しま

す。スライダーで選択した数値が表示されます。

Program 14.1.2 SliderSample.java つづき

```
public void stateChanged(javax.swing.event.ChangeEvent e) {
    label.setText(String.valueOf(slider.getValue()));
    repaint();
}

// Variables declaration - do not modify//GEN-BEGIN:variables
private javax.swing.JLabel label;
private javax.swing.JSlider slider;
// End of variables declaration//GEN-END:variables
}
```



図 14.1.1: JSlider の例

プログラム 14.1.1 がそのソースです。JSlider で発生するイベントを拾うためのインターフェイス `javax.swing.event.ChangeListener` がアプレットに追加されています。スライダーのインスタンス `slider`—に対して、インターフェイスが登録されます。

```
slider.addChangeListener(this);
```

スライダーで発生したイベントに対応する動作はメソッド `stateChanged` で定義されます。

```
public void stateChanged(javax.swing.event.ChangeEvent e) {
    label.setText(String.valueOf(slider.getValue()));
    repaint();
}
```

このように、スライダーを使った GUI の構築は

- インターフェイスの登録
- スライダーへのインターフェイス追加
- イベントに対応した動作の記述

の 3 段階で記述することができます。

14.2 RGB カラーを制御する

多くの GUI 環境と同様に、Java でも色は RGB (Red Green Blue) の 3 色で指定することができます。それぞれの成分は 0 から 255 までの整数で指定します。それぞれの成分をスライダーで設定することにします。

0 から 255 までの値を設定できる 3 つのスライダーを作り、イベントが起こるたびにそれぞれからの値を読み出すことで、そうした動きをするアプレットを作ることは可能です。しかし、ここでは、3 つのスライダーを一つのスライダーのように扱うことを考えます。全体の構成を図 14.2.1 に示します。

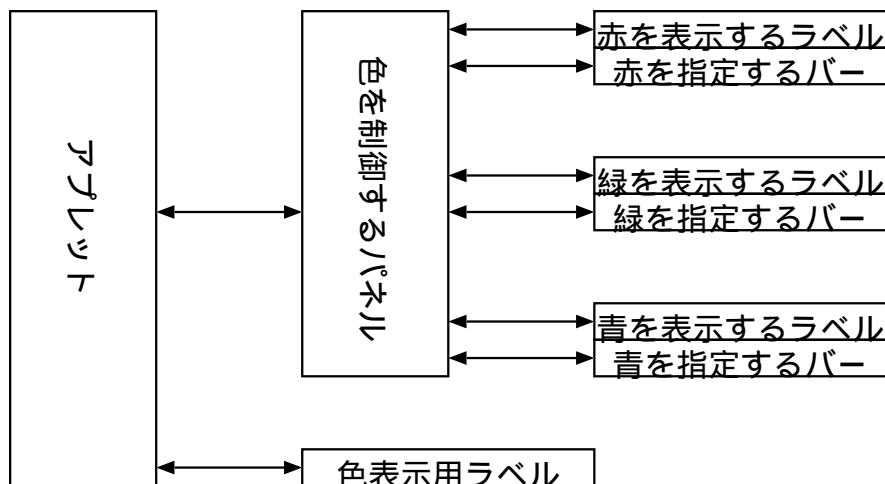


図 14.2.1: RGB カラーを制御するためのシステム概要

まず、3 つのスライダーを使って、色成分を設定するクラス `RGBColor` を作ります。このクラスは、`JPanel` 内に各色成分の名称を表すラベル (`JLabel`) とスライダー (`JSlider`) がそれぞれ 3 つずつ入っています。配置にはグリッドレイアウト (`GridLayout`) が使われています。各スライダーは値の範囲を 0 から 255 とします。これらの設定は、IDE のフォームエディタで行います。

各スライダーで発生するイベント `javax.swing.event.ChangeEvent` に対応したメソッドを `colorStateChanged` に記述します。どの色が変わっても、一つのメソッドが呼ばれます。

このメソッドの中で、それぞれのスライダーから数値を読み (`getValue()`)、色 (`rgb`) と成分を表す文字列 (`rgbvalue`) を生成します。

スライダーには、数値が変更になった際にそのイベントを伝えるためのリスナーを定義することができます。いま作成しているクラスにも同等の機能を持たせることで、3 つのスライダーを一つのように見せましょう。そのために、リスナー (`javax.swing.event.ChangeListener`) を登録することにします。リスナーを保持する変数 `listener` を定義するとともに、登録用メソッド `addChangeListener` を作成します。

スライダーの値が変更になった際には、このリスナーにイベントを伝えます。このイベントに対してどのように反応するかは定義しません。

更に、このクラスから、今の色情報を得るために、二つのメソッド `getrgb()` と `toString()` を用意します。

```
public java.awt.Color getrgb(){return rgb;}  
public String toString(){return rgbvalue;}
```

14.3 RGBColor を使う



図 14.3.1: ColorSample

前節で定義したクラス `RGBColor` は、スライダーのように使うことが可能です。メインのアプレットをプログラム 14.3.1 に示します。また表示例を図 14.3.1 に示します。

クラス `RGBColor` のインスタンス `rgbColor` にリスナーが追加されます。

```
rgbColor.addChangeListener(this);
```

これに対応したメソッドが定義されます。

```
public void stateChanged(javax.swing.event.ChangeEvent e) {  
    colorLabel.setText(rgbColor.toString());  
    colorLabel.setBackground(rgbColor.getrgb());  
    repaint();  
}
```

Program 14.2.1 RGBColor.java

```
/*
 * RGBColor.java
 *
 * Created on 2005/01/12, 11:35
 */

/**
 *
 * @author tadaki
 */
public class RGBColor extends javax.swing.JPanel
{
    private java.awt.Color rgb;
    private String rgbvalue;
    private javax.swing.event.ChangeListener listener;
    /** Creates new form RGBColor */
    public RGBColor() {
        initComponents();
    }

    private void initComponents() { //GEN-BEGIN:initComponents
        rl = new javax.swing.JLabel();
        r = new javax.swing.JSlider();
        gl = new javax.swing.JLabel();
        g = new javax.swing.JSlider();
        bl = new javax.swing.JLabel();
        b = new javax.swing.JSlider();

        setLayout(new java.awt.GridLayout(3, 2));

        setBackground(new java.awt.Color(204, 255, 204));
        rl.setText("Red");
        add(rl);

        r.setBackground(new java.awt.Color(204, 255, 204));
        r.setMaximum(255);
        r.addChangeListener(new javax.swing.event.ChangeListener() {
            public void stateChanged(javax.swing.event.ChangeEvent evt) {
                colorStateChanged(evt);
            }
        });

        add(r);
    }
}
```

Program 14.2.2 RGBColor.java つづき

```
gl.setText("Green");
add(gl);

g.setBackground(new java.awt.Color(204, 255, 204));
g.setMaximum(255);
g.addChangeListener(new javax.swing.event.ChangeListener() {
    public void stateChanged(javax.swing.event.ChangeEvent evt) {
        colorStateChanged(evt);
    }
});

add(g);

bl.setText("Blue");
add(bl);

b.setBackground(new java.awt.Color(204, 255, 204));
b.setMaximum(255);
b.addChangeListener(new javax.swing.event.ChangeListener() {
    public void stateChanged(javax.swing.event.ChangeEvent evt) {
        colorStateChanged(evt);
    }
});

add(b);

} //GEN-END: initComponents

private void colorStateChanged(javax.swing.event.ChangeEvent evt) {
//GEN-FIRST:event_colorStateChanged
    rgb=new java.awt.Color(r.getValue(),g.getValue(),b.getValue());
    rgbvalue="["+String.valueOf(r.getValue())+", "
        +String.valueOf(g.getValue())+", "
        +String.valueOf(b.getValue())+"]";
    if(listener!=null)
        listener.stateChanged(evt);
} //GEN-LAST:event_colorStateChanged

public void addChangeListener(javax.swing.event.ChangeListener listener){
    this.listener=listener;
}


```

Program 14.2.3 RGBColor.java つづき

```
public java.awt.Color getrgb(){return rgb;}
public String toString(){return rgbvalue;}

// Variables declaration - do not modify//GEN-BEGIN:variables
private javax.swing.JSlider b;
private javax.swing.JLabel bl;
private javax.swing.JSlider g;
private javax.swing.JLabel gl;
private javax.swing.JSlider r;
private javax.swing.JLabel rl;
// End of variables declaration//GEN-END:variables

}
```

Program 14.3.1 ColorSample.java

```
/*
 * ColorSample.java
 *
 * Created on 2005/01/12, 11:46
 */

/**
 *
 * @author tadaki
 */
public class ColorSample extends javax.swing.JApplet
implements javax.swing.event.ChangeListener{
    private RGBColor rgbColor;
    /** Initializes the applet ColorSample */
    public void init() {
        initComponents();
        rgbColor=new RGBColor();
        getContentPane().add(rgbColor, java.awt.BorderLayout.NORTH);
        rgbColor.addChangeListener(this);
    }

    private void initComponents() { //GEN-BEGIN:initComponents
        colorLabel = new javax.swing.JLabel();

        colorLabel.setOpaque(true);
        getContentPane().add(colorLabel, java.awt.BorderLayout.CENTER);

    } //GEN-END:initComponents

    public void stateChanged(javax.swing.event.ChangeEvent e) {
        colorLabel.setText(rgbColor.toString());
        colorLabel.setBackground(rgbColor.getrgb());
        repaint();
    }

    // Variables declaration - do not modify //GEN-BEGIN:variables
    private javax.swing.JLabel colorLabel;
    // End of variables declaration //GEN-END:variables
}
```

第15章 まとめ

15.1 セルオートマトン

最後に、簡単なシミュレーションプログラムを作りましょう。ここでは、セルオートマトン (Cellular Automaton) のシミュレーションを扱います。

セルオートマトンは、複雑なパターンの生成を簡単な代数法則で作ることが出来るモデルです。空間的・時間的に離散な規則を記述するため、様々な状況を簡単にモデル化することができます。そのため、自然科学、工学、社会科学など広い分野で利用されています。

ここでは、もっとも簡単なセルオートマトンを扱います。1次元の格子を考えます。各格子点は $\{0, 1, \dots, k-1\}$ の値を取るとします。各時刻で、各格子の値 s_i (i は格子の番号) は、直前の時刻のその格子の値と、距離 r の範囲の格子の値で計算されるとします。

$$s_i(t+1) = F(s_{i-r}(t), s_{i-r+1}(t), \dots, s_i(t), \dots, s_{i+r}(t)) \quad (15.1.1)$$

全ての格子の値は同時に更新されます。

もっとも簡単な場合、 $k=2$ かつ $r=1$ の場合を考えます。つまり、各格子は0または1を値として取り、隣接する格子と自分との値によって次の時刻の値を決定します。このようなセルオートマトンは、S. Wolfram によって基本セルオートマトンと名付けられ、詳細に研究されました。

状態変更規則は、隣接する格子と自分との値の3個の0または1の組に対して、0または1を割り振ることで記述されます。隣接する格子と自分との値の3個の0または1の組を2進数と考えると、8種類の整数に対応します。8個の整数に0または1を割り振るので、 $2^8 = 256$ 通りの割り振り方があります。この256個の割り振り方を規則の番号と呼ぶことにします。

例えば交通渋滞のモデルに使われる184という規則を考えます。184を2進表現します。

$$184 = (10111000)_2 \quad (15.1.2)$$

これは、次のような規則として考えます。

$$\left(\begin{array}{cccccccc} 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\ \hline 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{array} \right) \quad (15.1.3)$$

上の段は直前の隣接格子と自分の状態、下の段は次の時刻での状態を表します。

今回は、10進数の数値を与えると、0と1がでたらめになった初期状態から、系の状態が発展する様子をシミュレーションしましょう。

15.2 CAdef クラス

セルオートマトンのクラスを定義する Java プログラムをプログラム 15.2.1 に示します。このクラスでは、セルオートマトンの動作を定義すると同時に、JPanel に状態を表示します。

クラス定義の始めにある

```
final private int s=2;           //サイトの大きさ
final private int n=300;        //系のサイズ
final private int tmax=200;     //時間の上限
```

は各状態の表示を 2 ピクセルの正方形で表すこと、格子の総数が 300 であること、及び 200 回の更新で一画面とすることを定義しています。キーワード `final` は、これらが定数であることを示しています。なお、系の左右の両端は繋がっていて、輪になっています。このような境界を周期境界 (periodic boundary) と呼びます。

10 進数で与えられた更新規則はメソッド `mkrule` によって、大きさ 8 の配列に 0 と 1 を入れることで保存されます。また、規則の変更は `chrule` によって行われます。

状態の更新は、このクラスのオブジェクトに対して `updatestate` を行うことで駆動されます。`updatestate` では、状態更新の後、`image` へ図を作成します。

15.3 WolframCA クラス

プログラム 15.3.1 に、セルオートマトンのシミュレーションを行うメインの部分を示す。

アプレットの初期化 (`init`) において、スタート及びストップを行うボタン、状態変更規則を行うチョイス及び現在選択されている状態を示すラベルを生成します。

実際にシミュレーションを行う場合、シミュレーションの途中で停止させたり、再開させる必要がある場合があります。その場合には、ここでのプログラムのように、スレッドとしてシミュレーションを別に行うようにします。実行例を図 15.3.1 に示す。

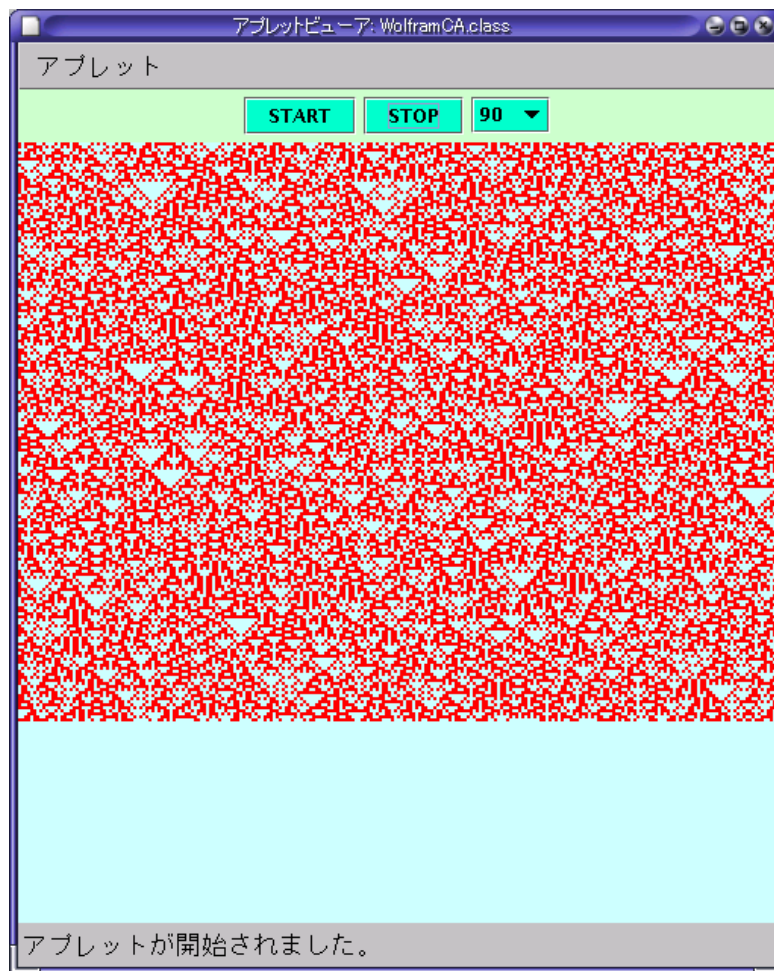


図 15.3.1: Wolfram のセルオートマトン

Program 15.2.1 CAdef.java

```
/*
 * CAdef.java
 *
 * Created on 2005/01/12, 18:22
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class CAdef extends javax.swing.JPanel {
    private int t;
    private int rule[]=new int [8]; //状態更新規則の配列
    final private int s=2;          //サイトの大きさ
    final private int n=300;        //系のサイズ
    final private int tmax=200;     //時間の上限
    private int site[];
    private int sited[];
    private Dimension imageSize;
    private java.awt.Image image=null;

    /** Creates new form CAdef */
    public CAdef() {
        initComponents();
        site=new int [n];
        sited=new int [n];
        imageSize = new Dimension(500,500);
        setPreferredSize(imageSize);
        clear();
    }

    private void cainit(){//初期状態生成
        for(int i=0;i<n;i++){site[i]=(int)(2*Math.random());}
    }

    public void clear(){
        image=this.createImage(imageSize.width,imageSize.height);
        if(image==null){
            System.out.println("null image");return;
        }
        Graphics g=image.getGraphics();
        g.setColor(getBackground());
        g.fillRect(0,0,imageSize.width,imageSize.height);
        t=0;
    }
}
```

Program 15.2.2 CAdef.java の続き

```

public void updatestate(){
    System.out.println(t);
    int in = 4*site[n-1]+2*site[0]+site[1];
    sited[0]=rule[in];
    for(int i=1;i<n-1;i++){
        in = 4*site[i-1]+2*site[i]+site[i+1];
        sited[i]=rule[in];
    }
    in = 4*site[n-2]+2*site[n-1]+site[0];
    sited[n-1]=rule[in];
    for(int i=0;i<n;i++)site[i]=sited[i];
    if(image!=null){
        Graphics g=image.getGraphics();
        g.setColor(getForeground());
        for(int i=0;i<n;i++){
            if(site[i]==1)
                g.fillRect(s*i,s*t,s,s);
        }
        t++;
        t=t%tmax;
        if(t==0)clear();
    }
}

private void mkrule(int r){//状態変更規則生成
    int m;
    for(int i=0;i<8;i++)rule[i]=0;
    int i=0;
    while(r!=0){
        m=r%2;
        rule[i]=m;
        r=r/2; i++;
    }
}

public void paint(Graphics g){
    super.paint(g);
    if(image==null)return;
    g.drawImage(image,0,0,imageSize.width,imageSize.height,this);
}

public void initialize(){t=0;}

public void chrule(int r){
    mkrule(r); t=0; cainit();
}

private void initComponents() { //GEN-BEGIN: initComponents

    setLayout(new java.awt.BorderLayout());

    setBackground(new java.awt.Color(204, 255, 255));
    setForeground(new java.awt.Color(255, 0, 0));
} //GEN-END: initComponents

// Variables declaration - do not modify //GEN-BEGIN: variables
// End of variables declaration //GEN-END: variables
}

```

Program 15.3.1 WolframCA.java

```
/*
 * WolframCA.java
 *
 * Created on 2005/01/12, 18:13
 */
import java.awt.*;
/**
 *
 * @author tadaki
 */
public class WolframCA extends javax.swing.JApplet
implements Runnable{
    private Thread runner;
    private int rr=90;
    private boolean started=false;
    private CAdef ca;
    /** Initializes the applet WolframCA */
    public void init() {
        initComponents();
        ca=new CAdef();
        ca.setVisible(true);
        getContentPane().add(ca, java.awt.BorderLayout.CENTER);
        MkRules();
    }
}
```

Program 15.3.2 WolframCA.java 続き

```

private void initComponents() { //GEN-BEGIN:initComponents
    jPanel1 = new javax.swing.JPanel();
    start = new javax.swing.JButton();
    stop = new javax.swing.JButton();
    ruleChoice = new javax.swing.JComboBox();

    jPanel1.setBackground(new java.awt.Color(204, 255, 204));
    start.setBackground(new java.awt.Color(0, 255, 204));
    start.setText("START");
    start.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            startActionPerformed(evt);
        }
    });

    jPanel1.add(start);

    stop.setBackground(new java.awt.Color(0, 255, 204));
    stop.setText("STOP");
    stop.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            stopActionPerformed(evt);
        }
    });

    jPanel1.add(stop);

    ruleChoice.setBackground(new java.awt.Color(0, 255, 204));
    ruleChoice.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            ruleChoiceActionPerformed(evt);
        }
    });

    jPanel1.add(ruleChoice);

    getContentPane().add(jPanel1, java.awt.BorderLayout.NORTH);
} //GEN-END:initComponents

private void ruleChoiceActionPerformed(java.awt.event.ActionEvent evt) { /
//GEN-FIRST:event_ruleChoiceActionPerformed
    rr=ruleChoice.getSelectedIndex()+1;
} //GEN-LAST:event_ruleChoiceActionPerformed

private void stopActionPerformed(java.awt.event.ActionEvent evt) {}
//GEN-FIRST:event_stopActionPerformed
    started=false;
    stop();
} //GEN-LAST:event_stopActionPerformed

private void startActionPerformed(java.awt.event.ActionEvent evt) {
//GEN-FIRST:event_startActionPerformed
    ca.chrule(rr);
    ca.clear();
    started=true;
    start();
} //GEN-LAST:event_startActionPerformed

```

Program 15.3.3 WolframCA.java 続き 2

```
public void start(){
    if(runner==null){
        runner=new Thread(this);runner.start();
    }
}

public void stop(){
    runner=null;
}

public void run() {
    while(true){
        if(started)ca.updatestate();
        repaint();
        try {Thread.sleep(50);}
        catch (InterruptedException e){}
    }
}

private void MkRules(){//状態変更規則の選択
    for(int i=1;i<256;i++){
        ruleChoice.addItem(String.valueOf(i));
    }
    rr=90;
    ruleChoice.setSelectedIndex(rr-1);
    ca.chrule(rr);
    ca.clear();
}

// Variables declaration - do not modify//GEN-BEGIN:variables
private javax.swing.JPanel jPanel1;
private javax.swing.JComboBox ruleChoice;
private javax.swing.JButton start;
private javax.swing.JButton stop;
// End of variables declaration//GEN-END:variables

}
```
