Java 入門

電子計算機工学 I 講座 Java 勉強会資料

2004 年度版

草苅良至 能登谷淳一

2004 年 3 月

本資料について

本資料は、2002 年度および 2003 年度に電子計算機工学 I 講座で行なったセミナー資料を基に改訂したものです。本資料では、主に以下のような表記を用います。

おおまかな書式は次のように 2 重枠で示します。

書式

プログラム内の記述は、次のように一重枠で示します。

```
int x;
```

ソースファイルは、次のよう左端に行番号付けながら罫線で挟んで示します。

リスト 1:HelloWorld.java

```
1  public class HelloWorld{
2      public static void main(String[] argv){
3          System.out.println("Hello World!");
4      }
5  }
```

実行方法と実行結果は、下のように丸枠で示します。なお、$は端末のコマンドプロンプトを表わしています。

```
$java HelloWorld
HelloWorld
$
```

目 次

表 目 次

図 目 次

第1章　Java概要

1.1　Javaとは

Javaは、1995年にSun Microsystems社が発表したプログラミング言語です。C言語と似ている部分もあるので、C言語を修得している人はに取り組みやすいはずです。GUI(Graphical User Interface)を扱うライブラリや、ネットワークを扱うライブラリが充実しており、これらのライブラリを用いて実用的なプログラムを比較的容易に作成できます。

Javaを用いてプログラムを作成するには、Javaの開発環境を選択しなければなりません。開発環境の一つとして、開発元であるSunから無償で提供されているJ2SE(Java 2 Platform,Standard Edtion)を利用する事があります。この、J2SEはwebサイト[1]等から入手可能です。本演習でもJ2SEを用いて演習を行ないます。また、同サイトからクラスライブラリマニュアル(API仕様)も入手可能です。Javaにはクラスライブラリが豊富にあるのですが、逆にすべてのライブラリを把握するのが困難です。これらのライブラフリを有効に用いるためには、API仕様が有用です。したがって、このAPI仕様も同サイトから入手して利用するといいでしょう[2]。入手したAPI仕様は、ブラウザを用いて利用することができます。図1.1に、API仕様の1ページを示します。

1.2　Javaの特徴

Javaを表わすキャッチフレーズとして、“Write at once,run everywhere”というものがあります。これは、Javaがプラットホーム(ハードウェア、OS等)に依存しない言語であることを示したものです。したがって、(例えば、MS-window、Unix、Macintosh等の)どのプラットホームでプログラムを作成しても、他のプラットホーム上でも同じように動作します。このことは、いろいろなマシンが接続しているネットワークにおいて、非常に有利な性質です。

図1.2では、コンピュータの階層構造を示しています。この図は下の方ほど、ハードウェアに近く、上の方ほど利用者に近く描かれています。ハードウェアはその製作会社特有の仕様などがあり、個々の機器によってその基本処理が異なることがあります。しかし、異なるハードウェアであっても、利用者はハードウェア上で動作する**オペレーティング・シ**

[1] http://jp.sun.com

[2] しかし、これらのAPI仕様はプログラマ用に記述されているので、Javaの知識がある程度必要となります。本演習を通じてその知識を修得して下さい。

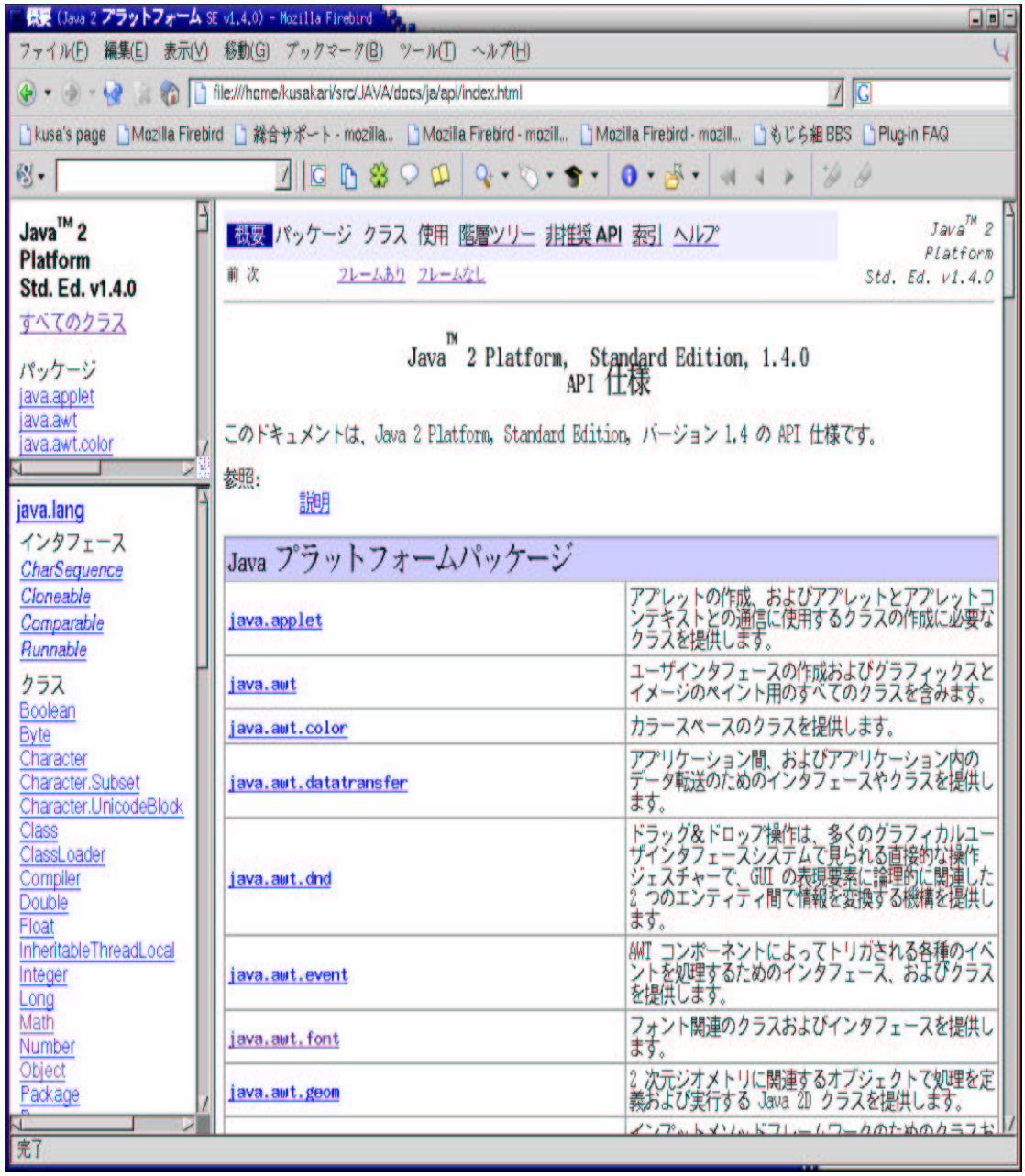

(Java 2 プラットフォーム SE v1.4.0) - Mozilla Firebird
ファイル(F) 編集(E) 表示(V) 移動(G) ブックマーク(B) ツール(T) ヘルプ(H)
file:///home/kusakari/src/JAVA/docs/ja/api/index.html
kusa's page
Mozilla Firebird
総合サポート - mozilla...
Mozilla Firebird - mozill...
Mozilla Firebird - mozill...
もじら組 BBS
Plug-in FAQ
Java™ 2 Platform Std. Ed. v1.4.0
すべてのクラス
パッケージ
java.applet
java.awt
java.awt.color
java.lang
インタフェース
CharSequence
Cloneable
Comparable
Runnable
クラス
Boolean
Byte
Character
Character.Subset
Character.UnicodeBlock
Class
ClassLoader
Compiler
Double
Float
InheritableThreadLocal
Integer
Long
Math
Number
Object
Package
概要 パッケージ クラス 使用 階層ツリー 非推奨 API 索引 ヘルプ
Java™ 2 Platform Std. Ed. v1.4.0
前 次
フレームあり フレームなし
Java™ 2 Platform, Standard Edition, 1.4.0 API 仕様
このドキュメントは、Java 2 Platform, Standard Edition, バージョン 1.4 の API 仕様です。
参照:
説明
Java プラットフォームパッケージ
java.applet
アプレットの作成、およびアプレットとアプレットコンテキストとの通信に使用するクラスの作成に必要なクラスを提供します。
java.awt
ユーザインタフェースの作成およびグラフィックスとイメージのペイント用のすべてのクラスを含みます。
java.awt.color
カラースペースのクラスを提供します。
java.awt.datatransfer
アプリケーション間、およびアプリケーション内のデータ転送のためのインタフェースやクラスを提供します。
java.awt.dnd
ドラッグ&ドロップ操作は、多くのグラフィカルユーザインタフェースシステムで見られる直接的な操作ジェスチャーで、GUI の表現要素に論理的に関連した 2 つのエンティティ間で情報を変換する機構を提供します。
java.awt.event
AWT コンポーネントによってトリガされる各種のイベントを処理するためのインタフェース、およびクラスを提供します。
java.awt.font
フォント関連のクラスおよびインタフェースを提供します。
java.awt.geom
2 次元ジオメトリに関連するオブジェクトで処理を定義および実行する Java 2D クラスを提供します。
完了

図 1.1: API 仕様

ステム (OS) が同一のものであれば、細かいハードウェア仕様がわからなくても、同じようにコンピュータが利用できます。すなわち、ハードウェアの差異を OS がある程度吸収してくれます。しかし、近年では OS の種類も増加し、OS 毎にアプリケーションソフトが開発されたりしました。このため、こんどは、ハードウェアの差異だけではなくて、OS などのプラットフォームの差異にも、注意を払わなくてはならなくなりました。そこで、それらの差異を吸収するためのソウトウェアが開発されるようになりました。このように、現在では、ハードウェアから利用者である人間までの間に、ソフトウェアの階層構造が作られることが多くなっています。特に、近年では、階層構造が多段化しています。中間層のソフトウェアを総称して、ミドルウェアと呼ばれることもあります。

プラットホーム非依存性を実現するために、Java では **Java 仮想マシン (JVM:Java Virtual Machine)** という仮想的なコンピュータを各プラットホーム上に実現します。すなわち、新たなミドルウェアを OS 上に構築します。この仮想的なコンピュータ上では、**バイトコード** と呼ばれるプログラム形態をインタプリタ形式で実行します。しかし、通常プログラマは、バイトコードを直接記述するのではなくて、Java の文法に従って**ソースコード** (テキスト) を記述します。このソースコードは、Java コンパイラによって 、バイトコードにコンパイルされます。このように、Java ではソースコードを、一旦バイトコードという中間的なコードに変換 (コンパイル) し、そのバイトコードは JVM(インタプリタ) によって実行されます。図 1.3 は、Java コンパラと JVM(インタプリタ) の関係を示した模式図です。

1.3 Javaプログラムの作成法と実行手順

ここでは、具体的に Java プログラムを実行するまでの手順を示します。

まずは、ソースコードを emacs 等のテキストエディタを用いて作成します。ここでは、まず例題として、`HelloWorld.java` というソースコードを作ることにします。なお、java 言語のソースコードの拡張子は `java` です。

```
$emacs HelloWorld.java &
$
```

`HelloWorld.java` のファイルに記述すべき内容をリスト 1 に示します。

リスト 1:`HelloWorld.java`

```
1  public class HelloWorld{
2      public static void main(String[] argv){
3          System.out.println("Hello World!");
4      }
5  }
```

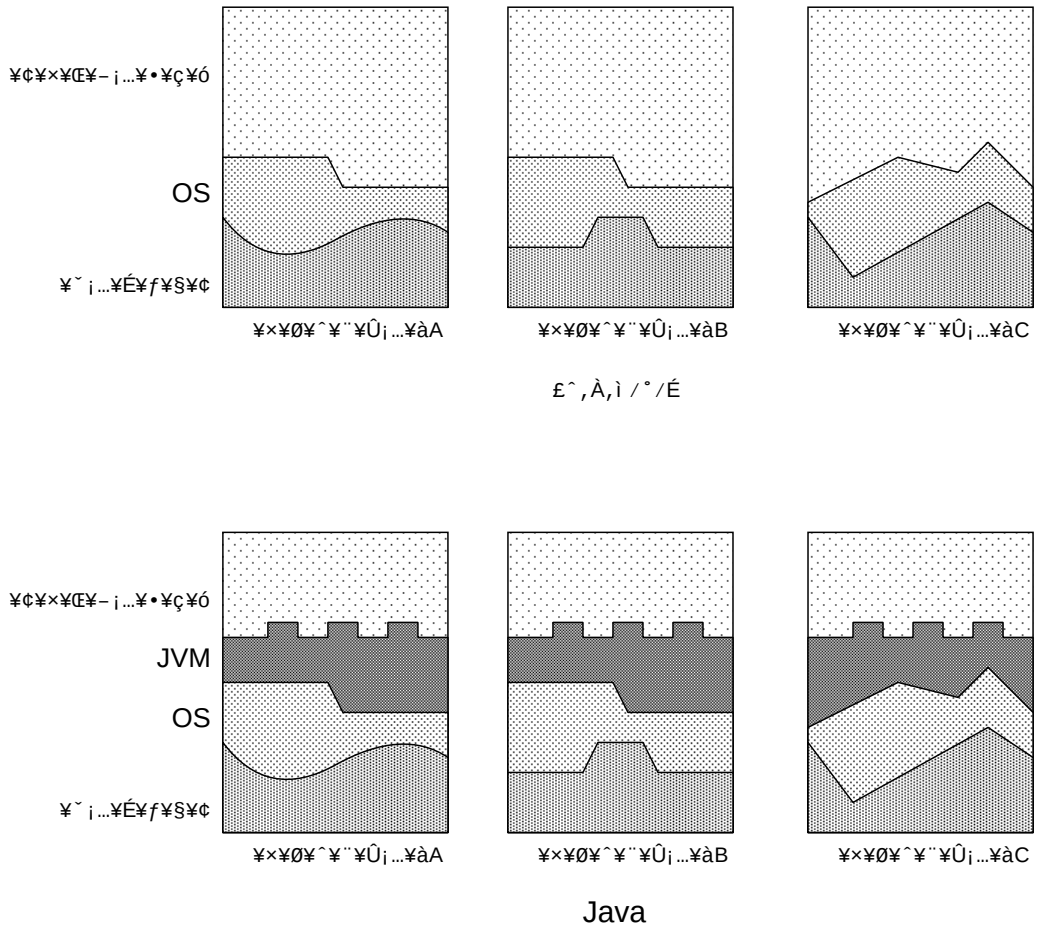


図 1.2: コンピュータの階層構造

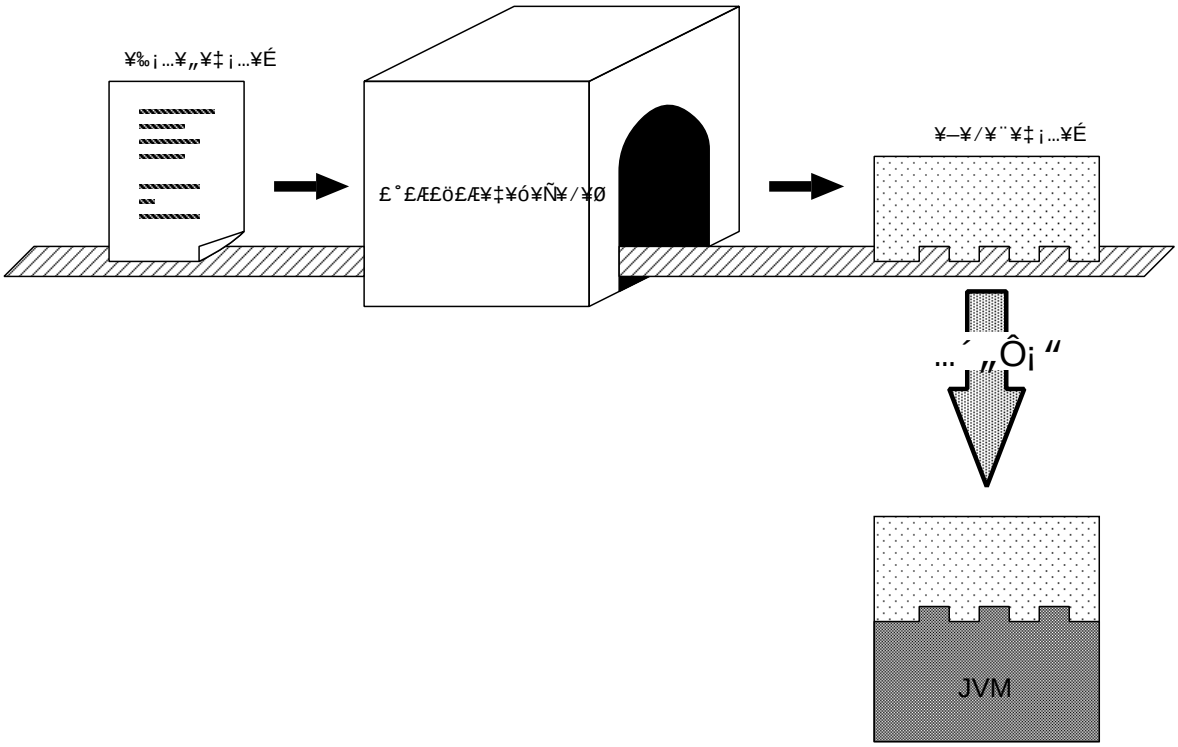


図 1.3: Javaコンパイラ

ここで注意ですが、C 言語のプログラムではソースファイル名は何でもよかったのですが、Java ではソースコード内の記述に依存して決めければいけません。具体的には、クラス名とファイル名が (拡張子を除いて) 同一でなければいけません。リスト 1 では、ソースコード内に `public class HelloWorld` の記述があるので、ソースファイル名は `HelloWorld.java` でなければなりません。次に注意して下さい。

ソースファイル名は「クラス名.java」にすること。

このソースコードをテキストエディタを用いて作成します。(この部分は、C 言語のプログラミングと同様です。)

それでは、HelloWorld.java をバイトコードにコンパイルしてみましょう。Java 言語のソースコードをコンパイルするコマンドは、`javac`[3]です。

javac ソースファイル名

端末で、`HalloWorld.java` のあるディレクトリにおいて、以下のように実行してして下さい。

```
$ls
HelloWorld.java
$javac HelloWorld.java
$ls
HelloWorld.class Helloworld.java
$
```

うまくコンパイルできれば、`HelloWorld.class` というファイルができているはずです。このファイルがバイトコードのファイルです。

ファイル名が「クラス名.class」であるものは、バイトコードのファイルを表わす。

バイトコードを実行するには、コマンドの `java` を用います。

java クラス名

ここで、「クラス名」は、バイトコードのファイル名から「`.class`」の拡張子を取り除いたものであることに注意しましょう。端末で、次のように記述すると、JVM 上でバイトコードが実行されます。

```
$ls
HelloWorld.class HelloWorld.java
$java  HelloWorld
Hello World!
$
```

この Java プログラムは、画面上に文字列を表示 (出力) するプログラムです。

[3] Java Compiler の略。

第2章　Javaの基本的な文法

本章では、Javaの基本的な文法について説明します。

2.1　コメント

Javaのコメントの記述法には、2種類あります[1]。1つ目は、C言語のコメントの書き型と同じで、「/*」と「*/」で挟んで書きます。

```
/* コメントです。*/
```

もう一つの書き方は、//から行末までの間にコメントを書きます。

```
// コメントです。(行末)
```

それでは、HelloWorld.javaにコメントを入れた例を示します。

リスト 2:HelloWorld.java

```
1   /*文字列を表示するJavaプログラム*/
2   public class HelloWorld{
3       public static void main(String[] argv){
4           System.out.println("Hello World!");//文字列表示後改行する。
5       }
6   }
```

コメントは、プログラムの可読性を高めるのに重要です。したがって、必要な情報を漏れなく、しかも簡潔に記述するように常に心がけて下ささい[2]。なお、本資料中では、コメントの1部は省略して示します。

リスト 3:TestComment.java

```
1   /*
```

[1]実は、ソースコードからドキュメントを自動的に作成するためのドキュメントコメントという種類もありありますが、本資料では割愛します。文献を参照して下さい。

[2]プログラミング演習でのスタイル規則等を参考にするといいでしょう。

```
2   2004/3/1 by 草苅良至
3   説明：javaでのコメントの書き方とその効果を確認するプログラム。
4   */
5   public class TestComment{
6       public static void main(String[] argv){
7           System.out.println("コメントの練習中");//日本語の表示
8       }
9   }
```

2.2　データ型

Java では、変数や定数をデータ型で区別します。Java のデータ型として、数値や文字などを直接扱うための基本型と、C 言語のポインタのようにデータを参照してから利用するための参照型があります。

2.2.1　基本型

Java の基本型には、論理値 (真偽値) を表す boolean 型、1 文字を表わす char 型、整数を表す int 型、浮動小数点数 (実数) を扱う double 型、などがあります。C 言語にない型として論理型があることに注意しましょう[3]。また、Java での文字型は、Unicode というコード体系を用いることに注意しましょう[4]。

基本型の種類とその値を表 2.1 にまとめて示します[5]。

表 2.1: 基本型

基本型の種類	基本型の意味	リテラルの例
`boolean`	論理値 (真偽値)	`true`, `false`
`char`	16 ビットの Unicode 文字 (1 文字)	`'A'`, `'あ'`, `'\u0061'`
`int`	32 ビットの整数 (符号付)	`1`, `-10`
`double`	64 ビットの浮動小数点数 (符号付)	`3.141592`, `6.0e24`

図 2.1 に基本型の概念図を示します。変数、変数名、値の区別をきちんと行なうようにして下さい。なお、あるデータ型の値をソースコードの中で直接書く書き方のことを、そ

[3]なお、C 言語では論理型を整数型で代用していました。C 言語では、0 は偽を表し、1 などの 0 以外の値は真を表わします。

[4]C 言語では、8 ビット (1 バイト) の ASCII コードを通常用いています。

[5]この表以外にも基本型はありますが、この表中の型で十分と考えられます。他の型については他の文献を調べて下さい。

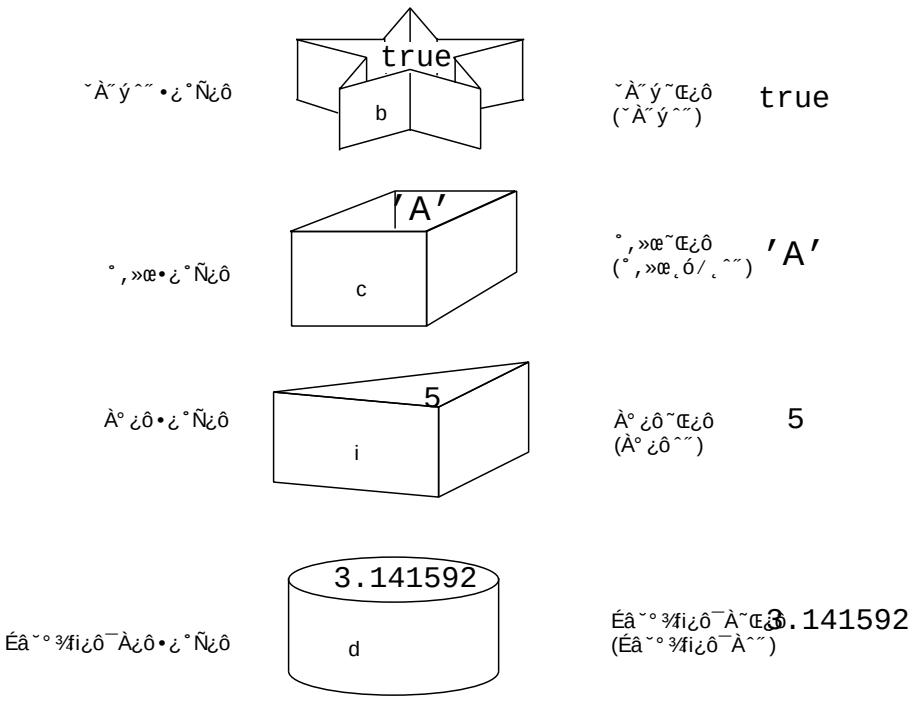


図 2.1: Java の基本型

のデータ型の**定数**あるいは**リテラル**といいます。

2.2.2 参照型

C 言語のポインタのようにデータを参照してから利用するために、Java には参照型があります。C 言語ではアドレスを保持する変数がポインタでしたが、Java での参照にはアドレスというものは陽には現れません。単に、参照というワンクッションをおいて、データ[6]を利用するためのものです。しかし、その効果は C 言語のポインタと類似点が多いので、対比して理解するといいでしょう。

参照型の例として、文字列型を意味する `String` 型があります[7]。例えば、`String` 型の変数である `s` に、4 文字の列である `book` 等を保存することができます。

[6]配列や、後述するオブジェクトのことです。
[7]文字型である char 型と区別しましょう。

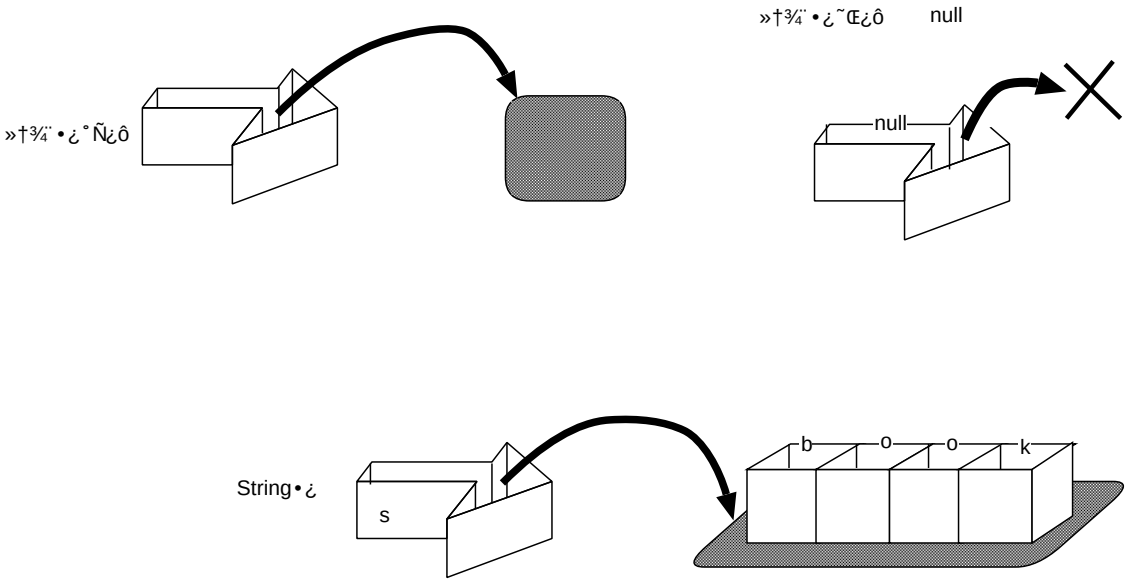


図 2.2: Java の参照型

参照型の定数として、**null** があります。**null** は、String 型の定数だけでなく、すべての参照型の定数です。参照型の変数に **null** が保持してあると、その変数は何も参照していないことを意味します。ソースコード内で文字列型 (String 型) の定数を表すには、文字列をダブルクォーテション「“」で挟みます。したがって、「**“book”**」などが、**String** 型の定数 (リテラル) です[8]。

図 2.2 に参照型の概念図を示します。

2.3　変数宣言と初期化

Java の変数は、宣言してから用います。

宣言の方法 (書き方) は、

データ型 変数名；

です。ですから、例えば、

```
int x;
```

のように記述すればいいです。また、同じ型の変数を 2 つ以上同時に宣言することもできます。その場合の書き方は次のようになります。

データ型 変数名 1, 変数名 2,...；

```
int i,j;
```

[8] **char** 型のリテラルはシングルクォーテーション「’」で一文字を挟んだものです。

ここで、注意ですが、C 言語では変数宣言を関数の最初にまとめて行ない[9]、演算等の後では行なえませんでした。それに対して、Java ではソースコード内のどこで行なってもいいです。すなわち、演算の後などであっても、変数宣言が可能です。さらに、C 言語との比較すると、Java では変数を初期化 (あるいは代入) しないと、コンパイル時にエラーになります。変数を初期化するには、宣言時に行なうか、宣言後に代入によって行なうかのどちらかで行ないます。典型的な初期化の書き方は、次の 2 つです。

データ型 変数名 1 = 初期値 1；

データ型 変数名 2;
変数名 2=初期値 2；

```
int x=0;
double y;
y=1.0;
```

リスト 2.1 に基本型を用いたプログラム例を示します。図 2.1 と対応させて、ソースコードを読むと良いでしょう。

リスト 4:TestBasicTypes.java

```
1  /* Java の基本型とその効果を見るサンプルプログラム*/
2  public class TestBasicTypes{
3      public static void main(String[] argv){
4          /*boolean*/
5          boolean b=true;  //変数宣言と初期化
6          System.out.print("boolean 型の表示 b=");//改行なし
7          System.out.println(b);
8
9          /*char*/
10         char c='A';
11         System.out.print("char 型の表示 c=");
12         System.out.println(c);//型で自動的に表示を制御している。
13
14         /*int*/
15         int i=5;
16         System.out.print("int 型の表示 i=");
17         System.out.println(i);
18
```

[9]C 言語では、ブロックの最初、すなわち中括弧「{」の直後であれば変数宣言ができます。このときには、宣言したブロック内でその変数を用いることができます。

```
19          /*double*/
20          double d=3.141592;
21          System.out.print("double 型の表示 d=");
22          System.out.println(d);
23
24          return;
25      }
26  }
```

次に、リスト 5 に参照型である String 型を用いたプログラム例を示します。図 2.2 と対応させて、ソースコードを読むと良いでしょう。

リスト 5:TestString.java

```
1   /* Java の String 型とその効果を見るサンプルプログラム*/
2   public class TestString{
3       public static void main(String[] argv){
4           /*String*/
5           String s=null;  //変数宣言と初期化
6           System.out.println(s);
7
8           s="book"; //文字列定数の代入
9           System.out.println(s);
10          return;
11      }
12  }
```

2.4　式と演算子

これまでは、あるデータ型を持つものとして、変数や定数 (リテラル) を勉強してきました。しかし、プログラムでは、これらのデータを処理 (加工) するために、何らかの演算を行なう必要があります。それらの演算は、決められた演算子を用いて行ないます。すなわち、演算子で変数や定数を結合して行ないます。この演算子で変数や定数を結合したものを**式**といいます。実は、変数や定数自体も式です。したがって、式と式を演算子で結合し、新たな式を作っていくことで、演算が行なわれます。式には、それ自体にも型と値があります。このことは、BNF 記法を用いると、次のように表現されます。

```
式 ::= 定数
     | 変数
     | 式
     | 式 演算子
     | 演算子 式
     | 式 演算子 式
```

ここで、「::=」は左辺を右辺で定義することを示す記号で、「|」は「または」を意味する記号です。定義が再帰的になっていることに注意して下さい。

表 2.2 に、Java の演算子一覧を示します。各演算子の詳細については別文献を参照して下さい。なお、「=」などの代入演算子を用いた演算では、右辺の式 (からの計算後) の値が左辺の変数に設定されるという**副作用**を持ちます。すべての代入演算子において、左辺が式ではなくて変数であることに注意しましょう[10]。

表 2.2: 演算子一覧

結合力	演算子の種類	演算子
強い	後置演算子	配列参照 [添字], 参照.フィールド, 参照.メソッド (引数), メソッド (引数), 変数++, 変数--
↑	前置演算子	++変数, --変数, +式, -式, ~式, !式
	new 演算子とキャスト	new 型, (型) 式
	乗除演算子	式*式, 式/式, 式%式
	加減演算子	式+式, 式-式
	シフト演算子	式 << 式, 式 >> 式, 式 >>> 式
	関係演算子	式 < 式, 式 > 式, 式 <= 式, 式 >= 式, instanceof 参照
	等値演算子	式 == 式, 式! = 式
	ビット AND	式&式
	ビット XOR	式∧式
	ビット OR	式 | 式
	論理積	式&&式
	論理和	式 || 式
↓	条件演算子	式?式 : 式
弱い	代入演算子	変数=式, 変数+=式, 変数-=式, 変数*=式, 変数/=式, 変数%=式, 変数<<=式, 変数>>=式, 変数>>>=式, 変数&=式, 変数∧ = 式, 変数|=式

[10]代入演算子で結合したものも式であることに注意しましょう。この式の値をさらに別の代入演算子を用いて新たな変数に代入することもできます。

ここでは、いくつかの演算子に対してだけ、その効果を調べていきます。特に、演算子の結合力に注意して下さい。結合力に自信がなければ、括弧を用いて自分の意図を明確に示すこともできます[11]。実際にコードに括弧を記述しなくても、頭の中で括弧を考えるだけでも便利なことが多いです。

リスト 6 に、算術演算子の効果を調べるプログラムを示します。算術演算子は、表 2.2 では、乗除演算子と加減演算子として示されています[12]。

リスト 6:TestArithmeticOperator.java

```
1  /* 算術算子の効果を見るサンプルプログラム */
2  public class TestArithmeticOperator{
3      public static void main(String[] argv){
4          /*2 次関数の計算 (結合力の確認)*/
5          double x=0.8;//初期値を変更する。
6          double y=0.0;
7
8          y=x*x - 3.0*x + 2.0; //関数定義
9
10         System.out.println("y=x*x- 3.0*x+2.0");
11         System.out.print("x=");
12         System.out.println(x);
13         System.out.print("y=");
14         System.out.println(y);
15         System.out.println();//改行
16
17         //左結合の確認
18         double f=0.0;
19         f=6.0/2.0*3.0;
20         System.out.println("f=6.0/2.0*3.0");
21         System.out.print("f=");
22         System.out.println(f);
23
24         return;
25     }
26 }
```

[11]しかし、必要以上に括弧を用いると、逆に、読みずらいコードになってしまうこともあります。
[12]結合力に差が付いていることに注意しましょう。

次に、比較演算子の効果を調べるプログラムをリスト 7 に示します。表 2.2 では、比較演算子は関係演算子と等値演算子として示されています[13]。

リスト 7:TestCompativeOperator.java

```
1  /* 比較演算子の効果を見るサンプルプログラム */
2  public class TestCompativeOperator{
3      public static void main(String[] argv){
4
5          System.out.print("1.0>2.0  ");
6          System.out.println(1.0>2.0);
7          System.out.print("1.0<2.0  ");
8          System.out.println(1.0<2.0);
9          System.out.println();
10
11         /*比較演算子の結合結果は、boolean 型。*/
12         boolean b=true;
13         b=2.0 < 1.0;
14
15         System.out.print("b= 2.0< 1.0  ");
16         System.out.println(b);
17
18
19         b=5.0+2.0<3.0;
20         System.out.print("b= 5.0+2.0<3.0  ");
21         System.out.println(b);
22         return;
23     }
24 }
```

C 言語には無い Java の演算子として、文字列を連結する連結演算子「+」があります。この連結演算子を用いることによって、容易に文字列を繋げることができます。なお、連結演算子「+」と算術演算子の「+」と区別は、演算子の両辺の型で自動的に判断されます。つまり、「+」の左辺か右辺いずれかの式の型が String 型である場合に、その記号は連結演算子と見なされ、文字列が連結されます。この場合、文字列でないものは、自動的に文字列に変換されます。連結演算子の効果を調べるプログラムをリスト 8 に示します。

[13] こちらも、結合力に差が付いていることに注意しましょう。

リスト 8:TestCatOperator.java

```
1  /* 連結演算子の効果を見るサンプルプログラム */
2  public class TestCatOperator{
3      public static void main(String[] argv){
4          String s="";//空文字列 (null と区別すること。)
5          System.out.println(s);
6
7          /*文字列の連結*/
8          s="Book"+"End";//連結演算
9          System.out.println(s);
10
11         s=s+"Pair";
12         System.out.println(s);
13
14         /*文字への変換 1*/
15         s=""+(1.0<2.0);
16         System.out.println(s);
17
18
19         /*文字への変換 2*/
20         double x=0.0;
21         double y=0.0;
22
23         x=0.3;
24         y=2.0*x*x;
25         System.out.println("x="+x+"のとき y=2.0*x*x の値は、y="+y);
26         return;
27     }
28 }
```

第3章　Javaの制御構造

前章までの知識で、電卓を用いて行なえる程度の簡単な計算を、Javaを用いて行なえるようになりました。本章では、より高度な計算をするために必要になる制御構造について説明します。制御構造は、アルゴリズムのJavaでの記述法とみなすことができます。すなわち、プログラムの流れを適切に制御するための記述法です。

アルゴリズム、すなわちプログラムの実行順序の制御には、主に次の3つの構造があります。

- 順次構造
- 分岐構造 (選択)
- 反復構造 (繰り返し)

図3.1に、これらの構造を表わすフローチャートを示します。この3つの構造を、**基本構造**と呼びます。この基本構造だけで、コンピュータで実行可能なすべてのアルゴリズム (プログラム) が表わせることが知られています。ただし、ここでは「原理的に表現可能とういだけ」なことに注意しましょう。実際にこの基本構造だけである程度以上の規模を持つプログラムを作成しようとすると、プログラムが複雑になり過ぎてしまうことが多く、きちんと動作するものを完成させることは困難です。これらの対処法としては、プログラムを小さな機能単位に分けて作成することが考えられます。機能分離として、C言語では関数が用いられましたが、Javaではクラス (オブジェクト) を用います。それらの詳細については、後述します。

3.1　文とブロック

節2.4では、式を説明しました。式の後に「;(セミコロン)」を付けることにより、**文**になります。文という単位でプログラムを理解することも重要です。これまでの例では、主に一行に記述されたものが文です[1]。また、いくつかの文を並べて中括弧 ({ と }) で囲んだものを**ブロック**あるいは**複文**といいます。複文もまた文の一種です。基本構造のそれぞれに対応して、文が定義されます。選択構造に対応した**選択文**, 反復構造に対応した**反復文**があります。これらの文法をBNF記法で示します[2]。

[1] 行の終りつけている「;」は、式を文にするための記号です。
[2] ここでは、Java文法全体ではなくて必要部分だけを抽出しています。

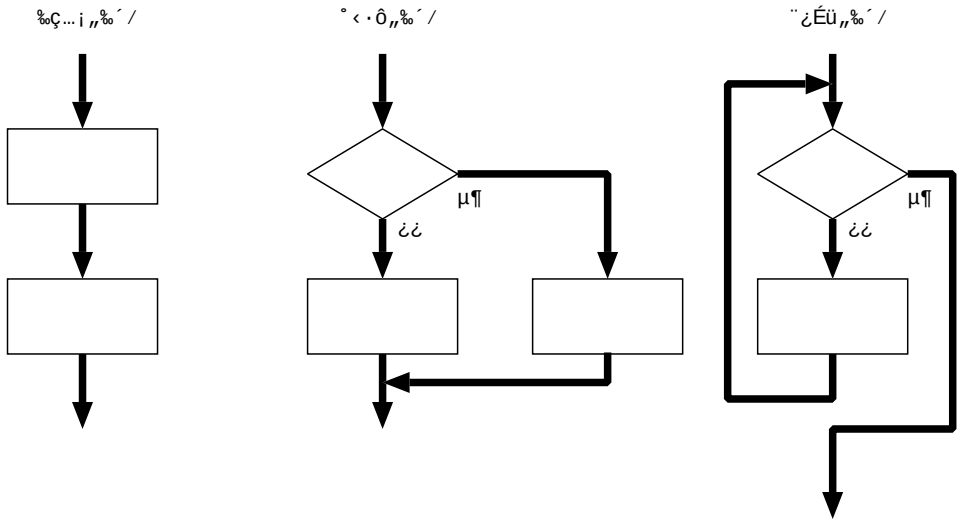


図 3.1: 基本制御構造

```
文 ::= 式 ; | ブロック | 選択文 | 反復文
ブロック ::= {
                文 1
                文 2
                .
                .
                .
                文 n
          }
```

文とブロックが相互に定義されていることに注意して下さい。図 3.1 においては、四角で表わしているものが一つの文[3]と考えられます。

3.2　順次構造

この構造は、一番直感的であり、ソースコードの上から順に文を記述することで実現されます。ブロックが文であることに注意すると、ブロック単位で上から並べても順次構造は実現されることもわかります。また、ブロック内の動作も (他の制御構造がなければ) 順次構造になります。図 3.1 の順次構造を参照して下さい。ここまでのソースコードの記述は、

[3]ブロックも文であることに注意すること。

すべてが順次構造で表わされていました。

3.3 分岐 (選択) 構造

図 3.1 の分岐構造で示すように、論理式の値にしたがってプログラムの流れを分岐させることが**分岐構造**です。図からもわかりますが、2 つの文のどちらかを選択して実行するという見かたもできるので、**選択構造**ともいいます。C 言語と同様、Java でも「`if`」や「`else`」を用いて分岐構造を表わします。BNF 記法では下記のようになります[4]。

選択文 ::= if(論理式) 選択実行文 1
| if(論理式) 選択実行文 1 else 選択実行文 2

ここで、論理式も式の一種であり、選択実行文も文の一種であることに注意しましょう。また、BNF 記法の 1 行目や 2 行目のように「`if`」を用いた選択文を「**if** 文」といい、2 行目のように「`if`」と「`else`」を用いた選択文を「**if-else** 文」ともいいます。

それでは、`if` 文を用いた分岐構造の例をリスト 9 に示します。リスト 9 では、多分岐の構造になっていますが、この構造を BNF 記法にしたがって、解析してみるといいでしょう。

リスト 9:`ChangeYear.java`

```
1  /*西暦を和暦に変換するプログラム (20 世紀以降)*/
2  public class ChangeYear{
3      public static void main(String[] argv){
4          int seireki=1901; //西暦
5          int wareki=1; //和暦
6
7          seireki=1900;//西暦の設定
8
9          System.out.print(seireki+"年は");
10         if(seireki<1901){
11             System.out.println("範囲外");
12         }else if(seireki<1912){
13             wareki=seireki-1867;
14             System.out.println("明治"+wareki+"年");
15         }else if(seireki<1926){
16             wareki=seireki-1911;
17             System.out.println("大正"+wareki+"年");
18         }else if(seireki<1989){
19             wareki=seireki-1925;
```

[4] ここでも、Java 文法全体ではなくて必要部分だけを抽出しています。

```
20              System.out.println("昭和"+wareki+"年");
21          }else{
22              wareki=seireki-1988;
23              System.out.println("平成"+wareki+"年");
24          }
25
26          return;
27      }
28  }
```

3.4　反復 (繰り返し) 構造

図 3.1 の反復構造で示すように、ある条件にしたがって、同じ文を繰り返し実行させることが、**反復構造**です。C 言語と同様、Java でも「for」や「while」を用いて反復構造を表わします。BNF 記法では下記のようになります

```
反復文 ::= for(初期化式; 論理式; 反復式) 反復実行文
          |while(論理式) 反復実行文
          |do 反復実行文 while(論理式);
```

ここで、1 行目の「for」を用いた反復文のことを for 文といいます。for 文において、初期化式は式の一種であり、反復実行文を反復して実行する直前に 1 度だけ実行されます。Java では、この初期化式において変数宣言もできます。初期化式で宣言された変数は、続く反復実行文[5]内で用いることができます。論理式は選択文のときと同様に、論理値を取る式です。反復式は反復実行文の直後に、反復して実行される式です。

2 行目の「while」を用いた反復文のことを「while 文」、3 行目の「do」と「while」を用いた反復文のことを「do-while 文」といいます。

それでは、for 文を用いた反復構造の例をリスト 10 に示します。リスト 10 では、反復の構造が 2 重になっています。このように反復が多重になっている反復文のことを、**多重ループ**ともいいます。リスト 10 では、**2 重ループ**の例が示されています。この構造も BNF 記法にしたがって解析したり、図 3.1 の図のように図的に解析したりするとといいでしょう。

リスト 10:Kuku.java

```
1   /*2 重ループを用いて九九を表示させるプログラム*/
2   public class Kuku{
3       public static void main(String[] argv){
```

[5]反復実行文や選択実行文は、ブロックになることが多いです。

```
4           for(int i=1;i<=9;i++){
5               for(int j=1;j<=9;j++){
6                   System.out.print(i+"×"+j+"="+(i*j)+"  ");
7               }
8               System.out.println();
9           }
10          return;
11      }
12  }
```

C 言語との相違で特に注意することは、Java では条件判断で用いる論理式の型がすべて `boolean` 型であるという点です。これは、選択構造の部分でにも言えることです。

第4章　クラスとオブジェクト

前章までの知識で、Javaでのアルゴリズムの表現が可能になりました。しかし、前述したように実用的なプログラムを作成するには、プログラムを幾つかの機能毎に分割して作成する必要があります。

C言語の場合には、主に関数を用いてプログラムを分割しました。

Javaでのプログラミングにおける、最も基本的な単位が**クラス**です。このクラスの組み合せによって、Javaのプログラムは作成されます。クラスという概念は抽象的で初学者の人にはわかりづらい面もあるのですが、慣れるとプログラミング効率が格段に上がります。

以下で、このクラスの概念と、Javaでのクラスの表現法について順に説明します。

4.1　オブジェクト

クラスは、**オブジェクト**の集合とみなすことができます。そこで、ここではまずオブジェクトについて説明します。Javaがオブジェクト指向言語と呼ばれるように、オブジェクトはJavaプログラミングを理解する上で重要な概念です。

コンピュータを用いて現実世界の諸問題を解決するには、その問題やとりまく環境をコンピュータ上で(形式的に)表現する必要があります。このことを**モデリング**といいます。オブジェクト指向言語が無かった時代には、現実世界の対象物とプログラムの形式的定義の間には大きなギャップがあり、そのギャップを埋めるために多大な労力が支払われていました。それらのギャップを少しでも埋めるために、現実世界の対象物の多くに共通している性質が考えだされました。すなわち、現実世界の多くの「モノ」は、「属性」と「機能」で表現が可能だと考えられるようにりました。この「属性」と「機能」で表わされる「モノ」のことを、**オブジェクト**といいます。すなわち、「オブジェクト」とは現実世界の「モノ」をモデル化したものです。また、**オブジェクト指向**とは、現実世界の対象物を「モノ」としてとらえる方針のことです。

ここでは、現実世界の「オブジェクト」が「属性」と「機能」を持っていることをいくつかの例で見ていきます。

例1：車

属性: 全長、排気量、色、燃費

機能: 走る

例 2：電卓

属性: 外形、ボタンの数、重さ

機能: 計算する (足す、引く、掛ける、割る)

例 3：人間

属性: 年齢、血液型、性別

機能: 考える、走る、食べる

例 4：携帯電話

属性: 色、重さ、値段

機能: 電話を通じさせる、写真を取る

このように、「属性」と「機能」で現実世界を見直すと、概念が整理されることがわかります。

4.2　クラス

オブジェクトは、「この車」とか「あの電卓」というように指示語を用いて表わすことができたり、「自分 1 人」とか「携帯 2 個」というように数えることができます。それに対して、「車」や「電卓」等の概念自体がクラスです。現実世界ではオブジェクトとクラスの区別を明確にしないことも多いです。例えば、「私の車のナンバーは××です。」という文では、「車」はオブジェクトの意味で使っていると考えられます 。また、「車にはタイヤがある」という文では、「車」はクラスの意味で用いていると考えられます。このように、オブジェクトとクラスは区別しにくいのですが、オブジェクト指向プログラミングでは明確に区別しなければなりません。

より厳密には、オブジェクトの集合のことを**クラス**といいます。例えば、

$$車 = \{x | x は世界にある全ての車 \}$$

となります。この左辺にある「車」はクラスであり、右辺の括弧の中にある「車」がオブジェクトです。

図 4.1 では、「三角形」というクラスと、「三角形」のオブジェクトの関係を示しています。

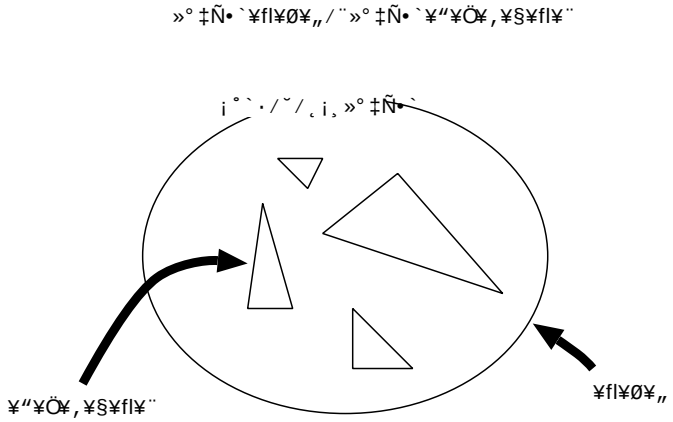


図 4.1: クラスとオブジェクト

4.3 Javaでのクラス (クラス定義)

節 4.1 では「オブジェクトは “属性” と “機能” で表現可能です」と述べ、節 4.2 では「クラスはオブジェクトの集合です」と述べました。このことから、Java のクラスは「属性」と「機能」を合わせ持つような表現法になります。

実は、属性だけや機能だけを表わす方法は、C 言語にもあります。属性はいくつかの変数の集合としてとらえることができるので、C 言語の構造体を用いれば表現可能です。また、機能とは節 4.1 中で「動詞」で表現されたように、変化する量で表現可能です。すなわち、C 言語では関数を用いることで、機能も表現可能です。しかし、これらの属性と機能を一つにまとめるための記述法が C 言語には存在しません。これに対して、Java にはこれらの属性と機能を一つにまとめるための記述法が存在します。属性と機能を一つにまとめて定義したものが Java におけるクラスであり、クラス定義を Java の文法にしたがって記述したものが Java プログラムのソースコードになります。

このように、Java でのクラスは、C 言語における構造体定義と関数定義を合わせたものとみなすこともできます。また、Java でのオブジェクトは、値の設定された構造体と関数を合わせたものとみなすことができます。なお、Java では、「属性」を**フィールド**と呼ばれる変数の集合で表わし、「機能」を**メソッド**と呼ばれる “関数” の集合で表わします。フィールドのことをクラスの**メンバ変数**と呼んだり、メソッドのことをクラスの**メンバ関数**と呼んだりすることもあります。また、フィールドとメソッドを合わせてそのクラスの**メンバ**と呼ぶこともあります。これは、クラスを「C 言語の構造体定義に関数定義を持たせたもの」という観点に立った呼び方といえます。

BNF 記法でのクラス定義を示します。

```
クラス定義 ::= [アクセス修飾子] class クラス名 [継承指定] {
                    フィールド定義
                    メソッド定義
               }

フィールド定義 ::= [アクセス修飾子] 型名 変数名;
                                        .
                                        .
                                        .

メソッド定義::= [アクセス修飾子] 型名 メソッド名 (引数リスト ) [送出例外] ブロック
                                        .
                                        .
                                        .
```

なお、「[」と「]」と囲まれた部分は、省略可能を意味します。また、「…」は、0回以上の繰り返しを意味します。フィールドが0個、すなわち、フィールドが無いクラスも定義可能なことに注意しましょう。また、同様に、メソッドが無いクラスも定義可能です。

図4.2に、Javaにおけるクラスの概念図を示します。

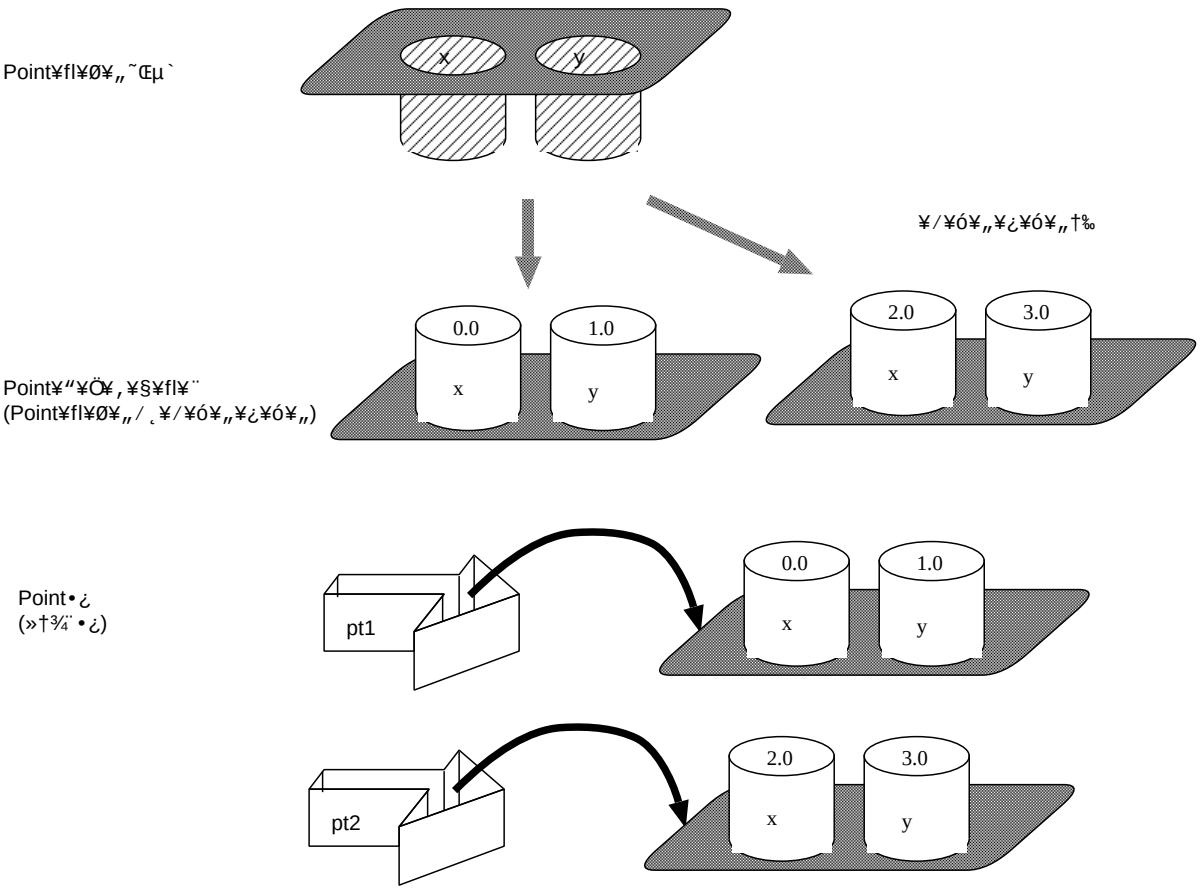


図 4.2: Javaでのクラス

ここでは、簡単なクラスから始めて、クラスを複雑に変更していくことで、Java でのプログラミングを見ていくことにします。

4.3.1 フィールド定義

リスト 11 に、メソッドの無いフィールドだけの簡単なクラスを示します。

リスト 11:Point.java(Ver.1)

```
1  /*クラス定義のサンプルプログラム*/
2  public class Point{
3      /*フィールド*/
4      double x; //x 座標
5      double y; //y 座標
6      /*メソッド*/
7      //このクラスには、メソッドは無い。
8  }
```

この定義では、Point という新しいクラスを定義しています。この新しいクラスは、Point 型という新しい参照型の定義にもなっています[1]。この例のように、C 言語の構造体との類似点も多いので、対応して理解するといいでしょう。構造体のメンバに対応するものが、クラスのフィールドです。なお、Java には構造体というものはありません。すべてクラスを用いてプログラミングします[2]。

リスト 11 は javac コマンドでコンパイルでき、Point.class のバイトコードが作成されます。しかし、このクラスには、main メソッドが無いために、java コマンドで実行できません。

```
$ ls
Point.java
$ javac Point.java
$
$ ls
Point.class  Point.java
$ java Point
Exception in thread "main" java.lang.NoSuchMethodError: main
$
```

[1] このように、Java では、クラスを定義すると、そのクラスに対応する参照型の定義にもなります。あるクラスのオブジェクトを参照することが出来る参照型の変数をその**クラスへの参照型**の変数と呼んだり、もっと省略してその**クラス型**の変数と呼んだりします。

[2] 明かに構造体で表現できることはクラスでも表現できるので、クラスは構造体の一種の拡張にもなっています。

Javaでは、mainメソッドからプログラムの実行を開始します。mainメソッドの無いバイトコードをjavaコマンドで実行しようとすると、このようなエラーメッセージが出力されます[3]。

このPointクラスのように、自分で定義したクラスを利用するには、次節以降のように行ないます。

4.3.2 オブジェクトの生成 (インスタンス化)

クラスを定義しただけでは、プログラム中でクラスを利用することはできません。Javaでは主にオブジェクトいう形にしてからクラスが利用されます[4]。クラス定義は、その要素であるオブジェクトを生成するための準備とみなすといいでしょう。あるクラスのオブジェクトの生成は、あるメソッド内で、new演算子をクラス名に適用することで行ないます[5]。なお、あるクラスに属するオブジェクトのことを、そのクラスの**インスタンス**ともいいます。また、あるクラスに属するオブジェクトを生成することを、そのクラスを**インスタンス化**するといいます。クラスはインスタンス化して、オブジェクトとうい形で利用します。

オブジェクトの生成は、new演算子をクラス名に適用して行ないます。new演算子は、そのクラスのオブジェクトを生成し、そのオブジェクトへの参照を返します。これらの参照は、予め宣言しておいたそのクラスへの参照型の変数に代入して利用します。したがって、次のような書式で記述します。

```
クラス名 変数;
変数 = new クラス名 ();
```

```
Point pt;//Point 型の宣言
pt=new Point();//Point オブジェクトの生成
```

あるいは、次のように、オブジェクトの生成をクラスへの参照型変数の初期化の時に行ないます。

```
クラス名 変数= new クラス名 ();
```

```
Point pt=new Point();//Point オブジェクトと参照変数
```

いずれにしても、クラス名は、そのクラスのオブジェクトを参照する参照型を表したり、そのクラスのインスタンスの生成指示を表わしたりすることに注意して下さい。これらを区別して、理解を深めるようにして下さい。このようにして生成されたオブジェクトは、そのクラス型の変数を通して用いられます。具体的には、「.(ドット)」演算子によって、オブジェクトのフィールドにアクセス出来ます。このように、ドット演算子でフィールド名と結合した式は、あたかも、そのフィールドを定義している型の変数のように取り扱え

[3]後述するが、javaコマンド以外にも、バイトコードを実行する方法があります。このPoint.classのようにmainメソッドの無いバイトコードは、javaコマンド以外の方法で利用されます。

[4]クラスのままで利用されることもあるが、その詳細は本資料では割愛する。

[5]表 2.2 を参照して下さい。

ます。なお、フィールドの定義では、基本型だけでなく、既に定義されたクラス型を用いた定義もできることに注意しましょう。

参照型変数名. フィールド名
pt.x　//double の型の変数として用いられる。

前節の Point クラスに対して、Point オブジェクトを生成して利用するプログラム例を示します。なお、この Point クラスは、同じクラスに main メソッドを追加して、その中で利用されます。

リスト 12:Point.java (Ver.2)

```
1   /*Point オブジェクトの生成法と利用法を調べるサンプルプログラム*/
2   public class Point{
3       /*フィールド*/
4       double x; //x 座標
5       double y; //y 座標
6
7       /*メソッド*/
8       //main メソッド
9       public static void main(String[] argv){
10          //pt1
11          Point pt1;  //Point 型変数の宣言
12          pt1=new Point(); //Point オブジェクトの生成
13
14          pt1.x=0.0;
15          pt1.y=0.0;
16
17          System.out.println("pt1=("+pt1.x+","+pt1.y+")");
18
19          //pt2
20          Point pt2=new Point();//Point 型変数宣言と初期化
21
22          pt2.x=1.0;
23          pt2.y=2.0;
24
25          System.out.println("pt2=("+pt2.x+","+pt2.y+")");
26
27          return;
```

```
28        }
29    }
```

4.3.3 他のクラスの利用

Javaのソースコードは、1クラスを1ファイルで記述する必要があります。したがって、クラスを複数定義するためには、ソースファイルも分割しなければなりません。ここでは、先の Point クラスに対して、それを利用する TestPoint クラスを示します。

リスト 13:Point.java (Ver.3、リスト 11 と同じ。)

```
1   /*クラス定義のサンプルプログラム*/
2   public class Point{
3       /*フィールド*/
4       double x; //x座標
5       double y; //y座標
6
7       /*メソッド*/
8       //このクラスには、メソッドは無い。
9   }
```

リスト 14:TestPoint.java (Ver.3)

```
1   /* Pointクラスを利用するクラス。*/
2   public class TestPoint{
3       public static void main(String[] argv){
4           Point pt1;//Point型の宣言,pt1が変数名
5           pt1=new Point(); //オブジェクトの生成(インスタンス化)と代入
6
7           pt1.x=0.0;
8           pt1.y=1.0;
9           System.out.println("pt1=("+pt1.x+","+pt1.y+")");
10
11
```

```
12          Point pt2=new Point();//宣言と初期化
13
14          pt2.x=2.0;
15          pt2.y=3.0;
16          System.out.println("pt2=("+pt2.x+","+pt2.y+")");
17          return;
18      }
19  }
```

このように、他ファイルで定義したクラスファイルを利用することができます。高度なプログラムを作成するには、多くのクラスを作る必要があるため、ソースファイルの数も多くなることでしょう。これらのファイルの管理については後述します。さしあたりは、これらの複数のクラスは、同じディレクトリに置くようにして下さい。

```
$ ls
Point.class  Point.java  TestPoint.class  TestPoint.java
$ java TestPoint
pt1=(0.0,1.0)
pt2=(2.0,3.0)
$
```

4.3.4 メソッド定義

C 言語の関数においては、各関数は独立しており、どの関数も “対等” の立場でした。しかし、オブジェクト指向言語である Java では、「メソッドはオブジェクトに付随する。」という考え方を取ります。このことを例を用いて説明します。

例えば、次のクラスを考えましょう。

例：車

属性: ガソリンの量、燃費

機能: 走る

このクラス「車」の 2 つのインスタンスを車 1、車 2 とします。車 1 と車 2 では燃費が異なるので、「走る」という同じ (?)「操作」の結果、減るガソリンの量も異なります。すなわち、オブジェクトとして車が異なれば、「走る」という「メソッド」の “効果” も異なります。

メソッドの定義は、次のような書式で定義できます[6]。

```
[public] class クラス名 {

      /*フィールド定義*/
      型 1 変数名 1 ;
      型 2 変数名 2 ;
            .
            .
            .

      /*メソッド定義*/
      戻り値型 1 メソッド名 1(型 1a 仮引数 1a, 型 1b 仮引数 1b, … ) {
            .
            .
            .
      }

      戻り値型 2 メソッド名 2(型 2a 仮引数 2a, 型 2b 仮引数 2b, … ) {
            .
            .
            .
      }

                        .
                        .
                        .
}
```

リスト 15 に、Point クラスに幾つかのメソッド定義を追加した例を示します。

リスト 15:`Point.java` (Ver.4)

```
1   /*メソッド定義のサンプルプログラム*/
2   public class Point{
3       /**********************フィールド*****************************/
4       double x; //x 座標
5       double y; //y 座標
```

[6]節 4.3 で定義した BNF 記法も見ること。次の書式が、BNF 記法と一致していることを確かめること。

```
6
7       /**********************メソッド****************************/
8       /*
9          原点までの距離を求めるメソッド
10         戻り値：原点までの距離(double)
11         引数　：なし
12      */
13      public double distanceToOrigin(){
14          return Math.sqrt(x*x+y*y);
15      }
16
17      /*
18         他の点までの距離を求めるメソッド
19         戻り値：引数で指定された点までの距離(double)
20         引数　：他の点(Point型)
21      */
22      public double distanceToOtherPoint(Point opt){
23          double diffX=x-opt.x;//x座標の差
24          double diffY=y-opt.y;//y座標の差
25          return Math.sqrt(diffX*diffX+diffY*diffY);
26      }
27  }
```

ここで、Math.sqrt()は平方根を求めるメソッドです。数学的な関数に対応するメソッドは、Math.××()として利用できます[7]。

また、フィールドとメソッド内のローカル変数の違いに注意して下さい。ローカル変数はメソッドの実現のためにそのメソッド内だけで用いられる変数で、オブジェクトの属性を表わすものではありません。リスト15のメソッドdistanceToOtherPoint()中で宣言されている変数diffX等は、x座標の差を一次的に蓄えているだけということに注意しましょう[8]。

このように定義されたメソッドは、フィールドの利用と同様に、ドット演算子で参照変数とメソッド名を結合して利用します。

参照型変数名.メソッド名(引数リスト)

したがって、主に次のように利用されることになります。オブジェクト毎に、メソッドの“効果”が異なることに注意して下さい。

[7]数学関数のより詳しい利用法は、API仕様や他の文献で調べること。

[8]diffXを用いなくとも、メソッドdistanceToOtherPoint()は実現できます。どのような変数を用意するかは、プログラマのセンスの問題です。ただし、必要以上に少い変数でソースコードを記述すると、可読性が著しく低下します。

```
pt1.distanceToOrigin() //原点までの距離を返す。
pt2.distanceToOtherPoint(pt1) //他の点までの距離を返す。
```

それでは、リスト 16 に、Point クラス (Ver.4) を利用するソースコードを示します。

リスト 16:TestPoint.java (Ver.4)

```
1  /*
2     Point クラス (Ver4) を利用するクラス。
3     main メソッドだけのクラス。
4  */
5  public class TestPoint{
6      /******************フィールド************************/
7      //なし
8
9      /******************メソッド*************************/
10     //main
11     public static void main(String[] argv){
12         Point pt1=new Point();//点 1
13
14         pt1.x=0.0;
15         pt1.y=1.0;
16         System.out.println("pt1=("+pt1.x+","+pt1.y+")");
17
18         double dist=pt1.distanceToOrigin(); //距離
19         System.out.println("点 1 から原点までの距離は"+dist+"です。");
20
21
22         Point pt2=new Point();//点 2
23         pt2.x=2.0;
24         pt2.y=3.0;
25
26         System.out.println("pt2=("+pt2.x+","+pt2.y+")");
27
28         dist=pt2.distanceToOtherPoint(pt1);
29         System.out.println("点 2 から点 1 までの距離は"+dist+"です。");
30
31         return;
32     }
```

```
33    }
```

図 4.3 に、Java におけるメソッドの概念図を示します。

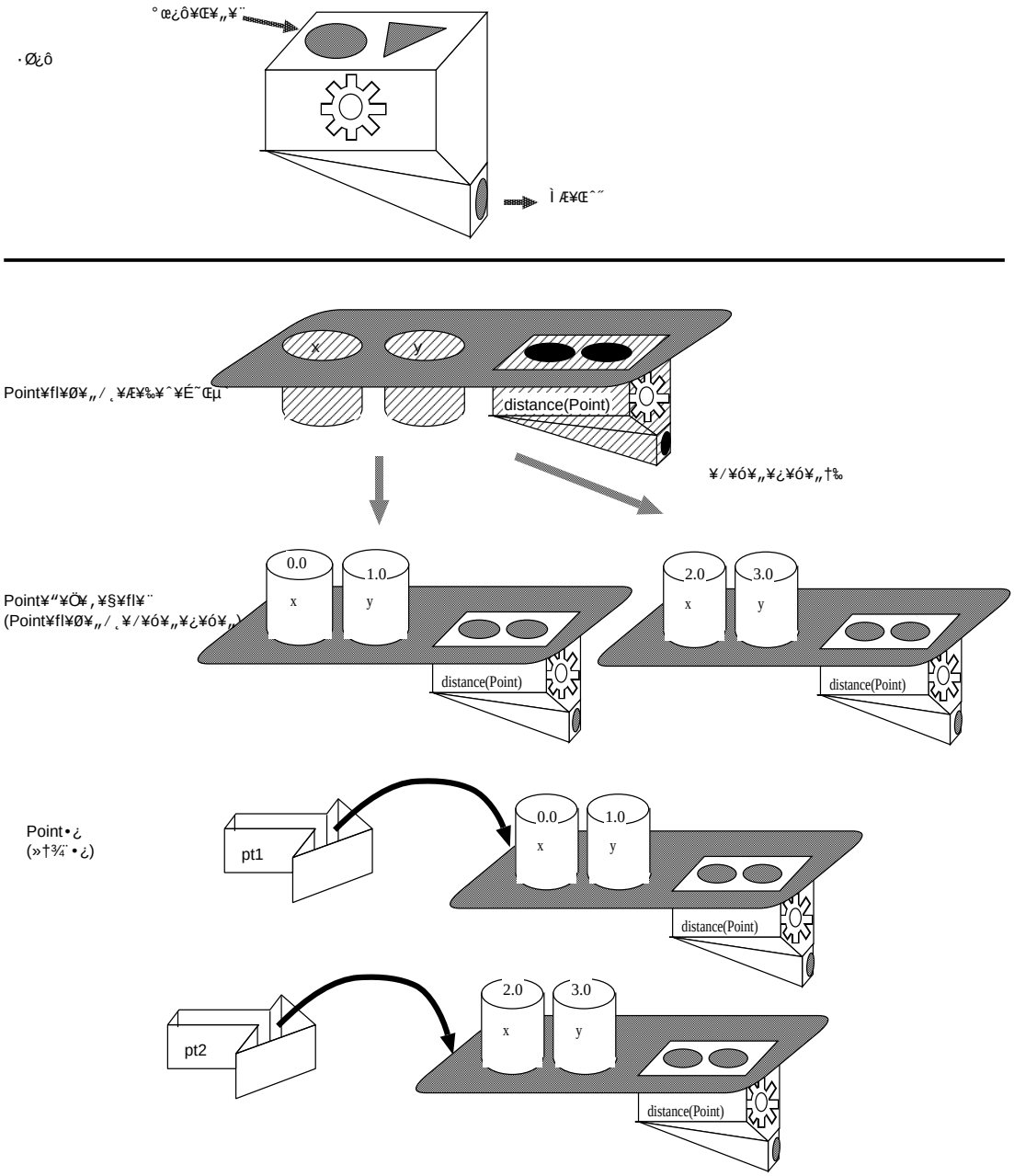


図 4.3: 関数とメソッド

4.3.5 コンストラクタ

これまで、オブジェクトの生成には、new 演算子を用いて次の形で記述していました。

new クラス名 ();

実は、new 演算子の直後にある「クラス名 ()」は**コンストラクタ**と呼ばれるものであり、インタンス化の時に実行する一連の処理を行なうメソッドのようなものです。(実際には、メソッドではありません。) コンストラクタの定義は、メソッド名をクラス名としたメソッドと同じように定義できます。すなわち、次のように定義できます。

コンストラクタ ::= クラス名 (引数リスト) ブロック

ここで、注意なのですが、コンストラクタには戻り値がなく、コンストラクタ定義のときには戻り値の型は記述しません[9]。

コンストラクタは、インスタンスのフィールドの初期化等、インスタンスの初期状態を指定する際に利用されます。リスト 17 に、Point クラスにコンストラクタを追加した例を示します。

リスト 17:Point.java (Ver.5)

```
1  /*コンストラクタを定義するサンプルプログラム*/
2  public class Point{
3      /**********************フィールド*****************************/
4      double x; //x 座標
5      double y; //y 座標
6
7      /**********************コンストラクタ**********************/
8      /*
9         引数   : (x 座標 (double) ,y 座標 (double));
10     */
11     Point(double inX,double inY){
12         x=inX;
13         y=inY;
14     }
15
16     /**********************メソッド*****************************/
17     /*
18        原点までの距離を求めるメソッド
19        戻り値 : 原点までの距離 (double)
20        引数   : なし
```

[9]コンストラクタに戻り値の型が無いのに対し、メソッドには必ず戻り値の型が必要です。戻り値が無いメソッドには、「void」という戻り値の型を持ちます。

```
21      */
22      public double distanceToOrigin(){
23          return Math.sqrt(x*x+y*y);
24      }
25
26      /*
27         他の点までの距離を求めるメソッド
28         戻り値:引数で指定された点までの距離(double)
29         引数   :他の点(Point型)
30      */
31      public double distanceToOtherPoint(Point opt){
32          double diffX=x-opt.x;//x座標の差
33          double diffY=y-opt.y;//y座標の差
34          return Math.sqrt(diffX*diffX+diffY*diffY);
35      }
36  }
```

コンストラクタを定義すると、その定義にしたがって、次の書式で利用できます。

```
new クラス名(引数リスト);
```

リスト18に、Pointクラス(Ver.5)を利用するプログラムを示します。

リスト 18:TestPoint.java (Ver.5)

```
1   /*
2      Pointクラス(Ver5)を利用するクラス。
3      mainメソッドだけのクラス。
4   */
5   public class TestPoint{
6       /******************フィールド************************/
7       //なし
8
9       /******************メソッド*************************/
10      //main
11      public static void main(String[] argv){
12          Point pt1=new Point(0.0,1.0);//点1
13          System.out.println("pt1=("+pt1.x+","+pt1.y+")");
14          double dist=pt1.distanceToOrigin(); //距離
```

```
15          System.out.println("点1から原点までの距離は"+dist+"です。");
16
17
18          Point pt2=new Point(2.0,3.0);//点2
19          System.out.println("pt2=("+pt2.x+","+pt2.y+")");
20          dist=pt2.distanceToOtherPoint(pt1);
21          System.out.println("点2から点1までの距離は"+dist+"です。");
22
23          return;
24      }
25  }
```

一度、コンストラクタを定義すると、定義以外の引数リストではインスタンス化されないことに注意しましょう。

4.3.6 多重定義 (オーバーロード)

C言語では、異なる関数でなければ、引数リストの形や種類を変えることはできませんでした。すなわち、C言語では、1つの関数名で、異なる引数リストを定義することはできませんでした。これに対して、Javaでは、引数リストが異なれば、同じ名前のメソッドを定義できます。このことを、**多重定義 (オーバーロード)** といいます。メソッドだけでなく、コンストラクタも**多重定義**することができます。

コンストラクタを多重定義した例を、リスト19に示します。

リスト 19:Point.java (Ver.6)

```
1   /*多重定義のサンプルプログラム*/
2   public class Point{
3       /**********************フィールド*****************************/
4       double x; //x座標
5       double y; //y座標
6
7       /**********************コンストラクタ**********************/
8
9       /*
10          2つの座標から点を作成する。
11          引数   :(x座標(double) ,y座標(double));
12      */
```

```
13      Point(double x,double  y){
14          this.x=x;
15          this.y=y;
16      }
17
18      /*
19         点のコピーを作成する。
20         引数   : 点 (Point)
21      */
22      Point(Point pt){
23          x=pt.x;
24          y=pt.y;
25      }
26
27      /*
28         デフォルトの点の作成。
29         (引数がなければ原点のコピーを作成する。)
30         引数   : なし
31      */
32      Point(){
33          this(0.0,0.0);
34      }
35
36      /**********************メソッド*****************************/
37      /*
38         原点までの距離を求めるメソッド
39         戻り値 : 原点までの距離 (double)
40         引数   : なし
41      */
42      public double distanceToOrigin(){
43          return Math.sqrt(x*x+y*y);
44      }
45
46      /*
47         他の点までの距離を求めるメソッド
48         戻り値 : 引数で指定された点までの距離 (double)
49         引数   : 他の点 (Point 型)
50      */
51      public double distanceToOtherPoint(Point opt){
```

```
52            double diffX=x-opt.x;//x 座標の差
53            double diffY=y-opt.y;//y 座標の差
54            return Math.sqrt(diffX*diffX+diffY*diffY);
55        }
56    }
```

このリスト 19 における、this は自分のオブジェクトを指す参照です。したがって、this.x とすると、自分のフィールドの x を指し示すことになります。この this 参照を用いることで、引数の x とフィールドの x を区別することができます[10]。図 4.3 に、this 参照の概念図を示します。

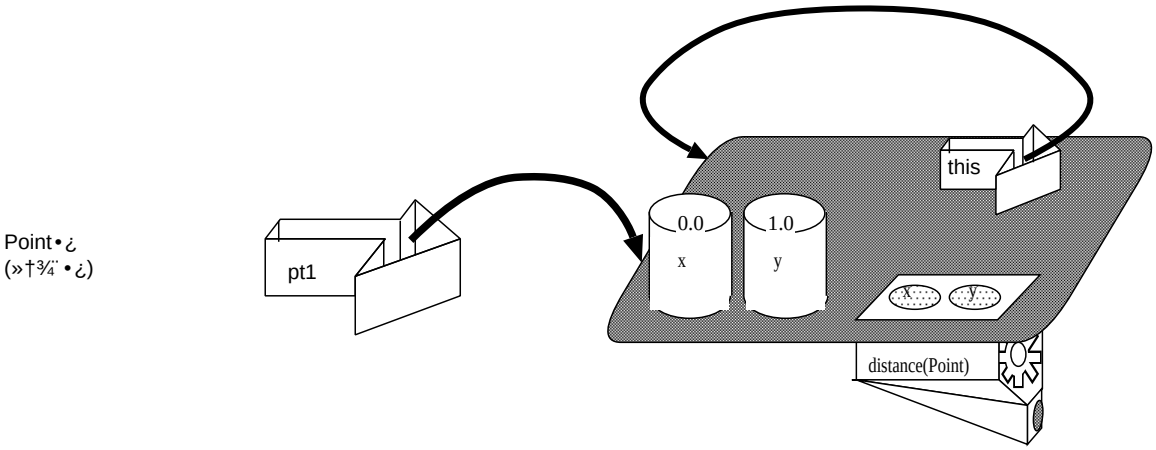


図 4.4: this 参照

また、コンストラクタ定義中に、他のコンストラクタを利用することもできます。この方法は、次のように行ないます。

this(引数リスト)

this の 2 つの使い方に注意して下さい。

リスト 20 に、Point クラス (Ver.6) を利用するプログラムを示します。引数リストの形にしたがって、どのコンストラクタが利用されるかが、自動的に選択されます。

リスト 20:TestPoint.java (Ver.6)

```
1   /*
2       Point クラス (Ver6) を利用するクラス。
```

[10] もちろん変数名を変えて取り扱うこともできるのですが、this を用いた方が便利なことも多いです。

```
 3  */
 4  public class TestPoint{
 5      public static void main(String[] argv){
 6          Point pt1=new Point(); //点1
 7          System.out.println("pt1=("+pt1.x+","+pt1.y+")");
 8
 9          Point pt2=new Point(1.0,2.0); //点2
10          System.out.println("pt2=("+pt1.x+","+pt1.y+")");
11
12          Point pt3=new Point(pt2); //点3
13          System.out.println("pt3=("+pt1.x+","+pt3.y+")");
14
15
16          double dist=pt3.distanceToOtherPoint(pt1);
17          System.out.println("点3から点1までの距離は"+dist+"です。");
18
19          return;
20      }
21  }
```

4.3.7 アクセス制御

ここでは、アクセス制御について説明します。これまでは, 他のクラスのフィールドであっても、「参照.フィールド」として直接利用[11]していました[12]。しかし一般に、このようにフィールドを直接に操作することは、プログラムの間違いの原因になりやすいので、行なうべきではありません。

例えば、現実世界のオブジェクトである車においても、車のエンジンやキャブレター等は、利用者は直接に接触して制御することはあまりありません。このように、オブジェクトの属性や操作には、他のオブジェクトから闇雲に変更を加えられるべきではありません。

Javaには、このような他のクラスのオブジェクトに対する**アクセス**を制御する仕組みが備わっています。具体的には、クラス定義の際に、**アクセス修飾子**を用いて、どのメンバにアクセスが可能であるかを明示します。

主な、アクセス修飾子には、次のようなものがあります。

ここで、アクセス修飾子が無いことにも意味があることに注意しましょう。しかし、同じディレクトリにあるクラスであれば、他のクラスのアクセス修飾子が無いメンバにもア

[11] ここでの利用とは、代入や計算のことです。すなわち、式の左辺や右辺に現れることです。

[12] なお、このように、他のクラスのフィールドやメソッドを利用することを、それらのメンバに**アクセス**するともいいます。

クセスすることができます。なぜなら、同じディレクトリであれば同じパッケージ[13]に属することになるからです。

また、protectedの説明にサブクラスとありますが、サブクラスについては後述します。

表 4.1: アクセス修飾子とアクセス制御

制御可能範囲	アクセス修飾子	アクセス制御の意味
狭い	private	同じクラスのメソッドからのみアクセス可能。
↑	(なし)	パッケージを意味する。同じクラスの他に、同じパッケージに属するクラスのメソッドからもアクセス可能。
↓	protected	同じパッケージに属するクラスか、あるいは、クラスのサブクラスのメソッドからアクセス可能。
広い	public	すべてのクラスのメソッドからアクセス可能。

このような、アクセス制御を用いた例を、リスト 21 に示します。

リスト 21:Point.java (Ver.7)

```
1  /*アクセス制限を調べるサンプルプログラム*/
2  public class Point{
3      /**********************フィールド*****************************/
4      private double x; //x 座標
5      private double y; //y 座標
6
7      /**********************メソッド*****************************/
8
9      /*
10         戻り値 : x 座標
11         引数   : なし
12     */
13     public double getX(){
14         return x;
15     }
16
17     /*
18         戻り値 : void
19         引数   : x 座標
```

[13]パッケージとは、クラスの仲間を集めたものです。パッケージの詳細については後述します。

```
20        */
21        public void setX(double x){
22            this.x=x;
23            return;
24        }
25
26        /*
27           戻り値 : y 座標
28           引数   : なし
29        */
30        public double getY(){
31            return y;
32        }
33
34        /*
35           戻り値 : void
36           引数   : y 座標
37        */
38        public void setY(double y){
39            this.y=y;
40            return;
41        }
42    }
43
44
45
46
```

この例のように、フィールドはすべて **private** にして外部から直接アクセスできないようにするといいでしょう。また、それらのアクセス制限されたフィールドは、メソッドを通じて利用するできるようにします。すなわち、**get メソッド**によって値を獲得したり、**set メソッド**によって値を設定したりできるようにするといいでしょう。このように、フィールドへ直接のアクセスを避けることによって、プログラムに間違いが混入することを減少させることができます。

リスト 22 に、Point クラス (Ver.7) を間違って利用するプログラムを示します。

リスト 22:**WrongTestPoint.java**

```
1   /*
2      Point クラス (Ver7) を利用する間違っているクラス。
3   */
4   public class WrongTestPoint{
5       public static void main(String[] argv){
6           Point pt=new Point(); //点
7
8           pt.x=0.0;
9           pt.y=0.0;
10
11          System.out.println("pt1=("+pt.x+","+pt.y+")");
12          return;
13      }
14  }
```

リスト 22 をコンパイルしようとすると、次のようにエラー表示されます。

```
$ ls
Point.class  Point.java  WrongTestPoint.java
$ javac WrongTestPoint.java
WrongTestPoint.java:8: x は Point で private アクセスされます。
        pt.x=0.0;
          ^
WrongTestPoint.java:9: y は Point で private アクセスされます。
        pt.y=0.0;
          ^
WrongTestPoint.java:11: x は Point で private アクセスされます。
        System.out.println("pt1=("+pt.x+","+pt.y+")");
                                    ^
WrongTestPoint.java:11: y は Point で private アクセスされます。
        System.out.println("pt1=("+pt.x+","+pt.y+")");
                                              ^
エラー 4 個
$
```

リスト 23 に、Point クラス (Ver.7) を正しく利用するプログラムを示します。

リスト 23:CorrectTestPoint.java

```
1   /*
```

```
2      Point クラス (Ver7) を正しく利用するクラス。
3   */
4   public class CorrectTestPoint{
5       public static void main(String[] argv){
6           Point pt=new Point(); //点
7
8           double x=0.0;
9           double y=2.0;
10
11          //値の設定
12          pt.setX(1.0);
13          pt.setY(y);
14
15          //値の獲得
16          x=pt.getX();
17          System.out.println("pt=("+x+","+pt.getY()+")");
18
19          return;
20      }
21  }
```

第5章　クラス階層

前章では、Javaプログラミングの基本単位であるクラスについて学びました。これで、Javaを用いて、基本的なプログラミングが行なえるようになったはずです。

本章では、クラスを用いたより高度なプログラミング技法について学びます。本章を通じ、小さいクラスを組み合わせて大きなプログラムを作成する技能を修得して下さい。また、本章では、オブジェクト指向の重要な概念である「継承」および「インターフェース」についても説明します。

5.1　複雑なクラス

本節では、クラスを組み合わせて新しいクラスを定義する一例を示します。

前述しましたが、クラスのフィールドの型には、基本型以外に参照型を用いることもできます。この仕組みを使えば、クラスを組み合わせることができます。例えば、フィールドを定義する際に自分で定義したクラス型を用いれば、そのクラスのインスタンスを一種の部品とするクラスが定義できます。以下では、Pointクラスと Pointオブジェクトの参照を表わす Point型を用いて、三角形を意味する Triangleクラスを定義します。なお、バージョン番号 (Ver.xx) が同じクラスは、同じディレクトリに置いて相互に利用するようにして下さい。

まず、Pointクラスを示します。前章のプログラムに自分自身の座標値を表示する printメソッドを追加したものです。

リスト 24:Point.java(Ver5.1)

```
1   /*Point クラス*/
2   public class Point{
3       /**********************フィールド*****************************/
4       private double x; //x 座標
5       private double y; //y 座標
6
7       /**********************コンストラクタ**********************/
8
9       /*
```

```
10          2つの座標から点を作成する。
11          引数　:(x座標(double) ,y座標(double));
12      */
13      Point(double x,double  y){
14          this.x=x;
15          this.y=y;
16      }
17
18      /*
19          点のコピーを作成する。
20          引数　:点(Point)
21      */
22      Point(Point pt){
23          x=pt.x;
24          y=pt.y;
25      }
26
27      /*
28          デフォルトの点の作成。
29          (引数がなければ原点のコピーを作成する。)
30          引数　:なし
31      */
32      Point(){
33          this(0.0,0.0);
34      }
35
36      /*******************メソッド*****************************/
37      /*
38          戻り値:x座標
39          引数　:なし
40      */
41      public double getX(){
42          return x;
43      }
44
45      /*
46          戻り値:void
47          引数　:x座標
48      */
```

```
49      public void setX(double x){
50          this.x=x;
51          return;
52      }
53
54      /*
55         戻り値:y 座標
56         引数  :なし
57      */
58      public double getY(){
59          return y;
60      }
61
62      /*
63         戻り値:void
64         引数  :y 座標
65      */
66      public void setY(double y){
67          this.y=y;
68          return;
69      }
70
71      /*
72         原点までの距離を求めるメソッド
73         戻り値:原点までの距離(double)
74         引数  :なし
75      */
76      public double distanceToOrigin(){
77          return Math.sqrt(x*x+y*y);
78      }
79
80      /*
81         他の点までの距離を求めるメソッド
82         戻り値:引数で指定された点までの距離(double)
83         引数  :他の点(Point 型)
84      */
85      public double distanceToOtherPoint(Point opt){
86          double diffX=x-opt.x;//x 座標の差
87          double diffY=y-opt.y;//y 座標の差
```

```
88
89          return Math.sqrt(diffX*diffX+diffY*diffY);
90      }
91
92      /*
93          自分の座標値を表示するメソッド
94          戻り値：なし
95          引数　：なし
96      */
97      public void print(){
98          System.out.print("("+x+","+y+")");//改行なし
99          return;
100     }
101 }
```

この Point 型を (Ver5.1) 利用して、次のように三角形クラス (Ver5.1) を定義することができます。

リスト 25:Triangle.java(Ver5.1)

```
1   /* 三角形クラス Point 型を利用*/
2   public class Triangle{
3       /***********フィールド*****************/
4       private Point a;//点 A
5       private Point b;//点 B
6       private Point c;//点 C
7
8       /***********コンストラクタ*************/
9       /*
10        与えられた 3 つの点を頂点とする三角形を作る。
11        各頂点は与えられた点のコピーとする。
12        引数:3 点
13      */
14      Triangle(Point a,Point b, Point c){
15          this.a=new Point(a);
16          this.b=new Point(b);
17          this.c=new Point(c);
18      }
```

```
19
20      /*
21        6つの座標から三角形を作る。
22        引数:3つの座標を表す6座標(double)
23      */
24      Triangle(double aX,double aY,double bX,double bY,double cX,double cY){
25          this.a=new Point(aX,aY);
26          this.b=new Point(bX,bY);
27          this.c=new Point(aX,bY);
28      }
29
30      /*
31        三角形のコピーを作る。
32        引数:三角形(Triangle型)
33      */
34      Triangle(Triangle t){
35          this.a=new Point(t.getA());
36          this.b=new Point(t.getB());
37          this.c=new Point(t.getC());
38      }
39
40      /*
41        デフォルトの三角形(すべての頂点が原点)。
42        引数:なし
43      */
44      Triangle(){
45          this.a=new Point();
46          this.b=new Point();
47          this.c=new Point();
48      }
49
50      /**********メソッド**************/
51      //点A
52      public Point getA(){
53          return a;
54      }
55
56      public void setA(Point a){
57          this.a=new Point(a);
```

```
58          return;
59      }
60
61      public void setA(double aX,double aY){
62          this.a=new Point(aX,aY);
63          return;
64      }
65
66      //点B
67      public Point getB(){
68          return b;
69      }
70
71      public void setB(Point b){
72          this.b=new Point(b);
73          return;
74      }
75
76      public void setB(double bX,double bY){
77          this.b=new Point(bX,bY);
78          return;
79      }
80
81      //点C
82      public Point getC(){
83          return c;
84      }
85
86      public void setC(Point c){
87          this.c=new Point(c);
88          return;
89      }
90
91      public void setc(double cX,double cY){
92          this.c=new Point(cX,cY);
93          return;
94      }
95
96      /*
```

```
97          自分自身の各頂点の座標値を表示するメソッド
98          戻り値：なし
99          引数：なし
100         */
101        public void print(){
102            a.print();
103            System.out.print("-");
104            b.print();
105            System.out.print("-");
106            c.print();//すべて改行なし
107        }
108    }
```

図 5.1 に、クラスを組み合わせて新しいクラスを構成する概念図を示します。

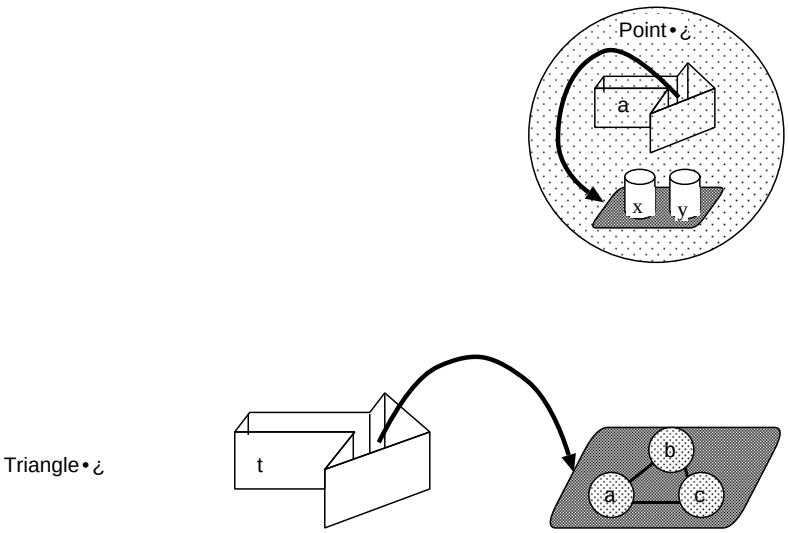


図 5.1: 複雑なクラス

この Triangle クラスを利用するクラスを次に示します。

リスト 26:TestTriangle.java(Ver5.1)

```
1   /*  Triangle クラスの利用を調べるクラス*/
2   public class TestTriangle{
3       //main メソッド
```

```
4       public static void main(String[] argv){
5           //三角形1
6           Triangle t1=new Triangle();
7           System.out.print("t1=");
8           t1.print();
9           System.out.println();
10
11          //三角形2
12          Point a=new Point();
13          Point b=new Point(3.0,0.0);
14          Point c=new Point(0.0,4.0);
15
16          Triangle t2=new Triangle(a,b,c);
17          System.out.print("t2=");
18          t2.print();
19          System.out.println();
20
21          //三角形3
22          Triangle t3=new Triangle(t2);
23          System.out.print("t3=");
24          t3.print();
25          System.out.println();
26
27          return;
28      }
29  }
```

このように、クラスを組み合わせることによって、より複雑なクラスを定義することができます。

5.2　継承

本節では、オブジェクト指向における重要な概念の一つである「継承」について説明します。まず、一般的なクラスの継承について述べ、次に Java 言語における継承の表現法について述べます。

5.2.1 オブジェクト指向における継承

クラスとは、オブジェクトの集合でした。逆の言い方をすると、「オブジェクトは、あるクラスに属している。」ということができます。しかし、オブジェクトは、ただ一つのクラスにだけ属するわけではありません。このことを、現実世界にある「モノ」に照らし合わせて見てみます。

クラス「車」のインスタンスの一つであるオブジェクトを「車 1」とします。このとき、「車 1」は、当然クラス「車」に属しています。しかし、「車 1」は乗り物でもあるので、クラス「乗り物」にも属していることがわかります。このように、一つのオブジェクトは、通常いくつかのクラスに属することになります。よく考えてみると、クラス「車」に属するすべてのオブジェクトは、クラス「乗り物」にも属します。したがって、クラス「車」を包含するように、クラス「乗り物」という概念があることがわかります。このように、クラス「車」は、クラス「乗り物」の一部とみなすことができます。このようなとき、クラス「車」はクラス「乗り物」を**継承**あるいは**拡張**しているといいます。乗り物の性質である「なにかを載せる」という性質を、車は受け継いで (継承して) いることに注意しましょう。また、「乗り物の中で、エンジンとタイヤがあるものが車である」という見方もできます。すなわち、『「乗り物」により詳しい性質を付け加えて拡張したものが「車」である』という見方もできます。この見方によると、「車」は「乗り物」を拡張しているといえます。

このような継承は、一段階とは限りません。例えば、「車 1」は機械でもあるので、クラス「機械」にも属します。ここで、クラス「乗り物」に属するすべてのオブジェクトは、クラス「機械」に属するとしましょう[1]。このとき、クラス「乗り物」はクラス「機械」を拡張しています[2]。このように、一般的には、クラスは階層構造を形成します。このようなクラスの階層構造を**クラス階層**といいます。

オブジェクト指向における継承は、集合論に基づいて説明することもできます。クラスはオブジェクトの集合でしたので、例えば、

$$「車」 = \{x | x は世界にある全ての車\}$$

$$「乗り物」 = \{x | x は世界にある全ての乗り物\}$$

といった集合とします。また、

$$「車」 \subset 「乗り物」$$

という関係があるとします。このとき、「車」は「乗り物」の**部分集合 (Sub Set)** であり、「乗り物」は「車」の**スーパーセット (Super Set)** です。この集合論の用語を援用して次のような用語が用いられることもあります。クラス階層において、クラス 1 がクラス 2 を継承しているとき、クラス 1 をクラス 2 の**サブクラス (Sub Class)** といいます。また、

[1] このように、現実世界の概念は境界がはっきりしているものだけではありません。しかし、Java 等のオブジェクト指向言語でプログラミングするためには、各概念の境界を明確にする必要があります。

[2] ベン図においては、狭いクラスの方が広いクラスを拡張していることに注意して下さい。

クラス 1 がクラス 2 を継承しているとき、クラス 2 をクラス 1 の**スーパークラス (Super Class)** といいます。

図 5.2 に、クラス階層の例を示します。

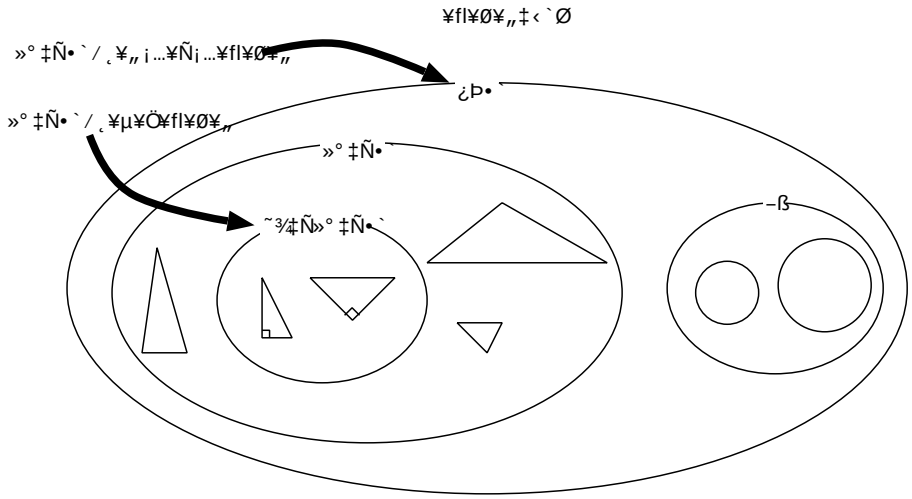


図 5.2: クラス階層

図 5.2 では、「図形」は「三角形」のスーパークラスであり、「直角三角形」は「三角形」のサブクラスであることを示しています。また、「図形」は「円」のスーパークラスにもなっていることに注意しましょう。

5.2.2 Javaにおける継承

オブジェクト指向の考え方では、あるクラスは複数のクラスを継承することがあるのですが、Java のプログラムではそれぞれのクラスに対して唯一つのスーパークラスしか指定できません。このことは、図 5.2 のようなベン図を用いた場合にクラスの境界を表わす“枠”が交差するような関係は、Java の継承では表現することができないことを意味します[3]。

Java での継承は、クラス定義の際に定義します。BNF 記法では、次のようになります[4]。

継承指定 ::= [extends スーパークラス名] [implements インターフェース名, …]

ここで、インターフェースににについては後述します。したがって、主に、次の書式で継承は表現されます。

```
public class クラス名 extends スーパークラス名{
        /*フィールド定義*/
        /*コンストラクタ定義*/
        /*メソッド定義*/
}
```

```
public class RightTraiangle extends Traiangle{

}
```

このように、extends を用いることによって、あるスーパークラスのメンバを継承するクラス (サブクラス) を定義できます。サブクラスの定義では、スーパークラスのメンバに“追加する”性質だけをメンバ[5]として記述します。

節 5.1 のリスト 26 で定義した三角形クラスを継承する「直角三角形」クラスを、リスト 27 に示します。

リスト 27:RightTriangle.java(Ver5.1)

```
1   /*直角 3 角形クラス。継承を調べるためのサンプルプログラム*/
2   public class RightTriangle extends Triangle{
3       /**************フィールド***************/
4       Point r;//直角な点
5
6       /**************コンストラクタ***********/
7       /*
8         指定がなければ点 A を直角とする。
```

[3]枠が交差するような複雑なクラス階層は、クラスの継承ではなくて**インターフェース**等の別の表現法を用います。

[4]p.26 のクラス定義を参照すること。

[5]フィールドやメソッド

```
 9          引数:点Aが直角な直角三角形(Point型)
10        */
11        RightTriangle(Triangle t){
12            super(t);//コピーの作成
13            setR(this.getA());
14        }
15
16        /*
17          デフォルトの直角三角形を生成する。
18          (すべてが原点、点Aが直角)
19          引数:なし
20        */
21        RightTriangle(){
22            super();//コピーの作成
23            setR(this.getA());
24        }
25
26        /*
27          指定された点が直角な直角三角形。
28          引数:直角な直角三角形(Point型)、直角な点
29        */
30        RightTriangle(Triangle t,Point r){
31            super(t);//コピーの作成
32            if(r==t.getA()){
33                this.r=getA();
34            }else if (t.getB()==r){
35                this.r=getB();
36            }else if(t.getC()==r){
37                this.r=getC();
38            }else{
39                System.out.println("ここは実行されないはず。");
40            }
41
42        }
43
44        /**********メソッド **************************/
45        public Point getR(){
46            return r;
47        }
```

```
48
49      public void setR(Point r){
50          if(r==getA()){
51              this.r=r;
52          }else if(r==getB()){
53              this.r=r;
54          }else if(r==getC()){
55              this.r=r;
56          }else{
57              System.out.println("ここは実行されないはず。");
58          }
59          return;
60      }
61
62      public double area(){
63          double base=0.0;//底辺
64          double height=0.0;//高さ
65
66          Point npt1;//直角でない点1
67          Point npt2;//直角でない点2
68
69          if(r==getA()){//点Aが直角
70              npt1=getB();
71              npt2=getC();
72          }else if (r==getB()){//点Bが直角
73              npt1=getC();
74              npt2=getA();
75          }else{//点Cが直角
76              npt1=getA();
77              npt2=getB();
78          }
79
80          base=r.distanceToOtherPoint(npt1);
81          height=r.distanceToOtherPoint(npt2);
82
83          return base*height/2.0;
84      }
85
86      /*
```

```
87          スーパークラスのメソッドをオーバーライドする。
88        */
89        public void print(){
90            super.print(); //super を削除してその効果を調てみて下さい。
91            System.out.print(":R("+r.getX()+","+r.getY()+")");
92            return;
93        }
94    }
```

図 5.3 に、RightTriangle クラスの概念図を示します。

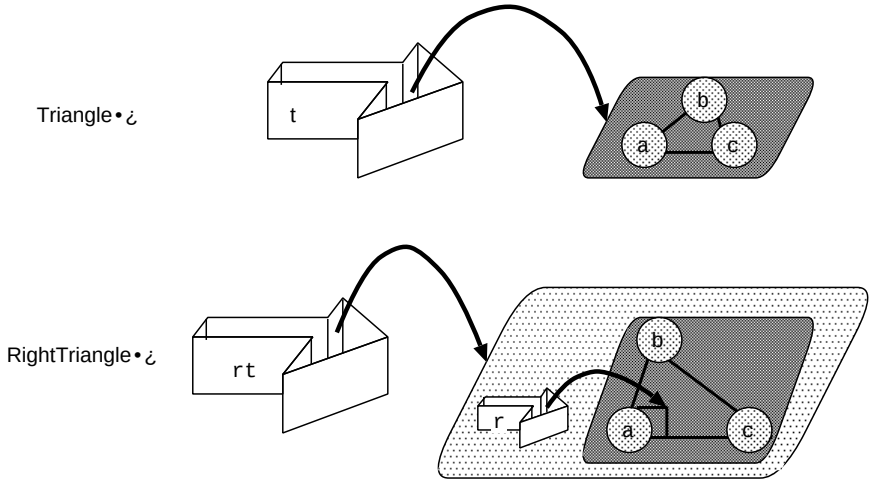


図 5.3: 継承によるサブクラス定義

このように定義されたサブクラスを利用する際には、サブクラス型の参照変数にドット演算子を適用することによって、サブクラスで定義されていないメンバであっても、利用りようすることができます。すなわち、次のような書式を用いることによって、あたかも、(スーパークラスのメンバが) サブクラスで定義されていうように用いることができます。

サブクラス型参照. フィールド名
サブクラス型. メソッド名 (引数リスト)

```
RightTriangle rt = new RightTriangle();
rt.a=new Point();//フィールドへのアクセス (ここではアクセス制限よりエラー)
pt=rt.getA();//メソッドへのアクセス
```

サブクラス定義はスーパークラスにメンバを “追加する” と述べましたが、サブクラスにおいてはスーパークラスと同じ名前のメンバ[6]を定義することもできます。特に、引数

[6]フィールドやメソッド

リストまで含めて同じメソッドをサブクラスで定義することを、メソッドの**上書き (オーバーライド)**[7]といいます。リスト 27における print() メソッドは、リスト 25 の print() メソッドをオーバーライドしています。

また、サブクラスとスーパークラスではインスタンス化の仕組みが異なるので、サブクラスにおいてコンストラクタを定義する必要があります。しかし、サブクラスのインスタンス化には、スーパークラスのインスタンス化とほとんど同じことが多いです。このような場合には、サブクラスのコンストラクタにおいて、スーパークラスのコンストラクタを利用できます。すなわち、

リスト 27にもありますが、

super(引数リスト)

とすれば、スーパークラスのスーパークラスのメンバやコンストラクタを利用することができます。また、スーパークラスのフィールドやメソッドも、super という特別な参照を用いてアクセスできます。super は、継承しているスーパクラスへの参照です。この super 参照を用いるいことによって、明示的にスーパークラスのメンバにアクセスできます[8]。

super. フィールド名
super. メソッド名 (引数リスト)

次のリスト 28 に、直角三角形クラスを利用するプログラムを示します。

リスト 28:TestRightTriangle.java(Ver5.1)

```
1   /*  RightTriangle クラスの利用を調べるクラス*/
2   public class TestRightTriangle{
3       //main メソッド
4       public static void main(String[] argv){
5           Point a=new Point();
6           Point b=new Point(3.0,0.0) ;
7           Point c=new Point(0.0,4.0);
8
9           //3 角形 1
10          Triangle t=new Triangle(a,b,c);
11          a=t.getA();
12          b=t.getB();
13          c=t.getC();
14
15          System.out.print("t=");
```

[7]オーバロードと区別して下さい。オーバロードでは引数リストが異なりましたが、オーバーライドでは引数リストまで同一です。

[8]this 参照と対応して理解するといいでしょう。

```
16          t.print();
17          System.out.println();
18
19          //直角3角形1
20          RightTriangle rt1=new RightTriangle(t);
21          a=t.getA();
22          b=t.getB();
23          c=t.getC();
24          Point r=rt1.getR();//直角を示す参照変数
25
26          System.out.print("rt1=");
27          rt1.print();
28          System.out.println();
29          System.out.println("rt1の面積は"+rt1.area()+"です。");
30
31          //直角3角形2
32          a=new Point(1.0,2.0);
33          b=new Point(3.0,2.0) ;
34          c=new Point(3.0,-1.0);
35          t=new Triangle(a,b,c);
36          r=t.getB();//コピーが生成されるので、bとt.getB()は異なる。
37
38          RightTriangle rt2=new RightTriangle(t,r);
39          a=t.getA();
40          b=t.getB();
41          c=t.getC();
42          r=rt2.getR();
43
44          System.out.print("rt2=");
45          rt2.print();
46          System.out.println();
47          System.out.println("rt2の面積は"+rt2.area()+"です。");
48
49          return;
50      }
51  }
```

Javaでは、スーパークラス型の参照変数には、サブクラスのインスタンスへの参照も代

入することができます。すなわち、例えば、次のようにスーパークラス型の変数にサブクラス型への参照を代入することができます。このことは、「直角三角形は三角形の一種である」という文からも理解できます。

スーパークラス型参照変数= サブクラス型参照

```
Triangle t = new RightTriangle();
```

リスト 29:ReferSubClass.java(Ver5.1)

```
1   /*サブクラス参照を調べるサンプルプログラム*/
2   public class ReferSubClass{
3       public static void main(String[] argv){
4           Point a=new Point();
5           Point b=new Point(3.0,0.0);
6           Point c=new Point(0.0,4.0);
7
8           RightTriangle rt =new RightTriangle(new Triangle(a,b,c));
9           Triangle t=new RightTriangle(new Triangle(a,b,c));
10
11          rt.print();
12          System.out.println();
13          t.print();
14          System.out.println();
15
16          return;
17      }
18  }
```

図 5.4 に、クラス階層における参照型の取り扱いを表す概念図を示します。

サブクラスのインスタンスをスーパークラス型の変数に代入したとしても、オブジェクトのメンバはサブクラスのままであることに注意して下さい。

5.3 抽象クラス

5.3.1 抽象クラス定義

図 5.2 における「図形」を考えましょう。「図形」は、「三角形」のスーパークラスであり、「円」のスーパークラスでもあります。図形の一般の共通の機能として、面積を求めることが考えられます。そこで、図形クラスには、面積を求めるメソッドがあると便利で

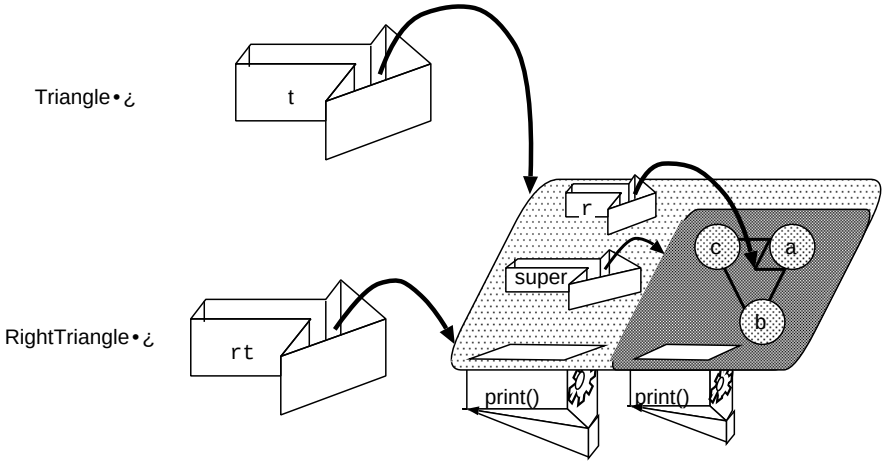


図 5.4: クラス階層における参照型

す。しかし、三角形と円では面積の求め方がまるで異なります。そこで、具体的な求め方は個々のサブクラスで定義することにして、図形クラスでは「面積を求める」機能を持つという事実だけを定義することができます。このように、具体的な求め方を示さずに、戻り値型、メソッド名、および引数リストだけを示したメソッドを**抽象メソッド**といいます[9]。また、抽象メソッドを持つようなクラスを**抽象クラス**といいます。例えば、面積を求める機能を持たせた図形クラスは、抽象クラスであり、「面積を求める」というメソッドが抽象メソッドになります。

Java では、クラス定義の際に、クラス名やメソッド名の前に `abstract` を指定することで、抽象クラスや抽象メソッドを示します。したがって、以下ような書式で、抽象クラスを定義します。

```
public abstract class サブクラス名 [継承指定]{
        /*フィールド定義*/
        /*コンストラクタ定義*/
        /*メソッド定義*/
        [アクセス修飾子] abstract 抽象メソッド名(引数リスト);
                        .
                        .
                        .
}
```

[9]C 言語におけるプロトタイプ宣言と対応して理解するといいでしょう。

```
public abstract Traiangle{
～
～


      public abstract double area();//面積を求める。
}
```

なお、抽象メソッドが一つでもあるクラスは抽象クラスです。また、抽象メソッドの定義は、メソッドの定義としては不完全であるので、抽象クラス自身の定義もクラス定義としては完全なものではありません。したがって、抽象クラスはインスタンス化することができません。あくまでインスタンスは、抽象クラスを継承した個々のサブクラスに対して作成されます。抽象メソッド area() を追加した Triangle クラスをリスト 30 に示します。抽象メソッドの追加によって、Triangle クラスが抽象クラスになることに注意しましょう。

リスト 30:Triangle.java(Ver5.2)

```
1   /* 三角形抽象クラス*/
2   public abstract class Triangle{
3       /***********フィールド*****************/
4       private Point a;//点 A
5       private Point b;//点 B
6       private Point c;//点 C
7
8       /***********コンストラクタ*************/
9       /*
10        3 点から三角形を作る。点はコピーにする。
11        引数:3 点
12      */
13      Triangle(Point a,Point b, Point c){
14          this.a=new Point(a);
15          this.b=new Point(b);
16          this.c=new Point(c);
17      }
18
19      /*
20        6 つの座標から三角形を作る。
21        引数:3 つの座標を表す 6 座標
22      */
23      Triangle(double aX,double aY,double bX,double bY,double cX,double cY){
```

```
24          this.a=new Point(aX,aY);
25          this.b=new Point(bX,bY);
26          this.c=new Point(aX,bY);
27      }
28
29      /*
30        三角形のコピーを作る。
31        引数:三角形 (Triangle 型)
32      */
33      Triangle(Triangle t){
34          this.a=new Point(t.getA());
35          this.b=new Point(t.getB());
36          this.c=new Point(t.getC());
37      }
38
39      /*
40        デフォルトの三角形 (すべてが原点) 。
41       */
42      Triangle(){
43          this.a=new Point();
44          this.b=new Point();
45          this.c=new Point();
46      }
47
48      /**********メソッド**************/
49      //点 A
50      public Point getA(){
51          return a;
52      }
53
54      public void setA(Point a){
55          this.a=new Point(a);
56          return;
57      }
58
59      public void setA(double aX,double aY){
60          this.a=new Point(aX,aY);
61          return;
62      }
```

```
63
64      //点B
65      public Point getB(){
66          return b;
67      }
68
69      public void setB(Point b){
70          this.b=new Point(b);
71          return;
72      }
73
74      public void setB(double bX,double bY){
75          this.b=new Point(bX,bY);
76          return;
77      }
78
79      //点C
80      public Point getC(){
81          return c;
82      }
83
84      public void setC(Point c){
85          this.c=new Point(c);
86          return;
87      }
88
89      public void setc(double cX,double cY){
90          this.c=new Point(cX,cY);
91          return;
92      }
93
94      /*
95        自分を表示させるメソッド
96        戻り値：なし
97        引数：なし
98       */
99      public void print(){
100         a.print();
101         System.out.print("-");
```

```
102             b.print();
103             System.out.print("-");
104             c.print();//すべて改行なし
105         }
106
107         /*抽象メソッド*/
108         public abstract double area();
109     }
```

このソースコードはこのままコンパイルできます。しかし、この Triangle クラスは、抽象クラスであるため、インスタンス化できません。インスタンス化しようとするソースコードをコンパイルすると、次のようにエラーが表示されます。

リスト 31:TestTriangle.java(Ver5.2)

```
1   /*抽象三角形クラスを調べる間違ったプログラム*/
2   public class TestTriangle{
3       //main メソッド
4       public static void main(String[] argv){
5           //三角形 1
6           Triangle t1=new Triangle();
7           return;
8       }
9   }
```

```
$ javac TestTriangle.java
TestTriangle.java:6: Triangle は abstract です。インスタンスを生成することはできません。
        Triangle t1=new Triangle();
                 ^
エラー 1 個
$
```

5.3.2 抽象メソッドの実装

抽象メソッドは、継承とオーバーライドの仕組みを用いることで、抽象クラスのサブクラスにおいて完全に定義されます。このことを、**抽象メソッドの実装**といいます。先の抽象三角形クラスのメソッドでは area() が抽象クラスでしたので、Triangle クラスを継

承して area() をオーバーライドすることによってインスタンス化可能なクラスを定義できます。リスト 32 にそのようなクラスを示します。

リスト 32:ConcreteTriangle.java(Ver5.2)

```
1  /*インスタンス化可能な 3 角形クラス*/
2  public class ConcreteTriangle extends Triangle{
3      /*コンストラクタ*/
4      ConcreteTriangle(Point a,Point b, Point c){
5          super(a,b,c);
6      }
7
8      ConcreteTriangle
9          (double aX,double aY,double bX,double bY,double cX,double cY){
10         super(aX,aY,bX,bY,cX,cY);
11     }
12
13     ConcreteTriangle(ConcreteTriangle t){
14         super(t);
15     }
16
17     ConcreteTriangle(){
18         super();
19     }
20
21
22     /*abstract メソッドの area() をオーバーライド*/
23     public double area(){
24         Point a=getA();
25         Point b=getB();
26         Point c=getC();
27
28         double e1=a.distanceToOtherPoint(b);
29         double e2=b.distanceToOtherPoint(c);
30         double e3=c.distanceToOtherPoint(a);
31
32         double d=(e1+e2+e3)/2;
33
34         return Math.sqrt(d*(d-e1)*(d-e2)*(d-e3));
```

```
35      }
36  }
```

この具象三角形クラス (ConcreteTriangle) を利用するクラスを次に示します。

リスト 33:TestConcreteTriangle.java(Ver5.2)

```
1   /*抽象クラスを継承したプログラムの効果を調べるサンプルプログラム*/
2   public class TestConcreteTriangle{
3       //main メソッド
4       public static void main(String[] argv){
5           //三角形 1
6           ConcreteTriangle t1=new ConcreteTriangle();
7           System.out.print("t1=");
8           t1.print();
9           System.out.println();
10
11          //三角形 2
12          Point a=new Point();
13          Point b=new Point(3.0,0.0);
14          Point c=new Point(0.0,4.0);
15
16          ConcreteTriangle t2=new ConcreteTriangle(a,b,c);
17          System.out.print("t2=");
18          t2.print();
19          System.out.println();
20
21
22          //三角形 3
23          ConcreteTriangle t3=new ConcreteTriangle(t2);
24          System.out.print("t3=");
25          t3.print();
26          System.out.println();
27
28          System.out.println("t3 の面積は、"+t3.area()+"です。");
29          return;
30      }
31  }
```

5.4 インターフェース

オブジェクト指向における重要な概念として、**インターフェース**があります。インターフェースとは、その名の通り「モノ」と「外界」との間 (inter, インター) にある面 (face、フェース) のことです。この「モノ」と「外界」との境界、すなわち、「モノ」の見た目を定めるものが「インターフェース」です

クラスが複数あって互いに利用することを考えましょう。クラスの数が増えてくると、利用法自体が膨大になってしまい、その利用法自体の把握が困難になることがあります。そのような状況では、利用の仕組を統一することが重要になります。「インターフェース」とは、このような利用法を統一したものという見方もできます。

ここで、車と人間を例にとって、インターフェースの重要性を説明します。

今、メソッドとして「方向を変える。」「加速する。」「減速する。」を持つクラスを考えましょう。現実世界で、もしこの「方向を変える」というメソッドが、「ファミリーカー」、「スポーツカー」、「トラック」、… 等の個々のクラスで異なっている場合を考えましょう。もし、このような状況は「車」を利用する人間にとっては、個々のクラスごとに運転技術を修得しなければならないでしょう。このようなことは、ものすごく繁雑で不便なことだと気がつくでしょう。しかし、現実には、「方向を変える」には「ハンドル」を、「加速する」には「アクセル」を、「減速する」には「ブレーキ」を用いれば良いです。このように、個々の繁雑な利用法をまとめて中間的な利用法を作成することで、複雑さが軽減されます。このように、利用法を抽象メソッドとしてまとめたものを、**インターフェース**といいます。いったんインターフェースを定めてしまえば、そのインターフェースを実装する個々のクラスは、いくらでも取り変えられます。また同様に、そのインターフェースを利用する個々のクラスもいくらでも取り変えられます。

図 5.5 に、インターフェースの概念図を示します。

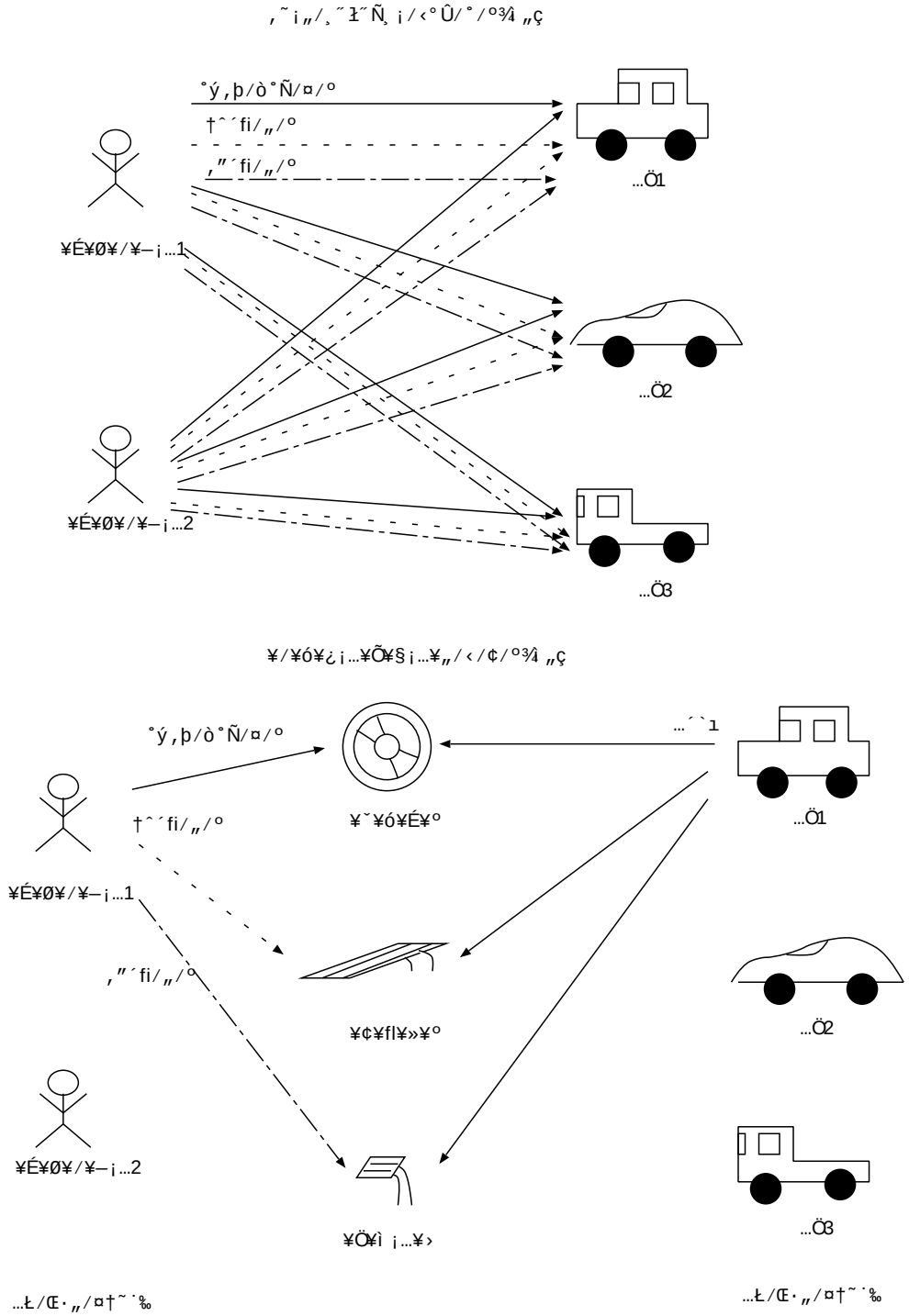


図 5.5: インターフェース

5.4.1 Javaによるインターフェース定義

Javaにおけるインターフェースは、抽象メソッドの集合として表現されます。すなわち、`class`の代わりに、`interface`を用いて次のような書式で定義します。Javaにおける、インターフェースは、クラスと同格に扱われることに注意して下さい。言い方を変えれば、Javaのプログラムはクラスとインターフェースから出来ています。なお、クラス定義において、スーパークラスは一つしか指定できませんでしたが、インターフェースは複数指定することができます。

```
インターフェース定義 ::=
  [アクセス修飾子] interface インターフェース名 [インターフェース継承]{
            [アクセス修飾子] 型 抽象メソッド名 (引数リスト);
                                .
                                .
                                .
}

インターフェース継承::= extends サブインターフェース名, …
```

リスト34に、インターフェースの定義例を示します。

リスト 34:`Movable.java`(Ver5.3)

```
1   /*インターフェース定義を示すサンプルプログラム*/
2   public interface Movable{
3       public void move(double dx,double dy);//(dx,dy)分移動させる。
4       /*abstractが無くても、抽象メソッドです。*/
5   }
```

このように、形式的には、抽象メソッドを並べることで、インターフェースを定義することができます。なお、インターフェース内で定義されるメソッドはすべて抽象メソッドであるので、`abstract`を付ける必要がありません。たとえ、`abstract`と書かれていなくても抽象メソッドとして扱われることに注意して下さい。

5.4.2 インターフェースの実装

ここでは、インターフェースで定義された抽象メソッドを実装する記述法について説明します。抽象クラスにおける抽象メソッドは、拡張(`extends`)とオーバーライドによって実

装されました。それに対して、インターフェース内の抽象メソッドは、**実装 (implements)** という仕組みによって、実装されます。下記に、BNF 記法を再掲します[10]。

継承指定 ::= [extends スーパークラス名] [implements インターフェース名, …]

したがって、インターフェースの実装は、クラス定義の際に implementes を用いて次のように行ないます。

```
public class サブクラス名 extends スーパークラス名 implements インターフェース名1, … {
        /*フィールド定義*/
        /*コンストラクタ定義*/
        /*メソッド定義*/
}
```

なお、インターフェースを実装するクラスでは、インターフェース内にあるすべての抽象メソッドに対して、メソッドの定義を与える必要があります。

リスト 35 に Movable インターフェースを実装する Point クラスを示します。

リスト 35:Point.java(Ver5.3)

```
1   /*Point クラス*/
2   public class Point implements Movable{
3       /**********************フィールド*****************************/
4       private double x; //x 座標
5       private double y; //y 座標
6
7       /**********************コンストラクタ**********************/
8
9       /*
10          2つの座標から点を作成する。
11          引数   : (x 座標 (double) ,y 座標 (double));
12      */
13      Point(double x,double  y){
14          this.x=x;
15          this.y=y;
16      }
17
18      /*
19          点のコピーを作成する。
20          引数   : 点 (Point)
```

[10] p.26 のクラス定義も参照のこと。

```
21      */
22      Point(Point pt){
23          x=pt.getX();
24          y=pt.getY();
25      }
26
27      /*
28         デフォルトの点の作成。
29         (引数がなければ原点のコピーを作成する。)
30         引数   : なし
31      */
32      Point(){
33          this(0.0,0.0);
34      }
35
36      /**********************メソッド*****************************/
37      /*
38         戻り値 : x 座標
39         引数   : なし
40      */
41      public double getX(){
42          return x;
43      }
44
45      /*
46         戻り値 : void
47         引数   : x 座標
48      */
49      public void setX(double x){
50          this.x=x;
51          return;
52      }
53
54      /*
55         戻り値 : y 座標
56         引数   : なし
57      */
58      public double getY(){
59          return y;
```

```
60      }
61
62      /*
63         戻り値：void
64         引数　：y座標
65      */
66      public void setY(double y){
67          this.y=y;
68          return;
69      }
70
71      /*
72         原点までの距離を求めるメソッド
73         戻り値：原点までの距離(double)
74         引数　：なし
75      */
76      public double distanceToOrigin(){
77          return Math.sqrt(x*x+y*y);
78      }
79
80      /*
81         他の点までの距離を求めるメソッド
82         戻り値：引数で指定された点までの距離(double)
83         引数　：他の点(Point型)
84      */
85      public double distanceToOtherPoint(Point opt){
86          double diffX=x-opt.getX();//x座標の差
87          double diffY=y-opt.getY();//y座標の差
88
89          return Math.sqrt(diffX*diffX+diffY*diffY);
90      }
91
92      /*
93         自分を表示するメソッド
94         戻り値：なし
95         引数　：なし
96      */
97      public void print(){
98          System.out.print("("+x+","+y+")");//改行なし
```

```
 99          return;
100      }
101
102      /*
103         点を (dx,dy) 分移動するメソッド
104         インターフェースの実装
105         戻り値 : なし
106         引数   : 移動分を表わす (double dx,double dy)
107      */
108      public void move(double dx,double dy){
109          x+=dx;
110          y+=dy;
111          return;
112      }
113  }
```

この Point クラスを利用するプログラムをリスト 36 に示します。

リスト 36:TestPoint.java(Ver5.3)

```
 1  /*
 2    インターフェースを実装した Point クラス
 3    を利用するサンプルプログラム
 4  */
 5  public class TestPoint{
 6      public static void main(String[] argv){
 7          //点
 8          Point pt=new Point(1.0,2.0);
 9          pt.print();
10          System.out.println();
11
12
13          double dx=1.0;
14          double dy=2.0;
15          System.out.println("move("+dx+","+dy+")");
16
17          pt.move(dx,dy);
18          pt.print();
```

```
19          System.out.println();
20
21          return;
22      }
23  }
```

拡張と実装を両方用いたクラス定義を、リスト 37 に示します。

リスト 37:ConcreteTriangle.java(Ver5.3)

```
1   /*インスタンス化可能な具象 3 角形クラス*/
2   public class ConcreteTriangle extends Triangle implements Movable{
3       /**********************コンストラクタ******************/
4       ConcreteTriangle(Point a,Point b, Point c){
5           super(a,b,c);
6       }
7
8       ConcreteTriangle
9           (double aX,double aY,double bX,double bY,double cX,double cY){
10          super(aX,aY,bX,bY,cX,cY);
11      }
12
13      ConcreteTriangle(ConcreteTriangle t){
14          super(t);
15      }
16
17      ConcreteTriangle(){
18          super();
19      }
20
21      /**********************メソッド******************/
22      /*
23        abstract メソッドの area() をオーバーライド
24        ヘロンの公式で面積を求める。
25        戻り値：面積
26        引数：なし
27      */
28      public double area(){
```

```
29          Point a=getA();
30          Point b=getB();
31          Point c=getC();
32
33          double e1=a.distanceToOtherPoint(b);//辺 AB の長さ
34          double e2=b.distanceToOtherPoint(c);//辺 BC の長さ
35          double e3=c.distanceToOtherPoint(a);//辺 CA の長さ
36
37          double d=(e1+e2+e3)/2;
38
39          return Math.sqrt(d*(d-e1)*(d-e2)*(d-e3));
40      }
41
42      /*
43        インターフェースの実装
44        説明：引数で指定された分、移動させる
45        戻り値:なし
46        引数:(x 座標の増分、y 座標の増分)
47      */
48      public void move(double dx,double dy){
49          Point a=getA();
50          Point b=getB();
51          Point c=getC();
52
53          a.move(dx,dy);
54          b.move(dx,dy);
55          c.move(dx,dy);
56
57          return;
58      }
59  }
```

また、リスト 37 を利用するプログラムをリスト 38 に示します。

リスト 38:TestConcreteTriangle.java(Ver5.3)

```
1   /*インターフェースの実装効果を調べるサンプルプログラム*/
2   public class TestConcreteTriangle{
```

```
3       //main メソッド
4       public static void main(String[] argv){
5           //三角形
6           Point a=new Point();
7           Point b=new Point(3.0,0.0);
8           Point c=new Point(0.0,4.0);
9
10          ConcreteTriangle t=new ConcreteTriangle(a,b,c);
11          System.out.print("t=");
12          t.print();
13          System.out.println();
14
15          double dx=1.0;
16          double dy=2.0;
17          System.out.println("move("+dx+","+dy+")");
18          t.move(dx,dy);
19          System.out.print("t=");
20          t.print();
21          System.out.println();
22
23          return;
24      }
25  }
```

インターフェースを定義すると、型も同時に定義されます。インターフェースの型を持つ参照変数には、インターフェースを実装しているクラスのインスタンスを保持することができます。このようなインターフェース型では、インターフェースにおける抽象メソッドがすべて利用可能なことに注意しましょう。

リスト 39 に、インターフェースの型を利用するプログラムを示します。

リスト 39:TestInterfaceType.java(Ver5.3)

```
1   /*インターフェースの型を調べるサンプルプログラム*/
2   public class TestInterfaceType{
3       public static void main(String[] argv){
4           //点
5           Point pt=new Point(1.0,2.0);
6           pt.print();
```

```
 7            System.out.println();
 8
 9            //三角形
10            Point a=new Point();
11            Point b=new Point(3.0,0.0);
12            Point c=new Point(0.0,4.0);
13
14            ConcreteTriangle t=new ConcreteTriangle(a,b,c);
15            System.out.print("t=");
16            t.print();
17            System.out.println();
18
19            //インターフェース型
20            Movable m1=pt;
21            Movable m2=t;
22
23            double dx=1.0;
24            double dy=2.0;
25            System.out.println("move("+dx+","+dy+")");
26            m1.move(dx,dy);
27            m2.move(dx,dy);
28
29            pt=(Point)m1;
30            t=(ConcreteTriangle)m2;
31            pt.print();
32            System.out.println();
33
34            t.print();
35            System.out.println();
36
37            return;
38        }
39    }
```

ここで、型変換について説明します。クラス名やインターフェス名のように、型を表す名前を括弧で囲むことによって、**キャスト演算子**になります。このキャスト演算子によって、型を変換することができます[11]。

[11]表 2.2 も参照して下さい

(型名) 参照型変数名

したがって、次のように代入することができます。必要に応じて型を変換するようにして下さい。

```
Point pt;
Movable m;
m=pt;
pt=(Point)m;
```

図 5.6 に、インターフェースの型の概念図を示します。

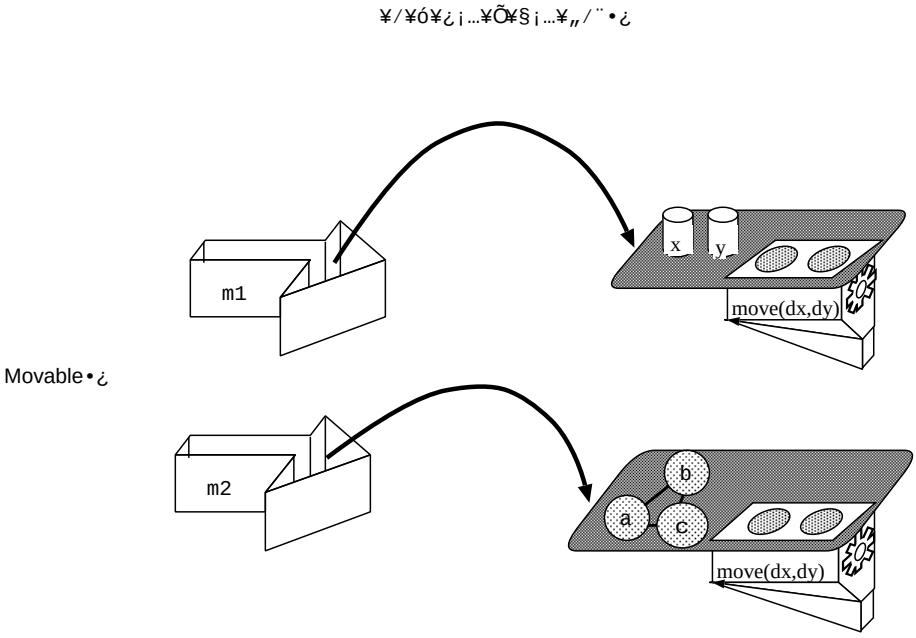


図 5.6: インターフェース型

第6章　例外処理

前章までの知識で、継承やインターフェース等を用いて、クラスを自由に組み合わせてプログラムが行なえるはずです。本章では、Javaでの例外処理について述べます。本章までで、Javaの基本的な文法を網羅することになります[1]。すなわち、本章までを理解すれば、API仕様等に記載してある説明の意味がわかるようになるはずです。

プログラムの実行中には、様々な要因によってエラー等の特殊な状況が発生します。そのような特殊な状況の要因として、例えば0で除算を行なおうとしたり、存在しないファイルを参照しようとしたりすることがあります。このような特殊な状況に対処するようにプログラムを作成することは、一般に通常の処理に対処するためだけのプログラムに比べて膨大な記述を必要とします。なぜならば、起り得る全ての可能性を考えて、それらの対処法を個々に記述する必要があるからです。しかも、これら特殊な状況への対処を記述すると、プログラムが必要以上に複雑になることが多いです。この結果、プログラム全体の動作にまで影響を及ぼすこともあります。

Javaには、このような特殊な状況に対する対処を、できるだけプログラムを複雑にしないで記述する仕組みがあります。Javaでは、特殊な状況についての対処の情報を表すオブジェクトを**例外 (exception)** と呼びます。この例外を利用することによって、エラー等の特殊な状況に対する対処を、比較的プログラムを複雑にせずに記述することができます。Javaでのこのような仕組みを**例外処理**と言うこともあります。

例外は、あるメソッドの中で**生成**され、**送出 (スロー、throw)** されます。一旦送出された例外は、そのメソッドを使っているメソッドにおいて**捕捉 (catch)** され、例外処理が実行されます。

図6.1に、例外の送出と捕捉の概念図を示します。

6.1　例外の定義

ここでは、例外の定義の仕方を述べます。

例外も、オブジェクトにすぎません。ある特別なクラス (`Exception`クラス) を継承することによって、例外が定義できます。したがって、例外は、通常のJavaの継承によって、次のように定義できます。

[1] 本資料の残りの部分では、Javaの豊富なクラスライブラリの中から、よく使われるものについて解説します。

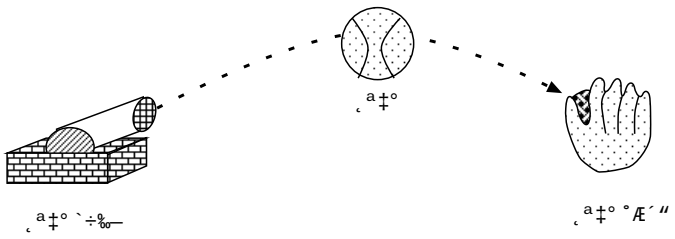


図 6.1: 例外の送出と捕捉

```
[アクセス指定] class 例外名 extends Exception{
        /*フィールド定義*/
        /*メソッド定義*/
}
```

リスト 40 に、直角指定の間違いを表わす例外を示します。この例外は、直角三角形クラス (RightTriangle クラス) で送出されることを想定しています。

リスト 40:WrongRightPointException.java(Ver6.1)

```
1   /*直角が無い事を表わす例外*/
2   public class WrongRightPointException extends Exception{
3       /***************フィールド***********/
4       private RightTriangle wrong;//直角の指定が間違っている直角三角形
5
6       /***************コンストラクタ***********/
7       /*
8         引数:
9         wrong:直角指定の間違っている三角形
10      */
11      public WrongRightPointException(RightTriangle wrong){
12          super();
13          this.wrong=wrong;
14      }
15
16      /**************メソッド***********/
17      /*
18        例外の内容を表示するメソッド
19        戻り値 : なし
20        引数 : なし
```

```
21        */
22        public void println(){
23            wrong.print();
24            System.out.println("における直角の指定が間違っています。");
25        }
26    }
```

6.2 例外の生成と送出

前節のように自分で定義した例外や Exception クラスの例外は、メソッドやコンストラクタの中で生成され、送出されます[2]。本節では、この記述法を説明します。

例外の生成は、通常のオブジェクトの生成と同じで、次のように記述すればいいです。

new 例外名（引数リスト）

```
new WrongRightPointException();
```

このように生成された例外を送出するには、throw を用いて次のように記述します。

throw 例外インスタンス

なお、ここでの例外インスタンスは、例外への参照で示します。

あるメソッド[3]の中で例外が送出されるときには、そのメソッド定義においても例外の送出を明示しなければなりません。すなわち、メソッドの引数リストの後ろに、次のような throws 節[4]を付加する必要があります。

例外送出::=throws 例外名, ⋯

したがって、例外を送出するメソッドの定義は、次のような書式になります。

[アクセス修飾子] メソッド名 (引数リスト) throws 例外名, ⋯ {
〜
}

```
public void setR(Point r) throws WrongRightPointException{

}
```

リスト 41 に、WrongRightPointException 例外を送出する直角三角形クラスを示します。

リスト 41:RightTriangle.java(Ver6.1)

[2]Exception クラスのスーパークラスに Throwable というクラスがあります。この Throwable クラスを継承しているクラスのオブジェクトは、送出の仕組みを利用できます。

[3]メソッドとコンストラクタ。例外の説明においては、メソッドだけで説明しますが、コンストラクタにおいても同様に記述することができます。

[4]先の throw とここでの throws は綴りが似ているので注意して下さい。

```
1  /*直角 3 角形クラス。例外を調べるためのサンプルプログラム*/
2  public class RightTriangle extends Triangle{
3      /**************フィールド***************/
4      Point r;//直角な点への参照
5
6      /**************コンストラクタ***********/
7      /*
8        指定がなければ点 A を直角とする。
9        引数:点 A が直角な直角三角形 (Point 型)
10     */
11     RightTriangle(Triangle t)throws WrongRightPointException{
12         super(t);//コピーの作成
13         r=this.getA();
14         if(!this.checkRightPoint()){
15             WrongRightPointException e=new WrongRightPointException(this);
16             throw e;
17         }
18     }
19
20     /*
21       指定された点が直角な直角三角形。
22       引数:直角な直角三角形 (Point 型)、直角な点
23     */
24     RightTriangle(Triangle t,Point r) throws WrongRightPointException{
25         super(t);//コピーの作成
26         if(r==t.getA()){
27             this.r=getA();
28         }else if (t.getB()==r){
29             this.r=getB();
30         }else if(t.getC()==r){
31             this.r=getC();
32         }else{
33             WrongRightPointException e=new WrongRightPointException(this);
34             throw e;
35         }
36
37         if(!this.checkRightPoint()){
38             WrongRightPointException e=new WrongRightPointException(this);
39             throw e;
```

```
40          }
41      }
42
43      /***********メソッド ***************************/
44      //点 R
45      public Point getR(){
46          return r;
47      }
48
49      /*
50        直角を指定するメソッド
51        戻り値：なし
52        引数：直角への参照
53      */
54      public void setR(Point r) throws WrongRightPointException{
55          if(r==getA()){
56              this.r=r;
57          }else if(r==getB()){
58              this.r=r;
59          }else if(r==getC()){
60              this.r=r;
61          }else{
62              WrongRightPointException e=new WrongRightPointException(this);
63              throw e;
64          }
65
66          if(this.checkRightPoint()){
67              WrongRightPointException e=new WrongRightPointException(this);
68              throw e;
69          }
70          return;
71      }
72
73      /*
74        面積を求めるメソッド
75        戻り値：面積
76        引数：なし
77      */
78      public double area(){
```

```
79          double base=0.0;//底辺
80          double height=0.0;//高さ
81
82          Point npt1;//直角でない点 1
83          Point npt2;//直角でない点 2
84
85          if(r==getA()){//点 A が直角
86              npt1=getB();
87              npt2=getC();
88          }else if (r==getB()){//点 B が直角
89              npt1=getC();
90              npt2=getA();
91          }else{//点 C が直角
92              npt1=getA();
93              npt2=getB();
94          }
95
96          base=r.distanceToOtherPoint(npt1);
97          height=r.distanceToOtherPoint(npt2);
98
99          return base*height/2.0;
100     }
101
102     /*
103       直角三角形かをチェックするメソッド
104       戻り値:直角指定の真偽値 (boolean)
105       引数 : なし
106     */
107     public boolean checkRightPoint(){
108         double base=0.0;//底辺
109         double height=0.0;//立辺
110         double slope=0.0;//斜辺
111         Point npt1;//直角でない点 1
112         Point npt2;//直角でない点 2
113
114         if(r==getA()){//点 A が直角
115             npt1=getB();
116             npt2=getC();
117         }else if (r==getB()){//点 B が直角
```

```
118                 npt1=getC();
119                 npt2=getA();
120             }else{//点 C が直角
121                 npt1=getA();
122                 npt2=getB();
123             }
124
125             base=r.distanceToOtherPoint(npt1);
126             height=r.distanceToOtherPoint(npt2);
127             slope=npt1.distanceToOtherPoint(npt2);
128
129             return slope*slope==(base*base+height*height);
130         }
131
132         /*
133           スーパークラスのメソッドをオーバーライドする。
134           自分の内容を表示するメソッド。
135           戻り値:なし
136           引数 : なし
137         */
138         public void print(){
139             super.print(); //super を削除してその効果を調てみて下さい。
140             System.out.print(":R("+r.getX()+","+r.getY()+")");
141             return;
142         }
143     }
```

6.3 例外の捕捉

前節のように例外を送出することが明示されているメソッドでは、単なるメソッド呼びしだけではそのメソッドを動作させることはできません。

リスト 42 に例外を送出するメソッドを間違って動作させるプログラムを示します。

リスト 42:TestNoTry.java(Ver6.1)

```
1   /* 例外を捕捉の効果を調べるための間違ったプログラム*/
2   public class TestNoTry{
```

```
3       public static void main(String[] argv){
4           Point a=new Point();
5           Point b=new Point(3.0,0.0) ;
6           Point c=new Point(0.0,4.0);
7
8           //直角 3 角形 1
9           ConcreteTriangle t=new ConcreteTriangle(a,b,c);
10          RightTriangle rt=new RightTriangle(t);
11          System.out.print("rt=");
12          rt.print();
13          System.out.println();
14
15          return;
16      }
17  }
```

このプログラムはコンパイルすることもできません。コンパイルしようとすると、次のようなエラーが表示されます。

```
$ javac TestNoTry.java
TestNoTry.java:10: 例外 WrongRightPointException は報告されません。スローするにはキャッチまたは、スロー宣言をしなければなりません。
        RightTriangle rt=new RightTriangle(t);
                              ^
エラー 1 個
$
```

このように、例外を送出するメソッドを動作させるには、次のような **try-catch** 構文を用いる必要があります。例外を送出するメソッドは、**try** 句の中に置いて記述します。もし例外が送出された場合には、例外の種類 (クラスの型) にしたがって、**catch** 句が実行されます。

```
try{
      ～
}catch(例外型 1 参照変数 1){
      ～
}catch(例外型 2 参照変数 2){
      ～
}
                                .
                                .
                                .
```

図 6.2 に、メソッドにおける例外の受け渡しの概念図を示します。

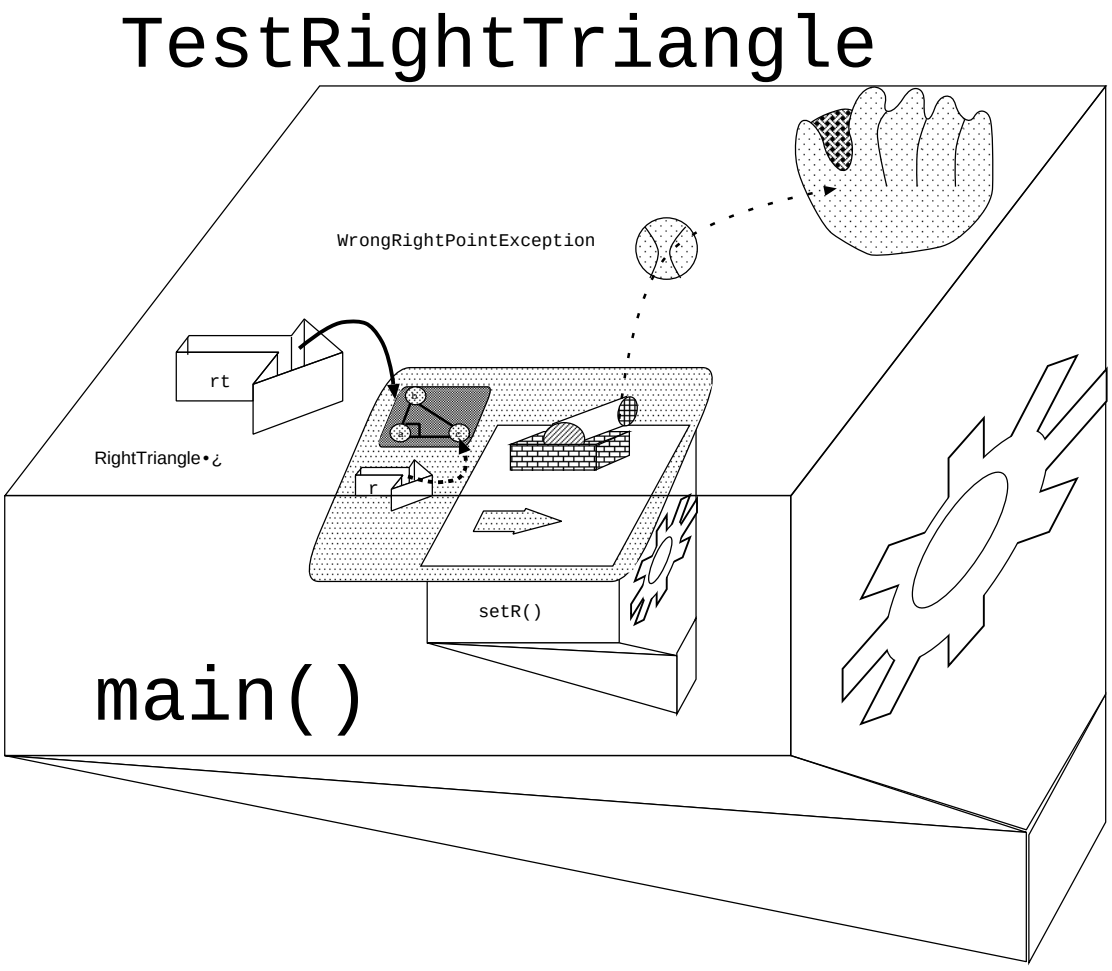


図 6.2: メソッド間の例外の受け渡し

リスト 42 に例外を送出するメソッドを正しく動作させるプログラムを示します。

リスト 43:RightTriangle.java(Ver6.1)

```
1  /*例外の捕捉を調べるサンプルプログラム*/
2  public class TestTry{
3      //main メソッド
4      public static void main(String[] argv){
5          try{
6              Point a=new Point();
7              Point b=new Point(3.0,0.0) ;
8              Point c=new Point(0.0,4.0);
9              //Point c=new Point(1.0,4.0);
10
11             //直角 3 角形 1
12             ConcreteTriangle t=new ConcreteTriangle(a,b,c);
13             RightTriangle rt1=new RightTriangle(t);
14             System.out.print("rt1=");
15             rt1.print();
16             System.out.println();
17
18             //直角 3 角形 2
19             a=new Point(1.0,2.0);
20             b=new Point(3.0,2.0) ;
21             c=new Point(3.0,-1.0);
22             t=new ConcreteTriangle(a,b,c);
23             Point r=t.getC(); //C は直角ではありません。
24             //Point r=t.getB();B は直角です。
25
26             RightTriangle rt2=new RightTriangle(t,r);
27             System.out.print("rt2=");
28             rt1.print();
29             System.out.println();
30
31         }catch(WrongRightPointException e){
32             e.println();
33         }
34
35         return;
```

```
36          }
37    }
38
```

このような例外の送出と捕捉は、1段階だけではありません。例外を送出するメソッドを用いる上位のメソッドは、例外を捕捉せずに、同じ例外をさらに上位に送出することができます。図6.3に、例外送出の連鎖の概念図を示します。

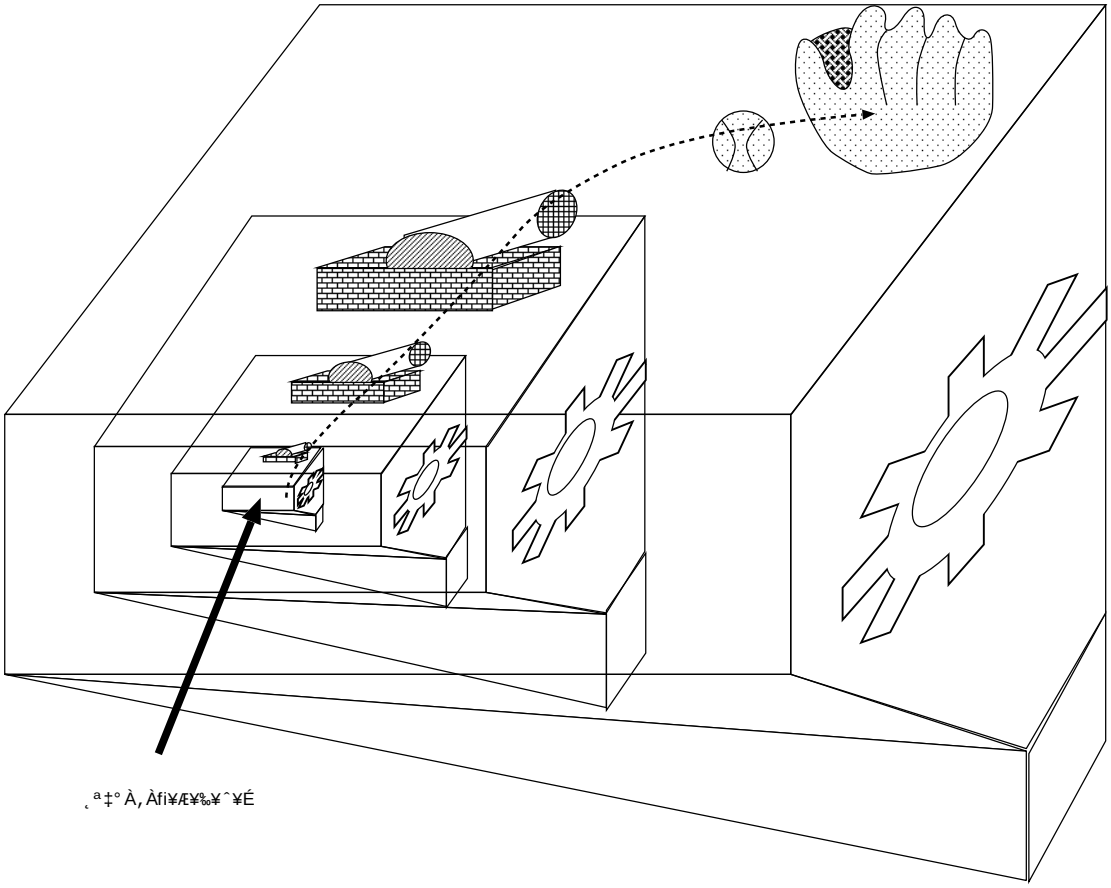


図 6.3: 例外送出の連鎖

第7章　Javaのデータ構造

前章までで、Javaの基本的な文法を網羅しました。これにより、API仕様も参照することができるようになったはずです。しかし、これまでは、各プログラムにおいて数個程度のオブジェクトしか取り扱ってきませんでした。コンピュータを用いる利点の一つに多量のデータを効率良く扱えることが有ります。そこで本章では、Javaで大量のデータを扱うための方法、すなわちJavaで用いられる主なデータ構造について説明します。

Javaには、多量のデータを扱うための仕組みがいろいろとあります。代表的な仕組みとして、配列とリスト構造を説明します。また、多量のデータを扱うためには、配列やリスト構造といった構造面だけでなく、データの利用法に注意を向ける必要もあります。利用法を統一的に扱うために抽象データ型について説明します[1]。

7.1　配列

C言語を含めた様々なプログラミング言語で、配列を扱うことができます。Javaにおいても、配列を扱うことができます。ここでは、Javaにおける配列の定義の仕方とその利用法について述べます。

7.1.1　基本型の配列

Javaで配列を準備するには、次のような書式を用います。

```
型名 [] 配列名；
配列名 = new 型名 [要素数]；
```

ここで、要素数には`int`の型を持つ式[2]を用いることができます。また、配列名は、配列を指す参照です。このように配列を指す参照を**配列参照**といいます[3]

例えば、次のようにして、`double`型の要素を5個持つ配列と、その配列を指す参照`d`が準備できます。

```
double[]  d;
d=new double[5];
```

また、次のように、配列参照と配列の準備を一緒にすることもできます。

---

[1] 抽象データ型は、インターフェースの一種と見なすこともできます。

[2] リテラルも式であることに注意して下さい。

[3] 配列参照は、通常のオブジェクトを指す参照とは若干異なるので、注意して下さい。配列参照の後には、`[]`の演算子を置くことができます。

型名 [] 配列参照= new 型名 [要素数] ;

```
double[]  d=new double[5];
```

図 7.1 に、double 型配列の概念図を示します。

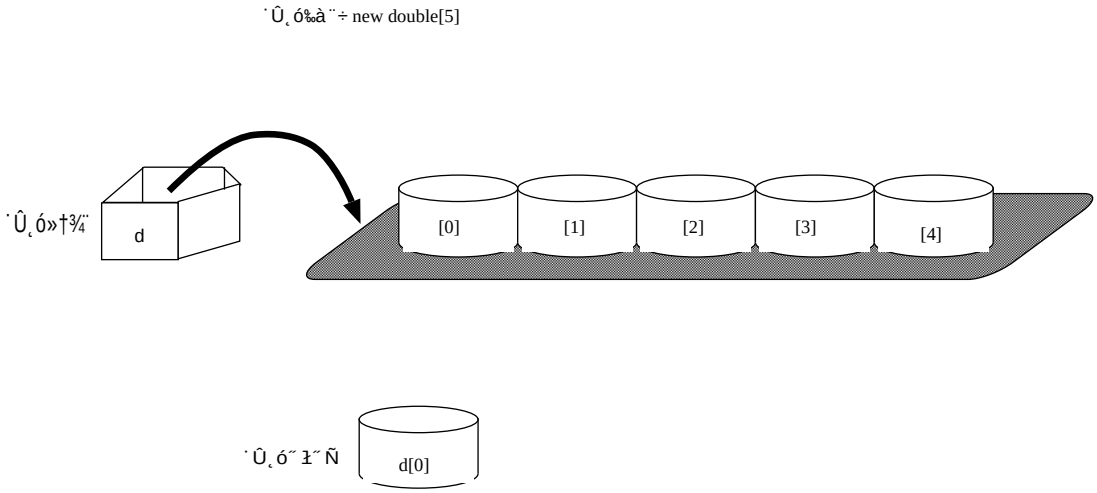


図 7.1: double 型の配列

配列を用いる場合には、

配列参照 [添字]

として、通常の変数のように扱うことができます。なお、添字部分には、int 型の式を用いることができます。

したがって、例えば、

```
d[3]
```

は、double 型の変数として扱うことができます。また、

配列参照.lengh ;

と記述すれば、その配列の要素数を表わします。例えば、

```
d.length
```

のように記述することで、配列の要素数を利用することができます[4]。

リスト 44 に、double 型の配列を用いたプログラムを示します。

リスト 44:TestDoubleArray.java(Ver7.1)

```
1   /*配列を調べるサンプルプログラム*/
2   public class TestDoubleArray{
3       /*フィールド*/
```

[4] for ループで配列を制御するときなどで用いると便利です。

```
4       private static int arrayNum=5;//配列の要素数
5
6       /*メソッド*/
7       public static void main(String[] argv){
8           /*配列参照の宣言*/
9           double[] d; //配列参照
10
11          /*配列の生成*/
12          d=new double[arrayNum];
13          //d[0]～d[arrayNum-1]までのdouble型変数を生成
14
15          /*配列要素数の表示*/
16          System.out.println("配列要素数は"+d.length+"です。");
17
18          /*値代入*/
19          for(int i=0;i<d.length;i++){
20              d[i]=(double)(i*i);
21          }
22
23          /*値表示*/
24          for(int i=0;i<d.length;i++){
25              System.out.println("d["+i+"]="+d[i]);
26          }
27
28          return;
29      }
30  }
```

配列名が参照であることを確かめるためのプログラムをリスト 45 に示します。また、その概念図を図 7.2 に示します。

リスト 45:TestArrayReference.java(Ver7.1)

```
1   /*配列参照を調べるサンプルプログラム*/
2   public class TestArrayReference{
3       /*フィールド*/
4       private static int arrayNum=5;//配列要素数
5
```

```
6       /*メソッド*/
7       public static void main(String[] argv){
8           /*配列準備*/
9           double[] d=new double[arrayNum];//オリジナル
10          double[] e=d;//コピー
11
12          /*配列要素数の表示*/
13          System.out.println("配列 d の要素数は"+d.length+"です。");
14          System.out.println("配列 e の要素数は"+e.length+"です。");
15
16          /*値代入*/
17          for(int i=0;i<d.length;i++){
18              d[i]=(double)(i*i);
19          }
20
21          /*値表示*/
22          for(int i=0;i<d.length;i++){
23              System.out.println("d["+i+"]="+d[i]);
24          }
25
26          for(int i=0;i<e.length;i++){
27              System.out.println("e["+i+"]="+e[i]);
28          }
29
30          /*値変更*/
31          System.out.println("e[] の値を変更します。");
32          for(int i=0;i<e.length;i++){
33              e[i]=(double)(i*i*i);
34          }
35
36          /*値表示*/
37          for(int i=0;i<d.length;i++){
38              System.out.println("d["+i+"]="+d[i]);
39          }
40
41          for(int i=0;i<e.length;i++){
42              System.out.println("e["+i+"]="+e[i]);
43          }
44
```

```
45            return;
46        }
47    }
```

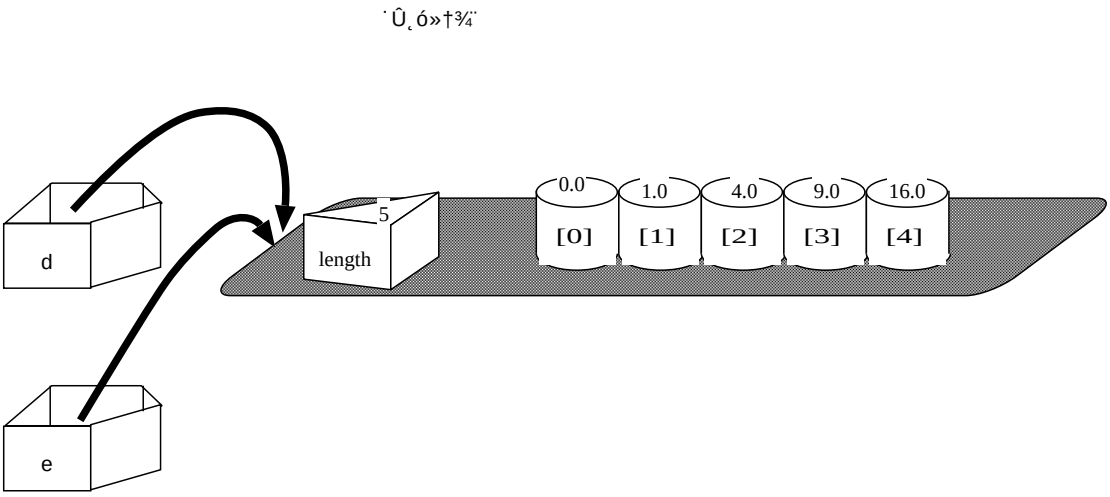


図 7.2: 配列参照

7.1.2 参照型の配列

前節では、基本型の要素を持つ配列について述べました。ここでは、参照型の要素を持つ配列について述べます。

参照型の配列であっても、基本型の配列と同様に準備あるいは利用できます。ただ、基本型には無かった注意点があります。すなわち、参照型の配列では、配列への参照とオブジェクトへの参照という 2 段階の参照を考える必要があるということです。

実際、参照型の配列において、その配列要素を **new** 演算によって生成すると、オブジェクトへの参照だけが用意されてオブジェクトが生成されません。したがって、実際のオブジェクトを利用するには、さらに **new** 演算によってインスタンスを生成する必要があります。このように、参照型の配列を用いる場合、**new** 演算を 2 段階以上適用することが多くなるかもしれません。図 7.3 に、その概念図を示します。

また、参照型の配列を通じて個々のオブジェクトを利用するには、図 7.3 にあるように、配列への参照からインスタンスへの参照へと 2 段階参照を辿る必要があります。すなわち、配列参照である配列名からインスタンスへのアクセスは、2 段以上の参照を通じて実現されます。

参照が多段になるとプログラムの状態が理解しにくくなりがちです。多段の参照を利用

するには、図 7.3 のようなしっかりとしたイメージを持ってプログラムすると良いでしょう[5]。

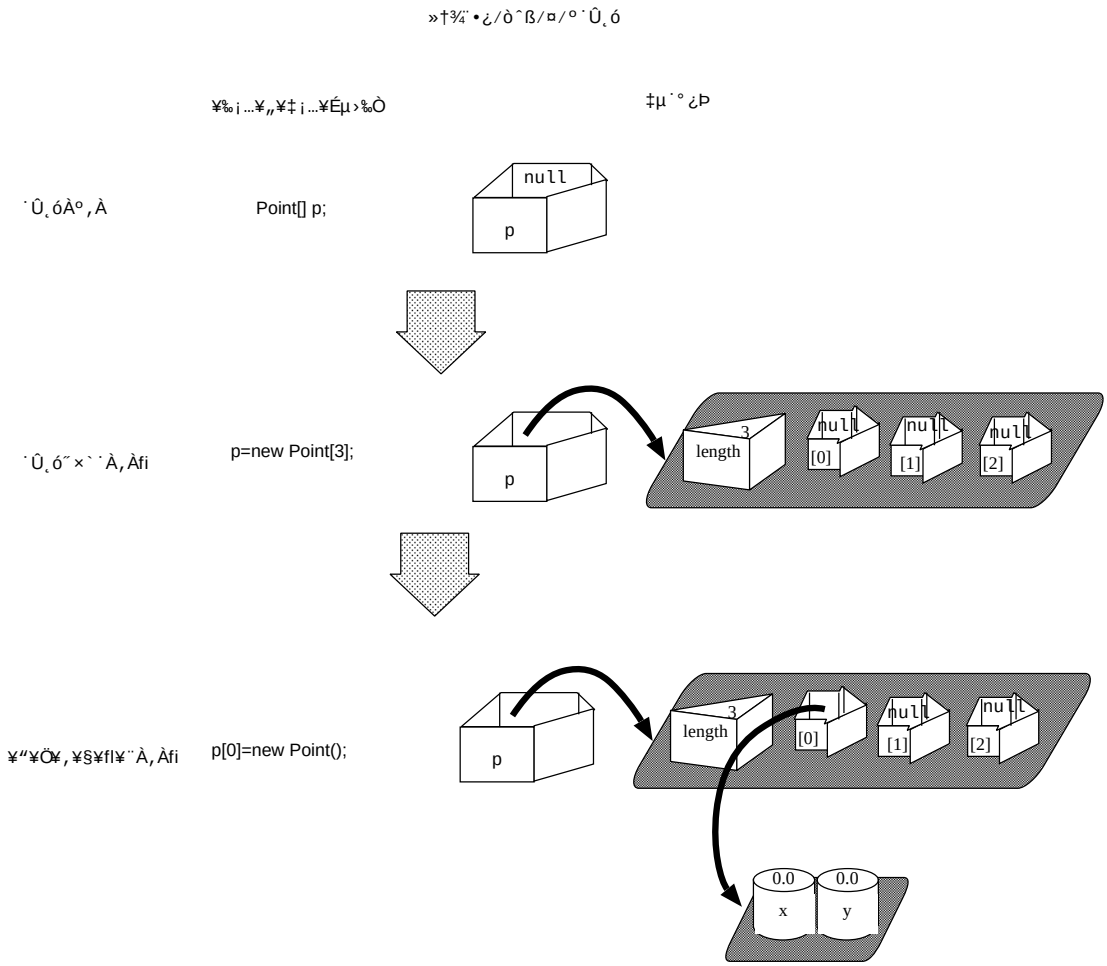


図 7.3: 参照型の配列

[5] これは配列参照に限らず、通常の参照にも言えることです。

参照型の配列を調べるために、Point クラスに toString() という自分自身を文字列に変換するメソッドを追加します。toString() は、特別なメソッド名であり、連結演算子「+」によって実行することができます。すなわち、

“”＋参照型変数名

と記述するのと、

参照型変数名.toString()

と記述するのでは、同じ効果を持ちます。この toString() は、どのようなクラスにも定義することができます。これまでに作ってきたクラスにも、toString() メソッドを追加すると良いでしょう。この toString() メソッドを追加することによって、

System.out.println(“”＋参照型変数名)

といった記述で統一的に文字列化したオブジェクトを表示することができます。なお、実は、Java では、Object というすべてのクラスの親クラスが存在します。toString() は、Object のメソッドであり、個々のサブクラス (Java で定義できるすべてのクラス) でオーバーライドされます。クラス定義において extends 節を記述しないと、この Object クラスを継承していると見なされます。すなわち、extends Object という継承指定だけが省略可能です[6]。

リスト 46 に、toString() を追加した Point クラスを示します。

リスト 46:Point.java(Ver7.1)

```
1   /*Point クラス*/
2   public class Point implements Movable{
3       /**********************フィールド*****************************/
4       private double x; //x 座標
5       private double y; //y 座標
6
7       /**********************コンストラクタ**********************/
8
9       /*
10          2 つの座標から点を作成する。
11          引数   : (x 座標 (double) ,y 座標 (double));
12      */
13      Point(double x,double  y){
14          this.x=x;
15          this.y=y;
16      }
17
```

[6] Object クラスの他のメンバを API 仕様等で調べる良いでしょう。

```
18      /*
19         点のコピーを作成する。
20         引数   : 点 (Point)
21      */
22      Point(Point pt){
23          x=pt.getX();
24          y=pt.getY();
25      }
26
27      /*
28         デフォルトの点の作成。
29         (引数がなければ原点のコピーを作成する。)
30         引数   : なし
31      */
32      Point(){
33          this(0.0,0.0);
34      }
35
36      /**********メソッド**********/
37      /*
38         戻り値 : x 座標
39         引数   : なし
40      */
41      public double getX(){
42          return x;
43      }
44
45      /*
46         戻り値 : void
47         引数   : x 座標
48      */
49      public void setX(double x){
50          this.x=x;
51          return;
52      }
53
54      /*
55         戻り値 : y 座標
56         引数   : なし
```

```
57      */
58      public double getY(){
59          return y;
60      }
61
62      /*
63         戻り値:void
64         引数   :y座標
65      */
66      public void setY(double y){
67          this.y=y;
68          return;
69      }
70
71      /*
72         原点までの距離を求めるメソッド
73         戻り値:原点までの距離(double)
74         引数   :なし
75      */
76      public double distanceToOrigin(){
77          return Math.sqrt(x*x+y*y);
78      }
79
80      /*
81         他の点までの距離を求めるメソッド
82         戻り値:引数で指定された点までの距離(double)
83         引数   :他の点(Point型)
84      */
85      public double distanceToOtherPoint(Point opt){
86          double diffX=x-opt.getX();//x座標の差
87          double diffY=y-opt.getY();//y座標の差
88
89          return Math.sqrt(diffX*diffX+diffY*diffY);
90      }
91
92      /*
93         自分を表示するメソッド
94         戻り値:なし
95         引数   :なし
```

```
96        */
97        public void print(){
98            System.out.print(this.toString());//改行なし
99            return;
100       }
101
102       /*
103           自分を文字列に変換するメソッド
104           戻り値：自分自身を表わす文字列
105           引数　：なし
106       */
107       public String toString(){
108           return "("+x+","+y+")";
109       }
110
111
112       /*
113           点を (dx,dy) 分移動するメソッド
114           インターフェースの実装
115           戻り値：なし
116           引数　：移動分を表わす (double dx,double dy)
117       */
118       public void move(double dx,double dy){
119           x+=dx;
120           y+=dy;
121           return;
122       }
123   }
```

この Point クラスを利用して、参照型の配列を調べるプログラムをリスト 47 に示します。

リスト 47:TestReferenceArray.java(Ver7.1)

```
1   /*参照の配列を調べるサンプルプログラム*/
2   public class TestReferenceArray{
3       /*フィールド*/
4       private static int arrayNum=5;//配列の要素数
```

```
 5
 6       /*メソッド*/
 7       public static void main(String[] argv){
 8           /*配列参照宣言*/
 9           System.out.println("配列参照宣言");
10           Point[] p;
11           p=null;
12           System.out.println("p="+p);
13
14           /*配列(各要素は参照)の生成*/
15           System.out.println("配列生成");
16           p=new Point[arrayNum];
17           //p[0]～p[arrayNum-1]までの参照を生成
18           //これでは参照しか生成しないことに注意
19           System.out.println("配列要素数は"+p.length+"です。");
20           System.out.println("p="+p);
21           for(int i=0;i<p.length;i++){
22               System.out.println("p["+i+"]="+p[i]);
23           }
24
25           /*インスタンス生成*/
26           System.out.println("配列要素のオブジェクト生成");
27           for(int i=0;i<p.length;i++){
28               p[i]=new Point((double) i,(double) i*i);
29           }
30           System.out.println("p="+p);
31           for(int i=0;i<p.length;i++){
32               System.out.println("p["+i+"]="+p[i]);
33           }
34
35           return;
36       }
37   }
```

7.1.3　多次元配列

Java においても、多次元配列を利用することができます。Java での多次元配列は、配列の配列として取り扱われます。次の書式で、多次元配列を用意することができます。

型 [][] 多次元配列名＝new 型名 [次元 1 要素数][];

```
double[][]  a=new double[5][];
```

ここでは、単に配列参照を要素数分準備しているだけに注意して下さい。多次元配列の個々の要素は、さらに new 演算子を用いて生成する必要があります。個々の要素までも一括して生成するためには、次のような書式を用いることもできます。

型 [][] 多次元配列名＝new 型名 [次元 1 要素数][次元 2 要素数];

```
double[][]  b=new double[3][4];
```

この書式によって、多次元配列が個々の要素まで生成されます。

上のようにして生成された多次元配列の各要素は、C 言語と同様に、次のような記述で用いることができます。

配列参照 [添字 1][添字 2]

したがって、例えば、a[1][2] や b[1][2] は、通常の double 型の変数として扱うことができ、次のようにソースコード内に記述できます。

```
a[1][2]=b[1][2];
```

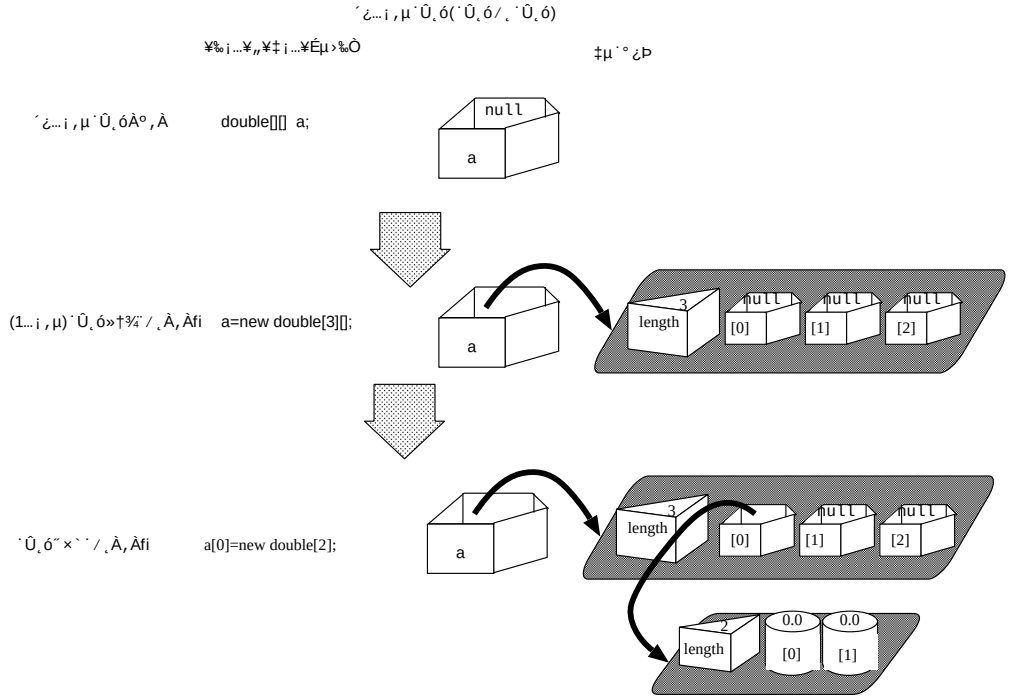
リスト 48 に、多次元配列を利用するプログラムを示します。

リスト 48:TestMultiDimensionArray.java(Ver7.1)

```
1   /*多次元配列を調べるサンプルプログラム*/
2   public class TestMultiDimensionArray{
3       /*フィールド*/
4       private static int dim1=3;//次元 1 の要素数
5       private static int dim2=4;//次元 2 の要素数
6
7       /*メソッド*/
8       public static void main(String[] argv){
9           /*多次元配列の宣言*/
10          double[][]  a=new double[dim1][];
11          //最初の次元の要素数は必ず指定
12          //a[0]～a[dim1-1] までの double[] 型変数を生成
13
14          /*配列要素の生成*/
15          for(int i=0;i<a.length;i++){
16              a[i]=new double[dim2];
```

```
17          }
18
19          /*ここまでの一連の記述は、次のようにも記述可能*/
20          //double[][] a=new double[dim1][dim2];
21
22          /*配列要素数の表示*/
23          for(int i=0;i<a.length;i++){
24              System.out.println("a[i].length="+a[i].length);
25          }
26
27          /*値代入*/
28          for(int i=0;i<a.length;i++){
29              for(int j=0;j<a[i].length;j++){
30                  a[i][j]=(double)i+(double)j/10.0;
31              }
32          }
33
34          /*値表示*/
35          System.out.println("a[i][j] の中身を表示");
36          System.out.print("j=");
37          for(int j=0;j<dim2;j++){
38              System.out.print("    "+j);
39          }
40          System.out.println();
41
42          for(int i=0;i<a.length;i++){
43              System.out.print("i="+i);
44              for(int j=0;j<a[i].length;j++){
45                  System.out.print("  "+a[i][j]);
46              }
47              System.out.println();
48          }
49
50          return;
51      }
52  }
53
```

図 7.4 に、多次元配列の概念図を示します。



´¿…ì ,µ˙Û˛ó/˛/“/-/º˙Û˛ó˝×``/˛°ì ‡çÀ,Àfi

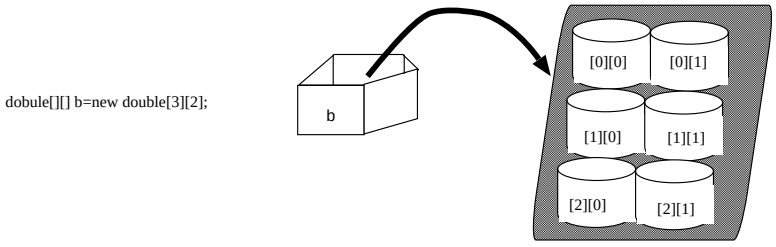


図 7.4: 多次元配列 (配列の配列)

7.2 リスト構造

本節では、多量のデータを扱う際に、配列と同じくらい良く用いられる**リスト構造**について説明します。Javaでは、リスト構造は、配列のように特別な仕組みが用意されている訳ではありません[7]。リスト構造は、クラスと参照を利用してプログラマが構築する**データ構造**の一つです。

通常、データ構造は、ある意味単位でまとめられたデータを一つの**セル**内に保持するようにし、それらのセルに相互参照する仕組みを付加することで設計されます。これらの相互参照の構造が一直線状[8]になるものを**リスト構造**あるいは**連結リスト**といいます。なお、リスト構造は、Javaだけでなく、C言語においても頻繁に用いられる考え方です[9]。

まず、リスト構造を構成するために、セルを定義します。リスト49にプログラムを示します。

リスト 49:ListCell.java(Ver7.2)

```
1  /*連結リストの個々の要素 (セル) を表わすクラス*/
2  public class ListCell{
3      /**************フィールド***************/
4      private double item;//セルに格納されるアイテム (double 値)
5      private ListCell next;//次のセルへの参照
6
7      /**************コンストラクタ************/
8      /*
9        セルを生成するコンストラクタ
10       引数:
11       item:格納する double 値
12       next:次のセルへの参照
13      */
14     public ListCell(double item,ListCell next){
15         this.item=item;
16         this.next=next;
17     }
18
19     /*
20       引数:
21       item:格納する double 値
```

[7]言語仕様 (文法) として用意はされていませんが、後述するようにライブラリとしては用意されています。
[8]直線状以外にも、様々な構造を考えることができます。よく利用されるものとして木構造があります。
[9]C 言語においては、通常、連結リストは構造体とポインタで実現されます。文献 [4] を参照して下さい。

```
22          next 参照は null を指定
23        */
24        public ListCell(double item){
25            this(item,null);
26        }
27
28        /*
29          デフォルトコンストラクタ
30          item は 0.0
31          next は null
32          引数:なし
33        */
34        public ListCell(){
35            this(0.0);
36        }
37
38        /**************メソッド*****************/
39        public double getItem(){
40            return item;
41        }
42
43        public void setItem(double item){
44            this.item=item;
45            return;
46        }
47
48        public ListCell getNext(){
49            return next;
50        }
51
52        public void setNext(ListCell next){
53            this.next=next;
54            return;
55        }
56    }
```

図 7.5 に、連結リストを構成するための、ListCell 型の概念図を示します。
このセルを直線上に連結したものが連結リストです。連結リストでは、セルを挿入や削

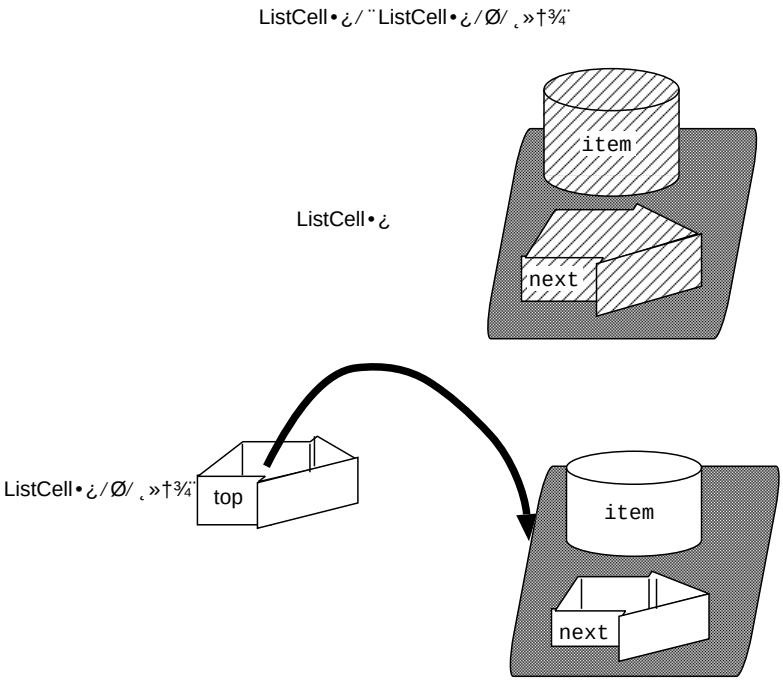


図 7.5: `ListCell` 型

除しても構造を直線状に保たなければなりません。そこで、セルの挿入や削除を管理して連結リストの状態を保つクラスが必要になってきます。ここでは、連結リストの先頭だけから、セルを挿入あるいは削除を行なうことにしましょう。連結リストの先頭だけから挿入削除を行なうデータ構造は、**スタック**[10]という特別な名称で呼ばれることもあります。スタックにデータを挿入することを**プッシュ (push)** と言ったり、スタックからデータを取り出すことを**ポップ (pop)** と言ったりします。

連結リストの構造を基にしたスタックを、リスト 50 にプログラムを示します。

リスト 50:`ListStack.java`(Ver7.2)

```
1   /*連結リストを用いたスタック*/
2   public class ListStack{
3       /**********フィールド************/
4       ListCell  top;//連結リストの先頭 (スタックのトップ)
5
6       /*************コンストラクタ*******/
7       ListStack(){
```

[10] スタックは、次節の抽象データ型でより詳しく説明します

```
8           top=null;
9       }
10
11      /***************メソッド*********/
12      /*
13        スタックのトップを返す。
14        引数：なし
15        戻り値：スタックのトップ
16       */
17      public ListCell getTop(){
18          return top;
19      }
20
21      /*
22        スタックのトップを設定する。
23        引数：セルへの参照
24        戻り値：なし
25       */
26      public void setTop(ListCell top){
27          this.top=top;
28          return;
29      }
30
31      /*
32        スタックのトップを設定する。
33        引数：他のスタックへの参照
34        戻り値：なし
35       */
36      public void setTop(ListStack other){
37          this.top=other.getTop();
38          return;
39      }
40
41      /*
42       スタックにアイテムを挿入する (プッシュ)；
43       引数：アイテム (double 値)
44       戻り値：なし
45      */
46      public void push(double item){
```

```
47          ListCell old=this.top;//更新前の先頭
48          ListCell newCell = new ListCell(item,old);//新しいセルの生成
49          this.top=newCell;//スタックの先頭の更新
50          return;
51      }
52
53      /*
54       スタックからアイテムを取り出す(ポップ);
55       引数:なし
56       戻り値:スタックトップのアイテム(double値)
57      */
58      public double pop(){
59          /*
60            スタックにアイテムが無い場合
61            (例外で対処すべきだが、とりあえずコメントだけを行なう。)
62          */
63          if(top==null){
64              System.out.println("スタックにアイテムが無いのにpopしてます。");
65              System.out.println("暫定的に0.0を返します。");
66              return 0.0;
67          }
68
69          /*スタックにアイテムがある場合*/
70          ListCell old=this.top;//更新前の先頭
71          double item=old.getItem();//アイテムの取得
72          this.top=old.getNext();//後処理
73          return item;
74      }
75  }
```

ListStackクラスを利用するプログラムをリスト51に示します。

リスト 51:TestListStack.java(Ver7.2)

```
1   /*ListStackクラスを調べるサンプルプログラム*/
2   public class TestListStack{
3       public static void main(String[] argv){
4           /*スタック準備*/
```

```
5           ListStack s=new ListStack();
6
7           System.out.println("pop():"+s.pop());
8
9           double a=7.0;
10          double b=2.0;
11          double c=8.0;
12          System.out.println("push("+a+")");
13          s.push(a);
14
15          System.out.println("push("+b+")");
16          s.push(b);
17
18          System.out.println("push("+c+")");
19          s.push(c);
20
21          System.out.println("pop():"+s.pop());
22          System.out.println("pop():"+s.pop());
23          System.out.println("pop():"+s.pop());
24          return;
25      }
26  }
```

図 7.6 に、連結リストで実現したスタックの概念図を示します。

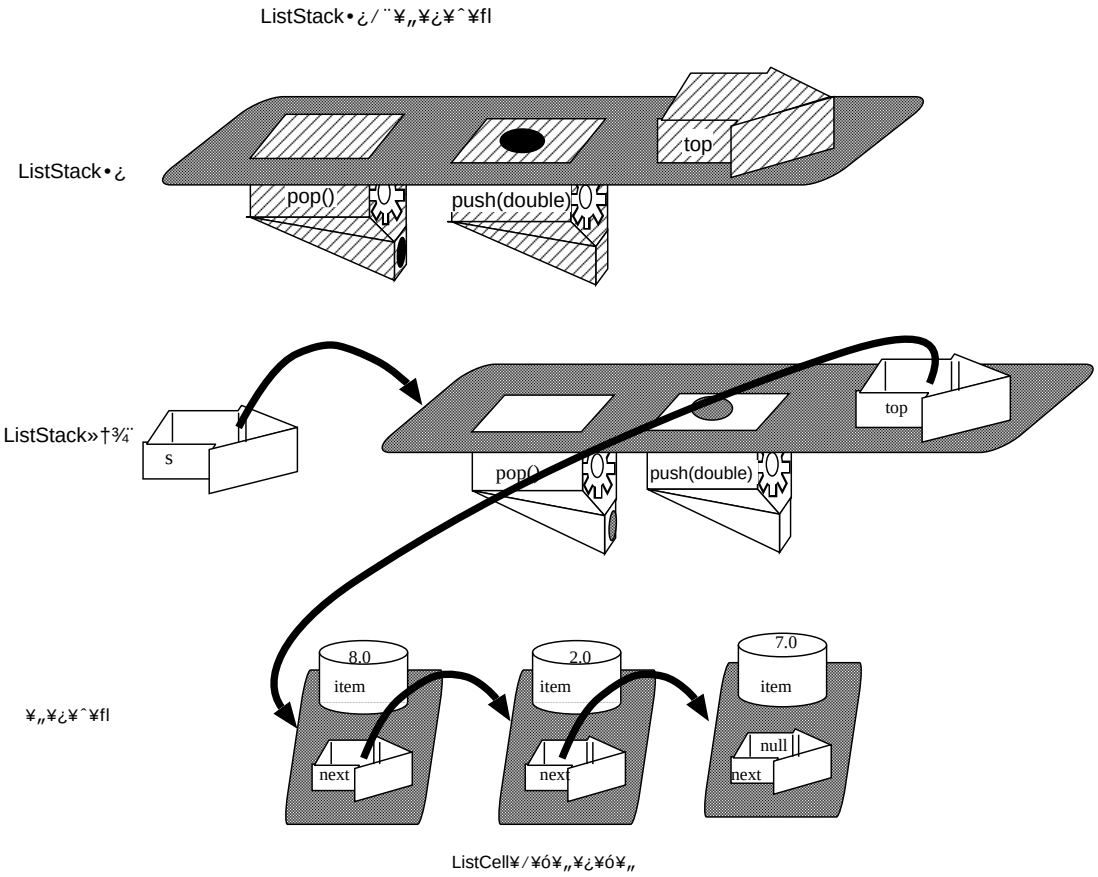


図 7.6: 連結リストによるスタック

7.3　抽象データ型

これまでに、多量のデータを扱うための仕組みを 2 種類見てきました。すなわち、節 7.1 では配列を、節 7.2 ではリスト構造を学んできました。このような、配列やリスト構造は、データを保持するための “構造” を定めるものです。しかし、データを利用する立場の「モノ」(オブジェト) においては、データを保持する構造よりもデータの利用法だけが重要なことも多いです。ですから、データ構造を作成する際に、配列やリスト構造などの詳細な構造面はデータ構造の内部で定義することにして、その利用法と効果だけをまとておくと便利です。このように、利用法と効果だけをまとめたデータ構造を、**抽象データ型 (Abstract Data Type,ADT)** といいます。この抽象データ型によって、データの利用を統一的に管理することができるようになります。

例えば、前節におけるスタックは、よく知られている抽象データ型です。スタックの利用法としては、データの挿入を行なう `push()` およびデータの取り出しを行なう `pop()` があります。また、スタックは、**後入れ先出し (Last In First Out,LIFO)** の効果を持つようなデータ構造です[11]。図 7.8 に、スタックの概念図を示します。

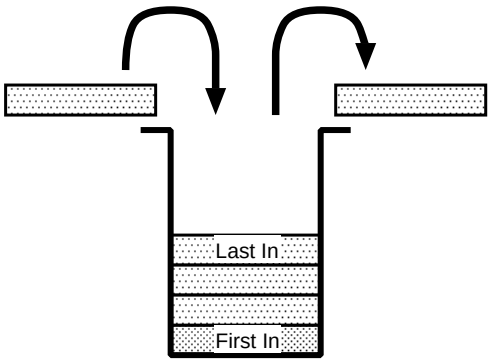


図 7.7: スタック

図 7.8 に、抽象データ型とその実装に関する概念図を示します。

[11]抽象データ型と言った場合、この効果までも含めることが多いです。しかし、Java ではこの効果までを表現できる仕様には現在のところなっていません

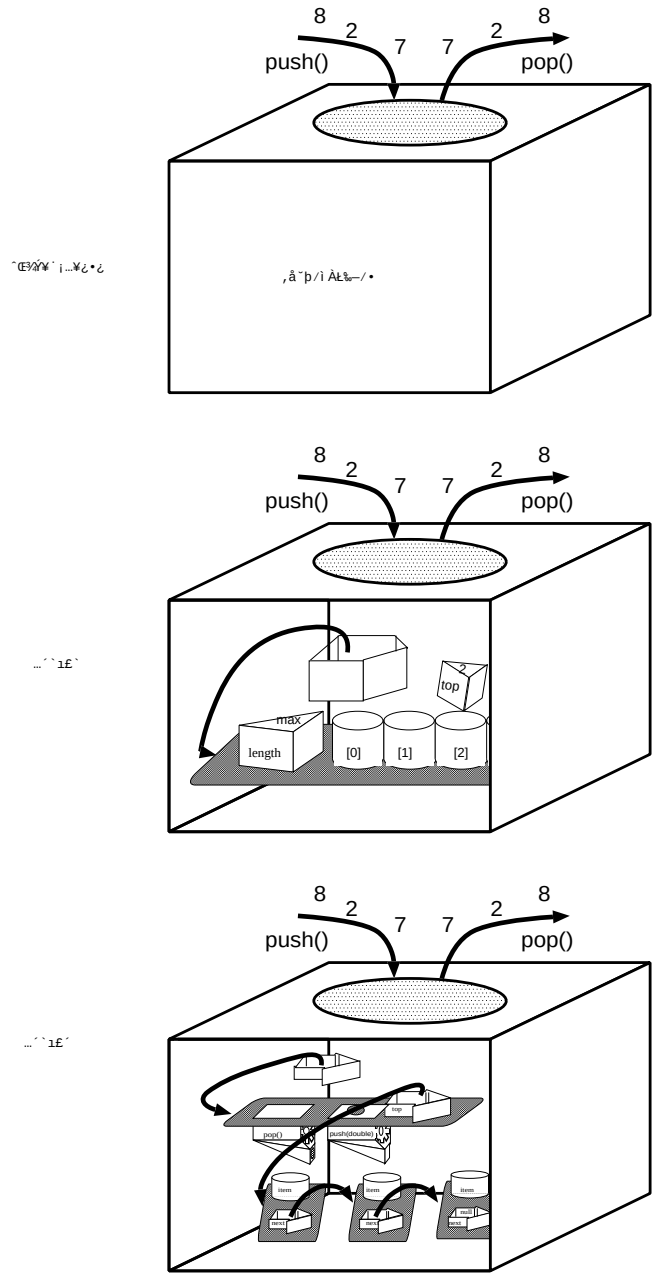


図 7.8: 抽象データ型とその実装 (スタック)

Java では、インターフェースを用いれば、抽象データ型を表現することができます。例えば、Java で抽象データ型のスタックを作成するには、抽象メソッドの `push()` と `pop()` をメンバとするようなインターフェースを定義すれば良いことがわかります。

この抽象データ型のスタック (`stack` インターフェース) をリスト 52 に示します。

リスト 52:`Stack.java`(Ver7.3)

```
1  /*
2    抽象データ型としてのスタックを表すインターフェース
3   */
4  public interface Stack{
5      /*
6        データの挿入
7        引数 : アイテム (double 値)
8        戻り値 : なし
9        送出例外:オーバーフロー
10     */
11     public void push(double item) throws StackOverflowException;
12
13     /*
14       データの取り出し
15       引数 : なし
16       戻り値 : アイテム (double 値)
17       送出例外:アンダーフロー
18     */
19     public double pop() throws StackUnderflowException;
20 }
```

この `Stack` インターフェースに対して、2 種類の実装を与えます。一つは連結リストを用いた実装で、もう一つは配列を用いた実装です[12]。また、スタックを扱うための例外も 3 種類与えます。

まず、3 種類の例外の定義から示します。スタックに関する例外を統一的に扱うための例外として `StackException` を、スタックが一杯のときの例外として `StackOverflowException` を、スタックが空のときの例外として `StackUnderflowException` を、それぞれリスト 53 〜リスト 55 に示します。

リスト 53:`StackException.java`(Ver7.3)

[12] 図 7.8 を参照して下さい。

```
1  /*スタックに関する例外*/
2  public class StackException extends Exception{
3      /*****フィールド********/
4      /*****コンストラクタ****/
5      /*****メソッド****/
6      public void println(){
7          System.out.println("スタックに関する例外が発生しました。");
8          return;
9      }
10
11     public String toString(){
12         return "スタックに関する例外";
13     }
14 }
```

リスト 54:StackOverflowException.java(Ver7.3)

```
1  /*スタックオーバーフロー*/
2  public class StackOverflowException extends StackException{
3      /*****フィールド********/
4      double triedValue;//蓄えようとした値。
5
6      /*****コンストラクタ****/
7      StackOverflowException(double triedValue){
8          super();
9          this.triedValue=triedValue;
10     }
11
12     StackOverflowException(){
13         this(0.0);//何も指定しなければ、0.0を挿入しようしたとみなす。
14     }
15
16     /*****メソッド****/
17     public void println(){
18         super.println();
19         System.out.println("スタックオーバーフロー："
20                            +triedValue
```

```
21                              +"は保存できません。");
22          return;
23      }
24
25      public String toString(){
26          return "スタックオーバーフロー";
27      }
28  }
```

リスト 55:StackUnderflowException.java(Ver7.3)

```
1   /*スタックアンダーフロー*/
2   public class StackUnderflowException extends StackException{
3       /*****フィールド********/
4       /*****コンストラクタ****/
5       StackUnderflowException(){
6           super();
7       }
8
9       /*****メソッド****/
10      public void println(){
11          super.println();
12          System.out.println("スタックアンダーフロー：スタックは空です。");
13          return;
14      }
15
16      public String toString(){
17          return "スタックアンダーフロー";
18      }
19  }
```

次に、連結リストを用いたスタックの実装をリストに示します。

リスト 56:ListStack.java(Ver7.3)

```
1   /*連結リストを用いたスタックの実装*/
```

```
2   public class ListStack implements Stack{
3       /**********フィールド***********/
4       ListCell  top;//連結リストの先頭(スタックのトップ)
5       int num; //現在スタックに保持しているセル数
6       int max; //スタックに保持できる最大のセル数(要素数)
7
8       /*************コンストラクタ*******/
9       ListStack(int max){
10          top=null;
11          num=0;
12          this.max=max;
13      }
14
15      ListStack(){
16          this(10);//デフォルトでは10セルとする。
17      }
18
19      /***************メソッド*********/
20      /*
21       現在保持しているデータ数を返す。
22        引数:なし
23        戻り値:アイテム数(int)
24       */
25      public int getNum(){
26          return num;
27      }
28
29      /*
30        スタックのトップを返す。
31        引数:なし
32        戻り値:スタックのトップ
33       */
34      public ListCell getTop(){
35          return top;
36      }
37
38      /*
39        スタックのトップを設定する。
40        引数:セルへの参照
```

```
41          戻り値：なし
42         */
43        public void setTop(ListCell top){
44            this.top=top;
45            return;
46        }
47
48        /*
49         スタックにアイテムを挿入する（プッシュ）；
50         引数：アイテム（double 値）
51         戻り値：なし
52        */
53        public void push(double item) throws StackOverflowException{
54            /*スタックオーバーフローが生じる場合*/
55            if(num>=max){
56                throw new StackOverflowException(item);
57            }
58
59            /*スタックにデータ保存可能な場合*/
60            ListCell old=this.top;//更新前の先頭
61            ListCell newCell = new ListCell(item,old);//新しいセルの生成
62            this.top=newCell;//スタックの先頭の更新
63            num++;//保持している要素数の更新
64
65            return;
66        }
67
68        /*
69         スタックからアイテムを取り出す（ポップ）；
70         引数：なし
71         戻り値：スタックトップのアイテム（double 値）
72        */
73        public double pop() throws StackUnderflowException{
74            /*
75              スタックにアイテムが無い場合
76            */
77            if(num<=0){
78                throw new StackUnderflowException();
79            }
```

```
80
81          /*スタックにアイテムがある場合*/
82          ListCell old=this.top;//更新前の先頭
83          double item=old.getItem();//アイテムの取得
84          this.top=old.getNext();//スタックの先頭の更新
85          num--;//保持している要素数の更新
86
87          return item;
88      }
89  }
```

次に、配列を用いたスタックの実装をリストに示します。

リスト 57:ArrayStack.java(Ver7.3)

```
1   /*
2     連結リストを用いたスタックの実装
3     スタックの底が配列の先頭であることに注意する。
4   */
5   public class ArrayStack implements Stack{
6       /**********フィールド************/
7       double[] items;//スタックを表わす配列
8       int top;//スタックのトップを表わす配列 item の添字
9       int max;//最大の要素数
10
11      /*************コンストラクタ*******/
12      ArrayStack(int max){
13          top=0;
14          this.max=max;
15          items= new double[max];
16      }
17
18      ArrayStack(){
19          this(10);//デフォルトでは 10 アイテム保持
20      }
21
22      /***************メソッド*********/
23      /*
```

```
24          現在保持しているアイテム数を返す。
25          引数：なし
26          戻り値：アイテム数
27         */
28        public int getNum(){
29            return (max-top);
30        }
31
32        /*
33         スタックにアイテムを挿入する (プッシュ)；
34         引数：アイテム (double 値)
35         戻り値：なし
36        */
37        public void push(double d) throws StackOverflowException{
38            /*スタックオーバーフローが生じる場合*/
39            if(top>=max){
40                throw new StackOverflowException(d);
41            }
42
43            /*スタックにデータ保存可能な場合*/
44            items[top]=d;
45            top++;  /*スタック先頭の更新*/
46            return;
47        }
48
49        /*
50         スタックからアイテムを取り出す (ポップ)；
51         引数：なし
52         戻り値：スタックトップのアイテム (double 値)
53        */
54        public double pop() throws StackUnderflowException{
55            /*
56              スタックにアイテムが無い場合
57            */
58            if(top<=0){
59                throw new StackUnderflowException();
60            }
61
62            /*スタックにアイテムがある場合*/
```

```
63            top--;  /*スタック先頭の更新*/
64            return items[top];
65        }
66    }
```

抽象データ型を用いたスタックの利用法をリストに示します。抽象データ型の便利な側面を理解するようにして下さい。

リスト 58:TestStack.java(Ver7.3)

```
1   /*抽象データ型としてのスタックを確かめるサンプルプログラム*/
2   public class TestStack{
3       public static void main(String[] argv){
4           /*スタック準備*/
5           Stack ls=new ListStack(3);
6           Stack as=new ArrayStack(3);
7           /*オーバーフローの例外を実験するには、
8             上の行をコメントアウトし、下のコメントをはずす。*/
9           //Stack as=new ArrayStack(2);
10
11          try{
12              System.out.println("ListStack のテスト");
13              testStack(ls);
14
15              System.out.println("ArrayStack のテスト");
16              testStack(as);
17          }catch(StackException e){
18              e.println();
19          }
20
21          return;
22      }
23
24      public static void testStack(Stack s) throws StackException{
25          /*アンダーフローの例外を実験するには、下のコメントをはずす。*/
26          //System.out.println("pop():"+s.pop());
27
28          double a=7.0;
```

```
29          double b=2.0;
30          double c=8.0;
31          System.out.println("push("+a+")");
32          s.push(a);
33
34          System.out.println("push("+b+")");
35          s.push(b);
36
37          System.out.println("push("+c+")");
38          s.push(c);
39
40          System.out.println("pop():"+s.pop());
41          System.out.println("pop():"+s.pop());
42          System.out.println("pop():"+s.pop());
43          return;
44      }
45  }
```

第8章　パッケージ

これまでみてきたように、Javaのプログラムを作成する際には、通常1つのクラス毎に一つのソースファイル (拡張子「.java」を持つファイル) を作成します。また、コンパイルすることによって、各クラス毎にバイトコードファイル (拡張子「.class」を持つファイル) が生成されます。このように、Javaのプログラミングでは、複数のファイルを管理する必要があります。一方、Javaのプログラミングでは、多数のクラス定義によってプログラムを構築することが多いです。したがって、Javaである程度の規模のプログラムを作成すると、管理しなければならないファイルの数が多くなる傾向にあります。しかし、このような多量のファイル管理はそれ自体が困難です。

また、「あるプログラムのコンパイルや実行に必要なファイルは同一のディレクトリに置く」という制約は、余計にファイル管理を困難にします。

また、これまでのように同一ディレクトリにだけファイルを置くようなファイル管理では、クラス名の選択にも制約を与えてしまいます。すなわち、「新たなクラスを作成する際に、同一ディレクトリ内の (多量にある) ファイル名は使えない」という制約があります。つまり、クラス名やインターフェース名の衝突を避けるという困難さが生じます。

Javaでは、このような状況を回避するため、パッケージという仕組を利用することができます。**パッケージ**とは、Javaの基本単位であるクラスやインターフェースをいくつかまとめたものです。

機能や、利用法等の、意味単位でクラスやインターフェスを分類し、パッケージを作成することが多いです。また、Javaでは、これまでに多くのクラスライブラリが作成されていますが、これらは意味単位にまとめられ、パーケージとして与えられることが多いです。したがって、Javaの多量のクラスライブラリを利用するためにも、パッケージの知識は欠かせません。

8.1　パッケージの作成

パッケージによるクラス[1]の管理には、ファイルシステムにおけるディレクトリの仕組みを利用します。すなわち、通常、同一パッケージ内のクラスは、同じディレクトリに置かれます。パッケージを作成するためには、ファイルシステム上にパッケージ名と同一のディレクトリを作成する必要があります。ここでは、Unixのファイルシステム上でプログラム開発を行なうものとして説明します。まず、次のようにしてディレクトリを作成しま

[1] クラスやインターフェース。本章では、この意味で「クラス」という用語を用いる。

す。(ディレクトリ名とパッケージ名が同じであることに注意すること。)

```
mkdir パッケージ名
```

```
$ls -R
.:
Movable.java  Point.java
$mkdir packageSample
$ls -R
.:
Movable.java  Point.java  packageSample

./packageSample:
$
```

パッケージ名を持つディレクトリの作成によって、パッケージに属するクラスの入れ物が作成されたことになります[2]。このディレクトリにパッケージに属する全てのファイル(ソースファイルおよびバイトコードファイル)を置くことになります。したがって、次のように移動する必要があります。

```
$ ls -R
.:
Movable.java  Point.java  packageSample

./packageSample:
$ mv Movable.java Point.java  packageSample/
$ ls -R
.:
packageSample

./packageSample:
Movable.java  Point.java
$
```

これらの移動したファイルの内容を変更するには、次のように行なうと便利です。

```
$ls -R
.:
packageSample

./packageSample:
Movable.java  Point.java
$emacs packageSample/Movable.java &
```

[2]通常、同じディレクトリにあるクラスは、同じパッケージに入ります。

一方、パッケージに属するソースコード内においても、そのクラスが属するパッケージを宣言する必要があります。すなわち、ソースコード内のクラス定義の前に次のように記述します[3]。

```
package パッケージ名;
```

```
package  packageSample;
```

リスト 59 に Movabl インターフェースを、リスト 60 に Point クラスを示します。こら ら 2 つのクラスとインターフェースは、パッケージ packageSample に属します。

リスト 59:packageSample/Movable.java(Ver8.1)

```
1  /*パッケージ定義を示すサンプルプログラム*/
2  package packageSample;//このクラスが属するパッケージの宣言
3
4  public interface Movable{
5      public void move(double dx,double dy);//(dx,dy) 分移動させる。
6      /*abstract が無くても、抽象メソッドです。*/
7  }
```

リスト 60:packageSample/Point.java(Ver8.1)

```
1  package packageSample;
2
3  /*Point クラス*/
4  public class Point implements Movable{
5      /**********************フィールド*****************************/
6      private double x; //x 座標
7      private double y; //y 座標
8
9      /**********************コンストラクタ**********************/
10
11     /*
12        2 つの座標から点を作成する。
13        引数   : (x 座標 (double) ,y 座標 (double));
14     */
15     public Point(double x,double  y){
```

[3]なお、パッケージを明示的に宣言しなくても、同じディレクトリ内のクラスは (名前の無い) 同一のパッケージに入っているとみなされます。

```
16          this.x=x;
17          this.y=y;
18      }
19
20      /*
21         点のコピーを作成する。
22         引数   : 点 (Point)
23      */
24      public Point(Point pt){
25          x=pt.getX();
26          y=pt.getY();
27      }
28
29      /*
30         デフォルトの点の作成。
31         (引数がなければ原点のコピーを作成する。)
32         引数   : なし
33      */
34      public Point(){
35          this(0.0,0.0);
36      }
37
38      /**********メソッド**********/
39      /*
40         戻り値 : x 座標
41         引数   : なし
42      */
43      public double getX(){
44          return x;
45      }
46
47      /*
48         戻り値 : void
49         引数   : x 座標
50      */
51      public void setX(double x){
52          this.x=x;
53          return;
54      }
```

```
55
56      /*
57         戻り値：y 座標
58         引数　：なし
59      */
60      public double getY(){
61          return y;
62      }
63
64      /*
65         戻り値：void
66         引数　：y 座標
67      */
68      public void setY(double y){
69          this.y=y;
70          return;
71      }
72
73      /*
74         原点までの距離を求めるメソッド
75         戻り値：原点までの距離 (double)
76         引数　：なし
77      */
78      public double distanceToOrigin(){
79          return Math.sqrt(x*x+y*y);
80      }
81
82      /*
83         他の点までの距離を求めるメソッド
84         戻り値：引数で指定された点までの距離 (double)
85         引数　：他の点 (Point 型)
86      */
87      public double distanceToOtherPoint(Point opt){
88          double diffX=x-opt.getX();//x 座標の差
89          double diffY=y-opt.getY();//y 座標の差
90
91          return Math.sqrt(diffX*diffX+diffY*diffY);
92      }
93
```

```
94        /*
95           自分を表示するメソッド
96           戻り値：なし
97           引数　：なし
98        */
99        public void print(){
100           System.out.print(this.toString());//改行なし
101           return;
102       }
103
104       /*
105          自分を文字列に変換するメソッド
106          戻り値：自分自身を表わす文字列
107          引数　：なし
108       */
109       public String toString(){
110           return "("+x+","+y+")";
111       }
112
113
114       /*
115          点を (dx,dy) 分移動するメソッド
116          インターフェースの実装
117          戻り値：なし
118          引数　：移動分を表わす (double dx,double dy)
119       */
120       public void move(double dx,double dy){
121           x+=dx;
122           y+=dy;
123           return;
124       }
125   }
```

こららのクラスやインターフェースをコンパイルするには、次のようにしてコンパイルすることができます[4]。

[4]コンパイルするディレクトリの位置に注意する必要があります。他の場所でコンパイルするにはコンパイラにファイルの位置を指定しなければなりません。細かい指定方法は本資料では省略します。

```
$ ls -R
.:
packageSample

./packageSample:
Movable.java  Point.java
$ javac packageSample/Movable.java
$ ls -R
.:
packageSample

./packageSample:
Movable.class  Movable.java  Point.java
$javac packageSample/Point.java
$ ls -R
.:
packageSample

./packageSample:
Movable.class  Movable.java  Point.class  Point.java
$
```

8.2 パッケージの利用

Javaでパッケージを利用するには、クラスの**完全修飾名**を用います。完全修飾名とは、通常のクラス名やインターフェース名の前にパッケージ名を加えたもので、次のように「.」で区切って記述します[5]。

パッケージ名.型名
`packageSample.Point pt = new packageSample.Point();`

リスト61に完全修飾名を利用してパッケージ宣言されたクラスを利用するプログラムを示します。

リスト 61:`TestPackage.java`(Ver8.1)

```
1   /*パッケージ利用のサンプルプログラム*/
```

[5]このように、Unixのファイルシステム上ではデイレクトリの区切記号として/を用いていますが、Javaのソースコード内ではパッケージの区切記号として.を用います。だから、/と.を対応させて、ファイルとクラスを管理して下さい。

```
2   public class TestPackage{
3       public static void main(String[] argv){
4           packageSample.Point pt=new packageSample.Point(1.0,2.0);
5           System.out.println(""+pt);
6
7           pt.move(1.0,2.0);
8           System.out.println(""+pt);
9
10          return;
11      }
12  }
```

リスト 61 のプログラムの実行結果を示します。コンパイルする際のディレクトリの位置に注意して下さい。

```
$ls
TestPackage.java  packageSample
$javac TestPackage.java
$ls
TestPackage.class  TestPackage.java  packageSample
$java TestPackage
(1.0,2.0)
(2.0,4.0)
$
$
```

このように完全修飾名でパッケージ宣言されたクラスを利用することができるのですが、クラスの完全修飾名を利用すると余分な記述が多くなりがちです。そこで、Java では、パッケージをあらがじめインポートすることによって、パッケージ名を省略することができます。パッケージをインポートするには、クラス定義の前に、次のように記述します。

```
import 完全修飾名;
```

```
import packageSample.Point;
```

このように記述すると、インポートしたクラスにおいてはパッケージ名を省略して記述することができるようになります。

リスト 62 にパッケージをインポートするプログラムを示します。

リスト 62:TestImport.java(Ver8.1)

```
1   /*import によるパッケージ利用のサンプルプログラム*/
2   import packageSample.Point;
```

```
3   /*
4      packageSample 内の Point クラスについて、
5      完全修飾名におけるパッケージ名を省略可能にする。
6   */
7
8   public class TestImport{
9       public static void main(String[] argv){
10          Point pt=new Point(1.0,2.0);
11          System.out.println(""+pt);
12
13          pt.move(1.0,2.0);
14          System.out.println(""+pt);
15
16          return;
17      }
18  }
```

また、import の宣言において、完全修飾名のクラス名の代りに「*」を記述することで、パッケージ内の全てのクラスをインポートすることもできます。

```
import パッケージ名.*;
```

```
import packageSample.*;
```

8.3 パッケージの構成

パッケージは、様々なクラスで利用されるように作成しなければなりません。極端に言えば、世界中の Java クラスの完全修飾名はすべて異なっていなければなりません。そこで、パッケージ名には、通常、所属組織のドメイン名を逆順に表記した文字列を付加することがきまりになっています[6] 。

リスト 63〜65 に、このようなプログラムを示します。これらのプログラムは、前節のスタックの実装の一部だけを示しています。必要な他のプログラムも同様にパッケージに属してあげる必要があります。なお、スタックにおいて、一般のオブジェクトが扱えるように、要素の型を Object に変更してあります。

リスト 63: jp/ac/akita_pu/util/StackException.java(Ver8.2)

```
1   package jp.ac.akita_pu.util;
```

[6] このようなパッケージ名を用いる場合、パッケージに属するファイル「.」を「/」に置き換えたディレクトリに置く必要があります。もちろん、必要ならディレクトリを作成して下さい。

```
2
3   /*スタックに関する例外*/
4   public class StackException extends Exception{
5       /*****フィールド********/
6       /*****コンストラクタ****/
7       public StackException(){
8           super();
9       }
10
11      /*****メソッド****/
12      public void println(){
13          System.out.println("スタックに関する例外が発生しました。");
14          return;
15      }
16
17      public String toString(){
18          return "スタックに関する例外";
19      }
20  }
```

リスト 64:　jp/ac/akita_pu/util/Stack.java(Ver8.2)

```
1   package jp.ac.akita_pu.util;
2
3   /*
4     抽象データ型としてのスタックを表すインターフェース
5    */
6   public interface Stack{
7       /*
8         データの挿入
9         引数：アイテム (Object)
10        戻り値：なし
11        送出例外:オーバーフロー
12      */
13      public void push(Object item) throws StackOverflowException;
14
15      /*
```

```
16          データの取り出し
17          引数：なし
18          戻り値：アイテム (Object)
19          送出例外:アンダーフロー
20        */
21        public Object pop() throws StackUnderflowException;
22    }
```

リスト 65: jp/ac/akita_pu/util/ArrayStack.java(Ver8.2)

```
1   package jp.ac.akita_pu.util;
2
3   /*
4     連結リストを用いたスタックの実装
5     スタックの底が配列の先頭であることに注意する。
6   */
7   public class ArrayStack implements Stack{
8       /**********フィールド************/
9       Object[] items;//スタックを表わす配列
10      int top;//スタックのトップを表わす配列 item の添字
11      int max;//最大の要素数
12
13      /*************コンストラクタ*******/
14      public ArrayStack(int max){
15          top=0;
16          this.max=max;
17          items= new Object[max];
18      }
19
20      public ArrayStack(){
21          this(10);//デフォルトでは 10 アイテム保持
22      }
23
24      /***************メソッド*********/
25      /*
26        現在保持しているアイテム数を返す。
27        引数：なし
```

```
28          戻り値：アイテム数
29         */
30        public int getNum(){
31            return (max-top);
32        }
33
34        /*
35         スタックにアイテムを挿入する (プッシュ)；
36         引数：アイテム (double 値)
37         戻り値：なし
38        */
39        public void push(Object ob) throws StackOverflowException{
40            /*スタックオーバーフローが生じる場合*/
41            if(top>=max){
42                throw new StackOverflowException(ob);
43            }
44
45            /*スタックにデータ保存可能な場合*/
46            items[top]=ob;
47            top++;  /*スタック先頭の更新*/
48            return;
49        }
50
51        /*
52         スタックからアイテムを取り出す (ポップ)；
53         引数：なし
54         戻り値：スタックトップのアイテム (Object 参照)
55        */
56        public Object pop() throws StackUnderflowException{
57            /* スタックにアイテムが無い場合 */
58            if(top<=0){
59                throw new StackUnderflowException();
60            }
61
62            /*スタックにアイテムがある場合*/
63            top--;  /*スタック先頭の更新*/
64            return items[top];
65        }
66    }
```

これらのパッケージ jp.ac.akita_pu.util を利用するプログラムをリスト 66 に示します。

リスト 66: jp/ac/akita_pu/util/TestStack.java(Ver8.2)

```
1  /*パッケージ利用のサンプルプログラム*/
2  import jp.ac.akita_pu.util.*;
3  /*
4      このパッケージに入っているすべてのクラスにおいて、
5      完全修飾名におけるパッケージ名ヲ省略できる。
6  */
7
8  public class TestStack{
9      public static void main(String[] argv){
10         /*スタック準備*/
11         Stack ls=new ListStack(3);
12         Stack as=new ArrayStack(3);
13
14         try{
15             System.out.println("ListStack のテスト");
16             testStack(ls);
17
18             System.out.println("ArrayStack のテスト");
19             testStack(as);
20         }catch(StackException e){
21             e.println();
22         }
23
24         return;
25     }
26
27     public static void testStack(Stack s) throws StackException{
28         Double a=new Double(7.0);
29         Double b=new Double(2.0);
30         Double c=new Double(8.0);
31
32         System.out.println("push("+a+")");
33         s.push(a);
```

```
34
35          System.out.println("push("+b+")");
36          s.push(b);
37
38          System.out.println("push("+c+")");
39          s.push(c);
40
41          System.out.println("pop():"+s.pop());
42          System.out.println("pop():"+s.pop());
43          System.out.println("pop():"+s.pop());
44          return;
45      }
46  }
```

なお、リスト 66 中の Double というクラスは、基本型の double の**ラッパークラス (wrapper class)** と呼ばれるものです。このようなラッパークラスを用いることで、基本型も他の参照型と同様に統一的に扱うことができるようになります。表 8.1 に各基本型におけるラッパークラスを示します。

表 8.1: ラッパークラス

基本型の種類	ラッパークラス	基本型の意味
boolean	Boolean	論理値 (真偽値)
char	Character	16 ビットの Unicode 文字 (1 文字)
byte	Byte	8 ビット整数 (符号付)
short	Short	16 ビット整数 (符号付)
int	Integer	32 ビット整数 (符号付)
long	Long	32 ビット整数 (符号付)
float	Float	32 ビットの浮動小数点数 (符号付)
double	Double	64 ビットの浮動小数点数 (符号付)

第9章　クラスライブラリ

Javaには、これまでのような基本的な仕組だけではなく、意味的な役割によって分類されたプログラム群、すなわち**クラスライブラリ**が存在します。これらのクラスライブラリを用いることで、より便利にJavaプログラミングを行なうことができます。Javaには膨大なクラスライブラリが標準で提供されており、それらを使いこなすにはAPI仕様が欠かせません[1] 。

なお、クラスライブラリは主にパッケージで提供されるので、パッケージをインポートすることによってクラスライブラリ中のクラスが手軽に利用可能になります。また、複数のパッケージによって構成されるクラスライブラリもあります。

本章では、主に利用されるクラスライブラリについて説明します。

9.1　コレクションフレームワーク

7.2節のリスト構造等、よく利用されるプログラムは、クラスライブラリとして提供されていることが多いです。7.2節でも見てきたように、リスト構造は自分で実装することもできます。しかし、自分でプログラムを作成することは必要以上に労力が必要だったり、バグが存在する可能性が高かったりします[2]。それに対して、クラスライブラリとして提供されているプログラムは、これまでに多く利用されており、バグが存在する可能性が極めて小さくなっています[3]。しかし、クラスライブラリでは、汎用な利用が想定されているため、必要以上の機能が提供されることが多いです。

Javaにおいて、リスト構造を実装しているクラスライブラリとして、**コレクションフレームワーク**があります。コレクションフレームワークには、データを便利に扱うためのクラスが多くあり、リスト構造の他にも集合というデータ構造があります。数学的にみれば、リストは順序集合とみなせます。したがって、Javaのコレクションフレームワークでは、順序集合と集合を適切に扱う仕組みが備わっていると考えられます[4]。

なお、コレクションフレームワークは独立したパーケージにはなっておらず、様々な便利クラスを集めたユーティリティパッケージ (`java.util`) に含まれています。

---

[1] 一口にクラスライブラリといっても、規模、種類、利用法は様々なので、クラスライブラリごとに個別に習熟する必要があります。

[2] ただ、自分で構築してみることは、プログラミング能力の向上には効果的です。また、クラスライブラリを用いるときにも、内部の仕組みを理解して用いることができるという利点があります。

[3] このように実際に用いられることによってバグ等の不具合が少なくなっている技術を「枯れた技術」ということがあります。

[4] コレクションとは、集合、順序集合、多重集合等を包括したものと捉えられます。

コレクションフレームワークには、集合を表すインターフェースとして Set があり、そのハッシュを用いた実装として HashSet があります。リスト 67 に集合を扱うプログラムを示します。

リスト 67:TestSet.java(Ver9.0)

```
1  import java.util.*;
2  /*Set は集合の抽象データ型
3    HashSet はハッシュを用いた実装*/
4
5  public class TestSet{
6      public static void main(String[] argv){
7
8          Set s=new HashSet();//集合 s の用意
9
10         Double b = new Double(3.0);
11         Point p =new Point(1.0,2.0);
12
13         s.add(b);
14         s.add(p);
15
16         System.out.println("集合 s の内容を表示します。");
17         System.out.println(s);
18         System.out.println("集合 s の要素数は"+s.size()+"です。");
19
20         return;
21     }
22 }
```

また、コレクションフレームワークには、順序集合を表すインターフェースとして List があり、その連結リストを用いた実装として LinkedList があります。リスト 68 に順序集合を扱うプログラムを示します。

リスト 68:TestList.java(Ver9.1)

```
1  import java.util.*;
2  /*List は順序集合の抽象データ型
3    LinkedList は連結リストを用いた実装*/
```

```
4
5   public class TestList{
6       public static void main(String[] argv){
7
8           List l=new LinkedList();//順序集合lの用意
9
10          Double b = new Double(3.0);
11          Point p =new Point(1.0,2.0);
12
13          l.add(b);
14          l.add(p);
15
16          System.out.println("順序集合lの内容を表示します。");
17          System.out.println(l);
18          System.out.println("順序集合lの要素数は"+l.size()+"です。");
19
20          return;
21      }
22  }
```

コレクションフレームワークでは、コレクション内にある各要素を参照するために、**反復子 (イテレータ)** が用意されています。このイテレータは Iterator というインターフェースとして定義されています。この Iterator には、コレクション内に次の要素が有るかどうかを調べる

```
boolean hasnext();
```

という抽象メソッドと、次の要素を取り出す

```
Object next();
```

という抽象メソッドが定義されています。イテレータは、コレクション中の要素を順に指し示す「カーソル」のような概念して捉えると良いでしょう。

リスト 69:TestIterator.java(Ver9.1)

```
1   import java.util.*;
2   /*反復子(イテレータ)の効果を調べるサンプルプログラム*/
3
4   public class TestIterator{
5       public static void main(String[] argv){
6
```

```
7           List l=new LinkedList();//順序集合 l の用意
8
9           Double b = new Double(3.0);
10          Point p =new Point(1.0,2.0);
11
12          l.add(b);
13          l.add(p);
14
15          Iterator i= l.iterator();//イテレータの生成
16          while( i.hasNext() ){
17              Object o = i.next();
18              System.out.print(" "+o);
19          }
20          System.out.println();
21          return;
22      }
23  }
```

9.2　入出力ライブラリ

これまでのプログラムでは、データを (標準出力へ) 出力することしか行なってきませんでした。しかし、一般に、プログラムは、入力を処理して出力を生成するものです。本節では、Java でのデータ入出力を取り扱います。

実は、Java では、この入出力の処理は、**I/O パッケージ** (`java.io`) と呼ばれる入出力ライブラリを用いて行なうことが一般的です。この I/O パッケージの利用は、C 言語の `scanf` 文のように単純ではなく、少々複雑な面もあります。そのため、これまで扱ってきませんでした。

9.2.1　標準入出力

リスト 70 に標準入力から入力を行なうためのプログラムを示します[5]。

リスト 70:　`jp/ac/akita_pu/util/Stdin.java`(Ver9.1)

```
1
```

[5] Unix のリダイレクションを用いることで、膨大なデータをファイルから読み込むこともできます。

```
2  package jp.ac.akita_pu.util;
3
4  import java.util.*;
5  import java.io.*;
6
7  /* 標準入力から int, double, String を切り出すためのサポートクラス */
8  public class Stdin {
9      /******************フィールド*****************/
10     private String delimiter = " ";//区切文字
11     private String line = "";//解析中の行
12     private StringTokenizer tokenizer = null;//行の解析クラス(java.util)
13     private BufferedReader reader = null; //解析中のキャラクタストリーム
14
15     /******************コンストラクタ*****************/
16     /** 区切文字を指定する場合のコンストラクタ。*/
17     public Stdin(String delim) {
18         delimiter = delim;
19         reader = new BufferedReader(new InputStreamReader(System.in));
20     }
21
22     /** 区切文字を指定しない場合のコンストラクタ。
23         デフォルトでスペースを区切文字とする。 */
24     public Stdin() {
25         this(" ");
26     }
27
28     /*****************メソッド*****************/
29     /** 整数を切り出す */
30     public int getInt() throws IOException {
31         return Integer.parseInt(this.getToken());
32     }
33
34     /** 浮動小数点数を切り出す */
35     public double getDouble() throws IOException {
36         return Double.parseDouble(this.getToken());
37     }
38
39     /** 文字列を切り出す */
40     public String getString() throws IOException {
```

```
41          return new String(this.getToken());
42      }
43
44      /** 入力からトークンを切り出す。 */
45      private String getToken() throws IOException {
46          /* 切り出すべき文字列が指定されていなかったり、
47           * もう文字列にトークンが存在しない場合は
48           * 新しい行を読む
49           */
50          while (tokenizer==null || !tokenizer.hasMoreTokens()) {
51              line = reader.readLine();
52              if (line==null) {
53                  throw new EOFException();
54              }
55              tokenizer = new StringTokenizer(line, delimiter);
56          }
57          // ここに来たということは、文字列中にまだトークンがあるということ
58          return tokenizer.nextToken();
59      }
60
61      /** 今読んでいる行を一行読み飛ばす。
62          今読んでいる行の次の行の先頭に移動する。
63          まだ一行も読んでいない場合は一行目の先頭に移動する。
64      */
65      public void nextLine() throws IOException {
66          /* 新しい行を読む
67           */
68          line = reader.readLine();
69          if (line==null) {
70              throw new EOFException();
71          }
72          tokenizer = new StringTokenizer(line, delimiter);
73          return;
74      }
75  }
```

リスト 71 に Stdin クラスを利用するプログラムを示します。

リスト 71:TestStdin.java(Ver9.1)

```
1  import jp.ac.akita_pu.util.*;
2  import java.io.*;
3
4  public class TestStdin{
5      public static void main(String[] argv){
6
7          Stdin stdin = new Stdin();
8          int intValue =0;
9
10         try{
11             while(true){
12                 intValue = stdin.getInt();
13                 System.out.println(intValue);
14             }
15         }catch(EOFException e){
16             System.out.println(""+e);
17         }catch(IOException e){
18             System.out.println(""+e);
19         }
20
21         return;
22     }
23 }
24
```

9.2.2 ファイル入出力

標準入出力だけでなくファイルへ直接読み書きしたいこともよくあります。リスト 72 にファイルからデータを読み込むためのプログラムを示します。

リスト 72: jp/ac/akita_pu/util/Filein.java(Ver9.2)

```
1  package jp.ac.akita_pu.util;
2
3  import java.io.*;
4  import java.util.*;
```

```
5
6   /** ファイルから int, double, String を切り出すためのサポートクラス */
7   public class Filein{
8       /**********************フィールド**********************/
9       private String delimiter = " ";//トークンの区切文字。
10      private String line = "";// 現在解析中の行。
11      private StringTokenizer tokenizer = null;// 現在の行を解析するトークナイザ
12      private BufferedReader reader = null;// 解析中のキャラクタストリーム。

13
14      /**********************コンストラクタ**********************/
15      /** 区切文字を指定する場合のコンストラクタ。
16          区切文字からなる文字列を引数に与える */
17      public Filein(String name,String delim)throws FileNotFoundException{
18          delimiter = delim;
19          reader = new BufferedReader(new FileReader(name));
20      }
21
22      /** 区切文字を指定しない場合のコンストラクタ。
23          デフォルトでスペースを区切文字とする。 */
24      public Filein(String name)throws FileNotFoundException{
25          this(name," ");
26      }
27
28
29      /**********************メソッド**********************/
30      /** 整数を切り出す */
31      public int getInt() throws IOException {
32          return Integer.parseInt(this.getToken());
33      }
34
35      /** 浮動小数点数を切り出す */
36      public double getDouble() throws IOException {
37          return Double.parseDouble(this.getToken());
38      }
39
40      /** 文字列を切り出す */
41      public String getString() throws IOException {
```

```
42          return new String(this.getToken());
43      }
44
45      /** 入力からトークンを切り出す。 */
46      private String getToken() throws IOException {
47          /* 切り出すべき文字列が指定されていなかったり、
48           * もう文字列にトークンが存在しない場合は
49           * 新しい行を読む
50       */
51          while (tokenizer==null || !tokenizer.hasMoreTokens()) {
52              line = reader.readLine();
53              if (line==null) {
54                  throw new EOFException();
55        }
56              tokenizer = new StringTokenizer(line, delimiter);
57          }
58          // ここに来たということは、文字列中にまだトークンがあるということ
59      return tokenizer.nextToken();
60      }
61
62    /** 今読んでいる行を一行読み飛ばす。
63          今読んでいる行の次の行の先頭に移動する。
64          まだ一行も読んでいない場合は一行目の先頭に移動する。
65    */
66      public void nextLine() throws IOException {
67          /* 新しい行を読む*/
68      line = reader.readLine();
69      if (line==null) {
70          throw new EOFException();
71      }
72      tokenizer = new StringTokenizer(line, delimiter);
73      return;
74      }
75  }
```

リスト 73 に Filein クラスを利用するプログラムを示します。

リスト 73:TestFilein.java(Ver9.2)

```
1  import jp.ac.akita_pu.util.*;
2  import java.io.*;
3
4  public class TestFilein{
5      public static void main(String[] argv){
6
7          try{
8              Filein fin = new Filein("inputFile");
9              int i=0;
10             int j=0;
11             double d=0.0;
12             String s="";
13
14             i=fin.getInt();
15             j=fin.getInt();
16             d=fin.getDouble();
17             s=fin.getString();
18
19             System.out.print(i+" ");
20             System.out.print(j);
21             System.out.println();
22             System.out.println(d);
23             System.out.println(s);
24         }catch(FileNotFoundException e){
25             System.out.println(""+e);
26         }catch(EOFException e){
27             System.out.println(""+e);
28         }catch(IOException e){
29             System.out.println(""+e);
30         }
31
32         return;
33     }
34 }
35
```

リスト 74 にファイルへデータを書き込むためのプログラムを示します。

リスト 74:　jp/ac/akita_pu/util/Fileout.java(Ver9.2)

```
1  package jp.ac.akita_pu.util;
2  import java.io.*;
3
4  /** ファイルへ文字を書き込むサポートクラス*/
5  public class Fileout{
6      /**********************フィールド**********************/
7      public PrintWriter out; //ファイルへ書き込むためのクラス
8      /**********************コンストラクタ**********************/
9      public Fileout(String name) throws IOException{
10         out=new PrintWriter(new FileWriter(name),true);
11         /*文字ストリームの自動書き込みを真に設定*/
12     }
13     /**********************メソッド**********************/
14 }
```

リスト 75 に Fileout クラスを利用するプログラムを示します。

リスト 75:TestFileout.java(Ver9.2)

```
1  import jp.ac.akita_pu.util.*;
2  import java.io.*;
3
4  public class TestFileout{
5      public static void main(String[] argv){
6
7          try{
8              Filein fin = new Filein("inputFile");
9              Fileout fout=new Fileout("outputFile");
10
11             int i=0;
12             int j=0;
13             double d=0.0;
14             String s="";
15
16             i=fin.getInt();
17             j=fin.getInt();
18             d=fin.getDouble();
```

```
19              s=fin.getString();
20
21              fout.out.print(i+" ");
22              System.out.print(i+" ");
23
24              System.out.print(j);
25              fout.out.print(j);
26
27              fout.out.println();
28              System.out.println();
29
30              fout.out.println(d);
31              System.out.println(d);
32
33              fout.out.println(s);
34              System.out.println(s);
35
36          }catch(FileNotFoundException e){
37              System.out.println(""+e);
38          }catch(EOFException e){
39              System.out.println(""+e);
40          }catch(IOException e){
41              System.out.println(""+e);
42          }
43
44          return;
45      }
46  }
47
```

9.3　AWT

これまでのプログラムはすべて文字による入出力しか取り扱えませんでした。

一方、Java には、ウィンドウを用いた GUI(Graphical User Interface) を持つプログラムを比較的容易に作成するためのクラスライブラリとして、**AWT(Abstract Window Toolkit) ライブラリ**や **Swing ライブラリ**があります。

ここでは、基本的な AWT クラスライブラリを用いたプログラムを幾つか示すだけにし

ます。なお、AWT クラスライブラリは、java.awt ではじまるパッケージ群で提供されます[6]。

まず、リスト 76 にウィンドウ上に文字を表示するプログラムを示します。

リスト 76:HelloAWT.java(Ver9.3)

```
1   import java.awt.*;
2
3   public class HelloAWT{
4       public static void main(String[] argv){
5
6           Frame f=new Frame();
7
8           Label l= new Label("Hello AWT");
9
10          f.add(l);
11          f.pack();
12
13          f.setVisible(true);
14
15          return;
16      }
17  }
18
```

このプログラムでは、ウィンドウ (フレーム) を一枚作成し、その中に「Hello AWT」という文字で構成されたラベルが配置されています. なお、AWT では、画面上に表示されるプログラム部品 (GUI 部品) を**コンポーネント**と呼び、Component クラスを継承しなければなりません。当然フレームを表わす Frame や、ラベルを表わす Label も Component クラスを継承しています[7]。なお、フレームとは、タイトルバーの付いたウインドウ (Window クラス) のことです。

このように生成されたフレームは、通常のマウスクリックすらも受け付けません[8]。マウスクリックによって終了するフレームを作成するためには、通常、AWT で用意している基本的なプログラム部品である Frame クラスを継承したクラスを作成することで行ない

[6]個々のライブラリ構成や利用法は、API 仕様や他の文献によって調べて下さい。

[7]AWT では、他にも Component を継承している様々なクラス、すなわち GUI 部品が用意されています。API 仕様や文献等で調べると良いでしょう。

[8]プログラムを起動した端末で C-c として終了して下さい。

ます。すなわち、継承によって**イベント処理機能**を GUI 部品に追加することで、マウスやキーボード等からも GUI 部品を動作・処理することができるようになります。

前章までのプログラムにおいては、クラスを利用するには別のクラスから呼び出す必要がありました。しかし、イベント処理機能を持ったプログラムでは、イベントが発生したことを自動的に判断して、自発的に動作を開始します。このようなプログラムの動作の仕組みを**イベント処理**と呼ぶことがあります。例えば、フレーム上でマウスをクリックすると、クリックに対応した「クリックイベント」が自動的に発生し、「クリックイベント」を期待しているクラスに、その「クリックイベント」が自動的に通知されます。しかし、リスト 76 のプログラムは、このイベント処理機能を持たないため、何の動作もしません[9]。

リスト 77 に `Frame` に、イベント処理機能を持たせたクラス (`EventFrame`) を示します。

リスト 77:`EventFrame.java`(Ver9.3)

```
1   import java.awt.*;
2
3   public class EventFrame extends Frame{
4
5       /***************フィールド************/
6       /***************コンストラクタ************/
7       public EventFrame(String title){
8           super(title);
9           enableEvents(AWTEvent.WINDOW_EVENT_MASK);
10      }
11
12      /**************メソッド************/
13      public void processEvent(AWTEvent event){
14          if(event.getID()==Event.WINDOW_DESTROY){
15              System.exit(0);
16          }
17          return;
18      }
19  }
```

このイベント処理機能を持ったフレーム (`EventFrame`) を利用するプログラムをリスト 78 に示します。

[9]なお、AWT の GUI 部品上で発生する「イベント」は、`AWTEvent` クラスを継承します。マウスのクリックやドラッグ、キーボードからの入力等様々な種類のイベントがあります。

リスト 78:HelloEvent.java(Ver9.3)

```
1   import java.awt.*;
2
3   public class HelloEvent{
4       public static void main(String[] argv){
5           Frame ef= new EventFrame("HelloEvent");
6
7           Label l=new Label("Hello Event!!");
8
9           ef.add(l);
10          ef.pack();
11
12          ef.setVisible(true);
13
14          return;
15      }
16  }
```

関連図書

[1] 岡野謙一、 重野寛、 古賀祐匠. *Java* **教科書**. ソフト・リサーチ・センター, 1999.

[2] ケン・アーノルド、 ジェームズ・ゴスリン、 デビッド・ホームズ. **プログラミング言語** *Java* **第** *3* **版**. ピアソン・エヂュケーション, 2001.

[3] 久野禎子、 久野靖. *Java* **によるプログラミング入門**. 共立出版, 2001.

[4] 茨木俊秀. *C* **によるアルゴリズムとデータ構造**. 昭晃堂, 1999.

[5] クリフォード・シェーファー. *Java* **によるデータ構造とアルゴリズム**. ピアソン・エヂュケーション, 1999.

[6] 草苅良至. 2002 年度セミナー 2(Java 編) 資料. 2002.

[7] 能登谷順一. Java 入門 (2003 年度セミナー 2 資料). 2003.

[8] J.D. ウルマン A.V. エイホ, J.E. ホップクロフト. **データ構造とアルゴリズム**. 培風館, 1987.

[9] G. Cornell C.E.Horstmann. **コア** *Java2 Vol.1* **基礎編**. アスキー, 2000.

[10] G. Cornell C.E.Horstmann. **コア** *Java2 Vol.2* **応用編**. アスキー, 2000.

[11] Robert Lafore. *Java* **で学ぶアルゴリズムとデータ構造**. ソフトバンク, 1999.

関連図書について

Java 言語

[1] プログラムのまったくの初心者用に対する、Java を用いたプログラミングの教科書。

[3] プログラムのまったくの初心者用に対する、Java を用いたプログラミングの教科書。[1] より厳密に文法を扱っており、本資料のように BNF 記法を用いている。

[2] Java 言語を作成した人による Java 言語の「バイブル」。これから Java 言語を使用する限り傍らに置くことを薦める。ただし、ある程度の素養が無いと読み進めるのは困難かも。本演習後に利用することを薦める。

[6] 2002 年度の資料。

[7] 2003 年度の資料。

[9] プログラマ用、初学者には読みづらいかも。主に、本資料の 4 章までの内容を扱っている。本演習後に利用することを薦める。

[10][9] の続編。様々な Java の API の中から、主に利用さえるものの解説。

オブジェクト指向

アルゴリズムとデータ構造について

[8] プログラミング言語に依存しない、一般的なデータ構造とアルゴリズムの教科書。著名な著者らによる本で、基礎からしっかりと学べる。Pascal が分かると読みやすいかも。

[4] C を用いたデータ構造とアルゴリズムの教科書。セミナーの教科書

[11] Java 言語を用いたデータ構造とアルゴリズムの教科書。

[5] Java を用いたデータ構造とアルゴリズムの教科書。