

主題科目:情報の仕組み  
Javaプログラミング(2010)

佐賀大学総合情報基盤センター  
只木進一

平成23年2月23日



## 第1章 この講義の目的

ジュネーブにある素粒子実験の国際研究所 CERN で、実験データを世界中の研究者と共有することを目的として、1990年に WWW(World Wide Web) の最初のサーバプログラムが開発されました。1993年には、現在使用されている WWW ブラウザの原型である Mosaic が NCSA で開発され、WWW はマルチメディアに対応できるデータ公表手段としてたちまち世界中に普及しました。1992年には、日本で最初の WWW サーバが高エネルギー物理学研究所 (現在の高エネルギー加速器研究機構) で稼働しました。1990年代の WWW の普及は、パーソナルコンピュータとインターネットの普及によって、キラーアプリケーションとなりました。つまり、爆発的な需要を生み出しました。

HTML 文書の相互リンク (link) で構成される WWW は、今では、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、dynamic 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、大きく分けて二つの方法があります。一つは、サーバ側で動的に情報を作成しているものです。もう一つは、クライアント側つまり WWW ブラウザ側で情報を生成しているものです。

WWW の技術の中で、Java は重要な役割を果たしています。サーバ側で動的にページを生成する servlet と、クライアント側で動的ページを生成する applet は Java で記述されています。

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ないことです。applet であれば、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。OS のバージョンが変わっても、その違いを JRE が吸収することで、安定的にアプリケーションを動かすことができるという利点もあります。

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。多くのプログラミング言語では、グラフィック用のライブラリが言語とは別に用意されています。また、グラフィック環境は、その動作するコンピュータ環境に大きく影響を受けます。しかし、Java では、グラフィックライブラリが、言語とともに配布されるだけでなく、上述の JRE さへあれば、動作させることができます。

理由の第三は、Java およびその開発環境が無償で配布されていることです。コンピュータを持ち、Java プログラムを書きたい人ならば、だれでも (成果を販売する場合には別です) Java の開発環境を入手し、プログラムを開発することができます。

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。その基本となる考え方は、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ものです。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を使って簡単なプログラムを作りながら、Javaの基本を学びます。また、Javaの開発環境の一つであるNetBeansを使います。

## 第2章 簡単なプログラム

### 2.1 はじめて NetBeans を使う場合

まず、NetBeans を起動します。本講義で使用するのは、「NetBeans IDE 6.9」です。デスクトップ上のアイコンをダブルクリックして起動します。

始めて起動すると、「NetBeans IDE を登録しよう」という画面が出ます。本学の共通 PC にインストールされているソフトウェアですので、各人が登録する必要はありません。「登録しない」を選びます。

図 2.1.1 が表示されます。次回から表示させないように、「起動時に表示」のチェックを外します。

図 2.1.1: NetBeans IDE によるこそ

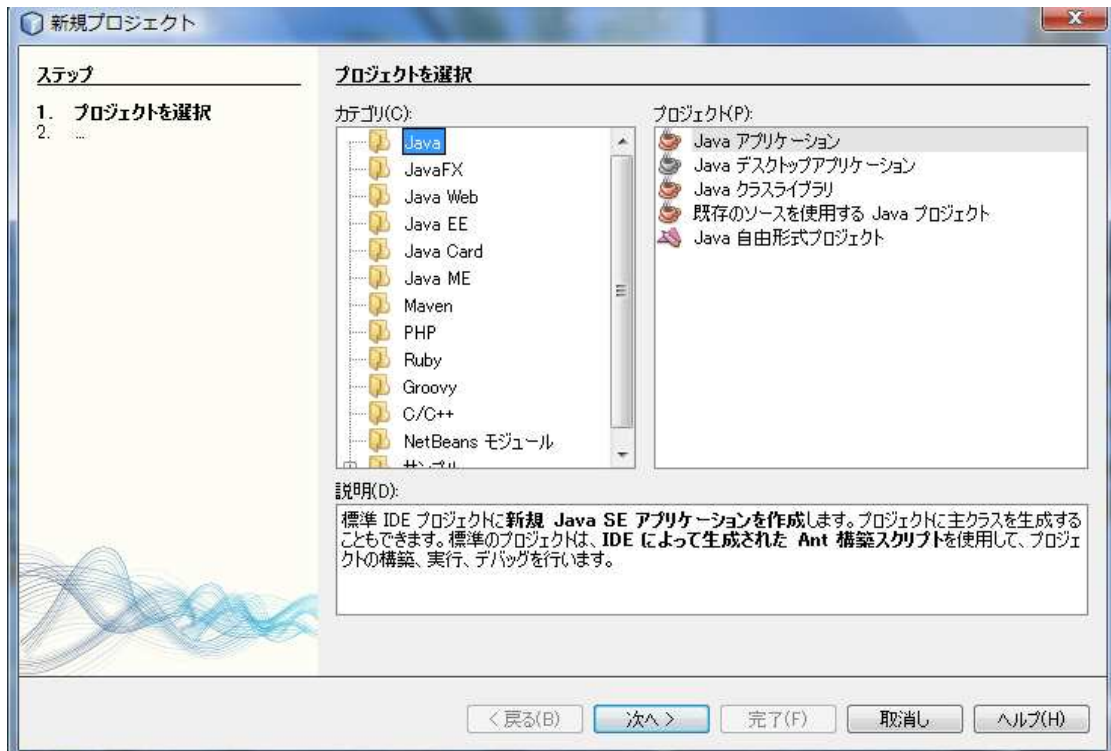


### 2.2 Hello World

NetBeans では、一つのアプリケーションは一つの「プロジェクト」として管理します。一般に、一つのプロジェクトは、複数のプログラムソースファイル、設定ファイルから構成されます。各プロジェクトは、それぞれのフォルダに格納されます。

では、最初のプロジェクト「Hello World」を作成しましょう。新しいプロジェクトを作成するために、メニューの「ファイル」から「新規プロジェクト」を選びます。次に、プロジェクトの種類

図 2.1.2: プロジェクトを作成する



を設定します(図 2.1.2)。ここでは、「Java」「Java アプリケーション」とし、「次へ」を押します。

始めて NetBeans を使う際には、必要な設定ファイルなどを作成するので、少し時間がかかります。

次に、プロジェクトの名前 (HelloWorld) と保存するフォルダを指定します。デフォルトでは、ホームディレクトリの下に **NetBeansProjects** の下にプロジェクト毎のフォルダが作られます。

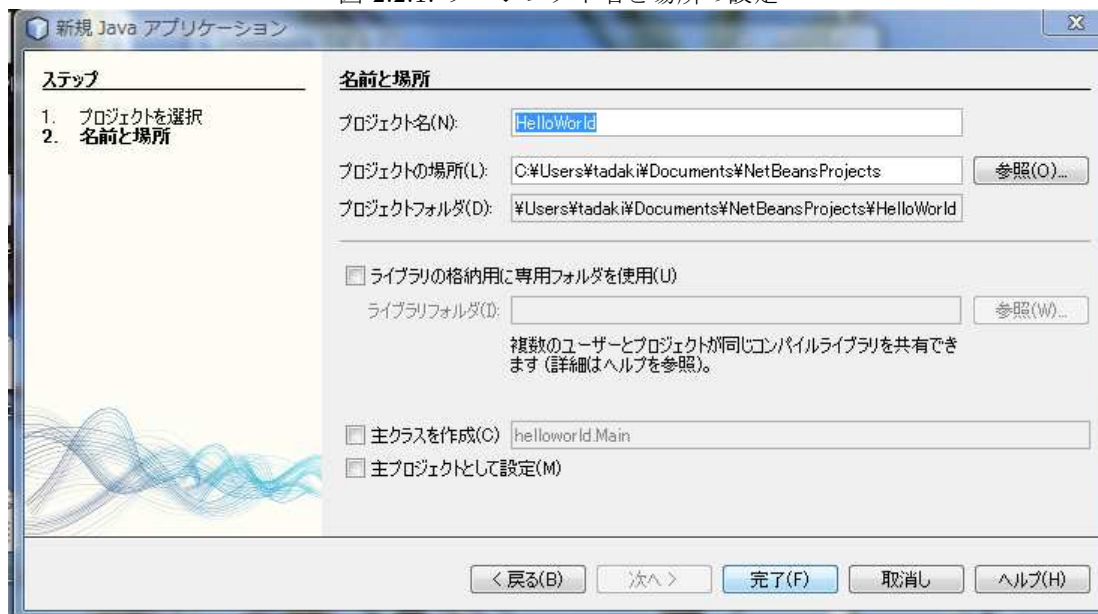
「主プロジェクトとして設定」と「主クラスを作成」のチェックボックスを外してください(図 2.2.1)。「完了」ボタンを押すと、設定が終り、プログラミングを開始できます。

プロジェクト開始直後の様子が図 2.2.2 です。次に Java のソースファイルを作成します。左のプロジェクト名の左のマークをクリックすることで、その項目を展開したり畳んだりすることができます。プロジェクトの下にある「ソースパッケージ」の更に下の「デフォルトパッケージ」の下にソースファイルを作っていきます。

新しいソースファイルを作るには、「デフォルトパッケージ」にマウスをあわせて、右ボタンを押します。そこで、「新規」を選び、表れるメニューの中から作成したいファイルの種類、ここでは「JFrame フォーム」を選びます。

しかし、始めて NetBeans で「JFrame フォーム」を作成する場合には、メニューの中に無いかも知れません。その場合には、メインのメニューで、「ファイル」→「新規ファイル」→「Swing GUI フォーム」→「JFrame フォーム」として、選択します。

図 2.2.1: プロジェクト名と場所の設定



「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。ここでは、メソッド `initComponents()` を省略しています。以後、同様にこのメソッドの部分は表示しません。

なお、`/*と*/`で囲まれた部分、および`//`で始まる行は、コメント、つまりプログラムの説明です。改めてタイプしたり、既にあるものを消す必要はありません。Program 2.2.1 および以降のプログラムでは、必要なコメントだけを残しています。

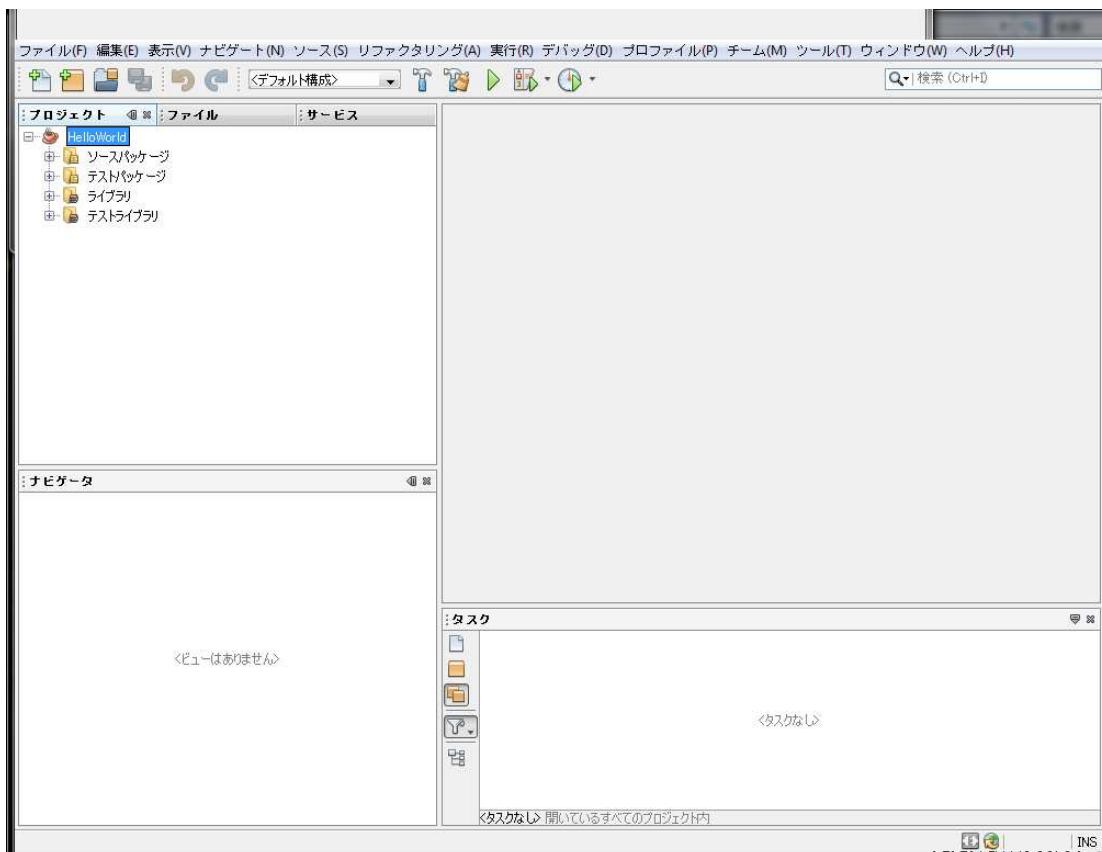
編集が終わったら、ソースコードをコンパイル (compile) する必要があります。コンパイルすることで、人間が読める文字列からコンピュータが実行できる言葉へ翻訳が行われ、`HelloWorld.class` ファイルが作成されます。

NetBeans 6.9 では、ソースファイルを保存すると自動でコンパイルが行われます。また、プロジェクト名 `HelloWorld` の上でマウス右ボタンを押し、「構築」を行うことでコンパイルが実行することができます。

複数のファイルからなるプログラムの場合、整合性が取れなくなる可能性もあります。その場合には、「生成物を削除して構築」を選び、全てのソースファイルから強制的にコンパイルすることもできます。

正しくコンパイルできたら、実行しましょう。「プロジェクトウィンドウ」内の `HelloWorld.java` にマウスを合わせ、右ボタンを押し、「ファイルを実行」を選択します。正しく実行できましたか (図 2.2.3)。

図 2.2.2: プロジェクト開始直後の NetBeans



## 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。例えば、銀行口座を考えましょう。銀行口座には、その所有者や口座の種類などの属性と、現在の預金残高のような状態があります。口座への預金振込や口座からの預金引出し、あるいは残高照会などの操作があります。

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。銀行口座の例でいくと、銀行口座という類形化されたクラスに対して、個々の口座は、名義や預金高という具体的なデータが入ったオブジェクトとなります。

オブジェクト指向プログラミング言語では、属性や状態などのデータとその操作する方法を組にして、クラスを定義します。クラスは、変数型のように使うことができます。そのクラスが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。最初のほうにある



---

**Program 2.2.1** HelloWorld.java

---

```
import java.awt.*;

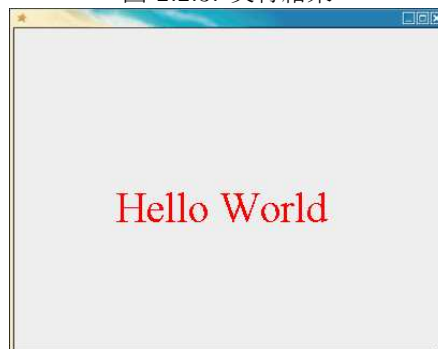
public class HelloWorld extends javax.swing.JFrame {
    //表示に使うフォントを定義
    Font font = new Font("TimesRoman", Font.PLAIN, 40);

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

    @Override
    public void paint(Graphics g) {
        super.paint(g);
        g.setFont(font); //フォントを設定
        g.setColor(Color.red); //色を設定
        g.drawString("Hello World", 100, 200); //文字列を表示
    }
}
```

---

図 2.2.3: 実行結果



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

は、プログラムの一部というよりも、使う名前の省略を定義しています。java.awt というクラスの集合 (パッケージと呼ぶ) の下にある様々なクラスの名前を、始めの部分 java.awt を省略して書くことを表しています。例えば、java.awt.Graphics というクラス名を、Graphics と省略できるようにします。

このプログラムは、HelloWorld という一つのクラスだけで構成されています。

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

キーワード public は、他のプログラムからこのクラスを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.JFrame 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。

オブジェクト指向プログラミング言語では、このように既に作成されているクラスを元にして、新しいクラスを定義することができます。新しいクラス中で、明示的に書き換えていないデータや操作は、元になったクラスのもので使われます。

クラスの操作、つまりクラスのオブジェクトのデータを操作したり、オブジェクトから何かを操作する方法をメソッド (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、描画を行う特別なメソッドであり、必須です。

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、つまり最も最初に実行されるメソッドです。

メソッド `initComponent` は、フォームエディタを使って GUI の設計をした際に使われます。しばらくは使いませんが、消してはいけません。

**演習 2.1** メソッド `drawString` にある文字列、数値、また `setColor` で指定される色を変更してみなさい。

## 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。まずは、簡単な計算の実行とその結果の表示です。

表 3.1: Java で使える原始型

<code>int</code>	整数型	32bit 符号付き整数 − $2^{31}$ から $2^{31} - 1$ までの整数
<code>long</code>	整数型	64bit 符号付き整数
<code>short</code>	整数型	16bit 符号付き整数
<code>byte</code>	整数型	8bit 符号付き整数
<code>char</code>	文字型	16bit Unicode 文字 16bit 符号無し整数と同じ
<code>boolean</code>	論理	<code>true</code> または <code>false</code>
<code>float</code>	浮動小数型	32bit 符号付き浮動小数
<code>double</code>	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、これらの変数に保存できる数値の種類 (型と言います) が整数であることを示しています。

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、全てのデータは全て 0 と 1 からなる二進数で表現されています。そこで、変数の型を定義することによって、その変数として保存されている 0 と 1 からなる列と値を対応つける方法を定義する必要があります。それが、変数の型を定義することにあたります。

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、プログラム中に現れる全ての変数に対して、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。右辺の計算を最初に行い、その結果を左辺に代入します。代入の両辺の型が同じになるようにするのが原則です。

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、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

---

```
import java.awt.*;

public class SimpleArithmetic extends javax.swing.JFrame {

    private Font f = new Font("TimesRoman", Font.PLAIN, 40);

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

    @Override
    public void paint(Graphics g) {
        super.paint(g);
        g.setFont(f);
        int a = 8;
        int b = 5;
        int c;
        // 足し算
        c = a + b;
        String s1 = String.valueOf(a) + "+" +
            String.valueOf(b) + "=" + String.valueOf(c);
        g.drawString(s1, 100, 150);
        //引き算
        c = a - b;
        String s2 = String.valueOf(a) + "-" +
            String.valueOf(b) + "=" + String.valueOf(c);
        g.drawString(s2, 100, 200);
    }
}
```

---

## 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も可能です。

一方、文字列型 (`String`) は原始型ではなく、クラスです、文字列はダブルクォーテーション" で区切ることで定義します。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、一旦生成されると変更できないという特徴があります。しばしば変更する必要がある場合には、別の文字列型 `StringBuilder` を使います。

表 3.2: 特殊文字

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

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 初めてのGUI

### 4.1 はじめに

Javaの大きな特徴の一つは、OSに依存しない点です。この点は、GUI (Graphical User Interface) についても同様です。

広く使われるプログラミング言語は、標準化が行われるために、異なるOSであっても、概ね動作するようになっています。しかし、システムに依存している部分、例えば、プロセス管理などは、共通にはできません。

さらに、ウィンドウ環境は、各OSによって標準的なものが異なります。従って、多くのプログラミング言語は、GUI部分を言語から外しています。GUIに関する部分は、言語・OSごとにライブラリとして提供されることになります。

それに対して、Javaの場合は、Javaが動作する環境では、共通のGUIを提供するようになっています。

GUI付のプログラムを作成すると、ユーザが使い易いだけではありません。開発側からみると、開発途中の動作確認・結果表示、さらにデモンストレーションにも役立てることが可能です。

GUIを作るのは、大変な作業です。要素の階層構造を作る、要素の属性を定める、要素の配置を制御する、そして要素の動作を決める、などの作業が必要だからです。しかし、NetBeansを使うとこうした作業を、マウスを使って、容易に実行することができます。

Javaの内容の進んでいく前に、今後のプログラムの基礎として、簡単なGUIを作ることにしましょう。次章からは、このプログラムを流用しながら、作業を進めましょう。



図 4.1.1: FirstGUI 実行画面

## 4.2 GUI 作成の流れ

### 4.2.1 全体の枠

図 4.1.1 を見てください。これが、今回作成するプログラムの GUI です。全体の枠の中に、二つの領域があります。上の領域の中に、二つのボタンがあります。このように、GUI には、階層的な構造があります。全体を入れる枠、他の要素を入れる領域、そして個別の役割を担った要素です。

Java には、`java.awt` という GUI ライブラリがあります。これをさらに改良したものが、今回使用する `javax.swing` というライブラリです。Swing では、図 4.2.1 のように、階層的に要素を積み上げていくことで GUI を作成します。

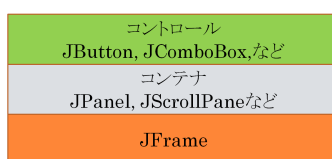


図 4.2.1: Java GUI の階層構造

以下、GUI 作成の手順を説明していきます。まず、新しいプロジェクト (今回の名前は `FirstGUI` としましょう) を作ります。新規ファイルとして、`JFrame` フォームを選び、名前をやはり `FirstGUI` とします。「デザイン」タブを開くと図 4.2.2 のようになっています。

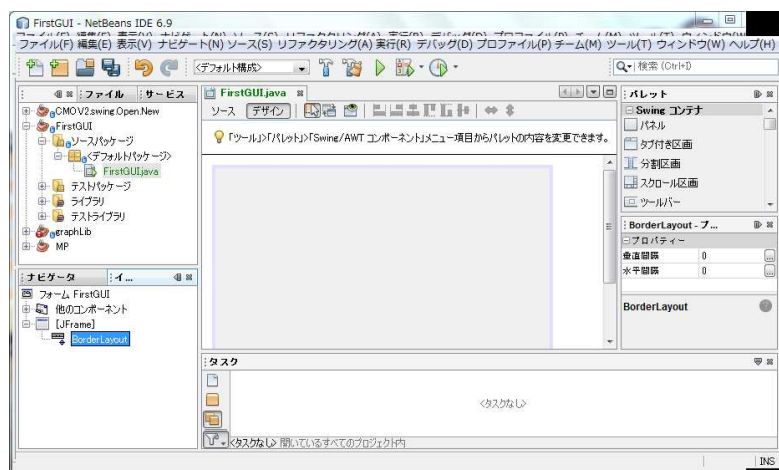


図 4.2.2: JFrame の状態

画面左の「ナビゲータ」部分が、GUI 要素の階層構造を表しています。[`JFrame`] でマウス右ボタンを押し、「レイアウトを設定」を選びます。このなかの、「ボーダーレイアウト」を選んで設定したものが図 4.2.2 の状態です。



レイアウトとは、この要素（ここでは [JFrame]）の子の要素の配置方法のことです。レイアウトを制御するライブラリが「レイアウトマネージャー」です。ここで言う配置には、初期配置だけでなく、要素の増減時の配置や、親の要素のサイズが変わった時の配置の制御が含まれています。

通常、GUI のあるプログラムでは、前述のように、一番外の枠の中に領域を作り、各領域にボタンなどを配置していきます。JFrame は、この一番外の枠ですから、ここに直接ボタンなどを置くことは、通常は行いません。

JFrame の子となる要素は、上下、左右、中央などに配置される場合が多くなります。そこで、そのような要素配置を制御する「ボーダーレイアウト」を用いることにします。

### 4.2.2 ボタン用パネル

次に、ボタンを置くためのパネルを作りましょう。画面右側の「パレット」に GUI の部品が表示されています。一番上が、「Swing コンテナ」と呼ばれる、他の部品を置くための部品です。一番上に「パネル」があります。このパネルをマウスの左ボタンを押しながら、中央の灰色の部分に移動させ、マウスを放すと [JFrame] の子として、JPanel のインスタンス `jPanel1` が作成されます。

このパネルの名前を変更しましょう。中央領域内の `jPanel1` で右ボタンを押すとメニューが現れます。一番上の「変数名を変更」を選び、今回は `buttons` と名を変更します。このような操作は、「ナビゲータ」領域で、該当する要素の箇所でも右ボタンを押すことでも行うことができます。

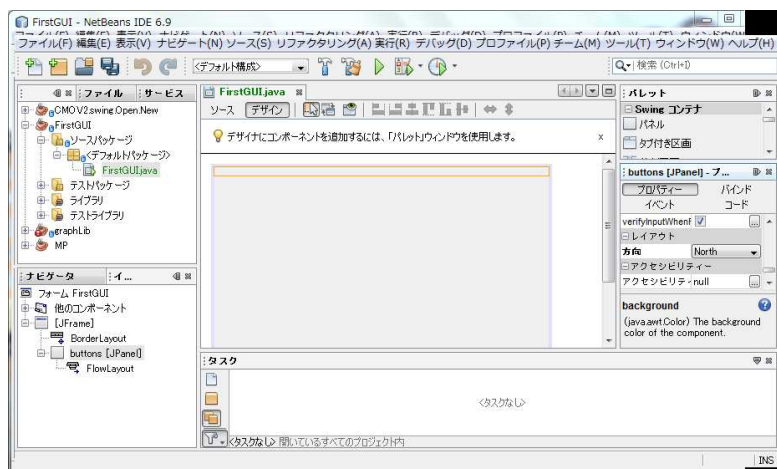


図 4.2.3: ボタン用パネルを置いた状態

同様に、メニューの中から「プロパティ」を選ぶと、要素の様々な性質を変更することができます。ボタンは、画面の上に配置します。メニューの中の「レイアウト」にある、「方向」を「North」に設定しましょう。ここまで行った状態を図 4.2.3 に示します。

ついでに、ボタン用パネルの背景色も設定しましょう。メニュー中の一番上に「background」があります。右の口を押すと、色選択画面が表示されます。好きな色を選びましょう。

選んだ要素のプロパティは、画面右にも表示されています。ここから変更することも可能です。

### 4.2.3 ボタン

画面右の「パレット」の「Swing コンテナ」の左の□を押すと、その項目が閉じて、「Swing コントロール」の項目が展開されます。その中に、「ボタン」があります。マウスでドラッグして、ボタン用パネル中に置きましょう。**JFrame**ではなく、「ボタン用パネル」**buttons**に置いてください。間違えても、マウスでドラッグして移動されることができます。

このボタンに表示される文字列を「終了」に、変数名を「quit」に設定します。色も適当に設定しましょう。

同様に、「実行」と表示されるボタン「exec」を作ってください。

このボタンの動作は次節で行います。

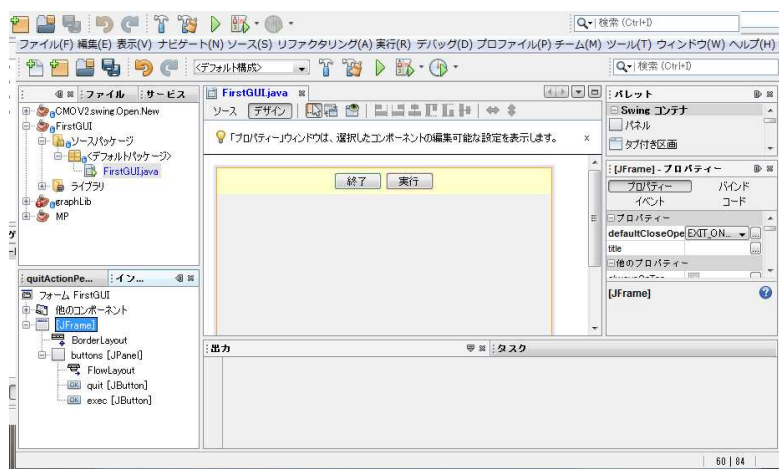


図 4.2.4: ボタン用パネルにボタンを置いた状態

### 4.2.4 テキスト区画

プログラムの実行結果を表示する領域を **JFrame** の中央に設置しましょう。図を表示するならば、再び「パネル」を置きますが、今回は、テキストを表示することにします。

「Swing コントロール」中の「テキスト区画」を **JFrame** の中央にドラッグします。プロパティ中の「レイアウト」「方向」は **Center** にします。

各要素には、大きさを定義することができます。このテキスト区画の大きさを設定しておきましょう。プロパティ中の「PreferredSize」(希望サイズ)を「[300,300]」に設定します。

左の「ナビゲーター」を見ると、「テキスト区画」に対応した「**JScrollPane1**」の下に階層があることが解ります。「**JScrollPane1**」はスクロールバーを表示する部品であり、その中にスクロールされる別の要素があるのです。「+」部分をクリックすると、下層に **JTextPane** のインスタンスがあります。実際に文字列が表示されるのは、この部分です。この変数名を **textPane** に変更しておきましょう。

この状態を図 4.2.5 に示します。ここまでで、GUI 部品の配置は終わりです。

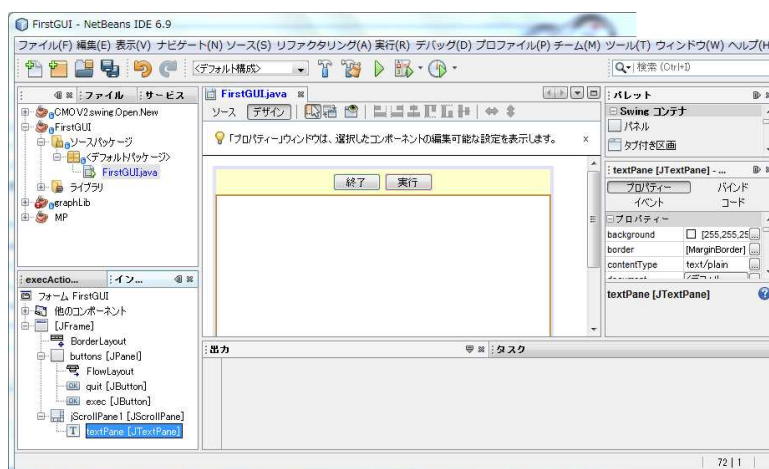


図 4.2.5: テキスト区画を置いた状態

GUI 要素の配置や、要素の色などの情報が、どのような形で保存されているかを少しだけ見ておきましょう。「ソース」タブを選び、プログラムソースを表示します。下のほうに、「Generated Code」と書かれた区画があります。左の「+」を押すと、その部分のソースファイルが展開されます。`initComponent()` メソッドの中身に対応しています。ここで、GUI 部品が配置され、性質が設定されています。

この部分は、水色になっていることに注意してください。この部分は、NetBeans のの中からは編集することができません。ソースファイルを置いているフォルダには、対応するソースファイル `FirstGUI.java` の他に、`FirstGUI.form` というファイルが置かれています。この中に GUI 部品の情報が格納され、ソースファイルとの間で連動しています。

ソースの表示の左の「-」をクリックすることで、展開していた部分を折りたたむことができます。

### 4.3 ボタンの動作

GUI のプログラムは、イベント駆動型のプログラムと呼ばれます。イベントとは、マウスのクリックや大きさの変更や、部品内の文字列等の変化などです。このようなイベントによって、プログラムの一部が開始されるものをイベント駆動と呼びます。

java のイベント駆動では、イベントを捕まえる聞き手 (listener) と対応する動作 (action) を定義することでプログラミングされます。NetBeans では、リスナとアクションを要素に対応付けて容易に定義することができます。

まず、「終了」ボタンの動作を定義しましょう。「デザイン」タブの状態で、「終了」ボタンをダブルクリックします。また h、「ナビゲーター」領域で「終了」ボタンの変数名「quit」をダブルクリックします。

画面は、「ソース」タブに変わり、編集すべき部分が表示されます。NetBeans が対応するアクションとして、`quitActionPerformed` を生成してくれます。図 4.3.1 に示します。

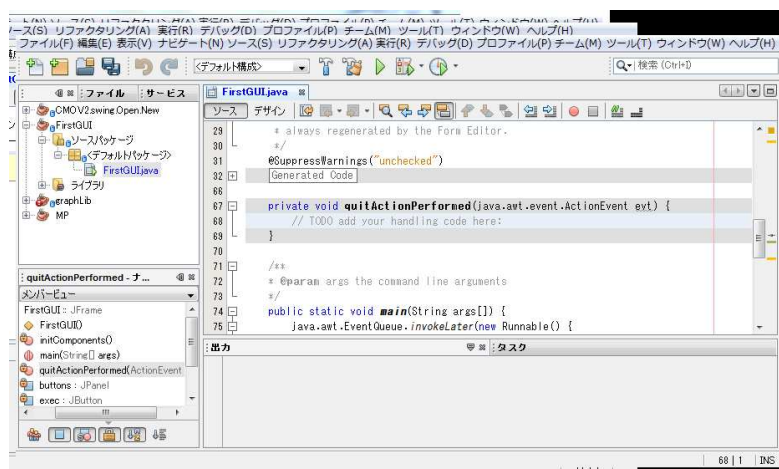


図 4.3.1: 「終了」ボタンの動作

このプログラムの終了は、JFrame 及びその子の部品が使っている資源をシステムに返却し、JFrame を閉じることです。そこで、`dispose()` というメソッドを呼出します。図 4.3.1 の

```
// TODO add your handling code here;
```

部分を消して、

```
dispose();
```

としましょう。

今回は、「実行」ボタンを押すと、現在の時刻が表示されるようにします。「終了」ボタンと同様の操作で、「実行」ボタンに対応するアクションを生成しましょう。

「実行」ボタンに対応するアクションの実装として

```
Date now = new Date();
textPane.setText(now.toString());
```

としましょう。

ソースファイルの左に赤いクエッションマークがでます。また、`Date` という文字列の下に赤い波線が表示されます。誤りがあるという情報です。

`Date` は、汎用ライブラリ `java.util.Date` を使いたいのですが、適切に `import java.util.Date` を記述していないのが原因です。赤いクエッションマークにマウスを併せて左ボタンを押すと対策一覧が表示されます。その中から「`java.util.Date` をインポートに追加」を選びます。こうすることでソースファイルの先頭部に `import java.util.Date` が追加されます。

ソースファイル中の

```
Date now = new Date();
```

は、現在の時刻が保存された `Date` のインスタンス `now` を生成します。そのインスタンスを文字列としてテキスト領域 `textPane` に表示しています。

## 第5章 配列と繰り返し

### 5.1 配列

表に数値が並んでいて、その和を計算することを考えましょう。データには番号  $i$  が付けられて、 $x_i$  と呼ぶことにしましょう。手で確実に和を計算するには、表の各行にそれまでの和を書いて行くと良いでしょう。表 5.1 の右端の列がその和になります。

$i$	$x_i$	S
0	20	20
1	4	24
2	5	29
3	21	40
4	15	55

表 5.1: 中央の列のデータの和を右端に表示する例。  $i$  行までの和が右端に表示される。

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。また表の各行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。

このように、大量のデータに同じ操作を繰り返して実行することが、コンピュータ利用の最も重要な使い方です。最初のコンピュータ (の原型) は、国勢調査の統計処理に使われました。

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。

表 5.1 の例を拡張して、 $N$  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。

$$\langle x \rangle = \frac{1}{N} \sum_{i=0}^{N-1} x_i \quad (5.1.1)$$

$$\langle \Delta x^2 \rangle = \frac{1}{N} \sum_{i=0}^{N-1} (x_i - \langle x \rangle)^2 = \frac{1}{N} \sum_{i=0}^{N-1} x_i^2 - \langle x \rangle^2 \quad (5.1.2)$$

ここで、記号

$$\sum_{i=0}^{N-1} x_i \quad (5.1.3)$$

は、 $x_i$  を  $i = 0$  から  $i = N - 1$  まで足す、つまり

$$\sum_{i=0}^{N-1} x_i = x_0 + x_1 + x_2 + \dots + x_{N-2} + x_{N-1} \quad (5.1.4)$$

を表しています。

式 (5.1.1) を見ると、右辺の計算が終わったあと、その結果を左辺に代入するのようになります。これでは、途中まで和を計算した値をどのように覚えておくのかが分かりません。

---

**Program 5.1.1** ArrayAndLoop.java : GUI を除く部分

---

```
public class ArrayAndLoop extends javax.swing.JFrame {
```

```
    private final int data_num = 10;
    private int data[];
    private double a;
    private double d;
    private String cr = System.getProperty("line.separator");
```

```
    /** Creates new form FirstGUI */
    public ArrayAndLoop() {
        initComponents();
    }
}
```

---

そこで和の計算を表 5.1 でやったように、見直してみます。始めの  $n$  個の和を

$$S_{n-1} = \sum_{i=0}^{n-1} x_i \quad (5.1.5)$$

と表すことにします。これを使うと

$$S_n = S_{n-1} + x_n \quad (5.1.6)$$

$$S_0 = x_0 \quad (5.1.7)$$

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。これが、表 5.1 で行っていることの数学的表現です。数学的表記では、左辺に、 $S_n$  のように、インデクス  $n$  が現れますが、コンピュータで計算する際には、全て同じ変数でも大丈夫です。

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できます。前述のように、コンピュータとはもともとそのような同じ操作の繰り返しが得意な機械です。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。前章で見たように、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の要素かを指定することができます。

Java では、配列の宣言を

---

**Program 5.1.2** ArrayAndLoop.java(続き)

---

```
/**
 * 計算の実行
 */
private void createData() {
    //データ作成
    data = new int[data_num];
    for (int i = 0; i < data_num; i++) {
        data[i] = (int) (100 * Math.random());
    }
}

/**
 * 平均と分散の計算
 */
private void calculation() {
    a = 0.;
    d = 0.;
    for (int i = 0; i < data.length; i++) {
        a += data[i]; //和
        d += data[i] * data[i]; //二乗の和
    }
    a /= (double) data_num;
    d = d / data_num - a * a;
    d = Math.sqrt(d);
}

public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {

        public void run() {
            new ArrayAndLoop().setVisible(true);
        }
    });
}
}
```

---

型 変数名 [];

または

型 [] 変数名;

で行います。記号 [] が配列を表しています。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。つまり、メソッドがあります。

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、配列の各要素には初期値として `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。インデクス `i` が `0` から始まる点に注意が必要です。

## 5.2 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。

乱数は、でたらめな数字の列のことです。シミュレーションなどで使います。でたらめな数の列とは、さいころを振るように、出る数が予想できないものです。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。



## 5.3 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、あらかじめ繰り返し回数を指定することをせず、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。

## 5.4 プログラムブロック

プログラムの中で `{}` で囲まれたプログラム文の集まりをプログラムブロックと呼びます。Java のプログラムは

```
クラス宣言{
    クラスの内容
}
```

というプログラムブロックを一番外側に、その中に

```
メソッド{
    メソッドの内容
}
```

というメソッドの記述が入っています。さらに、その内側に

```
命令{
    プログラム文
}
```

という構造が入ります。ここで「命令」とは、`for` や `while` のようなものです。

プログラムを書くときには、文章を書く時と同じように全体の構造を階層化して考えます。一つのモノに対応したプログラムブロックがクラスです。そのクラスの操作や動作に相当するプログラムブロックがメソッドになります。各メソッドは更に細かい処理の連続ですが、そのなかのまとまった処理が小さなプログラムブロックになります。

いろいろな話題が入り乱れた文章、特に解説や論文は非常に読みにくいですね。ですから、一つの段落には一つの話、関連した話題をまとめて節にし、それをまとめて章にするように階層的に書くことで読みやすい文章ができます。

プログラムでも同じです。プログラムでは、段落に相当するのが一つのプログラムブロックです。一つのプログラムブロックには一つの処理だけを書くようにします。

## 5.5 プログラムブロックと変数のスコープ

前述のように、Java では全ての変数は型とともに宣言しなければなりません。その変数は、それが宣言されたプログラムブロックの中だけで有効です。例えば、

```
int a=0;
for(int i=0;i<100;i++){
    int b=i*i;
    a += b;
}
```

というプログラムを考えます。`for` ループの中で定義されている変数 `b` は、このループ内だけで使える変数で、ループの外へは持ち出せません。ループの回数を数えるカウンタ `i` もループ内だけで有効です。一方、変数 `a` はループの外で定義されて、ループ内で値を変更すると、ループの外で新しい値を使うことができます。

このように変数が有効な範囲をプログラムブロックに制限することを局在化と言います。有効な範囲をスコープと呼びます。局在化することで、変数の値がプログラムを書く人の目の届かないところで変えられてしまうことを防ぐことができます。

## 5.6 プログラム作成

前回のプログラムを再利用することで、新しいプログラムを作成しましょう。

まず、NetBeans 内で、新しいプロジェクト `ArrayAndLoop` を作成します。そこに、前回のプロジェクト `FirstGUI` からメインのクラス `FirstGUI` をコピーして、リファクタリングしながら貼り付けます。貼り付ける際の名前は、`ArrayAndLoop` にしましょう。残念ながら、`main` メソッド中のクラス名 `FirstGUI` が残っていますので、手で `ArrayAndLoop` とします (5.1.2 の末尾参照)。

次に、プログラムの前半部分 (GUI の手前) に 5.1.1 及び 5.1.2 の部分を追加します。

最後に、ボタンを押した動作を追加します (5.6.1)。

---

**Program 5.6.1** ArrayAndLoop.java : ボタンの動作部分

---

```
private void actionPerformed(java.awt.event.ActionEvent evt) {
    createData();
    calculation();
    StringBuilder b = new StringBuilder();
    b.append("平均 : ");
    b.append(a);
    b.append(cr);
    b.append("標準偏差      : ");
    String r = String.valueOf(d);
    if (r.length() > 8) {
        r = String.valueOf(d).substring(0, 7);
    }
    b.append(r);
    textPane.setText(b.toString());
}
```

---

**演習 5.1** Program5.1.1 の for ループを while を使って書き換えなさい。

**演習 5.2** 与えられた数の和ではなく、0 から data\_num-1 までの和に書き換えなさい。



## 第6章 条件分岐

### 6.1 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ったり、元へ戻って再度実行する必要がある場合がたくさんあります。条件に応じてプログラムの実行する部分を分けることを条件分岐と言います。

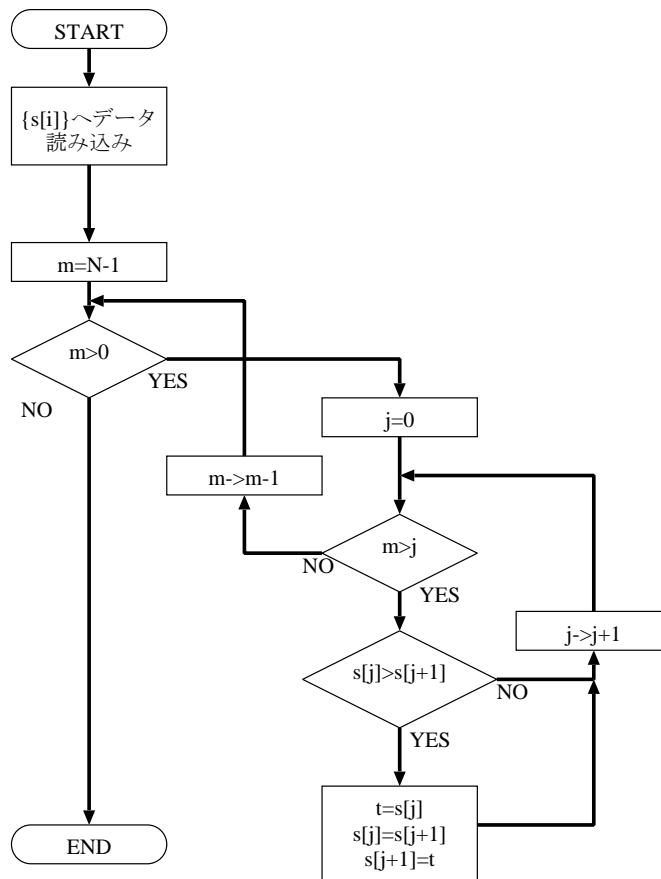


図 6.1.1: 泡立ち法

ここでは、でたた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。

## 6.2 泡立ち法による並べ替え

データの並び替えの方法として様々なものが知られています。ここではもっとも簡単な泡立ち法 (bubble sort) を扱います。泡立ち法という名前の由来は、右を上に見立てると、大きな (小さな) 値の順に、右端 (上端) に順次並んで行くことによります。

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 6.1.1 に示します。四角は処理を、菱形は条件分岐を表します。処理は、矢印に沿って進んでいきます。

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、配列の先頭から順番に、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。

同じ操作を先頭から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。

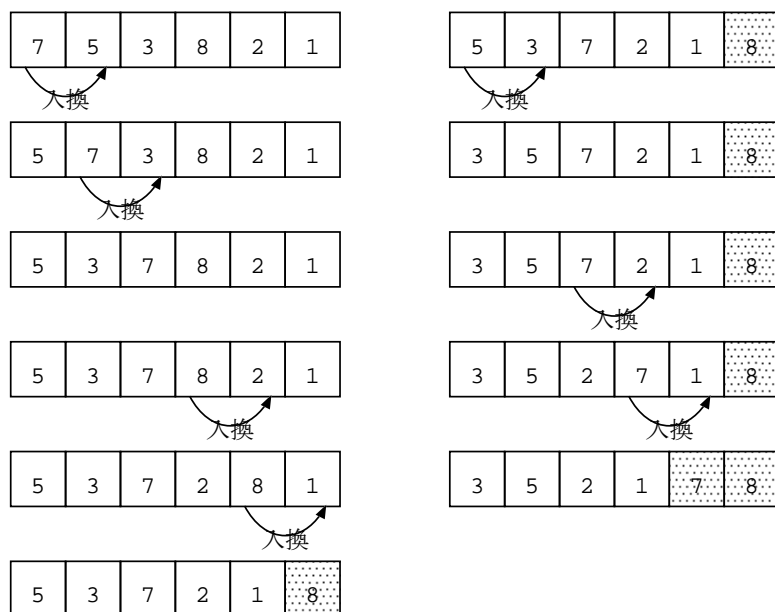


図 6.2.1: 泡立ち法によって配列の内容が変化の様子

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、最後の二つの要素  $s[N-2]$  と  $s[N-1]$  を比較する操作は不要です。実行しても害はありませんが、無断な処理になります。つまり、最初は、 $N-1$  回の比較

しなければなりません。次は  $N - 2$  回、その次は  $N - 3$  回と、次第に比較する回数を減らし、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、配列の内容は変化する様子を図 6.2.1 に示します。

**演習 6.1** 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。

## 6.3 泡立ち法の Java プログラム

---

### Program 6.3.1 Condition.java : GUI 以外

---

```
public class Condition extends javax.swing.JFrame {

    final int data_num = 10;
    int[] data = new int[data_num];

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

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

    /**
     * 泡立ち法
     * @param d データの入った配列
     * @return スワップの回数
     */
    public int bubble(int d[]) {
        int m = 0;
        for (int j = d.length - 1; j >= 1; j--) { //後ろからループを回す
            for (int i = 0; i < j; i++) {
                if (d[i] > d[i + 1]) { //順序が逆の場合
                    m++;
                    int c = d[i];
                    d[i] = d[i + 1];
                    d[i + 1] = c;
                }
            }
        }
        return m;
    }
}
```

---

**Program 6.3.2** Condition.java(続き)

---

```

/**
 * 配列内の数値を文字列化する
 * @param d データ配列
 * @return 結果の文字列
 */
private String mkstr(int d[]) {
    //可変な文字列を作る
    StringBuilder str = new StringBuilder();
    //配列の要素を","で区切って文字列化
    str.append(String.valueOf(d[0]));
    for (int i = 1; i < d.length; i++) {
        str.append(",");
        str.append(String.valueOf(d[i]));
    }
    return str.toString();//String にして戻す
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {

        @Override
        public void run() {
            new Condition().setVisible(true);
        }
    });
}
}

```

---

では、前述の泡立ちをを実際にプログラムしましょう。前回と同様に、`FirstGUI` をコピーして出発点を作成します。GUI 以外をプログラム 6.3.1 に示します。

このプログラムで必要なのは、ソートされるデータです。データの数 `data_num` とソートされるデータ `data` が最初に定義されています。

```

final int data_num = 10;
int[] data = new int[data_num];

```

次に、実際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

```

というメソッドに注目してください。引数の `int d[]` に並べ変えるべきデータが保存されています。このメソッド中に

```

    if(d[i] > d[i+1]){//順序が逆の場合
        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。つまり、条件 ( $d[i] > d[i+1]$ ) が成り立っている場合に、その二つの要素の順序を入れ換えます。

条件分岐の一般形は次のように書きます。

```

if(条件){
    操作 1
}

```

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。操作 1 は、一つのプログラム文とは限らず、複数のプログラム文でも構いません。また、操作 1 の部分は{と}に囲まれたプログラムブロックになっているため、その中だけで有効な局所的な変数を使うこともできます。

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。条件分岐の内部で更に条件分岐する入れ子 (nesting) 構造も許されます。

条件を記述する際には、表 6.1 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。条件は、数値の比較だけでなく、論理型 (boolean) の戻り値を持つメソッドを書くこともできます。たとえば文字列を  $==$  で比較することはできません。文字列を比較する場合には、`String.equals(String)` という論理型のメソッドを使います。

表 6.1: 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
$>$	$>$	$<$	$>$
$>=$	$\geq$	$<=$	$\leq$
$==$	$=$	$!=$	$\neq$

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]
```

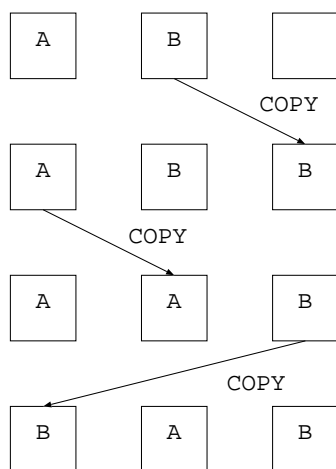


図 6.3.1: 二つのデータの入れ換えの概念図

が成り立つ場合に、二つの配列要素を入れ換えます。人間が両手を使って、二つのものの場所を置き換えるように、コンピュータは二つの要素の中身を直接に入れ換えるということではできません。そこで、 $d[i]$  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。この時使う変数  $c$  は、ここだけで必要な変数です。従って、このプログラムブロック内の局所変数として定義されています。図 6.3.1 に二つのデータ入れ換えの概念図を示します。

---

**Program 6.3.3** Condition.java : 「実行」 ボタンの動作
 

---

```
private void actionPerformed(java.awt.event.ActionEvent evt) {
    String cr = System.getProperty("line.separator");
    initarray(data);
    String input = mkstr(data);
    int status = bubble(data);
    String output = mkstr(data);
    StringBuilder str = new StringBuilder();
    str.append("Input : ").append(input).append(cr);
    str.append("Output: ").append(output).append(cr);
    str.append("# of swap: ").append(status);
    textPane.setText(str.toString());
}
```

---

ボタンを押した時の動作を、見ましょう。プログラム 6.3.3 を見てください。「実行」 ボタンを押すと、`initarray` メソッドを呼び、データ `data` を生成します。データを文字列としたものが `input` に保存されます。そのデータを `bubble` メソッドでソートします。データ入れ替え回数が変数 `status` に保存されます。ソートされたデータを文字列に変換し、`output` に保存します。あとは、これらを `textPane` に表示します。

**演習 6.2** プログラム 6.3.1 において、変数  $m$  の役割を考察し、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 6.1 で予想した結果と比較しなさい。

## 6.4 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。更に場合を分けたい場合には、`if` ブロックの中で更に `if` で分岐します。入れ子 (nesting) 構造と言います。

しかし、ある状態が多数ある状態のいずれに相当するかなど、二つ以上に場合分けをしたい場合もあります。このような多数の条件への分岐を `if` で行くと入れ子構造が多重となって、プログラムが非常に読みにくくなります。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、`switch` 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。

## 6.5 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。引数は、数学の関数の変数に相当します。メソッド `paint` から呼び出す際の変数名とこのメソッドの定義に表れる引数の名前が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7章 簡単な作図

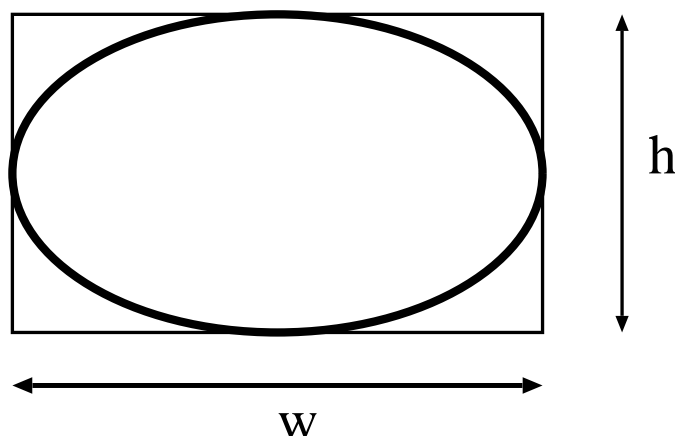
### 7.1 簡単な作図の基礎

いよいよ簡単な図を描いてみましょう。まず、アプリケーション上の座標について整理しておきます。コンピュータのウィンドウでは、表示された領域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。画面の解像度が異なると、実際に表示される大きさが異なります。

図 7.1.1: 円や楕円の座標指定

`drawOval(int x, int y, int w, int h)`

`(x,y)`



基本的な描画メソッドを見ていきます。全て `Graphics` クラスのインスタンスに対する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 Polygon クラス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。ウィンドウには背景色 (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、ウィンドウ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。

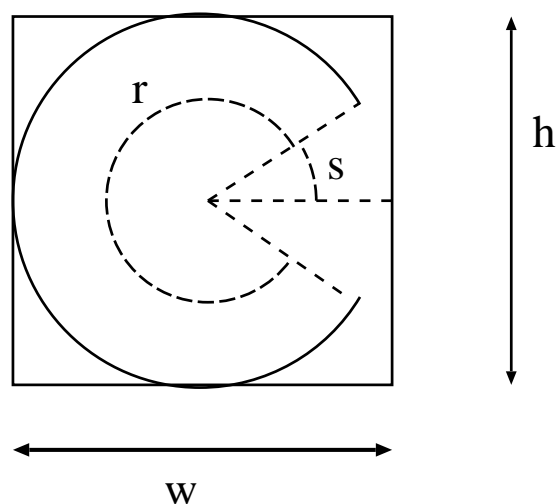
文字列の表示は、今までも使ってきましたが、基本的性質をまとめておきましょう。文字列を表示するにはフォントが必要です。そのために、Font オブジェクトを生成しておきます。

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

図 7.1.2: 弧の座標指定

`drawArc(int x, int y, int w, int h, int s, int r)`

`(x,y)`



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。

## 7.2 簡単な図形を描くプログラム

プログラム 7.2.1 は、簡単な作図をするプログラムです。いつものように、自動生成される部分を省略しています。

図形の種類は

```
static public enum Shape {

    SQUARE(Color.green),
    THREEDSQUARE(Color.yellow),
    CIRCLE(Color.pink),
    POLYGON(Color.orange),
    NONE(Color.white);
    private Color color;
```

**Program 7.2.1** SimpleGraphics.java

---

```
import java.awt.*;

public class SimpleGraphics extends javax.swing.JFrame {

    private Font f = new Font("TimesRoman", Font.PLAIN, 20);
    //形の種類を定義

    static public enum Shape {

        SQUARE(Color.green),
        THREESQUARE(Color.yellow),
        CIRCLE(Color.pink),
        POLYGON(Color.orange),
        NONE(Color.white);
        private Color color;

        Shape(Color c) {
            color = c;
        }

        public Color getColor() {
            return color;
        }
    };
    private Shape shape = Shape.NONE;//形の既定値

    /** Creates new form SimpleGraphics */
    public SimpleGraphics() {
        initComponents();
        for (Shape s : Shape.values()) {
            shapeSelect.addItem(s);
        }
        shapeSelect.setVisible(true);
        pack();
    }
}
```

---

```
        Shape(Color c) {
            color = c;
        }

        public Color getColor() {
            return color;
        }
    };
    private Shape shape=Shape.NONE;//形の既定値
```

で定義しています。既定値は NONE、つまり形が定義されていません。引数は、表示する際の色を指定しています。図形の定義は列挙型と呼ばれる型を使って定義しています。詳しくは後述します。



---

**Program 7.2.2** SimpleGraphics.java(続き)

---

```
@Override
public void paint(Graphics gg) {
    super.paint(gg);
    Graphics g = drawPanel.getGraphics();
    g.setFont(f);
    g.drawString(shape.toString(), 100, 240);
    g.setColor(shape.getColor());

    switch (shape) {
        case SQUARE://正方形
            g.fillRect(50, 50, 50, 50);
            break;
        case THREEDSQUARE://3次元正方形
            g.fill3DRect(50, 50, 50, 50, true);
            break;
        case CIRCLE://円
            g.fillOval(50, 50, 50, 50);
            break;
        case POLYGON://多角形
            int x[] = {100, 35, 124, 160, 148, 150};
            int y[] = {15, 83, 135, 90, 180, 150};
            int np = x.length;
            g.fillPolygon(x, y, np);
            break;
        default://どれにも当てはまらない
            break;
    }
}
```

---

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、変数 `shape` の値に応じて図形が作図されています。

`paint` メソッドの最初にある `super.paint(g)` は、親クラス `JFrame` の `paint` メソッドを実行することを示しています。ここで、ウィンドウに描かれた図が一旦消去され、設定されている背景色で塗りつぶされます。

**演習 7.1** 複数の図形を組み合わせた絵を描くプログラムを作成しなさい。

## 7.3 列挙型

データに順番を付けて管理する際に用いるのが列挙型です。これは、Java5 から新しく加わった機能です。プログラム 7.2.1 では、図形の種類を管理するのに用いています。

図形の種類は

```
static public enum Shape {
```

**Program 7.2.3** SimpleGraphics.java(続き)

---

```

private void quitActionPerformed(java.awt.event.ActionEvent evt) {
    this.dispose();
    System.exit(0);
}

private void shapeSelectActionPerformed(java.awt.event.ActionEvent evt) {
    shape = (Shape) shapeSelect.getSelectedItemAt();
    repaint();
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {

        @Override
        public void run() {
            new SimpleGraphics().setVisible(true);
        }
    });
}
}

```

---

```

    SQUARE(Color.green),
    THREEDSQUARE(Color.yellow),
    CIRCLE(Color.pink),
    POLYGON(Color.orange),
    NONE(Color.white);
private Color color;

```

```

Shape(Color c) {
    color = c;
}

```

```

public Color getColor() {
    return color;
}

```

```

};

```

で定義されています。キーワード `enum` が列挙型を示し、その型の名称として `Shape` が定義されています。図形として、`SQUARE` など、5種類が順番に定義されています。

ここでの例の場合、各要素に対して、その要素名を文字列にした名称が定義されます。例えば、要素 `SQUARE` の名称は "`SQUARE`" という文字列です。この名称を得るには、メソッド `toString()` を

用います。各項目には、引数を指定することも可能です。ここでは、色が指定されています。

列挙型変数は、多重分岐 `case` のスイッチとしても使うことができます。プログラム 7.2.1 の中で

```
switch(shape){
    case SQUARE://正方形
        ...
        break;
    case THREEDSQUARE://3次元正方形
        ...
        break;
    case CIRCLE://円
        ...
        break;
    case POLYGON://多角形
        ...
        break;
    default://どれも当てはまらない
        break;
}
```

が多重分岐に相当しています。`Shape` 型の変数 `shape` を使って `switch` をしています。場合分けの鍵になっているのは、`Shape` 型のメンバーであることに注意してください。

## 7.4 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 `netBeans` 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。この機能を使うとプログラミングが非常に楽になります。

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/javase/ja/6/docs/ja/api/index.html>

標準的なクラスやそのメソッドの使い方、内部に持っているフィールドを調べたいときに使います。



## 第8章 簡単な動画

### 8.1 描画と再描画

---

**Program 8.1.1** Clock.java : GUIとして自動で生成される部分を省略している。

---

```
import java.util.Date;

public class Clock extends javax.swing.JFrame implements Runnable {

    private Date theDate;
    private Thread runner;
    private volatile boolean id = true;

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

    @Override
    public void run() { //スレッドで実行すること
        while (id) {
            theDate = new Date();
            label.setText(theDate.toString());
            repaint();
            //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) {
            }
        }
    }

    private void start() {
        id = true;
        runner = new Thread(this);
        runner.start();
    }

    private void stop() {
        id = false;
    }
}
```

---

ここでは、簡単な動きをするプログラムを作成します。

アプリケーション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、既に存在しているビデオなどの動画ファイル(mpeg など)を表示することではありません。ここでの動画とは、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。つまり、スピードの早い紙芝居を見せることで、動きを表現することを考えます。

最初に、文字列が次々に変化する例題を作成してみましょう。ここでは、現在の時刻が表示され、次々と新しい時刻に更新される時計のプログラムを作成します。

動画を表示する基本となるのは、描画の繰り返しです。Java では、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実現します。

---

**Program 8.1.2** Clock.java の続き

---

```
private void quitActionPerformed(java.awt.event.ActionEvent evt) {
    dispose();
}

private void startActionPerformed(java.awt.event.ActionEvent evt) {
    start();
}

private void stopActionPerformed(java.awt.event.ActionEvent evt) {
    stop();
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {

        @Override
        public void run() {
            new Clock().setVisible(true);
        }
    });
}
```

---

このように描画を繰り返す場合、スレッド (thread) というの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)、前の処理が終ると次の処理に移るようになっていきます。今回のプログラムでは、通常の初期化から最初の描画までのスレッドとは別に、繰り返し再描画するためのスレッドを作ります。

このようにスレッドを複数動作させるプログラムをマルチスレッド (multithread) プログラムと呼びます。つまり、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。しかし、時間を短く分割し、複数の処理を交替に実行することで、あたかも多数の処理が並行して動作しているように見せることができます。Java はマルチスレッドを簡単に作成できるプログラミング言語です。

マルチスレッドで時刻表示のプログラムを作るには、初期化処理で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。

スレッドを使った時刻表示プログラム 8.1.1 に示します。スレッドから呼び出しを行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。スレッドから見ると、このクラスの中で何をしても、`Runnable` という「クラス」の仲間であって、スレッドから起動できると判断されます。

スレッドから呼ばれた際の動作は `run()` というメソッドで記述されます。このメソッドはインターフェイス `Runnable` を使用する際には、必ず記述しなければなりません。このメソッドの中で、現在の時刻を取得し、`repaint()` を使って再描画をします。メソッド `repaint()` は、描画の `paint` メソッドを呼びます。その後、

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

を使って、1000 ミリ秒停止します。

このプログラムでは、新しいスレッドとして `Thread` というクラスのオブジェクトとして生成しています。ここでは、`runner` という名のスレッドが宣言されています。

```
runner = new Thread(this);
```

は、新しいスレッド `runner` が起動するのが、このプログラム `this` であることを示しています。

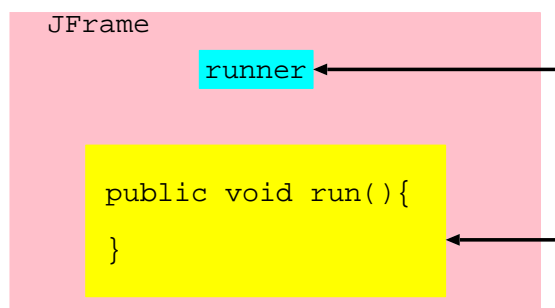


図 8.1.1: スレッドによる呼び出し

実際の時刻を描画することは、`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 `JFrame` 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調べることができます。

```
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。

## 8.2 動く正方形

---

### Program 8.2.1 MovingSquare.java

---

```
import java.awt.Color;
import java.awt.Graphics;

public class MovingSquare extends javax.swing.JFrame implements Runnable {

    private Thread runner;
    private int t = 0;
    private int sqr[] = {10, 10, 50, 50};
    private volatile boolean id = false;
    private Color color=Color.red;

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

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

    public void stop() {
        id = false;
    }
}
```

---

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 8.2.1 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。

もちろん、実際に図形が画面中を移動するわけではありません。各時刻に、画面を背景色で塗りつぶして古い図形を消し、その後で新しい位置に図形を描けば、図形が移動しているように見えます。

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。

表 8.1:  $x$  座標と高さ  $h$  が時刻  $t$  とともに変化する様子

$t$	0	5	10	15	20	25	30	35	40	45	50
$x$	10	15	20	25	30	35	40	45	50	55	60
$h$	0	10	20	30	40	50	40	30	20	10	0



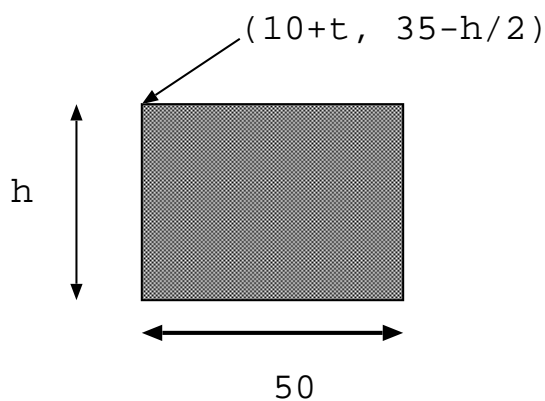


図 8.2.1: 描画する矩形

```
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` が時刻から座標を計算し、設定します。時刻  $t$  とともに、左下の頂点の位置が

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

で移動します。同時に高さが

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

```
if(h>50) h = 100 -h;
```

で大きくなります。こうすることで、各時刻で、 $x$  の位置に高さ  $h$  の長方形を描くことで、四角系が伸び縮みしながら移動しているように見せることができます。 $x$  座標と高さ  $h$  が時刻  $t$  とともに、どのように変化するかを表 8.1 に染まします。

**演習 8.1** スリープする時間を変更して、様子の変化を見なさい。

**演習 8.2** 三角形が伸び縮みしながら並行移動するプログラムを作成しなさい。

---

**Program 8.2.2** MovingSquare.java の続き

---

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

@Override
public void paint(Graphics gg) {
    super.paint(gg);
    Graphics g = drawPanel.getGraphics();
    g.setColor(color);
    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;
}
}
```

---

---

**Program 8.2.3** MovingSquare.java の続き

---

```
private void quitActionPerformed(java.awt.event.ActionEvent evt) {
    dispose();
}

private void startActionPerformed(java.awt.event.ActionEvent evt) {
    start();
}

private void stopActionPerformed(java.awt.event.ActionEvent evt) {
    stop();
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {

        @Override
        public void run() {
            new MovingSquare().setVisible(true);
        }
    });
}
```

---



## 第9章 クラスを作る

### 9.1 クラスとは

---

**Program 9.1.1** MovingSquareWithMyShape.java: NetBeans が自動生成する GUI の要素部分を省略している。

---

```
import java.awt.Color;
import java.awt.Graphics;

public class MovingSquareWithMyShape extends javax.swing.JFrame
    implements Runnable {

    private Thread runner;
    private int t = 0;
    private volatile boolean id = false;
    private Color color = Color.red;
    private MyShape shape;

    /** Creates new form Clock */
    public MovingSquareWithMyShape() {
        initComponents();
        shape = new MySquare(color);
    }

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

    public void stop() {
        id = false;
    }
}
```

---

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が重要です。このようにすることで、プログラムを読みやすくすることができます。また、その結果、開発コストを小さな小さくすることができます。

説明や報告を目的とした文章を書くときのことを考えましょう。文章も、章や節、段落を適切に立てることで、読みやすい文章を書くことができます。章や節、段落の組み立ての基本は、一つのそれぞれの文のグループ内では一つテーマを扱うことです。このように章や節に適切に分けられた

---

**Program 9.1.2** MovingSquareWithMyShape 続き

---

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

@Override
public void paint(Graphics gg) {
    super.paint(gg);
    Graphics g = drawPanel.getGraphics();
    shape.set(t);
    shape.draw(g);
}
```

---

文章は、手直しや再利用が容易になります。文章を書く基本とプログラムを書く基本は、いろいろなところでつながっています。

先週のプログラム `MovingSquare.java` を思い出しましょう。実際に動画を描いていたのは `paint` メソッドの中でした。そこでは、最初に全体を背景色で塗りつぶし、新しい矩形の情報を計算し、その図形を前景色で描きました。`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。

このプログラムを前述の「読みやすいプログラム」という観点から考えましょう。このプログラムでは、どういうタイミングで絵を再描画するかということと、実際にどのような絵が書かれるかが、一つのプログラム内に入っていました。別の絵を描くためには、メソッド `setsqr` だけでなく、矩形の情報を持っていた配列 `sqr` も書き換えなければなりませんでした。

そこで、絵を描くことと、描かれる絵を構成することを分けることを考えましょう。つまり、矩形の位置を計算して描く部分を次のような方向に書き換えることを考え、別の図形に置き換えることができるようにしましょう。

まず、矩形に対応するモノをひとかたまりとして考えます。メインのプログラムは、このモノに対して、「再設定せよ」、「描画せよ」と命令するだけにします。メインのプログラムの方からは、描かれる図形が矩形であるのか別の図形なのか、更に図形の情報がどのように保持されているかを知らなくて良いようにします。

このように、対象（モノ）の動作・操作を中心に考えて、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・類型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、そのクラスを実体化したオブジェクトを生成します。

**Program 9.1.3** MovingSquareWithMyShape 続き

---

```

private void quitActionPerformed(java.awt.event.ActionEvent evt) {
    dispose();
}

private void startActionPerformed(java.awt.event.ActionEvent evt) {
    start();
}

private void stopActionPerformed(java.awt.event.ActionEvent evt) {
    stop();
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {

        @Override
        public void run() {
            new MovingSquareWithMyShape().setVisible(true);
        }
    });
}
}

```

---

そのオブジェクトに「再設定せよ」、「描画せよ」と動作の命令・操作をするだけにするのです。

矩形のクラスをプログラム 9.1.4 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す。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 9.1.1 からは、クラス `MySquare` のオブジェクトが定義され、コンストラクタで前景色が指定されます。

```
sqr = new MySquare(color);
```

その後、`paint` メソッド内では、矩形クラスのメソッド `set` 及び `draw` が呼ばれているだけで、`set` 及び `draw` の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。

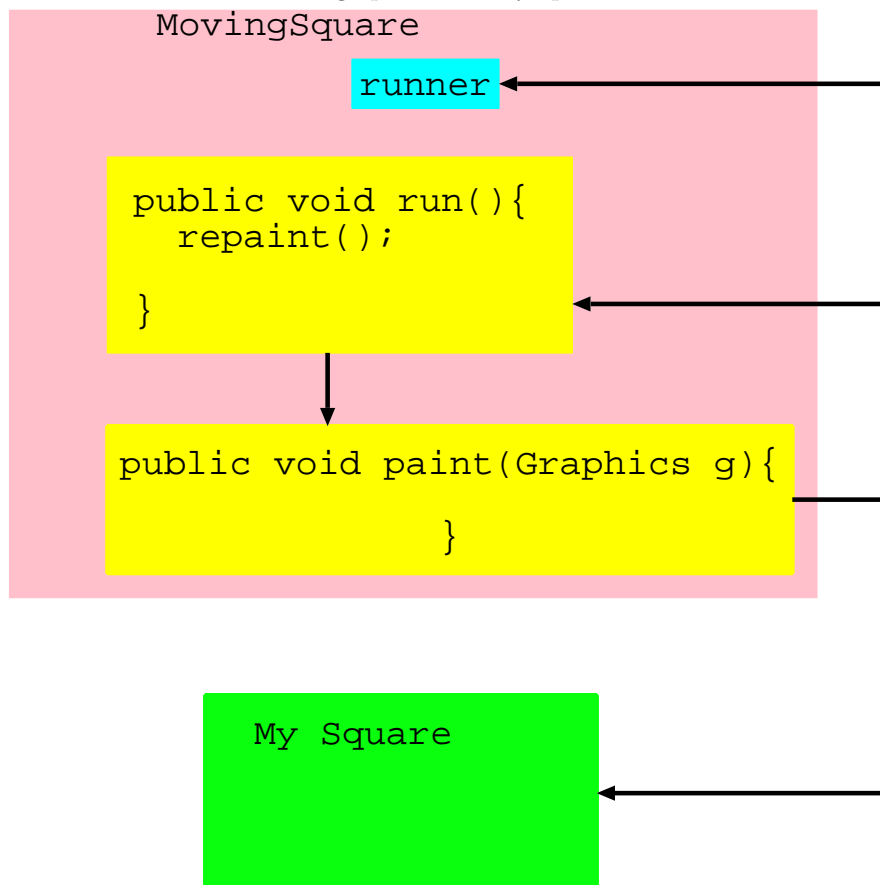
## 9.2 インターフェイス

プログラム 9.1.1 では、もうすこし一般化がされています。つまり、形のオブジェクトが

```
private Myshape shape;
```

と宣言されています。これに対応してプログラム 9.1.4 では

図 9.1.1: MovingSquare から MySquare を呼び出す



```
public class MySquare implements MyShape
```

となっています。

オブジェクト指向プログラミング言語では、すでに定義されているクラスを拡張して、新しいクラスを定義することができます。「クラスの拡張」あるいは「クラスの継承」といいます。つまり、ある程度一般的なクラスを定義しておく、後でそれを再利用できます。再利用の際には、プログラムを書き直すのではなく、機能などを追加したものを新しいクラスとして定義します。

`MySquare` の定義に現れる `MyShape` は、インターフェイスと呼ばれる特殊なクラスです。インターフェイスは、メソッドの名前と引数並びだけが定義されているクラスです。つまり、インターフェイスは、他のクラスから方法を、呼ばれるメソッド名だけで定義しているものです。

プログラム 9.2.1 を見てください。二つのメソッド `set()` と `draw()` が、その名前と引数だけ定義され、中身が書かれていません。一方、プログラム 9.1.4 では、このインターフェイスがインプリメント（実装）され、メソッドの中身が記述されています。メインのクラス 9.1.1 では、中身がどうなっているかを考えずに二つのメソッドを呼び出しています。



---

**Program 9.1.4** MySquare.java

---

```
import java.awt.*;

public class MySquare implements MyShape {
    private int x,y;
    private int w=50,h;
    private Color fore;

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

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

    @Override
    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);
    }
}
```

---

### 9.3 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。メインのプログラムは何が描かれるか知る必要はありません。つまり、メソッド `set` の中で何が計算されているかを知る必要がありません。

プログラム 9.3.1 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。インターフェイス `MyShape` をインプリメントしています。それに対応して、メソッド `set` の中で三角形の座標を計算しています。

三つの頂点の座標は、図 9.3.1 のように指定します。中心座標  $(x, y)$  はを考えます。 $x$  座標は時間とともに  $x = 10 + t$  で右に動きますが、 $y$  座標が変わりません。三角形の頂点の座標は、この中心からの相対座標として指定することができます。

**演習 9.1** 三角形が回転しながら移動するプログラムを作成しなさい。

**演習 9.2** 自分のオリジナル図形、例えば楕円形や多角形をクラスとして定義し、それが移動するプログラムを作成しなさい。

---

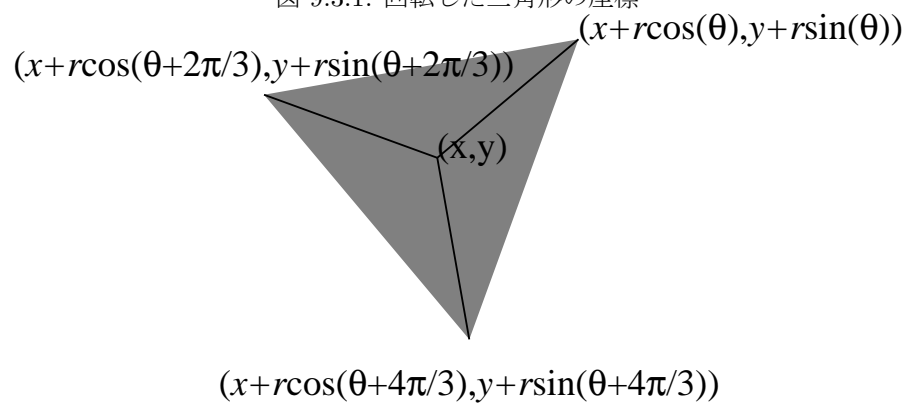
**Program 9.2.1** MyShape.java

---

```
public interface MyShape {  
    /**  
     * 時刻に応じて形を定義する  
     * @param t 時刻  
     */  
    public void set(int t);  
    /**  
     * Graphics に描画する  
     * @param g 描画すべき Graphics  
     */  
    public void draw(java.awt.Graphics g);  
}
```

---

図 9.3.1: 回転した三角形の座標



---

**Program 9.3.1** MyTriangle.java

---

```
import java.awt.*;

public class MyTriangle implements MyShape{
    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 fore;

    public MyTriangle(Color fore) {
        this.fore=fore;
    }

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

---



## 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** DrawPanel.java

---

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class DrawPanel extends javax.swing.JPanel
    implements MouseListener, MouseMotionListener {
    //矩形のリスト

    private List<MySquareWithCorners> sqrs = null;
    private MySquareWithCorners tmp;//作業中の新しい矩形

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

    public void initialize() {
        //マウスの動作を拾うインターフェイスの登録
        addMouseListener(this);
        addMouseMotionListener(this);
        //矩形のリストの初期化
        clearSquares();
    }

    public void clearSquares() {
        sqrs = Collections.synchronizedList(
            new ArrayList<MySquareWithCorners>());
        tmp = null;
    }
}
```

---

GUI(Graphical User Interface) のあるプログラムでは、マウスやキーボードによってプログラム内の機能の動作を停止・開始することができます。マウスやキーボードによって、プログラムに対して働きかけを行っていることとなります。マウスを押す、キーボードを操作するなどのイベント

---

**Program 10.1.2** DrawPanel.java つづき

---

```
@Override
public void paint(Graphics g) {
    super.paint(g);
    Color c = getForeground();
    for (MySquareWithCorners s : sqrs) { //既に確定した四角形の描画
        s.draw(g, c);
    }
    if (tmp != null) { //現在作成中の四角形
        tmp.draw(g, Color.red);
    }
}

@Override
public void mouseDragged(MouseEvent e) { //ボタンを押しながら移動した時
    //新しい矩形の右下隅の座標を更新する
    tmp.set(e.getPoint());
    repaint();
}

@Override
public void mousePressed(MouseEvent e) { //ボタンを押した時
    //新しい矩形を作り始める
    tmp = new MySquareWithCorners(e.getPoint());
}

@Override
public void mouseReleased(MouseEvent e) { //ボタンを離れた時
    //新しい矩形の右下隅の座標を更新する
    tmp.set(e.getPoint());
    //新しい矩形を矩形のリストに追加して確定する
    sqrs.add(new MySquareWithCorners(tmp));
    tmp = null;
    repaint();
}
```

---

トをきっかけに様々な動作させるようなプログラムの作りを、イベント駆動 (event-driven) 型のプログラムといいます。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画面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。ボタンなどの GUI の操作でもイベントが発生していました。

本章では、マウスの動きやマウスのボタン操作によって生じるイベントを使うプログラムを作成します。まず、マウスの動作を使って、矩形を描くプログラムを作成しましょう (図 10.1.1)。

まず、マウスイベントを拾うために、メインクラスにインターフェイスを追加します。インターフェイスとは、他のクラスからの呼び出しの形式を定めるものです。クラス定義に記述します。

```
implements MouseListener, MouseMotionListener
```

ここで、`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。このインターフェイスが定義されたことで、Java の

---

**Program 10.1.3** DrawPanel.java つづき: メソッド `initComponents()` は省略している。

---

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

@Override
public void mouseClicked(MouseEvent e) {}

@Override
public void mouseEntered(MouseEvent e) {}

@Override
public void mouseExited(MouseEvent e) {}

//initComponent() 省略
//GUI コンポーネント定義省略
}
```

---

VM から、マウスイベントが起こった際の呼び出し方が定義されます。

さらに、初期化の段階で、イベントを観測しておくのが、このクラスであることを定義します。別の言い方をすると、このクラスの表示された部分で起こったマウスのイベントを観測することを定義します。

```
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。以前に使用したインターフェイス `Runnable` の場合には、一つのメソッド `run` だけを記述しました。しかし、今回は使用するインターフェイスにはメソッドが複数あります。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。

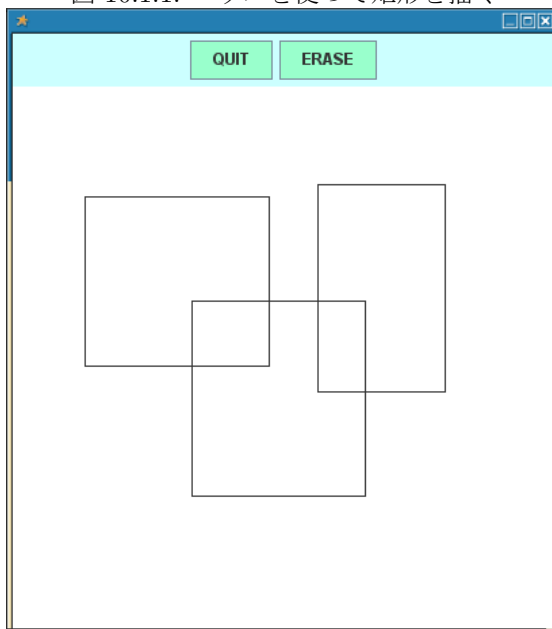
使用している統合環境 NetBeans は、追加したインターフェイスに対応したメソッドの追加を支援する機能を有しています。インターフェイスを追加した後に、「ソース」、「メソッドをオーバーライド」を選びます。新しく開いたウィンドウで、「抽象メソッドのみを表示」を選ぶと、インターフェイスに対応して、追加しなければならないメソッドの一覧が表示されます。追加すべきメソッド名をマウス右ボタンで選択した後に、「了解」を押すと、ソースコードが生成されます。

今回のプログラムにおいては、`MouseListener` と `MouseMotionListener` 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。一方、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

```
public void
```

として定義されます。また、引数は `MouseEvent` 型です。

図 10.1.1: マウスを使って矩形を描く



## 10.2 マウスの動きに応じて矩形を描く

プログラム 10.1.1 での動きを見て行きます。

マウスを押すことで (`mousePressed`) 新しい矩形の描画を開始します。マウスを押しながら移動して (`mouseDragged`) いる間は、その矩形の右下の座標が更新されます。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。

矩形の情報は、次節で説明するクラス `MySquareWithCorners` に保存します。マウスボタンを押して移動中には、一時的な矩形 `tmp` として扱われ、描画を行います。マウスボタンを放すと矩形が確定したと判断し、`tmp` にある矩形データ、保存用のリスト `sqr`s に登録します。

マウスを押したイベントが発生すると、そのイベント情報を `MouseEvent` クラスのオブジェクトとして得ることができます。マウスが押された位置は `MouseEvent` クラスのメソッド `getPoint()` を使って得ることができます。

マウスには、複数のボタンがついていますが、どのボタンを押されたかの区別はできません。なぜならば、Java は OS によらずに動作することを重要な性質として設計されており、一方、マウスにいくつのボタンがあるかは、OS に依存するからです。この点は、Windows 上の GUI アプリケーションや、UNIX で使われている X Windows 上の GUI アプリケーションと大きく異なる点です。

各マウスイベントの処理が終了と `repaint()` で再描画をします。描画 (`paint()`) の際には、まず確定している矩形 (配列 `sqr`s 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
<code>mouseClicked</code>	この部分にマウスがクリックされると呼び出される
<code>mouseEntered</code>	この部分にマウスが入ると呼び出される
<code>mouseExited</code>	この部分からマウスが出ると呼び出される
<code>mousePressed</code>	この部分でマウスボタンが押されると呼び出される
<code>mouseReleased</code>	この部分でマウスボタンが放されると呼び出される
<code>mouseDragged</code>	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
<code>mouseMoved</code>	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

## 10.3 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。今回使用している矩形クラス `MySquareWithCorners` 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。

コンストラクタ

```
MySquareWithCorners(final MySquareWithCorners s)
```

は、他の矩形をコピーするのに用います。引数が自分と同じクラスの別のオブジェクトになっていることに注意してください。キーワード `final` は、その中身を変更しないことを宣言しています。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。このように、自分と同じクラスをコピーして新しいオブジェクトを作るコンストラクタをコピーコンストラクタと呼びます。

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

## 10.4 矩形を List に保存する

このプログラムの中では、矩形をたくさん保存します。同じ型のものをたくさん保存する方法の一つは配列を使うことです。しかし、配列を使う場合、その大きさを予め指定しなくてはなりません。また、途中の要素を削除してしまった場合には、前に詰めるなどの処理が必要になります。今回のようなプログラムでは取扱が繁雑になりそうです。

そこで、要素が一行に並んだもの、つまりリストを扱うクラス `java.util.List` を使うことにします。List を使うことで、要素の数の制限が無くなるとともに、途中の要素を削除することが容易にできるようになります。クラス List は Java が有している Collection と呼ばれる汎用的クラスの一つです。Collection は、要素としてクラスのオブジェクトを持つことができます。保存される基本のクラスは 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。つまり、List には、どんなオブジェクトでも保存することができます。

**Program 10.3.1** MySquareWithCorners.java

---

```

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Point;

public class MySquareWithCorners{
    private Point start,end;//左上隅と右下隅の点
    private int w,h;//幅と高さ

    /**
     * Creates a new instance of MySquareWithCorners
     */
    public MySquareWithCorners(Point p) {
        start = new Point(p);
    }

    /** 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 MySquareWithCorners(final MySquareWithCorners s){
        this(s.start);
        set(s.end);
    }

    /**
     * 右下隅の設定
     * @param p 右下隅の座標
     */
    public void set(Point p){
        end = new Point(p);
        w = end.x - start.x;
        h = end.y - start.y;
    }
}

```

---

**List** にどんなオブジェクトでも保存できるということは、便利なようですが、困った状況が発生する原因でもあります。**List** に想定外のを保存してしまったり、その結果としてそこから取り出したモノが何だか予想ができない状況が発生します。

普通は、プログラムの作成者は、**List** に保存するオブジェクトの種類を知っています。誤って異なる種類のオブジェクトを保存してしまわないほうが良いでしょう。つまり、どういうクラスに属するオブジェクトを**List** の要素として持つことができるかを定めることで、コンパイル段階で誤りを見付け、プログラムの安全性を高めることができます。

矩形の一覧を保持する変数 **sqr**s に保存するのは、クラス **MySquareWithCorners** のオブジェクトだけです。そこで、

```
private List<MySquareWithCorners> sqrList=null;
```

と定義します。ここで **List<MySquareWithCorners>** という表記は、このリストの中に要素としてクラス **MySquareWithCorners** のオブジェクトだけを持つことができることを示しています。ここに現れた<クラス名>を型パラメタと呼びます。

---

**Program 10.3.2** MySquareWithCorners.java

---

```
/**
 * 図形を描く
 * @param g Graphics 属性
 * @param c 色
 */
public void draw(Graphics g,Color c){
    g.setColor(c);
    g.drawRect(start.x,start.y,w,h);
}

/**
 * 与えられた点が図形の内側かを判定する
 * @param p 指定した点
 * @return 指定した点がこの図形の内側ならば真
 */
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;
}

/**
 * 右下隅の座標を得る
 * @return 右下隅の座標
 */
public Point getEnd() {return end;}
}
```

---

実は、List はクラスではなく、インターフェイスです。そのため、以下のように実体として `java.util.ArrayList` を使って初期化します。

```
sqrs = Collections.synchronizedList(
    new ArrayList<MySquareWithCorners>());
```

Java はスレッドを使う環境であることは以前に紹介しました。クラス `ArrayList` はスレッド間で同期することができません。 `Collections.synchronizedList` メソッドを使うことで、同期の保護を行っています。

クラス List への要素の追加は `add` メソッドで行います。上記のように保存するオブジェクトのクラスを指定することで、コンパイル時に、正しい種類のオブジェクトが保存されていることを確認することができます。異なるクラスのオブジェクトを `sqrs` に追加しようとすると、コンパイル時にエラーが発生します。

クラス List から要素を得るには `get` メソッドを使います。保存されているオブジェクトのクラ

スが指定されていますので、戻り値が `MySquareWithCorners` であることが分かります。

クラス `List` から要素を削除するには、`remove` を使います。

クラス `List` に保存されている全ての要素に、何かの操作を行う際には

```
for(MySquareWithCorners s:sqrs){//既に確定した四角形の描画
    s.draw(g,c);
}
```

のように書くことができます。ここではリスト `sqrs` の各要素 `s` について、`draw()` メソッドを実行しています。

**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。

## 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 DrawSquareWithKeyEvents.java

ボタン操作に関する部分のみを載せている

---

```
private void quitActionPerformed(java.awt.event.ActionEvent evt) {
    System.exit(0);
}

private void eraseActionPerformed(java.awt.event.ActionEvent evt) {
    drawPanel.clearSquares();
}

```

---

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

今回は、前回のマウスイベントを使って矩形を描く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 Java で処理されます。まず、マウスイベント処理の場合と同様に、クラスに `KeyListener` というインターフェイスを `implements` します。更に、コンストラクタで

```
addKeyListener(this);
```

をつかって、キーボードイベントを聞くようにします。

マウスは、画面上の位置があり、どのウィンドウを操作しているかを指定できます。しかし、キーボードには画面上の位置がありません。X Windows 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);
```

---

**Program 11.1.2** DrawPanel.java

---

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class DrawPanel extends javax.swing.JPanel
    implements MouseListener, MouseMotionListener, KeyListener {
    //矩形のリスト

    private List<MySquareWithCorners> sqrs = null;
    private MySquareWithCorners tmp;//作業中の新しい矩形
    private MySquareWithCorners selected=null;//作業中の新しい矩形
    private Point present = new Point(0, 0);//マウスポインタの位置

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

    public void initialize() {
        //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);
        //矩形のリストの初期化
        clearSquares();
    }
}
```

---

インターフェイス `KeyListener` によって定義されているキーボードイベントを扱う関数は3種類です。表 11.1 に示します。マウスと同様に、全て定義しなくてはなりません。引数は全て `KeyEvent` 型です。

実際にキーが押された時の操作をメソッド `keyPressed` の中で記述しています。キー `x` が押されると、選択されている矩形が消去されます。通常の文字に対応するキーが押された場合、そのキーの値は `KeyEvent.getKeyChar()` で調べることができます。

一方、カーソルキーなど、文字に対応していないキーの処理は、`KeyEvent.getKeyCode()` でコードを調べて行います。クラス `KeyEvent` 内で、文字コードに対応した定数が定義されています。ここでは、カーソルキーによって、矩形の右下隅の座標が増減できるようにしています。

---

**Program 11.1.3** DrawPanel.java つづき

---

```
public void clearSquares() {
    sqrs = Collections.synchronizedList(
        new ArrayList<MySquareWithCorners>());
    tmp = null;
    repaint();
}

@Override
public void paint(Graphics g) {
    if (sqrs == null) {
        return;
    }
    super.paint(g);
    Color c = getForeground();
    for (MySquareWithCorners s : sqrs) { //既に確定した四角形の描画
        Color cc = c;
        if (s.equals(selected)) {
            cc = Color.blue; //選択した矩形の色を青に
        }
        s.draw(g, cc);
    }
    if (tmp != null) { //現在作成中の四角形
        tmp.draw(g, Color.red);
    }
    //マウスポインタの位置座標を描く
    g.setColor(c);
    String coor = "(" + String.valueOf(present.x)
        + "," + String.valueOf(present.y) + ")";
    g.drawString(coor, 10, 10);
}
```

---

## 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、マウスが動くたびに、`mouseMoved()` メソッドを使って、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。このようにしておくと、プログラムの動作の確認が容易になります。また、このマウスポインタが入っている矩形を `selected` としてマークしておきます。

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しまっていました。今回のプログラムでは、`mouseReleases()` メソッドの中で、矩形の大きさを計り、マウスのクリックによって作成されてしまう大きさゼロの矩形を破棄するように改良しました。

大きさゼロの矩形が保存されないようにする一方で、マウスのクリックを矩形の選択に使うようにしました。つまり、マウスをクリックすると、マウス位置を含む矩形を選択します。マウスクリックに対応する `mouseClicked()` メソッドから矩形の `isInside()` メソッドを呼ぶことで、どの矩形が選択されたかを調べます。選択された矩形を赤で表示することにします。

---

**Program 11.1.4** DrawPanel.java つづき

---

```
@Override
public void mouseDragged(MouseEvent e) { // ボタンを押しながら移動した時
    // 新しい矩形の右下隅の座標を更新する
    tmp.set(e.getPoint());
    repaint();
}

@Override
public void mousePressed(MouseEvent e) { // ボタンを押した時
    // 新しい矩形を作り始める
    tmp = new MySquareWithCorners(e.getPoint());
}

@Override
public void mouseReleased(MouseEvent e) { // ボタンを離れた時
    // 新しい矩形の右下隅の座標を更新する
    tmp.set(e.getPoint());
    if (tmp.getWidth() != 0 || tmp.getHeight() != 0) { // 大きさゼロの矩形は破棄する
        sqrs.add(tmp);
    }
    tmp = null;
    selected = null;
    repaint();
}

@Override
public void mouseMoved(MouseEvent e) {
    present.setLocation(e.getPoint());
    if (sqrs == null) {
        return;
    }
    for (MySquareWithCorners s : sqrs) { // マウスが入っている矩形を探す
        if (s.isInside(present)) {
            selected = s;
        }
    }
    repaint();
}

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

@Override
public void mouseEntered(MouseEvent e) {
    requestFocus();
}

@Override
public void mouseExited(MouseEvent e) {
}
```

---



---

**Program 11.1.5** DrawPanel.java つづき

---

```
@Override
public void keyPressed(KeyEvent e) {
    if (e.getKeyChar() == 'x') { // 'x' を押した時の動作
        if (selected != null) {
            sqrs.remove(selected);
            selected = null;
        }
    }
    if (selected != null) {
        Point end = selected.getEnd();
        switch (e.getKeyCode()) { // カーソルキーを押した場合
            case KeyEvent.VK_UP:
                end.y--;
                break;
            case KeyEvent.VK_DOWN:
                end.y++;
                break;
            case KeyEvent.VK_LEFT:
                end.x--;
                break;
            case KeyEvent.VK_RIGHT:
                end.x++;
                break;
            default:
                break;
        }
        selected.set(end);
    }
    repaint();
}

@Override
public void keyReleased(KeyEvent e) {
}

@Override
public void keyTyped(KeyEvent e) {
}

/** 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.
 */
@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code"> // GEN-BEGIN: initComponents
private void initComponents() {
    // 省略
} // </editor-fold> // GEN-END: initComponents
// Variables declaration - do not modify // GEN-BEGIN: variables
// End of variables declaration // GEN-END: variables
}
```

---



## 第12章 マウスを使った簡単なゲーム

### 12.1 15ゲーム

マウスイベントを使ったプログラムの応用例として、15ゲームを作りましょう。15ゲームはとてもポピュラーなゲーム(図 12.1.1)です。16個のマスのに、15個の数字が書かれたピースがでたらめにならんでいます。一つの空いているマスを使いながら、この15個のピースを数字の順に並べかえるゲームです。

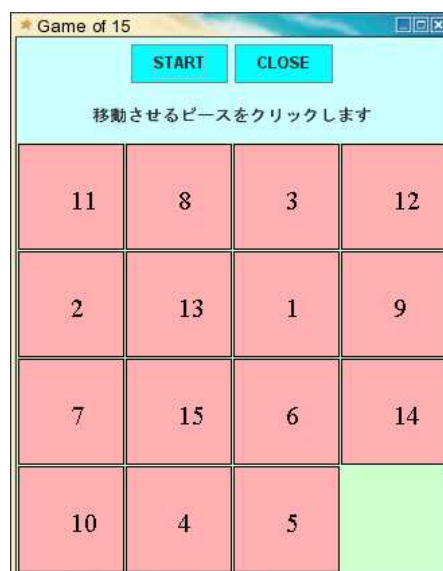


図 12.1.1: 15 ゲーム

ゲームのプログラムを書く際には、まず、ゲームのルールを整理する必要があります。15ゲームの場合は、ピースの動かし方をプログラムに書ける精度まで詳しく表すことが必要です。

マウスで選んだピースの位置を  $(x, y)$  で表すことにします。  $x$  と  $y$  は、それぞれ 0 から 3 までの数字です。横方向は左端を 0 とします。縦方向は上端を 0 とします。このピースの隣が空いていれば、ピースを空きに移動します。それ以外の場合には、ピースを動かすことはできません。

ピースの無い空きマスの座標を  $(x', y')$  としましょう。マウスで選んだピースの座標を  $(x, y)$  とピースの無い空きマスの座標が隣接しているときにだけピースを動かすことができます。この次の条件で表されます。

$$(|x - x'| = 1 \wedge y = y') \vee (x = x' \wedge |y - y'| = 1) \quad (12.1.1)$$

**Program 12.1.1** Piece.java

---

```

import java.awt.*;

public class Piece {

    private Dimension dimension;
    private Point offset;
    private Rectangle rect;
    private int k = 0;
    private Point p = null;
    private Font font;
    private int fontsize = 20;

    /**
     * Creates a new instance of GamePanel
     * @param k ピースの番号
     * @param dimension ピースの大きさ
     */
    public Piece(int k, Dimension dimension) {
        font = new Font("TimesRoman", Font.PLAIN, fontsize);
        this.k = k;
        this.dimension = dimension;
        p = new Point(0, 0);
        offset = new Point(1, 1);
        rect = new Rectangle(offset.x, offset.y,
            dimension.width - 2 * offset.x,
            dimension.height - 2 * offset.y);
    }

    /**
     * ピースの番号を返す
     * @return ピースの番号
     */
    public int getValue() {
        return k;
    }
}

```

---

15 ゲームの終りは、番号順にピースが並んだときです。一般の 15 ゲームでは、左上隅から右方向に数字を並べたり、下方向に並べたりと様々なゲームの終了を設定できます。しかし、ここでは、以下のように並んだところで終りにしましょう。左上隅に 1、そこから右に 4 まで、次の段の左から 5 から 8 まで、という順序に並んだところが終りとします。

左上隅の座標は (0,0) です。一番上の左から右へは、(0,0)、(1,0)、(1,0) そして (3,0) と  $x$  座標だけがが増えて行きます。二段目の左端は (0,1) で、再び  $x$  座標だけがが増えて、二段目右端は (3,1) となります。つまり、全てのマスに、その座標  $(x,y)$  に対応して

$$4 * y + x + 1 \tag{12.1.2}$$

の数字のピースがあればゲームが終了になります。

## 12.2 ピースのクラス

まず、各ピースを表すクラス `Piece` を作ります。このクラスは、ピースの番号 `k` と置かれている位置 `p` を保持しています。ピースの番号に 1 加えたものが、そのピースの表面に書かれている番号とします。

ピースの大きさは `Dimension` クラスの `dimension` で指定されています。従って、クラス `Point` で表される位置 `p` はこのピースの左上隅の画面上の座標を表します。位置 `p` は、前節で使った位置ではなく、画面の座標であることに注意してください。つまり、前節で使った座標  $(x, y)$  にあるピースは、

$$p.x = \text{dimension.width} \times x \quad (12.2.1)$$

$$p.y = \text{dimension.height} \times y \quad (12.2.2)$$

に置かれています。

コンストラクタでは、ピースの番号と大きさを指定します。メソッド `setPoint()` でピースの置かれた位置を指定します。この時に、ピースとして描く矩形 `rect` の位置を移動しています。

ピースを実際に描くのは `draw()` メソッドです。矩形の中を塗りつぶし (`g.fill(rect)`)、後で矩形の枠を描いています (`g.draw(rect)`)。最後に数字を描きます。

ピースを選択するメソッドが `isSelected()` です。引数で指定された座標 `pp` がピースの内部であれば、`true` を、そうでなければ `false` を返します。

## 12.3 ゲーム全体のクラス

---

### Program 12.3.2 Game.java 続き

---

```
//0 から m までの数字をでたために並べ直す
private int[] mkRandomInt(int m) {
    int ids[] = new int[m];
    ArrayList<Integer> v = new ArrayList<Integer>();
    for (int i = 0; i < m; i++) {
        v.add((Integer) i);
    }
    int mm = 2 * (m / 2);
    int i = 0;
    int len = v.size();
    while (i < mm) {
        int k = (int) (len * Math.random());
        int l = (int) (len * Math.random());
        if (k != l) {
            int xk = v.get(k);
            int xl = v.get(l);
            v.set(k, xl);
            v.set(l, xk);
            i++;
        }
    }

    for (Integer x : v) {
        ids[x] = v.get(x);
    }
    return ids;
}

/**
 * ピースを描画する
 * @param g グラフィックス
 */
@Override
public void paint(Graphics g) {
    if (pieces == null) {
        return;
    }
    super.paint(g);
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < n; i++) {
            if (pieces[i][j] != null) {
                pieces[i][j].draw(g);
            }
        }
    }
}
}
```

---

ゲーム全体を表すクラスとして、`Game` を作ります。ゲームは、15 個のピースを描く `Panel` として作ることにします。これを後で、`JApplet` にはめ込むことにします。

ピースを生成して、でたらめな順番で並べるのが `initialize()` メソッドです。0 から 14 までの数字をでたらめに並べるメソッド `mkRandomInt()` を使っています。このメソッドの詳細は省略します。生成されたピースは  $4 \times 4$  の配列 `pieces` に保存されています。配列 `pieces` のインデックスは位置  $(x, y)$  ( $x = 0, 1, 2, 3, y = 0, 1, 2, 3$ ) を表します。

このゲームパネルの大きさは  $320 \times 320$  とします。従って、一つ一つのピースは  $80 \times 80$  になります。

ピースの選択は、マウスクリックで行います。マウスクリックが発生したときの処理は `mouseClicked()` メソッドに記述します。まず、マウスイベントから、マウスクリックが発生した座標を読みだします。

```
p=new Point(e.getPoint());
```

座標は `Point` 型に保存します。この座標 `p` を内部とするピースを捜し出します。その座標が `x` と `y` になります。同時に、ピースの無いマスの座標 `xx` と `yy` も探します。これら座標は、0 から 3 までの数字です。

選択したピースの位置と、ピースの無いマスの位置から、選択したピースが動けるか否かを判定します。動ける場合には、ピースの無い場所にピースを移動します。

ゲームの終了判定は、`isFinished()` メソッドで行います。

---

**Program 12.1.2** Piece.java 続き

---

```
/**
 * ピースの場所を設定する
 * @param pp ピースの場所
 */
public void setPoint(Point pp) {
    int dx = pp.x - p.x;
    int dy = pp.y - p.y;
    p = pp;
    rect.translate(dx, dy);
}

/**
 * このピースが選択されたかを判定
 * @param pp マウスの位置
 * @return このピースが設定されていれば true
 */
public boolean isSelect(Point pp) {
    boolean b = false;
    if ((pp.x > p.x && pp.x < p.x + dimension.width) &&
        (pp.y > p.y && pp.y < p.y + dimension.height)) {
        b = true;
    }
    return b;
}

/**
 * このピースを描く
 * @param gg 使用する Graphics
 */
public void draw(Graphics gg) {
    Graphics2D g = (Graphics2D) gg;
    g.setColor(Color.pink);
    g.fill(rect);
    g.setColor(Color.BLACK);
    g.draw(rect);
    g.setFont(font);
    g.drawString(String.valueOf(k + 1),
        p.x + dimension.width / 2,
        p.y + dimension.height / 2 + fontsize / 2);
}
}
}
```

---



---

**Program 12.3.1** Game.java (initComponent は省略している)

---

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

public class Game extends javax.swing.JPanel
    implements MouseListener //マウスの動作を拾うためのインターフェイスの定義
{

    private Piece pieces[][]; //各ピース
    private int n = 4;        //板のサイズは 4 x 4
    private Dimension d;     //一つのピースの大きさ
    private Point p = null;
    private Font font;

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

    /**
     * ピースを並べる
     */
    public void initialize() {
        //マウスの動作を拾うインターフェイスの登録
        addMouseListener(this);
        d = new Dimension(80, 80);
        font = new Font("TimesRoman", Font.PLAIN, 20);
        setBackground(new java.awt.Color(204, 255, 204));
        int ids[] = mkRandomInt(n * n - 1);
        pieces = new Piece[n][n];
        for (int i = 0; i < n * n - 1; i++) {
            int x = i % n;
            int y = i / n;
            pieces[x][y] = new Piece(ids[i], d);
            pieces[x][y].setPoint(new Point(x * d.width, y * d.height));
        }
        repaint();
    }
}
```

---

---

**Program 12.3.3** Game.java 続き

---

```
/**
 * マウスがクリックされた時の動作
 * @param e
 */
@Override
public void mouseClicked(MouseEvent e) {
    p = new Point(e.getPoint());
    int x = -1;
    int y = -1;
    int xx = 0;
    int yy = 0;
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < n; i++) {
            if (pieces[i][j] != null) { //マウスで選択したピースを見つける
                if (pieces[i][j].isSelect(p)) {
                    x = i;
                    y = j;
                }
            } else {
                xx = i;
                yy = j; //ピースの無いマスの座標
            }
        }
    }
    if (x >= 0 && y >= 0) {
        if ((Math.abs(x - xx) == 1 && Math.abs(y - yy) == 0)
            || (Math.abs(x - xx) == 0 && Math.abs(y - yy) == 1)) {
            pieces[xx][yy] = pieces[x][y];
            pieces[x][y] = null;
            pieces[xx][yy].setPoint(new Point(xx * d.width, yy * d.height));
        }
    }
    if (isFinished()) {
        setBackground(Color.blue);
    }
    repaint();
}
```

---

---

**Program 12.3.4** Game.java 続き

---

```
//ゲーム終了の判定
private boolean isFinished() {
    boolean b = true;
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < n; i++) {
            int k = n * j + i;
            if (pieces[i][j] != null) {
                if (k != pieces[i][j].getValue()) {
                    b = false;
                }
            }
        }
    }
    return b;
}

@Override
public void mouseReleased(MouseEvent e) {
}

@Override
public void mousePressed(MouseEvent e) {
}

@Override
public void mouseExited(MouseEvent e) {
}

@Override
public void mouseEntered(MouseEvent e) {
}
}
```

---



## 第13章 再帰的関数

### 13.1 Sierpinski クラス

---

**Program 13.1.1** Sierpinski.java initComponents() は省略している

---

```
import java.awt.*;
import java.awt.geom.Point2D;

public class Sierpinski extends javax.swing.JPanel {

    private Image image = null;
    private int t = 0;
    private double l = 250.;
    Point2D.Double points[];
    private Dimension imageSize;

    /** Creates new form Sierpinski */
    public Sierpinski() {
        initComponents();
        imageSize = new Dimension(300, 300);
    }
}
```

---

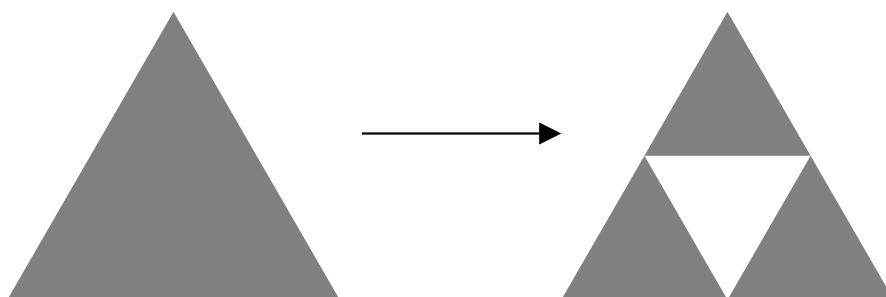


図 13.1.1: Sierpinski Gasket の一回の操作

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。フラクタル図形は、私達の身の回りにたくさんあります。海岸線の形、葉脈、雲の形などです。これらの特徴は、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ということです。例えば、海岸線は、様々な縮尺の地図で、同じような形に見えます。自然界には、様々なところでこのような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、その生成機構の解

**Program 13.1.2** Sierpinski.java 続き

---

```

/** 描画イメージの初期化 */
public void init_image() {
    //イメージの初期化
    image = 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 Point2D.Double[3];
    points[0] = new Point2D.Double(20., 20.);
    points[1] = new Point2D.Double(20 + 1, 20.);
    points[2] = new Point2D.Double(
        20 + 1 / 2., 20. + 0.5 * 1 * Math.sqrt(3.));
}

/**
 * 実際にイメージを作成する
 */
public void update_state() {
    init_image();
    update_state_sub(points, 0);
    repaint();
}

```

---

明は、20世紀の後半から、様々な研究分野で注目されています。

Sierpinski Gasket は、簡単な数学的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。この正三角形の各辺の中点を結ぶ、中心に正三角形をつくります。この中心の三角形をもとの正三角形からくり抜きます。3個の一辺の長さ  $L/2$  の正三角形が残ります。

残ったそれぞれの正三角形について、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。この操作を無限回繰り返した図形を Sierpinski Gasket と呼びます。

一回のくり抜き操作のたびに、普通の意味での面積が  $3/4$  になります。従って、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、すかさず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  $2^{-n}L$  の三角形を一つ取り出します。すると、その真中に一辺の長さ  $2^{-n-1}L$  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。この様子は、どんな  $n$  を選んでも変わりません。つまり、最初に見始めた長さ  $2^{-n}L$  によりません。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)、どのようなスケールでも変わる事がないこと、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。再帰的関数の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。これはまた、

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

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。

このようにある関数が、その関数自身を用いて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。

## 13.2 Sierpinski クラス

Sierpinski Gasket を描くクラスのプログラムをプログラム 13.1.1 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。

メインのプログラムをプログラム 13.2.1 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 13.1.3** Sierpinski.java 続き

---

```
/**
 * 再帰的に作図する
 * @param p 外側の三角形の頂点
 * @param n 再帰の回数
 */
private void update_state_sub(Point2D.Double p[], int n) {
    if (n < t) {
        //内部の3個の正三角形を再帰的に呼ぶ
        subTriangle(p[0], mPoint(p[0], p[1]), mPoint(p[0], p[2]), n);
        subTriangle(mPoint(p[0], p[1]), p[1], mPoint(p[1], p[2]), n);
        subTriangle(mPoint(p[0], p[2]), mPoint(p[1], p[2]), p[2], n);
    } else {
        //実際に三角形を描く
        Graphics g = image.getGraphics();
        drawTriangle(getForeground(), (Graphics2D) g, p);
    }
}

/**
 * 内部の三角形
 * @param t 上の頂点
 * @param l 左の頂点
 * @param r 右の頂点
 * @param n 再帰回数
 */
private void subTriangle(
    Point2D.Double t, Point2D.Double l, Point2D.Double r, int n) {
    Point2D.Double p[] = new Point2D.Double[3];
    p[0] = new Point2D.Double(t.x, t.y);
    p[1] = new Point2D.Double(l.x, l.y);
    p[2] = new Point2D.Double(r.x, r.y);
    update_state_sub(p, n + 1); //再帰呼び出し
}

/**
 * 中点を求める
 * @param p 頂点 1
 * @param q 頂点 2
 * @return 二つの頂点の中点
 */
private Point2D.Double mPoint(Point2D.Double p, Point2D.Double q) {
    return new Point2D.Double((p.x + q.x) / 2, (p.y + q.y) / 2);
}

```

---



---

**Program 13.1.4** Sierpinski.java 続き

---

```
/**
 * 初期化
 * @param t 繰り返し回数
 */
public void setInit(
    int t) {
    this.t = t;
    update_state();
}

/**
 * 描画
 * このメソッドで実際に図形を作らない。単にイメージを表示することに注意。
 */
@Override
public void paint(Graphics g) {
    super.paint(g);
    if (image == null) {
        return;
    }
    g.drawImage(image, 0, 0, imageSize.width, imageSize.height, this);
}

/**
 * 一つの正三角形を作る
 * @param g グラフィック
 * @param p 頂点
 */
private void drawTriangle(
    Color cl, Graphics2D g, 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);
}
}
```

---

---

**Program 13.2.1** SierpinskiGasket.java nitComponents() は省略している

---

```
public class SierpinskiGasket extends javax.swing.JFrame {

    /** Creates new form SierpinskiGasket */
    public SierpinskiGasket() {
        initComponents();
        for (int i = 0; i < 10; i++) {
            repeat.addItem(String.valueOf(i));
        }
        sierpinski.setInit(0);
        pack();
    }

    private void repeatActionPerformed(java.awt.event.ActionEvent evt) {
        int t = repeat.getSelectedIndex();
        sierpinski.setInit(t);
        repaint();
    }
}
```

---

## 第14章 階層化されたグラフィカルユーザインターフェイス

### 14.1 スライダー

---

**Program 14.1.1** SliderSample.java initComponents() は省略している

---

```
public class SliderSample extends javax.swing.JApplet
    implements javax.swing.event.ChangeListener{

    /** Initializes the applet SliderSample */
    public void init() {
        try {
            java.awt.EventQueue.invokeLater(new Runnable() {
                public void run() {
                    initComponents();
                }
            });
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        //slider の変換のイベントを登録する
        slider.addChangeListener(this);
    }

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

---

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、マウスでスライダ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。

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に、インターフェイス `javax.swing.event.ChangeListener` をアプレットに追加します。次に、スライダーのインスタンス `slider`—に対して、インターフェイスを登録します。

```
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` で記述します。

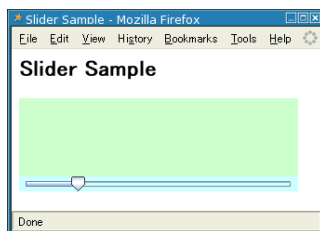


図 14.1.1: JSlider の例

```
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 段階です。

これは、NetBeans の GUI から行う事ができます。次の節で、スライダーの組の状態変化を調べる必要があります。そこで、GUI を使わずにスライダーの扱いをまとめておきました。

## 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。

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。

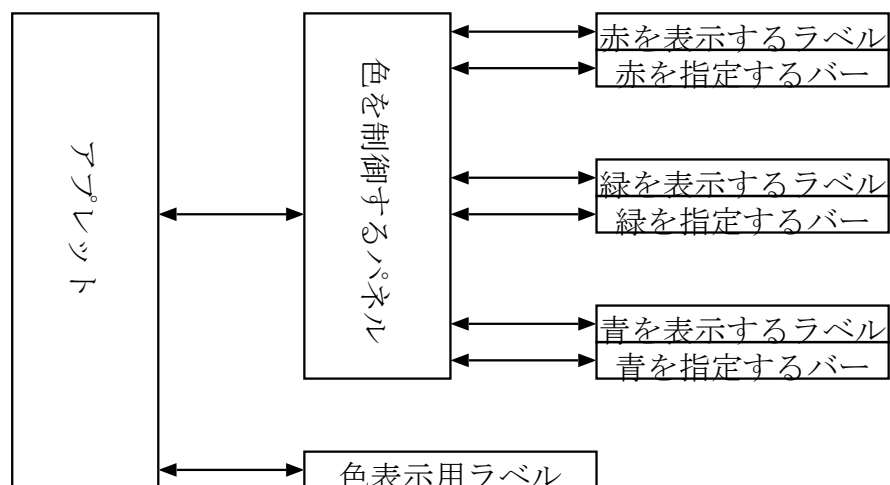


図 14.2.1: RGB カラーを制御するためのシステム概要

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;
}

@Override
public String toString() {
    return rgbvalue;
}

```

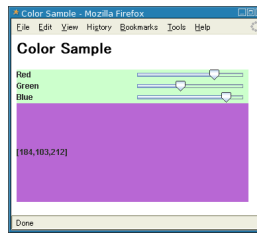


図 14.3.1: ColorSample

### 14.3 RGBColor を使う

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。メインのアプリケーションをプログラム 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 initComponents() は省略している

---

```
import javax.swing.event.ChangeListener;
public class RGBColor extends javax.swing.JPanel {

    private java.awt.Color rgb;
    private String rgbvalue;
    private ChangeListener listener;

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

    private void colorStateChanged(javax.swing.event.ChangeEvent evt) {
        rgb = new java.awt.Color(r.getValue(), g.getValue(), b.getValue());
        rgbvalue = "[" + String.valueOf(r.getValue()) + "," +
            String.valueOf(g.getValue()) + "," +
            String.valueOf(b.getValue()) + "]" + "#" +
            Integer.toHexString(r.getValue()) +
            Integer.toHexString(g.getValue()) +
            Integer.toHexString(b.getValue()) +
            ";";
        if (listener != null) {
            listener.stateChanged(evt);
        }
    }

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

    public java.awt.Color getrgb() {
        return rgb;
    }

    @Override
    public String toString() {
        return rgbvalue;
    }
}
```

---

---

**Program 14.3.1** ColorSample.java initComponents() は省略している

---

```
public class ColorSample extends javax.swing.JApplet
    implements javax.swing.event.ChangeListener{
    /** Initializes the applet ColorSample */
    @Override
    public void init() {
        try {
            java.awt.EventQueue.invokeAndWait(new Runnable() {
                @Override
                public void run() {
                    initComponents();
                }
            });
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        rgbColor.addChangeListener(this);
    }

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

---



## 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。

$$\left( \frac{000}{0}, \frac{001}{0}, \frac{010}{0}, \frac{011}{1}, \frac{100}{1}, \frac{101}{1}, \frac{110}{0}, \frac{111}{1} \right) \quad (15.1.3)$$

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。

---

### Program 15.2.1 CAdef.java

---

```
import java.awt.*;

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;
    }
}
```

---

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実行するようにします。実行例を図 15.3.1 に示す。

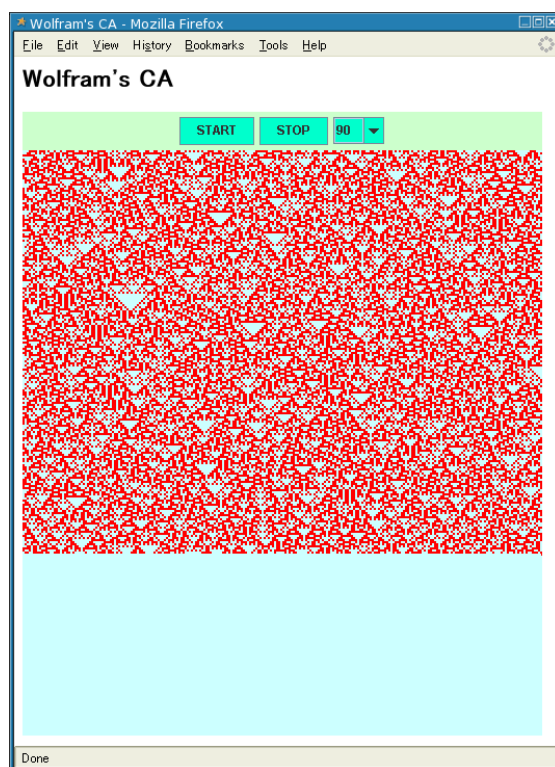


図 15.3.1: Wolfram のセルオートマトン

---

**Program 15.2.2** 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() {
    setLayout(new java.awt.BorderLayout());

    setBackground(new java.awt.Color(204, 255, 255));
    setForeground(new java.awt.Color(255, 0, 0));
}
}

```

---

---

**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() {
        try {
            java.awt.EventQueue.invokeAndWait(new Runnable() {
                public void run() {
                    initComponents();
                }
            });
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        ca=new CAdef();
        ca.setVisible(true);
        getContentPane().add(ca, java.awt.BorderLayout.CENTER);
        MkRules();
    }
}
```

---

---

**Program 15.3.2** WolframCA.java 続き

---

```
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();
}

private void ruleChoiceActionPerformed(java.awt.event.ActionEvent evt) {/
    rr=ruleChoice.getSelectedIndex()+1;
}

private void stopActionPerformed(java.awt.event.ActionEvent evt) {}
    started=false;
    stop();
}

private void startActionPerformed(java.awt.event.ActionEvent evt) {
    ca.chrule(rr);
    ca.clear();
    started=true;
    start();
}
}
```

---

---

**Program 15.3.3** WolframCA.java 続き 2

---

```
private void 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);
}

private javax.swing.JPanel jPanel1;
private javax.swing.JComboBox ruleChoice;
private javax.swing.JButton start;
private javax.swing.JButton stop;
}
```

---