主題科目:情報の仕組み
J++を使った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15年1月27日

第1章　この講義の目的

WWW(World Wide Web) のホームページは、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基本的な技術になっています。ホームページの中には、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。

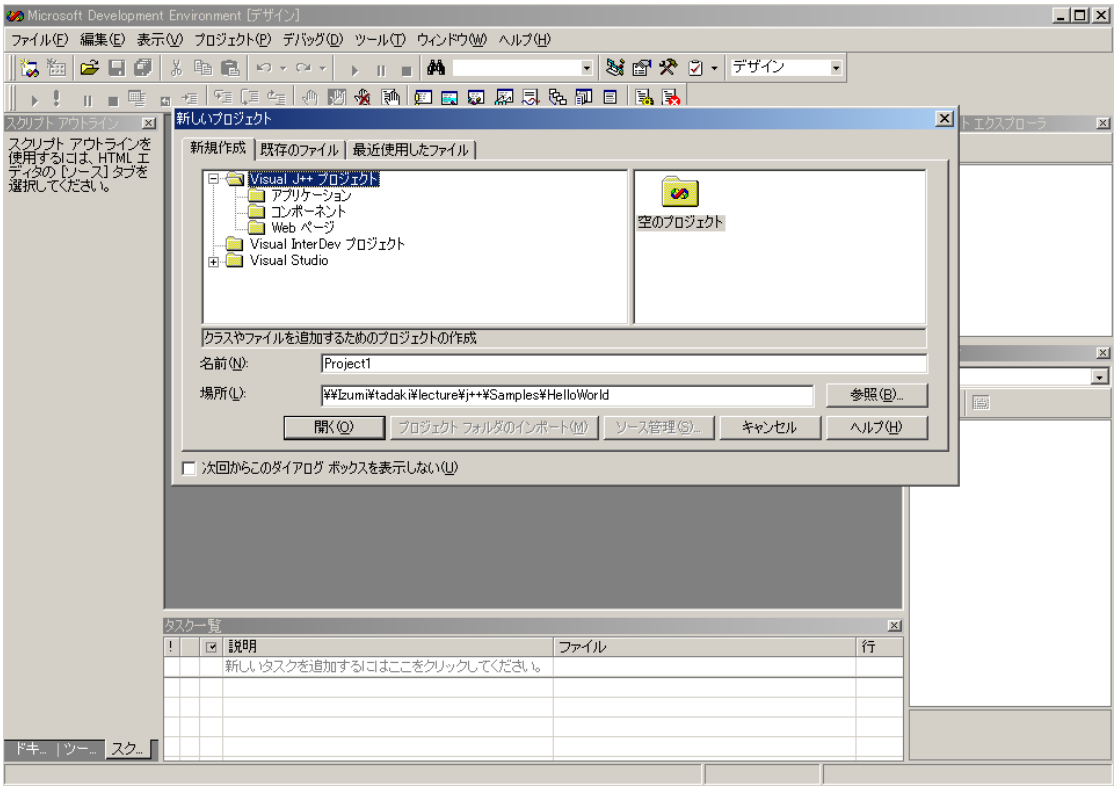
Java はオブジェクト指向 (Object Oriented) プログラミング言語の一つと呼ばれています。オブジェクト指向は、プログラミングの考え方として新しいものの一つです。現実の世界で起こる現象を、対象をデータの集まりとして捕らえ、それらの動作としてプログラムを書こうという考えが基本になっています。Java は他のオブジェクト指向プログラミング言語、例えば C++と共通の側面を多く有しています。Java を学ぶことで、他のプログラミング言語の習得が容易になるでしょう。

本講義では、Java の開発環境の一つである Microsoft Visual Studio 上の J++を使います。

第2章　簡単なプログラム

2.1　Hello World

図 2.1.1: J++起動画面



まず、「スタート」ボタンを押し、「プログラム」の中から、「Microsoft Visual J++」を選択して起動しましょう。あるいは、デスクトップにある「Microsoft Visual J++」のアイコンをダブルクリックして起動しましょう。図 2.1.1 のような画面が現れます。「Visual J++ プロジェクト」で「空のプロジェクト」を選びましょう。

プロジェクト名に「HelloWorld」を指定して、「開く」を押すと「HelloWorld」というプロジェクトが作成されます。右の「プロジェクトエクスプローラ」にプロジェクト名が現れます。

次に、右の「プロジェクトエクスプローラ」に現れたプロジェクト名にマウスを移動し、右ボタ

ンを押します。そこから、「追加」の中の「クラスの追加」を選択します。項目の追加メニューで、「クラス」を選択します。名前として「HelloWorld.java」を指定します。プログラムを編集して、Program 2.1.1 のようにしましょう。

Program 2.1.1 HelloWorld.java

```
import java.awt.*;

public class HelloWorld extends java.applet.Applet
{
        Font f = new Font("TimesRoman",Font.PLAIN,40);

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

}
```

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「ビルド」から「ビルド」を選択することでコンパイルが実行されます。

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、右の「プロジェクトエクスプローラ」の中の「HelloWorld」において「追加」を行います。今度は、「Web ページの追加」を選びます。名前も `HelloWorld.htm` にしましょう。

画面中央に HTML ファイルのソースが表示されます。その画面の下の「ソース」タグをクリックして、HTML ソースを見ましょう。`<BODY>`ダグのすぐしたに、アプレットを読み出す部分を作成しましょう。

Program 2.1.2 HelloWorld.htm

```
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="Microsoft Visual Studio 6.0">
<TITLE></TITLE>
</HEAD>
<BODY>
<APPLET code="HelloWorld.class" width=300 height=100>
</APPLET>
<P> </P>

</BODY>
</HTML>
```

この HTML ファイルが Web ブラウザで表示される様子を見るには、「クイックビュー」タグをクリックします。

2.2 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 java.applet.Applet
```

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 java.applet.Applet` は、既に定義されているクラス `java.applet.Applet` 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。

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。

Program 3.1.1 SimpleArithmetic.java

```
import java.awt.*;

public class Simple extends java.applet.Applet
{
    Font f = new Font("TimesRoman",Font.BOLD,20);

    public void paint(Graphics g){
        g.setFont(f);
        int a=8;
        int b=5;
        int c;

        c=a+b;
        g.drawString(String.valueOf(a)+"+"+String.valueOf(b)+"="+String.valueOf(c),0,20);
        c=a-b;
        g.drawString(String.valueOf(a)+"-"+String.valueOf(b)+"="+String.valueOf(c),0,40);
    }
}
```

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。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。

プログラム中の変数は、コンピュータのメモリ上に割り当てられます。どのような値を割り当てるかを定義するのが型です。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、通常の四則演算 (+、-、*、/) 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、整数同士は除算の余りを計算する演算 (%) も定義されています。

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

3.2　文字と文字列

一つの文字を表す文字型変数と、文字の列からなら文字列は、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';
```

一方、文字列は原始型ではなく、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。

Program 3.2.1 SimpleString.java

```
import java.awt.*;

public class SimpleString extends java.applet.Applet
{
    Font f = new Font("TimesRoman",Font.BOLD,20);

    public void paint(Graphics g){
        g.setFont(f);
        String s="Saga";
        String u="University";

        String c;
        c=s+" "+u;

        g.drawString(c,0,50);
    }
}
```

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。

クラス Graphics では、文字列を描くメソッド drawString が定義されていますが、数値を書くメソッドはありません。そこで、Program3.1.1 では、数値を文字列に変換する文字列クラス String のメソッド valueOf を使っています。

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

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
x++	x=x+1
x--	x=x-1
x+=y	x=x+y
x-=y	x=x-y
x*=y	x=x*y
x/=y	x=x/y
x%=y	x=x%y

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、同じ型のデータを表のように大量に保持する配列 (array) というデータ構造と、繰り返し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。

Program 4.1.1 ArrayAndLoop.java

```
import java.awt.*;
import java.lang.Math;

public class ArrayAndLoop extends java.applet.Applet
{
  Font f = new Font("TimesRoman",Font.BOLD,20);


  public void paint(Graphics g){
    g.setFont(f);

    final int data_num=10;
    final int max=100;
    int[] data=new int[data_num];
    for(int i=0;i<data_num;i++){
      data[i] = (int)(100*Math.random());
    }

    double a=0;
    double d=0.;
    for(int i=0;i<data_num;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),0,20);
    g.drawString("RMS      : "+String.valueOf(d),0,40);
  }
}
```

N 個のデータ $\{x_i\}$ 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。

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

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

ここで和の計算を次のように見直してみます。

$$S_{N-1} = \sum_{i=0}^{N-1} x_i \tag{4.1.3}$$

$$S_n = S_{n-1} + x_n \tag{4.1.4}$$

$$S_0 = x_0 \tag{4.1.5}$$

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ことが分かります。

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てきます。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で何番目かを指定できるます。

配列の宣言は

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。そこで、以下のように大きさ `n` を指定して、オブジェクトを生成します。

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

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};
```

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。`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「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。

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() を呼んでいます。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ないような無限ループにならないような注意が必要です。

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。

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た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。

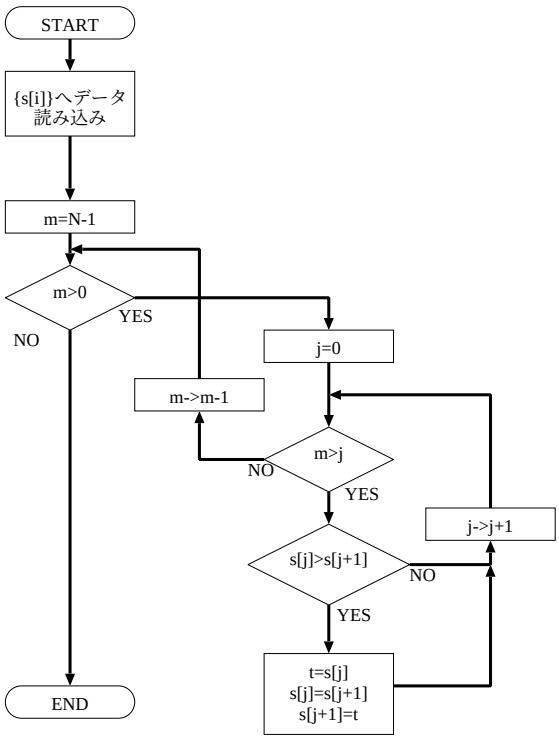


図 5.1.1: 泡立ち法

5.2　泡立ち法による並べ変え

データの並び変えには様々な方法が知られていますが、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りまで一旦行うと、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。

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

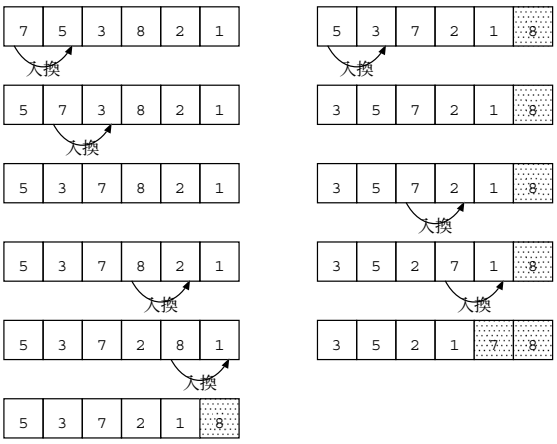


図 5.2.1: 泡立ち法によって配列の内容が変化する様子

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。

5.3 泡立ち法の Java プログラム

Program 5.3.1 Condition.java

```
import java.awt.*;
import java.lang.Math;

public class Condition extends java.applet.Applet
{
  Font f = new Font("TimesRoman",Font.BOLD,20);

  public void paint(Graphics g){
    g.setFont(f);

    final int data_num=10;
    int[] data=new int[data_num];
    initarray(data,data_num);

    String input=mkstr(data,data_num);
    int status=bubble(data,data_num);
    String output=mkstr(data,data_num);

    g.drawString("Input : "+input,0,20);
    g.drawString("Output: "+output,0,40);
  }


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

  public int bubble(int d[],int n){//泡立ち法
    int m=0;
    for(int j=n-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 data[],int n){
    String str="";
    str += String.valueOf(data[0]);
    for(int i=1;i<n;i++){
      str += "," + String.valueOf(data[i]);
    }
    return str;
  }
}
```

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ていますので、順に説明していきましょう。

まず、

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

という部分に注目してください。ここが、泡立ち法の中心部分です。この中に

```
if(d[i] > d[i+1]){
  m++;
  int c=d[i];
  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
>	$>$	<	$>$
>=	$\leq$	<	$\geq$
==	$=$	!=	$\neq$

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。

各場合に整数の番号 k を付けてあるとしましょう。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[],int n){//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) は二つです。一つはデータの入っている配列、二番目は要素数です。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

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。クラスのメソッドの一覧はヘルプから見る事もできます。

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 fillPorygon 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");
```

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=1;
```

のように、形に応じ整数変数 flag の値を指定しています。
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。

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。

Program 6.3.2 SimpleGraphics.java

```
import java.awt.*;

public class SimpleGraphics extends java.applet.Applet
{
  int flag;
  Font f = new Font("TimesRoman",Font.PLAIN,20);
  String s="NULL";
  public void init(){
    s=getParameter("shape");
    if(s.equals("square"))flag=1;
    if(s.equals("3Dsquare"))flag=2;
    if(s.equals("circle"))flag=3;
    if(s.equals("polygon"))flag=4;
  }

  public void paint(Graphics g){
    g.setFont(f);
    g.setColor(Color.black);

    g.drawString(s+":"+String.valueOf(flag),10,140);

    switch(flag){
    case 1://正方形
      g.setColor(Color.green);
      g.fillRect(10,10,50,50);
      break;
    case 2://3 次元正方形
      g.setColor(Color.red);
      g.fill3DRect(10,10,50,50,true);
      break;
    case 3://円
      g.setColor(Color.pink);
      g.fillOval(10,10,50,50);
      break;
    case 4://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;
    }
  }
}
```

第7章　簡単な動画

7.1　描画と再描画

アプレット上に動画を表示することを考えましょう。ここで云う動画とは、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.htm

```
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="Microsoft Visual Studio 6.0">
<TITLE></TITLE>
</HEAD>
<BODY>

<P>ただ今の時刻</P>
<APPLET code=Clock.class width=500 height=50>
</APPLET>
</BODY>
</HTML>
```

現在の時刻を表示するアプレットをプログラム 7.1.2 に示します。

動画を作成する基本となるのは、描画の操作 (paint) に対して、再描画 (repaint() メソッド) を繰り返すことで行います。

前章で見たように、アプレットが含まれるページが表示されるたびに start() メソッドが呼ばれ、アプレットを含むページから他のページに移動する際に stop() メソッドが呼ばれます。従って、start() メソッドの中で、無限ループで repaint() メソッドを呼べば良いように思えます。while ループは、非常に速く回転するので、1 秒ごとに表示されるように sleep しておきます。

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

の部分は、sleep している間に起こった割り込みを処理する部分です。ところが、このアプレットは動作しません。

Program 7.1.2 Clock.java

```
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
public class Clock extends java.applet.Applet
{
  Font f = new Font("TimesRoman", Font.BOLD, 24);
  Date theDate;

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

  public void start() {//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){}
    }
  }

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

7.2　スレッドを使う

前節のプログラム 7.1.2 では、start() メソッドで while ループを回して刻ヶと時刻を再描画しようとします。ところが、paint() メソッドは、start() 後に行われるので、最初の描画ができず、従って再描画もできない状況になっています。

そこで、スレッド (thread) というのを使って解決しましょう。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。前節のプログラムは、処理の道筋が一本しかなく (シングルスレッドと云う)、前の処理が終ると次の処理に移るようになっています。従って、最初の描画が終るまで再描画ができない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、初期化処理では、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 7.2.1 Clock.java

```
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
public class Clock extends java.applet.Applet
  implements Runnable //スレッドを使うことを指示
{
  Font f = new Font("TimesRoman", Font.BOLD, 24);
  Date theDate;
  Thread runner;

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

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

スレッドを使った時刻表示アプレットをプログラム 7.2.1 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ています。

7.3　滑らかな動き

次に、動きを滑らかにする手法を付けましょう。現在のコンピュータの処理速度や描画速度から、相当面倒な描画で無い限り、描画が滑らかでないという印象はありません。ここでは、再描画の手順を理解する助けとして、滑らかな動きを考えましょう。

再描画に際して、次のことが起こります。まず、repaint() メソッドが呼ばれます。repaint() メソッドは、update() メソッドというメソッドを呼びます。update() メソッドは、背景色でアプレット全体を塗りつぶし (fillRect メソッドを使っている)、その後に paint() メソッドを呼びます。この、アプレット全体を背景色で塗る過程が遅いと、動きが滑らかでないように見えます。

プログラム 7.3.1 は、正方形が延び縮みしながら移動するアプレットです。update() メソッドは、単に paint() メソッドを呼ぶだけに書き換えられています。それに対応して、paint() メソッド内で、直前に描いた正方形を背景色で塗りつぶして、新しい正方形の座標等を計算し、前景色で新しい正方形を描いています。

演習 7.1　三角形が回転しながら並行移動するアプレットを作成しなさい。三角関数は java.lang.Math パッケージに含まれています。メソッド Math.sin(double) と Math.cos(double) を使うことができます。ただし、角度はラヂアン (radian) です。また、定数 Math.PI は π の数値を与えます。

Program 7.3.1 MovingSquare.java

```
import java.awt.*;

public class MovingSquare extends java.applet.Applet
  implements Runnable
{
  Thread runner;
  int t=0;
  int sqr[]={10,10,50,50};

  public void init() {//前景色と背景色を設定
    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.stop(); runner = null;
    }
  }

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

  public void paint(Graphics g) {
    g.setColor(getBackground());// 背景色で部分的に塗り潰す
    g.fillRect(sqr[0],sqr[1],sqr[2],sqr[3]);
    g.setColor(getForeground());
    setsqr();
    g.fillRect(sqr[0],sqr[1],sqr[2],sqr[3]);
  }

  public void update(Graphics g){ paint(g);}

  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;
  }
}
```

第8章　クラスを作る

8.1　クラスとは

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 8.1.1 MySquare.java

```
import java.awt.*;
public class MySquare
{
  int x,y;
  int w=50,h;
  Color back,fore;

  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);
  }
}
```

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があり、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。主たる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。

Program 8.1.2 MovingSquare.java

```
import java.awt.*;

public class MovingSquare extends java.applet.Applet
  implements Runnable
{
  Thread runner;
  int t=0;
  MySquare sqr;

  public void init() {//前景色と背景色を設定
    setBackground(Color.cyan);
    setForeground(Color.red);
    sqr = new MySquare(getBackground(),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.stop(); runner = null;
    }
  }

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

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

  public void update(Graphics g){ paint(g);}
}
```

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) とよび、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。

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」と命令するだけにするのです。

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、init メソッドで背景色と前景色が指定されます。その後、paint メソッド内では、単にメソッド 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 8.1.3 MovingSquare.html

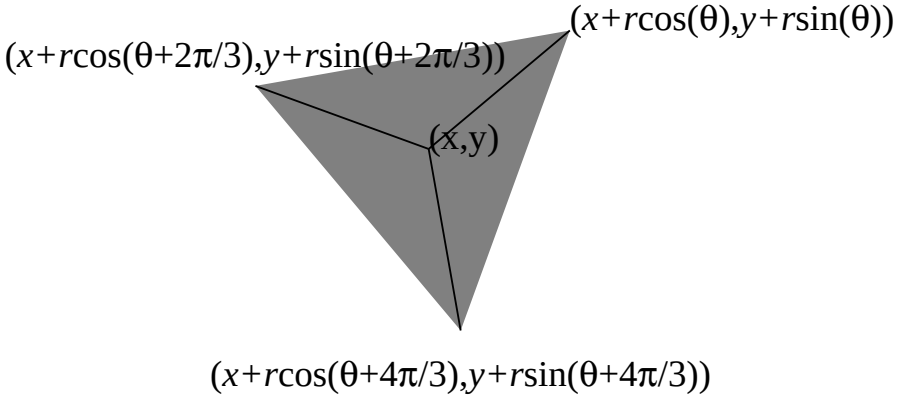
```
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="Microsoft Visual Studio 6.0">
<TITLE></TITLE>
</HEAD>
<BODY>
<APPLET code=MovingSquare.class  width=500 height=70>
</APPLET>
</BODY>
</HTML>
```

プログラム 8.1.3 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

プログラム 8.2.1 は、先週の課題の回転する三角形に対応したクラスです。座標は、図 8.2.1 になります。このクラスを先週と同様の APPLET から呼び出せば、回転しながら移動する三角形を描くことができます。

図 8.2.1: 回転した三角形の座標



Program 8.2.1 MyTriangle.java

```
import java.awt.*;
public class MyTriangle
{
  int x;
  final int y=150;
  final int r=100;
  int xx[]={0,0,0};
  int yy[]={0,0,0};
  Color back,fore;

  MyTriangle(Color b,Color f){
    back=b; fore=f;
  }

  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);// 背景色で部分的に塗り潰す
    g.fillPolygon(xx,yy,3);
  }

  public void draw(Graphics g){
    g.setColor(fore);// 全景色で部分的に塗り潰す
    g.fillPolygon(xx,yy,3);
  }
}
```

第9章　マウスの動きを使う

9.1　マウスイベント

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。

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。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。

Program 9.1.1 DrawSquare.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class DrawSquare extends java.applet.Applet
  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
{
  final int MAXSQR=100;
  //四角形
  MySquare sqr[]= new MySquare[MAXSQR];
  MySquare tmp;
  int currentsqr=0;

  public void init() {
    //マウスの動作を拾うインターフェイスの登録
    addMouseListener(this);
    addMouseMotionListener(this);
    setBackground(Color.yellow);
    setForeground(Color.black);
  }

  public void paint(Graphics g){
    Color c=getForeground();
    for(int i=0;i<currentsqr;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);
    }
  }
```

Program 9.1.2 DrawSquare.java つづき

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

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

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

}
```

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。さらに、init() においてインターフェイスを追加します。

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

インターフェイスを追加したら、マウスの動きに対応する関数を定義していきます。表 9.1 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

```
public void
```

として定義します。また、引数は MouseEvent 型です。

プログラム 9.1.1 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。

表 9.1: 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

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。

9.2 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 9.1 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 9.2.1 MySquare.java

```
import java.awt.*;
//四角形のクラス
public class MySquare
{
  Point start,end;//左上隅と右下隅の座標
  int w,h;//幅と高さ

  MySquare(int x,int y){
    start = new Point(x,y);
  }

  MySquare(final MySquare s){
    start=new Point(s.start);
    end = new Point(s.end);
    w = end.x - start.x;
    h = end.y - start.y;
  }

  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;
  }

  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10章　キーボードからの入力を使う

10.1　キーボードイベント

今回は、前回のプログラムに改良を加えながら、マウスからの入力も使って矩形を操作しましょう。

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。

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いけません。引数は全て KeyEvent 型です。

表 10.1: キーボートイベント

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

プログラム 10.2.1 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。

10.2　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 10.2.1 DrawSquare.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class DrawSquare extends java.applet.Applet
  implements MouseListener, MouseMotionListener, KeyListener
  //マウスとキーの動作を拾うためのインターフェイスの定義
{
  SqrList sqrlist = new SqrList();//四角形のリスト
  MySquare tmp;//作図中の四角形
  Point present=new Point(0,0);
  int selected=-1;//選択された四角形を表す

  public void init() {
    //マウスとキーの動作を拾うインターフェイスの登録
    addMouseListener(this);
    addMouseMotionListener(this);
    addKeyListener(this);
    setBackground(Color.yellow);
    setForeground(Color.black);
  }

  public void paint(Graphics g){
    Color c=getForeground();
    int maxsqr=sqrlist.getSize();
    for(int i=0;i<maxsqr;i++){//既に確定した四角形の描画
      if(i!=selected){
        sqrlist.getItem(i).draw(g,c);
      }
    }
    if(tmp!=null) {//現在作成中の四角形
      tmp.draw(g,Color.blue);
    }
    if(selected>=0){//選択されている四角形
      sqrlist.getItem(selected).draw(g,Color.red);
    }
    //カーソルの位置座標の表示
    g.setColor(getBackground());
    g.fillRect(0,0,100,10);
    g.setColor(getForeground());
    String coor="("+String.valueOf(present.x)+",";
    coor+=String.valueOf(present.y)+")";
    g.drawString(coor,10,10);
  }
```

Program 10.2.2 DrawSquare.java つづき

```
//マウスのイベントを拾う
public void mousePressed(MouseEvent evt){//ボタンを押した時
  tmp=new MySquare(evt.getX(),evt.getY());
}

public void mouseReleased(MouseEvent evt) {//ボタンを離した時
  tmp.set(evt.getX(),evt.getY());
  if(tmp.w!=0 || tmp.h!=0){//大きさゼロの四角形は廃棄する
    sqrlist.addItem(tmp);
  }
  tmp=null;
  repaint();
}

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

public void mouseMoved(MouseEvent e){//マウスの移動
  present.setLocation(e.getX(),e.getY());
  repaint();
}

public void mouseClicked(MouseEvent e){//マウスクリック
  present.setLocation(e.getX(),e.getY());
  int maxsqr=sqrlist.getSize();
  for(int i=0;i<maxsqr;i++){// マウスが入っている四角形を選ぶ
    if(sqrlist.getItem(i).isInside(present)){selected=i;}
  }
  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 mouseEntered(MouseEvent e){;}
public void mouseExited(MouseEvent e){;}
```

Program 10.2.3 DrawSquare.java つづき

```
  public void keyPressed(KeyEvent e){
    if(e.getKeyChar()=='x'){//x キーが押された場合
      if(selected>=0){
        sqrlist.deleteItem(selected);
        selected=-1;
      }
    }
    repaint();
  }

  //使わないキー動作も定義しなくてはならない
  public void keyReleased(KeyEvent e){;}
  public void keyTyped(KeyEvent e){;}
}
```

10.3　矩形をリストに保存する

Program 10.3.1 SqrList.java

```
//四角形のリスト
public class SqrList
{
  Node head,tail;
  int size=0;

  private class Node{//四角形を保持するノードのクラス
    MySquare sqr;
    Node next;

    Node(MySquare s){
    sqr=new MySquare(s); next=null;
    }
  }

  SqrList(){//初期化
    head=null; tail=null;
  }

  public void addItem(MySquare s){//要素追加
    if(head==null){
      head=new Node(s);
      tail=head;
    } else {
      Node newnode=new Node(s);
      tail.next=newnode;
      tail = newnode;
    }
    size++;
  }
```

Program 10.3.2 SqrList.java つづき

```
  public MySquare getItem(int n){//i 番目の要素を返す
    if(head==null)return null;
    if(n>=size)return null;
    int i=0;
    Node node= head;
    while(node!=null &&i<n){
      node=node.next;
      i++;
    }
    return node.sqr;
  }

  public void deleteItem(int n){//i 番目の要素を削除する
    if(head==null)return;
    if(n>=size || n<0)return;
    if(n==0){//先頭を削除
      Node node=head.next;
      head=null;
      head=node;
      size--;
      return;
    }
    int i=0;
    Node node= head;
    while(node!=null &&i<n-1){
      node=node.next;
      i++;
    }
    Node target=node.next;
    Node next=target.next;
    if(target==tail){//終端を削除する場合
      tail=node;
    }
    target=null;
    node.next=next;
    size--;
  }

  public int getSize(){return size;}
}
```

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。コンピュータのメモリ上にリストを作る方法一つは、要素を数珠繋ぎにすることです。図 10.3.1 に概要を示します。要素と次の箱への接続情報を持つ箱 (ノード、node) を定義します。このように繋ぐことで、先頭の要素から、順にリストの要素へアクセスすることができます。

プログラム 10.3.1 は、要素として矩形 (`MySquare`) を保持するリストのクラスを定義していま

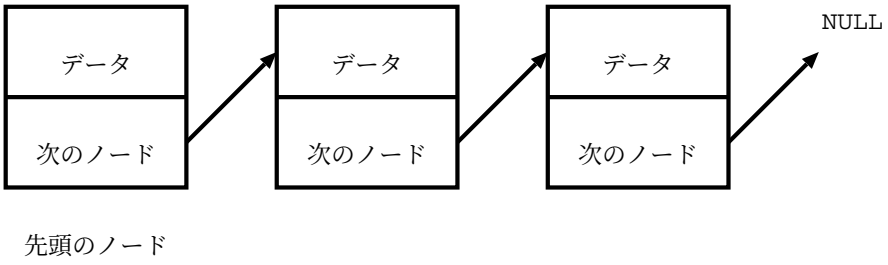


図 10.3.1: リスト

す。要素の追加 (addItem) は、リストの終端に要素を追加します。getItem は指定された位置にある矩形を返します。deleteItem は指定された位置の要素をリストから削除します。

第11章　再帰的関数

11.1　Sierpinski Gasket

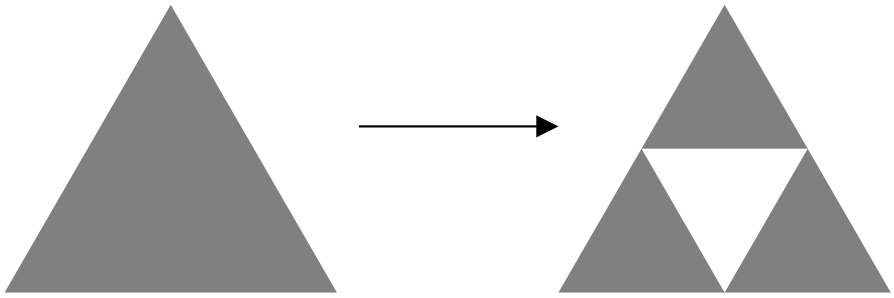


図 11.1.1: Sierpinski Gasket の一回の操作

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。

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、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。

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と呼びます。

では、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

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

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す。

Program 11.1.1 Sierpinski.java

```
//Sierpinki Gasket
import java.awt.*;
import java.lang.Math;

public class Sierpinski
{
  int n,t;
  Graphics g;
  Point original[];
  Color f,b;
  Sierpinski(int nn,Point origin,int size,Graphics gg,Color ff,Color bb){
    f=ff; b=bb;
    g=gg;n=nn;t=0;
    //一番外側の三角形
    original = new Point[3];
    original[0]=new Point(origin.x,origin.y);
    original[1]=new Point(origin.x+size,origin.y);
    original[2]=new Point(origin.x+size/2,origin.y-(int)(size*Math.sin(Math.PI/3.)));
  }

  public void draw(){
    Point A=new Point(original[0]);
    Point B=new Point(original[1]);
    Point C=new Point(original[2]);
    drawdraw(f,A,B,C);
    t++;t=t%n;
    draw_sub(A,B,C,0,t);
  }
```

11.2　Sierpinski クラス

Sierpinski Gasket を描くクラスのプログラムをプログラム 11.1.1 に示します。このクラスは背景色と前景色、三角形の左下の頂点などを変数として定義されます。再帰繰り返しの最大値は `n` です。作図の際にある繰り返し数までが表示されるように、整数 `t` を用意し、作図要求 `draw` が行われる度に、整数 `t` を 1 増やした後に、それを繰り返し数の上限として作図を行います。

再帰的に三角形の中心をくり抜くのは

```
  void draw_sub(Point A,Point B,Point C,int m,int n)
```

で行っています (図 11.1.1)。この関数は、三角形を構成する三つの頂点 A、B 及び C の中心をくり抜き、残った三つの三角形に同じ操作を要請します。ただし、繰り返し回数 m が繰り返しの上限 n 以下の場合です。

Program 11.2.1 Sierpinski.java 続き

```
  //三角形を描く
  void drawdraw(Color c,Point A,Point B,Point C){
    int x[]={0,0,0};
    int y[]={0,0,0};
    x[0]=A.x; x[1]=B.x; x[2]=C.x;
    y[0]=A.y; y[1]=B.y; y[2]=C.y;
    g.setColor(c);
    g.fillPolygon(x,y,3);
  }

  //再帰的に中央の三角形をくり抜く
  void draw_sub(Point A,Point B,Point C,int m,int nn){
    if(m<nn){
      Point D=new Point((int)((A.x+C.x)/2),(int)((A.y+C.y)/2));
      Point E=new Point((int)((A.x+B.x)/2),(int)((A.y+B.y)/2));
      Point F=new Point((int)((B.x+C.x)/2),(int)((B.y+C.y)/2));
      drawdraw(b,D,E,F);// 中央の三角形をくり抜く
      //残りの三角形に同じ操作を行う
      draw_sub(A,E,D,m+1,nn);
      draw_sub(E,B,F,m+1,nn);
      draw_sub(D,F,C,m+1,nn);
    }
  }
}
```

メソッド void drawdraw(Color c,Point A,Point B,Point C) では、三つの頂点 A、B 及び C の中を色 c で塗りつぶします。

クラス Sierpinski を使って作図するメインとクラス SierpinskiGasket をプログラム 11.2.2 に示します。マウスをクリックするたびに、再描画 repaint が呼ばれます。

このプログラムでは、背景色として Color(255,255,255) のように数値で指定しています。三つの数値は、それぞれ赤、青、緑の成分を表し、それぞれの値は 0 から 255 までの整数で与えます。全ての成分が 0 の場合は黒、全ての成分が 255 の場合には白となります。

11.3 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 11.3.1)。

木を描くクラス Tree をプログラム 11.3.1 に示します。メソッド draw_sub で再帰的に枝が二本

Program 11.2.2 SierpinskiGasket.java

```
import java.awt.*;
import java.awt.event.*;

public class SierpinskiGasket extends java.applet.Applet implements MouseListener
{
  int n;
  Sierpinski ss;

  public void init(){
    if(ss==null){//新しい SierpinskiGasket を定義
      n=8;
      Point p=new Point(10,230);
      setForeground(Color.red);
      setBackground(new Color(255,255,255));
      ss = new Sierpinski(n,p,250,getGraphics(),getForeground(),getBackground());
    }
    addMouseListener(this);
  }

  public void paint(Graphics g){ ss.draw();}

  public void mousePressed(MouseEvent e){;}
  public void mouseReleased(MouseEvent e){;}
  public void mouseClicked(MouseEvent e){repaint();}
  public void mouseEntered(MouseEvent e){;}
  public void mouseExited(MouseEvent e){;}
}
```

に分かれていきます。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。

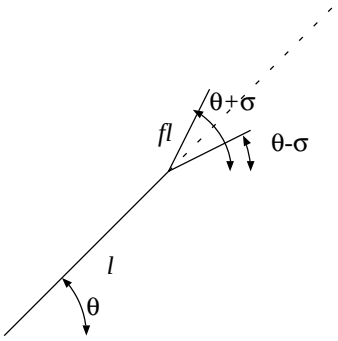


図 11.3.1: 枝の生成規則

Program 11.3.1 Tree.java

```
import java.awt.*;
import java.lang.Math;
public class Tree
{
  int n,t;
  Graphics g;
  Point origin;
  int size;
  Color f,b;

  Tree(int nn,Point o,int s,Graphics gg,Color ff,Color bb){
    f=ff; b=bb; g=gg; n=nn; t=0; size=s;
    origin=new Point(o);
  }

  public void draw(){
    Point root=new Point(0,0);
    Point top = new Point(0,size);
    g.drawLine(origin.x+root.x,origin.y-root.y,origin.x+top.x,origin.y-top.y);
    t++; t=t%n;
    draw_sub(root,top,0,t);
  }

  //再帰的に枝を伸ばす
  private void draw_sub(Point o,Point tp,int m,int nn){
    if(m<nn){
      g.drawLine(origin.x+o.x,origin.y-o.y,origin.x+tp.x,origin.y-tp.y);
      double l=Math.sqrt((tp.x-o.x)*(tp.x-o.x)+(tp.y-o.y)*(tp.y-o.y));
      double angle=Math.PI/2.;
      if(Math.abs(tp.x-o.x)>0.){
        angle=Math.atan((tp.y-o.y)/(tp.x-o.x));
        if(tp.x-o.x<0){
          angle+=Math.PI;
        }
      }
      double da=0.5;
      double ff=0.6;
      Point np = new Point(tp.x+(int)(ff*l*Math.cos(angle-da)),
                           tp.y+(int)(ff*l*Math.sin(angle-da)));
      draw_sub(tp,np,m+1,nn);
      np.setLocation(tp.x+(int)(ff*l*Math.cos(angle+da)),
                     tp.y+(int)(ff*l*Math.sin(angle+da)));
      draw_sub(tp,np,m+1,nn);
    }
  }
}
```

Program 11.3.2 DrawTree.java

```
import java.awt.*;
import java.awt.event.*;

public class FractalTree extends java.applet.Applet implements MouseListener
{
  int n;
  Tree ss;

  public void init(){
    if(ss==null){//新しい SierpinskiGasket を定義
      n=10;
      Point p=new Point(200,300);
      setForeground(Color.red);
      setBackground(new Color(255,255,255));
      ss = new Tree(n,p,100,
          getGraphics(),getForeground(),getBackground());
    }
    addMouseListener(this);
  }

  public void paint(Graphics g){
    ss.draw();
  }

  public void mousePressed(MouseEvent e){;}
  public void mouseReleased(MouseEvent e){;}
  public void mouseClicked(MouseEvent e){
    repaint();
  }
  public void mouseEntered(MouseEvent e){;}
  public void mouseExited(MouseEvent e){;}

}
```

第12章　ボタンなど

12.1　ボタンを作る

これまでアプレットを作成するために、java.awt.*というクラス群を使って来ました。この中には、ボタンなどを使って利用者から情報を得る機能 (UI, User Interface)、ボタンなどの配置を調整する機能、及びボタンなどの動作を記述する機能が含まれています。ここでは、簡単な UI を作成しましょう。

Program 12.1.1 TestButton.java

```
import java.awt.*;
import java.awt.event.*;

public class TestButton extends java.applet.Applet implements ActionListener
{
  Button start, stop;

  public void init(){
    start=new Button("Start"); add(start);
    start.addActionListener(this);
    stop=new Button("Stop"); add(stop);
    stop.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    String cname=e.getActionCommand();
    Graphics g=getGraphics();
    if(cname.equals("Start")){
      g.setColor(getForeground());
    }
    if(cname.equals("Stop")){
      g.setColor(getBackground());
    }
    g.fillRect(0,0,100,100);
  }
}
```

プログラム 12.1.1 は、”Start” と”Stop” というラベルを持つ二つのボタンを生成する例題です。start=new Button("Start") でボタンを生成します。引数はボタンに表示される文字列です。生成したボタンをアプレットに add(start) で追加します。

ボタンの動作を見るには、インターフェイス ActionListener をクラスの定義に追加する必要があります。更に start.addActionListener(this) を使ってインターフェイスを有効にします。

メソッド actionPerformed(ActionEvent e) にボタンが押された際の動作を記述します。ActionEvent.getActionCo
を使うことで、押されたボタンについているラベルを見る事ができます。このプログラムでは、ボタンに応じて矩形を描いています。

12.2　一覧から選ぶ

Program 12.2.1 TestChoice.java

```
import java.awt.*;
import java.awt.event.*;

public class TestChoice extends java.applet.Applet implements ItemListener
{
  ColorList clist[] = {//色一覧
    new ColorList(Color.black,"Black"),
    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")
  };

  Choice colors;
  int ncolor,n;

  public void init(){
    colors = new Choice();
    ncolor=clist.length;
    for(int i=0;i<ncolor;i++){// 色一覧作成
      colors.addItem(clist[i].name);
    }
    add(colors);
    colors.addItemListener(this);
  }
```

次に説明する UI は一覧からの選択です。一覧を表示するには Choice クラスを用います。一覧の各項目は Choice.addItem(String) で行います。

一覧の状態の変化を見るにはインターフェイス ItemListener を用います。変化が起こった際にはメソッド itemStateChanged(ItemEvent) が呼ばれます。

プログラム 12.2.1 では、一覧に色の名前が表示されています。色を選択すると指定されて色で矩形が塗りつぶされます。

Program 12.2.2 TestChoice.java つづき

```
  public void itemStateChanged(ItemEvent e){
    n=colors.getSelectedIndex();
    repaint();
  }

  public void paint(Graphics g){
    g.setColor(clist[n].c);
    g.fillRect(100,100,100,100);
    g.drawString(String.valueOf(n)+clist[n].name,100,220);
  }
}

class ColorList{
  Color c;
  String name;

  ColorList(Color cc,String s){
    c=cc; name=new String(s);
  }
}
```

12.3 スクロールバー

スクロールバーは、画像やテキストをスクロールするだけでなく、一定範囲の整数を入力するのに用いることができます。ここでは、色は赤緑青の要素を数値で入力する例を見ましょう。

スクロールバーは

```
r  = new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,256);
```

のように定義します。最初の引数は方向を指定します。垂直の場合には `Scrollbar.VERTICAL` と指定します。二番目の数値は初期値、三番目の引数は可視部分の量を表しますが今は使いません。四番目と五番目が数値の範囲を表します。

スクロールバーの動作を見るためにはインターフェイス `AdjustmentListener` を追加します。スクロールバーが動くとメソッド `adjustmentValueChanged(AdjustmentEvent e)` が呼ばれます。

プログラム 12.3.1 では、スクロールバーが呼ばれると、三色のスクロールバーから現在の値が `getValue()` メソッドを使って読み取られ、背景色に設定されています。

UI 部品のレイアウトは、あまり細かくできません。プログラム 12.3.1 ではグリッドレイアウトという形を使っています。

```
setLayout(new GridLayout(4,2));
```

は、横に 2 個、縦に 4 個の部品を並べることを指示しています。このレイアウトでは、全ての UI 部品の大きさは同じであると仮定して、アプレットの大きさ一杯に合わせようとします。

Program 12.3.1 ColorSample.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class ColorSample extends java.applet.Applet implements AdjustmentListener
{
  Scrollbar r,g,b;
  Label  rl,gl,bl;
  Label color;

  public void init(){
    setLayout(new GridLayout(4,2));

    rl = new Label("Red"); add(rl);
    r  = new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,256);  add(r);
    r.addAdjustmentListener(this);

    gl = new Label("Green"); add(gl);
    g  = new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,256);  add(g);
    g.addAdjustmentListener(this);

    bl = new Label("Blue"); add(bl);
    b  = new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,256);  add(b);
    b.addAdjustmentListener(this);

    color = new Label("(0,0,0)"); add(color);
  }

  public void adjustmentValueChanged(AdjustmentEvent  e){
    setBackground(new Color(r.getValue(),g.getValue(),b.getValue()));
    String l="("+String.valueOf(r.getValue())+","
                +String.valueOf(g.getValue())+","
                +String.valueOf(b.getValue())+")";
    color.setText(l);
    repaint();
  }
}
```

第13章　階層化されたUI

13.1　はじめに

第 12.3 節で、スクロールバーを動かして、色を変えるアプレットを作成しました。その際、アプレット本体の直下に三つのスクロールバーや色を表示するラベルなどが図に示すように論理的に配置されていました (図 13.1.1)。

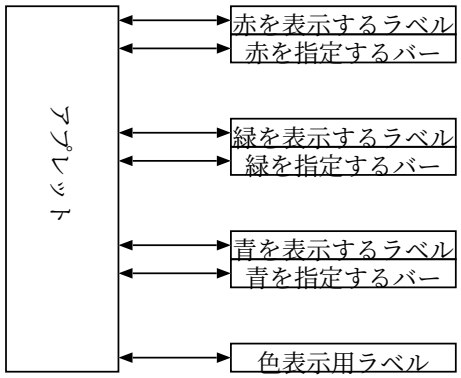


図 13.1.1: 各部品の配置 (変更前)

大規模なアプリケーションを作成する場合、ボタンなどもグループに分けて、階層的に配置するのが、開発効率の上で有効です。例えば、ワードプロセッサを見ると、メニューのグループがあり、その各メニューの中にサブメニューが見えます。これら、利用者から見える構造がプログラムの中にも構造として記述されているのが望ましいでしょう。

そこで、図 13.1.2 のように。色成分を制御するパネルを置き、その下に各色成分の制御部分を置くようにしましょう。

13.2　Panelクラス

Java が提供する UI(ユーザーインターフェイス) には、ボタンなどのように、それ自身で動作などを行うものと、他の UI 部品を内部に持つもの (Container) があります。Panel クラスも Container の一つです。

各色の成分の数値を変更するスクロールバーと各色の名称を表す文字列を表示する label を一まとめにした Panel を作成しましょう。クラス `RGBColor` をクラス `java.awt.Panel` の拡張として

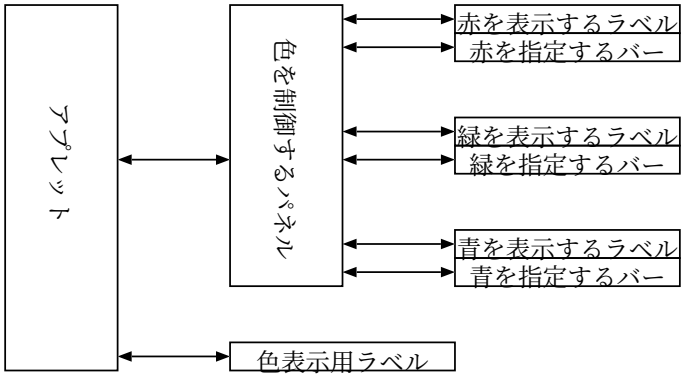


図 13.1.2: 各部品の配置 (変更後)

定義します (プログラム 13.2.1)。

内部に含まれる各色成分を変更するスクロールバーの動きに応じて、色 rgb を変更するために、AdjustmentListener インターフェイスを追加します。

スクロールバーが動くと、メソッド

```
adjustmentValueChanged(AdjustmentEvent e)
```

が呼ばれます。その中で、色の値 rgb を色成分を表す文字列 rgbvalue が設定されています。

問題は、この rgb の値が変更されたことを、この RGBColor クラスを呼んでいるプログラムに教えることです。出来事 (Event) の処理は、一般に次のような形で行われます。まず、出来事が起こったことを知らせるオブジェクトにリスナーを追加します。そのオブジェクトに、出来事が起こった際の動作を記述します。

さて、Panel クラスにも、出来事を処理するリスナーがありますが、ここでの利用には適していません。そこで、adjustmentListener が使えるように拡張し、色 rgb の変化という出来事を扱えるようにします。

まず、メソッド addAdjustmentListener(AdjustmentListener l) を追加します。ここで変数 l は、出来事が起こった際に動作するオブジェクトになります。この変数を ll として保存しておきます。

実際に色成分の変更が起こった場合には、adjustmentValueChanged(AdjustmentEvent e) が呼ばれますから、その中から

```
ll.adjustmentValueChanged(e)
```

で、オブジェクト ll のメソッドを起動することにします。

ここで注意すべきは、色成分が変わったことを検知して、オブジェクト ll がどのような動作をするかは記述していないことです。その理由は、オブジェクト ll は、ここで定義している色を制御するパネルとは独立に定義されるべきものだからです。

Program 13.2.1 RGBColor.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class RGBColor extends java.awt.Panel  implements AdjustmentListener
{
  Scrollbar r,g,b;
  Label  rl,gl,bl;
  Color rgb;
  String rgbvalue;
  AdjustmentListener ll;

  RGBColor(){//コンストラクタ
    setLayout(new GridLayout(3,2));
    //赤
    rl = new Label("Red"); add(rl);
    r  = new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,256);  add(r);
    r.addAdjustmentListener(this);// スクロールバーの動作を聞く
    //緑
    gl = new Label("Green"); add(gl);
    g  = new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,256);  add(g);
    g.addAdjustmentListener(this);// スクロールバーの動作を聞く
    //青
    bl = new Label("Blue"); add(bl);
    b  = new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,256);  add(b);
    b.addAdjustmentListener(this);// スクロールバーの動作を聞く
  }
```

13.3 RGBColor クラスを使う

次に、ここで作った RGBColor クラスを使う例を見ましょう (プログラム 13.3.1)。ColorSample クラス—の中で、RGBColor クラスのオブジェクト rgb と ColorLabel クラスのオブジェクト color が使われています。オブジェクト rgb に対して、先程作成した addAdjustmentListener がオブジェクト color を引数として設定されています。

ColorLabel クラスは Label クラスの拡張です。adjustmentValueChanged(AdjustmentEvent e) 内では、オブジェクト rgb から色要素とそれを表す文字列が得られ、それを表示しています。

コンストラクタ内に現れる super は、このクラスの継承元、ここでは Label クラスを表しています。つまり、Label クラスのコンストラクタを起動した後に、オブジェクト rgb を保存しています。

Program 13.2.2 RGBColor.java つづき

```
  public void adjustmentValueChanged(AdjustmentEvent  e){//スクロールバーが動いた際の処理
    rgb = new Color(r.getValue(),g.getValue(),b.getValue());// 色の定義
    rgbvalue="("+String.valueOf(r.getValue())+","
                +String.valueOf(g.getValue())+","
                +String.valueOf(b.getValue())+")";// 色を表す数値の文字列
    ll.adjustmentValueChanged(e);// このクラスの動作を聞いているオブジェクトの変更
  }

  public Color getrgb(){return rgb;}//色を返す
  public String getrgbvalue(){return rgbvalue;}//色を表す数値の文字列を返す

  //このクラスの動作を聞くオブジェクトの登録
  public void addAdjustmentListener(AdjustmentListener  l){ll=l;}
}
```

Program 13.3.1 ColorSample.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ColorSample extends java.applet.Applet
{
  RGBColor rgb;//色を制御するオブジェクト
  ColorLabel color;//色を表示するオブジェクト

  public void init(){
    setLayout(new GridLayout(2,1));

    rgb = new  RGBColor(); add(rgb);

    color = new ColorLabel("  (  0,  0,  0)  ",rgb); add(color);
    rgb.addAdjustmentListener(color);
  }
}

//色を表示するクラス
class ColorLabel extends java.awt.Label implements AdjustmentListener
{
  RGBColor rgb;

  ColorLabel(String s,RGBColor rgbd){
    super(s);
    rgb=rgbd;
  }

  public void adjustmentValueChanged(AdjustmentEvent  e){
    setBackground(rgb.getrgb());
    setText(rgb.getrgbvalue());
    repaint();
  }
}
```

13.4 別ウィンドウを開く

Program 13.4.1 ColorSample.java(フレーム版)

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ColorSample extends java.applet.Applet
{
  Frame frame;
  RGBColor rgb;//色を制御するオブジェクト
  ColorLabel color;//色を表示するオブジェクト

  public void init(){
    frame = new Frame("RGB");

    rgb = new  RGBColor(); frame.add(rgb);
    frame.setSize(200,100);

    color = new ColorLabel("  (  0,  0,  0)  ",rgb); add(color);
    rgb.addAdjustmentListener(color);
  }

  public void start(){
    frame.setVisible(true);
  }
}

//色を表示するクラス
//省略
```

次に、色制御の部分を別のウィンドウとして表示することを考えましょう。プログラム 12.3.1 では、色制御を Panel としてまとめられていました。Container の一種である Frame は、アプレットウィンドウとは独立した別のウィンドウを開きその中に UI を置くことが出来るクラスです。

プログラム 13.4.1 では、プログラム 12.3.1 の一部を書き換えています。`Frame` クラスのオブジェクト `frame` が生成され、色制御のクラス `RGBColor` のオブジェクト `rgb` は、`frame.add(rgb)` という形で、フレームの中に加えられています。

また、ブラウザがこのアプレットを含むページに戻るたびに色制御の別ウィンドウが出るように、別ウィンドウの表示 `frame.setVisible(true)` がメソッド `start` の中で起動されています。

第14章　まとめ

14.1　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 $\{0, 1, \ldots, k-1\}$ 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。

$$s_i(t+1) = F\left(s_{i-r}(t), s_{i-r+1}(t), \ldots, s_i(t), \ldots, s_{i+r}(t)\right) \tag{14.1.1}$$

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。

$$184 = (10111000)_2 \tag{14.1.2}$$

これは、次のような規則として考えます。

$$\left(\frac{000}{0}, \frac{001}{0}, \frac{010}{0}, \frac{011}{1}, \frac{100}{1}, \frac{101}{1}, \frac{110}{0}, \frac{111}{1}\right) \tag{14.1.3}$$

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。

14.2　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 14.2.1 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、Canvas に状態を表示します。

Program 14.2.1 CAdef.java

```
import java.awt.*;
import java.lang.Math;
import java.util.Date;
public class CAdef extends java.awt.Canvas
{
  int t;
  int rule[]= new int[8];//状態更新規則の配列
  final int s=2;          //サイトの大きさ
  final int n=300;        //系のサイズ
  final int tmax=200;     //時間の上限


  int site[] = new int[n];
  int sited[] = new int[n];

  CAdef(Color fg, Color bg, int rr){
    setSize(500,500);
    setForeground(fg); setBackground(bg);
    mkrule(rr); //整数の状態更新規則を状態更新規則配列へ変換
    cainit();
  }

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

  private void updatestate(){//状態更新
    int in;
    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];
    }
  }
```

クラス定義の始めにある

```
      final int s=2;          //サイトの大きさ
      final int n=300;        //系のサイズ
      final 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。

Program 14.2.2 CAdef.java の続き

```
  private void showstate(int j,Graphics g){//状態表示
    for(int i=0;i<n;i++){
      if(site[i]==1){
        g.fillRect(s*i,s*j,s,s);
      }
    }
  }

  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){
    t++;
    if(t==tmax){initialize();
          g.setColor(getBackground());
          g.fillRect(0,0,n*s,tmax*s);
          g.setColor(getForeground());
    }
    else{
      showstate(t,g);updatestate();
    }
  }

  public void update(Graphics g){paint(g);}

  public void initialize(){t=0;}
  public void clear(){t=tmax-1;}
  public void chrule(int r){
    mkrule(r);  t=0; cainit();
  }
}
```

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 update を行うことで駆動されます。update から paint が呼ばれ、状態の更新 updatestate、及び状態の表示 showstate が行われます。

14.3　WolframCA クラス

プログラム 14.3.1 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。

Program 14.3.1 WolframCA.java

```
// Wolfram の基本セルオートマトン

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class WolframCA extends java.applet.Applet
  implements Runnable, ActionListener, ItemListener
{
  Thread runner;
  CAdef ca;//セルオートマトンクラス

  Button start, stop;
  Label rulelabel;
  Choice rules;
  Frame frame;

  int started=0;
  int rr=90;

  public void init(){//アプレットの初期設定
    start=new Button("Start");add(start);// スタートボタン
    start.addActionListener(this);

    stop=new Button("Stop");add(stop);// ストップボタン
    stop.addActionListener(this);

    MkRules();//規則変更用一覧

    rulelabel= new Label();add(rulelabel);
    rulelabel.setText("Rule #="+String.valueOf(rr));

    frame = new Frame("Cellular Automata");//状態表示用フレーム
    frame.setSize(500,500);
    setBackground(Color.cyan); setForeground(Color.red);
    ca=new CAdef(getForeground(),getBackground(),rr);
    frame.add(ca);
  }
```

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する。状態変更は、フレームとして別ウィンドウに表示する。

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多い。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しなければならない。

ここでは、スタートボタンを押すことで、フレームが現れシミュレーションが始まり、スタートボタンが押されるとシミュレーションが停止するとともにフレームが閉じるようにした。

Program 14.3.2 WolframCA.java 続き

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

  public void stop() {
    if (runner != null){
      runner.stop(); runner = null;
    }
  }

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

  public void paint(Graphics g){
    if(started==1){
      ca.update(ca.getGraphics());
    }
  }
```

ボタンが押された場合の動作は、actionPerformed に記述されている。スタートとストップのいずれのボタンかの判断は、ボタンにつけられている文字列で判断している。

状態変更規則を選択するチョイスの動作は、itemStateChanged に記述されている。

Program 14.3.3 WolframCA.java 続き

```
  public void actionPerformed(ActionEvent e){
    String cname=e.getActionCommand();
    if(cname.equals("Start")){// スタートボタンが押された場合
      runner.resume();
      started=1;
      frame.setVisible(true);
    }
    if(cname.equals("Stop")){// ストップボタンが押された場合
      runner.suspend();
      ca.clear();
      started=0;
      frame.setVisible(false);
    }
  }

  public void itemStateChanged(ItemEvent e){//状態変更規則の変化
      rr=rules.getSelectedIndex()+1;
      ca.chrule(rr);
      rulelabel.setText("Rule #="+String.valueOf(rr));
      runner.suspend();
      ca.clear();
      runner.resume();
      started=1;
      frame.setVisible(true);
  }

  private void MkRules(){//状態変更規則の選択
    rules = new Choice();
    for(int i=1;i<256;i++){
      rules.addItem(String.valueOf(i));
    }
    rules.select(rr-1);
    add(rules);
    rules.addItemListener(this);
  }
}
```