人にやさしいプログラミングの哲学

CreW Project
http://www.crew.sfc.keio.ac.jp/

2003 年 5 月 8 日

目次

はじめに

はじめに

「人にやさしいプログラミングの哲学」は、初めてのプログラミングから、オブジェクト指向プログラミングの基礎までを学習するためのカリキュラムです。このカリキュラムは、単にプログラミング言語の文法や記法などの知識を得るのではなく、

> 実際にそれらの知識を使ったプログラムが書けるようになること
> 単に動くプログラムが書けるようになるだけでなく、使いやすく、メンテナンスのしやすいプログラムが書けるようになること
> 手続き指向やオブジェクト指向などの技法の背景にある「考え方」を捉えること

を目的としています。

プログラミングは文法などの「知識」だけを記憶しても実際に書けるようになりません。これは、例えば、野球のルールを知っていても、野球をプレーすることが出来ないのと同じ理由です。ルールはすぐに覚えることが出来ますが、実際に野球をプレーできるようになるためには、”いかに勝つか”といった視点で、基本的な戦術を学びながら、練習を積むことが必要です。このテキストでは、小規模ではありますが実際のアプリケーションを例題として用い、その開発過程をたどりながら、練習の方法を示していきますので、それに従ってプログラムを書く練習を積みましょう。

また、実際のプログラミング開発現場で求められるのは、単に動くプログラムを書けることだけでなく、使いやすく、メンテナンスのしやすいプログラムが書けることです。大規模なソフトウエアの開発現場では、複数の人が一緒に開発することが多いので、知識を共有するためには、プログラムの構造が整理された、人間が理解できるプログラムである必要があります。このテキストでは、そのようなプログラムのガイドラインを示していきますので、最初は真似をしながら、そしてそれを自分のプログラムに取り入れながら、自分のものとするように頑張ってください。

カリキュラムの取り組みかた

「考え方」を捉えよう

このカリキュラムの「哲学」という名前には、単にプログラミング言語の文法や記法などの知識を記憶するのではなく、背景にある「考え方」を身に付けるためのカリキュラムであるという意味がこめられています。

移り変わりが激しいのが IT 技術の特徴です。このテキストではプログラミング言語として Java を扱いますが、5 年後、10 年後も Java が最前線の技術として使われているとは言い切れません。しかし、背景にある「考え方」はそう簡単に変わるものではありません。例えば、オブジェクト指向手法は全く新しい手法ではなく、構造化手法の問題を改善するために考えられてきた手法です。その背景には、いかに人に理解できるプログラムを書くかという議論の積み重ねがあるのです。

こうした基本的な「考え方」は、新しい技術の習得を助けてくれます。このカリキュラムは、そうして身についた考え方を基礎として、技術を適切に適用、応用していくことが出来るように人材を育てることを目指しています。

それゆえ、このカリキュラムでは、新しい文法や記法などの知識が、「何故必要なのか?」「どのように使ったら効果的なのか?」という視点から議論をすすめます。是非よいプログラムを書くにはどうしたらよいか、という視点から、議論し、徐々に「考え方」や基本原理の理解を深めてください。

こうした理由から、テキストには”あえて”空欄を作ってあります。講義や議論を通して、空欄を自分なりに埋めていきましょう。最終的に自分が学習したプログラムの「考え方」のテキストが出来上がるはずです。

人にやさしいプログラムを目指そう

このテキストで学ぶソフトウエア作成技法の指針となっているのは、本テキストのタイトルともなっているように、「人にやさしいプログラムを書く」ということです。

「人にやさしいプログラム」の基本は、

> 意図した通りに動く（バグのない）プログラム
> 使う人にとって使いやすいプログラム
> 人に理解できるソースコードが書かれたプログラム

であることです。

この基本的な「考え方」はこのテキストの全てに貫かれた基本姿勢です。そのため、このカリキュラムでは、例え最初の小規模なプログラムでも、これらのことを意識して「人にやさしいプログラム」を示していきます。

テキストの構成

本テキストは 2 部構成になっています。

第一部　構造化プログラミング編　第一部では、人にやさしいプログラミングの基本を学習します。

プログラムの書法から、HCP チャートを用いた設計の技法、変数や手続きを用いた抽象化技法を使って、いかに人が理解できるプログラムを書くかという議論をします。

第二部　オブジェクト指向プログラミング編　さらに人にやさしいプログラミングを書く技法として、オブジェクト指向プログラミングに挑戦します。

オブジェクト指向技法の考え方自体はさほど難しくありませんが、重要なのは、どのようにオブジェクト指向の手法を使うかというところにあります。第二部では、オブジェクト指向手法を使って、いかに人が理解できるプログラムを書くかという議論をします。

第 I 部

構造化プログラミング編

第 1 章

人にやさしいプログラムの書法

この章で学習すること

人にやさしいプログラムの書法でプログラムが書ける
- プログラムにインデントが付けられる
- プログラムに適切な変数名が付けられる
- プログラムに適切なコメントが付けられる

HCP チャートを使ってプログラムの構造を設計できる

ユーザが使うことを考慮したプログラムが書ける
- ユーザインターフェイスの基本構造を説明できる
- ユーザの不正な入力を考慮したプログラムが書ける

1.1 人にやさしいソースコードとは

次のプログラム (リスト 1 とリスト 2) を比較してみましょう。

リスト 1: 足し算プログラム（一般的な教科書のスタイル）

```
 1: public class AddTwoNumberSample {
 2:
 3:   public static void main(String[] args) {
 4:     int a, b, c;
 5:     a = Input.getInt();
 6:     b = Input.getInt();
 7:     c = a + b;
 8:     System.out.println(c);
 9:   }
10:
11: }
```

リスト 2: 足し算プログラム（このテキストのスタイル）

```
 1: /**
 2:  * 足し算計算機アプリケーション
 3:  *
 4:  * ２つの数をキーボードから読み込み、足し算の結果を表示する
 5:  * ２数とも、０であったら、終了する
 6:  *
 7:  * @author Manabu Sugiura
 8:  * @version $Id: AddTwoNumberApplication.java,v 1.9 2003/05/20 12:24:47 duskin Exp $
 9:  */
10: public class AddTwoNumberApplication {
11:
12:   public static void main(String[] args) {
13:     AddTwoNumberApplication addTwoNumberApplication =
14:       new AddTwoNumberApplication();
15:     addTwoNumberApplication.main();
16:   }
17:
18:   void main() {
19:
20:     int firstNumber; //１番目に入力された整数
21:     int secondNumber; //２番目に入力された整数
22:     int result; //２つの整数の足し算の結果
23:
24:     //アプリケーションの説明をする
25:     System.out.println("２つの数の和を求めます。");
26:     System.out.println("（2 数に 0 を入力すると終了します）");
27:
28:     //１番目の数を入力する
29:     System.out.print("１つ目の整数を入力してください>>");
30:     System.out.flush();
31:     firstNumber = Input.getInt();
32:
33:     //２番目の数を入力する
34:     System.out.print("２つ目の整数を入力してください>>");
35:     System.out.flush();
36:     secondNumber = Input.getInt();
37:
38:     //加算を繰り返す
39:     while (firstNumber != 0 || secondNumber != 0) { //２数とも０なら終了コード
40:
41:       //計算する
42:       result = firstNumber + secondNumber;
43:
44:       //結果を表示する
45:       System.out.println("２つの数の和は" + result + "です。");
46:
47:       //１番目の数を入力する
48:       System.out.print("１つ目の整数を入力してください>>");
49:       System.out.flush();
50:       firstNumber = Input.getInt();
51:
52:       //２番目の数を入力する
53:       System.out.print("２つ目の整数を入力してください>>");
54:       System.out.flush();
```

```
55:       secondNumber = Input.getInt();
56:     }
57:
58:     //アプリケーションが終了したことを知らせる
59:     System.out.println("アプリケーションを終了しました。");
60:   }
61: }
```

どちらも同じことをするプログラムですが、見た目は大分異なりますね。

考えてみよう

気が付いたことを議論してみましょう。

1.2　人にやさしいソースコードの書法

次のプログラムを読んでみましょう。このプログラムがどのようなプログラムかが分かるでしょうか。答えは大方の人が No だと思います。(分かったとしても相当時間がかかったのではないでしょうか。)

これからこのプログラムを、人にやさしいソースコードに直していきましょう。

リスト 3: 例題プログラム

```
 1: public class Example {
 2:
 3:   public static void main(String[] args) {
 4:     Example example = new Example();
 5:     example.main();
 6:   }
 7:
 8: void main(){
 9: int a = 49;
10: int b = 73;
11: int c = 100;
12: int d = 45;
13: int e = 25;
14: double x = a + b + c + d + e;
15: double y = x / 5.0;
16: y = y * 10;
17: if((y % 10) >= 5){
18: y = y + 10;
19: }
20: int z = (int)(y / 10);
21: System.out.println(z);
22: }
23: }
```

オマジナイ

Java はオブジェクト指向言語であるがゆえに、どんなに簡単なプログラムでも、初心者にとって理解するのが難しいいくつかの記述が必要です。

そんな”オマジナイ”をここで確認しておきましょう。

package—, import— この記述は、テキスト作成上の都合で記述されているものです。実際のプログラムでは**記述しないで**下さい。

クラス宣言 Java のプログラムは全て「クラス」と呼ばれるプログラムの単位の中に記述されます。（詳しくはオブジェクト指向編で）ですから、どのような小さなプログラムでも必ず一つクラスを作る必要があります。

クラスの記法は次のとおりです。クラス名は自分で決められますが、他はオマジナイです。中括弧「{}」で囲まれたブロックの中にメソッドなどのプログラムの要素が記述されます。

```
public class [クラス名]{

  （ここにプログラムの要素を記述する）

}
```

また、Java のプログラムではファイル名に気をつける必要があります。ファイル名は必ず**クラス名に「.java」をつけたもの**にしなければなりません。

例えば、この例ですと、ファイル名は「Sample.java」にする必要があります。

public static void main(String args[]) **メソッド** クラスの中に、中括弧で囲まれた記述が２つあります。これをメソッドといいます。クラスの中には、いくつでもメソッドを記述することができますが、第 4 章でメソッドを詳しく学習するまで、2 つのメソッドでプログラムを記述します。

このメソッド記述は全てオマジナイです。次のような書式で記述してください。

```
public static void main(String[] args){
    [クラス名] [クラス名の先頭を小文字にした変数] = new [クラス名]();
    [クラス名の先頭を小文字にした変数].main();
}
```

void main() メソッド　2 つのメソッドのうち、こちらがアプリケーションプログラムを記述するメソッドです。プログラムは main() メソッドから始まる [1]ことを覚えておきましょう。書式は次のとおりで、中括弧の中にアプリケーションプログラム (行いたい仕事) を記述します。

```
void main(){

    （ここにアプリケーションプログラムを記述する）

}
```

[1] 本当は public static void main(String[] args) メソッドからプログラムが始まります。今回は public static void main(String[] args) から main() メソッドを呼び出し（メソッドに関しては 4 章以降を参照）ています。

1.2.1 意味のまとまりを意識する

ただプログラムの処理がだらだらと書かれているソースコードは、人にやさしくありません。処理の意味を考えて、いくつかのまとまりを作ると人間にとってわかりやすくなります。

プログラムにおいて、処理のまとまりのことをブロックと呼びます。ブロックが分かりやすいように記述するには、インデントと空行を効果的に使うことが重要です。例をリスト 4 に示します。

リスト 4: 例題プログラム（ブロックを意識した）

```
 1: public class Example {
 2:
 3:   public static void main(String[] args) {
 4:     Example example = new Example();
 5:     example.main();
 6:   }
 7:
 8:   void main() {
 9:
10:     int a = 49;
11:     int b = 73;
12:     int c = 100;
13:     int d = 45;
14:     int e = 25;
15:
16:     double x = a + b + c + d + e;
17:     double y = x / 5.0;
18:
19:     y = y * 10;
20:     if ((y % 10) >= 5) {
21:       y = y + 10;
22:     }
23:     int z = (int) (y / 10);
24:
25:     System.out.println(z);
26:   }
27: }
```

ブロック プログラムにおいて、処理のまとまりのことをブロック (図 1.1) と呼び、Java 言語では、中括弧「{}」で囲って表現します。（クラスやメソッドもブロックの一種です。）Java のプログラムはブロックの入れ子で構成されています。

インデント インデント (図 1.2) とは、字下げをすることです。字下げには TAB やスペースを使用します。（TAB なら 1 つ分、スペースなら 2 つ分を目安とするとよいでしょう）字下げをすることによって、入れ子になったブロックの関係をはっきり表すことができます。

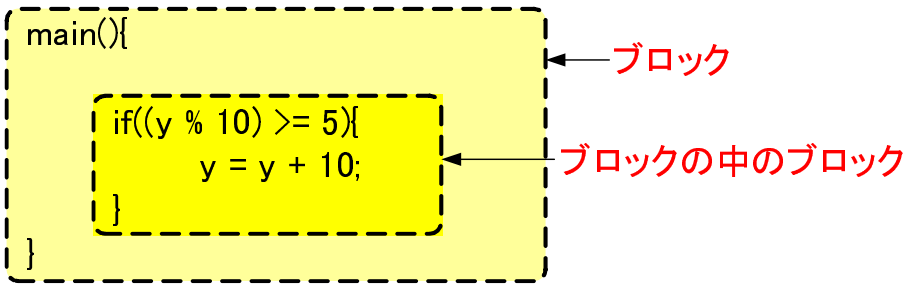


図 1.1: ブロック

```
main(){
    if((y % 10) >= 5){
        y = y + 10;
    }
}
```

図 1.2: インデント

空行で表現されたブロック　中括弧で囲まれたブロックだけでなく、空行を入れることにより、ブロックの中に意味のまとまりを作って、プログラムをより見やすくします。(図 1.3)

```
double x = a + b + c + d + e;
double y = x / 5.0;

y = y * 10;
if ((y % 10) >= 5) {
        y = y + 10;
}
int z = (int) (y / 10);

System.out.println(z);
```

図 1.3: 空行によるブロック

1.2.2 変数に適切な名前をつける

Java 言語では、クラスや変数、メソッドの名前は自由につけられます。では「a」や「bbb」のような名前がついていた場合、他の人が果たしてその意味を理解してくれるでしょうか。人に分かりやすいソースコードを書くためには、 クラスや変数の名前はその意味が分かるような名前にする必要があります。クラスや変数に名前をつけるときは、子供に名前を付ける気持ちで望みましょう。クラスや変数、メソッドについて後ほど詳しく説明します。

リスト 5: 成績管理プログラム（適切な変数名をつけた）

```
 1: public class ScoreAdministratorSample {
 2:
 3:   public static void main(String[] args) {
 4:     ScoreAdministratorSample scoreAdministratorSample =
 5:       new ScoreAdministratorSample();
 6:     scoreAdministratorSample.main();
 7:   }
 8:
 9:   void main() {
10:
11:     int japanese = 49;
12:     int mathematics = 73;
13:     int science = 100;
14:     int civics = 45;
15:     int english = 25;
16:
17:     double total = japanese + mathematics + science + civics + english;
18:     double average = total / 5.0;
19:
20:     average = average * 10;
21:     if ((average % 10) >= 5) {
22:       average = average + 10;
23:     }
24:     int result = (int) (average / 10);
25:
26:     System.out.println(result);
27:   }
28: }
```

1.2.2.1 Java の命名規則

Java 言語を記述するときの一般的な命名規則 [2]の説明をします。本テキストのソースコードはすべてこの規則に従っています。

[2] クラス名やメソッド名、変数名に関しては、数字が先頭にくるもの、予約語、演算子を含むものは使用できません。

クラス名

– 最初の文字は大文字に

– 複数の単語の時は各単語の最初の文字を大文字に

例えば、旅券予約アプリケーションには、「TicketReserveApplication」となります。クラス名とファイル名は同じにしなければならないので、この場合ファイル名は「TicketReserveApplication.java」です。

メソッド名

– 最初の文字は小文字に

– 複数の単語の時は各単語の最初の文字を大文字に

メソッドはある処理を行うものなので大抵は動詞から始めます。例えば、現在の時刻を調べるメソッドは、「getCurrentTime」 といったメソッド名になります。

変数名

– 最初の文字は小文字に

– 複数の単語の時は各単語の最初の文字を大文字に

変数名は、メソッド名と同様の規則です。例えば、苗字を記憶する変数名は、「familyName」 といった変数名になります。

1.2.3 コメントをつける

ソースコードの中にコメントを書き込むことができます。このコメントはプログラムの処理には全く影響しません。コメントは内容が重要です。内容については次項で詳しく議論します。とりあえずコメントをつけたプログラムをリスト 6 に示します。

リスト 6: 成績管理プログラム（とりあえずのコメントを付けたもの）

```
 1: public class ScoreAdministratorSample {
 2:
 3:   public static void main(String[] args) {
 4:     ScoreAdministratorSample scoreAdministratorSample =
 5:       new ScoreAdministratorSample();
 6:     scoreAdministratorSample.main();
 7:   }
 8:
 9:   void main() {
10:     //変数を宣言し値を代入する
11:     int japanese = 49; //整数型 japanese という変数
12:     int mathematics = 73; //整数型 mathematics という変数
13:     int science = 100; //整数型 science という変数
14:     int civics = 45; //整数型 civics という変数
15:     int english = 25; //整数型 english という変数
16:
17:     //合計を保存しておく変数 total を 5 で割り、変数 average に代入する
18:     double total = japanese + mathematics + science + civics + english;
19:     double average = total / 5.0;
20:
21:     //average を 10 倍し，10 で割ったあまりを調べる
22:     average = average * 10;
23:     if ((average % 10) >= 5) { //余りが 5 以上なら
24:       average = average + 10; //10 を加える
25:     }
26:     //結果を result に代入する
27:     int result = (int) (average / 10);
28:
29:     //変数 result の値を表示する
30:     System.out.println(result);
31:   }
32: }
```

1.2.3.1 Java でのコメントの記述方法

コメントの記述方法を間違えると、コンパイルエラーになるので注意しましょう。

範囲指定コメント プログラム中、「/*」 から 「*/」までのすべての文字は、コメントとして扱われます。複数行でも構いません。

```
/* この中がコメント */
```

Java の慣習として（Javadoc というドキュメントを自動生成するため）始まりは「/**」とし、複数行にわたる場合は、行の始めに「*」をつけます。

```
/**
 * Java での標準形式
 */
```

行コメント　プログラム中、「//」からその行の最後までのすべての文字は、コメントとして扱われます。次の行までの効果はありません。

```
int x;//この行のここから先はすべてコメント
```

考えてみよう

先ほどの (リスト 6) を読んでみてこのプログラムが何をするものであるのか考えてみましょう

1.2.4 ひとにやさしいコメントの書法

1.2.4.1 プログラムの目的を明らかにするコメント

コメントはただ書けばよいというものではありません。
具体的に見てみましょう。次のコードは先のリスト 6 から抜き出したものです。

```
//average を 10 倍し，10 で割ったあまりを調べる
average = average * 10;
if ((average % 10) >= 5) { //余りが 5 以上なら
  average = average + 10; //10 を加える
}
```

このコメントを次のものと比べてみましょう。

```
//平均を四捨五入する
average = average * 10;
if ((average % 10) >= 5) { //1の位が5以上なら
  average = average + 10; //繰り上げる
}
int result = (int) (average / 10);
```

人にやさしいコメントをつけたプログラムをリスト 7 に示します。

リスト 7: 成績管理プログラム（人にやさしいコメント付）

```
 1: /**
 2:  * ×○中学校の成績管理プログラム
 3:  *
 4:  * 五教科（国語・数学・理科・公民・英語）の平均（四捨五入済み）を求める
 5:  *
 6:  * @author Manabu Sugiura
 7:  * @version $Id: ScoreAdministratorSample.java,v 1.9 2003/05/08 10:11:04 gackt Exp $
 8:  */
 9: public class ScoreAdministratorSample {
10:
11:   public static void main(String[] args) {
```

```
12:     ScoreAdministratorSample scoreAdministratorSample =
13:       new ScoreAdministratorSample();
14:     scoreAdministratorSample.main();
15:   }
16:
17:   void main() {
18:
19:     //各教科の点数を設定する
20:     int japanese = 49; //国語
21:     int mathematics = 73; //数学
22:     int science = 100; //理科
23:     int civics = 45; //公民
24:     int english = 25; //英語
25:
26:     //５教科の平均を求める
27:     double total = japanese + mathematics + science + civics + english;
28:     //５教科の合計点を求める
29:     double average = total / 5.0; //平均を計算する
30:
31:     //平均を四捨五入する
32:     average = average * 10;
33:     if ((average % 10) >= 5) { //１の位が５以上なら
34:       average = average + 10; //繰り上げる
35:     }
36:     int result = (int) (average / 10);
37:
38:     //四捨五入した平均を表示する
39:     System.out.println(result);
40:   }
41: }
```

1.2.4.2 コメントの種類

コメントは書く位置によって役割が異なります。次の 3 種類を目安にすると分かりやすいでしょう。

タイトル（見出し）コメント

タイトルコメントは、プログラムの一番初めに書くコメントで、そのプログラムが何を目的としたものであるかを簡潔に表記します。タイトルコメントの要素として以下のものが挙げられます。

標題 (タイトルとプログラム目的)　プログラムの名称と、その機能を簡潔に表したもの。

ファイル名　そのプログラム自身のファイル名。

作者　プログラムを作成した人の名前。設計した人が別の人ならば、設計者の名前を書いておきます。後からそのプログラムを変更する人が、誰に尋ねれば情報を得られるのかを簡単に知ることができます。複数の人でプログラムを書くときに役に立ちます。

バージョンアップの履歴　改造や修正によっていったん出来上がったプログラムが変更されると、そのプログラムのバージョンが変わることになります。このバージョンアップが行われた日付と、行った人の名前を記入しておくと、誰によっていつ変更されたものなのかが分かります。

ブロックコメント

対象ブロックが何を目的としたプログラムなのかを記述するコメントです。（ブロックが始まる前に書く）ブロック全体に通用します。(図 1.4)

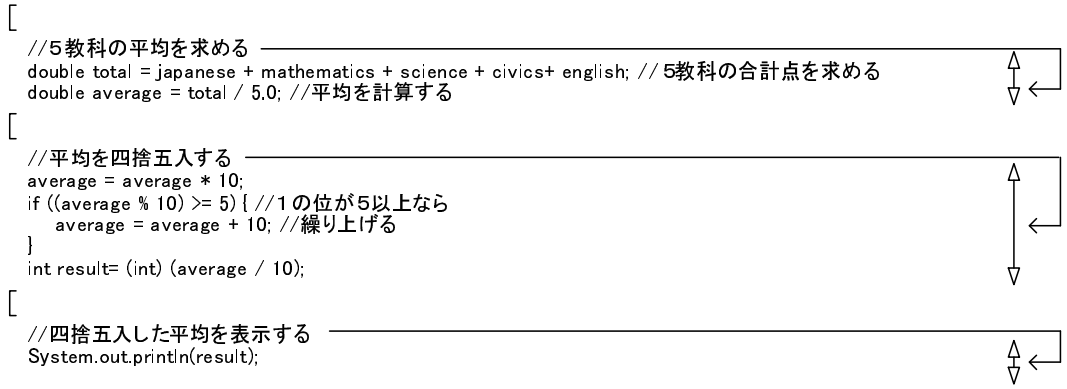


図 1.4: ブロックコメントの有効範囲

行コメント

その行は何を目的として書かれているのかを行末に書くコメントです。その行だけに通用する小目的を書きます。

1.3　プログラムの目的と階層構造

1.3.1　目的の階層構造

前節で学習したのは、コメントにはプログラムの手段ではなく、目的を書くことが重要ということでした。先ほど紹介してきた「成績を管理する（５教科の平均を求め四捨五入をする)」というプログラムの目的を、さらに細かく見ていきましょう。

各ブロックに記載されているコメントを抜き出すと次のようにまとめられます。

５教科の平均（四捨五入済み）を求める
- 各教科の点数を設定する
- ５教科の平均を求める
- 平均を四捨五入する
- 四捨五入した平均を表示する

このように小さなプログラムでも目的を階層化し、構造を整理することができます。このように整理されたプログラムは、どのようなプログラムであるかが、プログラムを読めない人でさえ一目瞭然です。

1.3.2　HCP チャートによるプログラムの設計

プログラムのソースコードを実際に書き始める前に、これから作るプログラムの構造を分析、整理しまとめることを、プログラムの設計といいます。

この「人にやさしいプログラミング哲学」の前半では、NTT で開発された、「HCP チャート (Hierarchical and Compact Description Chart：階層化コンパクトチャート)」を用いて設計を行います。

HCP チャートの特徴は、プログラムの目的の階層構造を図で表現できることです。また、プログラムでよく出てくる繰り返しの処理や、条件によって処理を変更するなどといった、プログラムの処理の構造を簡単に書くことができる記号も用意されています。

例として、成績管理プログラムにおける HCP チャートを図 1.5 に示します。

HCP チャートによる設計において一番重要なのは、プログラムの目的をしっかり捕らえ、明確な日本語として記述して、それを階層構造に整理することです。記法は多少適当でかまいませんので、日本語とその構造をしっかり記述してください。

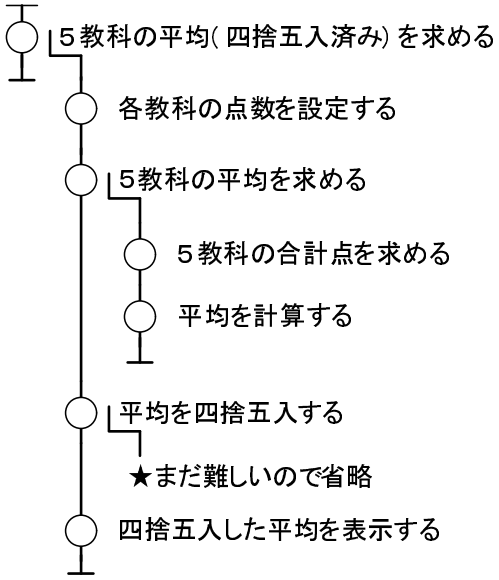


図 1.5: 成績管理プログラムの HCP チャート

1.3.3　HCP チャートを反映したソースコードの記述

HCP チャート (図 1.5) と、最終的に人にやさしいソースコードになったソースコード (リスト 7) を見比べてみましょう。

HCP チャートを描くとプログラムの目的の階層構造がきれいに整理されます。あとはこの階層構造にあわせて、それぞれの目的を実現するようにソースコードを書けば、自然に人にやさしいプログラムを書くことができます。

設計が終ったら、まずソースコードにコメントとして HCP チャートに書かれた目的を書いてしまいましょう。**ソースコードはコメントから書く**のが人にやさしいプログラマーへの道です。

1.4　人にやさしいユーザインターフェイス

1.4.1　ユーザインターフェイスの基本

ソフトウエアは人が使うために、そして人の助けをするために開発されます。本節では、いかに” 使う人” にやさしいプログラムを書くかということを考えていきたいと思います。

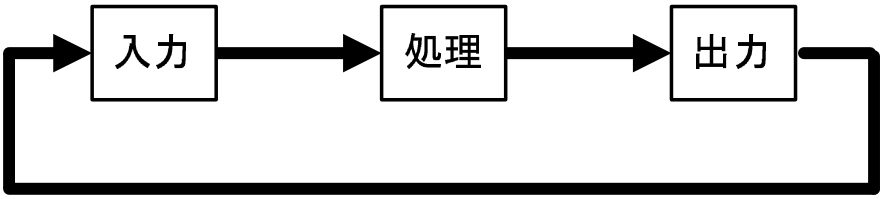


図 1.6: ユーザインターフェイスの基本

ユーザインターフェイスを備えたプログラムの基本構造は、図 1.6 に示すように、入力、処理、出力の繰り返しです。入力はユーザの入力、出力は処理結果の表示と考えるとより分かりやすいと思います。この構造は、今日広く使われている GUI でも、本テキストで扱っていく CUI でも変わりません。

この基本構造のうち、プログラムの使い手（ユーザ）がプログラムに接する部分、つまりコンピュータの処理以外の入力、出力の個所のことをユーザインターフェイスと呼び、プログラムの使い勝手のことを表します。本テキストでは、ここからほとんど全ての例題（–Application と名前がついているもの）について、使いやすい、人にやさしいユーザインターフェイスを備えたプログラムを示すようにしています。

本テキストではじめてのユーザインターフェイスを備えたプログラムの HCP を図 1.7 に、ソースをリスト 8 に示します。

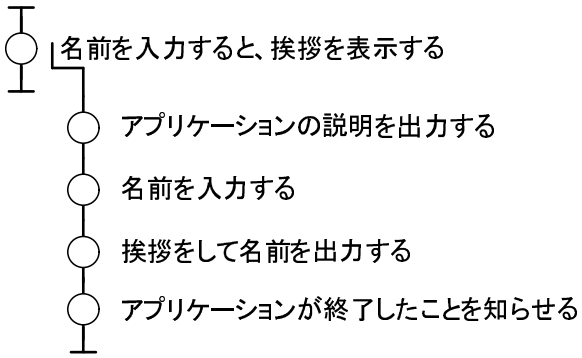


図 1.7: 名前を入力すると挨拶を表示するアプリケーションの HCP チャート

リスト 8: 名前を入力すると挨拶を表示するアプリケーション

```
 1: /**
 2:  * 自分の名前をキーボードから入力し、挨拶を表示するアプリケーション
 3:  *
 4:  * @author Manabu Sugiura
 5:  * @version $Id: GreetingApplication.java,v 1.3 2003/05/04 16:25:14 macchan Exp $
 6:  */
 7: public class GreetingApplication {
 8:
 9:   public static void main(String[] args) {
10:     GreetingApplication greetingApplication = new GreetingApplication();
11:     greetingApplication.main();
12:   }
13:
14:   void main() {
15:
16:     String name; //表示する名前
17:
18:     //アプリケーションの説明を出力する
19:     System.out.println("名前を入力すると、挨拶を表示します。");
20:
21:     //名前を入力する
22:     System.out.println("名前を入力してください");
23:     System.out.print(">>");
24:     System.out.flush();
25:     name = Input.getString();
26:
27:     //挨拶をして名前を出力する
28:     System.out.println("こんにちは。" + name + "さん。");
29:     System.out.println("これから長いお付き合いになりますね。");
30:
31:     //アプリケーションが終了したことを知らせる
32:     System.out.println("アプリケーションが終了しました。");
33:   }
34: }
```

人にやさしいユーザインターフェイスにするために、ここでは 2 つのことに気をつけています。

タイトル プログラムを開始したときに、そのプログラムがいったい何をするものであるかが分かると、使う人は、次に何をしてよいか、どのように使ったらよいかが分かりやすくなります。そのため、プログラムの開始と同時に、それがどのようなプログラムなのかということをタイトルとして表示します。これが「人にやさしいプログラム」です。

プロンプト プログラムが実行して、何か入力して欲しいと（あなたが勝手に）思っていても、あなた以外の他人には、何を入力してほしいか分かりません。このため、何を入力するかというのを表示してあげなければなりません。これが「プロの道」です。

1.4.2　繰り返し

先ほどの名前と挨拶を表示するアプリケーションでは、一度実行されただけで終了するものでした。これではいろいろな名前を入れて試してみたいときはいちいちプログラムを起動しないといけません。

これは電卓を使う際、一つ計算をするたびに機械が終了してしまい、次の計算をするときはまたスイッチを入れ直さないといけないことと同じことです。

この不便を解消するために繰り返しを用いてプログラムを書き直したのがリスト 9 です。

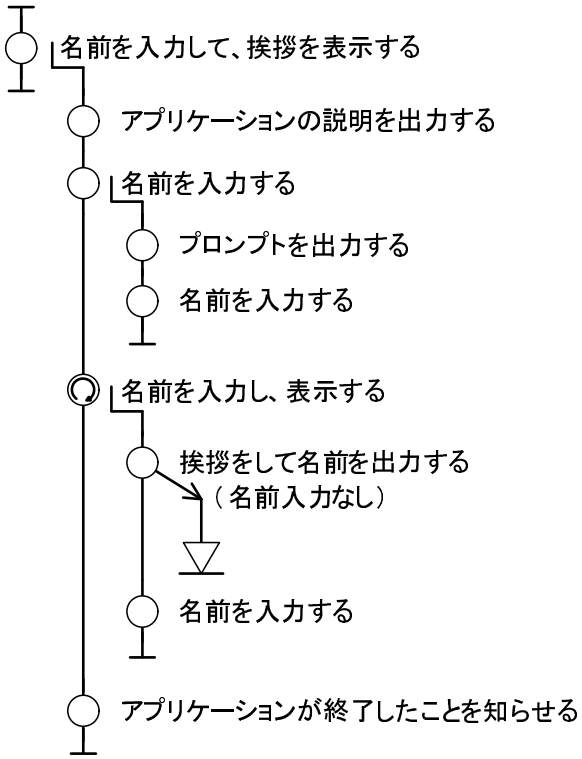


図 1.8: 名前を入力すると挨拶を表示するアプリケーションの HCP チャート

リスト 9: 名前を入力すると挨拶を表示するアプリケーション（繰り返しを導入）

```
 1: /**
 2:  * 名前をキーボードから入力し、挨拶を表示することを繰り返し行うアプリケーション
 3:  *
 4:  * なにも入力せずに改行するとアプリケーションを終了する
 5:  *
 6:  * @author Manabu Sugiura
 7:  * @version $Id: GreetingApplication.java,v 1.5 2003/05/04 20:30:23 gackt Exp $
 8:  */
 9: public class GreetingApplication {
10:
11:   public static void main(String[] args) {
12:     GreetingApplication greetingApplication = new GreetingApplication();
13:     greetingApplication.main();
14:   }
15:
16:   void main() {
17:
18:     String name; //表示する名前
19:
20:     //アプリケーションの説明を出力する
21:     System.out.println("名前をすると、挨拶を表示します。名前が空文字の場合は終了します。");
22:
23:     //名前を入力する
24:     System.out.println("名前を入力してください");
25:     System.out.print(">>");
26:     System.out.flush();
27:     name = Input.getString();
28:
29:     //名前を入力し、表示する
30:     while (name.length() != 0) {
31:
32:       //挨拶をして名前を出力する
33:       System.out.println("こんにちは。" + name + "さん。");
34:       System.out.println("これから長いお付き合いになりますね。");
35:
36:       //次の名前を入力する
37:       System.out.println("名前を入力してください");
38:       System.out.print(">>");
39:       System.out.flush();
40:       name = Input.getString();
41:     }
42:
43:     //アプリケーションが終了したことを知らせる
44:     System.out.println("アプリケーションが終了しました。");
45:   }
46: }
```

1.4.2.1　while 文

while 文は繰り返しを行う場合に用います。
while 文の書式を次に示します。

```
while ( [継続条件] ) {
  （繰り返す処理）
}
```

継続条件を調べて、合致すれば、中括弧内に書かれた処理を行い、また継続条件を調べるということを繰り返し、継続条件が合致しなくなるまで、処理を繰り返します。この制御の流れをフローチャートで示したものが、図 1.9 です。また、HCP チャートで書くと図 1.10 のようになります。

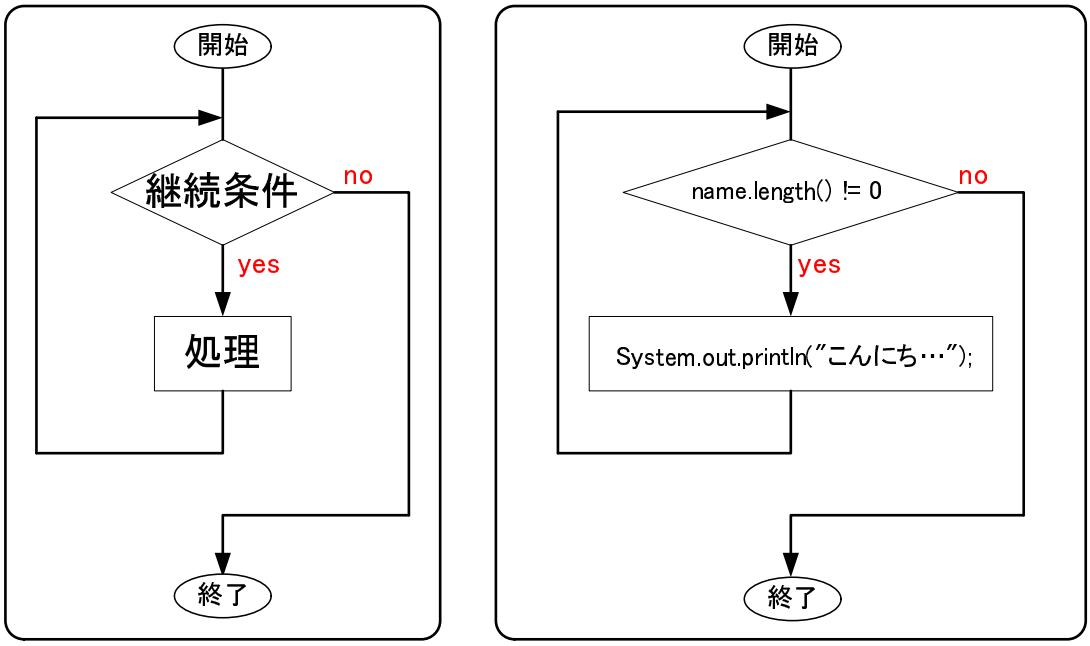


図 1.9: while 文のフローチャート

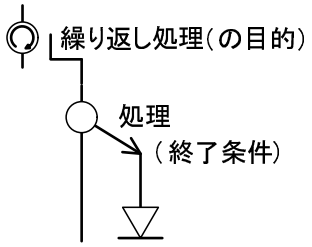


図 1.10: 繰り返しの HCP チャート

1.4.3 不正な入力の処理

ユーザは必ずしもプログラムを作った人が期待するような入力だけをするとは限りません。ユーザの不正な入力などによってプログラムに予期しない命令が入力されることがあります。

人にやさしいプログラムは、そのような場合のことも考慮されている必要があります。最悪でも、止まってしまうようなプログラムは避けたいものですね。

割り算を行うアプリケーション (リスト 10) を見てみましょう。割り算では割る数に 0 が入ってしまうと計算ができません。このプログラムでは、これを防ぐために次のような工夫がされています。

> 2 番目の数字の入力プロンプトで、0 以外の数字を入れるように予めユーザに注意させている。
> もし割る数に 0 が入力された場合は、計算を行わず、ユーザに 0 では割り算ができないことを伝える。

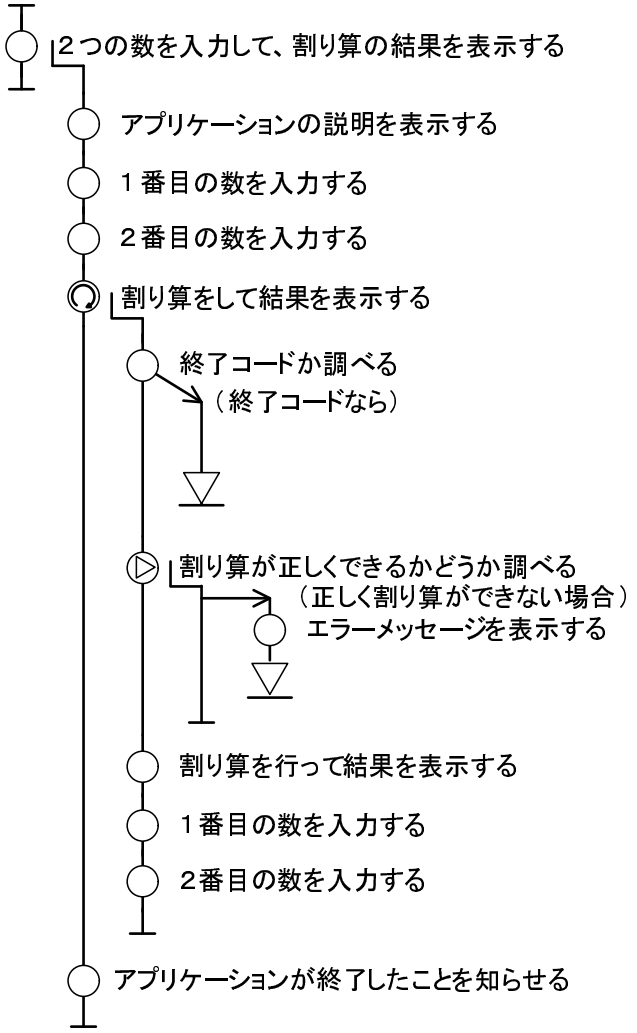


図 1.11: 割り算を行うプログラムの HCP チャート

リスト 10: 割り算を行うアプリケーション

```
 1: /**
 2:  * 割り算計算機アプリケーション
 3:  *
 4:  * ２つの数を入力すると、割り算の結果を出力する
 5:  * 入力が両方０の場合、アプリケーションを終了する
 6:  * （エラーチェックバージョン）
 7:  *
 8:  * @author Manabu Sugiura
 9:  * @version $Id: DivideTwoNumberApplication.java,v 1.10 2003/05/04 20:29:51 gackt Exp $
10:  */
11: public class DivideTwoNumberApplication {
12:
13:   public static void main(String[] args) {
14:     DivideTwoNumberApplication divideTwoNumberApplication =
15:       new DivideTwoNumberApplication();
```

```
16:     divideTwoNumberApplication.main();
17:   }
18:
19:   void main() {
20:
21:     int firstNumber; //１番目の整数
22:     int secondNumber; //２番目の整数
23:     int result; //割り算の結果
24:
25:     //アプリケーションの説明をする
26:     System.out.println("２つの数の割り算の結果を求めます。");
27:     System.out.println("（少数点以下は計算できません）");
28:
29:     //１番目の数を入力する
30:     System.out.println("１つ目の整数を入力してください");
31:     System.out.print(">>");
32:     System.out.flush();
33:     firstNumber = Input.getInt();
34:
35:     //２番目の数を入力する
36:     System.out.println("２つ目の整数を入力してください（割る数は０以外）");
37:     System.out.print(">>");
38:     System.out.flush();
39:     secondNumber = Input.getInt();
40:
41:     //割り算をして結果を表示する
42:     while (firstNumber != 0 || secondNumber != 0) {
43:
44:       //割り算が正しくできるかどうか調べる
45:       if (secondNumber == 0) { //正しく割り算ができない場合
46:         System.out.println("エラー：０で割り算はできません!");
47:         break; //アプリケーションを終了する
48:       }
49:
50:       //割り算を行って結果を表示する
51:       result = firstNumber / secondNumber;
52:       System.out.println("２つの数の商は" + result + "です");
53:
54:       //１番目の数を入力する
55:       System.out.println("１つ目の整数を入力してください");
56:       System.out.print(">>");
57:       System.out.flush();
58:       firstNumber = Input.getInt();
59:
60:       //２番目の数を入力する
61:       System.out.println("２つ目の整数を入力してください（割る数は０以外）");
62:       System.out.print(">>");
63:       System.out.flush();
64:       secondNumber = Input.getInt();
65:     }
66:
67:     //アプリケーションが終了したことを知らせる
68:     System.out.println("アプリケーションが終了しました。");
69:   }
```

```
70: }
```

1.5 練習問題

練習問題 1

自分用のタイトルコメントのテンプレートを作ってください。練習問題 2 からそれを使ってください。

練習問題 2

健康を管理するプログラム（HealthCareSample.java）を「人にやさしいプログラム」にしてください。

練習問題 3

足し算計算機アプリケーション（AddTwoNumberApplication.java）に、非常に大きな数字や、数字以外の文字などを入力し、どうなるか調べてみましょう。

第 2 章

変数を使った抽象化 (1)

この章で学習すること

数字に意味を与え、変数（定数）として定義できる
変数と値の関係が説明できる
変数の宣言・代入・評価の過程を変数表（表モデル）を描いて説明できる
式が与えられたとき、どのように値が導かれるかを評価の概念を用いて説明できる

2.1 変数と値

2.1.1 データをどう扱うか

この章ではじゃんけんができるゲーム（アプリケーション）を作ろうと思います。交互に互いの手を入力しあって、じゃんけんができるようなアプリケーションです。HCP チャートを使って大まかな仕様を考えてみました。(図 2.1)

目的の階層構造は固まりましたが、問題となるのは、データ（グー、チョキ、パーの手）をどうコンピュータで扱えるようにするかです。コンピュータでは、そのままこの概念を表すことができないので、ここでは、1,2,3 の数値で表すという方法で実現することを考えましょう。(図 2.2)

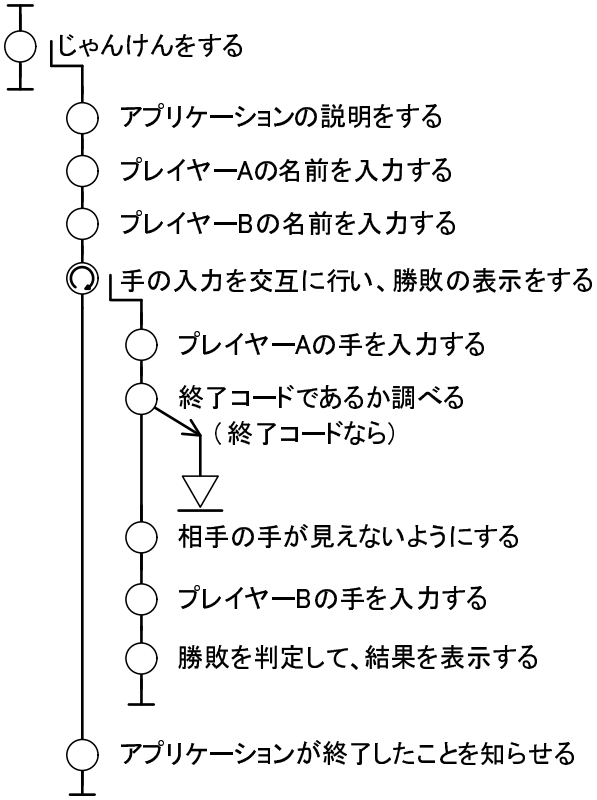


図 2.1: じゃんけんアプリケーションの HCP チャート

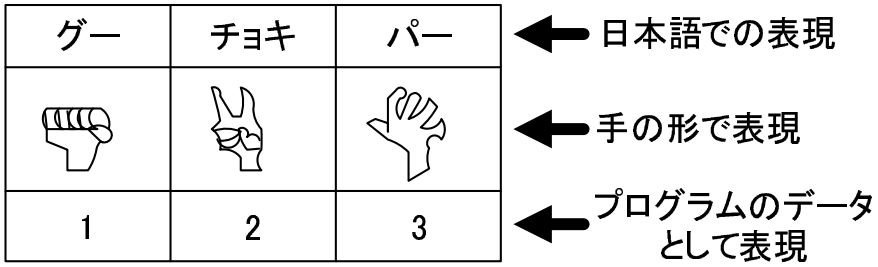


図 2.2: グー、チョキ、パーの表し方

このように、新しいプログラムを作成するためには、データをコンピュータが扱う形式に変換するという作業が必要になります。この作業のことを「データ構造」の設計といいます。

これに対して、第 1 章で学習した、手順（目的）の階層構造を設計することを「アルゴリズム」の設計といいます。「アルゴリズムとデータ構造」の 2 つが、プログラムを構成する重要な要素です。本章では、データ構造の設計を議論していきましょう。

じゃんけんアプリケーションでは、グー、チョキ、パーをそれぞれ 1,2,3 で表現することにして、HCP チャートにデータの表を書き足しました（図 2.3)。実装したものが、リスト 11 です。

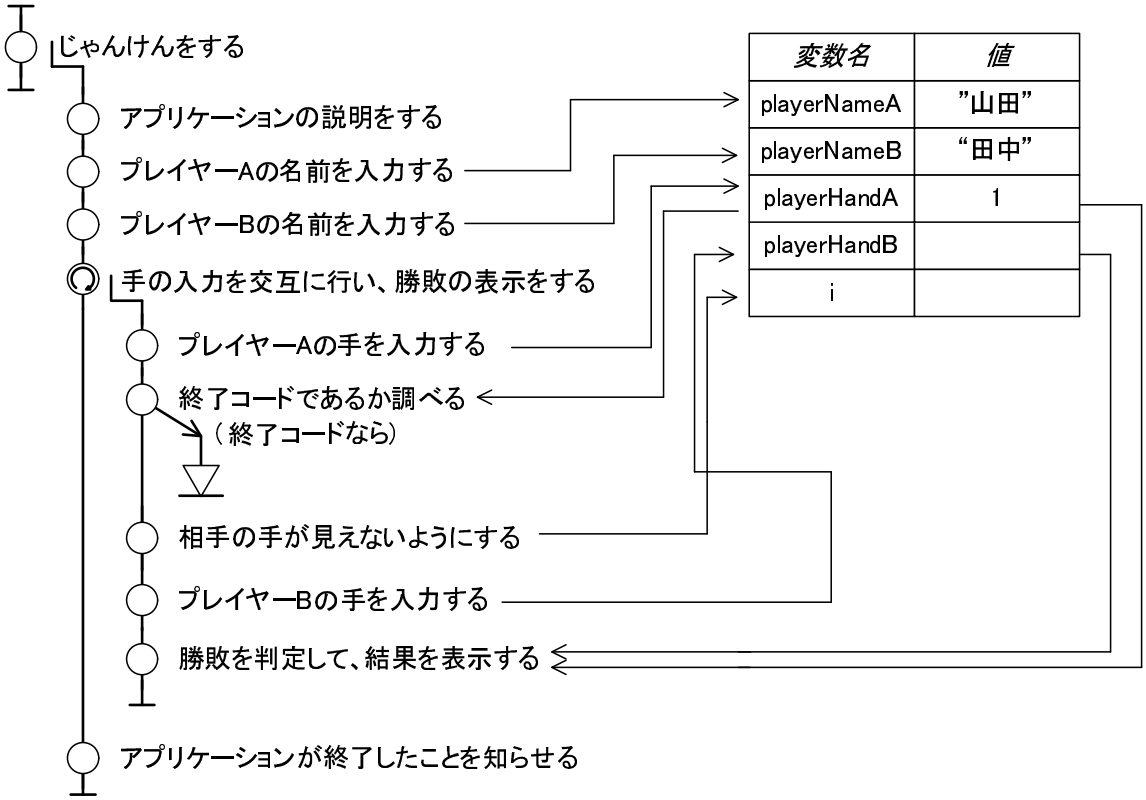


図 2.3: じゃんけんアプリケーションの HCP チャート（データあり）

リスト 11: じゃんけんアプリケーション

```
 1: /**
 2:  * じゃんけんアプリケーション
 3:  *
 4:  * 二人のプレイヤーがじゃんけんの手を交互に入力すると、勝敗を調べて、
 5:  * 勝者の名前を表示する
 6:  *
 7:  * @author macchan
 8:  * @version $Id: JankenApplication.java,v 1.11 2003/05/04 20:54:13 gackt Exp $
 9:  */
10: public class JankenApplication {
11:
12:   public static void main(String[] args) {
13:     JankenApplication jankenApplication = new JankenApplication();
14:     jankenApplication.main();
15:   }
16:
17:   void main() {
18:
19:     String playerNameA; //プレイヤー A の名前
20:     String playerNameB; //プレイヤー B の名前
21:     int playerHandA; //プレイヤー A の出した手
22:     int playerHandB; //プレイヤー B の出した手
23:     int i; //ループ用
24:
```

```
25:     //アプリケーションの説明をする
26:     System.out.println("じゃんけん♪アプリケーション");
27:     System.out.println("（プレイヤー A の手に 0 を入力すると終了します）");
28:
29:     //プレイヤー A の名前を入力する
30:     System.out.println("プレイヤー A の名前を入力してください");
31:     System.out.print(">>");
32:     System.out.flush();
33:     playerNameA = Input.getString();
34:
35:     //プレイヤー B の名前を入力する
36:     System.out.println("プレイヤー B の名前を入力してください");
37:     System.out.print(">>");
38:     System.out.flush();
39:     playerNameB = Input.getString();
40:
41:     //手の入力を交互に行い、勝敗の表示をする
42:     while (true) {
43:
44:       //プレイヤー A の手を入力する
45:       System.out.println(playerNameA + "さんの手を入力してください");
46:       System.out.println("1. グー, 2. チョキ, 3. パー (0. 終了)");
47:       System.out.print(">>");
48:       System.out.flush();
49:       playerHandA = Input.getInt();
50:
51:       //終了コードであるか調べる
52:       if (playerHandA == 0) { //終了コードなら
53:         break; //アプリケーションを終了する
54:       }
55:
56:       //相手の手が見えないようにする
57:       i = 0;
58:       while (i < 100) {
59:         System.out.println();
60:         i = i + 1;
61:       }
62:
63:       //プレイヤー B の手を入力する
64:       System.out.println(playerNameB + "さんの手を入力してください");
65:       System.out.println("1. グー, 2. チョキ, 3. パー");
66:       System.out.print(">>");
67:       System.out.flush();
68:       playerHandB = Input.getInt();
69:
70:       //勝敗を判定して、結果を表示する
71:       if (playerHandA == 1 && playerHandB == 2) { //グー VS チョキ
72:         System.out.println(playerNameA + "さんのかち");
73:       } else if (playerHandA == 1 && playerHandB == 3) { //グー VS パー
74:         System.out.println(playerNameB + "さんのかち");
75:       } else if (playerHandA == 2 && playerHandB == 1) { //チョキ VS グー
76:         System.out.println(playerNameB + "さんのかち");
77:       } else if (playerHandA == 2 && playerHandB == 3) { //チョキ VS パー
78:         System.out.println(playerNameA + "さんのかち");
```

```
79:       } else if (playerHandA == 3 && playerHandB == 1) { //パー VS グー
80:         System.out.println(playerNameA + "さんのかち");
81:       } else if (playerHandA == 3 && playerHandB == 2) { //パー VS チョキ
82:         System.out.println(playerNameB + "さんのかち");
83:       } else if (playerHandA == playerHandB) { //あいこ
84:         System.out.println("あいこでした");
85:       } else { //不正な入力の処理
86:         System.out.println("不正な入力です");
87:       }
88:     }
89:
90:     //アプリケーションが終了したことを知らせる
91:     System.out.println("アプリケーションが終了しました。");
92:   }
93: }
```

2.1.2 変数と値

リスト 11 の playerHandA と playerHandB に注目してみます。これにグーやチョキやパーといったじゃんけんの「手」のデータが入ります。このような具体的なデータのことを「値」と呼びます。値には、1,2,3 といった整数、0.05 のような実数、”山田”のような文字などの種類があります。

これに対して、具体的な値を格納する場所のことを「変数」といいます。具体的な値 (例えば 1) に対して、意味のある名前 (例えば playerHand) が結び付いていることを想像してみましょう。

本テキストでは、その様子を図 2.4 のような表を使って表していきます。一般的に「変数表」と呼ばれるものですが、このテキストでは「表モデル [1]」と呼んでいます。

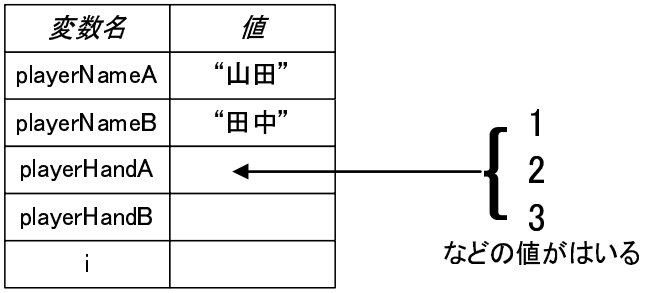


図 2.4: 変数と値の表モデル

[1] 先に書かれたデータ構造付き HCP チャート (図 2.3) では、データの表現として、表モデルを使用しています。これは、HCP チャートの正しい書きかたではありませんが、利便性のため、このテキストでは全ての HCP チャート上のデータを表モデルで記述しています。

2.1.3　プログラムの実行と変数の評価

プログラムが実行されるとデータがどのように移り変わっていくのかを、表モデルを利用して見ていきましょう。

2.1.3.1　変数の宣言

変数は宣言しないと使えません。変数の宣言は「型　変数名」という書式で記述します。変数を宣言すると、表モデルの左側の列に変数名が書き込まれます（図 2.5）。

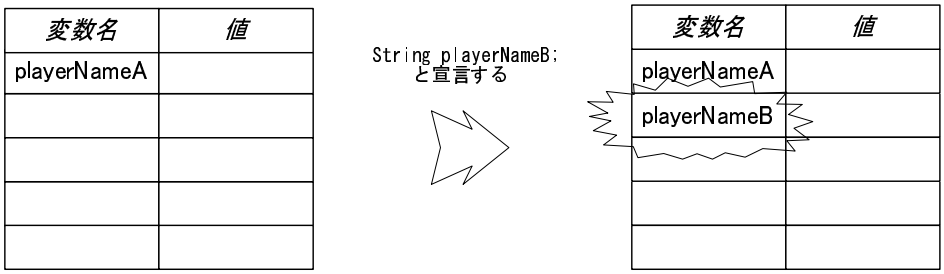


図 2.5: 宣言の表モデル

2.1.3.2　代入

宣言した変数には具体的な値を対応付けて使うことになります。この具体的な値を対応付けることを代入といいます。代入は「=」演算子を使います。数学の「=」とは違って、右から左に代入するという意味になります。「じゃんけんアプリケーション」では一人目のプレイヤーの名前が入力し終わると、図 2.6 のような表の状態になっています。

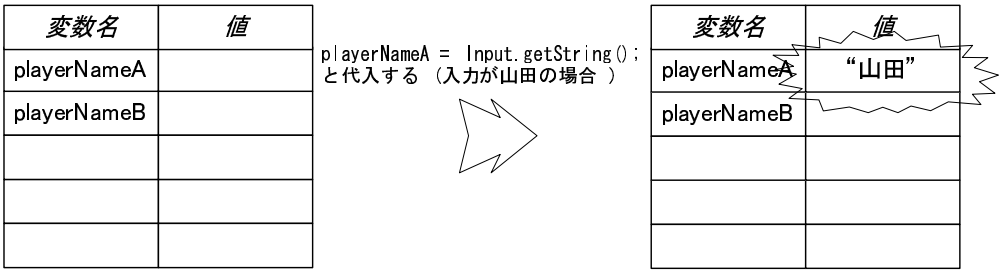


図 2.6: 代入の表モデル

2.1.3.3 変数の評価

宣言した変数は使用される際に、代入されている具体的な値との置き換えが起こります（図 2.7）。この置き換えのことを変数の「評価」といいます。

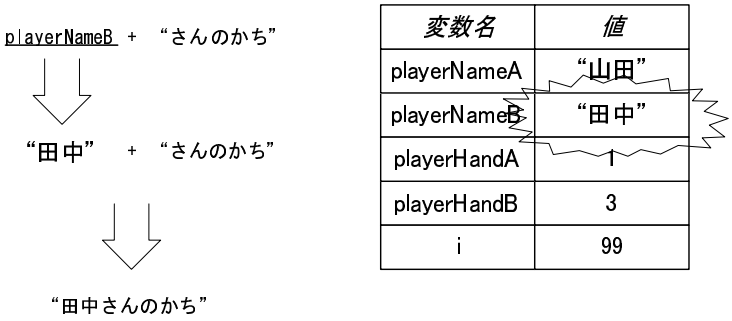


図 2.7: 評価の表モデル

2.1.3.4 プログラムの実行と表の変化

プログラムの実行に伴って、図 2.8 のように、表が変化していきます。

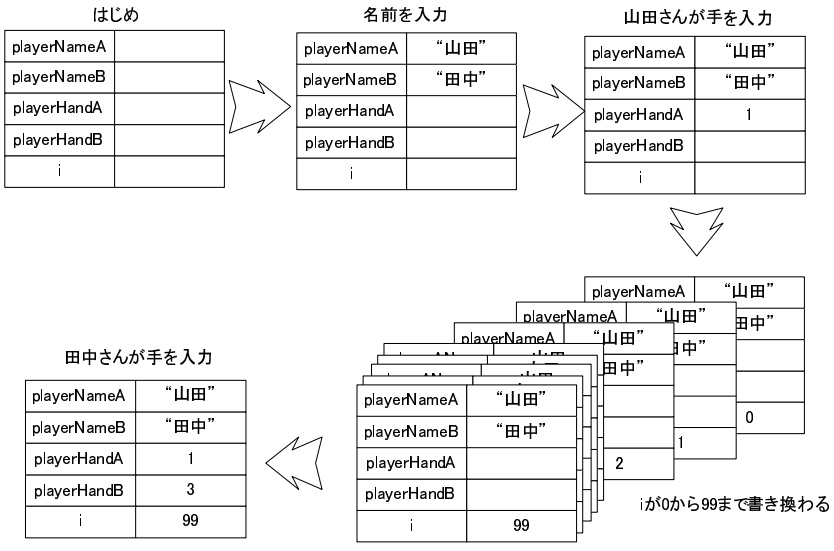


図 2.8: プログラムの実行と表モデルの変化

2.2　式と評価

2.2.1　式と値

評価はこれまで説明したように変数に限って行われるものではなく、「i + 1」のような記述の実行でも行われ、評価されて値に置き換えられます（図 2.9)。このようにプログラム上で評価を行う対象になりうるものを**式**と呼びます。

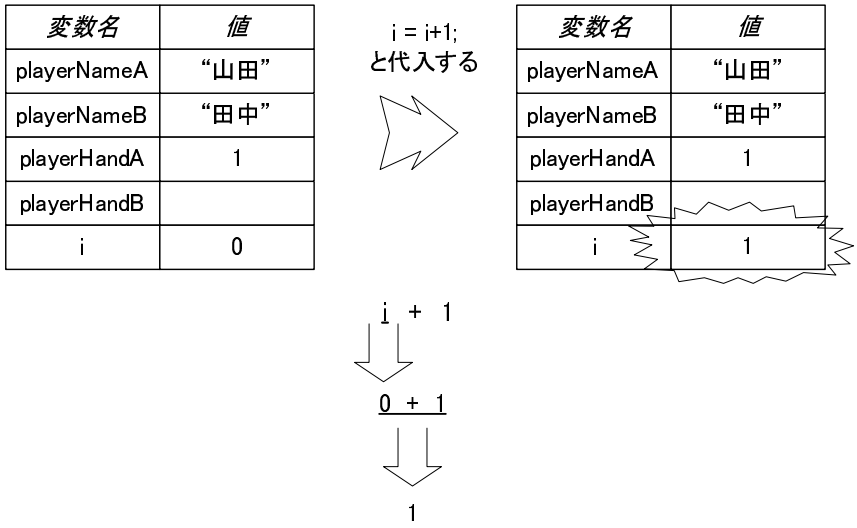


図 2.9: 式の評価の表モデル

2.2.2　条件式の評価

2.2.2.1　if 文

「じゃんけんアプリケーション」では、プレイヤーがそれぞれ出した手によって、じゃんけんの勝敗を判別して、結果を表示させることが必要でした。このように、条件に応じてプログラムが処理する内容を変えたいというときは if 文を用います。

```
if ( [条件式] ) {
  文　//条件式を評価して成立 (true) のときはここを実行
} else {
  文　//条件式を評価して不成立 (false) のときはここを実行
}
```

if 文は上のような文法で、条件式が成立すればその後の文を実行し、不成立の時は、else 節を実行します。else 節は省略可能で、その場合不成立の時には何も実行しません。

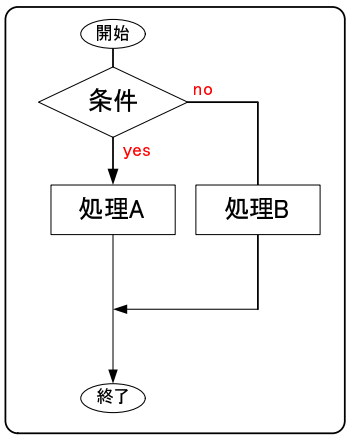


図 2.10: if 文のフローチャート

制御の流れは図 2.10 のようになります。

3 つ以上の条件で場合分けしたい時は下記のように、if else 文を使ってさらに条件分岐を増やします。

```
if ( [条件式 1] ) {
  文　//条件式 1 を評価して成立（true）のときはここを実行
} else if( [条件式 2] ){
  文　//条件式 2 を評価して成立（true）のときはここを実行
} else {
  文　//条件式 1 も条件式 2 を評価しても不成立（false）のときはここを実行
}
```

HCP チャートで条件分岐を表現する際は図 2.11 のように記述します。今後、プログラムで条件分岐を使うようになるので、覚えておくと便利です。

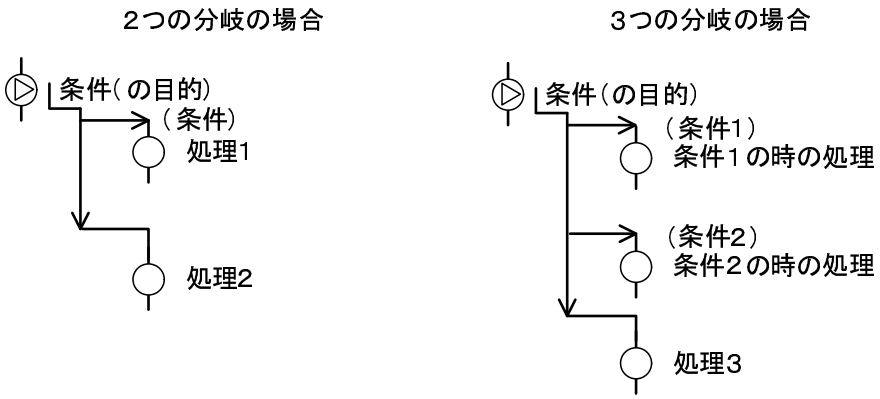


図 2.11: if 文の HCP

2.2.2.2　真偽値型と演算子

真偽値型

Java での条件式は、すべて真偽値型 (boolean) で判定されます。真偽値型は評価すると true か false のどちらかの値が入るデータの型です。while 文の継続条件も if 文の条件文も、評価すれば値は必ず true か false の値になります。

比較演算子と論理演算子

表 2.1: Java における数値の比較演算子（A,B は同じ型の変数とする）

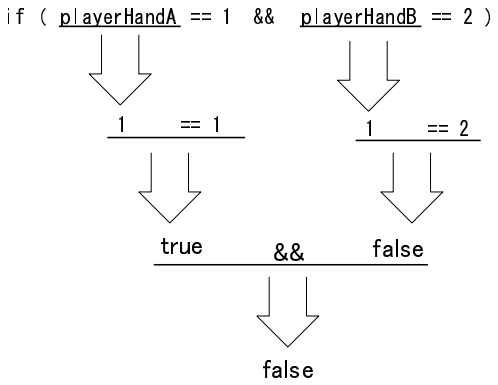
演算子	表記例	評価
==	A==B	A と B が同じ値の時 true
!=	A!=B	A と B が同じ値でない時 true
>	A>B	A が B より大きい時 true
<	A<B	A が B より小さい時 true
>=	A>=B	A が B 以上の時 true
<=	A<=B	A が B 以下の時 true

表 2.2: Java における論理演算子（A,B は真偽値型）

演算子	表記例	意味
&&	A&&B	A と B 両方とも true の時 true
||	A||B	A もしくは B が true の時 true

2.2.2.3　条件式の評価

if 文を考える際にも「評価」の考え方を使えば簡単です。例えば、if(playerHandA==1 && playerHandB==2) という式は、図 2.12 のように評価が行われます。これまでの式の評価とまったく同じです。



playerNameA	“山田”
playerNameB	“田中”
playerHandA	1
playerHandA	2
i	99

図 2.12: 条件式の評価

2.3　変数の型

2.3.1　変数の型

2.1.3.1 節で変数を宣言する時には、変数の「型」を指定する必要があると説明しました。これは、Java では、変数の種類によって、それぞれ記憶できるデータの種類も決まっているからです。

例えば、整数型 (int) なら-1, 0, 1, 2, 3 のといった整数の値が代入できますし、文字列型 (String) なら”山田”, ”田中”のといった文字列の値が代入できます (図 2.13)。型が違う値を代入することは出来ません。。

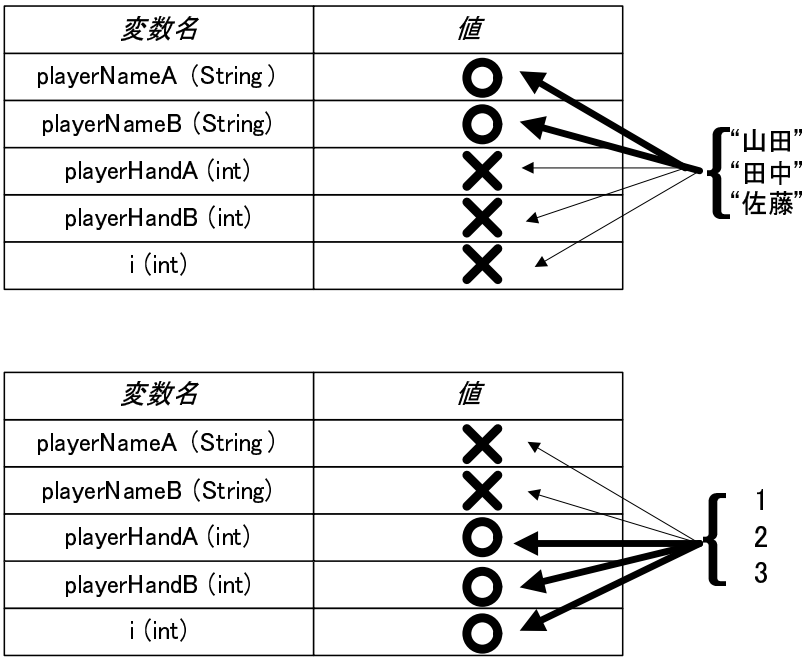


図 2.13: 変数の型と代入できる値

Java には 8 種類の基本データ型 [2]が用意されています。

実は、今までよく使ってきた String 型は、基本データ型ではありません。Java では、基本データ型のほかに、基本データ型をいくつか組み合わせた型が存在します。（String 型はその一種です。）組み合わせ型については、オブジェクト指向編で詳しく説明します。

2.3.2　型の変換

変数は型によって扱いがことなるので、型の違う変数同士を足したり比較したりすることはできません。正札を出力するアプリケーション (リスト 12) のように、浮動小数点型

[2] 基本データ型の一覧は「Java 言語プログラミングレッスン (上)」の付録 H を参照してください。

の消費税率や割引率と整数値型の定価を掛け合わせて税込価格や価格を計算したい場合は
型の変換（キャスト）を行います。

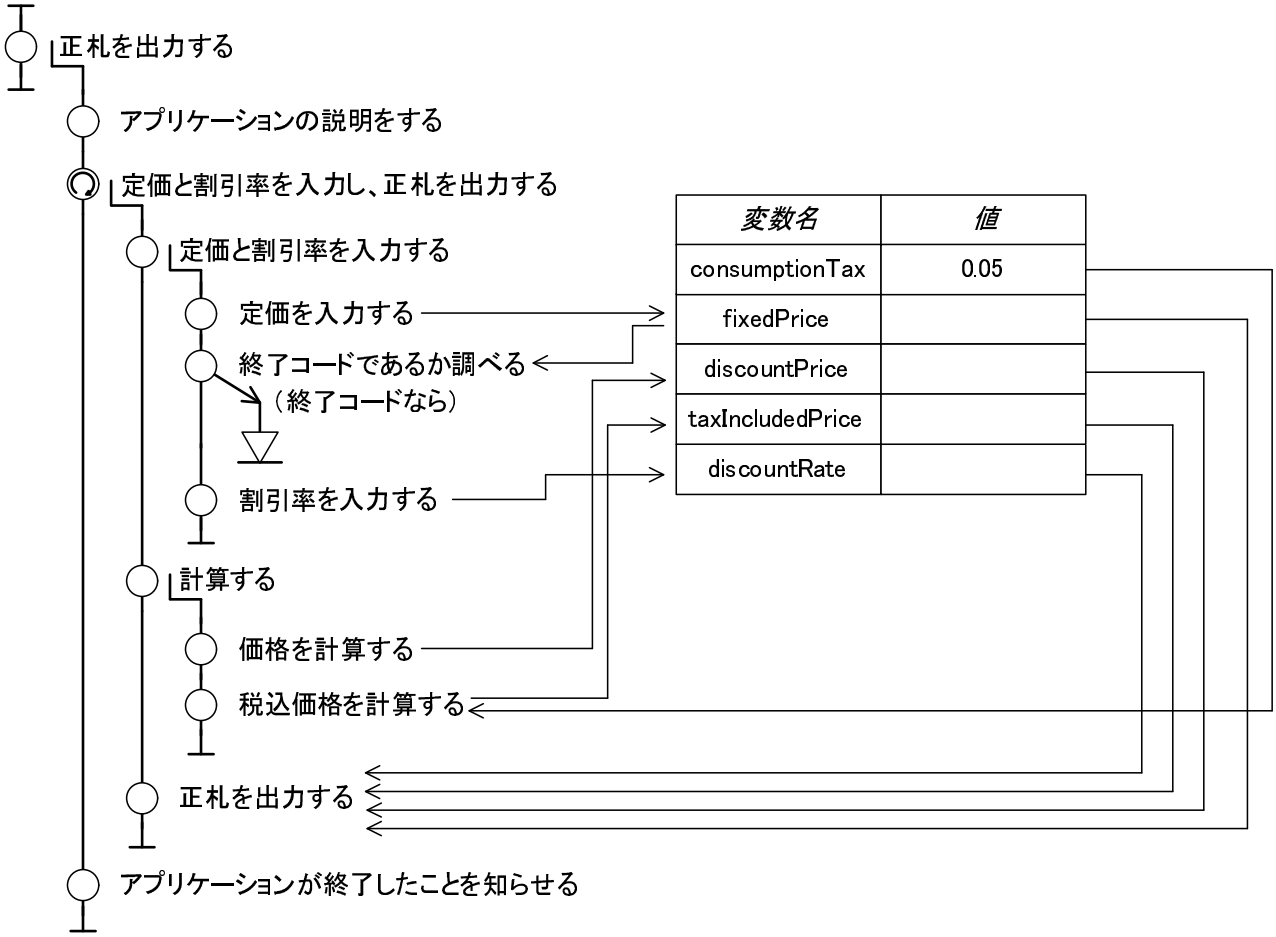


図 2.14: 正札を出力するアプリケーションの HCP チャート

リスト 12: 正札を出力するアプリケーション

```
 1: /**
 2:  * 正札を出力するアプリケーション
 3:  *
 4:  * 商品の定価と割引率を入力すると、定価、割引率、価格、税込み価格を書き込んだ
 5:  * 下記のような正札（値札のようなもの）を出力する。
 6:  * 定価が0ならばアプリケーションを終了させる。
 7:  * （if 文と break を用いる）
 8:  *
 9:  * --------------------
10:  * 定価:6000 円
11:  * --------------------
12:  * 価格:3600 円（40%OFF）
13:  * --------------------
14:  * 税込価格：3780 円
```

```
15:  * --------------------
16:  *
17:  * @author Manabu Sugiura
18:  * @version $Id: PriceTagApplication.java,v 1.8 2003/05/06 22:36:14 gackt Exp $
19:  */
20: public class PriceTagApplication {
21:
22:   public static void main(String[] args) {
23:     PriceTagApplication priceTagApplication = new PriceTagApplication();
24:     priceTagApplication.main();
25:   }
26:
27:   void main() {
28:
29:     double consumptionTax = 0.05; // 消費税率
30:     int fixedPrice; // 定価
31:     int discountPrice; //（割引後の）価格
32:     int taxIncludedPrice; // 税込み価格
33:     int discountRate; //割引率
34:
35:     //アプリケーションの説明をする
36:     System.out.println("正札を出力します");
37:     System.out.println("※定価の入力が０の場合に終了します");
38:
39:     //定価と割引率を入力し、正札を出力する
40:     while (true) {
41:
42:       //定価を入力する
43:       System.out.println("定価を入力してください");
44:       System.out.print(">>");
45:       System.out.flush();
46:       fixedPrice = Input.getInt();
47:
48:       //終了コードか調べる
49:       if (fixedPrice == 0) { //終了コードなら（定価が０なら）
50:         System.out.println("アプリケーションを終了します");
51:         break; //終了する
52:       }
53:
54:       //割引率を入力する
55:       System.out.println("割引率（%）を入力してください");
56:       System.out.print(">>");
57:       System.out.flush();
58:       discountRate = Input.getInt();
59:
60:       //価格を計算する
61:       discountPrice =
62:         (int) (fixedPrice * (1 - (double) discountRate / 100));
63:
64:       //税込価格を計算する
65:       taxIncludedPrice = (int) (discountPrice * (1 + consumptionTax));
66:
67:       //正札を出力する
68:       System.out.println("--------------------");
```

```
69:       System.out.println("定価:" + fixedPrice + "円");
70:       System.out.println("--------------------");
71:       System.out.println(
72:         "価格:" + discountPrice + "円（" + discountRate + "%OFF）");
73:       System.out.println("--------------------");
74:       System.out.println("税込価格：" + taxIncludedPrice + "円");
75:       System.out.println("--------------------");
76:     }
77:
78:     //アプリケーションが終了したことを知らせる
79:     System.out.println("アプリケーションが終了しました。");
80:   }
81: }
```

Java Tips

キャスト

キャストは変換したい値に対して、変換先の型を () で囲ったものを前につけます。

break 文

while 文による繰り返し処理を中断したいときには break 文を用います。

2.4　プログラムの意味と変数

2.4.1　定数

章の冒頭で紹介した「じゃんけんアプリケーション」（リスト 11）は、1,2,3 という数字でじゃんけんの手である、グー、チョキ、パーを意味していました。しかし、最初に紹介したソースコードは、そのような数字の意味をコメントで補っていました。

1,2,3 という数字にグー、チョキ、パーという意味がある数字であるからには、それに名前を付けることで、よりプログラムの意味が明確になります（図 2.15）。

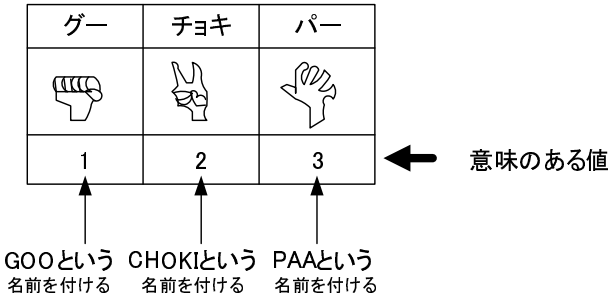


図 2.15: 意味のある値に名前を付ける

意味のある値に名前を付けたじゃんけんアプリケーションをリスト 13 に示します。

リスト 13: じゃんけんアプリケーション (定数をつけた)

```
 1: /**
 2:  * じゃんけんアプリケーション
 3:  *
 4:  * 二人のプレイヤーがじゃんけんの手を交互に入力すると、勝敗を調べて、
 5:  * 勝者の名前を表示する
 6:  * （定数を用いた）
 7:  *
 8:  * @author macchan
 9:  * @version $Id: JankenApplication.java,v 1.5 2003/05/04 20:54:13 gackt Exp $
10:  */
11: public class JankenApplication {
12:
13:   public static void main(String[] args) {
14:     JankenApplication jankenApplication = new JankenApplication();
15:     jankenApplication.main();
16:   }
17:
18:   void main() {
19:
20:     final int GOO = 1; //グーを表す定数
21:     final int CHOKI = 2; //チョキを表す定数
22:     final int PAA = 3; //パーを表す定数
```

```
23:
24:     String playerNameA; //プレイヤー A の名前
25:     String playerNameB; //プレイヤー B の名前
26:     int playerHandA; //プレイヤー A の出した手
27:     int playerHandB; //プレイヤー B の出した手
28:     int i; //ループ用
29:
30:     //アプリケーションの説明をする
31:     System.out.println("じゃんけん♪アプリケーション");
32:     System.out.println("（プレイヤー A の手に 0 を入力すると終了します）");
33:
34:     //プレイヤー A の名前を入力する
35:     System.out.println("プレイヤー A の名前を入力してください");
36:     System.out.print(">>");
37:     System.out.flush();
38:     playerNameA = Input.getString();
39:
40:     //プレイヤー B の名前を入力する
41:     System.out.println("プレイヤー B の名前を入力してください");
42:     System.out.print(">>");
43:     System.out.flush();
44:     playerNameB = Input.getString();
45:
46:     //手の入力を交互に行い、勝敗の表示をする
47:     while (true) {
48:
49:       //プレイヤー A の手を入力する
50:       System.out.println(playerNameA + "さんの手を入力してください");
51:       System.out.println("1. グー, 2. チョキ, 3. パー (0. 終了)");
52:       System.out.print(">>");
53:       System.out.flush();
54:       playerHandA = Input.getInt();
55:
56:       //終了コードであるか調べる
57:       if (playerHandA == 0) { //終了コードなら
58:         break; //アプリケーションを終了する
59:       }
60:
61:       //相手の手が見えないようにする
62:       i = 0;
63:       while (i < 100) {
64:         System.out.println();
65:         i = i + 1;
66:       }
67:
68:       //プレイヤー B の手を入力する
69:       System.out.println(playerNameB + "さんの手を入力してください");
70:       System.out.println("1. グー, 2. チョキ, 3. パー");
71:       System.out.print(">>");
72:       System.out.flush();
73:       playerHandB = Input.getInt();
74:
75:       //勝敗を判定して、結果を表示する
76:       if (playerHandA == GOO && playerHandB == CHOKI) { //グー VS チョキ
```

```
77:         System.out.println(playerNameA + "さんのかち");
78:       } else if (playerHandA == GOO && playerHandB == PAA) { //グー VS パー
79:         System.out.println(playerNameB + "さんのかち");
80:       } else if (playerHandA == CHOKI && playerHandB == GOO) { //チョキ VS グー
81:         System.out.println(playerNameB + "さんのかち");
82:       } else if (playerHandA == CHOKI && playerHandB == PAA) { //チョキ VS パー
83:         System.out.println(playerNameA + "さんのかち");
84:       } else if (playerHandA == PAA && playerHandB == GOO) { //パー VS グー
85:         System.out.println(playerNameA + "さんのかち");
86:       } else if (playerHandA == PAA && playerHandB == CHOKI) { //パー VS チョキ
87:         System.out.println(playerNameB + "さんのかち");
88:       } else if (playerHandA == playerHandB) { //あいこ
89:         System.out.println("あいこでした");
90:       } else { //不正な入力の処理
91:         System.out.println("不正な入力です");
92:       }
93:     }
94:
95:     //アプリケーションが終了したことを知らせる
96:     System.out.println("アプリケーションが終了しました。");
97:   }
98: }
```

「じゃんけんアプリケーション」ではプログラムの途中で、数字の意味が変わることがないので、一度値が設定されると、後から値を変更する（代入する）必要はありません。そのような場合は、変数ではなく、定数として宣言します。

定数を宣言する場合は以下のように行います。[3]

```
final [定数の型] [定数名] = [値];
```

定数にすると、「値に変更がなく、単一の意味を表す」ということを明確に表現できます。

[3] このテキストでは、定数の名前はすべて大文字で表記し、単語と単語の区切りは「_（アンダーバー）」で結ぶことにします。

2.5 練習問題

練習問題 1

男女の名前及び、男性、女性それぞれについて相手のことを好きか嫌いかを入力してもらい、両方の結果から相性を判断するアプリケーションを作成してください。(何度も入力を繰り返して、実行できるようにしてください)

練習問題 2

次のプログラム(リスト 14)の出力結果がどうなるかを予想して、どうしてそう思うのかという理由を表モデルを書き、評価の概念を用いて説明してください。また、問題があればどうしたら良いかを考え、どうしてその方法でよいと思うのか説明してください。

リスト 14: 2 つの数を足して、結果を表示するサンプルプログラム

```
/**
 * ２つの数を足して結果を表示するサンプルプログラム
 *
 * @author Manabu Sugiura
 * @version $Id: AddTwoNumbersSample.java,v 1.4 2003/05/04 17:16:47 macchan Exp $
 */
public class AddTwoNumbersSample {

  public static void main(String args[]) {
    AddTwoNumbersSample addTwoNumbersSample = new AddTwoNumbersSample();
    addTwoNumbersSample.main();
  }

  void main() {

    int numberA; // １番目の整数
    int numberB; // ２番目の整数

    //整数に値を設定する
    numberA = 3;
    numberB = 5;

    // 足し算の結果を表示する
    System.out.println(
      numberA + "と" + numberB + "の和は" + numberA + numberB + "です");
  }
}
```

練習問題 3*

肉屋のレシートを出力するアプリケーションを作ってください。仕様は次の通りとします。

1. 買う肉の種類の名前、その肉のグラム単価、買う量（g）を 1 つ入力すると、税込価格の値段が表示される
2. 税込価格に対して、支払うお金の金額を入力する
3. 支払い金額が税込価格より小さければ、お金が足りないというエラーを出力して、再度支払い価格の入力を求める
4. 支払い金額が税込価格より大きければ、お釣りの金額を計算する
5. レシートを出力する

   買う肉の種類の名前、その肉のグラム単価、買う量（g)、定価、税込価格、支払い金額を表示。おつりがある場合はおつりの金額も表示する

第 3 章

変数を使った抽象化 (2)

この章で学習すること

配列を使ったプログラムが書ける
情報を管理する簡単なアルゴリズムを考えて、実装できる
静的エラーと動的エラーの違いを説明できる

3.1 同種の変数をまとめて扱う

3.1.1 配列

この章では、成績管理アプリケーションを作ろうと思います。イメージ図を図 3.1 に示しました。たくさんの人たちの成績を入力すると、成績の一覧表と平均点を計算して表示してくれるという仕様です。

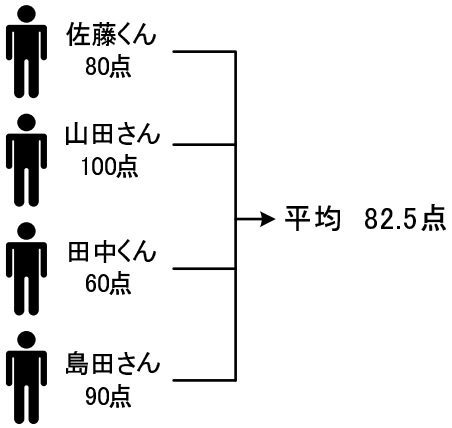


図 3.1: 成績管理アプリケーションのイメージ

成績管理アプリケーションを、HCP チャートを使って図 3.2 のように設計しました。成績を管理する人数は 10 人だと仮定すると、10 個の変数を用意する必要があります。

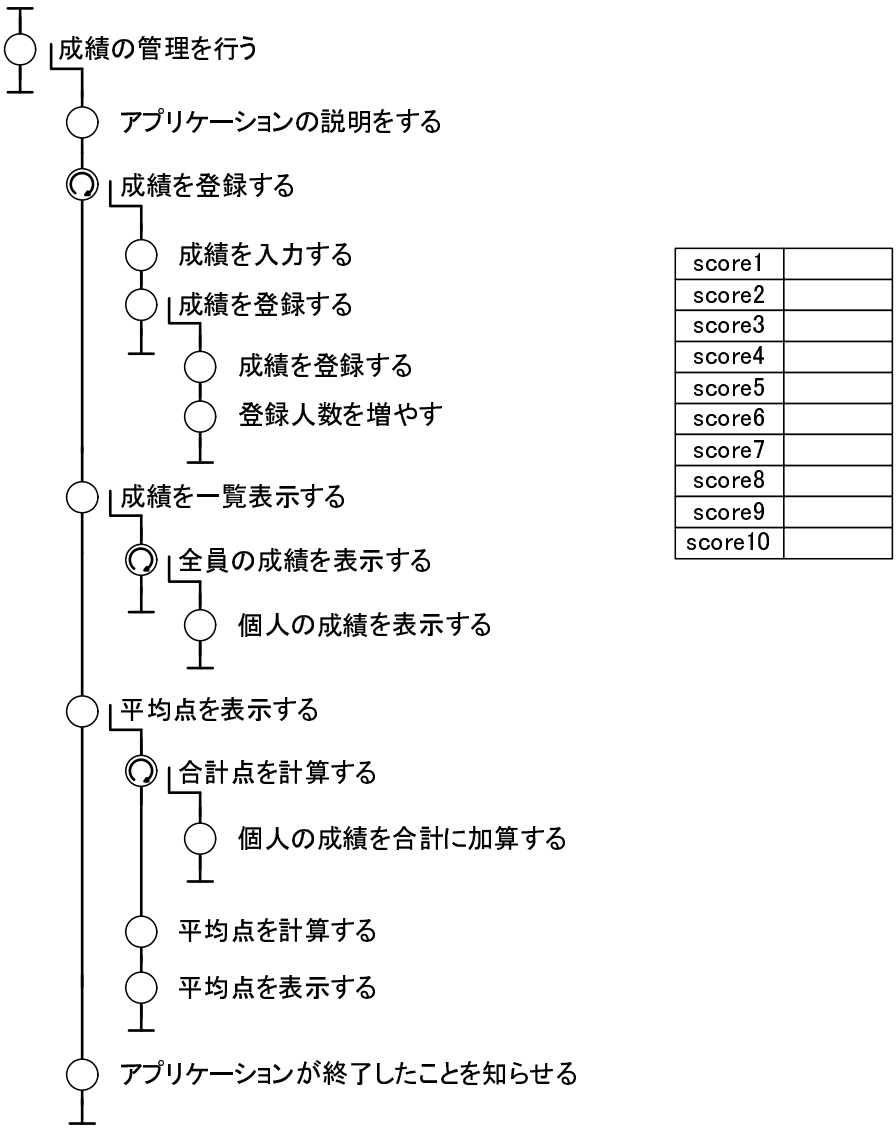


図 3.2: 成績管理アプリケーションの HCP チャート

データに注目してみましょう。これを実装するとなると、次のようなプログラムを書かなければなりません。

リスト 15: たくさんの変数の宣言

```
int score1;
int score2;
int score3;
int score4;
int score5;
int score6;
int score7;
int score8;
int score9;
```

```
int score10;
.....................
```

このように書いていくと、プログラムの量が膨大になってしまうことが想像できます。また、10 人などという規模であればまだ良いかも知れませんが、100 人の成績を管理するということになれば、プログラムを書くよりも手計算のほうが早いぐらいの作業量になるでしょう。

このような問題は、「配列」を使うと解決することが出来ます。配列を使うと、同種の変数をまとめることができます。

配列を使って書かれた成績管理アプリケーションがリスト 16 です。

リスト 16: 成績管理アプリケーション

```
 1: /**
 2:  * 成績を管理するアプリケーション
 3:  *
 4:  * 科目の成績を１０人分登録すると、以下のように成績の一覧と、平均点を出力する
 5:  *
 6:  * ★★★★★★成績一覧表★★★★★★
 7:  * 1 人目の成績:100
 8:  * 2 人目の成績:80
 9:  * 3 人目の成績:60
10:  * 4 人目の成績:90
11:  * 5 人目の成績:70
12:  * 6 人目の成績:80
13:  * 7 人目の成績:90
14:  * 8 人目の成績:50
15:  * 9 人目の成績:40
16:  * 10 人目の成績:85
17:  * ★★★★★★平均点★★★★★★
18:  * 平均点：74.5 点
19:  *
20:  * @author Manabu Sugiura
21:  * @version $Id: ScoreAdministratorApplication.java,v 1.7 2003/05/04 17:20:23 gackt Exp $
22:  */
23: public class ScoreAdministratorApplication {
24:
25:   public static void main(String[] args) {
26:     ScoreAdministratorApplication scoreAdministratorApplication =
27:       new ScoreAdministratorApplication();
28:     scoreAdministratorApplication.main();
29:   }
30:
31:   void main() {
32:
33:     final int SCORE_SIZE = 10; // 入力できる成績の数
34:
35:     int[] scores = new int[SCORE_SIZE]; //全員の成績
36:     int currentScoreIndex = 0; //現在登録されている成績の数
```

```
37:     double average; //平均点
38:     double total = 0.0; //合計点
39:     int score; //個人の成績
40:     int i; //繰り返し用
41:
42:     //アプリケーションの説明をする
43:     System.out.println("★★★★★★成績管理アプリケーション★★★★★★");
44:     System.out.println("（成績を登録すると成績一覧と平均点を出力します）");
45:
46:     //成績を登録する
47:     while (currentScoreIndex < SCORE_SIZE) {
48:
49:       //成績を入力する
50:       System.out.println("成績を入力してください");
51:       System.out.print(">>");
52:       System.out.flush();
53:       score = Input.getInt();
54:
55:       //成績を登録する
56:       scores[currentScoreIndex] = score;
57:       currentScoreIndex++; //登録人数を増やす
58:     }
59:
60:     //成績を一覧表示する
61:     i = 0; //繰り返し用の変数を初期化する
62:     System.out.println("★★★★★★成績一覧表★★★★★★");
63:     while (i < SCORE_SIZE) {
64:       System.out.println((i + 1) + "人目の成績:" + scores[i]);
65:       i++;
66:     }
67:
68:     //平均点を計算する
69:     i = 0; //繰り返し用の変数を初期化する
70:     while (i < SCORE_SIZE) { //合計点を計算する
71:       total = total + scores[i];
72:       i++;
73:     }
74:     average = total / SCORE_SIZE; //平均点を計算する
75:
76:     //平均点を表示する
77:     System.out.println("★★★★★★平均点★★★★★★");
78:     System.out.println("平均点：" + average + "点");
79:
80:     //アプリケーションが終了したことを知らせる
81:     System.out.println("アプリケーションが終了しました。");
82:   }
83: }
```

3.1.2 配列の表モデル

ここでは、今まで変数に対して使ってきた表モデルを少し拡張して、配列でも同じような考え方で使えるようにします。

3.1.2.1 配列の宣言

配列を宣言すると、図 3.3 のように表モデルの左側にタブができる [1]と考えるようにしましょう。タブには、配列の名前 [2]が入ります。配列を宣言した瞬間にはタブが表モデルの左に現れて、配列の大きさの数だけ表の領域が確保される [3]というイメージです。

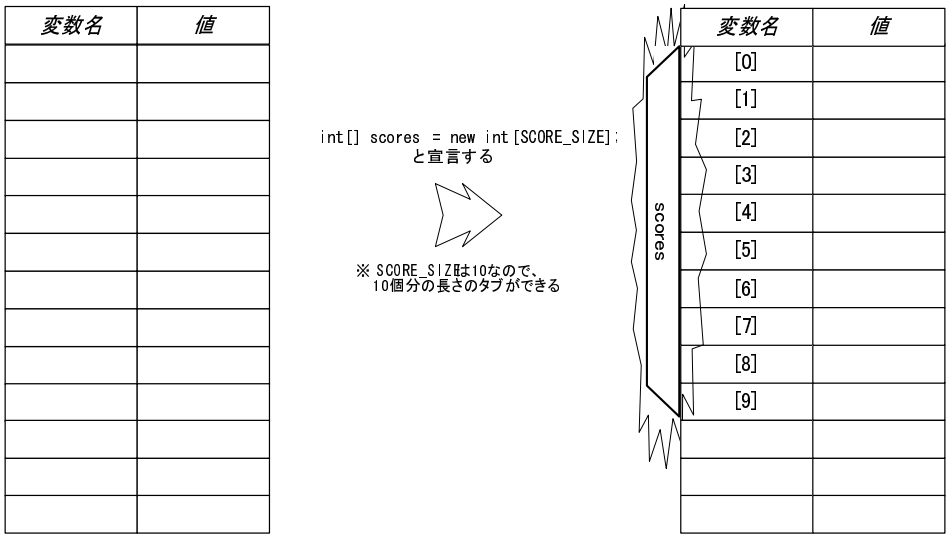


図 3.3: 配列の宣言

3.1.2.2 配列への代入

配列に値を代入する方法は、代入したい値が配列の何番目に入るのか（番地と呼ぶことにします）を指定して行います。番地を指定するには、[ ]（ブラケット）の中に番地の値を指定します。

一番最初の要素の番号は 0 なので注意してください。図 3.3 の場合、[0] から [9] の番地の 10 個の値を使うことができるようになります。

---

[1] これは変数表においては一般的な考え方ではありません。

[2] このテキストでは、配列の変数名は複数形で付けることにしています。

[3] Java の場合 int（整数型）の配列を宣言すると、初期値として 0 が代入（図では省略）されます。

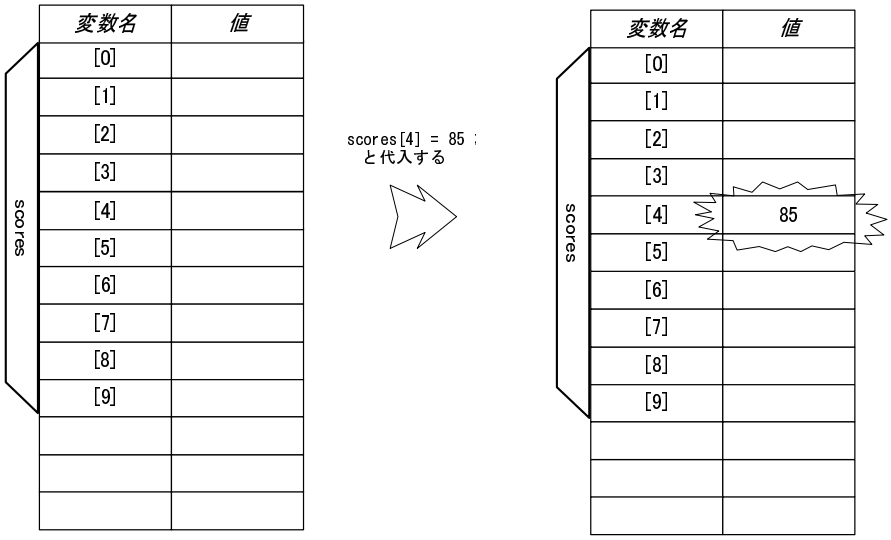


図 3.4: 配列への代入

3.1.3　配列を使ったプログラムの利点

3.1.3.1　while 文を利用する

配列の利点は、番地を指定する値を変数にすることができる点です。その変数に値を代入するときなど、繰り返し文を使うことでプログラムを簡潔に書くことができます。

配列の番地に変数を用いた場合の評価の仕組みを図 3.5 に示します。

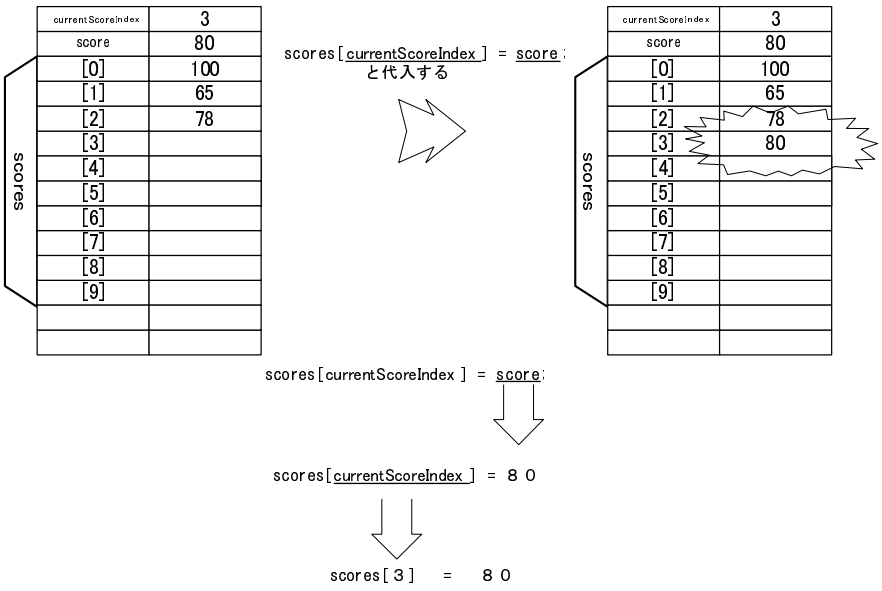


図 3.5: 番地に変数を用いた場合の代入

3.1.3.2 for 文を利用する

今までは、同じ処理を繰り返す方法として while 文を使ってきました。ここで、処理を繰り返すための新しい方法である for 文を紹介します。リスト 16 を for 文を使って書き直したものが、リスト 17 です。

リスト 17: 成績管理アプリケーション（for 文を使った場合）

```
 1: /**
 2:  * 成績を管理するアプリケーション
 3:  * （for 文を用いた）
 4:  *
 5:  * 科目の成績を１０人分登録すると、以下のように成績の一覧と、平均点を出力する
 6:  *
 7:  * ★★★★★★成績一覧表★★★★★★
 8:  * 1 人目の成績:100
 9:  * 2 人目の成績:80
10:  * 3 人目の成績:60
11:  * 4 人目の成績:90
12:  * 5 人目の成績:70
13:  * 6 人目の成績:80
14:  * 7 人目の成績:90
15:  * 8 人目の成績:50
16:  * 9 人目の成績:40
17:  * 10 人目の成績:85
18:  * ★★★★★★平均点★★★★★★
19:  * 平均点：74.5 点
20:  *
21:  * @author Manabu Sugiura
22:  * @version $Id: ScoreAdministratorApplication.java,v 1.6 2003/05/04 17:19:05 gackt Exp $
23:  */
24: public class ScoreAdministratorApplication {
25:
26:   public static void main(String[] args) {
27:     ScoreAdministratorApplication scoreAdministratorApplication =
28:       new ScoreAdministratorApplication();
29:     scoreAdministratorApplication.main();
30:   }
31:
32:   void main() {
33:
34:     final int SCORE_SIZE = 10; // 入力できる成績の数
35:
36:     int[] scores = new int[SCORE_SIZE]; //全員の成績
37:     int currentScoreIndex; //現在登録されている成績の数
38:     double average; //平均点
39:     double total = 0.0; //合計点
40:     int score; //個人の成績
41:     int i; //ループ用
42:
43:     //アプリケーションの説明をする
```

```
44:     System.out.println("★★★★★★成績管理アプリケーション★★★★★★");
45:     System.out.println("（成績を登録すると成績一覧と平均点を出力します）");
46:
47:     //成績を登録する
48:     for (currentScoreIndex = 0;
49:       currentScoreIndex < SCORE_SIZE;
50:       currentScoreIndex++) {
51:
52:       //成績を入力する
53:       System.out.println("成績を入力してください");
54:       System.out.print(">>");
55:       System.out.flush();
56:       score = Input.getInt();
57:
58:       //成績を登録する
59:       scores[currentScoreIndex] = score;
60:     }
61:
62:     //成績を一覧表示する
63:     System.out.println("★★★★★★成績一覧表★★★★★★");
64:     for (i = 0; i < SCORE_SIZE; i++) {
65:       System.out.println((i + 1) + "人目の成績:" + scores[i]);
66:     }
67:
68:     //平均点を計算する
69:     for (i = 0; i < SCORE_SIZE; i++) { //合計点を計算する
70:       total = total + scores[i];
71:     }
72:     average = total / SCORE_SIZE; //平均点を計算する
73:
74:     //平均点を表示する
75:     System.out.println("★★★★★★平均点★★★★★★");
76:     System.out.println("平均点：" + average + "点");
77:
78:     //アプリケーションが終了したことを知らせる
79:     System.out.println("アプリケーションが終了しました。");
80:   }
81: }
```

for 文の書式は以下のようになります。

```
for（[初期条件] ; [継続条件] ; [繰り返すたびに行う処理]）{

   （繰り返したい処理）

}
```

初期処理　繰り返しが始まるときに一回だけ評価される（宣言や代入などを行っても良い）
継続条件　繰り返すたびに評価され、値が真の場合に処理を繰り返す
繰り返すたびに行う処理　一回の繰り返し処理が終わるごとにこの処理を評価する

for 文の制御の流れ図を図 3.6 に示します。図の右側のフローチャートは成績管理アプリケーション（リスト 17）の成績一覧を表示する部分で利用している具体例です。

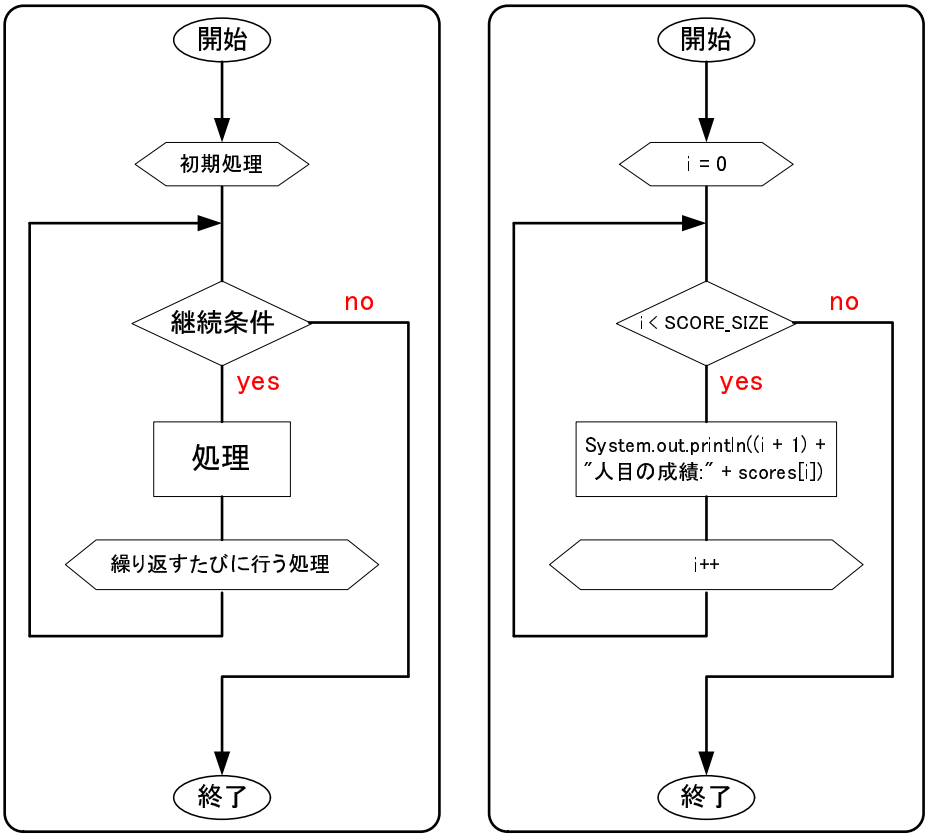


図 3.6: for 文のフローチャート

3.2　配列を利用したアルゴリズム

3.2.1　コマンド入力を受け付けるアプリケーション

次のリスト 18 はいままで出てきたアプリケーションと様子が多少違います。このアプリケーションは起動すると、メニューを表示し、ユーザーからのコマンド入力（指示）を待ちます。ユーザーがコマンドを入力すると、該当する処理を行ってからまたメニューを表示します（図 3.7 [4]）。

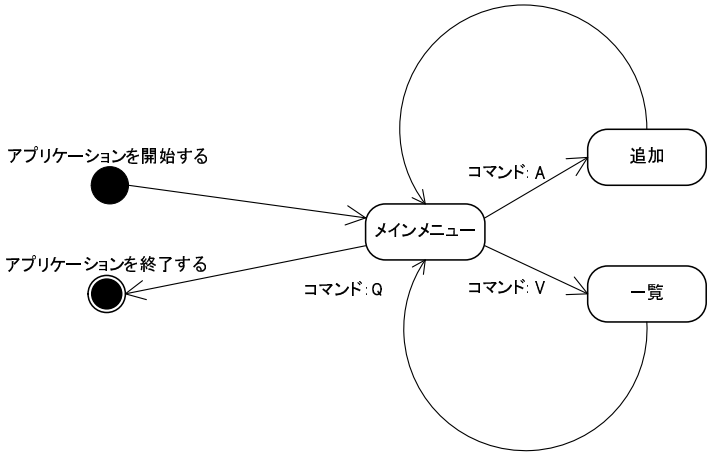


図 3.7: コマンド入力を受け付けるアプリケーションの様子

1 章でプロンプトを出力したり、プログラムの説明を出力することが、「人にやさしいインターフェース」であるという議論をしました。これに加え、コマンドの入力を受け付けるアプリケーションを作る際には、ユーザーが困らないようにコマンドの一覧を出力したり、間違えたコマンドを打った時にそれを知らせたりと、アプリケーションがユーザーに対して配慮すべき事が増えています。

また、このアプリケーションでは、名前と成績をペアにして管理できるようにしました。これで、例えば、山田さんは 100 点というように、人の名前と点数を関連させて管理できるようになり、ようやく本格的なアプリケーションとして使えるようになりました。

[4] 厳密な状態遷移図ではありません。

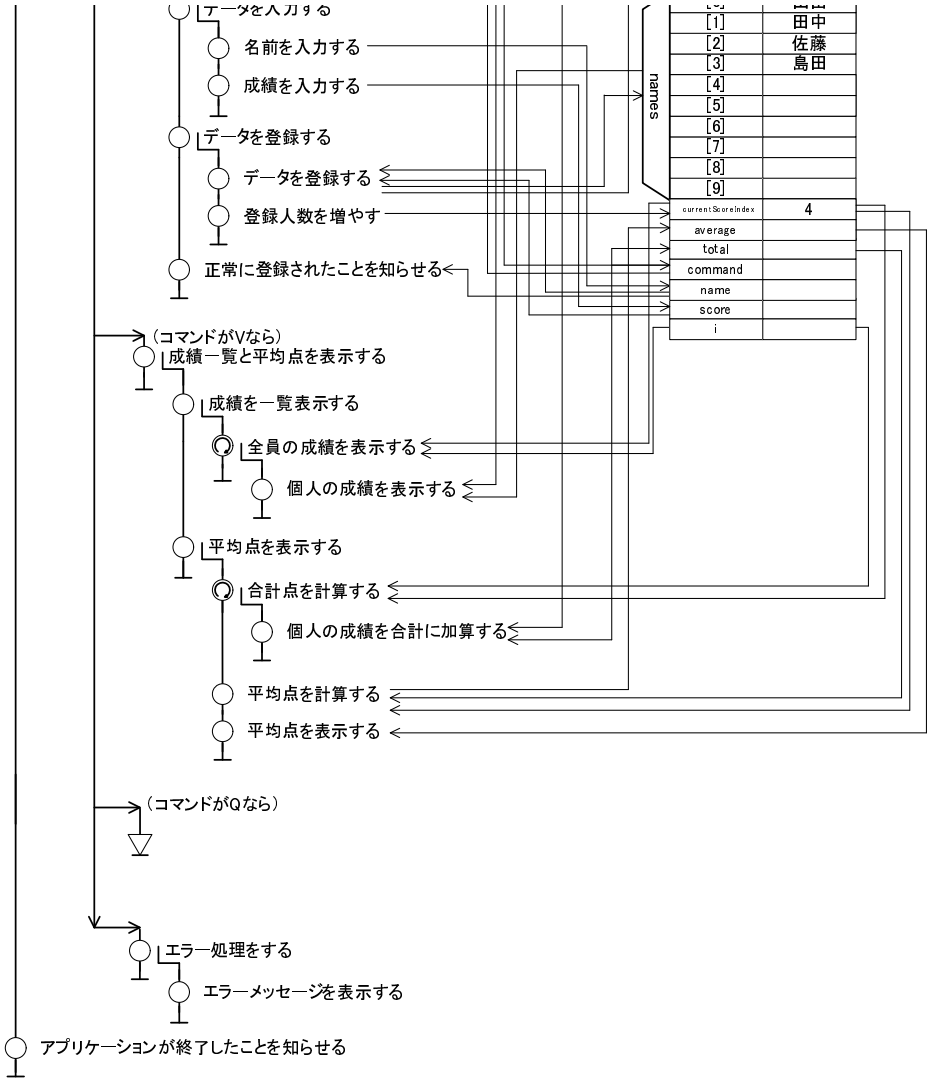


図 3.8: 成績管理アプリケーション（コマンドを入力する場合）の HCP チャート

リスト 18: 成績管理アプリケーション（コマンドを入力する場合）

```
 1: /**
 2:  * 成績を管理するアプリケーション
 3:  *
 4:  * ・名前と成績の登録（コマンド：A）
 5:  * ・成績一覧と平均表示（コマンド：V）
 6:  *
 7:  * 科目の成績を登録すると、上の動作をコマンドの選択によって行う
 8:  *
 9:  * @author Manabu Sugiura
10:  * @version $Id: ScoreAdministratorApplication.java,v 1.8 2003/05/08 17:30:38 gackt Exp $
11:  */
12: public class ScoreAdministratorApplication {
13:
14:   public static void main(String[] args) {
15:     ScoreAdministratorApplication scoreAdministratorApplication =
16:       new ScoreAdministratorApplication();
17:     scoreAdministratorApplication.main();
18:   }
19:
20:   void main() {
21:
22:     final int SCORE_SIZE = 10; // 入力できる成績の数
23:
24:     int[] scores = new int[SCORE_SIZE]; //全員の成績
25:     String[] names = new String[SCORE_SIZE]; //全員の名前
26:
27:     int currentScoreIndex = 0; //現在登録されているデータの数
28:     double average; //平均点
29:     double total; //合計点
30:     String command; //入力されたコマンド
31:     String name; //入力された個人の名前
32:     int score; // 入力された個人の成績
33:     int i; //ループ用
34:
35:     //アプリケーションの説明をする
36:     System.out.println("★★★★★★成績管理アプリケーション★★★★★★");
37:     System.out.println("（名前と成績の登録、成績一覧・平均の出力ができます。）");
38:
39:     //メニューを出力し、コマンドで指定された処理を行う
40:     while (true) {
41:
42:       //メニューを出力し、コマンドを入力する
43:       System.out.println("コマンドを入力してください");
44:       System.out.println("(A:成績の登録，V:成績一覧と平均点の表示，Q:終了)");
45:       System.out.print(">>");
46:       System.out.flush();
47:       command = Input.getString();
48:
49:       //コマンドで指定された処理を行う
50:       if (command.equals("A")) { //成績を登録する
51:
52:         //名前を入力する
53:         System.out.println("名前を入力してください");
```

```
54:         System.out.print(">>");
55:         System.out.flush();
56:         name = Input.getString();
57:
58:         //成績を入力する
59:         System.out.println("成績を入力してください");
60:         System.out.print(">>");
61:         System.out.flush();
62:         score = Input.getInt();
63:
64:         //データを登録する
65:         names[currentScoreIndex] = name;
66:         scores[currentScoreIndex] = score;
67:         currentScoreIndex++; //登録人数を増やす
68:
69:         //正常に登録されたことを知らせる
70:         System.out.println(name + "さんの成績を登録しました");
71:
72:       }
73:       else if (command.equals("V")) { //成績一覧と平均点を表示する
74:
75:         //成績を一覧表示する
76:         System.out.println("★★★★★★成績一覧表★★★★★★");
77:         for (i = 0; i < currentScoreIndex; i++) {
78:           System.out.println(names[i] + "さん:" + scores[i] + "点");
79:         }
80:
81:         //平均点を計算する
82:         total = 0.0; //合計点を初期化する
83:         for (i = 0; i < currentScoreIndex; i++) { //合計点を計算する
84:           total = total + scores[i];
85:         }
86:         average = total / currentScoreIndex; //平均点を計算する
87:
88:         //平均点を表示する
89:         System.out.println("★★★★★★平均点★★★★★★");
90:         System.out.println("平均点：" + average + "点");
91:
92:       }
93:       else if (command.equals("Q")) { //アプリケーションを終了する
94:
95:         break; //終了
96:
97:       }
98:       else { //エラー処理
99:
100:        //エラーメッセージを表示する
101:        System.out.println("そのようなコマンドはありません");
102:      }
103:    }
104:
105:    //アプリケーションが終了したことを知らせる
106:    System.out.println("アプリケーションが終了しました。");
107:  }
```

```
108: }
```

3.2.2 配列への要素の追加

リスト 18 は、追加を行える機能と、一覧、平均点の表示をする機能がありましたが、これから改造をしていくことにします。名前から成績が検索ができたり、保存されている成績の削除ができるようにしていきましょう。

配列に名前と成績を追加するためにどのような手順（アルゴリズム）を行えば良いでしょうか。追加部分のプログラムと HCP チャートを見てみましょう。

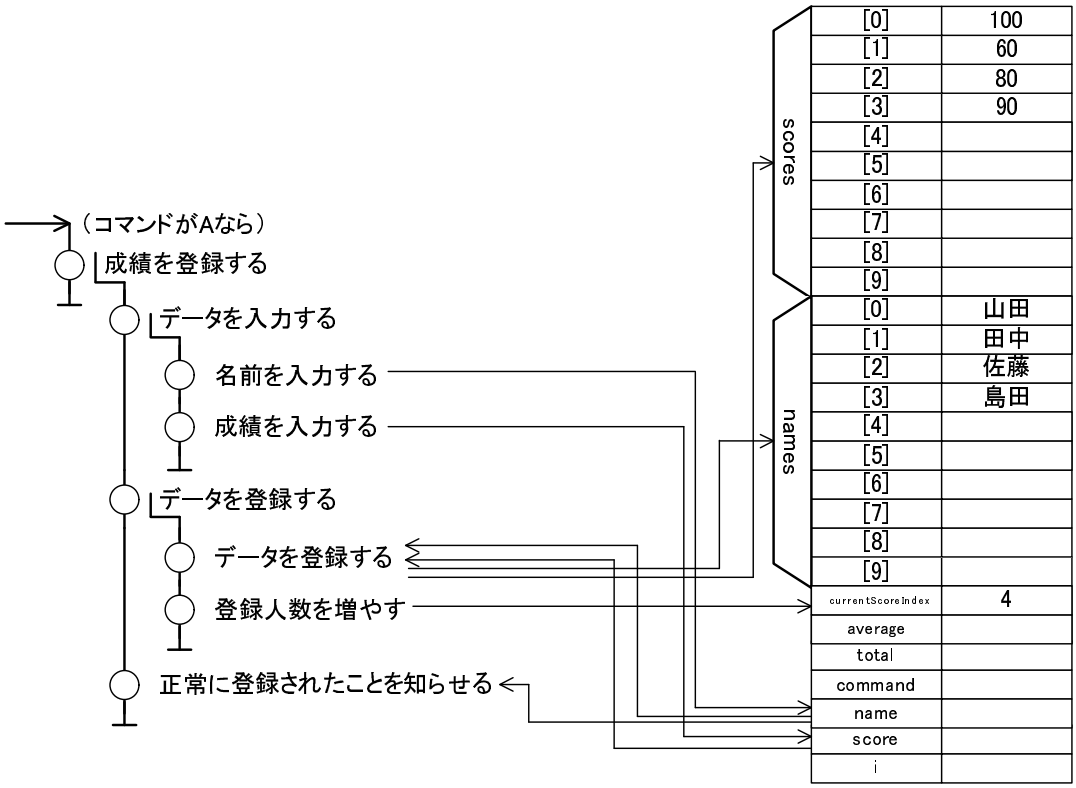


図 3.9: 成績を管理するアプリケーションの追加部分の HCP

リスト 19: 成績を管理するアプリケーションの追加部分

```
52:       if (command.equals("A")) { //成績を登録する
53:
54:         //名前を入力する
55:         System.out.println("名前を入力してください");
56:         System.out.print(">>");
57:         System.out.flush();
58:         name = Input.getString();
59:
60:         //成績を入力する
61:         System.out.println("成績を入力してください");
62:         System.out.print(">>");
```

```
63:         System.out.flush();
64:         score = Input.getInt();
65:
66:         //データを登録する
67:         names[currentScoreIndex] = name;
68:         scores[currentScoreIndex] = score;
69:         currentScoreIndex++; //登録人数を増やす
70:
71:         //正常に登録されたことを知らせる
72:         System.out.println(name + "さんの成績を登録しました");
73:
74:       }
```

追加は比較的単純な手順で実現することができます。注意しなければいけない点は保存されている全体数（人数）を増やすことが必要ということです。

3.2.3 配列にある要素の検索

このアプリケーションには、名前から成績を検索する[5]機能が搭載したいと考えています。この機能を実現するためにはどのような手順で行えばよいでしょうか。HCP チャートとリスト 20 を見てみましょう。

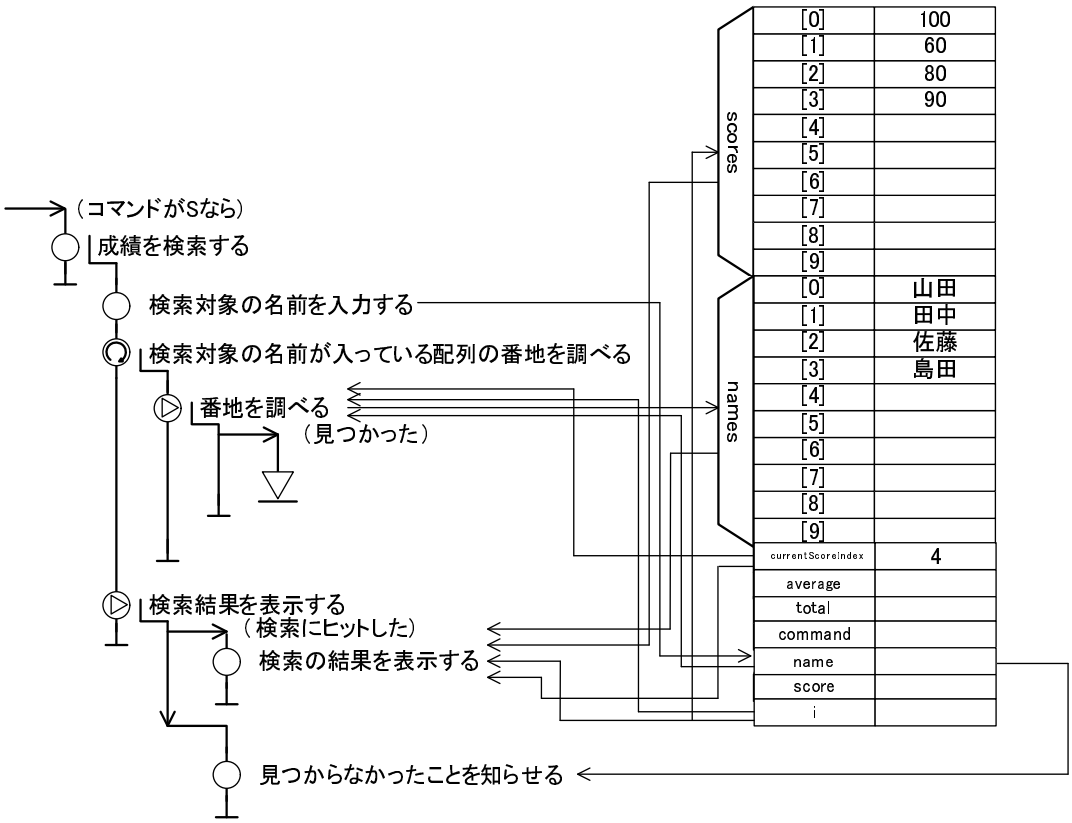


図 3.10: 成績を管理するアプリケーションの検索部分の HCP

リスト 20: 成績を管理するアプリケーションの検索部分

```
75:       else if (command.equals("S")) { //成績を検索する
76:
77:         //検索対象の名前を入力をする
78:         System.out.println("成績を検索したい名前を入力してください");
79:         System.out.print(">>");
80:         System.out.flush();
81:         name = Input.getString();
82:
83:         //検索対象の名前が入っている配列の番地を調べる
```

[5] ここで扱う検索の方法を「リニアサーチ」と言います。検索方法について詳しくは 6 章を参照してください。

```
84:         for (i = 0; i < currentScoreIndex; i++) {
85:           if (name.equals(names[i])) { //見つかった
86:             break;
87:           }
88:         }
89:
90:         //検索結果を表示する
91:         if (i < currentScoreIndex) { //検索にヒットした場合
92:           System.out.println(names[i] + "さんの成績は" + scores[i] + "点です");
93:         } else { //検索にヒットしなかった場合
94:           System.out.println(name + "さんは登録されていません");
95:         }
96:
97:       }
```

3.2.4 配列からの要素の削除

今度は一度入力した成績を消す、削除機能をつけてみてみましょう。名前からデータを削除するにはどうしたらよいでしょうか。HCP チャートとリスト 21 を見てみましょう。

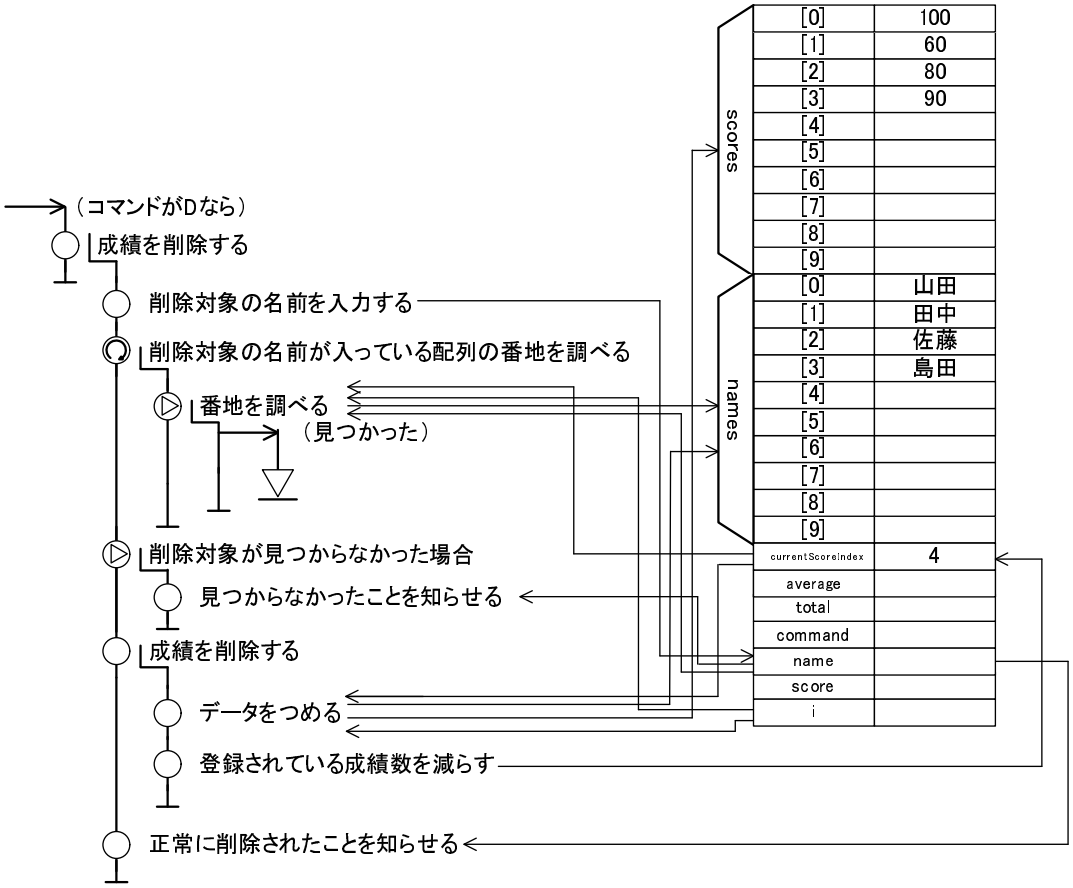


図 3.11: クラスの成績を管理するアプリケーション削除部分の HCP

リスト 21: 成績を管理するアプリケーションの削除部分

```
 98:       else if (command.equals("D")) { //成績を削除する
 99:
100:         //削除対象の名前を入力をする
101:         System.out.println("成績を削除したい名前を入力してください");
102:         System.out.print(">>");
103:         System.out.flush();
104:         name = Input.getString();
105:
106:         //削除対象の名前が入っている配列の番地を調べる
107:         for (i = 0; i < currentScoreIndex; i++) {
108:           if (name.equals(names[i])) { //見つかった
109:             break;
```

```
110:           }
111:         }
112:
113:         //削除対象が見つからなかった場合
114:         if (i >= currentScoreIndex) {
115:           System.out.println(name + "さんは登録されていません");
116:           continue;
117:         }
118:
119:         //成績を削除する
120:         for (; i < currentScoreIndex - 1; i++) { //データをつめる
121:           names[i] = names[i + 1];
122:           scores[i] = scores[i + 1];
123:         }
124:         currentScoreIndex--; //登録されている成績数を減らす
125:
126:         //正常に削除されたことを知らせる
127:         System.out.println(name + "さんの成績を削除しました");
128:
129:       }
```

成績を削除するためにはまず、削除したい人の成績と名前が配列の何番目に入っているかを調べます。これは検索と同じです。

場所を見つけたらその部分を削除するために配列の要素をつめます。

考えてみよう

削除の時、配列の要素をつめただけで、現在登録されているデータの数（currentScoreIndex）を減らさないとどうなるでしょうか？表（タブ）モデルを使って説明してみましょう。

3.2.5 成績管理アプリケーション

最後にすべての機能を搭載した成績管理アプリケーションのプログラム (リスト 22) を見てみましょう。

リスト 22: 成績管理アプリケーション（名前と成績を管理し、検索・削除機能を搭載）

```
 1: /**
 2:  * 成績を管理するアプリケーション
 3:  *
 4:  * ・名前と成績の登録（コマンド：A）
 5:  * ・名前から成績検索（コマンド：S）
 6:  * ・名前と成績の削除（コマンド：D）
 7:  * ・成績一覧と平均表示（コマンド：V）
 8:  *
 9:  * 科目の成績を登録すると、上の動作をコマンドの選択によって行う
10:  *
11:  * @author Manabu Sugiura
12:  * @version $Id: ScoreAdministratorApplication.java,v 1.7 2003/05/08 17:30:38 gackt Exp $
13:  */
14: public class ScoreAdministratorApplication {
15:
16:   public static void main(String[] args) {
17:     ScoreAdministratorApplication scoreAdministratorApplication =
18:       new ScoreAdministratorApplication();
19:     scoreAdministratorApplication.main();
20:   }
21:
22:   void main() {
23:
24:     final int SCORE_SIZE = 10; // 入力できる成績の数
25:
26:     int[] scores = new int[SCORE_SIZE]; //全員の成績
27:     String[] names = new String[SCORE_SIZE]; //全員の名前
28:
29:     int currentScoreIndex = 0; //現在登録されているデータの数
30:     double average; //平均点
31:     double total; //合計点
32:     String command; //入力されたコマンド
33:     String name; //入力された個人の名前
34:     int score; // 入力された個人の成績
35:     int i; //ループ用
36:
37:     //アプリケーションの説明をする
38:     System.out.println("★★★★★★成績管理アプリケーション★★★★★★");
39:     System.out.println("（名前と成績の登録、検索、削除、成績一覧・平均の出力ができます。）");
40:
41:     //メニューを出力し、コマンドで指定された処理を行う
42:     while (true) {
43:
44:       //メニューを出力し、コマンドを入力する
45:       System.out.println("コマンドを入力してください");
```

```
46:       System.out.println("(A:成績の登録，S:成績の検索，D:成績の削除，V:成績一覧と平均点の
表示，Q:終了)");
47:       System.out.print(">>");
48:       System.out.flush();
49:       command = Input.getString();
50:
51:       //コマンドで指定された処理を行う
52:       if (command.equals("A")) { //成績を登録する
53:
54:         //名前を入力する
55:         System.out.println("名前を入力してください");
56:         System.out.print(">>");
57:         System.out.flush();
58:         name = Input.getString();
59:
60:         //成績を入力する
61:         System.out.println("成績を入力してください");
62:         System.out.print(">>");
63:         System.out.flush();
64:         score = Input.getInt();
65:
66:         //データを登録する
67:         names[currentScoreIndex] = name;
68:         scores[currentScoreIndex] = score;
69:         currentScoreIndex++; //登録人数を増やす
70:
71:         //正常に登録されたことを知らせる
72:         System.out.println(name + "さんの成績を登録しました");
73:
74:       }
75:       else if (command.equals("S")) { //成績を検索する
76:
77:         //検索対象の名前を入力をする
78:         System.out.println("成績を検索したい名前を入力してください");
79:         System.out.print(">>");
80:         System.out.flush();
81:         name = Input.getString();
82:
83:         //検索対象の名前が入っている配列の番地を調べる
84:         for (i = 0; i < currentScoreIndex; i++) {
85:           if (name.equals(names[i])) { //見つかった
86:             break;
87:           }
88:         }
89:
90:         //検索結果を表示する
91:         if (i < currentScoreIndex) { //検索にヒットした場合
92:           System.out.println(names[i] + "さんの成績は" + scores[i] + "点です");
93:         } else { //検索にヒットしなかった場合
94:           System.out.println(name + "さんは登録されていません");
95:         }
96:
97:       }
98:       else if (command.equals("D")) { //成績を削除する
```

```
 99:
100:         //削除対象の名前を入力をする
101:         System.out.println("成績を削除したい名前を入力してください");
102:         System.out.print(">>");
103:         System.out.flush();
104:         name = Input.getString();
105:
106:         //削除対象の名前が入っている配列の番地を調べる
107:         for (i = 0; i < currentScoreIndex; i++) {
108:           if (name.equals(names[i])) { //見つかった
109:             break;
110:           }
111:         }
112:
113:         //削除対象が見つからなかった場合
114:         if (i >= currentScoreIndex) {
115:           System.out.println(name + "さんは登録されていません");
116:           continue;
117:         }
118:
119:         //成績を削除する
120:         for (; i < currentScoreIndex - 1; i++) { //データをつめる
121:           names[i] = names[i + 1];
122:           scores[i] = scores[i + 1];
123:         }
124:         currentScoreIndex--; //登録されている成績数を減らす
125:
126:         //正常に削除されたことを知らせる
127:         System.out.println(name + "さんの成績を削除しました");
128:
129:       }
130:       else if (command.equals("V")) { //成績一覧と平均点を表示する
131:
132:         //成績を一覧表示する
133:         System.out.println("★★★★★★成績一覧表★★★★★★");
134:         for (i = 0; i < currentScoreIndex; i++) {
135:           System.out.println(names[i] + "さん:" + scores[i] + "点");
136:         }
137:
138:         //平均点を計算する
139:         total = 0.0; //合計点を初期化する
140:         for (i = 0; i < currentScoreIndex; i++) { //合計点を計算する
141:           total = total + scores[i];
142:         }
143:         average = total / currentScoreIndex; //平均点を計算する
144:
145:         //平均点を表示する
146:         System.out.println("★★★★★★平均点★★★★★★");
147:         System.out.println("平均点：" + average + "点");
148:
149:       }
150:       else if (command.equals("Q")) { //アプリケーションを終了する
151:
152:         break; //終了
```

```
153:
154:       }
155:       else { //エラー処理
156:
157:         //エラーメッセージを表示する
158:         System.out.println("そのようなコマンドはありません");
159:       }
160:     }
161:
162:     //アプリケーションが終了したことを知らせる
163:     System.out.println("アプリケーションが終了しました。");
164:   }
165: }
```

3.3 静的エラーと動的エラー

3.3.1 二種類のエラー

エラーには大きく分けて二つの種類があります。

動的エラー プログラムを実行してから起こるエラーのことを動的エラーと呼びます。

静的エラー プログラムの実行前に起こるエラーのことを静的エラーと呼びます。静的エラーとはプログラムの文法間違い（コンパイルエラー）を指します。

エラーと関係のある言葉として「バグ」という言葉があります。バグとは一般的に動的なエラーが出ることと、エラーは出ないがプログラムが意図した動作をしないことの総称です。バグを修正することをデバッグと言います。またデバッグを行うツールのことをデバッガといいます。

みなさんが書いたプログラムがコンパイルができない時、それはみなさんが書いたプログラムに静的エラーがあるということです。つまりコンパイルができなかった時にも「バグがあった」というのは「バグ」という言葉の正しい使い方でなく、この場合は「コンパイルエラーがあった」という言葉の方が正しいといえます。

3.3.2 バグを解決するために

最初からバグがないプログラムを書くことは非常に難しいことです。プログラマも人間ですから、間違いはあります。

ここではバグが出た場合の解決の仕方を議論していきましょう。

3.3.2.1 例外を読む

```
java.lang.ArrayIndexOutOfBoundsException: 10
  at sp.scoread.command_add_avg_find_del_debugprint.
  ScoreAdministratorApplication.main(ScoreAdministratorApplication.java:72)
  at sp.scoread.command_add_avg_find_del_debugprint.
  ScoreAdministratorApplication.main(ScoreAdministratorApplication.java:24)
Exception in thread "main"
```

図 3.12: 例外の例

リスト 22 の成績管理アプリケーションですが、10 人以上の成績と名前を追加しようとすると、急にプログラムが止まってしまします。そして、図 3.12 のような表示が出てしまいました。

これを例外といい、動的なエラーです。

実は、このリスト 22 のプログラムにはバグがあったのです。最大登録人数を 10 人としているのにも関らず、10 人以上追加できてしまうため、存在しない配列の番地である 10（本当は 0 から 9 までの 10 個の番地しかない）に代入をしようとしてしまうので、こうした例外が起こってしまったわけです。

例外は動的エラーが発生した時にその原因を教えてくれる Java の仕組みです。これを利用しない手はありませんから、例外の内容を参考にして、発生したエラーの原因を探すようにすると、エラーが早く解決できます。

例外のうまい読み方を以下にまとめました。

例外の種類を見る　例外には種類があって、発生したエラーの種類によって例外の種類も異なります。みなさんがよく目にすると思われる例外に関する情報を載せたので参考にしてください。

例外のメッセージを見る　例外はどのようなエラーが起こったのかを示すメッセージを表示する仕組みになっています。例外の種類が出ている右横「:」をはさんで、具体的にどのようなことが起こってエラーが発生したかのメッセージが出ています。

例外が発生した行を調べる　例外が発生した行番号が例外の種類の下に表示されています。これを参考にしてプログラムを確かめると、エラーの原因が早く特定できます。

Java の例外一覧

NumberFormatException
- 文字列を数値型に変換しようとしたとき、文字列の形式が正しくない時

ArrayIndexOutOfBoundsException
- 不正な番地（負やサイズを超える）を使って配列にアクセスした時

IOExceptionException
- 入出力（ファイルなど）になんらかの問題があった時

ArithmeticException
- 算術計算で例外的条件が発生した時、たとえば、整数を 0 除算した時

3.3.2.2　デバッグプリント

プログラムが実行している間は変数の値は変化していますし、コマンドを受け付けるアプリケーションの場合、現在どのコードの部分を実行しているのかさえ分かりにくいものです。たとえバグを発見できても、こうした状態ではそのバグの原因を突き止めるのは非常に難しいといえます。そこで、バグの原因と思われるような部分に、実行中のプログラムの様子が分かるような情報を出力させ、バグの原因を探るという方法があります。これ

をデバッグプリントといいます。

成績管理アプリケーションで正しく削除が行われているかをチェックするには、リスト 23 のように、削除の前に行われる検索の様子と、削除するために配列の要素をつめる様子が分かるように表示するようにすると効果的です。これで、見つからなかったと表示されても実は削除されていた、というような分かりにくいバグも発見しやすくなります。デバッグプリントを入れたリスト 23 を実行した結果が図 3.13 です。

リスト 23: 成績を管理するアプリケーションのデバッグプリント（削除部分）

```
 99:       else if (command.equals("D")) { //成績を削除する
100:
101:         //削除対象の名前を入力をする
102:         System.out.println("成績を削除したい名前を入力してください");
103:         System.out.print(">>");
104:         System.out.flush();
105:         name = Input.getString();
106:
107:         //削除対象の名前が入っている配列の番地を調べる
108:         for (i = 0; i < currentScoreIndex; i++) {
109:
110:           //デバッグ用
111:           System.out.println(
112:             "DEBUG: names["
113:               + i
114:               + "] = "
115:               + names[i]
116:               + " 判定 "
117:               + name.equals(names[i]));
118:
119:           if (name.equals(names[i])) { //見つかった
120:             break;
121:           }
122:         }
123:
124:         //削除対象が見つからなかった場合
125:         if (i >= currentScoreIndex) {
126:           System.out.println(name + "さんは登録されていません");
127:           continue;
128:         }
129:
130:         //成績を削除する
131:         for (; i < currentScoreIndex - 1; i++) { //データをつめる
132:
133:           //デバッグ用
134:           System.out.println(
135:             "DEBUG: "
136:               + names[i
137:               + 1]
138:               + "さんのデータを"
139:               + (i + 1)
140:               + "番地から"
141:               + i
```

```
142:                 + "番地に移動します");
143:
144:             names[i] = names[i + 1];
145:             scores[i] = scores[i + 1];
146:           }
147:           currentScoreIndex--; //登録されている成績数を減らす
148:
149:           //正常に削除されたことを知らせる
150:           System.out.println(name + "さんの成績を削除しました");
151:
152:         }
```

```
コマンドを入力してください
(A:成績の登録，S:成績の検索，D:成績の削除，V:成績一覧と平均点の表示，Q:終了)
>>D
成績を削除したい名前を入力してください
>>佐藤
DEBUG: names[0] = 山田 判定 false
DEBUG: names[1] = 田中 判定 false
DEBUG: names[2] = 佐藤 判定 true
DEBUG: 島田さんのデータを 3 番地から 2 番地に移動します
佐藤さんの成績を削除しました
```

図 3.13: デバッグプリントの様子

3.3.2.3　人にやさしいソースコードとエラー

プログラムが自分から、また他人から見ても分かりやすければ、エラーは発生しにくくなります。それは、プログラムを書いている最中にも間違いが発見しやすく、エラーが発生してからも、どこに原因があるのか分かりやすいからです。

人にやさしいソースコードはエラーの予防策になるだけでなく、エラーが起こってからの対処も迅速に行うことができるのです。

3.3.3　静的エラーとコンパイラの役割

3.3.3.1　コンパイル

皆さんが Java を書いているときには、javac コマンドを使ってコンパイルという作業を行ってから、java というコマンドで、プログラムを実行しています。プログラムをコンパイルするというのは、人が読めるソースコードをコンピュータが実行できる形式に変換することを意味しています。この変換の過程でのエラーのことをコンパイルエラーといい

ます。このエラーは、コンパイラがソースコードを正しく解釈できなかったときに起こります。

コンパイルエラーが起こった時に、ただ闇雲にソースコードを眺めているだけでは、問題は解決しません。コンパイラはエラーの内容を表示するようにできています。ですから、コンパイルエラーの内容を落ち着いて読むことで、早くエラーの箇所と種類を特定できます。問題が起こった行の番号とエラーの内容が表示されるので、その行をまず確認し、エラーの内容をヒントにして原因を探るようにしましょう。

考えてみよう

静的エラーと動的エラーでは、どちらが原因を究明し対処するのが簡単か議論してみよう。

考えてみよう

Java では変数や配列に型を定義することが必要です。実は型を必要としないプログラミング言語もあります。型がないと、特に変数に関係付けられる値を意識しなくても良くなるので、一見便利そうに思えます。ではなぜ Java には型があるのでしょうか。

3.4　練習問題

練習問題 1

本章で紹介した成績管理アプリケーション (リスト 22: ScoreAdministratorApplication.java) には、動的なエラーがもう一つあります。エラーを探して取り除いてください。デバッグプリントも試して見てください。(ヒント:登録人数が 0 人の時)

練習問題 2

成績管理アプリケーション（リスト 22: ScoreAdministratorApplication.java）を参考にして、電話帳アプリケーションを作成してください。

次の順序で作成しましょう。

1. まずは一覧表示と追加ができるものだけを作る
2. 次に検索機能を作る
3. 最後に削除機能を作る

第 4 章

手続きを使った抽象化 (1)

この章で学習すること

手続きを使ったプログラムを記述できる
手続きを使ったプログラムの実行の過程を説明できる
変数の有効範囲を説明できる
引数を用いて手続きを抽象化できる
実引数と仮引数の結合の様子を図を用いて説明できる

4.1 手続き

4.1.1 メソッド

Java における手続きのことをメソッドといいます。メソッドとは何度も同じ仕事を書かなくて済むように、仕事をまとめて書くことができる仕組みです。

リスト 2 の足し算プログラムでは、プロンプト表示をする部分で 2 度同じソースコードが出てきます。このプロンプト表示部分をメソッドを用いてリスト 24 のようにひとまとめにして扱うようにしましょう。

リスト 24: 足し算アプリケーション (プロンプトの表示をそれぞれメソッド化した)

```
 1: /**
 2:  * 足し算計算機アプリケーション
 3:  *
 4:  * ２つの数をキーボードから読み込み、足し算の結果を表示する
 5:  * ２数とも、０であったら、終了する
 6:  *（プロンプトの表示部分をメソッド化した）
 7:  *
 8:  * @author Manabu Sugiura
 9:  * @version $Id: AddTwoNumberApplication.java,v 1.8 2003/05/07 11:40:38 gackt Exp $
10:  */
11: public class AddTwoNumberApplication {
```

```
12:
13:   public static void main(String[] args) {
14:     AddTwoNumberApplication addTwoNumberApplication =
15:       new AddTwoNumberApplication();
16:     addTwoNumberApplication.main();
17:   }
18:
19:   void main() {
20:
21:     int firstNumber; //１番目に入力された整数
22:     int secondNumber; //２番目に入力された整数
23:     int result; //２つの整数の足し算の結果
24:
25:     //アプリケーションの説明をする
26:     System.out.println("２つの数の和を求めます。");
27:     System.out.println("（2 数に 0 を入力すると終了します）");
28:
29:     //１番目の数を入力する
30:     showFirstPrompt();
31:     firstNumber = Input.getInt();
32:
33:     //２番目の数を入力する
34:     showSecondPrompt();
35:     secondNumber = Input.getInt();
36:
37:     //加算を繰り返す
38:     while (firstNumber != 0 || secondNumber != 0) { //２数とも０なら終了コード
39:
40:       //計算する
41:       result = firstNumber + secondNumber;
42:
43:       //結果を表示する
44:       System.out.println("２つの数の和は" + result + "です。");
45:
46:       //１番目の数を入力する
47:       showFirstPrompt();
48:       firstNumber = Input.getInt();
49:
50:       //２番目の数を入力する
51:       showSecondPrompt();
52:       secondNumber = Input.getInt();
53:     }
54:
55:     //アプリケーションが終了したことを知らせる
56:     System.out.println("アプリケーションを終了しました。");
57:   }
58:
59:   /**
60:    * 1 番目の数の入力を促すプロンプトを表示する
61:    */
62:   void showFirstPrompt() {
63:     System.out.print("1 つ目の整数を入力してください>>");
64:     System.out.flush();
65:   }
```

```
66:
67:   /**
68:    * 2 番目の数の入力を促すプロンプトを表示する
69:    */
70:   void showSecondPrompt() {
71:     System.out.print("2 つ目の整数を入力してください>>");
72:     System.out.flush();
73:   }
74: }
```

4.1.2 メソッドの書法

4.1.2.1 メソッドの宣言

メソッド宣言の書式は以下のようになります。

```
void [メソッド名](){

   （処理）

}
```

ここで重要なのは適切な名前をつけることです。HCP チャートに書かれた目的をなるべく反映するような名前をつけましょう。

メソッド名の前にある「void」は今はオマジナイで、そのメソッドに戻り値がないことを示しています。戻り値については次章で詳しく説明します。

メソッドコメント メソッドには必ずどのような目的を持った仕事をするのかコメントを付けます。そうすることで、内容を詳しく読まなくても、どのような仕事を目的としたメソッドであるかが分かるようになります。

```
/**
 * メソッドコメント
 */
void [メソッド名](){

   （処理）

}
```

メソッドを書く位置　Java ではメソッドは図 4.1 のようにクラスの中に書きます。他のメソッドの宣言の中で別のメソッドを宣言することはできません。

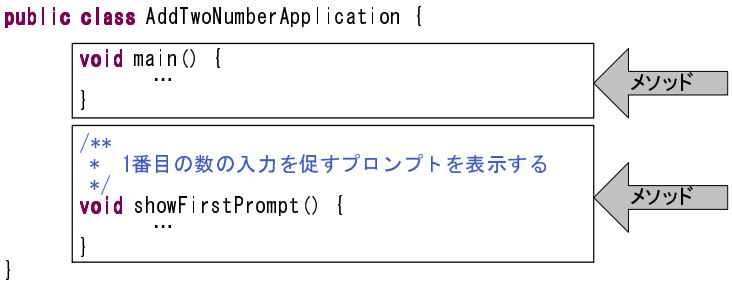


図 4.1: メソッドを書く位置　正しい例

まずい例

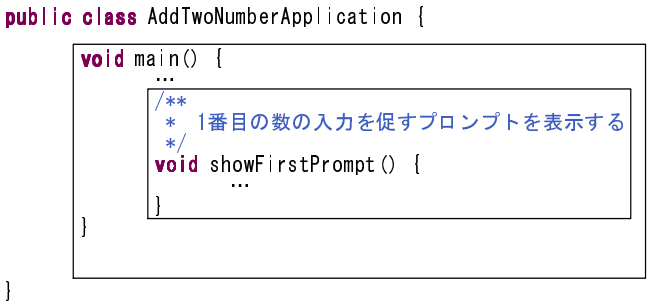


図 4.2: メソッドを書く位置　まずい例

4.1.2.2　メソッドの呼び出し

メソッドは宣言しただけでは意味がありません。プログラムを実行して、自動的に呼ばれるのは、main メソッドだけです。メソッドは、メソッドから呼び出されて仕事をします。

メソッド呼び出しの書式は以下のとおりです。

[メソッド名]();

プログラム上で上のように記述されているところがメソッドの呼び出し部分になります。メソッドに書いた仕事をさせるためには、仕事をさせたい場所でそのメソッドを呼び出す必要があります。例えばリスト 24 の場合、showFisrtPrompt() メソッドは 34,51 行目で呼び出されています。

4.1.3 メソッドと HCP チャート

メソッドを用いても、プログラムの目的の階層構造は変わりませんが、HCP チャートは図 4.3 のように分割して書くことになります。

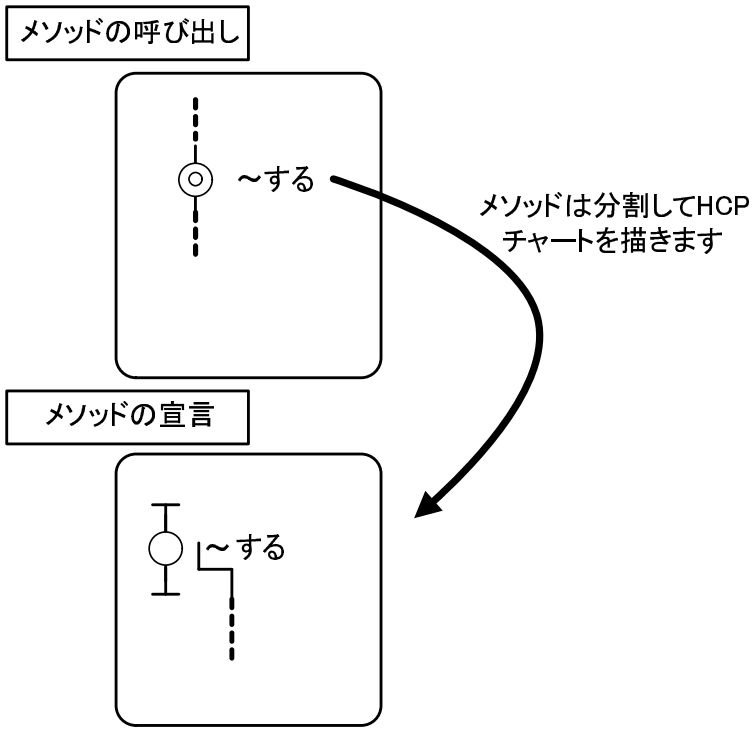


図 4.3: HCP チャートはメソッドごとに分割

2 つの数を足すアプリケーションでの具体例を図 4.4 に示します。

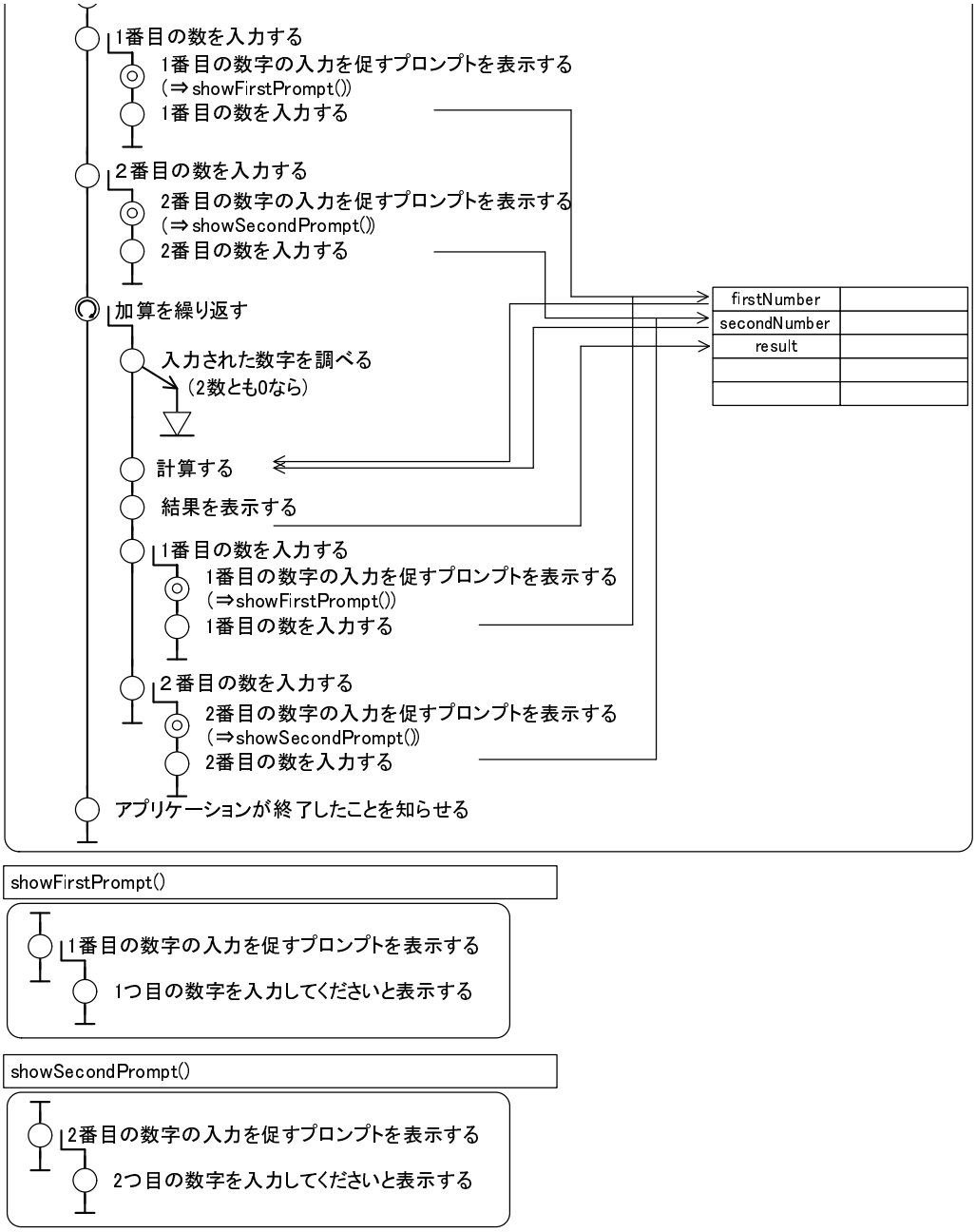


図 4.4: 2 つの数を足すアプリケーションの HCP チャート

4.1.4 変数のスコープ

前例では、プロンプトの表示をメソッド化しましたが、実は、入力された数字を取得するところまでが重複コードになっています。

どうせなら、そこまでメソッド化したいものです。しかし、リスト 25 のようなプログラムを書いてしまうとコンパイルエラーとなってしまいます。

リスト 25: 2 つの整数の和を求めるアプリケーション (エラーあり)

```
 1: package sp.addtwonumber.method_error;
 2:
 3: import common.Input;
 4:
 5: /**
 6:  * 足し算計算機アプリケーション
 7:  * ２つの数をキーボードから読み込み、足し算の結果を表示する
 8:  * ２数とも、０であったら、終了する
 9:  * （コンパイルエラーバージョン）
10:  *
11:  * @author Manabu Sugiura
12:  * @version $Id: AddTwoNumberApplication.txt,v 1.3 2003/05/07 07:09:50 gackt Exp $
13:  */
14: public class AddTwoNumberApplication {
15:
16:   public static void main(String[] args) {
17:     AddTwoNumberApplication addTwoNumberApplication =
18:       new AddTwoNumberApplication();
19:     addTwoNumberApplication.main();
20:   }
21:
22:   void main() {
23:
24:     int firstNumber; //１番目に入力された整数
25:     int secondNumber; //２番目に入力された整数
26:     int result; //２つの整数の足し算の結果
27:
28:     //アプリケーションの説明をする
29:     System.out.println("２つの数の和を求めます。");
30:     System.out.println("（2 数に 0 を入力すると終了します）");
31:
32:     //１番目の数を入力する
33:     showFirstPrompt();
34:
35:     //２番目の数を入力する
36:     showSecondPrompt();
37:
38:     //加算を繰り返す
39:     while (firstNumber != 0 || secondNumber != 0) { //２数とも０なら終了コード
40:
41:       //計算する
42:       result = firstNumber + secondNumber;
```

```
43:
44:       //結果を表示する
45:       System.out.println("２つの数の和は" + result + "です。");
46:
47:       //１番目の数を入力する
48:       showFirstPrompt();
49:
50:       //２番目の数を入力する
51:       showSecondPrompt();
52:     }
53:
54:     //アプリケーションが終了したことを知らせる
55:     System.out.println("アプリケーションを終了しました。");
56:   }
57:
58:   /**
59:    * 1 番目の数の入力を促すプロンプトを表示する
60:    */
61:   void showFirstPrompt() {
62:     System.out.print("1 つ目の整数を入力してください>>");
63:     System.out.flush();
64:     firstNumber = Input.getInt();
65:   }
66:
67:   /**
68:    * 2 番目の数の入力を促すプロンプトを表示する
69:    */
70:   void showSecondPrompt() {
71:     System.out.print("2 つ目の整数を入力してください>>");
72:     System.out.flush();
73:     secondNumber = Input.getInt();
74:   }
75: }
```

何故コンパイルエラーとなってしまうのかを表モデルを用いて説明します。

変数の表モデルはメソッド毎に別々に作られます。メソッドが呼び出された瞬間に新しい表が生まれ、そのメソッド内で宣言された変数は新しいほうの表に書き込まれます（図 4.5)。

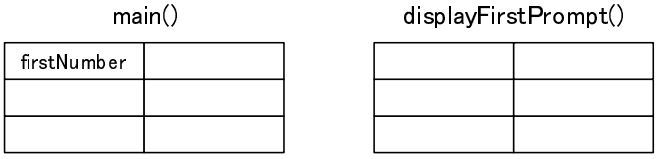


図 4.5: メソッドの表モデル

具体的に図 4.5 の変数 rstNumber　に注目してみましょう。この変数は main メソッドで宣言されているので、main メソッドの表に書き込まれます。

この状態では、rstNumber が displayFirstPrompt メソッドでは宣言されていない変数なので、displayFirstPrompt メソッドで代入することができません。

これはもし実行できたらの話だったのですが、実行できないことはコンパイル時に明らかなので、コンパイラがコンパイルエラー (図 4.6) として警告します。

```
AddTwoNumberApplication.java:64: シンボルを解決できません。
シンボル: 変数 firstNumber
場所    : sp.addtwonumber.method_error.AddTwoNumberApplication の クラス
                firstNumber = Input.getInt();
                ^
AddTwoNumberApplication.java:73: シンボルを解決できません。
シンボル: 変数 secondNumber
場所    : sp.addtwonumber.method_error.AddTwoNumberApplication の クラス
                secondNumber = Input.getInt();
                ^
エラー 2 個
```

図 4.6: コンパイルエラー

このように変数はその宣言が行われたブロック内でしか評価できません。(図 4.5) この評価可能な範囲のことをその変数のスコープといいます。

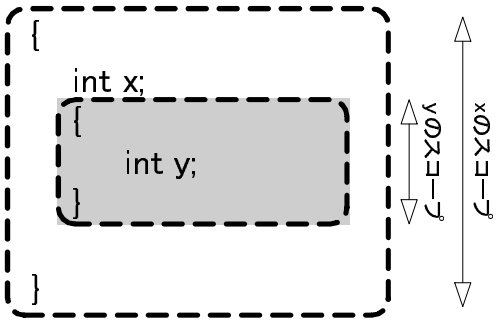


図 4.7: 変数のスコープ

変数の値の変更もその変数のスコープ内でのみ有効です。つまりメソッド内で宣言されている変数はそのメソッドが終了してしまうと、変数の値はどこにも反映されません。

4.2　引数による手続きの抽象化

4.2.1　汎用的な手続き

リスト 24 の足し算アプリケーションでは、1 つ目の数の入力プロンプトと 2 つ目の数の入力プロンプトをそれぞれ別々にメソッド化していました。

この 2 つのメソッドをよく見比べてみましょう。

リスト 26: 1 番目の数の入力を促すプロンプトを表示するメソッド

```
62:   void showFirstPrompt() {
63:     System.out.print("1 つ目の整数を入力してください>>");
64:     System.out.flush();
65:   }
```

リスト 27: 2 番目の数の入力を促すプロンプトを表示するメソッド

```
70:   void showSecondPrompt() {
71:     System.out.print("2 つ目の整数を入力してください>>");
72:     System.out.flush();
73:   }
```

すぐに分かるように、この 2 つのメソッドは表示するプロンプトのテキスト (文字列) が、1 つ目用か 2 つ目用かという違いだけで、他は全く同じです。このような、仕事は同じだけれども扱うデータが違うメソッドは**引数**を使うことにより、一つにまとめることができます。

今、何番目の数字の入力とするかは main メソッドが決めることであり、プロンプト表示メソッドはそのスコープの外になります。引数を使うと、スコープ外のデータのやり取りをすることが出来ます。

リスト 28: 2 つの整数の和を求めるアプリケーション（引数のあるメソッドを導入）

```
1: /**
2:  * 足し算計算機アプリケーション
3:  *
4:  * ２つの数をキーボードから読み込み、足し算の結果を表示する
5:  * ２数とも、０であったら、終了する
6:  *（引数ありメソッド化バージョン）
7:  *
8:  * @author Manabu Sugiura
9:  * @version $Id: AddTwoNumberApplication.java,v 1.7 2003/05/07 07:09:50 gackt Exp $
```

```
10:  */
11: public class AddTwoNumberApplication {
12:
13:   public static void main(String[] args) {
14:     AddTwoNumberApplication addTwoNumberApplication =
15:       new AddTwoNumberApplication();
16:     addTwoNumberApplication.main();
17:   }
18:
19:   void main() {
20:
21:     int firstNumber; //１番目に入力された整数
22:     int secondNumber; //２番目に入力された整数
23:     int result; //２つの整数の足し算の結果
24:
25:     //アプリケーションの説明をする
26:     System.out.println("２つの数の和を求めます。");
27:     System.out.println("（2 数に 0 を入力すると終了します）");
28:
29:     //１番目の数を入力する
30:     showPrompt(1);
31:     firstNumber = Input.getInt();
32:
33:     //２番目の数を入力する
34:     showPrompt(2);
35:     secondNumber = Input.getInt();
36:
37:     //加算を繰り返す
38:     while (firstNumber != 0 || secondNumber != 0) { //２数とも０なら終了コード
39:
40:       //計算する
41:       result = firstNumber + secondNumber;
42:
43:       //結果を表示する
44:       System.out.println("２つの数の和は" + result + "です。");
45:
46:       //１番目の数を入力する
47:       showPrompt(1);
48:       firstNumber = Input.getInt();
49:
50:       //２番目の数を入力する
51:       showPrompt(2);
52:       secondNumber = Input.getInt();
53:     }
54:
55:     //アプリケーションが終了したことを知らせる
56:     System.out.println("アプリケーションを終了しました。");
57:   }
58:
59:   /**
60:    * 数の入力を促すプロンプトを表示する
61:    */
62:   void showPrompt(int i) {
63:     System.out.print(i + "つ目の整数を入力してください>>");
```

```
64:     System.out.flush();
65:   }
66:
67: }
```

4.2.1.1 引数ありメソッドの書法

引数のあるメソッドの場合、呼び出し部分と宣言部分は次のようになります。

メソッド宣言

```
void [メソッド名]([引数の型 引数名]*){

  （処理）

}
```

メソッド呼び出し

```
[メソッド名]([引数]*);
```

引数はいくつでも定義することができます。
呼び出し部分の引数を**実引数**、宣言部分の引数を**仮引数**と呼びます。

4.2.2 プログラムの実行と引数

4.2.2.1 引数が値の場合

図 4.8 は、引数のあるメソッドが実行される様子を、表モデルを使って表しています。

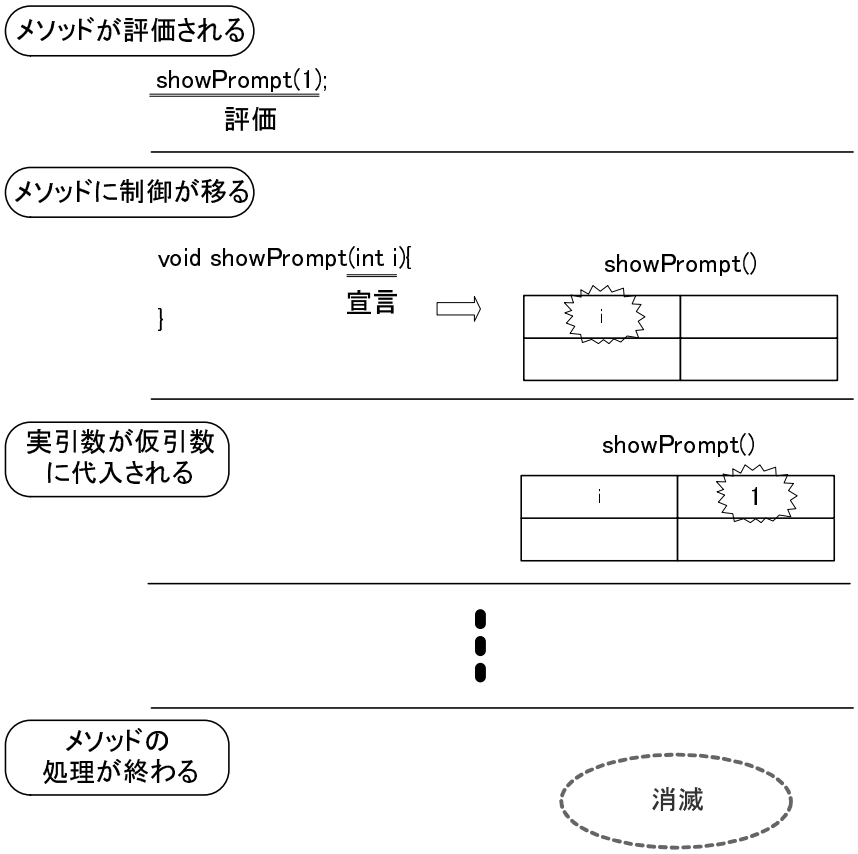


図 4.8: 引数が値の場合の評価

メソッドが呼び出されると新たな表が生成され、実引数が仮引数に代入されます。そしてメソッドの処理が終わるとメソッドの表が消滅するというのがメソッド実行の大まかな流れです。

4.2.2.2 引数が変数の場合

リスト 29 は、引数を使って足し算プログラムの足し算結果を表示する部分をメソッドにしたものです。

リスト 29: 2 つの整数の和を求めるアプリケーション (足し算の結果表示部分をメソッド化)

```
 1: /**
 2:  * 足し算計算機アプリケーション
 3:  *
 4:  * ２つの数をキーボードから読み込み、足し算の結果を表示する
 5:  * ２数とも、０であったら、終了する
 6:  *（足し算の結果表示部分をメソッド化）
 7:  *
 8:  * @author Manabu Sugiura
 9:  * @version $Id: AddTwoNumberApplication.java,v 1.8 2003/05/07 11:49:43 gackt Exp $
10:  */
11: public class AddTwoNumberApplication {
12:
13:   public static void main(String[] args) {
14:     AddTwoNumberApplication addTwoNumberApplication =
15:       new AddTwoNumberApplication();
16:     addTwoNumberApplication.main();
17:   }
18:
19:   void main() {
20:
21:     int firstNumber; //１番目に入力された整数
22:     int secondNumber; //２番目に入力された整数
23:     int result; //２つの整数の足し算の結果
24:
25:     //アプリケーションの説明をする
26:     System.out.println("２つの数の和を求めます。");
27:     System.out.println("（2 数に 0 を入力すると終了します）");
28:
29:     //１番目の数を入力する
30:     showPrompt(1);
31:     firstNumber = Input.getInt();
32:
33:     //２番目の数を入力する
34:     showPrompt(2);
35:     secondNumber = Input.getInt();
36:
37:     //加算を繰り返す
38:     while (firstNumber != 0 || secondNumber != 0) { //２数とも０なら終了コード
39:
40:       //足し算の結果を表示する
41:       showAddResult(firstNumber, secondNumber);
42:
43:       //１番目の数を入力する
```

```
44:       showPrompt(1);
45:       firstNumber = Input.getInt();
46:
47:       //２番目の数を入力する
48:       showPrompt(2);
49:       secondNumber = Input.getInt();
50:     }
51:
52:     //アプリケーションが終了したことを知らせる
53:     System.out.println("アプリケーションを終了しました。");
54:   }
55:
56:   /**
57:    * 数の入力を促すプロンプトを表示する
58:    */
59:   void showPrompt(int i) {
60:     System.out.print(i + "つ目の整数を入力してください>>");
61:     System.out.flush();
62:   }
63:
64:   /**
65:    * 足し算の結果を表示する
66:    */
67:   void showAddResult(int firstNumber, int secondNumber) {
68:
69:     //足し算をする
70:     int additionResult = firstNumber + secondNumber;
71:
72:     //結果を表示する
73:     System.out.println("2 つの数の和は" + additionResult + "です。");
74:   }
75: }
```

このプログラムの実行の様子を、図 4.9 を見ながら追っていきましょう。

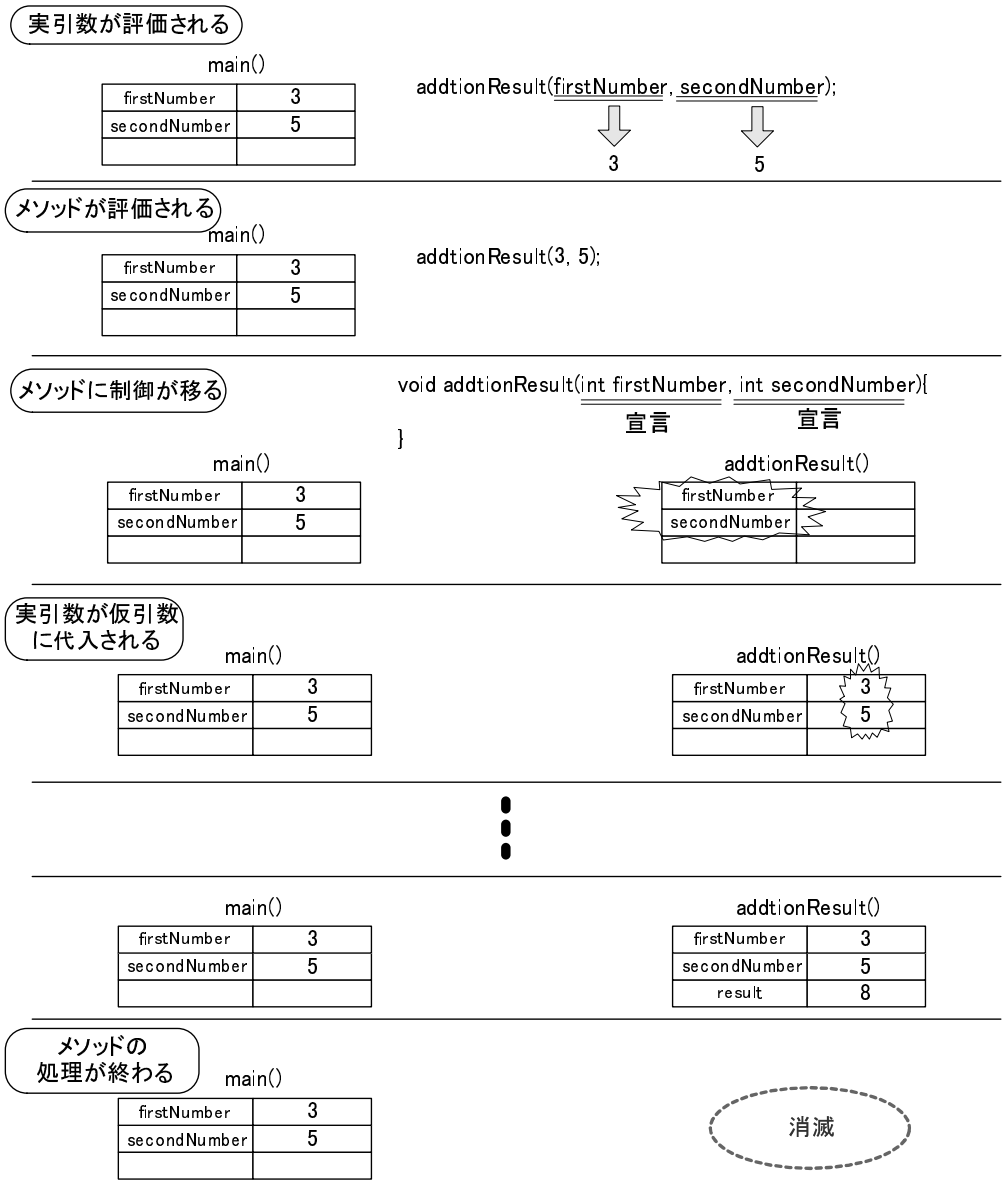


図 4.9: 引数が変数の場合の評価

実引数が変数や式である場合は、表モデルよりそれらを評価して値にします。呼び出し部分の評価が済むと、新たにそのメソッドをスコープとした表ができます。

続いてメソッド宣言部が評価され、それぞれの仮引数が表に追加され、最後に対応する実引数の値が代入されます。

4.2.2.3 引数が配列の場合

リスト 16 を改良して、成績を新規に登録する部分をメソッドにしたものがリスト 30 になります。このようにメソッドの引数には配列を渡すこともできます。

リスト 30: 成績管理をするアプリケーション (新規登録部分をメソッド化)

```
 1: /**
 2:  * 成績を管理するアプリケーション
 3:  *
 4:  * ・名前と成績の登録（コマンド：A）
 5:  * ・成績一覧と平均表示（コマンド：V）
 6:  *
 7:  * 科目の成績を登録すると、上の動作をコマンドの選択によって行う
 8:  *
 9:  * @author Masahiro Kawamura
10:  * @version $Id: ScoreAdministratorApplication.java,v 1.11 2003/05/08 03:50:25 duskin Exp $
11:  */
12: public class ScoreAdministratorApplication {
13:
14:   public static void main(String[] args) {
15:     ScoreAdministratorApplication scoreAdministratorApplication =
16:       new ScoreAdministratorApplication();
17:     scoreAdministratorApplication.main();
18:   }
19:
20:   void main() {
21:
22:     final int SCORE_SIZE = 10; // 入力できる成績の数
23:
24:     int[] scores = new int[SCORE_SIZE]; //全員の成績
25:     String[] names = new String[SCORE_SIZE]; //全員の名前
26:
27:     int currentScoreIndex = 0; //現在登録されているデータの数
28:     String command; //入力されたコマンド
29:
30:     //アプリケーションの説明をする
31:     System.out.println("★★★★★★成績管理アプリケーション★★★★★★");
32:     System.out.println("（名前と成績の登録、成績一覧・平均の出力ができます。）");
33:
34:     //メニューを出力し、コマンドで指定された処理を行う
35:     while (true) {
36:
37:       //メニューを出力し、コマンドを入力する
38:       System.out.println("コマンドを入力してください");
39:       System.out.println("(A:成績の登録，V:成績一覧と平均点の表示，Q:終了)");
40:       System.out.print(">>");
41:       System.out.flush();
42:       command = Input.getString();
43:
44:       //コマンドで指定された処理を行う
```

```
45:       if (command.equals("A")) { //成績を登録する
46:
47:         // 名前と成績を登録する
48:         add(scores, names, currentScoreIndex);
49:         currentScoreIndex++; //登録人数を増やす
50:
51:       } else if (command.equals("V")) { //成績一覧と平均点を表示する
52:
53:         showScoreList(scores, names, currentScoreIndex);
54:
55:       } else if (command.equals("Q")) { //アプリケーションを終了する
56:
57:         break; //終了
58:
59:       } else { //エラー処理
60:
61:         //エラーメッセージを表示する
62:         System.out.println("そのようなコマンドはありません");
63:       }
64:     }
65:
66:     //アプリケーションが終了したことを知らせる
67:     System.out.println("アプリケーションが終了しました。");
68:   }
69:
70:   /**
71:    * 名前と成績を登録する
72:    */
73:   void add(int[] scores, String[] names, int currentScoreIndex) {
74:
75:     String name; //入力された個人の名前
76:     int score; // 入力された個人の成績
77:
78:     //名前を入力する
79:     System.out.println("名前を入力してください");
80:     System.out.print(">>");
81:     System.out.flush();
82:     name = Input.getString();
83:
84:     //成績を入力する
85:     System.out.println("成績を入力してください");
86:     System.out.print(">>");
87:     System.out.flush();
88:     score = Input.getInt();
89:
90:     //データを登録する
91:     names[currentScoreIndex] = name;
92:     scores[currentScoreIndex] = score;
93:
94:     //正常に登録されたことを知らせる
95:     System.out.println(name + "さんの成績を登録しました");
96:   }
97:
98:   /**
```

```
 99:     * 成績一覧と平均点を表示する
100:     */
101:    void showScoreList(int[] scores, String[] names, int currentScoreIndex) {
102:
103:      double average; //平均点
104:      double total; //合計点
105:      int i; //ループ用
106:
107:      //成績を一覧表示する
108:      System.out.println("★★★★★★成績一覧表★★★★★★");
109:      for (i = 0; i < currentScoreIndex; i++) {
110:        System.out.println(names[i] + "さん:" + scores[i] + "点");
111:      }
112:
113:      //平均点を計算する
114:      total = 0.0; //合計点を初期化する
115:      for (i = 0; i < currentScoreIndex; i++) { //合計点を計算する
116:        total = total + scores[i];
117:      }
118:      average = total / currentScoreIndex; //平均点を計算する
119:
120:      //平均点を表示する
121:      System.out.println("★★★★★★平均点★★★★★★");
122:      System.out.println("平均点：" + average + "点");
123:    }
124: }
```

引数を配列にした場合のメソッドの評価は図 4.10 のようになります。配列の場合も変数の時と同様に実引数から仮引数への代入が行われます。

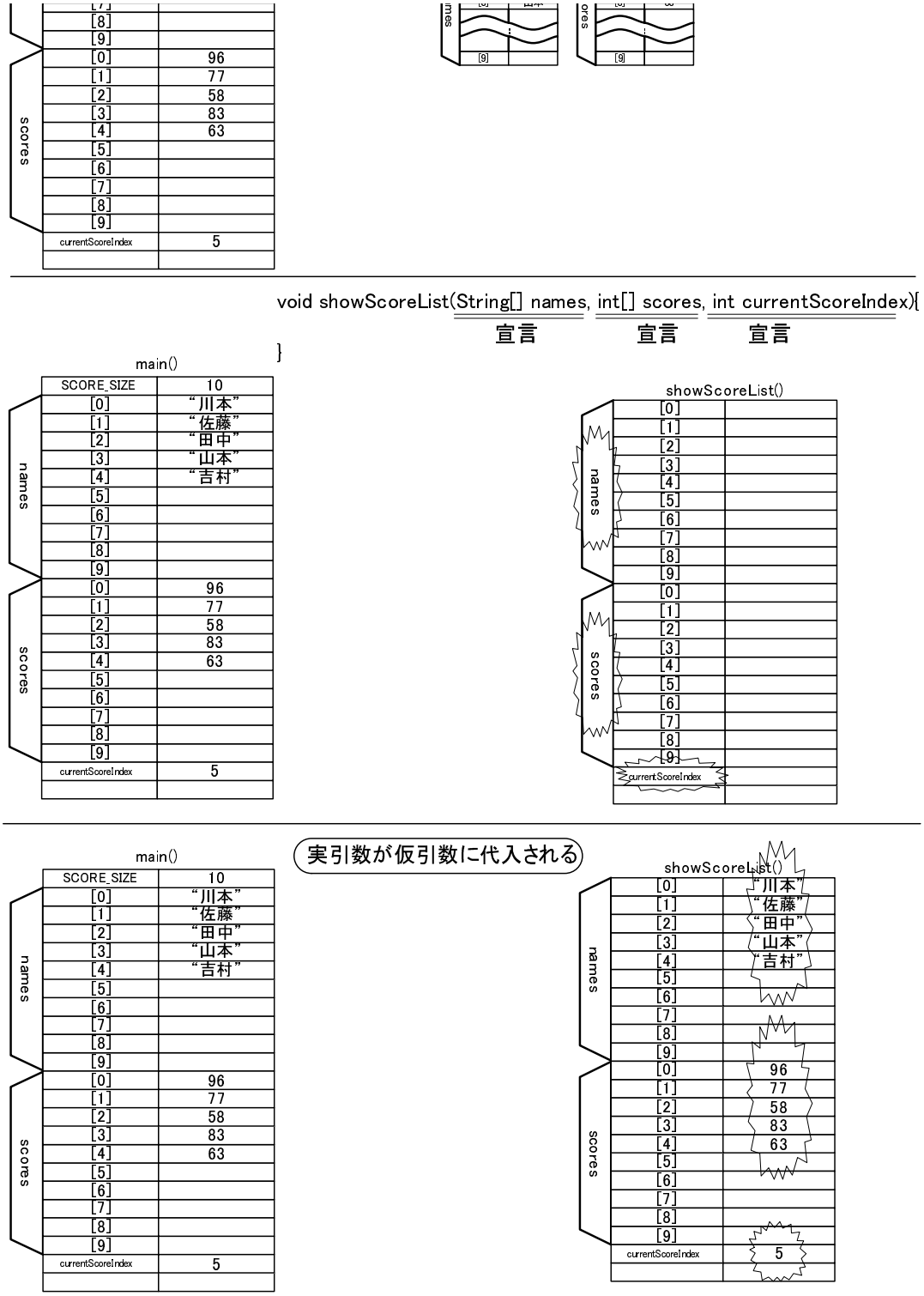


図 4.10: 配列を引数にする場合の評価 (showScoreList メソッド)

次に、add メソッドの場合のプログラム実行の様子を図 4.11 と図 4.12 で追ってみましょう。

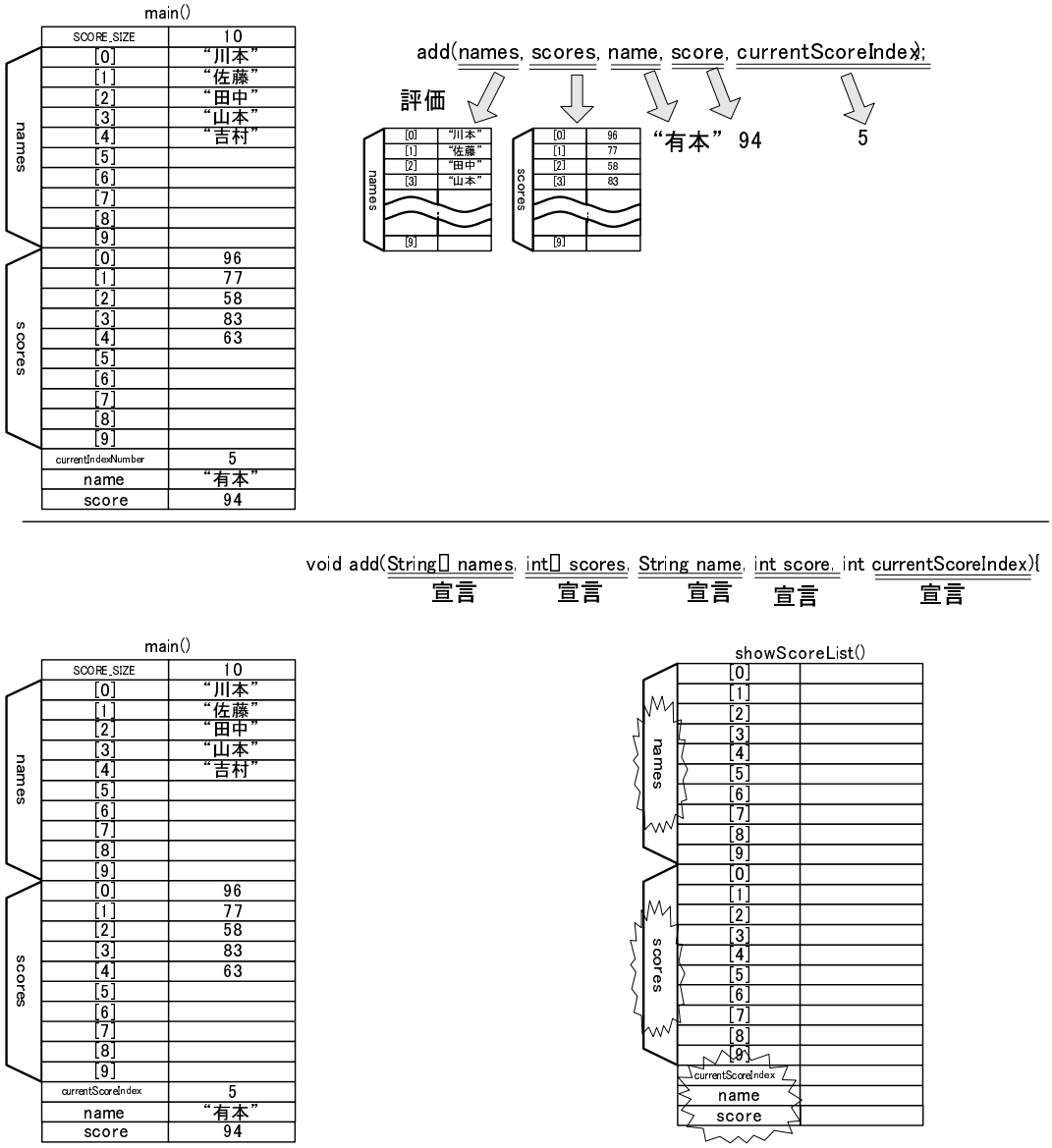


図 4.11: 配列を引数にする場合の評価 (add メソッド)(1)

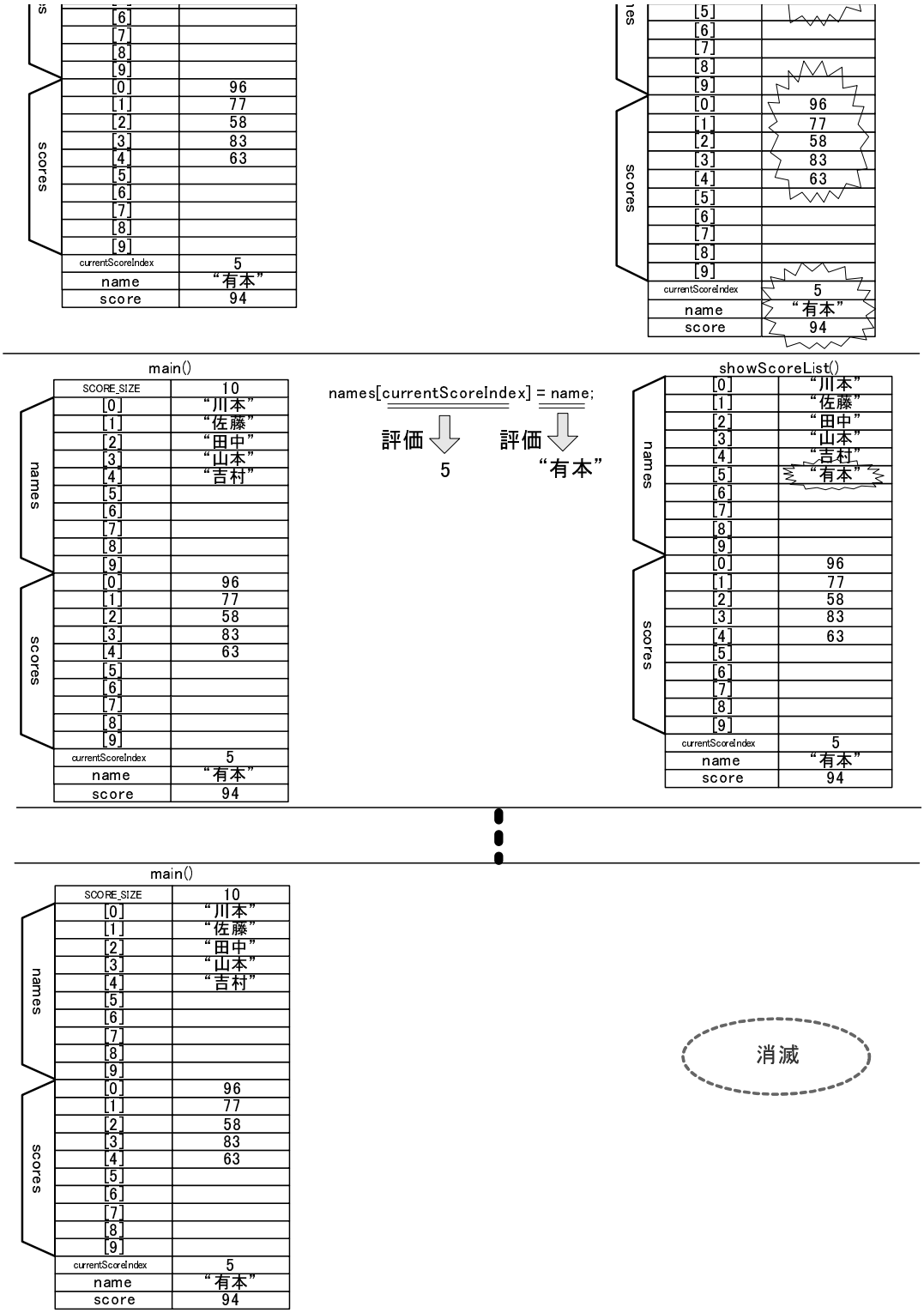


図 4.12: 配列を引数にする場合の評価 (add メソッド)(2)

図 4.11 と図 4.12 で示したプログラム実行のプロセスを見て、疑問に思ったことは無いでしょうか。

これまでの表モデルの評価のプロセスから考えれば、メソッドで新しい成績を追加しても、その変更は main メソッドには反映されないので、メソッドが消滅してしまうと追加した分も消えてしまうことになります。

しかし、実はこれでプログラムは正しく動きます。なぜなら、配列の場合は、今までのモデルの説明とは違った動作をするからです。

配列の場合は図 4.13 のように、違うメソッドにある配列で値の変化が同期していて、[1]一方のメソッドでの配列の変更がすぐに別のメソッドの配列にも反映されます。

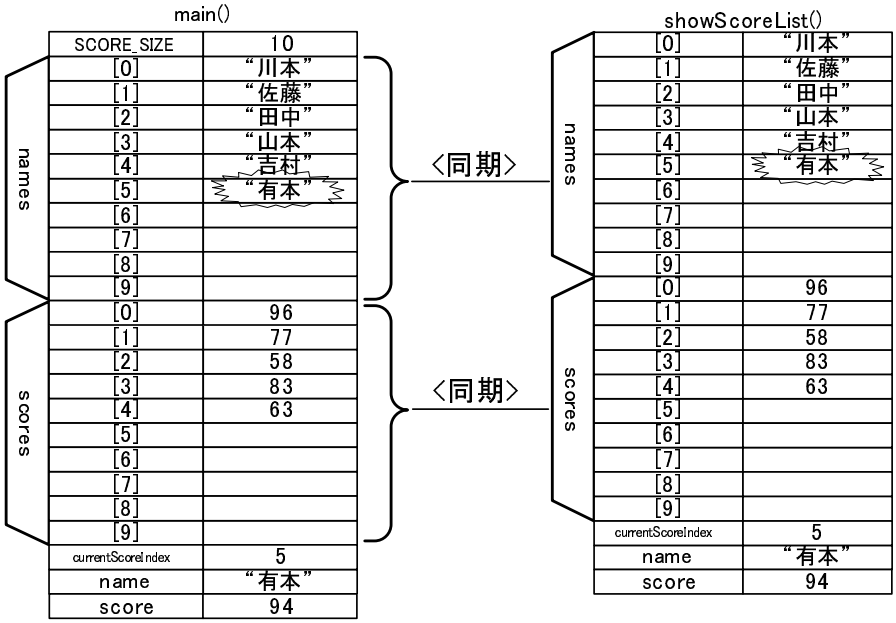


図 4.13: 配列を引数にする場合の表モデル

このような配列の場合の特別な仕組みを頭に入れておきましょう。

[1] 実際には同期しているのではなく、「同じオブジェクト」として扱われているために、あたかも同期しているような動作をします。より正確な仕組みは 10.1.1.5 節を参照してください。

4.3　練習問題

練習問題 1

絵文字を表示するプログラム（DrawJapanSample.java）をメソッドを用いて書き直してください。

練習問題 2

正札出力アプリケーション（PriceTagApplication.java）のアプリケーションの説明表示部分、正札出力部分に関して、メソッドを用いて書き直してください。

練習問題 3

前章の練習問題で作った電話帳アプリケーションのコマンド一覧部分をメソッドにしてください。

第 5 章

手続きを使った抽象化 (2)

この章で学習すること

戻り値のあるメソッドが書ける
戻り値のあるメソッドの実行の仕組みを説明できる
手続きが階層化されたプログラムが書ける
インスタンス変数を使ったプログラムが書ける

5.1 手続きと値

5.1.1 手続きの評価と戻り値

戻り値のあるメソッド

前章では、2 つの数の足し算をするアプリケーションを改良して、重複コードをメソッド化してきました。ここでは、さらに戻り値を利用する方法を考えていきます。

前章のリスト 29 では、プロンプトを表示するだけだったメソッドを、プロンプトを表示して入力された数字も取得する getNumber というメソッドに書き換えたのがリスト 31 です。ある手続きで処理を行った結果を得たいという要求は、今まで紹介したメソッドの使い方では実現することができませんでした。今回、戻り値を使うことで、呼び出した側からメソッドで行った処理の結果を得ることができるようになりました。

リスト 31: 2 つの整数の和を求めるアプリケーション（メソッドの戻り値を利用する）

```
1: /**
2:  * 足し算計算機アプリケーション
3:  *
4:  * ２つの数をキーボードから読み込み、足し算の結果を表示する
5:  * ２数とも、０であったら、終了する
6:  *（プロンプトを表示して入力を受け付ける部分もメソッド化）
7:  *
8:  * @author Manabu Sugiura
```

```
 9:  * @version $Id: AddTwoNumberApplication.java,v 1.8 2003/05/07 11:50:28 gackt Exp $
10:  */
11: public class AddTwoNumberApplication {
12:
13:   public static void main(String[] args) {
14:     AddTwoNumberApplication addTwoNumberApplication =
15:       new AddTwoNumberApplication();
16:     addTwoNumberApplication.main();
17:   }
18:
19:   void main() {
20:
21:     int firstNumber; //１番目に入力された整数
22:     int secondNumber; //２番目に入力された整数
23:     int result; //２つの整数の足し算の結果
24:
25:     //アプリケーションの説明をする
26:     System.out.println("２つの数の和を求めます。");
27:     System.out.println("（2 数に 0 を入力すると終了します）");
28:
29:     //2 数を入力する
30:     firstNumber = getNumber(1);
31:     secondNumber = getNumber(2);
32:
33:     //加算を繰り返す
34:     while (firstNumber != 0 || secondNumber != 0) { //２数とも０なら終了コード
35:
36:       //足し算の結果を表示する
37:       showAddResult(firstNumber, secondNumber);
38:
39:       //2 数を入力する
40:       firstNumber = getNumber(1);
41:       secondNumber = getNumber(2);
42:     }
43:
44:     //アプリケーションが終了したことを知らせる
45:     System.out.println("アプリケーションを終了しました。");
46:   }
47:
48:   /**
49:    * 足し算の結果を表示する
50:    */
51:   void showAddResult(int firstNumber, int secondNumber) {
52:
53:     //足し算をする
54:     int result = firstNumber + secondNumber;
55:
56:     //結果を表示する
57:     System.out.println("2 つの数の和は" + result + "です。");
58:   }
59:
60:   /**
61:    * プロンプトを表示して、数を入力する
62:    */
```

```
63:   int getNumber(int i) {
64:
65:     int number; //入力された数
66:
67:     //入力する
68:     System.out.print(i + "つ目の整数を入力してください>>");
69:     System.out.flush();
70:     number = Input.getInt();
71:
72:     //入力された値を戻り値とする
73:     return number;
74:   }
75: }
```

戻り値のあるメソッドの書法

戻り値を使ったメソッドの書法は以下のとおりです。

```
[戻り値の型] [メソッド名] ([引数の型 引数名]*){

  (処理)

  return [戻り値];
}
```

いままで使っていた void という戻り値は「戻り値がない」ということなのでした。
引数はいくつでも設定できますが、戻り値は 1 つしか返すことができません。

プログラムの実行と戻り値

ここでは、表モデルを使った戻り値のあるメソッドの評価プロセスについて説明します。

戻り値のあるメソッドを使用した場合の評価のプロセスは図 5.1 のようになります。

rstNumber = getNumber(1)　に注目してみましょう。まず getNumber メソッドが評価されます。

getNumber メソッドでは仮引数がまず評価され、変数の表に変数 i が書き込まれます。実引数が仮引数に代入され、値が 1 に決まります。ここまでは今までのメソッドのプロセスと同様です。

number = Input.getInt(); の場合も右辺の Input.getInt() が評価されて、キーボードからの入力（この場合は 3）が評価された値となり、左辺の number に代入ます。

最後に return の文が評価されると、number の値 3 が、**getNumber メソッドが評価された値として**main メソッドに返され、main メソッドの rstNumber　に代入されます。

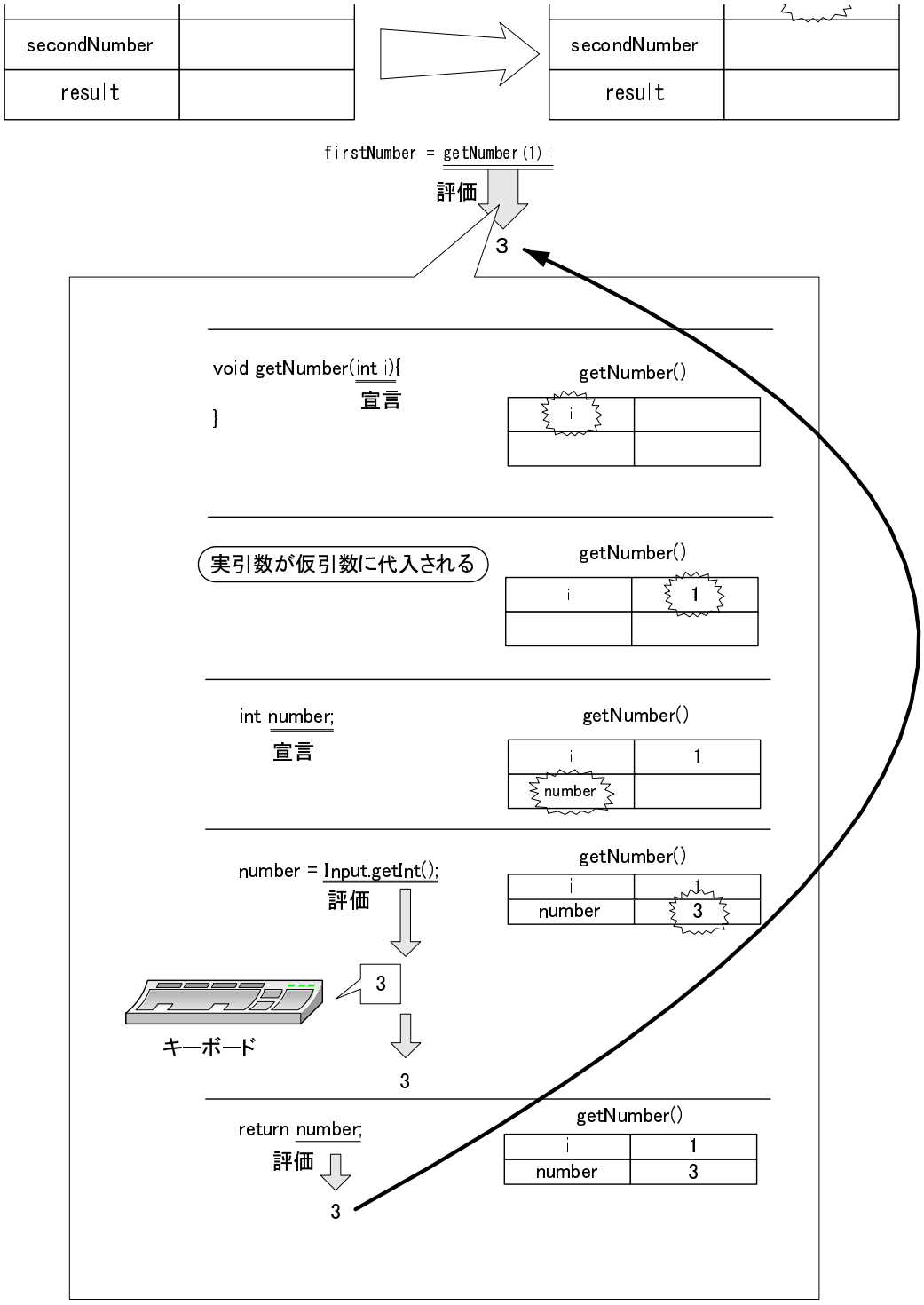


図 5.1: 戻り値のあるメソッドの評価

5.1.2　変数, 手続きと評価

今までは、「変数は評価し、具体的な値にする」そして、「手続き (メソッド) は評価されると処理を行う」というように区別してプログラムの評価のプロセスを学習してきました。しかし、戻り値を使うと、メソッドは変数と同じように、評価を行うと意味のある値に具体化することができます。そう考えると、変数も手続きも「評価されるとある値を返す」という点で同じもののように考えることが出来ます。

「評価して値を返す」というプロセスを考えれば、今まで使ってきた Input.getString() や Input.getInt() なども、「評価を行うとコンソールから入力された文字や整数型の値になる」と考えると、もうオマジナイではなくなります。

まだ、なぜ「Input.」という記述が必要かということはまだ明らかになっていませんが [1]、いまは「戻り値のあるメソッドなのだ」という理解で十分です。

考えてみよう

変数と手続きは何が同じで、何が違うのか考えてみよう。

[1] Input.getString() などは static メソッドという特別なメソッドです。

5.2　プログラムの意味を明確にするための手続き

5.2.1　何をメソッドにすべきか

一通りメソッドが書け、使えるようになりましたが、人にやさしいプログラムを書くためには、どこでどのようなメソッドを作るか、ということを考えねばなりません。

ここでは何をメソッドにしていくかについてを議論していきましょう。

重複コードのメソッド化

考えてみよう

重複コードをメソッド化するとどんな利点がありますか。

意味を考えたメソッド化

考えてみよう

コメントとメソッド化の関係を考えてみよう。

意味と重複コードの関係

考えてみよう

重複コードと、プログラムの目的にはどのような関係があるでしょう。

5.2.2　意味を明確にするメソッド化

メソッドを使うと、処理をまとめることができるので、一つ一つのプログラムをより小さい単位で管理することが出来ます。これは、たとえ重複コードでなくても、メソッドを使うことの利点の一つです。

このテキストで 1 章から学んできたのは、人に分かりやすいプログラム、目的が明確なプログラムをいかに書くかということでした。ここでは、メソッドを使って、いかに意味が明確なプログラムを書くかという議論をしたいと思います。

条件文のメソッド化

次のプログラム (リスト 32) は、おなじみの加算アプリケーションの戻り値ありメソッド版 (リスト 31) に対して、さらにメソッドを増やしたプログラムです。

どこがメソッドになったのか、比べてみましょう。

リスト 32: 2 つの整数の和を求めるアプリケーション （条件式メソッド化版）

```
 1: /**
 2:  * 足し算計算機アプリケーション
 3:  *
 4:  * ２つの数をキーボードから読み込み、足し算の結果を表示する
 5:  * ２数とも、０であったら、終了する
 6:  *（条件式部分をメソッド化）
 7:  *
 8:  * @author Manabu Sugiura
 9:  * @version $Id: AddTwoNumberApplication.java,v 1.9 2003/05/07 11:50:11 gackt Exp $
10:  */
11: public class AddTwoNumberApplication {
12:
13:   public static void main(String[] args) {
14:     AddTwoNumberApplication addTwoNumberApplication =
15:       new AddTwoNumberApplication();
16:     addTwoNumberApplication.main();
17:   }
18:
19:   void main() {
20:
21:     int firstNumber; //１番目に入力された整数
22:     int secondNumber; //２番目に入力された整数
23:     int result; //２つの整数の足し算の結果
24:
25:     //アプリケーションの説明をする
26:     System.out.println("２つの数の和を求めます。");
27:     System.out.println("（2 数に 0 を入力すると終了します）");
28:
29:     //2 数を入力する
30:     firstNumber = getNumber(1);
```

```
31:     secondNumber = getNumber(2);
32:
33:     //加算を繰り返す
34:     while (isExitCode(firstNumber, secondNumber)) {
35:
36:       //足し算の結果を表示する
37:       showAddResult(firstNumber, secondNumber);
38:
39:       //2 数を入力する
40:       firstNumber = getNumber(1);
41:       secondNumber = getNumber(2);
42:     }
43:
44:     //アプリケーションが終了したことを知らせる
45:     System.out.println("アプリケーションを終了しました。");
46:   }
47:
48:   /**
49:    * 足し算の結果を表示する
50:    */
51:   void showAddResult(int firstNumber, int secondNumber) {
52:
53:     //足し算をする
54:     int additionResult = firstNumber + secondNumber;
55:
56:     //結果を表示する
57:     System.out.println("2 つの数の和は" + additionResult + "です。");
58:   }
59:
60:   /**
61:    * プロンプトを表示して、数を入力する
62:    */
63:   int getNumber(int i) {
64:
65:     int number; //入力された数
66:
67:     //入力する
68:     System.out.print(i + "つ目の整数を入力してください>>");
69:     System.out.flush();
70:     number = Input.getInt();
71:
72:     //入力された値を戻り値とする
73:     return number;
74:   }
75:
76:   /**
77:    * 終了コードかどうか調べる
78:    */
79:   boolean isExitCode(int firstNumber, int secondNumber) {
80:     return firstNumber != 0 || secondNumber != 0; //2 数とも 0 なら終了コードとする
81:   }
82: }
```

これは、たった 1 行の条件文のメソッド化という問題です。オリジナル (リスト 31) のほうでの条件文では、プログラムだけではこれが何を目的とした条件なのかが明らかではありません。(そのためにコメントが必要なのでした。) しかし、条件文をメソッド化 (リスト 32) した場合、コメントが無くても目的が明らかになると思います。

このように、意味のある処理の単位をメソッド化することにより、コメントに頼らずにプログラムの意味を明確にすることができます。

ただし、このようなメソッド化をいつも行うとは限りません。細かすぎるメソッドは、逆にプログラムが分かりにくくなってしまいますので、プログラムやメソッド全体の長さや、分かりやすさが考慮されるべきです。

考えてみよう

次のプログラム (リスト 33) は、１章で登場した挨拶アプリケーション (リスト 9) の条件式メソッド化版です。

このプログラムを読んで、このメソッド化は有用かどうか議論してみましょう。

リスト 33: 名前と挨拶を表示するアプリケーション（条件式メソッド化版）

```
 1: /**
 2:  * 名前をキーボードから入力し、挨拶を表示することを繰り返し行うアプリケーション
 3:  *
 4:  * なにも入力せずに改行するとアプリケーションを終了する。
 5:  *（条件式メソッド版）
 6:  *
 7:  * @author Manabu Sugiura
 8:  * @version $Id: GreetingApplication.java,v 1.5 2003/05/04 20:30:36 gackt Exp $
 9:  */
10: public class GreetingApplication {
11:
12:   public static void main(String[] args) {
13:     GreetingApplication greetingApplication = new GreetingApplication();
14:     greetingApplication.main();
15:   }
16:
17:   void main() {
18:
19:     String name; //表示する名前
20:
21:     //アプリケーションの説明を出力する
22:     System.out.println("名前をすると、挨拶を表示します。名前が空文字の場合は終了します。");
23:
24:     //名前を入力する
25:     System.out.println("名前を入力してください");
26:     System.out.print(">>");
27:     System.out.flush();
28:     name = Input.getString();
29:
```

```
30:     //名前を入力し、表示する
31:     while (isNoCharacter(name)) {
32:
33:       //挨拶をして名前を出力する
34:       System.out.println("こんにちは。" + name + "さん。");
35:       System.out.println("これから長いお付き合いになりますね。");
36:
37:       //次の名前を入力する
38:       System.out.println("名前を入力してください");
39:       System.out.print(">>");
40:       System.out.flush();
41:       name = Input.getString();
42:     }
43:
44:     //アプリケーションが終了したことを知らせる
45:     System.out.println("アプリケーションが終了しました。");
46:   }
47:
48:   /**
49:    * 入力された文字が空かどうかを調べる
50:    */
51:   boolean isNoCharacter(String name) {
52:     if (name.length() != 0) { // 文字数が 0 ならば空である
53:       return true;
54:     } else {
55:       return false;
56:     }
57:   }
58:
59: }
```

5.2.3　手続きの階層構造

手続きを呼ぶ手続き

次に示すアプリケーション (リスト 34) は、3 章で作った、成績管理アプリケーションのいくつかの処理をメソッド化したものです。

このプログラムでは、メソッドがメソッドを呼んでいるところがあります。

考えてみよう

リスト 34 を読んで、メソッド呼び出しの様子をあなたが分かりやすいように図解してみよう。

リスト 34: 成績管理アプリケーション (メソッド化したもの)

```
 1: /**
 2:  * 成績を管理するアプリケーション
 3:  *
 4:  * ・名前と成績の登録（コマンド：A）
 5:  * ・成績一覧と平均表示（コマンド：V）
 6:  *
 7:  * 科目の成績を登録すると、上の動作をコマンドの選択によって行う
 8:  *
 9:  * @author Masahiro Kawamura
10:  * @version $Id: ScoreAdministratorApplication.java,v 1.6 2003/05/08 03:50:18 duskin Exp $
11:  */
12: public class ScoreAdministratorApplication {
13:
14:   public static void main(String[] args) {
15:     ScoreAdministratorApplication scoreAdministratorApplication =
16:       new ScoreAdministratorApplication();
17:     scoreAdministratorApplication.main();
18:   }
19:
20:   /*******************************************************
21:   /* メイン
22:   /*******************************************************/
23:
24:   void main() {
25:
26:     // 定数
27:     final int SCORE_SIZE = 10; // 入力できる成績の数
28:
29:     // 変数
30:     int[] scores = new int[SCORE_SIZE]; //全員の成績
31:     String[] names = new String[SCORE_SIZE]; //全員の名前
32:     String command; //入力されたコマンド
33:     int currentScoreIndex = 0; //現在登録されているデータの数
34:
35:     //アプリケーションの説明をする
36:     showTitle();
37:
38:     //メニューを出力し、コマンドで指定された処理を行う
39:     while (true) {
40:
41:       //メニューを出力し、コマンドを入力する
42:       command = getCommand();
43:
44:       //コマンドで指定された処理を行う
45:       if (command.equals("A")) { //成績を登録する
46:
47:         currentScoreIndex = addScore(scores, names, currentScoreIndex);
48:
49:       } else if (command.equals("V")) { //成績一覧と平均点を表示する
50:
51:         showScoreList(scores, names, currentScoreIndex);
52:
53:       } else if (command.equals("Q")) { //アプリケーションを終了する
```

```
54:
55:         break; //終了
56:
57:       } else { //エラー処理
58:
59:         showError();
60:       }
61:     }
62:
63:     //アプリケーションが終了したことを知らせる
64:     showEndTitle();
65:   }
66:
67:   /*******************************************************
68:   /* コマンド処理
69:   /*******************************************************/
70:
71:   /**
72:    * メニューを出力し、コマンドを入力する
73:    */
74:   String getCommand() {
75:
76:     String command; //入力されたコマンド
77:
78:     //メニューを出力し、コマンドを入力する
79:     System.out.println("コマンドを入力してください");
80:     System.out.println("(A:成績の登録，V:成績一覧と平均点の表示，Q:終了)");
81:     System.out.print(">>");
82:     System.out.flush();
83:     command = Input.getString();
84:     return command;
85:   }
86:
87:   /*******************************************************
88:   /* コマンド管理群
89:   /*******************************************************/
90:
91:   /**
92:    * 名前と成績を登録する
93:    * 登録人数を戻り値とする
94:    */
95:   int addScore(int[] scores, String[] names, int currentScoreIndex) {
96:
97:     String name; //入力された個人の名前
98:     int score; // 入力された個人の成績
99:
100:    //名前を入力する
101:    System.out.println("名前を入力してください");
102:    System.out.print(">>");
103:    System.out.flush();
104:    name = Input.getString();
105:
106:    //成績を入力する
107:    System.out.println("成績を入力してください");
```

```
108:     System.out.print(">>");
109:     System.out.flush();
110:     score = Input.getInt();
111:
112:     //データを登録する
113:     names[currentScoreIndex] = name;
114:     scores[currentScoreIndex] = score;
115:
116:     //正常に登録されたことを知らせる
117:     System.out.println(name + "さんの成績を登録しました");
118:
119:     //登録人数を増やす
120:     currentScoreIndex++;
121:
122:     return currentScoreIndex;
123:   }
124:
125:   /**
126:    * 成績一覧と平均点を表示する
127:    */
128:   void showScoreList(int[] scores, String[] names, int currentScoreIndex) {
129:
130:     double average; //平均点
131:     int i; //ループ用
132:
133:     //成績を一覧表示する
134:     System.out.println("★★★★★★成績一覧表★★★★★★");
135:     for (i = 0; i < currentScoreIndex; i++) {
136:       System.out.println(names[i] + "さん:" + scores[i] + "点");
137:     }
138:
139:     //平均点を表示する
140:     System.out.println("★★★★★★平均点★★★★★★");
141:     average = getAverageScore(scores, currentScoreIndex);
142:     System.out.println("平均点：" + average + "点");
143:   }
144:
145:   /**
146:    * エラーメッセージを表示する
147:    */
148:   void showError() {
149:     System.out.println("そのようなコマンドはありません");
150:   }
151:
152:   /******************************************************
153:   /* コマンド用部品
154:   /******************************************************/
155:
156:   /**
157:    * 平均点を計算する
158:    * 平均点を戻り値とする
159:    */
160:   double getAverageScore(int[] scores, int currentScoreIndex) {
161:
```

```
162:     double average; //平均点
163:     double total; //合計点
164:     int i; //ループ用
165:
166:     //平均点を計算する
167:     total = 0.0; //合計点を初期化する
168:     for (i = 0; i < currentScoreIndex; i++) { //合計点を計算する
169:       total = total + scores[i];
170:     }
171:     average = total / currentScoreIndex; //平均点を計算する
172:     return average;
173:   }
174:
175:   /*******************************************************
176:   /* タイトル表示
177:   /*******************************************************/
178:
179:   /**
180:    * アプリケーションの説明をする
181:    */
182:   void showTitle() {
183:     System.out.println("★★★★★★成績管理アプリケーション★★★★★★");
184:     System.out.println("（名前と成績の登録、成績一覧・平均の出力ができます。）");
185:   }
186:
187:   /**
188:    * アプリケーションが終了したことを知らせる
189:    */
190:   void showEndTitle() {
191:     System.out.println("アプリケーションが終了しました。");
192:   }
193: }
```

手続きの階層化

メソッドが呼ぶメソッドのことを俗にサブメソッドといい、さらにそのメソッドがよぶメソッドをサブサブメソッド、孫メソッドなどと呼ばれることもあります。どのようなプログラムでもトップレベルのメソッドは main() メソッドで、必ずそこからプログラムが始まります。

考えてみよう

HCP チャートとメソッドの階層構造について議論してみましょう。

5.3　共有される変数

5.3.1　インスタンス変数

前に紹介した成績管理アプリケーションでは、成績を記憶しておくための配列と名前を記憶しておくための配列、をメインメソッドの中で宣言していたので、個々のメソッドにこれらの配列を受け渡す必要がありました。

階層が上位のメソッドに配列を定義して、階層の下位のメソッドに徐々に受け渡すということを行うと、引数の数が多くなって、プログラムを書くにも読むにも大変です。

このような問題を解決するために、変数をメソッドブロック内で有効な**ローカル変数** [2]ではなく、クラス内ブロック内で有効な**インスタンス変数** [3]にして、変数を複数のメソッド間共有するという方法があります。

メソッドは、一番外側のブロックであるクラスブロックに宣言されています。変数をメソッドの中ではなく、メソッドの外のこのクラスブロックで宣言することで、クラス内の全てのメソッドで有効なインスタンス変数として宣言することが出来ます。

インスタンス変数を利用して、メソッド間で変数や配列を受け渡す手間が減る場合は、プログラムがシンプルになります。

リスト 35 では、成績管理アプリケーションを、インスタンス変数を導入して書き直してみました。

リスト 35: 成績管理アプリケーション（インスタンス変数を導入）

```
 1: /**
 2:  * 成績を管理するアプリケーション
 3:  *
 4:  * ・名前と成績の登録（コマンド：A）
 5:  * ・成績一覧と平均表示（コマンド：V）
 6:  * （インスタンス変数導入）
 7:  *
 8:  * 科目の成績を登録すると、上の動作をコマンドの選択によって行う
 9:  *
10:  * @author Masahiro Kawamura
11:  * @version $Id: ScoreAdministratorApplication.java,v 1.9 2003/05/08 11:43:47 duskin Exp $
12:  */
13: public class ScoreAdministratorApplication {
14:
15:   public static void main(String[] args) {
16:     ScoreAdministratorApplication scoreAdministratorApplication =
17:       new ScoreAdministratorApplication();
18:     scoreAdministratorApplication.main();
19:   }
```

[2] 実際はメソッドブロック内でも for 文や if 文のブロック内にもローカル変数が存在します。

[3] インスタンス変数という言葉の意味についてはオブジェクト指向編 7.1.2.1 節で詳しく解説します。ここではこういう名前であるということだけ覚えておいてください。

```
20:
21:   /*******************************************************
22:   /* インスタンス変数
23:   /*******************************************************/
24:
25:   // 定数
26:   final int SCORE_SIZE = 10; // 入力できる成績の数
27:
28:   // 変数
29:   int[] scores = new int[SCORE_SIZE]; //全員の成績
30:   String[] names = new String[SCORE_SIZE]; //全員の名前
31:   int currentScoreIndex = 0; //現在登録されているデータの数
32:
33:
34:   /*******************************************************
35:   /* メイン
36:   /*******************************************************/
37:
38:   void main() {
39:
40:     //ローカル変数
41:     String command; //入力されたコマンド
42:
43:     //アプリケーションの説明をする
44:     showTitle();
45:
46:     //メニューを出力し、コマンドで指定された処理を行う
47:     while (true) {
48:
49:       //メニューを出力し、コマンドを入力する
50:       command = getCommand();
51:
52:       //コマンドで指定された処理を行う
53:       if (command.equals("A")) { //成績を登録する
54:
55:         currentScoreIndex = addScore();
56:
57:       } else if (command.equals("V")) { //成績一覧と平均点を表示する
58:
59:         showScoreList();
60:
61:       } else if (command.equals("Q")) { //アプリケーションを終了する
62:
63:         break; //終了
64:
65:       } else { //エラー処理
66:
67:         showError();
68:       }
69:     }
70:
71:     //アプリケーションが終了したことを知らせる
72:     showEndTitle();
73:   }
```

```
 74:
 75:   /*******************************************************
 76:   /* コマンド処理
 77:   /*******************************************************/
 78:
 79:   /**
 80:    * メニューを出力し、コマンドを入力する
 81:    */
 82:   String getCommand() {
 83:
 84:     //ローカル変数
 85:     String command; //入力されたコマンド
 86:
 87:     //メニューを出力し、コマンドを入力する
 88:     System.out.println("コマンドを入力してください");
 89:     System.out.println("(A:成績の登録，V:成績一覧と平均点の表示，Q:終了)");
 90:     System.out.print(">>");
 91:     System.out.flush();
 92:     command = Input.getString();
 93:     return command;
 94:   }
 95:
 96:   /*******************************************************
 97:   /* コマンド管理群
 98:   /*******************************************************/
 99:
100:   /**
101:    * 名前と成績を登録する
102:    * 登録人数を戻り値とする
103:    */
104:   int addScore() {
105:
106:     //ローカル変数
107:     String name; //入力された個人の名前
108:     int score; // 入力された個人の成績
109:
110:     //名前を入力する
111:     System.out.println("名前を入力してください");
112:     System.out.print(">>");
113:     System.out.flush();
114:     name = Input.getString();
115:
116:     //成績を入力する
117:     System.out.println("成績を入力してください");
118:     System.out.print(">>");
119:     System.out.flush();
120:     score = Input.getInt();
121:
122:     //データを登録する
123:     names[currentScoreIndex] = name;
124:     scores[currentScoreIndex] = score;
125:
126:     //正常に登録されたことを知らせる
127:     System.out.println(name + "さんの成績を登録しました");
```

```
128:
129:     //登録人数を増やす
130:     currentScoreIndex++;
131:
132:     return currentScoreIndex;
133:   }
134:
135:   /**
136:    * 成績一覧と平均点を表示する
137:    */
138:   void showScoreList() {
139:
140:     //ローカル変数
141:     double average; //平均点
142:     int i; //ループ用
143:
144:     //成績を一覧表示する
145:     System.out.println("★★★★★★成績一覧表★★★★★★");
146:     for (i = 0; i < currentScoreIndex; i++) {
147:       System.out.println(names[i] + "さん:" + scores[i] + "点");
148:     }
149:
150:     //平均点を表示する
151:     System.out.println("★★★★★★平均点★★★★★★");
152:     average = getAverageScore();
153:     System.out.println("平均点：" + average + "点");
154:   }
155:
156:   /**
157:    * エラーメッセージを表示する
158:    */
159:   void showError() {
160:     System.out.println("そのようなコマンドはありません");
161:   }
162:
163:   /******************************************************
164:   /* コマンド用部品
165:   /******************************************************/
166:
167:   /**
168:    * 平均点を計算する
169:    * 平均点を戻り値とする
170:    */
171:   double getAverageScore() {
172:
173:     //ローカル変数
174:     double average; //平均点
175:     double total; //合計点
176:     int i; //ループ用
177:
178:     //平均点を計算する
179:     total = 0.0; //合計点を初期化する
180:     for (i = 0; i < currentScoreIndex; i++) { //合計点を計算する
181:       total = total + scores[i];
```

```
182:     }
183:     average = total / currentScoreIndex; //平均点を計算する
184:     return average;
185:   }
186:
187:   /*******************************************************
188:   /* タイトル表示
189:   /*******************************************************/
190:
191:   /**
192:    * アプリケーションの説明をする
193:    */
194:   void showTitle() {
195:     System.out.println("★★★★★★成績管理アプリケーション★★★★★★");
196:     System.out.println("（名前と成績の登録、成績一覧・平均の出力ができます。）");
197:   }
198:
199:   /**
200:    * アプリケーションが終了したことを知らせる
201:    */
202:   void showEndTitle() {
203:     System.out.println("アプリケーションが終了しました。");
204:   }
205: }
```

5.4 練習問題

練習問題 1

第 1 章の練習問題で使った健康を管理するプログラム（HealthCareSample.java）をメソッドを使って書き直してください。

練習問題 2

パスワードをチェックするプログラム（CheckPasswordSample.java）のパスワードチェック部分の条件式をメソッド化してみてください。

できたら、メソッド化前のものと比較して、メソッド化するべきかどうか議論してみましょう。

練習問題 3

前回の練習問題で作った電話帳アプリケーションをメソッドを使って書き直してください。

第 6 章

プログラムの効率

この章で学習すること

検索とソートの代表的なアルゴリズムについて説明できる
アルゴリズムの性能について比較検討し、説明できる
再帰を使ったプログラムが書ける

6.1 検索アルゴリズムの効率

6.1.1 リニアサーチ

本章では、前回までのプログラム作成技法を利用して、様々なアルゴリズムのプログラミングに挑戦してみましょう。

まず、前章までで何度か出てきた配列の内容の検索とはどういうものだったかを考えながら、前回までのプログラムのアルゴリズムを使って書いた商品検索プログラム (リスト 36) を見てみましょう。

リスト 36: 商品を番号で検索するサンプルプログラム（リニアサーチ）

```
 1: /**
 2:  * 商品番号を検索するサンプルプログラム
 3:  * （リニアサーチ）
 4:  *
 5:  * @author kawam
 6:  * @version $Id: ItemListSample.java,v 1.12 2003/05/07 05:27:16 duskin Exp $
 7:  */
 8: public class ItemListSample {
 9:
10:   public static void main(String args[]) {
11:     ItemListSample itemListSample = new ItemListSample();
12:     itemListSample.main();
13:   }
14:
```

```
15:   // 定数
16:   final int LIMITED_ITEM_NUMBER = 10000000; //商品の最大限界個数
17:
18:   // 変数
19:   int[] itemNumber = new int[LIMITED_ITEM_NUMBER]; //商品番号
20:   int maxItemNumber; //商品の登録個数
21:
22:   void main() {
23:
24:     // 登録する商品数を入力する
25:     while (true) {
26:       System.out.println("登録する商品数を入力してください");
27:       System.out.print(">>");
28:       System.out.flush();
29:       maxItemNumber = Input.getInt();
30:       if (maxItemNumber <= LIMITED_ITEM_NUMBER) {
31:         break;
32:       } else { // 最大登録可能個数を上回った場合は入力しなおし
33:         System.out.println("最大" + LIMITED_ITEM_NUMBER + "までです");
34:       }
35:     }
36:
37:     // 商品を登録する
38:     for (int itemNo = 0; itemNo < LIMITED_ITEM_NUMBER; itemNo++) {
39:       addItem(itemNo);
40:     }
41:
42:     // 検索キーを入力する
43:     System.out.println("検索したい商品番号を入力してください");
44:     System.out.print(">>");
45:     System.out.flush();
46:     int searchKey = Input.getInt();
47:
48:     // 商品を検索する
49:     boolean isFound = linearSearch(searchKey);
50:
51:     //検索結果を表示する
52:     if (isFound) { //見つかったとき
53:       System.out.println("商品" + searchKey + "は見つかりました");
54:     } else { //見つからなかったとき
55:       System.out.println("商品" + searchKey + "は見つかりませんでした");
56:     }
57:
58:   }
59:
60:   /**
61:    * 商品をランダムに追加する
62:    */
63:   void addItem(int itemIndex) {
64:     int itemNum = (int) (maxItemNumber * 10 * Math.random());
65:     itemNumber[itemIndex] = itemNum;
66:   }
67:
68:   /**
```

```
69:    * 商品を検索する
70:    */
71:   boolean linearSearch(int searchKey) {
72:     for (int i = 0; i < LIMITED_ITEM_NUMBER; i++) {
73:       if (searchKey == itemNumber[i]) { //一致するものが見つかったとき
74:         return true;
75:       }
76:     }
77:     //一致するものが見つからなかったとき
78:     return false;
79:   }
80: }
```

検索を行っているのは linearSerach というメソッドです。このメソッドは検索対象が見つかったときには true を、見つからなかったときには false を戻り値とするメソッドです。今までと同様に for 文の繰り返しを用いて、配列の先頭から順番に比較していくことで、検索しています。

このように、配列の先頭から順番に内容を調べてゆき、検索対象と一致するものが見つかるまで調べるというアルゴリズムのことを**リニアサーチ (linear search)** といいます。

考えてみよう

リニアサーチの HCP チャートを書いてみよう。

6.1.2 バイナリサーチ

検索のアルゴリズムにはリニアサーチ以外に**バイナリサーチ (binary search)** というのもあります。そのアルゴリズムは以下のとおりです。

> アルゴリズム
> バイナリサーチではリニアサーチと違って、配列の中身が番号順に並んでいることが前提となっています。バイナリサーチは、まず配列の中央の内容に着目し、それが検索対象より大きいか小さいかを調べます。
> **検索対象が中央よりも大きい場合**　中央から最後までの間の中心の内容と検索対象を比較する
> **検索対象が中央よりも小さい場合**　先頭から中央までの間の中心の内容と検索対象を比較する
> 一回の比較で検索対象がある可能性のある範囲が半分に絞り込まれています。あとはこれを何度も繰り返して範囲を狭めれば、最後に目的のものを見つけることができます。

先ほどの商品を番号で検索するサンプルプログラムの linearSearch メソッドをバイナリサーチのアルゴリズムで書き換えたものがリスト 37 です。

リスト 37: 商品を番号で検索するサンプルプログラム（バイナリサーチ）

```
75:   boolean binarySearch(int searchKey) {
76:
77:     int lower = 0; //探す範囲の最小値
78:     int upper = LIMITED_ITEM_NUMBER - 1; //探す範囲の最大値
79:     int center; //次に調べる配列の番地
80:
81:     while (true) {
82:       center = (lower + upper) / 2; //次に調べ始める番地を範囲の中央に設定
83:       if (itemNumber[center] == searchKey) {
84:         return true;
85:       } else if (lower > upper) {
86:         return false;
87:       } else { //見つからないので、次に調べる範囲を狭める
88:         if (itemNumber[center] < searchKey) {
89:           lower = center + 1; //範囲を上半分にする
90:         } else {
91:           upper = center - 1; //範囲を下半分にする
92:         }
93:       }
94:     }
95:   }
```

考えてみよう

バイナリサーチの HCP チャートを書いてみよう。

6.1.3 効率の比較

検索のアルゴリズムとして二通りを紹介しました。これらのアルゴリズムを使い分けるために、効率を比較していきます。

検索のアルゴリズムの効率は、検索対象と配列の内容と比較がどのくらい行われるかで比較することができます。比較の回数が多ければ、その分コンピュータの計算回数が増えるので、処理速度が遅くなります。

それぞれの検索アルゴリズムにおいて、検索対象と配列の内容と比較回数を調べていきましょう。

リニアサーチ
配列の要素数が N 個であった場合、もし探しているものが配列の先頭にあった場合比較は 1 回で済みますが、配列の最後にあった場合は N 個全ての要素との比較が行われることになります。
つまり普通の場合はだいたい両者の平均である (N+1) /2 回であると考えることができます。
ここでもしも配列の要素数 N が 2 倍に増えたら、比較の回数も同様に 2 倍になり、N が 3 倍になれば、比較する回数も 3 倍になる、というように N に比例して増えていきます。
バイナリサーチ
もし探しているものが中央にあった場合は 1 回で済みます。バイナリサーチは、一度比較が行われる毎に比較の範囲が半分に減っていくので、最大でも配列の要素数が N 個であった場合、 平均すると比較の回数は log 2N 回となります。
配列の要素数 N が 2 倍になったら、比較の回数は log2(2) 倍となり、3 倍になったら、比較の回数は log2(3) 倍となるので、log2 N に比例して増えることになります。

BigO 記法

BigO 記法とは、アルゴリズムの効率を表すための記法で、その名の通り大文字の O を使います。O は order(次数) の意味です。

例えばリニアサーチの場合、配列の要素数を N としたとき、検索の効率は要素数 N に比例しているので、BigO 記法では O(N) と表現され、その効率は N のオーダーであるといいます。

厳密には (N+1)/2 回ですが、N が非常に大きい数字のとき、分子の+1 や分母の 2 は相対的に小さくなり影響がなくなるので，BigO 記法ではこのような定数は省略されます。

同様にバイナリサーチの場合、検索の効率は log2 N に比例しているので、BigO 記法では O(logN) と表現されます。

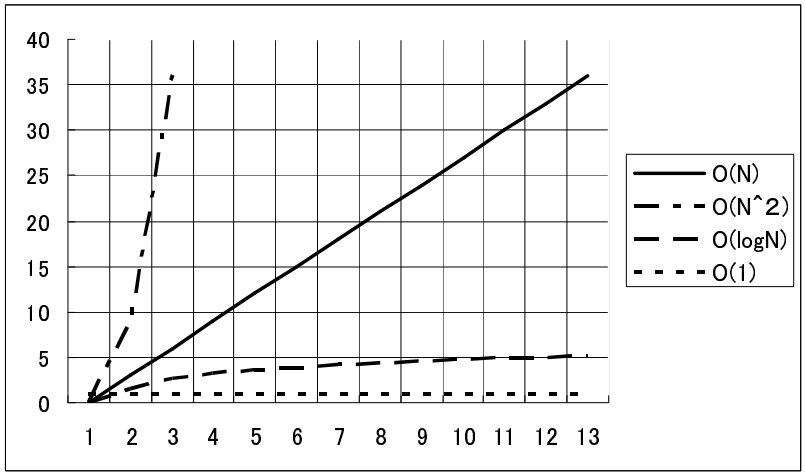


図 6.1: 増加の速度

6.2 並び替えアルゴリズムの効率

リニアサーチは既に小さい順に並び替えが済んでいることが前提となっていました。ではこの要素の並び替えはどのような手順で行えばよいでしょうか。検索同様、並び替え（ソート）のアルゴリズムにも様々なものがあります。

ここでは、基本的なアルゴリズムを紹介します。

6.2.1 バブルソート

アルゴリズム

- 隣り合う 2 つの要素を比較する
- 左側のほうが大きければ入れ替える
- 右へ一つ移動する
- 並べ替えの終わっていない部分（毎回小さくなる）に対して、同じステップを繰り返す

バブルソートのプログラムをリスト 38 に示します。

リスト 38: 商品を商品番号順に並び替えるサンプルプログラム（バブルソート）

```
 1: /**
 2:  * 商品を番号順に並べ替えるサンプルプログラム
 3:  * （バブルソート）
 4:  *
 5:  * @author kawam
 6:  * @version $Id: ItemListSample.java,v 1.9 2003/05/07 05:27:17 duskin Exp $
 7:  */
 8: public class ItemListSample {
 9:
10:   public static void main(String args[]) {
11:     ItemListSample itemListSample = new ItemListSample();
12:     itemListSample.main();
13:   }
14:
15:   // 定数
16:   final int LIMITED_ITEM_NUMBER = 10000000; //商品の最大登録可能個数
17:
18:   // 変数
19:   int[] itemNumber = new int[LIMITED_ITEM_NUMBER]; //商品番号
20:   int maxItemNumber; //商品の登録個数
21:
22:   void main() {
23:
24:     // 登録する商品数を入力する
25:     while (true) {
26:       System.out.println("登録する商品数を入力してください");
27:       System.out.print(">>");
```

```
28:       System.out.flush();
29:       maxItemNumber = Input.getInt();
30:       if (maxItemNumber <= LIMITED_ITEM_NUMBER) {
31:         break;
32:       } else { // 最大登録可能個数を上回った場合は入力しなおし
33:         System.out.println("最大" + LIMITED_ITEM_NUMBER + "までです");
34:       }
35:     }
36:
37:     //商品を登録する
38:     System.out.println("商品を" + maxItemNumber + "個登録します");
39:     for (int i = 0; i < maxItemNumber; i++) {
40:       addItem(i);
41:     }
42:
43:     //商品を番号順に並べ替える
44:     System.out.println("並べ替えを開始します");
45:     bubbleSort(); //並べ替え
46:     System.out.println("並べ替え完了");
47:   }
48:
49:   /**
50:    * 商品を番号順に並べ替える
51:    */
52:   void bubbleSort() {
53:     for (int i = maxItemNumber - 1; i > 1; i--) {
54:       for (int j = 0; j < i; j++) {
55:         if (itemNumber[j] > itemNumber[j + 1]) { // 小さいものが後ろにあったら
56:           swap(j, j + 1); // j 番目と j+1 番目を入れ替える
57:         }
58:       }
59:     }
60:   }
61:
62:   /**
63:    * 隣り合う二つの商品の順番を入れ替える
64:    */
65:   void swap(int i, int j) {
66:     int tmp = i;
67:     itemNumber[i] = j;
68:     itemNumber[j] = tmp;
69:   }
70:
71:   /**
72:    * 商品をランダムに追加する
73:    */
74:   void addItem(int itemIndex) {
75:     int itemNum = (int) (maxItemNumber * 10 * Math.random());
76:     itemNumber[itemIndex] = itemNum;
77:   }
78: }
```

バブルソートの効率

比較の回数は、最初のループで N-1 回、次のループで N-2 回というふうになり最後のループは 1 回なので、 全部で N-1 + N-2 + ⋯ + 2 + 1 = N(N-1)/2 回となり、N の二乗に比例します。また入れ替えは、大きい数字が小さい数字の前に来たときだけ起きるので、その確率は 1/2 と考え、 回数は (N/2) 2 = N^2/4 回となります。いずれも N^2 のオーダーなので BigO 記法では O(N^2) となります。

考えてみよう

バブルソートの HCP チャートを書いてみよう。

6.2.2 選択ソート

アルゴリズム

- 全ての要素の中から一番小さいものを探す
- 見つかった最小のものと一番左の要素を入れ替える
- 並べ替えの終わっていない部分（毎回小さくなる）に対して、同じステップを繰り返す

選択ソートのプログラムをリスト 39 に示します。

リスト 39: 商品を商品番号順に並び替えるサンプルプログラムのソート部分（選択ソート）

```
54:   void selectionSort() {
55:
56:     int min = 0; //番号が最小のもののインデックス
57:
58:     for (int i = 0; i < maxItemNumber - 1; i++) {
59:       for (int j = i + 1; j < maxItemNumber; j++) {
60:         if (itemNumber[j] < itemNumber[min]) {
61:           min = j;
62:         }
63:         swap(i, min); // 最小のものと左端とを入れ替える
64:       }
65:     }
66:   }
```

選択ソートの効率

比較の回数はバブルソートと同じ N(N-1)/2 回ですが、入れ替えの回数は N よりも小さくなります。 オーダーは N 2 でバブルソートと同じですが、入れ替えの回数が少ないのでバブルソートよりも効率がよくなります。

考えてみよう

選択ソートの HCP チャートを書いてみよう。

6.2.3 挿入ソート

アルゴリズム

- 途中までソート済みとする
- ソートが終わっていない中で、一番左にある要素をソート済みの配列の中のあるべき位置に挿入する
- 並べ替えの終わっていない部分（毎回小さくなる）に対して、同じステップを繰り返す

挿入ソートのプログラムをリスト 40 に示します。

リスト 40: 商品を商品番号順に並び替えるサンプルプログラムのソート部分（挿入ソート）

```
52:   void insertionSort() {
53:     for (int markPoint = 1; markPoint < maxItemNumber; markPoint++) {
54:       int temp = itemNumber[markPoint]; //注目している項目を一時退避
55:       int i = markPoint;
56:       while (i > 0
57:         && itemNumber[markPoint] >= temp) {
58:         //注目している項目より小さいものが現れるまで，一つずつ右にずらす
59:         itemNumber[i] = itemNumber[i - 1];
60:         i--;
61:       }
62:       itemNumber[i] = temp; // 注目していた項目を挿入する
63:     }
64:   }
```

挿入ソートの効率

比較の回数は、最初の最初は 1 回、次は最大で 2 回、…、最後の場合も最大で N-1 回となりますので、 最大でも 1 + 2 + 3 + ⋯ + N-2 + N-1 = N(N-1)/2 回となります．毎回の平均はその半分となるので、N(N-1)/4 回となり，N^2 のオーダー、すなわち O(N^2) となります。

しかし、挿入ソートの場合は入れ替えは一度も行われず、コピーのみが行われるので、普通の場合は選択ソートよりも効率がよくなります。

考えてみよう

挿入ソートの HCP チャートを書いてみよう。

6.3　手続きの再帰呼び出し

6.3.1　プログラムの実行と再帰

指定した数（n）の階乗を計算するプログラムを考えてみましょう。階乗は 1 からその値 (n) までを掛け合わせた数ですから、繰り返しを用いて書くことができます。しかし n の階乗は n-1 の階乗の結果に n を掛け合わせたものだとも言えます。このことを利用して次のようなプログラムを作ることができます。

リスト 41: 階乗計算アプリケーション

```
 1: /**
 2:  * 指定された数の階乗を計算するアプリケーション
 3:  *
 4:  * @author duskin
 5:  * @version $Id: FactorialCalculatorApplication.java,v 1.8 2003/05/07 05:31:59 duskin Exp $
 6:  */
 7: public class FactorialCalculatorApplication {
 8:
 9:   public static void main(String args[]) {
10:     FactorialCalculatorApplication factorialCalculatorApplication =
11:       new FactorialCalculatorApplication();
12:     factorialCalculatorApplication.main();
13:   }
14:
15:   void main() {
16:
17:     int inputNumber = 0; //入力された数値
18:     int factorial; //階乗の結果
19:
20:     // アプリケーションの説明を表示する
21:     System.out.println("指定された数の階乗を計算します。");
22:
23:     // 階乗計算を繰り返す
24:     while (inputNumber < 0) {
25:
26:       // 階乗を計算したい数を入力する
27:       System.out.print("階乗を計算したい数を入力してください （負の数を入力すると終了）
>>");
28:       System.out.flush();
29:       inputNumber = Input.getInt();
30:       if (inputNumber < 0) { //負の数を入力すると終了
31:         break;
32:       }
33:
34:       //階乗を計算する
35:       factorial = calculateFactorial(inputNumber);
36:
37:       //計算結果を表示する
38:       System.out.println(inputNumber + "の階乗は " + factorial + "です");
```

```
39:       //結果を表示する
40:     }
41:
42:     // アプリケーションの終了を伝える
43:     System.out.println("アプリケーションが終了しました。");
44:   }
45:
46:   /**
47:    * 階乗を計算する
48:    */
49:   int calculateFactorial(int number) {
50:     if (number == 0) { //求める階乗が 0 なら
51:       return 1; //答えは１
52:     } else { //0 でなければ
53:       return number * calculateFactorial(number - 1); //再帰的に自分自身を呼ぶ
54:     }
55:   }
56: }
```

階乗を計算している calculateFactorial メソッドに注目してください。このメソッドでは与えられた数値を一つ小さい数の階乗に掛け合わせることしかやっていません。一つ小さい数の階乗は、自分自身を呼び出すことで求めています。

あとはひたすら自分自身を呼び続け、求める階乗が 0 になったとき、0 の階乗は 1 であると戻り値を返すようになっています。

このように手続きが自分自身を呼び出すことを再帰と言います。

自分自身のメソッドを再帰的に呼び出す場合でも今までと同様に、呼び出しの度に新しい表が生まれます。

考えてみよう

リスト 41 の calclateFactrial メソッドの実行の様子を表モデルを書いて調べてみよう。

6.3.2 マージソート

再帰を用いた並べ替えのアルゴリズムとしてマージソート (merge sort) というものがあります。

アルゴリズム
- 要素が 1 つになるまで配列を 2 分割し続ける
- 隣り合う要素同士を比較して大きいほうを右にする
- 分割された 2 つの配列をマージする

マージソートのプログラムをリスト 42 に示します。

リスト 42: 商品を商品番号順に並び替えるサンプルプログラム（マージソート）

```
 1: /**
 2:  * 商品を番号順に並べ替えるサンプルプログラム
 3:  * （マージソート）
 4:  *
 5:  * @author kawam
 6:  * @version $Id: ItemListSample.java,v 1.10 2003/05/07 05:27:16 duskin Exp $
 7:  */
 8: public class ItemListSample {
 9:
10:   public static void main(String args[]) {
11:     ItemListSample itemListSample = new ItemListSample();
12:     itemListSample.main();
13:   }
14:
15:   // 定数
16:   final int LIMITED_ITEM_NUMBER = 10000000; //商品の最大登録可能個数
17:
18:   // 変数
19:   int[] itemNumber = new int[LIMITED_ITEM_NUMBER]; //商品番号
20:   int maxItemNumber; //商品の登録個数
21:
22:   void main() {
23:
24:     // 登録する商品数を入力する
25:     while (true) {
26:       System.out.println("登録する商品数を入力してください");
27:       System.out.print(">>");
28:       System.out.flush();
29:       maxItemNumber = Input.getInt();
30:       if (maxItemNumber <= LIMITED_ITEM_NUMBER) {
31:         break;
32:       } else { // 最大登録可能個数を上回った場合は入力しなおし
33:         System.out.println("最大" + LIMITED_ITEM_NUMBER + "までです");
34:       }
35:     }
36:
```

```
37:     //商品を登録する
38:     System.out.println("商品を" + maxItemNumber + "個登録します");
39:     for (int i = 0; i < maxItemNumber; i++) {
40:       addItem(i);
41:     }
42:
43:     //商品を番号順に並べ替える
44:     System.out.println("並べ替えを開始します");
45:     mergeSort(0, maxItemNumber); //並べ替え
46:     System.out.println("並べ替え完了");
47:   }
48:
49:   /**
50:    * 商品を番号順に並べ替える
51:    */
52:   void mergeSort(int startIndex, int endIndex) {
53:     if (startIndex == endIndex) { // 要素の数がひとつの場合は並べ替え終了
54:       return;
55:     } else {
56:       // 配列を前後で分ける
57:       int middleIndex = (startIndex + endIndex) / 2;
58:
59:       // 分けたものをそれぞれソートする
60:       mergeSort(startIndex, middleIndex); // 前半
61:       mergeSort(middleIndex + 1, endIndex); // 後半
62:
63:       // 両者をマージする
64:       merge(startIndex, middleIndex + 1, endIndex);
65:     }
66:   }
67:
68:   /**
69:    * 二つの配列をマージする
70:    */
71:   void merge(int startIndex, int middleIndex, int endIndex) {
72:
73:     int[] workspace = new int[endIndex - startIndex + 1];// 並べ替えのために内容を一
時的に記憶する配列
74:     int workspaceIndexPointer = 0; // 一時配列のポインタのスタートの位置
75:     int lowerIndexPointer = startIndex;
76:     int upperIndexPointer = middleIndex;
77:
78:     // 2 つの配列の先頭同士を比較し，小さいものを前に入れる
79:     while (lowerIndexPointer < middleIndex
80:       && upperIndexPointer <= endIndex) {
81:       if (itemNumber[lowerIndexPointer]
82:         < itemNumber[upperIndexPointer]) {
83:         workspace[workspaceIndexPointer++] =
84:           itemNumber[lowerIndexPointer++];
85:       } else {
86:         workspace[workspaceIndexPointer++] =
87:           itemNumber[upperIndexPointer++];
88:       }
89:     }
```

```
 90:
 91:     // 一方の配列のポインタが最後まで行った後，他方の配列の残りをコピーする
 92:     while (lowerIndexPointer < middleIndex) {
 93:       workspace[workspaceIndexPointer++] =
 94:         itemNumber[lowerIndexPointer++];
 95:     }
 96:     while (upperIndexPointer <= endIndex) {
 97:       workspace[workspaceIndexPointer++] =
 98:         itemNumber[upperIndexPointer++];
 99:     }
100:
101:     // 一時配列から元の配列に内容をコピーする
102:     for (int i = startIndex; i <= endIndex; i++) {
103:       itemNumber[i] = workspace[i - startIndex];
104:     }
105:   }
106:
107:   /**
108:    * 商品をランダムに追加する
109:    */
110:   void addItem(int itemIndex) {
111:     int itemNum = (int) (maxItemNumber * 10 * Math.random());
112:     itemNumber[itemIndex] = itemNum;
113:   }
114: }
```

マージソートの効率

マージソートの作業スペースのコピー回数、コピーの合計回数、比較の回数は以下のようになります。

表 6.1: マージソートの効率

N	作業スペースへのコピー回数	コピーの合計回数	比較回数（最大：最小）	log2N
2	2	4	1:1	1
4	8	16	5:4	2
8	24	48	17:12	3
16	64	128	49:32	4
32	160	320	129:80	5

マージソートでは比較の回数はコピーの回数より常に少なくなります。コピーの回数は Nlog2 N に比例しているので、オーダーは NlogN で、O(NlogN) となります。

考えてみよう

マージソートの HCP チャートを書いてみよう。

6.4　練習問題

練習問題 1

本章で紹介した各サーチおよびソートのアルゴリズムの処理時間を測定し、考察してください。

処理時間を計測するには、その処理の直前と直後の時刻を記憶し、その差分を求めればよいので次のようになります。

リスト 43: 処理時間の計測方法

```
long startTime; //開始した時間を保存する変数
long stopTime; //停止した時間を保存する変数

startTime = System.currentTimeMillis(); //開始した時間を保存しておく
（ここにサーチ / ソートの処理を書く）
stopTime = System.currentTimeMillis(); //停止した時間を保存しておく

System.out.println("処理にかかった時間は" + (stopTime - startTime) + "ミリ秒です。") //かかった時間を表示する
```

この stopTime と startTime の差が処理時間（単位はミリ秒）になります。

練習問題 2

1 から 4 までの数字の合計が 10、というように、1 からランダムに表示される数までの合計をユーザに計算させて入力してもらい、間違えるまで次々と質問し続けるクイズプログラムを作成してください。

例えば、4 と表示されたならば 10、5 と表示されたならば 15 を入力すると正解となって次の問題が出題されます。

ランダムに数字を表示させるには乱数を用います。Math.random() で 0 以上 1 未満の任意の小数（double 型の値）がランダムに求められます。たとえば 1 から 7 までの整数の乱数ならば以下のように Math.random を 7 倍したものを整数型にキャストし 1 を加えることで求められます。

```
int randomNumber = (int) (Math.random() * 7) + 1;
```

第 II 部

オブジェクト指向プログラミング編

第 7 章

オブジェクトとしての抽象化 (1)

この章で学習すること

クラスを使うことの利点を説明できる
クラスとオブジェクトの違いを説明できる
インスタンスの関係をインスタンスの入れ子モデルで図に書くことができる

7.1 クラスとインスタンス

人は複雑なものを一度に扱うことができません。複雑なプログラムを抽象化して、人が一度に理解できるようにするために、構造化プログラミング編では変数、配列、手続きといった手段をつかってきました。オブジェクト指向編では、新たに**クラス**と**オブジェクト**という考え方を用いてプログラムを抽象化し、人にやさしいプログラムを目指していきます。

7.1.1 変数のまとまりをクラスに

リスト 44: 社員名簿アプリケーション

```
 1: import java.io.PrintStream;
 2:
 3: /**
 4:  * 社員名簿管理アプリケーション
 5:  *
 6:  * 本アプリケーションの機能
 7:  * ○名簿の管理
 8:  *       ・社員の追加
 9:  *       ・社員の削除
10:  *       ・社員の編集
11:  * ○名簿の閲覧
12:  *       ・カーソルの移動
13:  * ○名簿の保存
```

```
14:  *     ・セーブ
15:  *     ・ロード
16:  *
17:  * @author bam
18:  * @version $Id: EmployeeDirectoryApplication.java,v 1.3 2003/05/04 12:10:30 macchan Exp $
19:  */
20: public class EmployeeDirectoryApplication {
21:
22:   public static void main(String[] args) {
23:     EmployeeDirectoryApplication employeeDirectoryApplication =
24:       new EmployeeDirectoryApplication();
25:     employeeDirectoryApplication.main();
26:   }
27:
28:   /*****************************************************
29:   /* 定数
30:   /*****************************************************/
31:
32:   //  保存できる最大データの数
33:   final int RECORD_MAX = 20;
34:
35:   //コマンド類
36:   final String ADD = "A"; //データの追加
37:   final String REMOVE = "R"; //データの削除
38:   final String EDIT = "E"; //データの編集
39:   final String UP_CURSOR = "U"; //カーソルを上へ
40:   final String DOWN_CURSOR = "D"; //カーソルを下へ
41:   final String SAVE = "S"; //セーブ
42:   final String LOAD = "L"; //ロード
43:   final String QUIT = "Q"; //終了
44:   final String[] COMMANDS =
45:     { ADD, REMOVE, EDIT, UP_CURSOR, DOWN_CURSOR, SAVE, LOAD, QUIT };
46:
47:   //表示文字列類
48:   final String LINE =
49:     "-----------------------------------------------------------------------";
50:   final String SPACE = "  ";
51:   final String SC = ":";
52:   final String ADD_MSG = ADD + SC + "追加" + SPACE;
53:   final String RM_MSG = REMOVE + SC + "削除" + SPACE;
54:   final String EDIT_MSG = EDIT + SC + "編集" + SPACE;
55:   final String UC_MSG = UP_CURSOR + SC + "前のデータへ" + SPACE;
56:   final String DC_MSG = DOWN_CURSOR + SC + "次のデータへ" + SPACE;
57:   final String SAVE_MSG = SAVE + SC + "セーブ" + SPACE;
58:   final String LOAD_MSG = LOAD + SC + "ロード" + SPACE;
59:   final String QUIT_MSG = QUIT + SC + "終了" + SPACE;
60:   final String COMMAND_INFO_MESSAGE =
61:     "コマンド"
62:       + "("
63:       + SPACE
64:       + ADD_MSG
65:       + RM_MSG
66:       + EDIT_MSG
67:       + UC_MSG
```

```
68:       + DC_MSG
69:       + SAVE_MSG
70:       + LOAD_MSG
71:       + QUIT_MSG
72:       + ")";
73:   final String COMMAND_ERROR_MESSAGE = "不適当なコマンドでした";
74:   final String INCORRECT_INPUT_ERROR_MESSAGE = "数字を入力してください";
75:   final String COMMAND_PROMPT = " >> ";
76:   final String REMOVE_CONFIRM_MESSAGE = "本当に削除してもいいですか？ y/n";
77:   final String NO_EDITABLE_ELEMENT_MESSAGE = "編集可能なデータがありません";
78:   final String DATA_OVERFLOW_ERROR_MESSAGE = "データがいっぱいで、もう追加できません";
79:   final String COMPANY_NAME = "○×コミュニケーションズ";
80:   final String YES = "Y";
81:   final String SEPARATE_LINE = "\n"; //改行記号
82:
83:   //カーソル移動用
84:   final int UP = 1;
85:   final int DOWN = 2;
86:
87:   /*****************************************************
88:   /* インスタンス変数
89:   /*****************************************************/
90:
91:   int currentRecordIndex = -1; //現在カーソルがあるインデックス。何もないときは-1
92:   int recordCount = 0; //現在の要素数
93:
94:   //データを保存する配列群
95:   int id[] = new int[RECORD_MAX]; //社員 ID
96:   String section[] = new String[RECORD_MAX]; //部署
97:   String name[] = new String[RECORD_MAX]; //漢字の名前
98:   String address[] = new String[RECORD_MAX]; //住所
99:   int birthYear[] = new int[RECORD_MAX]; //誕生年
100:  int age[] = new int[RECORD_MAX]; //年齢
101:
102:  /*****************************************************
103:  /* メイン
104:  /*****************************************************/
105:
106:  void main() {
107:
108:    String command; //入力されたコマンド
109:
110:    //初期画面を表示
111:    updateView();
112:
113:    //コマンドの入力と実行
114:    while (!(command = inputCommand()).equals(QUIT)) {
115:      executeCommand(command);
116:      updateView();
117:    }
118:
119:    //終了処理
120:    System.exit(0);
121:  }
```

```
122:
123:   /*****************************************************
124:   /* コマンド処理
125:   /*****************************************************/
126:
127:   /**
128:    * 社員名簿のコマンドを入力し、妥当なコマンドであれば、入力されたコマンドを返す。
129:    * もしも、不適当なコマンドであれば、再入力を要求する。
130:    */
131:   String inputCommand() {
132:
133:     String command; //入力されたコマンド
134:
135:     while (true) {
136:       //プロンプトを出す
137:       showPrompt(COMMAND_INFO_MESSAGE);
138:
139:       //入力する
140:       command = Input.getString();
141:       command = command.toUpperCase(); //入力されたコマンドを大文字に
142:
143:       //正しいコマンドならばループを抜けて入力コマンドを返す。
144:       //正しくなければエラーを表示して再入力
145:       if (isCorrectCommand(command)) {
146:         break;
147:       } else {
148:         showError(COMMAND_ERROR_MESSAGE);
149:       }
150:     }
151:     return command;
152:   }
153:
154:   /**
155:    * 指定された社員名簿のコマンドを実行する
156:    */
157:   void executeCommand(String command) {
158:     if (command.equals(ADD)) { //追加
159:       addEmployee();
160:     } else if (command.equals(REMOVE)) { //削除
161:       removeEmployee();
162:     } else if (command.equals(EDIT)) { //編集
163:       editEmployee();
164:     } else if (command.equals(UP_CURSOR)) { //カーソルを上へ
165:       moveCursor(UP);
166:     } else if (command.equals(DOWN_CURSOR)) { //カーソルを下へ
167:       moveCursor(DOWN);
168:     } else if (command.equals(SAVE)) { //セーブ
169:       save();
170:     } else if (command.equals(LOAD)) { //ロード
171:       load();
172:     }
173:   }
174:
175:   /*****************************************************
```

```
176:   /* 名簿管理コマンド群
177:   /*****************************************************/
178:
179:   /**
180:    * 社員名簿データに、新しい社員を追加する
181:    */
182:   void addEmployee() {
183:     //データが満杯だったらエラーを表示して終了する
184:     if (recordCount >= RECORD_MAX) {
185:       showError(DATA_OVERFLOW_ERROR_MESSAGE);
186:       return;
187:     }
188:
189:     //データを入力する
190:     showPrompt("社員 ID");
191:     id[recordCount] = getValidInt();
192:     showPrompt("所属部署 ");
193:     section[recordCount] = Input.getString();
194:     showPrompt("名前 ");
195:     name[recordCount] = Input.getString();
196:     showPrompt("住所 ");
197:     address[recordCount] = Input.getString();
198:     showPrompt("生まれた年 ");
199:     birthYear[recordCount] = getValidInt();
200:     showPrompt("年齢 ");
201:     age[recordCount] = getValidInt();
202:
203:     //追加の後処理
204:     recordCount++;
205:     currentRecordIndex = recordCount - 1;
206:   }
207:
208:   /**
209:    * 名簿データの削除を行う
210:    */
211:   void removeEmployee() {
212:
213:     //社員が存在しないとき
214:     if (recordCount == 0) {
215:       showError(NO_EDITABLE_ELEMENT_MESSAGE);
216:       return;
217:     }
218:
219:     //プロンプトを出す
220:     System.out.println(REMOVE_CONFIRM_MESSAGE);
221:
222:     //入力する
223:     String input = Input.getString();
224:     input = input.toUpperCase(); //入力されたコマンドを大文字に
225:
226:     //削除確認がとれなかったときはなにもしない
227:     if (!input.toUpperCase().equals(YES)) {
228:       return;
229:     }
```

```
230:
231:     //削除する（要素をつめる）
232:     for (int i = currentRecordIndex; i < RECORD_MAX - 1; i++) {
233:       id[i] = id[i + 1];
234:       name[i] = name[i + 1];
235:       section[i] = section[i + 1];
236:       address[i] = address[i + 1];
237:       birthYear[i] = birthYear[i + 1];
238:       age[i] = age[i + 1];
239:     }
240:
241:     //削除の後始末
242:     recordCount--;
243:     if (recordCount == currentRecordIndex) { //最後の要素が削除された場合
244:       currentRecordIndex--;
245:     }
246:   }
247:
248:   /**
249:    * 社員の編集を行う
250:    */
251:   void editEmployee() {
252:
253:     //社員が存在しないとき
254:     if (recordCount == 0) {
255:       showError(NO_EDITABLE_ELEMENT_MESSAGE);
256:       return;
257:     }
258:
259:     //部署
260:     showPrompt("所属部署 " + section[currentRecordIndex]);
261:     String inputSection = Input.getString();
262:     if (inputSection.equals("")) { //空白なら，変更しないとする
263:       inputSection = section[currentRecordIndex];
264:     }
265:
266:     //名前
267:     showPrompt("名前 " + name[currentRecordIndex]);
268:     String inputName = Input.getString();
269:     if (inputName.equals("")) { //空白なら，変更しないとする
270:       inputName = name[currentRecordIndex];
271:     }
272:
273:     //住所
274:     showPrompt("住所 " + address[currentRecordIndex]);
275:     String inputAddress = Input.getString();
276:     if (inputAddress.equals("")) { //空白なら，変更しないとする
277:       inputAddress = address[currentRecordIndex];
278:     }
279:
280:     //誕生年
281:     showPrompt("誕生年 " + birthYear[currentRecordIndex]);
282:     int inputBirthYear = getValidInt();
283:
```

```
284:     //年齢
285:     showPrompt("年齢 " + age[currentRecordIndex]);
286:     int inputAge = getValidInt();
287:
288:     //データの変更を行う
289:     section[currentRecordIndex] = inputSection;
290:     name[currentRecordIndex] = inputName;
291:     address[currentRecordIndex] = inputAddress;
292:     birthYear[currentRecordIndex] = inputBirthYear;
293:     age[currentRecordIndex] = inputAge;
294:   }
295:
296:   /**
297:    * 指定された方向にカーソルを移動する
298:    */
299:   void moveCursor(int direction) {
300:
301:     int nextRecordIndex = 0; //次のカーソル位置
302:
303:     //データが存在しないとき
304:     if (recordCount == 0) {
305:       return;
306:     }
307:
308:     //次のカーソル位置を計算する
309:     switch (direction) {
310:       case UP : //上に移動
311:         nextRecordIndex = currentRecordIndex - 1;
312:         if (nextRecordIndex < 0) {
313:           nextRecordIndex = 0;
314:         }
315:         break;
316:       case DOWN : //下に移動
317:         nextRecordIndex = currentRecordIndex + 1;
318:         if (nextRecordIndex >= recordCount) {
319:           nextRecordIndex = recordCount - 1;
320:         }
321:         break;
322:       default :
323:         break;
324:     }
325:
326:     //次のカーソル位置を設定する
327:     currentRecordIndex = nextRecordIndex;
328:   }
329:
330:   /**
331:    * 名簿データのセーブを行う.
332:    */
333:   void save() {
334:
335:     //ファイル名を入力する
336:     showPrompt("セーブするファイル名を入力してください"); //プロンプト
337:     String fileName = Input.getString();
```

```
338:
339:     //前処理
340:     PrintStream writer = FileIO.openForWrite(fileName); //ファイルオープン
341:
342:     //書き込み
343:     for (int i = 0; i < recordCount; i++) {
344:       //CSV 形式
345:       writer.println(
346:         id[i]
347:           + ","
348:           + section[i]
349:           + ","
350:           + name[i]
351:           + ","
352:           + address[i]
353:           + ","
354:           + birthYear[i]
355:           + ","
356:           + age[i]);
357:     }
358:
359:     //後処理
360:     writer.close(); //ファイルクローズ
361:     System.out.println("正常にセーブされました");
362:   }
363:
364:   /**
365:    * 名簿データのロードを行う.
366:    */
367:   void load() {
368:
369:     //ファイル名を入力する
370:     showPrompt("ロードするファイル名を入力してください");
371:     String fileName = Input.getString();
372:
373:     //前処理
374:     ReadStream reader = FileIO.openForRead(fileName); //ファイルオープン
375:
376:     //読み込み
377:     for (recordCount = 0; !reader.isEnd(); recordCount++) {
378:       //CSV 形式のデータを分割する
379:       String line = reader.readLine();
380:       String[] elements = line.split(",");
381:
382:       //データを追加する
383:       id[recordCount] = Integer.parseInt(elements[0]);
384:       section[recordCount] = elements[1];
385:       name[recordCount] = elements[2];
386:       address[recordCount] = elements[3];
387:       birthYear[recordCount] = Integer.parseInt(elements[4]);
388:       age[recordCount] = Integer.parseInt(elements[5]);
389:     }
390:
391:     //後処理
```

```
392:     reader.close(); //ファイルクローズ
393:     currentRecordIndex = 0; //カーソルの初期化
394:     System.out.println("正常にロードされました");
395:   }
396:
397:   /*****************************************************
398:   /* コマンド用部品
399:   /*****************************************************/
400:
401:   /**
402:    * 数値の入力を受け取る
403:    * 入力が正しく無い場合は再入力を促す
404:    */
405:   int getValidInt() {
406:     while (true) {
407:       //入力する
408:       String input = Input.getString();
409:
410:       //入力が数字に変換可能なら入力された文字列を返す，そうでなければ再入力
411:       if (Input.isInteger(input)) {
412:         return Integer.parseInt(input);
413:       } else {
414:         showPrompt(INCORRECT_INPUT_ERROR_MESSAGE);
415:
416:       }
417:     }
418:   }
419:
420:   /**
421:    * 指定されたコマンドが正しいコマンドかどうかを調べる
422:    */
423:   boolean isCorrectCommand(String command) {
424:     for (int i = 0; i < COMMANDS.length; i++) {
425:       if (COMMANDS[i].equals(command)) { //妥当なコマンドならば
426:         return true;
427:       }
428:     }
429:     return false;
430:   }
431:
432:   /*****************************************************
433:   /* 画面表示用部品
434:   /*****************************************************/
435:
436:   /**
437:    * 指定されたエラーメッセージを表示する
438:    */
439:   void showError(String errorMessage) {
440:     System.out.print(errorMessage);
441:   }
442:
443:   /**
444:    * 指定されたメッセージを持つプロンプトを表示する
445:    */
```

```
446:   void showPrompt(String message) {
447:     System.out.println();
448:     System.out.print(message);
449:     System.out.print(COMMAND_PROMPT);
450:     System.out.flush();
451:   }
452:
453:   /**
454:    * 表示を更新する
455:    */
456:   void updateView() {
457:     showTitle();
458:     showDirectory();
459:   }
460:
461:   /**
462:    * タイトルを表示する
463:    */
464:   void showTitle() {
465:     System.out.println(COMPANY_NAME + "社員管理システム");
466:   }
467:
468:   /**
469:    * 社員の一覧情報を表示する
470:    */
471:   void showDirectory() {
472:     for (int i = 0; i < recordCount; i++) {
473:       showDirectoryRecord(i);
474:     }
475:   }
476:
477:   /**
478:    * 社員一人分の情報を表示する
479:    */
480:   void showDirectoryRecord(int recordIndex) {
481:
482:     //注目行マークをつける
483:     setMark(recordIndex);
484:
485:     //データを書き出す
486:     System.out.println(
487:       "\t"
488:         + (recordIndex + 1)
489:         + "\t"
490:         + id[recordIndex]
491:         + "\t"
492:         + section[recordIndex]
493:         + "\t"
494:         + name[recordIndex]
495:         + "\t"
496:         + address[recordIndex]);
497:     System.out.println(
498:       "\t"
499:         + "\t"
```

```
500:         + birthYear[recordIndex]
501:         + "年生まれ\t"
502:         + age[recordIndex]
503:         + "才\t");
504:
505:     //境界線を書く
506:     System.out.println(LINE);
507:   }
508:
509:   /**
510:    * 注目行マークをつける
511:    */
512:   void setMark(int recordIndex) {
513:     if (recordIndex == currentRecordIndex) {
514:       System.out.print("*");
515:     } else {
516:       System.out.print("  ");
517:     }
518:   }
519:
520: }
```

考えてみよう

1. もし、社員の電話番号も管理する必要がでたときどのような変更が必要になりますか？
2. このプログラムを読んでみて、わかりやすさという点での問題点をあげてください

変数のまとまりとしてのオブジェクト

第 2 章では、じゃんけんの「手」というデータをコンピュータで扱うために、現実世界でのグー、チョキ、パーといった意味のあるデータを 1, 2, 3 という数値に置き換えて表現しました。

社員名簿アプリケーション (リスト 44) の例でいえば、これまでは、図 7.1 のようにいくつかの配列で個別のデータを表現していて、「社員」という意味のあるデータは、「配列の番地が同じ」ということで表現していました。

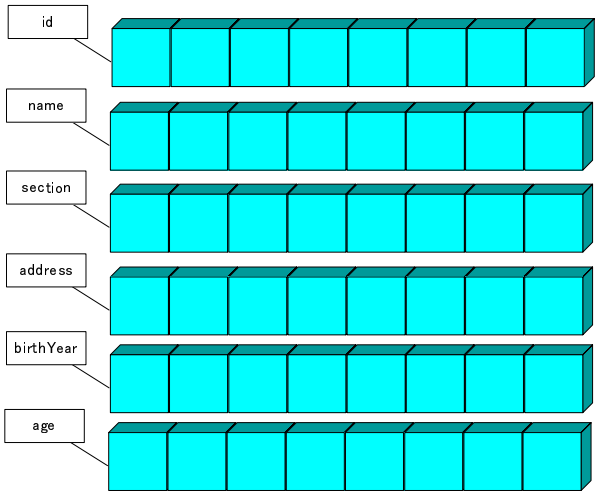


図 7.1: これまでの「社員」の表現

しかし、人間からみればやはりじゃんけんの手は数値ではなく、グー、チョキ、パーといった意味で捉えていますから、プログラムでもなるべく直接表現できたほうが人に分かりやすいといえるでしょう。

これを実現するために、オブジェクト指向という考え方が生まれました。オブジェクト指向では、データを組み合わせて**オブジェクト**という、人間にとって意味のあるデータの単位でプログラムを記述することが出来ます。

社員名簿アプリケーション (リスト 44) の例でいえば、図 7.2 のようにそれぞれの配列で扱っていた名前や部署を一つ一つまとめて「社員」という単位で扱うことができます。

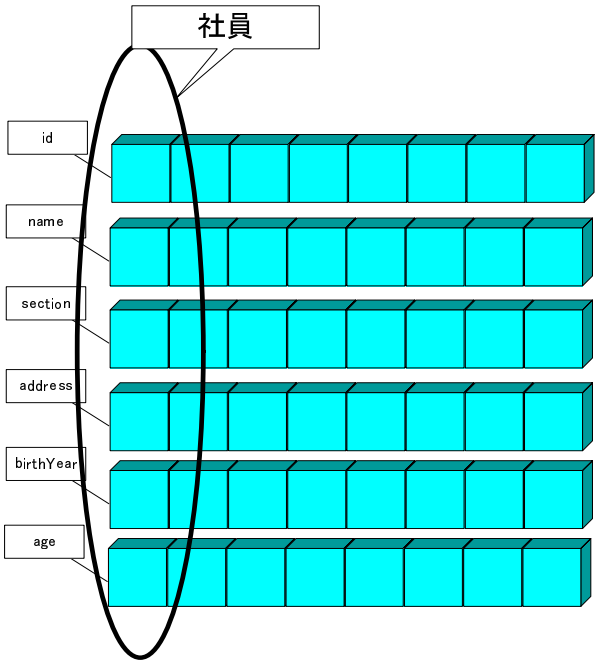


図 7.2: 変数のまとまりとしてのオブジェクト

第 II 部では、このようなオブジェクト指向の考え方を使って、現実のデータや目的が直接表現されたプログラムを書く手法を議論していきます。

クラスとインスタンス

Java ではオブジェクトをいきなり使うことは出来ません。オブジェクトを使うためには、その使いたいオブジェクトはどのような変数をまとめたものなのかを定義する必要があります。この定義のことを Java では**クラス**と呼びます。また、定義 (クラス) から生成される実体（オブジェクト）のことをインスタンスと呼びます。またクラスからインスタンスを生成することをインスタンス化といいます。

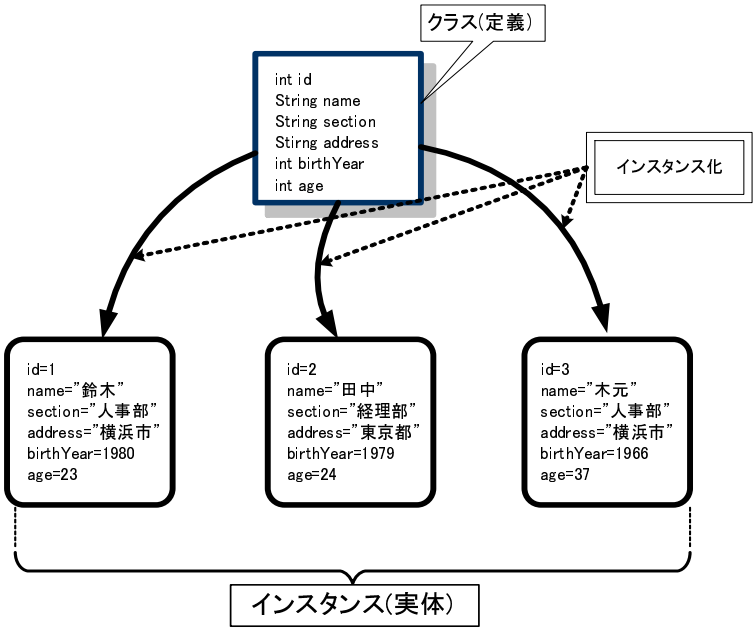


図 7.3: クラスとインスタンス

7.1.2 クラスを使ったプログラムの記述

それでは実際に社員クラスを定義し、社員クラスのインスタンスを使うサンプルプログラムを書いてみましょう。[1]

リスト 45: クラス（社員）を使ったサンプルプログラム

```
 1: /**
 2:  * クラス（社員）を使ったサンプルプログラム
 3:  *
 4:  * @author bam
 5:  * @version $Id: EmployeeClassSample.java,v 1.4 2003/05/08 16:54:53 bam Exp $
 6:  */
 7: public class EmployeeClassSample {
 8:
 9:   public static void main(String[] args) {
10:     EmployeeClassSample employeeClassSample = new EmployeeClassSample();
11:     employeeClassSample.main();
12:   }
13:
14:   void main() {
15:
16:     //社員 A を生成する
17:     Employee employeeA; /*社員を格納する変数の宣言*/
18:     employeeA = new Employee(); /*社員インスタンスの生成*/
19:     employeeA.id = 3; /*社員の id を 3 に設定する*/
20:     employeeA.name = "田中"; /*社員の名前を田中に設定する*/
21:
22:     //社員 B を生成する
23:     Employee employeeB;
24:     employeeB = new Employee();
25:     employeeB.id = 4;
26:     employeeB.name = "山本";
27:
28:     //社員のデータの表示
29:     System.out.println(employeeA.id + ":" + employeeA.name);/*社員 A の ID と名前を取
得*/
30:     System.out.println(employeeB.id + ":" + employeeB.name);
31:   }
32: }
33:
34: /**
35:  * 社員クラス
36:  * （ID と名前だけに簡素化してある）
37:  */
38: class Employee {
39:
```

[1] このプログラムは、クラス/インスタンス関連のコードが分かりやすいように、あえてプログラムの解説用コメントをつけています。さらに、変数名に tanaka などの意味のある名前ではなく、employeeA という無機質な名前にしてあります。

```
40:   int id; //社員 ID
41:   String name; //漢字の名前
42:
43: }
```

7.1.2.1 クラスの定義とインスタンス変数

クラスを定義するには

リスト 46: クラスの定義

```
class [クラス名] {

（ここに内容を書く）

}
```

という書式を使います。プログラムの中で使うクラスは、プログラム本体のスコープの外に定義する必要があります。クラスの命名規則は今までアプリケーションの命名をしてきたものと同じです。そのクラスがどんな変数のまとまりを意味するのかが一目でわかる名前をつけます。

クラスがどのような変数をまとめたものであるかを定義するために、クラスが持つ変数を記述します。クラスが持つ変数を定義するためには class のスコープの中で、変数を宣言します。このような変数のことをインスタンス変数と呼んだり、クラスの属性（フィールド）と呼んだりします。

7.1.3 入れ子モデル

プログラムを実行してインスタンスを生成する過程を、「インスタンスの入れ子モデル」という、表モデルを拡張したモデルをつかってあらわします。クラスを使ったサンプルプログラム (リスト 45) をつかってインスタンスが生成する過程を追ってみましょう。

7.1.3.1 クラスと型

インスタンスを使うためには、まずインスタンスを入れるための変数を宣言する必要があります。変数の宣言には型名としてクラス名を使います。クラスを定義するということは、新たな int, boolean と同じような型を定義するということであると考えることができます。

リスト 47: インスタンスを格納する変数の宣言

```
17:     Employee employeeA; /*社員を格納する変数の宣言*/
```

この式が評価されたときの表モデルは図 7.4 のようになります。

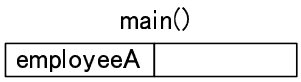


図 7.4: 表モデルとクラス

7.1.3.2 インスタンスの生成

クラスからインスタンスを生成するためには、new [クラス名]() という書式を使います。new で生成したインスタンスをプログラムで使うために、生成したインスタンスを変数に代入して使います。

リスト 48: インスタンスの生成と代入

```
18:     employeeA = new Employee(); /*社員インスタンスの生成*/
```

この式を評価するとまず右辺の new Employee() が先に評価されインスタンスが生成されます。インスタンスが生成されることをインスタンスの入れ子モデルでは新たな表がつくられることであらわします。また、生成されたインスタンスがどのクラスのインスタ

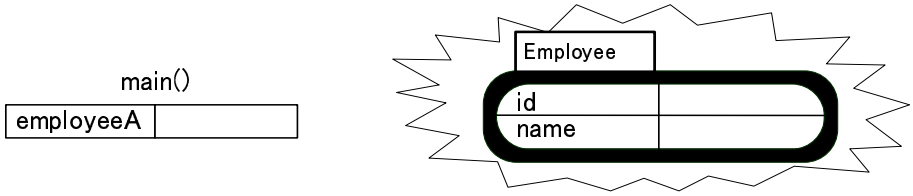


図 7.5: インスタンスの生成

ンスなのかを、表の上部に記述します。ここまでが評価されたときの表モデルは図 7.5 のようになります。

次に左辺の変数が評価され、最後に代入式が評価されます。インスタンス変数への代入は、インスタンスが変数の値として、表の中に入っていると考えます。ここまでが評価されたときの表モデルは図 7.6 のようになります。

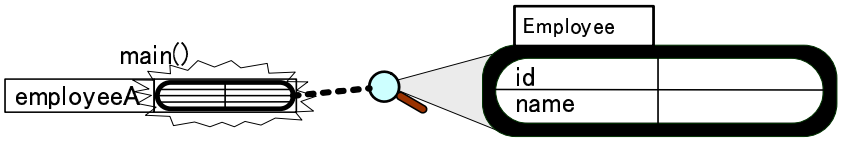


図 7.6: インスタンスの変数への代入

7.1.3.3　インスタンス変数の代入

プログラムからインスタンス変数にアクセスするためには、アクセスしたいインスタンスを保管してある変数のあとに”.”をつけ、”.”のあとにアクセスしたいインスタンス変数名をつけてアクセスします。

リスト 49: インスタンス変数への代入

```
19:     employeeA.id = 3; /*社員の id を 3 に設定する*/
```

代入は上のようにして行います。 この式を評価するとまず右辺が評価され、次に左辺が評価されます。左辺はまず変数 employee が評価され、実際のインスタンスが返ってきます。次に”.”の後が評価され、インスタンスの中の表にある変数の値が返ってきます。最後に代入文が評価されて表に値が代入されます。

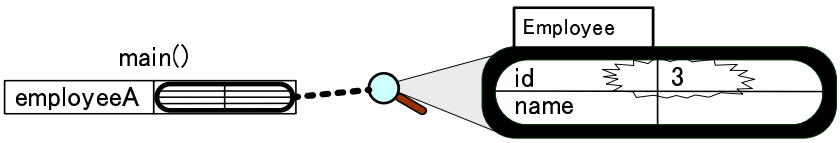


図 7.7: インスタンス変数への代入

7.1.3.4 インスタンス変数の取得

リスト 50: インスタンス変数の取得

```
 29:     System.out.println(employeeA.id + ":" + employeeA.name);/*社員Aの ID と名前を取
得*/
```

この式を評価するとまず変数 employee が評価され、実際のインスタンスが返ってきます。次に”.”の後が評価され、インスタンスの表に格納されている値が返ってきます。

Topics インスタンスとオブジェクト

インスタンスとオブジェクトという言葉は、Java を使ってプログラムを書く場合基本的に同義です。しかし、オブジェクトはプログラムが扱う変数のまとまりとしての意味合いが強いのに対し、インスタンスはクラスという定義から生成された具体物という意味合いを強く持っています。

Topics String クラス

ここまでクラスについて学んだところで、今まであまり深く触れてこなかった String について考えてみましょう。String という型は、int などの他の型と違い頭文字が大文字です。これは、String が文字列をあらわすクラスであるということです。

それではなぜ String というクラスのインスタンスを使う時には new が必要ないのでしょうか？実は String の場合は String s=”文字列”; という書式が new と同じ意味をあらわします。今まで使ってきた書式は new を省略したものだったのです。

このように String は本来はオブジェクトですが、インスタンスの入れ子モデルでは、簡略化するために一つの表ではなく、int などと同じような値として表します。

7.1.4　複雑な構造を持ったオブジェクト

クラスとインスタンスの基本的な使い方を学んだところで、プログラムが実行されるとインスタンスの状態がどのように変化していくのかを見てみましょう。

バスの運行をシミュレートするサンプルプログラムを使ってプログラムが実行される過程を追ってみます。

リスト 51: 入れ子モデル例題　バスの運行シミュレートサンプルプログラム

```
 1: /**
 2:  * バスの運行をシミュレートするサンプルプログラム
 3:  *
 4:  * @author bam
 5:  * @version $Id: BusSimulatorSample.java,v 1.2 2003/05/07 10:03:42 macchan Exp $
 6:  */
 7: public class BusSimulatorSample {
 8:
 9:   public static void main(String[] args) {
10:     BusSimulatorSample busSample = new BusSimulatorSample();
11:     busSample.main();
12:   }
13:
14:   void main() {
15:
16:     //乗客（田中と山本）を生成する
17:     Passenger tanaka = new Passenger();
18:     //-------------------(1)---------------------
19:     tanaka.name = "田中";
20:     //-------------------(2)---------------------
21:     Passenger yamamoto = new Passenger();
22:     yamamoto.name = "山本";
23:
24:     //浜松町行きバスを生成する
25:     Bus bus = new Bus();
26:     bus.destination = "浜松町";
27:     //-------------------(3)---------------------
28:
29:     //バス停（台場、テレポート、浜松町）を生成する
30:     BusStop daiba = new BusStop();
31:     daiba.name = "台場";
32:     BusStop teleport = new BusStop();
33:     teleport.name = "テレポート";
34:     BusStop hamamatsutyo = new BusStop();
35:     hamamatsutyo.name = "浜松町";
36:     //-------------------(4)---------------------
37:
38:     //山本と田中が台場のバス停でバス待ちをする
39:     daiba.waitingPassengers[0] = yamamoto;
40:     daiba.waitingPassengers[1] = tanaka;
41:     //-------------------(5)---------------------
42:
```

```
43:     //バスが台場に停車
44:     daiba.stoppingBus = bus;
45:     //------------------(6)--------------------
46:
47:     //山本と田中が台場のバス停でバス待ちの列から離れる
48:     daiba.waitingPassengers[0] = null;
49:     daiba.waitingPassengers[1] = null;
50:     //------------------(7)--------------------
51:
52:     //山本と田中がバスに乗る
53:     bus.ridingPassengers[0] = yamamoto;
54:     bus.ridingPassengers[1] = tanaka;
55:     //------------------(8)--------------------
56:
57:     //バスが台場を発車
58:     daiba.stoppingBus = null;
59:     //------------------(9)--------------------
60:
61:     //バスがテレポートに停車
62:     teleport.stoppingBus = bus;
63:     //------------------(10)--------------------
64:
65:     //山本がバスを降りる
66:     bus.ridingPassengers[0] = null;
67:     //------------------(11)--------------------
68:
69:     //バスがテレポートを発車
70:     teleport.stoppingBus = null;
71:     //------------------(12)--------------------
72:
73:     //バスが浜松町に停車
74:     hamamatsutyo.stoppingBus = bus;
75:     //------------------(13)--------------------
76:
77:     //田中がバスを降りる
78:     bus.ridingPassengers[1] = null;
79:     //------------------(14)--------------------
80:   }
81: }
82:
83: /**
84:  * バス停クラス
85:  * バスが停車して、待っている客が乗ることが出来る
86:  */
87: class BusStop {
88:
89:   String name; //停留所名
90:   Bus stoppingBus; //停車中のバス
91:   Passenger[] waitingPassengers = new Passenger[10]; //待っている客
92:
93: }
94:
95: /**
96:  * バスクラス
```

```
 97:  * 目的地に向かって移動する
 98:  * 乗客を乗せることができる
 99:  */
100: class Bus {
101:
102:   String destination; //目的地
103:   Passenger[] ridingPassengers = new Passenger[100]; //乗客
104:
105: }
106:
107: /**
108:  * 客クラス
109:  * 停留所で待ち、バスで移動する
110:  */
111: class Passenger {
112:
113:   String name; //名前
114:
115: }
```

7.1.4.1 new の評価

new によってインスタンスが生成されると図 7.8 のようになります。

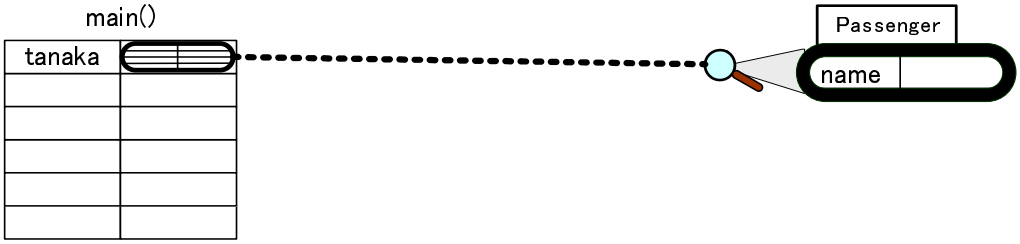


図 7.8: (1) 乗客（田中と山本）を生成する

7.1.4.2 値の代入

次にインスタンスに値が設定されます。入れ子モデルでは図 7.9 のように表します。

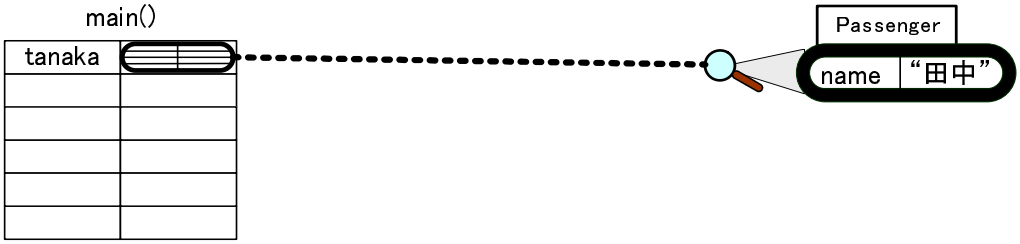


図 7.9: (2) 乗客の値を設定

7.1.4.3　インスタンスの配列

構造化プログラミング編では int や String の配列を使いましたが、インスタンスも配列で扱うことができます。使い方は int の場合と同じです。配列が初期化されると図 7.10 のようなモデルになります。

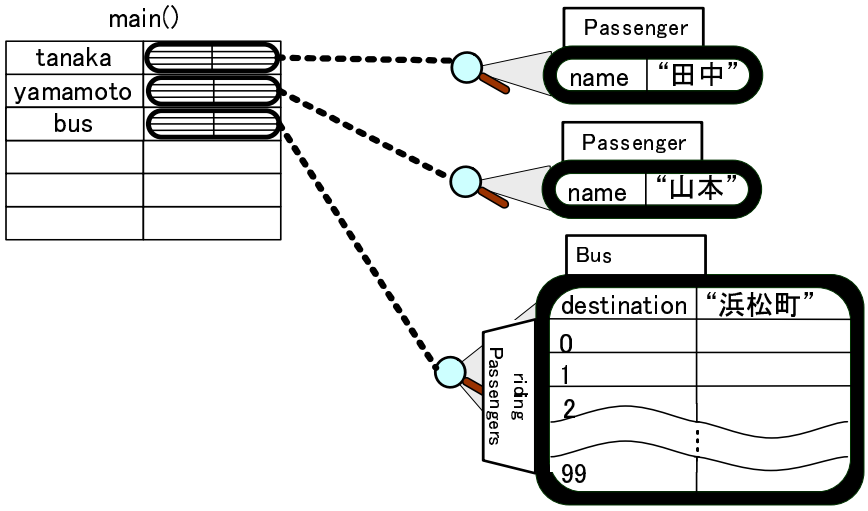


図 7.10: (3) 浜松町行きバスを生成する

バス停が生成されると図 7.11 のようなモデルになります。

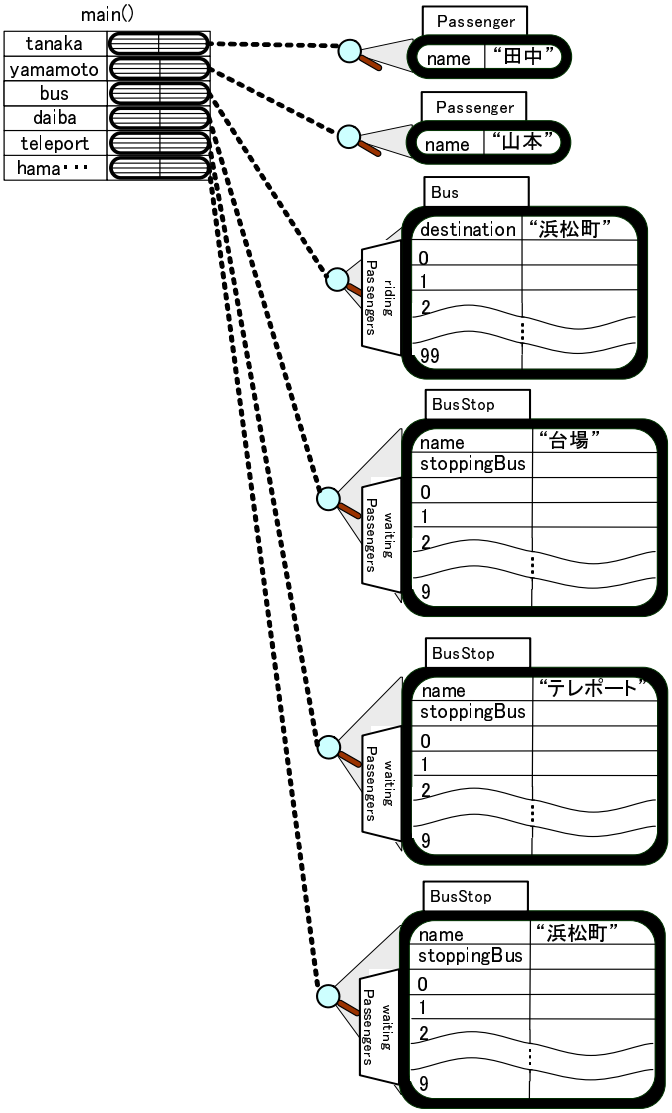


図 7.11: (4) バス停（台場、テレポート、浜松町）を生成する

7.1.4.4　インスタンスの入れ子

インスタンスの代入は、変数に対応する値の部分に他のインスタンスが入れ子になって入っているというモデルで表します。

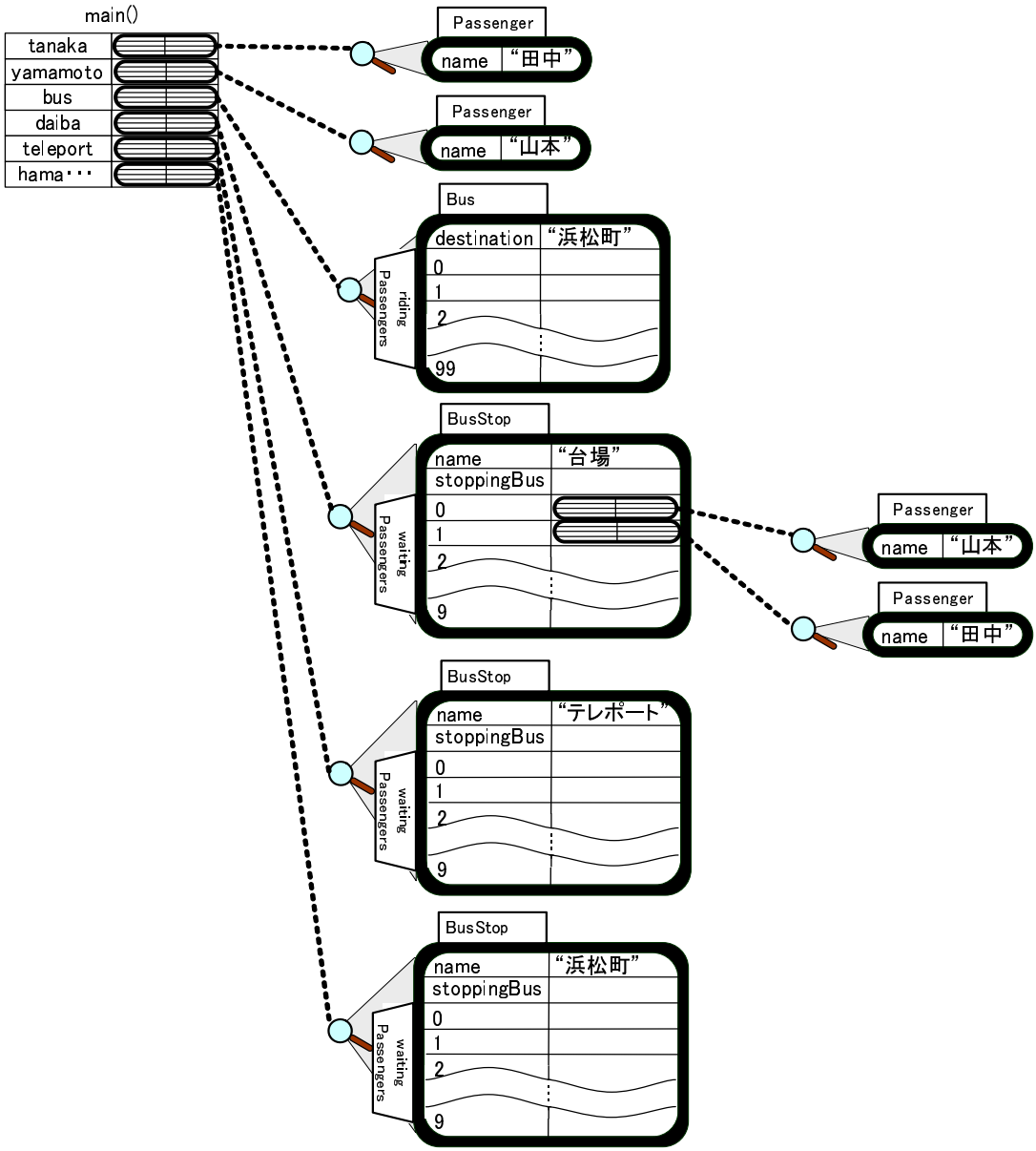


図 7.12: (5) 山本と田中が台場のバス停でバス待ちをする

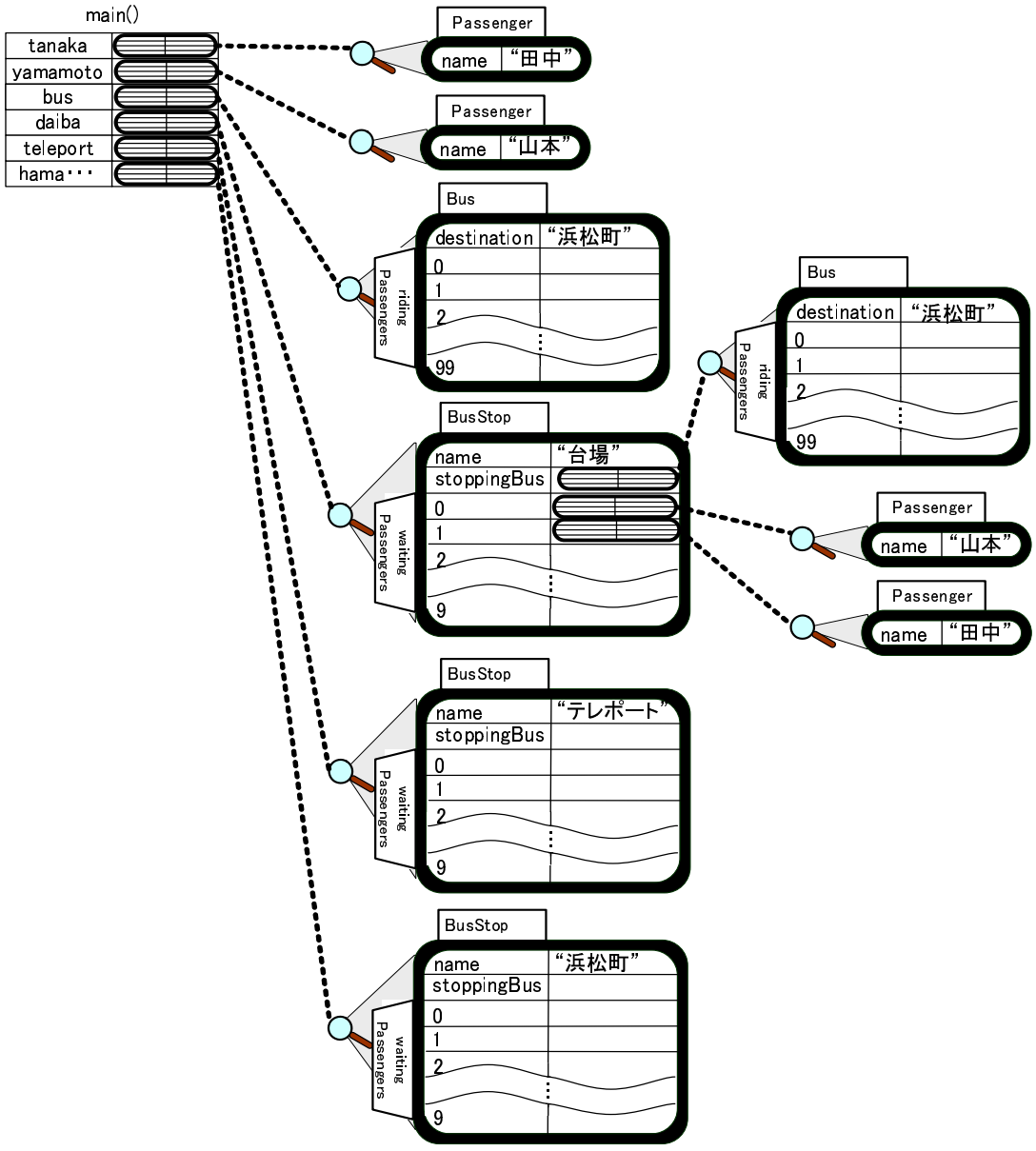


図 7.13: (6) バスが台場に停車

7.1.4.5　何も入っていない状態 (null)

配列の要素は変数に new した結果を代入することで使うことができることは上で説明したとおりです。では代入をする前は何が入っているのでしょうか？

答えは「何も入っていない」です。この何も入っていないことを Java では null といいます。[2]

バスのサンプルプログラムでは、バス停で待っている人がいなくなることを null が入ることであらわしています。

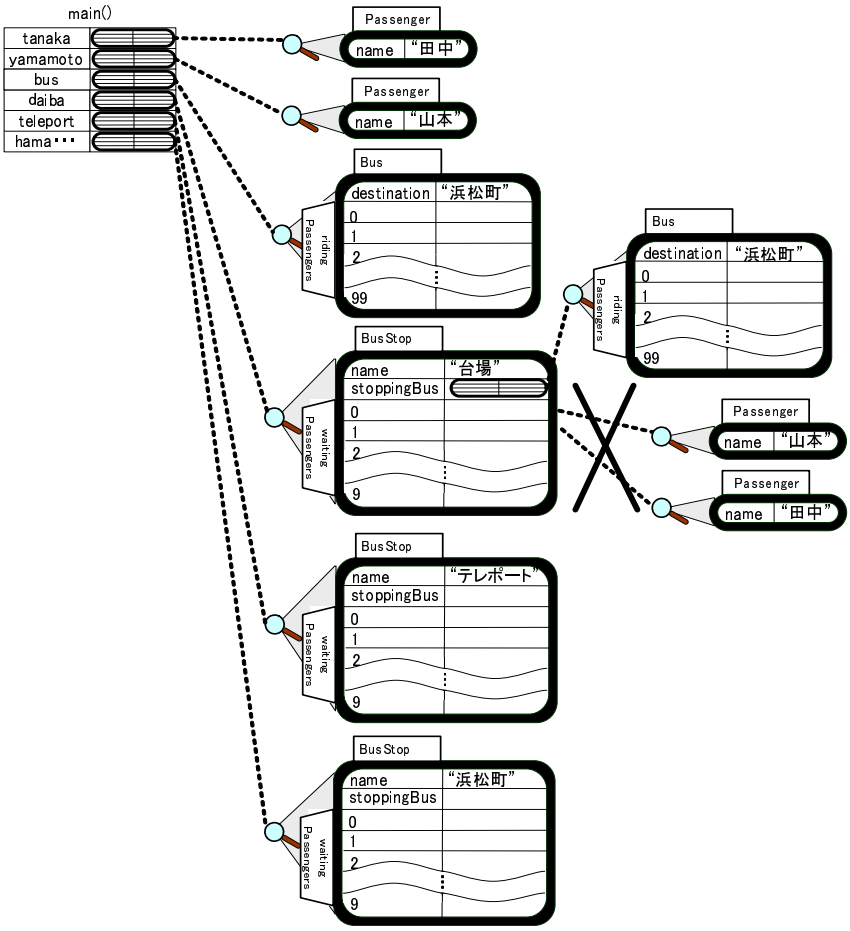


図 7.14: (7) 山本と田中が台場のバス停でバス待ちの列から離れる

[2] この状態を「null が入っている」などとも呼びます。

Topics NullPointerException

式を評価して返ってきたインスタンスが null であったときに、変数にアクセスしたり、メソッドを呼んだりしても、インスタンスがないので評価をすることができません。このようなときに発生する動的エラーのことを NullPointerException といいます。

リスト 52: NullPointerExeception を起こすプログラム

```
 1: /**
 2:  * NullPointerException を起こすサンプルプログラム
 3:  *
 4:  * @author bam
 5:  * @version $Id: NullPointerExceptionSample.java,v 1.2 2003/05/02 12:06:33 macchan Exp $
 6:  */
 7: public class NullPointerExceptionSample {
 8:
 9:   public static void main(String[] args) {
10:     NullPointerExceptionSample nullPointerExceptionSample =
11:       new NullPointerExceptionSample();
12:     nullPointerExceptionSample.run();
13:   }
14:
15:   void run() {
16:     Passenger[] passengers = new Passenger[10];
17:     System.out.println(passengers[0].name);
18:   }
19: }
20:
21: /**
22:  * 客クラス
23:  */
24: class Passenger {
25:   String name; //名前
26: }
```

リスト 52 を実行すると図 7.15 ような結果になります。

```
java.lang.NullPointerException
  at oop.chapter7.nullpointer.NullPointerExceptionSample.run
  (NullPointerExceptionSample.java:19)
  at oop.chapter7.nullpointer.NullPointerExceptionSample.main
  (NullPointerExceptionSample.java:14)
Exception in thread "main"
```

図 7.15: NullPointerException の実行結果

これは run メソッドの 19 行目で、式の評価の結果が null であったとき、その変数にアクセスしたために NullPointerException がでたということを表しています。

Topics Application クラス

今までプログラムを書くときには必ず public class XxxApplication という書式を使ってきました。実は、これもアプリケーションをあらわす一つのクラスです。ではどこでこのアプリケーションクラスは、インスタンス化されているのでしょうか？今まで無視してきた public static void main を見てください。

リスト 53: main メソッド

```
 9:   public static void main(String[] args) {
10:     NullPointerExceptionSample nullPointerExceptionSample =
11:       new NullPointerExceptionSample();
12:     nullPointerExceptionSample.run();
13:   }
```

この中をみてみるとアプリケーションのクラスがインスタンス化されていることがわかります。このように今までも実はクラスをつかってプログラムを実行してきたのです。run メソッドについては次章で詳しくやることになるのでここでは無視してもらって構いません。 public static void main はプログラムが実行されると自動で最初に呼び出される特別なメソッドだとここでは理解してください。

7.1.5 クラスを使ったプログラム

ここまでクラスとインスタンスについて学んできたところで、最初に登場した社員名簿アプリケーションをオブジェクトを使って書き直してみましょう。

リスト 54: クラスを使った社員名簿アプリケーション

```
 1: import java.io.PrintStream;
 2:
 3: /**
 4:  * 社員名簿管理アプリケーション
 5:  *
 6:  * 本アプリケーションの機能
 7:  * ○名簿の管理
 8:  *      ・社員の追加
 9:  *      ・社員の削除
10:  *      ・社員の編集
11:  * ○名簿の閲覧
12:  *      ・カーソルの移動
13:  * ○名簿の保存
14:  *      ・セーブ
15:  *      ・ロード
16:  *
17:  * @author bam
18:  * @version $Id: EmployeeDirectoryApplication.java,v 1.16 2003/05/07 20:40:57 bam Exp $
19:  */
20: public class EmployeeDirectoryApplication {
21:
22:   public static void main(String[] args) {
23:     EmployeeDirectoryApplication employeeDirectoryApplication =
24:       new EmployeeDirectoryApplication();
25:     employeeDirectoryApplication.run();
26:   }
27:
28:   /*****************************************************
29:   /* 定数
30:   *****************************************************/
31:
32:   //  保存できる最大データの数
33:   final int RECORD_MAX = 20;
34:
35:   //コマンド類
36:   final String ADD = "A"; //データの追加
37:   final String REMOVE = "R"; //データの削除
38:   final String EDIT = "E"; //データの編集
39:   final String UP_CURSOR = "U"; //カーソルを上へ
40:   final String DOWN_CURSOR = "D"; //カーソルを下へ
41:   final String SAVE = "S"; //セーブ
42:   final String LOAD = "L"; //ロード
43:   final String QUIT = "Q"; //終了
44:   final String[] COMMANDS =
45:     { ADD, REMOVE, EDIT, UP_CURSOR, DOWN_CURSOR, SAVE, LOAD, QUIT };
```

```
46:
47:   //表示文字列類
48:   final String LINE =
49:     "------------------------------------------------------------------------";
50:   final String SPACE = "  ";
51:   final String SC = ":";
52:   final String ADD_MSG = ADD + SC + "追加" + SPACE;
53:   final String RM_MSG = REMOVE + SC + "削除" + SPACE;
54:   final String EDIT_MSG = EDIT + SC + "編集" + SPACE;
55:   final String UC_MSG = UP_CURSOR + SC + "前のデータへ" + SPACE;
56:   final String DC_MSG = DOWN_CURSOR + SC + "次のデータへ" + SPACE;
57:   final String SAVE_MSG = SAVE + SC + "セーブ" + SPACE;
58:   final String LOAD_MSG = LOAD + SC + "ロード" + SPACE;
59:   final String QUIT_MSG = QUIT + SC + "終了" + SPACE;
60:   final String COMMAND_INFO_MESSAGE =
61:     "コマンド"
62:       + "("
63:       + SPACE
64:       + ADD_MSG
65:       + RM_MSG
66:       + EDIT_MSG
67:       + UC_MSG
68:       + DC_MSG
69:       + SAVE_MSG
70:       + LOAD_MSG
71:       + QUIT_MSG
72:       + ")";
73:   final String COMMAND_ERROR_MESSAGE = "不適当なコマンドでした";
74:   final String INCORRECT_INPUT_ERROR_MESSAGE = "数字を入力してください";
75:   final String COMMAND_PROMPT = " >> ";
76:   final String REMOVE_CONFIRM_MESSAGE = "本当に削除してもいいですか？ y/n";
77:   final String NO_EDITABLE_ELEMENT_MESSAGE = "編集可能なデータがありません";
78:   final String DATA_OVERFLOW_ERROR_MESSAGE = "データがいっぱいで、もう追加できません";
79:   final String COMPANY_NAME = "○×コミュニケーションズ";
80:   final String YES = "Y";
81:   final String SEPARATE_LINE = "\n"; //改行記号
82:
83:   //カーソル移動用
84:   final int UP = 1;
85:   final int DOWN = 2;
86:
87:   /*****************************************************
88:   /* インスタンス変数
89:   /*****************************************************/
90:
91:   int currentRecordIndex = -1; //現在カーソルがあるインデックス。何もないときは-1
92:   int recordCount = 0; //現在の要素数
93:
94:   Employee[] employees = new Employee[RECORD_MAX]; //社員の情報を保存する配列
95:
96:   /*****************************************************
97:   /* メイン
98:   /*****************************************************/
99:
```

```
100:   void run() {
101:
102:     String command; //入力されたコマンド
103:
104:     //初期画面を表示
105:     updateView();
106:
107:     //コマンドの入力と実行
108:     while (!(command = inputCommand()).equals(QUIT)) {
109:       executeCommand(command);
110:       updateView();
111:     }
112:
113:     //終了処理
114:     System.exit(0);
115:   }
116:
117:   /*****************************************************
118:   /* コマンド処理
119:   /*****************************************************/
120:
121:   /**
122:    * 社員名簿のコマンドを入力し、妥当なコマンドであれば、入力されたコマンドを返す。
123:    * もしも、不適当なコマンドであれば、再入力を要求する。
124:    */
125:   String inputCommand() {
126:
127:     String command; //入力されたコマンド
128:
129:     while (true) {
130:       //プロンプトを出す
131:       showPrompt(COMMAND_INFO_MESSAGE);
132:
133:       //入力する
134:       command = Input.getString();
135:       command = command.toUpperCase(); //入力されたコマンドを大文字に
136:
137:       //正しいコマンドならばループを抜けて入力コマンドを返す。
138:       //正しくなければエラーを表示して再入力
139:       if (isCorrectCommand(command)) {
140:         break;
141:       } else {
142:         showError(COMMAND_ERROR_MESSAGE);
143:       }
144:     }
145:     return command;
146:   }
147:
148:   /**
149:    * 指定された社員名簿のコマンドを実行する
150:    */
151:   void executeCommand(String command) {
152:     if (command.equals(ADD)) { //追加
153:       addEmployee();
```

```
154:     } else if (command.equals(REMOVE)) { //削除
155:       removeEmployee();
156:     } else if (command.equals(EDIT)) { //編集
157:       editEmployee();
158:     } else if (command.equals(UP_CURSOR)) { //カーソルを上へ
159:       moveCursor(UP);
160:     } else if (command.equals(DOWN_CURSOR)) { //カーソルを下へ
161:       moveCursor(DOWN);
162:     } else if (command.equals(SAVE)) { //セーブ
163:       save();
164:     } else if (command.equals(LOAD)) { //ロード
165:       load();
166:     }
167:   }
168:
169:   /*******************************************************
170:   /* 名簿管理コマンド群
171:   /*******************************************************/
172:
173:   /**
174:    * 社員名簿データに、新しい社員を追加する
175:    */
176:   void addEmployee() {
177:     //データが満杯だったらエラーを表示して終了する
178:     if (recordCount >= RECORD_MAX) {
179:       showError(DATA_OVERFLOW_ERROR_MESSAGE);
180:       return;
181:     }
182:
183:     //データを入力する
184:     showPrompt("社員 ID");
185:     int id = getValidInt();
186:     showPrompt("所属部署 ");
187:     String section = Input.getString();
188:     showPrompt("名前 ");
189:     String name = Input.getString();
190:     showPrompt("住所 ");
191:     String address = Input.getString();
192:     showPrompt("生まれた年 ");
193:     int birthYear = getValidInt();
194:     showPrompt("年齢 ");
195:     int age = getValidInt();
196:
197:     //入力されたデータから新しい社員を生成する
198:     Employee employee = new Employee();
199:     employee.id = id;
200:     employee.section = section;
201:     employee.name = name;
202:     employee.address = address;
203:     employee.birthYear = birthYear;
204:     employee.age = age;
205:
206:     //名簿に社員を追加する
207:     employees[recordCount] = employee;
```

```
208:
209:     //追加の後処理
210:     recordCount++;
211:     currentRecordIndex = recordCount - 1;
212:   }
213:
214:   /**
215:    * 名簿データの削除を行う
216:    */
217:   void removeEmployee() {
218:
219:     //社員が存在しないとき
220:     if (recordCount == 0) {
221:       showError(NO_EDITABLE_ELEMENT_MESSAGE);
222:       return;
223:     }
224:
225:     //プロンプトを出す
226:     System.out.println(REMOVE_CONFIRM_MESSAGE);
227:
228:     //入力する
229:     String input = Input.getString();
230:     input = input.toUpperCase(); //入力されたコマンドを大文字に
231:
232:     //削除確認がとれなかったときはなにもしない
233:     if (!input.toUpperCase().equals(YES)) {
234:       return;
235:     }
236:
237:     //削除する（要素をつめる）
238:     for (int i = currentRecordIndex; i < RECORD_MAX - 1; i++) {
239:       employees[i] = employees[i + 1];
240:     }
241:
242:     //削除の後始末
243:     recordCount--;
244:     if (recordCount == currentRecordIndex) { //最後の要素が削除された場合
245:       currentRecordIndex--;
246:     }
247:   }
248:
249:   /**
250:    * 社員の編集を行う
251:    */
252:   void editEmployee() {
253:
254:     //社員が存在しないとき
255:     if (recordCount == 0) {
256:       showError(NO_EDITABLE_ELEMENT_MESSAGE);
257:       return;
258:     }
259:
260:     //部署
261:     showPrompt("所属部署 " + employees[currentRecordIndex].section);
```

```
262:     String inputSection = Input.getString();
263:     if (inputSection.equals("")) { //空白なら，変更しないとする
264:       inputSection = employees[currentRecordIndex].section;
265:     }
266:
267:     //名前
268:     showPrompt("名前 " + employees[currentRecordIndex].name);
269:     String inputName = Input.getString();
270:     if (inputName.equals("")) { //空白なら，変更しないとする
271:       inputName = employees[currentRecordIndex].name;
272:     }
273:
274:     //住所
275:     showPrompt("住所 " + employees[currentRecordIndex].address);
276:     String inputAddress = Input.getString();
277:     if (inputAddress.equals("")) { //空白なら，変更しないとする
278:       inputAddress = employees[currentRecordIndex].address;
279:     }
280:
281:     //誕生年
282:     showPrompt("誕生年 " + employees[currentRecordIndex].birthYear);
283:     int inputBirthYear = getValidInt();
284:
285:     //年齢
286:     showPrompt("年齢 " + employees[currentRecordIndex].age);
287:     int inputAge = getValidInt();
288:
289:     //データの変更を行う
290:     employees[currentRecordIndex].section = inputSection;
291:     employees[currentRecordIndex].name = inputName;
292:     employees[currentRecordIndex].address = inputAddress;
293:     employees[currentRecordIndex].birthYear = inputBirthYear;
294:     employees[currentRecordIndex].age = inputAge;
295:   }
296:
297:   /**
298:    * 指定された方向にカーソルを移動する
299:    */
300:   void moveCursor(int direction) {
301:
302:     int nextRecordIndex = 0; //次のカーソル位置
303:
304:     //データが存在しないとき
305:     if (recordCount == 0) {
306:       return;
307:     }
308:
309:     //次のカーソル位置を計算する
310:     switch (direction) {
311:       case UP : //上に移動
312:         nextRecordIndex = currentRecordIndex - 1;
313:         if (nextRecordIndex < 0) {
314:           nextRecordIndex = 0;
315:         }
```

```
316:         break;
317:       case DOWN : //下に移動
318:         nextRecordIndex = currentRecordIndex + 1;
319:         if (nextRecordIndex >= recordCount) {
320:           nextRecordIndex = recordCount - 1;
321:         }
322:         break;
323:       default :
324:         break;
325:     }
326:
327:     //次のカーソル位置を設定する
328:     currentRecordIndex = nextRecordIndex;
329:   }
330:
331:   /**
332:    * 名簿データのセーブを行う.
333:    */
334:   void save() {
335:
336:     //ファイル名を入力する
337:     showPrompt("セーブするファイル名を入力してください"); //プロンプト
338:     String fileName = Input.getString();
339:
340:     //前処理
341:     PrintStream writer = FileIO.openForWrite(fileName); //ファイルオープン
342:
343:     //書き込み
344:     for (int i = 0; i < recordCount; i++) {
345:       //CSV 形式
346:       writer.println(
347:         employees[i].id
348:           + ","
349:           + employees[i].section
350:           + ","
351:           + employees[i].name
352:           + ","
353:           + employees[i].address
354:           + ","
355:           + employees[i].birthYear
356:           + ","
357:           + employees[i].age);
358:     }
359:
360:     //後処理
361:     writer.close(); //ファイルクローズ
362:     System.out.println("正常にセーブされました");
363:   }
364:
365:   /**
366:    * 名簿データのロードを行う.
367:    */
368:   void load() {
369:
```

```
370:     //ファイル名を入力する
371:     showPrompt("ロードするファイル名を入力してください");
372:     String fileName = Input.getString();
373:
374:     //前処理
375:     ReadStream reader = FileIO.openForRead(fileName); //ファイルオープン
376:
377:     //読み込み
378:     for (recordCount = 0; !reader.isEnd(); recordCount++) {
379:       //CSV 形式のデータを分割する
380:       String line = reader.readLine();
381:       String[] elements = line.split(",");
382:
383:       //データを追加する
384:       employees[recordCount].id = Integer.parseInt(elements[0]);
385:       employees[recordCount].section = elements[1];
386:       employees[recordCount].name = elements[2];
387:       employees[recordCount].address = elements[3];
388:       employees[recordCount].birthYear = Integer.parseInt(elements[4]);
389:       employees[recordCount].age = Integer.parseInt(elements[5]);
390:     }
391:
392:     //後処理
393:     reader.close(); //ファイルクローズ
394:     currentRecordIndex = 0; //カーソルの初期化
395:     System.out.println("正常にロードされました");
396:   }
397:
398:   /*******************************************************
399:   /* コマンド用部品
400:   /*******************************************************/
401:
402:   /**
403:    * 数値の入力を受け取る
404:    * 入力が正しく無い場合は再入力を促す
405:    */
406:   int getValidInt() {
407:     while (true) {
408:       //入力する
409:       String input = Input.getString();
410:
411:       //入力が数字に変換可能なら入力された文字列を返す，そうでなければ再入力
412:       if (Input.isInteger(input)) {
413:         return Integer.parseInt(input);
414:       } else {
415:         showPrompt(INCORRECT_INPUT_ERROR_MESSAGE);
416:       }
417:     }
418:   }
419:
420:   /**
421:    * 指定されたコマンドが正しいコマンドかどうかを調べる
422:    */
423:   boolean isCorrectCommand(String command) {
```

```
424:     for (int i = 0; i < COMMANDS.length; i++) {
425:       if (COMMANDS[i].equals(command)) { //妥当なコマンドならば
426:         return true;
427:       }
428:     }
429:     return false;
430:   }
431:
432:   /*******************************************************
433:   /* 画面表示用部品
434:   *******************************************************/
435:
436:   /**
437:    * 指定されたエラーメッセージを表示する
438:    */
439:   void showError(String errorMessage) {
440:     System.out.print(errorMessage);
441:   }
442:
443:   /**
444:    * 指定されたメッセージを持つプロンプトを表示する
445:    */
446:   void showPrompt(String message) {
447:     System.out.println();
448:     System.out.print(message);
449:     System.out.print(COMMAND_PROMPT);
450:     System.out.flush();
451:   }
452:
453:   /**
454:    * 表示を更新する
455:    */
456:   void updateView() {
457:     showTitle();
458:     showDirectory();
459:   }
460:
461:   /**
462:    * タイトルを表示する
463:    */
464:   void showTitle() {
465:     System.out.println(COMPANY_NAME + "社員管理システム");
466:   }
467:
468:   /**
469:    * 社員の一覧情報を表示する
470:    */
471:   void showDirectory() {
472:     for (int i = 0; i < recordCount; i++) {
473:       showDirectoryRecord(i);
474:     }
475:   }
476:
477:   /**
```

```
478:    * 社員一人分の情報を表示する
479:    */
480:   void showDirectoryRecord(int recordIndex) {
481:
482:     //注目行マークをつける
483:     setMark(recordIndex);
484:
484:     //データを書き出す
485:     System.out.println(
486:       "\t"
487:         + (recordIndex + 1)
488:         + "\t"
489:         + employees[recordIndex].id
490:         + "\t"
491:         + employees[recordIndex].section
492:         + "\t"
493:         + employees[recordIndex].name
494:         + "\t"
495:         + employees[recordIndex].address);
496:     System.out.println(
497:       "\t"
498:         + "\t"
499:         + employees[recordIndex].birthYear
500:         + "年生まれ\t"
501:         + employees[recordIndex].age
502:         + "才\t");
503:
504:     //境界線を書く
505:     System.out.println(LINE);
506:   }
507:
508:   /**
509:    * 注目行マークをつける
510:    */
511:   void setMark(int recordIndex) {
512:     if (recordIndex == currentRecordIndex) {
513:       System.out.print("*");
514:     } else {
515:       System.out.print("  ");
516:     }
517:   }
518: }
519:
520: /**
521:  * 社員クラス
522:  */
523: class Employee {
524:
525:   int id; //社員 ID
526:   String section; //部署
527:   String name; //漢字の名前
528:   String address; //住所
529:   int birthYear; //誕生年
530:   int age; //年齢
531:
```

```
532:
533: }
```

7.2　練習問題

練習問題 1

バス運行管理アプリケーションの (8) から (14) のそれぞれについて、入れ子モデルを使ってインスタンスの状態を図に記述してください。

練習問題 2

成績計算アプリケーション（PointCalculatorApplication.java）について、クラスを使って作り直してください。

また、作り直したプログラムと元のプログラムを比較して、クラスを使う利点を議論してください。

練習問題 3

辞書アプリケーション（DictionaryApplication.java）は削除機能にバグがあります。バグを発見し、直してください。

ヒント：辞書に単語を二つ以上登録し、最初に登録した単語を削除してから、他の単語を検索してみましょう。

練習問題 4*

成績計算アプリケーション（PointCalculatorApplication.java）を参考にして、直方体の縦、横、高さを入力すると体積を計算して表示するプログラムを作ってください。ただし、それ以外の仕様は自由に決めてかまいません。

練習問題 5*

クラスを使った社員名簿プログラム (EmployeeDirectoryApplication.java) にソートの機能を追加してください。

第 8 章

オブジェクトとしての抽象化 (2)

この章で学習すること

データと手続きの関係を説明できる
導出の考え方を説明できる

8.1 データ抽象

前回までで、クラスとインスタンスの仕組みを使って複数の変数をまとまりとし、1 つの”オブジェクト”として扱うということを学びました。つまり「オブジェクトが持つデータ=オブジェクトが変数として持っているデータ」ということになります。しかし、オブジェクトが持つデータとは変数として持っているデータだけとは限りません。計算で求めることにより得られるデータもあります。この章では、データとは何なのかということについて考えていきます。

8.1.1 導出されるデータ

リスト 55: 前回までの社員クラス

```
524: class Employee {
525:
526:   int id; //社員 ID
527:   String section; //部署
528:   String name; //漢字の名前
529:   String address; //住所
530:   int birthYear; //誕生年
531:   int age; //年齢
532:
533: }
```

考えてみよう

もし、1 月 1 日になって年がかわったら前の章までの社員名簿システムではどのようなことをする必要があるでしょう？

ヒント:現在のところ名簿には誕生日を入力するフィールドはないので、全員が 1 月 1 日に誕生日を迎えることにします。

8.1.1.1 導出

データを毎年更新する必要をなくすためには、年齢などの毎年更新が必要なデータを、現在の年と、誕生年から算出するようなプログラムに変更します。このように、誕生年などの計算したい値の元となる生データから年齢などの値を導き出すことを、データを導出するといいます。

データを導出するかどうかは、プログラムの性質によります。社員名簿は導出することによって、変更の手間が省け、導出が効果的に使われている例だといえます。しかし、導出のための計算に多くの手間が取られてしまう場合など、導出を使わないほうがいい場合もあります。

リスト 56: 年齢を導出する場合の社員クラス（年齢が削除されている）

```
527: class Employee {
528:
529:   int id; //社員 ID
530:   String section; //部署
531:   String name; //漢字の名前
532:   String address; //住所
533:   int birthYear; //誕生年
534:
535: }
```

リスト 57: 年齢を導出するメソッド

```
429:   int getAge(Employee employee) {
430:     int age = Calendar.getCurrentYear() - employee.birthYear;
431:     return age;
432:   }
```

リスト 58: 年齢を導出するメソッドをつかった名簿レコード表示メソッド

```
482:   void showDirectoryRecord(int recordIndex) {
483:
484:     //注目行マークをつける
485:     setMark(recordIndex);
486:
487:     //データを書き出す
488:     Employee employee = employees[recordIndex];
489:     System.out.println(
490:       "\t"
```

```
491:         + (recordIndex + 1)
492:         + "\t"
493:         + employee.id
494:         + "\t"
495:         + employee.section
496:         + "\t"
497:         + employee.name
498:         + "\t"
499:         + employee.address);
500:     System.out.println(
501:       "\t"
502:         + "\t"
503:         + employee.birthYear
504:         + "年生まれ \t"
505:         + getAge(employee)
506:         + "才 \t");
507:
508:     //境界線を書く
509:     System.out.println(LINE);
510:   }
```

8.1.1.2 インスタンスメソッド

ここまで変数をまとめるためにクラスとオブジェクトの考え方をつかってきました。変数を定義したクラスにそのクラスのインスタンスが持つ変数を操作したり、取得するためのメソッドをつけることができます。これをインスタンスメソッドと呼びます。インスタンスメソッドの定義の書法は、今まで main のあるクラスに定義してきたメソッドと同じです。

リスト 59: 導出を行うインスタンスメソッドを導入した社員クラス

```
519: class Employee {
520:
521:   int id; //社員 ID
522:   String section; //部署
523:   String name; //漢字の名前
524:   String address; //住所
525:   int birthYear; //誕生年
526:
527:   /**
528:    * 年齢を取得する
529:    */
530:   int getAge() {
531:     int age = Calendar.getCurrentYear() - birthYear;
532:     return age;
533:   }
534: }
```

インスタンスメソッドをクラスの外部から評価するには、インスタンス変数にアクセスするときと同じように、インスタンスが格納された変数名のあとに. をつけて評価します。

リスト 60: インスタンスメソッドを使うサンプルプログラム

```
1: //山田さんを生成する
2: Employee employee = new Employee();
3: employee.name="山田";
4: employee.birthYear=1980;
5:
6: //山田さんの年齢を表示する
7: System.out.println(employee.getAge());
```

インスタンスメソッドを使ってプログラムを書くことで、アプリケーションのクラスは、社員のインスタンスに対してインスタンスメソッドを使って必要な値を取得したり、インスタンスの値の変更をしたりすることができます。このようなオブジェクトにそのオブジェクトに関するメソッドをつけてオブジェクト同士の連携でプログラムが成り立っていると考えるのがオブジェクト指向の考え方です。

8.1.1.3 Getter の導入

ここまでで、年齢を導出する社員クラスが完成しました。しかし、このままでは名前や部署など年齢以外のデータ構造を変更した場合は、アプリケーションまで書き換える必要が出てしまいます。

このような問題に対応するために、他のクラスからデータを使うような社員クラスのようなクラスには、そのクラスのインスタンスがもつ変数の値を取得するようなインスタンスメソッドを定義します。このようなメソッドのことを getter といい、一般的に”get[取得する変数名]”というような名前をつけます。

リスト 61: getter を導入した社員クラス

```
519: class Employee {
520:
521:   int id; //社員 ID
522:   String section; //部署
523:   String name; //漢字の名前
524:   String address; //住所
525:   int birthYear; //誕生年
526:
527:   /**
528:    * 社員 ID を取得する
```

```
529:    */
530:   int getId() {
531:     return id;
532:   }
533:
534:   /**
535:    * 部署を取得する
536:    */
537:   String getSection() {
538:     return section;
539:   }
540:
541:   /**
542:    * 名前を取得する
543:    */
544:   String getName() {
545:     return name;
546:   }
547:
548:   /**
549:    * 住所を取得する
550:    */
551:   String getAddress() {
552:     return address;
553:   }
554:
555:   /**
556:    * 誕生年を取得する
557:    */
558:   int getBirthYear() {
559:     return birthYear;
560:   }
561:
562:   /**
563:    * 年齢を取得する
564:    */
565:   int getAge() {
566:     int age = Calendar.getCurrentYear() - birthYear;
567:     return age;
568:   }
569: }
```

Topics 現在の年を取得するには

人にやさしいプログラミングの哲学では、現在の年をとるための専用のライブラリ (みんなが使う機能を提供するクラス集のこと) が用意されています。Java で現在の年をとるプログラムはオブジェクト指向編を最後まで終えたときに、どのようなことをすればそれが実現できるのかがわかるので、ここでは深く触れません。

リスト 62: 現在の年を表示するサンプルプログラム

```
 1: /**
 2:  * 現在の年を表示するサンプルプログラム
 3:  *
 4:  * @author bam
 5:  * @version $Id: ShowYearSample.java,v 1.1 2003/05/03 15:31:30 macchan Exp $
 6:  */
 7: public class ShowYearSample {
 8:
 9:   public static void main(String[] args) {
10:     ShowYearSample showYearSample = new ShowYearSample();
11:     showYearSample.run();
12:   }
13:
14:   void run() {
15:     int year = Calendar.getCurrentYear();
16:     System.out.println("今年は" + year + "年です");
17:   }
18:
19: }
```

このプログラムを実行すると以下のような結果になります。

```
今年は 2003 年です
```

図 8.1: 現在の年を表示するサンプルプログラム実行結果

考えてみよう

本章の冒頭で取りあげた疑問点についてもう一度考えてください。オブジェクトが持つデータとは一体何を指すのでしょうか？

8.1.1.4　データ抽象

この節で学んだように、オブジェクトが持つデータとは、クラスに定義されている変数だけとは限りません。手続きによって取得できる値も十分に、データと呼ぶことができます。なぜなら、そのクラスのインスタンスを手続きを通して利用するプログラムから見れば、年齢などの値が変数として定義されたものであるのか、計算した結果求められたものなのかは、全く意識する必要のないことであるからです。

このことからオブジェクトが持つデータとは、そのオブジェクトから何らかの手段によって得ることができる値であると言うことができます。その手段が手続きであるか、変数としてであるかは全く意識する必要はありません。

この節で学んだ getter をつかってオブジェクトにアクセスするということは、オブジェクトがどのような変数を持っているかを意識する必要がなくなるということです。つまり、オブジェクトがどのような変数を持っていたとしても、オブジェクトを使ってプログラムを書くときには、オブジェクトの使い方、つまりメソッドだけを理解していればよくなります。オブジェクトがどのような変数をもって情報を管理しているかをオブジェクトのデータ構造といい、このようなオブジェクトと、その使い方を定義してデータ構造を隠蔽する考え方をデータ抽象といいます。このような考え方を導入することで、プログラムを書くときに複雑なデータ構造を気にする必要がなくなり、わかりやすいプログラムを記述することができるようになります。

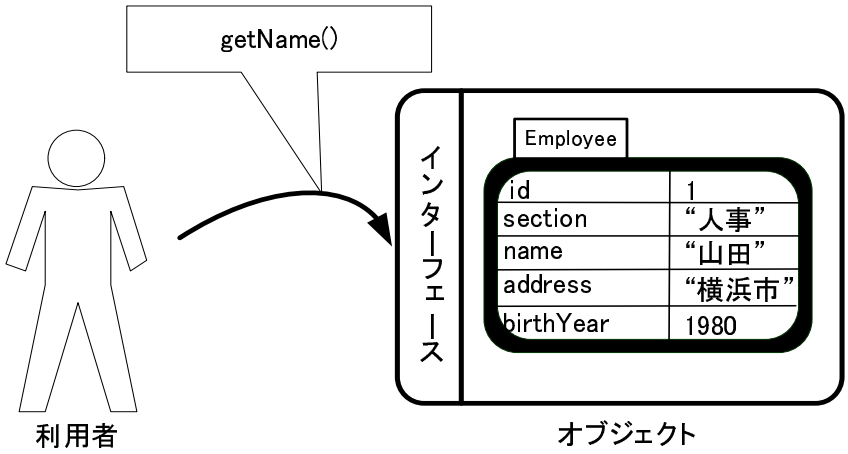


図 8.2: オブジェクト同士のやりとり

8.1.2 複雑さの隠蔽

8.1.2.1 複雑なものは一度に扱えない

人が一度に理解できる情報の量には限界があります。これはつまり、人は複雑なものを一度に管理することができないということです。では複雑なアプリケーションを管理するにはどうすればいいでしょう？答えは複雑なものを分割して、それぞれを理解できる単位まで小さくすることです。この節では前の節で学んだデータ抽象の考え方を使って、社員名簿アプリケーションをよりわかりやすくしていきます。

8.1.2.2 配列を隠す

今の社員名簿アプリケーションを複雑にしているものはなんでしょう？その最大の理由は配列でしょう。配列の操作は、配列の大きさや、挿入、削除のアルゴリズムなど複雑な処理がつきまといます。これを隠蔽することでアプリケーションをわかりやすくすることができます。

社員名簿で扱う社員の配列を隠蔽する、EmployeeList というクラスを導入して、メインのアプリケーションでは、配列の中身を操作する複雑な処理やデータ構造を気にしなくてもいいようにします。

リスト 63: 社員リストクラス

```
526: class EmployeeList {
527:
528:   final int ARRAY_SIZE = 20; //配列の大きさ
529:
530:   Employee[] employees = new Employee[ARRAY_SIZE]; //社員を保存する配列
531:   int size; //要素数
532:
533:   /**
534:    * 社員を追加する
535:    */
536:   void add(Employee employee) {
537:     employees[size] = employee;
538:     size++;
539:   }
540:
541:   /**
542:    * 指定された番地の社員を削除する
543:    */
544:   void remove(int index) {
545:     for (int i = index; i < size - 1; i++) { //要素を詰める
546:       employees[i] = employees[i + 1];
547:     }
548:     size--;
```

```
549:   }
550:
551:   /**
552:    * 全ての要素を削除する
553:    */
554:   void removeAll() {
555:     employees = new Employee[ARRAY_SIZE];
556:     size = 0;
557:   }
558:
559:   /**
560:    * 指定された番地の社員を取得する
561:    */
562:   Employee get(int index) {
563:     return employees[index];
564:   }
565:
566:   /**
567:    * 社員数を取得する
568:    */
569:   int size() {
570:     return size;
571:   }
572:
573:   /**
574:    * 社員が満杯かどうか調べる
575:    */
576:   boolean isFull() {
577:     if (size < ARRAY_SIZE) {
578:       return false;
579:     } else {
580:       return true;
581:     }
582:   }
583:
584:   /**
585:    * 社員が空かどうか調べる
586:    */
587:   boolean isEmpty() {
588:     if (size == 0) {
589:       return true;
590:     } else {
591:       return false;
592:     }
593:   }
594: }
```

リスト 64: 社員名簿アプリケーション (社員リスト導入)

```
20: public class EmployeeDirectoryApplication {
21:
```

```
22:   public static void main(String[] args) {
23:     EmployeeDirectoryApplication employeeDirectoryApplication =
24:       new EmployeeDirectoryApplication();
25:     employeeDirectoryApplication.run();
26:   }
27:
28:   /*************************************************
29:   /* 定数
30:   /*************************************************/
31:
32:   //コマンド類
33:   final String ADD = "A"; //データの追加
34:   final String REMOVE = "R"; //データの削除
35:   final String EDIT = "E"; //データの編集
36:   final String UP_CURSOR = "U"; //カーソルを上へ
37:   final String DOWN_CURSOR = "D"; //カーソルを下へ
38:   final String SAVE = "S"; //セーブ
39:   final String LOAD = "L"; //ロード
40:   final String QUIT = "Q"; //終了
41:   final String[] COMMANDS =
42:     { ADD, REMOVE, EDIT, UP_CURSOR, DOWN_CURSOR, SAVE, LOAD, QUIT };
43:
44:   //表示文字列類
45:   final String LINE =
46:     "-------------------------------------------------------------------------";
47:   final String SPACE = "  ";
48:   final String SC = ":";
49:   final String ADD_MSG = ADD + SC + "追加" + SPACE;
50:   final String RM_MSG = REMOVE + SC + "削除" + SPACE;
51:   final String EDIT_MSG = EDIT + SC + "編集" + SPACE;
52:   final String UC_MSG = UP_CURSOR + SC + "前のデータへ" + SPACE;
53:   final String DC_MSG = DOWN_CURSOR + SC + "次のデータへ" + SPACE;
54:   final String SAVE_MSG = SAVE + SC + "セーブ" + SPACE;
55:   final String LOAD_MSG = LOAD + SC + "ロード" + SPACE;
56:   final String QUIT_MSG = QUIT + SC + "終了" + SPACE;
57:   final String COMMAND_INFO_MESSAGE =
58:     "コマンド"
59:       + "("
60:       + SPACE
61:       + ADD_MSG
62:       + RM_MSG
63:       + EDIT_MSG
64:       + UC_MSG
65:       + DC_MSG
66:       + SAVE_MSG
67:       + LOAD_MSG
68:       + QUIT_MSG
69:       + ")";
70:   final String COMMAND_ERROR_MESSAGE = "不適当なコマンドでした";
71:   final String INCORRECT_INPUT_ERROR_MESSAGE = "数字を入力してください";
72:   final String COMMAND_PROMPT = " >> ";
73:   final String REMOVE_CONFIRM_MESSAGE = "本当に削除してもいいですか？ y/n";
74:   final String NO_EDITABLE_ELEMENT_MESSAGE = "編集可能なデータがありません";
75:   final String DATA_OVERFLOW_ERROR_MESSAGE = "データがいっぱいで、もう追加できません";
```

```
 76:   final String COMPANY_NAME = "○×コミュニケーションズ";
 77:   final String YES = "Y";
 78:   final String SEPARATE_LINE = "\n"; //改行記号
 79:
 80:   //カーソル移動用
 81:   final int UP = 1;
 82:   final int DOWN = 2;
 83:
 84:   /*******************************************************
 85:   /* インスタンス変数
 86:   /*******************************************************/
 87:
 88:   int currentRecordIndex = -1; //現在カーソルがあるインデックス。何もないときは-1
 89:
 90:   EmployeeList employees = new EmployeeList(); //社員を格納するリスト
 91:
 92:   /*******************************************************
 93:   /* メイン
 94:   /*******************************************************/
 95:
 96:   void run() {
 97:
 98:     String command; //入力されたコマンド
 99:
100:     //初期画面を表示
101:     updateView();
102:
103:     //コマンドの入力と実行
104:     while (!(command = inputCommand()).equals(QUIT)) {
105:       executeCommand(command);
106:       updateView();
107:     }
108:
109:     //終了処理
110:     System.exit(0);
111:   }
112:
113:   /*******************************************************
114:   /* コマンド処理部品
115:   /*******************************************************/
116:
117:   /**
118:    * 社員名簿のコマンドを入力し、妥当なコマンドであれば、入力されたコマンドを返す。
119:    * もしも、不適当なコマンドであれば、再入力を要求する。
120:    */
121:   String inputCommand() {
122:
123:     String command; //入力されたコマンド
124:
125:     while (true) {
126:       //プロンプトを出す
127:       showPrompt(COMMAND_INFO_MESSAGE);
128:
129:       //入力する
```

```
130:       command = Input.getString();
131:       command = command.toUpperCase(); //入力されたコマンドを大文字に
132:
133:       //正しいコマンドならばループを抜けて入力コマンドを返す。
134:       //正しくなければエラーを表示して再入力
135:       if (isCorrectCommand(command)) {
136:         break;
137:       } else {
138:         showError(COMMAND_ERROR_MESSAGE);
139:       }
140:     }
141:     return command;
142:   }
143:
144:   /**
145:    * 指定された社員名簿のコマンドを実行する
146:    */
147:   void executeCommand(String command) {
148:     if (command.equals(ADD)) { //追加
149:       addEmployee();
150:     } else if (command.equals(REMOVE)) { //削除
151:       removeEmployee();
152:     } else if (command.equals(EDIT)) { //編集
153:       editEmployee();
154:     } else if (command.equals(UP_CURSOR)) { //カーソルを上へ
155:       moveCursor(UP);
156:     } else if (command.equals(DOWN_CURSOR)) { //カーソルを下へ
157:       moveCursor(DOWN);
158:     } else if (command.equals(SAVE)) { //セーブ
159:       save();
160:     } else if (command.equals(LOAD)) { //ロード
161:       load();
162:     }
163:   }
164:
165:   /*****************************************************
166:   /* コマンド群
167:   /*****************************************************/
168:
169:   /**
170:    * 社員名簿データに、新しい社員を追加する
171:    */
172:   void addEmployee() {
173:     //データが満杯だったらエラーを表示して終了する
174:     if (employees.isFull()) {
175:       showError(DATA_OVERFLOW_ERROR_MESSAGE);
176:       return;
177:     }
178:
179:     //データを入力する
180:     showPrompt("社員 ID");
181:     int id = getValidInt();
182:     showPrompt("所属部署 ");
183:     String section = Input.getString();
```

```
184:     showPrompt("名前 ");
185:     String name = Input.getString();
186:     showPrompt("住所 ");
187:     String address = Input.getString();
188:     showPrompt("生まれた年 ");
189:     int birthYear = getValidInt();
190:
191:     //入力されたデータから新しい社員を生成する
192:     Employee employee = new Employee();
193:     employee.id = id;
194:     employee.section = section;
195:     employee.name = name;
196:     employee.address = address;
197:     employee.birthYear = birthYear;
198:
199:     //社員を追加する
200:     employees.add(employee);
201:     currentRecordIndex = getDirectoryRecordCount() - 1;
202:   }
203:
204:   /**
205:    * 名簿データの削除を行う
206:    */
207:   void removeEmployee() {
208:
209:     //社員が存在しないとき
210:     if (employees.isEmpty()) {
211:       showError(NO_EDITABLE_ELEMENT_MESSAGE);
212:       return;
213:     }
214:
215:     //プロンプトを出す
216:     System.out.println(REMOVE_CONFIRM_MESSAGE);
217:
218:     //入力する
219:     String input = Input.getString();
220:     input = input.toUpperCase(); //入力されたコマンドを大文字に
221:
222:     //削除確認がとれなかったときはなにもしない
223:     if (!input.toUpperCase().equals(YES)) {
224:       return;
225:     }
226:
227:     //削除する
228:     employees.remove(currentRecordIndex);
229:
230:     //削除の後始末
231:     if (getDirectoryRecordCount() == currentRecordIndex) { //最後の要素が削除された場
合
232:       currentRecordIndex--;
233:     }
234:   }
235:
236:   /**
```

```
237:     * 社員の編集を行う
238:     */
239:    void editEmployee() {
240:
241:      //社員が存在しないとき
242:      if (employees.isEmpty()) {
243:        showError(NO_EDITABLE_ELEMENT_MESSAGE);
244:        return;
245:      }
246:
247:      //新しい情報を入力する
248:
249:      Employee employee = employees.get(currentRecordIndex);
250:
251:      //部署
252:      showPrompt("所属部署 " + employee.getSection());
253:      String inputSection = Input.getString();
254:      if (inputSection.equals("")) { //空白なら，変更しないとする
255:        inputSection = employee.getSection();
256:      }
257:
258:      //名前
259:      showPrompt("名前 " + employee.getName());
260:      String inputName = Input.getString();
261:      if (inputName.equals("")) { //空白なら，変更しないとする
262:        inputName = employee.getName();
263:      }
264:
265:      //住所
266:      showPrompt("住所 " + employee.getAddress());
267:      String inputAddress = Input.getString();
268:      if (inputAddress.equals("")) { //空白なら，変更しないとする
269:        inputAddress = employee.getAddress();
270:      }
271:
272:      //誕生年
273:      showPrompt("誕生年 " + employee.getBirthYear());
274:      int inputBirthYear = getValidInt();
275:
276:      //データの変更を行う
277:
278:      //新しいデータを作成する
279:      Employee newEmployee = new Employee();
280:      newEmployee.section = inputSection;
281:      newEmployee.name = inputName;
282:      newEmployee.address = inputAddress;
283:      newEmployee.birthYear = inputBirthYear;
284:
285:      //データの取り替え
286:      employees.remove(currentRecordIndex); //古いデータを削除
287:      employees.add(newEmployee); //新しいデータを追加
288:    }
289:
290:    /**
```

```
291:    * 指定された方向にカーソルを移動する
292:    */
293:   void moveCursor(int direction) {
294:
295:     int nextRecordIndex = 0; //次のカーソル位置
296:
297:     //データが存在しないとき
298:     if (getDirectoryRecordCount() == 0) {
299:       return;
300:     }
301:
302:     //次のカーソル位置を計算する
303:     switch (direction) {
304:       case UP : //上に移動
305:         nextRecordIndex = currentRecordIndex - 1;
306:         if (nextRecordIndex < 0) {
307:           nextRecordIndex = 0;
308:         }
309:         break;
310:       case DOWN : //下に移動
311:         nextRecordIndex = currentRecordIndex + 1;
312:         if (nextRecordIndex >= getDirectoryRecordCount()) {
313:           nextRecordIndex = getDirectoryRecordCount() - 1;
314:         }
315:         break;
316:       default :
317:         break;
318:     }
319:
320:     //次のカーソル位置を設定する
321:     currentRecordIndex = nextRecordIndex;
322:   }
323:
324:   /**
325:    * 名簿データのセーブを行う.
326:    */
327:   void save() {
328:
329:     //ファイル名を入力する
330:     showPrompt("セーブするファイル名を入力してください"); //プロンプト
331:     String fileName = Input.getString();
332:
333:     //前処理
334:     PrintStream writer = FileIO.openForWrite(fileName); //ファイルオープン
335:
336:     //書き込み
337:     for (int i = 0; i < getDirectoryRecordCount(); i++) {
338:       Employee employee = employees.get(i);
339:       //CSV 形式
340:       writer.println(
341:         employee.getId()
342:           + ","
343:           + employee.getSection()
344:           + ","
```

```
345:           + employee.getName()
346:           + ","
347:           + employee.getAddress()
348:           + ","
349:           + employee.getBirthYear());
350:     }
351:
352:     //後処理
353:     writer.close(); //ファイルクローズ
354:     System.out.println("正常にセーブされました");
355:   }
356:
357:   /**
358:    * 名簿データのロードを行う.
359:    */
360:   void load() {
361:
362:     //ファイル名を入力する
363:     showPrompt("ロードするファイル名を入力してください");
364:     String fileName = Input.getString();
365:
366:     //前処理
367:     ReadStream reader = FileIO.openForRead(fileName); //ファイルオープン
368:
369:     employees.removeAll(); //名簿を初期化
370:
371:     //読み込み
372:     while (!reader.isEnd()) {
373:       //CSV 形式のデータを分割する
374:       String line = reader.readLine();
375:       String[] elements = line.split(",");
376:
377:       //データを追加する
378:       Employee employee = new Employee();
379:       employee.id = Integer.parseInt(elements[0]);
380:       employee.section = elements[1];
381:       employee.name = elements[2];
382:       employee.address = elements[3];
383:       employee.birthYear = Integer.parseInt(elements[4]);
384:
385:       //社員を名簿に追加する
386:       employees.add(employee);
387:     }
388:
389:     //後処理
390:     reader.close(); //ファイルクローズ
391:     currentRecordIndex = 0; //カーソルの初期化
392:     System.out.println("正常にロードされました");
393:   }
394:
395:   /*****************************************************
396:   /* コマンド用部品
397:   /*****************************************************/
398:
```

```
399:   /**
400:    * 数値の入力を受け取る
401:    * 入力が正しく無い場合は再入力を促す
402:    */
403:   int getValidInt() {
404:     while (true) {
405:       //入力する
406:       String input = Input.getString();
407:
408:       //入力が数字に変換可能なら入力された文字列を返す，そうでなければ再入力
409:       if (Input.isInteger(input)) {
410:         return Integer.parseInt(input);
411:       } else {
412:         showPrompt(INCORRECT_INPUT_ERROR_MESSAGE);
413:       }
414:     }
415:   }
416:
417:   /**
418:    * 指定されたコマンドが正しいコマンドかどうかを調べる
419:    */
420:   boolean isCorrectCommand(String command) {
421:     for (int i = 0; i < COMMANDS.length; i++) {
422:       if (COMMANDS[i].equals(command)) { //妥当なコマンドならば
423:         return true;
424:       }
425:     }
426:     return false;
427:   }
428:
429:   /*******************************************************
430:   /* 画面表示用部品
431:   /*******************************************************/
432:
433:   /**
434:    * 指定されたエラーメッセージを表示する
435:    */
436:   void showError(String errorMessage) {
437:     System.out.print(errorMessage);
438:   }
439:
440:   /**
441:    * 指定されたメッセージを持つプロンプトを表示する
442:    */
443:   void showPrompt(String message) {
444:     System.out.println();
445:     System.out.print(message);
446:     System.out.print(COMMAND_PROMPT);
447:     System.out.flush();
448:   }
449:
450:   /**
451:    * 表示を更新する
452:    */
```

```
453:   void updateView() {
454:     showTitle();
455:     showDirectory();
456:   }
457:
458:   /**
459:    * タイトルを表示する
460:    */
461:   void showTitle() {
462:     System.out.println(COMPANY_NAME + "社員管理システム");
463:   }
464:
465:   /**
466:    * 社員の情報を 1 画面に書く。
467:    */
468:   void showDirectory() {
469:     for (int i = 0; i < getDirectoryRecordCount(); i++) {
470:       showDirectoryRecord(i);
471:     }
472:   }
473:
474:   /**
475:    * 社員一人分の情報を表示する
476:    */
477:   void showDirectoryRecord(int recordIndex) {
478:
479:     setMark(recordIndex);
480:
481:     Employee employee = employees.get(recordIndex);
482:     System.out.println(
483:       "\t"
484:         + (recordIndex + 1)
485:         + "\t"
486:         + employee.getId()
487:         + "\t"
488:         + employee.getSection()
489:         + "\t"
490:         + employee.getName()
491:         + "\t"
492:         + employee.getAddress());
493:     System.out.println(
494:       "\t"
495:         + "\t"
496:         + employee.getBirthYear()
497:         + "年生まれ\t"
498:         + employee.getAge()
499:         + "才");
500:
501:     System.out.println(LINE);
502:   }
503:
504:   /**
505:    * 注目行マークをつける
506:    */
```

```
507:   void setMark(int recordIndex) {
508:     if (recordIndex == currentRecordIndex) {
509:       System.out.print("*");
510:     } else {
511:       System.out.print("  ");
512:     }
513:   }
514:
515:   /**
516:    * 名簿の要素数を取得する
517:    */
518:   int getDirectoryRecordCount() {
519:     return employees.size();
520:   }
521: }
```

8.1.2.3 目的が直接表現されたプログラム

これまで社員を追加するときにはリスト 65 のようなプログラムを書いていました。

リスト 65: 社員を追加するプログラム (配列版)

```
//社員を追加する
employees[recordCount] = employee;
recordCount++;
```

これは配列に追加し、要素数を表す変数を 1 増やすという、社員を追加するための手段を書いたプログラムです。

リストを導入すると、社員を追加する処理はリスト 66 のようなプログラムになります。

リスト 66: 社員を追加するプログラム (リスト版)

```
//社員を追加する
employees.add(employee);
```

このように複雑なものを隠蔽するクラスを導入すると、それを使う側のプログラムは、リストに対して、追加、削除、などの操作を行うだけのプログラムを書くことができます。これは今までコメントに書いていた「社員を追加する」という、直接目的が書かれたプログラムであるということができます。このように、手段ではなく目的に近いプログラムを書くことができることが、オブジェクト指向の大きな利点ということができます。

考えてみよう

削除のプログラムも比較してみましょう。

8.1.2.4 変更に強いプログラム

オブジェクト指向でデータが抽象化され、目的が直接的に書かれたプログラムは、読みやすいだけでなく、変更にも強いプログラムになるといわれています。

このことを議論するために、社員管理プログラムにおいて、配列を管理するクラスを導入した後、配列を管理するクラスのデータ構造が配列の要素数を変数では持たないように変更することを考えてみましょう。

プログラムはリスト 67 のようになります。

リスト 67: 社員リストクラス (size 変数を持たないように変更)

```
526: class EmployeeList {
527:
528:   final int ARRAY_SIZE = 20; //配列の大きさ
529:
530:   Employee[] employees = new Employee[ARRAY_SIZE]; //社員を保存する配列
531:
532:   /**
533:    * 社員を追加する
534:    */
535:   void add(Employee employee) {
536:     employees[size()] = employee;
537:   }
538:
539:   /**
540:    * 指定された番地の社員を削除する
541:    */
542:   void remove(int index) {
543:     //削除する
544:     employees[index] = null;
545:
546:     //要素を詰める
547:     for (int i = index; i < size() - 1; i++) {
548:       employees[i] = employees[i + 1];
549:     }
550:   }
551:
552:   /**
553:    * 全ての要素を削除する
554:    */
555:   void removeAll() {
556:     employees = new Employee[ARRAY_SIZE];
557:   }
558:
559:   /**
560:    * 指定された番地の社員を取得する
561:    */
562:   Employee get(int index) {
563:     return employees[index];
564:   }
565:
566:   /**
567:    * 社員数を取得する
568:    */
569:   int size() {
570:     int size = 0; //要素数
```

```
571:
572:     //社員が入っている数を数える
573:     while (employees[size] != null) {
574:       size++;
575:     }
576:
577:     return size;
578:   }
579:
580:   /**
581:    * 社員が満杯かどうか調べる
582:    */
583:   boolean isFull() {
584:     if (size() < ARRAY_SIZE) {
585:       return false;
586:     } else {
587:       return true;
588:     }
589:   }
590:
591:   /**
592:    * 社員が空かどうか調べる
593:    */
594:   boolean isEmpty() {
595:     if (size() == 0) {
596:       return true;
597:     } else {
598:       return false;
599:     }
600:   }
601: }
```

考えてみよう

リスト 67 を読んで、アプリケーションのクラスでどのような変更が起きたか議論してみましょう。

オブジェクト指向のプログラムは何故変更に強いと言われているのでしょうか?

8.2　練習問題

練習問題 1

社員管理システムに対して、以下のような機能追加を行ってください。

干支を表示する
元号 (昭和, 平成など) での誕生年を表示する

練習問題 2

健康管理システム (HealthManagementApplication.java) は、Person クラスが実装されていないので、実装してプログラムを動かしてみてください。

練習問題 3

社員管理システムに対して、星座を表示するという機能を追加する場合、どのようにデータを設計するべきか、その利点とともに説明してください。

練習問題 4

辞書アプリケーション（DictionaryApplication.java）に WordList クラスを導入してください。

練習問題 5*

社員管理システムの EmployeeList クラスにソート機能を実装してください。一旦出来たら、そのソートアルゴリズムを別のものに変更してみましょう。そして、アルゴリズムを変更した時に、EmployeeList クラス以外の部分のどこが変更されたか報告してください。

第 9 章

オブジェクトとしての抽象化 (3)

この章で学習すること

カプセル化の利点を説明できる
データ構造とアルゴリズムを結合させる意味を説明できる
スタックとキューを使ったプログラムが書ける
クラスの単体テストを行うことができる

9.1 カプセル化

8 章ではデータ抽象の考え方をつかって、クラスの内部の処理を隠蔽し、より目的に近いプログラムを書くための手法を学んできました。この章ではデータ抽象の考え方をより完全なものにするためにカプセル化という考え方を学び、クラスが持つ意味と責任について考えていきます。

9.1.1 破られる紳士協定

9.1.1.1 カプセル化の必要性

前の章では配列の操作などの複雑な処理や、配列や、社員が実際に持つデータを、それらを管理するクラスと、メソッドを導入することによって実際に何が行われているのかを隠蔽しました。そうすることによって、中で何が行われているかを意識せずともそれらのクラスを利用してプログラムを書くことができるようになります。しかし、このような利点は、あくまでもクラスを利用する人が、そのクラスの予定された使用方法を理解していた場合の話です。

前の章で導入した EmployeeList クラスはそのよい例です。例えばアプリケーションのクラスの中で以下のようなプログラムが書かれたらどのようなことが起きるでしょう。

リスト 68: 紳士協定を破ることができるプログラム

```
//index 番目の要素を削除する
employeeList.remove(index);
employeeList.size--;
```

現状のプログラムでは削除を行うときには remove() メソッドを呼べば、そのメソッドの中でリストの要素数に関する処理などはやってくれるという約束事は、あくまでもプログラマ間の暗黙の了解 (紳士協定) でした。つまり約束事を忘れて紳士協定を破るプログラムを書いてしまう可能性があったのです。このことは一人で開発を行っている場合はさしたる問題にはなりませんが、グループでの開発を行う場合などは大きな問題になります。

今までの紳士協定にすぎなかった約束事を明示し、誤った使い方をできなくするという考え方をオブジェクト指向ではカプセル化といいます。カプセル化を行うことで、オブジェクトをその明示された使い方だけを理解していれば内部のことを全く意識せずに使うことができるようになります。

9.1.1.2 アクセス修飾子

カプセル化の考え方を実現するために Java ではアクセス修飾子という書法が用意されています。クラス、変数、メソッドにこのアクセス修飾子をつけることで、それらがどのクラスのオブジェクトからならアクセスすることができるのかを明示します。

public・・・・・・・・・どのクラスのオブジェクトからでもアクセスできる
private・・・・・・・・このメソッドまたは変数が宣言されたクラスのオブジェクトしかアクセスできない
なにもつけない・・・・・・・現状では public と同じです。ただし将来同じ動作をしなくなる可能性があるので [1]、どのクラスからでもアクセスできるようにしたいときは public を必ずつけてください

リスト 69 は社員リストクラスにアクセス修飾子をつけてカプセル化をしたものです。

リスト 69: アクセス修飾子つけた EmployeeList

[1] アクセス修飾子をつけないことは、実際にはパッケージ内からのアクセスのみ許されるということをあらわしています。

```
 1: /**
 2:  * 社員を格納するリストクラス
 3:  *
 4:  * @author bam
 5:  * @version $Id: EmployeeList.java,v 1.1 2003/05/03 15:31:27 macchan Exp $
 6:  */
 7: class EmployeeList {
 8:
 9:   private final int ARRAY_SIZE = 20; //配列の大きさ
10:
11:   private Employee[] employees = new Employee[ARRAY_SIZE]; //社員を保存する配列
12:   private int size; //要素数
13:
14:   /**
15:    * 社員を追加する
16:    */
17:   public void add(Employee employee) {
18:     employees[size] = employee;
19:     size++;
20:   }
21:
22:   /**
23:    * 指定された番地の社員を削除する
24:    */
25:   public void remove(int index) {
26:     for (int i = index; i < size - 1; i++) { //要素を詰める
27:       employees[i] = employees[i + 1];
28:     }
29:     size--;
30:   }
31:
32:   /**
33:    * 全ての要素を削除する
34:    */
35:   public void removeAll() {
36:     employees = new Employee[ARRAY_SIZE];
37:     size = 0;
38:   }
39:
40:   /**
41:    * 指定された番地の社員を取得する
42:    */
43:   public Employee get(int index) {
44:     return employees[index];
45:   }
46:
47:   /**
48:    * 社員数を取得する
49:    */
50:   public int size() {
51:     return size;
52:   }
53:
54:   /**
```

```
55:    * 社員が満杯かどうか調べる
56:    */
57:   public boolean isFull() {
58:     if (size < ARRAY_SIZE) {
59:       return false;
60:     } else {
61:       return true;
62:     }
63:   }
64:
65:   /**
66:    * 社員が空かどうか調べる
67:    */
68:   public boolean isEmpty() {
69:     if (size == 0) {
70:       return true;
71:     } else {
72:       return false;
73:     }
74:   }
75: }
```

このカプセル化した EmployeeList を使ったプログラムを書くときに、アクセスの許されていない変数やメソッドにアクセスしようとするとどのようになるか見てみましょう。

リスト 70: 不正なアクセスをしようとするプログラム

```
 1: /**
 2:  * カプセル化されているクラスに対して、それを破ろうとするサンプルプログラム
 3:  *
 4:  * @author bam
 5:  * @version $Id: CapsuleExample.txt,v 1.1 2003/05/04 07:13:28 macchan Exp $
 6:  */
 7: public class CapsuleExample {
 8:
 9:   public static void main(String[] args) {
10:     CapsuleExample capsuleExample=new CapsuleExample();
11:     capsuleExample.run();
12:   }
13:
14:   private void run(){
15:     EmployeeList employees = new EmployeeList();
16:     System.out.println(employees.size);
17:   }
18: }
```

このようなプログラムを書くと以下のようなコンパイルエラーがでます。アクセス修飾子を使うと、不正なアクセスを静的エラーとして検出することができプログラムの信頼性を上げることができます。

```
CapsuleExample.java:16: size は EmployeeList で private アクセスされます。
                System.out.println(employees.size);
                                           ^
エラー 1 個
```

図 9.1: 不正なアクセスを行った場合のコンパイルエラー

9.1.1.3 Setter の導入

社員クラスも社員リストクラスと同じように変数を private アクセスにしてみましょう。そうすることで変数に直接アクセスできないようにすることができます。

しかし、このままでは変数が private なために、変数に直接アクセスしたり代入することができなくなり、オブジェクトに値を設定することができなくなります。

これを解決するためにオブジェクトに値を設定するメソッドを定義しましょう。定義したメソッドを public アクセスにします。このようなメソッドは getter に対して setter と呼ばれ、一般的に set[変数名] という名前をつけます。

リスト 71: setter をつけた Employee クラス

```
 1: /**
 2:  * 社員クラス
 3:  *
 4:  * @author bam
 5:  * @version $Id: Employee.java,v 1.2 2003/05/04 12:10:30 macchan Exp $
 6:  */
 7: public class Employee {
 8:
 9:   private int id; //社員 ID
10:   private String section; //部署
11:   private String name; //漢字の名前
12:   private String address; //住所
13:   private int birthYear; //誕生年
14:
15:   /**
16:    * 社員 ID を取得する
17:    */
18:   public int getId() {
19:     return id;
20:   }
21:
22:   /**
23:    * 社員ＩＤを設定する
24:    */
25:   public void setId(int id) {
26:     this.id = id;
27:   }
28:
29:   /**
```

```
30:    * 部署を取得する
31:    */
32:   public String getSection() {
33:     return section;
34:   }
35:
36:   /**
37:    * 部署を設定する
38:    */
39:   public void setSection(String section) {
40:     this.section = section;
41:   }
42:
43:   /**
44:    * 名前を取得する
45:    */
46:   public String getName() {
47:     return name;
48:   }
49:
50:   /**
51:    * 名前を設定する
52:    */
53:   public void setName(String name) {
54:     this.name = name;
55:   }
56:
57:   /**
58:    * 住所を取得する
59:    */
60:   public String getAddress() {
61:     return address;
62:   }
63:
64:   /**
65:    * 住所を設定する
66:    */
67:   public void setAddress(String address) {
68:     this.address = address;
69:   }
70:
71:   /**
72:    * 誕生年を取得する
73:    */
74:   public int getBirthYear() {
75:     return birthYear;
76:   }
77:
78:   /**
79:    * 誕生年を設定する
80:    */
81:   public void setBirthYear(int birthYear) {
82:     this.birthYear = birthYear;
83:   }
```

```
84:
85:   /**
86:    * 年齢を取得する
87:    */
88:   public int getAge() {
89:     int age = Calendar.getCurrentYear() - birthYear;
90:     return age;
91:   }
92:
93: }
```

9.1.1.4 スコープを超える変数へのアクセス (this)

リスト 72: 社員の名前を設定する setter

```
53:   public void setName(String name) {
54:     this.name = name;
55:   }
```

リスト 72 のように setter を定義すると、「setter の仮引数もフィールドと同じ変数名にしたい」というような名前の衝突が起きます。このようなとき、this. という書法を使います。ここでは this. とは「その呼び出しが行われたクラスのスコープの」という意味です。これまでメソッドを呼び出すときに、this. をつけなかったのは this. を省略していたと考えることもできます。

リスト 73: this. を使ったメソッド呼び出し

```
this.inputCommand()
```

考えてみよう

ではなぜ setter を用意する必要があるのでしょうか？ setter を public にするのならば、変数を public にするのと実際にはできることは同じです。

Topics ファイルの分割

ここまで Employee クラスにたくさんのインスタンスメソッドを追加してきました。そろそろファイルが膨大な行数になって管理しづらくなってきたのではないでしょうか？これを解決するためにクラスごとにファイルを分割するということができます。移動したいクラス全体を別のファイルに移動して、ファイル名を [クラス名].java にします。

このとき、分割した二つのファイルは同じディレクトリにおいてある必要がありますこのような二つ以上のクラスを使ったアプリケーションをコンパイルするときには使用する全てのファイルをコンパイルする必要があります。

```
C:\Documents and Settings\hitoyasa\>javac *.java
```

図 9.2: ディレクトリにある java ファイルをすべてコンパイルする

図 9.3: ファイルの構成

9.2 オブジェクトの初期化

社員の ID は普通後から変更されることはありません。また、せっかく社員追加のときに既存の ID と重複しないことを検査しているのに、他の部分で ID を変更するようなプログラムを書いてしまうと将来検索などの機能を ID をもとに行うときにうまく動かなくなる可能性があります。このような場合、ID には setter を作らず、オブジェクトを初期化する時に初期値として設定することができます。

9.2.1 コンストラクタ

9.2.1.1 引数を使った初期化

オブジェクトを初期化するために Java にはコンストラクタという書法があります。

リスト 74: setter とアクセス修飾子をつけた Employee

```
 1: /**
 2:  * 社員クラス
 3:  *
 4:  * @author bam
 5:  * @version $Id: Employee.java,v 1.2 2003/05/04 12:10:30 macchan Exp $
 6:  */
 7: public class Employee {
 8:
 9:   private int id; //社員 ID
10:   private String section; //部署
11:   private String name; //漢字の名前
12:   private String address; //住所
13:   private int birthYear; //誕生年
14:
15:   /**
16:    * コンストラクタ
17:    */
18:   public Employee(
19:     int id,
20:     String section,
21:     String name,
22:     String address,
23:     int birthYear) {
24:     this.id = id;
25:     this.section = section;
26:     this.name = name;
27:     this.address = address;
28:     this.birthYear = birthYear;
29:   }
30:
31:   /**
32:    * 社員 ID を取得する
```

```
33:    */
34:   public int getId() {
35:     return id;
36:   }
37:
38:   /**
39:    * 部署を取得する
40:    */
41:   public String getSection() {
42:     return section;
43:   }
44:
45:   /**
46:    * 部署を設定する
47:    */
48:   public void setSection(String section) {
49:     this.section = section;
50:   }
51:
52:   /**
53:    * 名前を取得する
54:    */
55:   public String getName() {
56:     return name;
57:   }
58:
59:   /**
60:    * 名前を設定する
61:    */
62:   public void setName(String name) {
63:     this.name = name;
64:   }
65:
66:   /**
67:    * 住所を取得する
68:    */
69:   public String getAddress() {
70:     return address;
71:   }
72:
73:   /**
74:    * 住所を設定する
75:    */
76:   public void setAddress(String address) {
77:     this.address = address;
78:   }
79:
80:   /**
81:    * 誕生年を取得する
82:    */
83:   public int getBirthYear() {
84:     return birthYear;
85:   }
86:
```

```
 87:   /**
 88:    * 誕生年を設定する
 89:    */
 90:   public void setBirthYear(int birthYear) {
 91:     this.birthYear = birthYear;
 92:   }
 93:
 94:   /**
 95:    * 年齢を取得する
 96:    */
 97:   public int getAge() {
 98:     int age = Calendar.getCurrentYear() - birthYear;
 99:     return age;
100:   }
101:
102: }
```

コンストラクタはクラス名と同じ名前で普通のメソッドのようなものをクラスの中に定義します。ただし、戻り値の型はいりません。 このようなコンストラクタを用意すると、今までのように new Employee() という書式でインスタンスを生成することはできません。以下のような書式で、インスタンスを生成する際に引数を渡す必要があります。

リスト 75: 引数を使った初期化

```
Employee employee = new Employee(101,"人事部","山田","横浜市旭区",1980,2002);
```

コンストラクタはメソッドと同じように、引数が違うコンストラクタを複数定義することもできます。もし、今までのように引数なしでのインスタンス生成も行いたい場合は、引数なしのコンストラクタを個別に用意する必要があります。

9.2.1.2 引数なしのコンストラクタ

コンストラクタを使うとオブジェクトを初期化する際の処理を書くことができます。今まで EmployeeList はインスタンス変数の宣言と同時に配列の初期化を行ってきました。これを引数なしのコンストラクタを用意し、そこで配列の初期化をするようにしてみましょう。

リスト 76: EmployeeList のコンストラクタ

```
/**
 * コンストラクタ
 */
public EmployeeList() {
  employees = new Employee[ARRAY_SIZE]; //配列を初期化する
}
```

引数なしのコンストラクタは他の引数を持つコンストラクタを定義しなければ省略することができます。この省略されていたコンストラクタのことをデフォルトコンストラクタといいます。これまで new Employee() という書式でインスタンスを生成することができたのはデフォルトコンストラクタが呼ばれていたのです。

リスト 77: 社員名簿アプリケーション (カプセル化導入)

```
20: public class EmployeeDirectoryApplication {
21:
22:   public static void main(String[] args) {
23:     EmployeeDirectoryApplication employeeDirectoryApplication =
24:       new EmployeeDirectoryApplication();
25:     employeeDirectoryApplication.run();
26:   }
27:
28:   /*****************************************************
29:   /* 定数
30:   /*****************************************************/
31:
32:   //コマンド類
33:   final String ADD = "A"; //データの追加
34:   final String REMOVE = "R"; //データの削除
35:   final String EDIT = "E"; //データの編集
36:   final String UP_CURSOR = "U"; //カーソルを上へ
37:   final String DOWN_CURSOR = "D"; //カーソルを下へ
38:   final String SAVE = "S"; //セーブ
39:   final String LOAD = "L"; //ロード
40:   final String QUIT = "Q"; //終了
41:   final String[] COMMANDS =
42:     { ADD, REMOVE, EDIT, UP_CURSOR, DOWN_CURSOR, SAVE, LOAD, QUIT };
43:
44:   //表示文字列類
45:   final String LINE =
46:     "-----------------------------------------------------------------------";
47:   final String SPACE = "  ";
48:   final String SC = ":";
49:   final String ADD_MSG = ADD + SC + "追加" + SPACE;
50:   final String RM_MSG = REMOVE + SC + "削除" + SPACE;
51:   final String EDIT_MSG = EDIT + SC + "編集" + SPACE;
52:   final String UC_MSG = UP_CURSOR + SC + "前のデータへ" + SPACE;
53:   final String DC_MSG = DOWN_CURSOR + SC + "次のデータへ" + SPACE;
54:   final String SAVE_MSG = SAVE + SC + "セーブ" + SPACE;
55:   final String LOAD_MSG = LOAD + SC + "ロード" + SPACE;
56:   final String QUIT_MSG = QUIT + SC + "終了" + SPACE;
57:   final String COMMAND_INFO_MESSAGE =
58:     "コマンド"
59:       + "("
60:       + SPACE
61:       + ADD_MSG
```

```
62:       + RM_MSG
63:       + EDIT_MSG
64:       + UC_MSG
65:       + DC_MSG
66:       + SAVE_MSG
67:       + LOAD_MSG
68:       + QUIT_MSG
69:       + ")";
70:   final String COMMAND_ERROR_MESSAGE = "不適当なコマンドでした";
71:   final String INCORRECT_INPUT_ERROR_MESSAGE = "数字を入力してください";
72:   final String COMMAND_PROMPT = " >> ";
73:   final String REMOVE_CONFIRM_MESSAGE = "本当に削除してもいいですか？ y/n";
74:   final String NO_EDITABLE_ELEMENT_MESSAGE = "編集可能なデータがありません";
75:   final String DATA_OVERFLOW_ERROR_MESSAGE = "データがいっぱいで、もう追加できません";
76:   final String COMPANY_NAME = "○×コミュニケーションズ";
77:   final String YES = "Y";
78:   final String SEPARATE_LINE = "\n"; //改行記号
79:
80:   //カーソル移動用
81:   final int UP = 1;
82:   final int DOWN = 2;
83:
84:   /*****************************************************
85:   /* インスタンス変数
86:   *****************************************************/
87:
88:   int currentRecordIndex = -1; //現在カーソルがあるインデックス。何もないときは-1
89:
90:   EmployeeList employees = new EmployeeList(); //社員を格納するリスト
91:
92:   /*****************************************************
93:   /* メイン
94:   *****************************************************/
95:
96:   void run() {
97:
98:     String command; //入力されたコマンド
99:
100:    //初期画面を表示
101:    updateView();
102:
103:    //コマンドの入力と実行
104:    while (!(command = inputCommand()).equals(QUIT)) {
105:      executeCommand(command);
106:      updateView();
107:    }
108:
109:    //終了処理
110:    System.exit(0);
111:  }
112:
113:  /*****************************************************
114:  /* コマンド処理部品
115:  *****************************************************/
```

```
116:
117:   /**
118:    * 社員名簿のコマンドを入力し、妥当なコマンドであれば、入力されたコマンドを返す。
119:    * もしも、不適当なコマンドであれば、再入力を要求する。
120:    */
121:   String inputCommand() {
122:
123:     String command; //入力されたコマンド
124:
125:     while (true) {
126:       //プロンプトを出す
127:       showPrompt(COMMAND_INFO_MESSAGE);
128:
129:       //入力する
130:       command = Input.getString();
131:       command = command.toUpperCase(); //入力されたコマンドを大文字に
132:
133:       //正しいコマンドならばループを抜けて入力コマンドを返す。
134:       //正しくなければエラーを表示して再入力
135:       if (isCorrectCommand(command)) {
136:         break;
137:       } else {
138:         showError(COMMAND_ERROR_MESSAGE);
139:       }
140:     }
141:     return command;
142:   }
143:
144:   /**
145:    * 指定された社員名簿のコマンドを実行する
146:    */
147:   void executeCommand(String command) {
148:     if (command.equals(ADD)) { //追加
149:       addEmployee();
150:     } else if (command.equals(REMOVE)) { //削除
151:       removeEmployee();
152:     } else if (command.equals(EDIT)) { //編集
153:       editEmployee();
154:     } else if (command.equals(UP_CURSOR)) { //カーソルを上へ
155:       moveCursor(UP);
156:     } else if (command.equals(DOWN_CURSOR)) { //カーソルを下へ
157:       moveCursor(DOWN);
158:     } else if (command.equals(SAVE)) { //セーブ
159:       save();
160:     } else if (command.equals(LOAD)) { //ロード
161:       load();
162:     }
163:   }
164:
165:   /*****************************************************
166:   /* コマンド群
167:   /*****************************************************/
168:
169:   /**
```

```
170:    * 社員名簿データに、新しい社員を追加する
171:    */
172:   void addEmployee() {
173:     //データが満杯だったらエラーを表示して終了する
174:     if (employees.isFull()) {
175:       showError(DATA_OVERFLOW_ERROR_MESSAGE);
176:       return;
177:     }
178:
179:     //データを入力する
180:     showPrompt("社員 ID");
181:     int id = getValidInt();
182:     showPrompt("所属部署 ");
183:     String section = Input.getString();
184:     showPrompt("名前 ");
185:     String name = Input.getString();
186:     showPrompt("住所 ");
187:     String address = Input.getString();
188:     showPrompt("生まれた年 ");
189:     int birthYear = getValidInt();
190:
191:     //入力されたデータから新しい社員を生成する
192:     Employee employee = new Employee(id, section, name, address, birthYear);
193:
194:     //社員を追加する
195:     employees.add(employee);
196:     currentRecordIndex = getDirectoryRecordCount() - 1;
197:   }
198:
199:   /**
200:    * 名簿データの削除を行う
201:    */
202:   void removeEmployee() {
203:
204:     //社員が存在しないとき
205:     if (employees.isEmpty()) {
206:       showError(NO_EDITABLE_ELEMENT_MESSAGE);
207:       return;
208:     }
209:
210:     //プロンプトを出す
211:     System.out.println(REMOVE_CONFIRM_MESSAGE);
212:
213:     //入力する
214:     String input = Input.getString();
215:     input = input.toUpperCase(); //入力されたコマンドを大文字に
216:
217:     //削除確認がとれなかったときはなにもしない
218:     if (!input.toUpperCase().equals(YES)) {
219:       return;
220:     }
221:
222:     //削除する
223:     employees.remove(currentRecordIndex);
```

```
224:
225:     //削除の後始末
226:     if (getDirectoryRecordCount() == currentRecordIndex) { //最後の要素が削除された場
合
227:       currentRecordIndex--;
228:     }
229:   }
230:
231:   /**
232:    * 社員の編集を行う
233:    */
234:   void editEmployee() {
235:
236:     //社員が存在しないとき
237:     if (employees.isEmpty()) {
238:       showError(NO_EDITABLE_ELEMENT_MESSAGE);
239:       return;
240:     }
241:
242:     //新しい情報を入力する
243:
244:     Employee employee = employees.get(currentRecordIndex);
245:
246:     //部署
247:     showPrompt("所属部署 " + employee.getSection());
248:     String inputSection = Input.getString();
249:     if (inputSection.equals("")) { //空白なら，変更しないとする
250:       inputSection = employee.getSection();
251:     }
252:
253:     //名前
254:     showPrompt("名前 " + employee.getName());
255:     String inputName = Input.getString();
256:     if (inputName.equals("")) { //空白なら，変更しないとする
257:       inputName = employee.getName();
258:     }
259:
260:     //住所
261:     showPrompt("住所 " + employee.getAddress());
262:     String inputAddress = Input.getString();
263:     if (inputAddress.equals("")) { //空白なら，変更しないとする
264:       inputAddress = employee.getAddress();
265:     }
266:
267:     //誕生年
268:     showPrompt("誕生年 " + employee.getBirthYear());
269:     int inputBirthYear = getValidInt();
270:
271:     //データの変更を行う
272:
273:     //新しいデータを作成する
274:     Employee newEmployee =
275:       new Employee(
276:         employee.getId(),
```

```
277:         inputSection,
278:         inputName,
279:         inputAddress,
280:         inputBirthYear);
281:
282:     //データの取り替え
283:     employees.remove(currentRecordIndex); //古いデータを削除
284:     employees.add(newEmployee); //新しいデータを追加
285:   }
286:
287:   /**
288:    * 指定された方向にカーソルを移動する
289:    */
290:   void moveCursor(int direction) {
291:
292:     int nextRecordIndex = 0; //次のカーソル位置
293:
294:     //データが存在しないとき
295:     if (getDirectoryRecordCount() == 0) {
296:       return;
297:     }
298:
299:     //次のカーソル位置を計算する
300:     switch (direction) {
301:       case UP : //上に移動
302:         nextRecordIndex = currentRecordIndex - 1;
303:         if (nextRecordIndex < 0) {
304:           nextRecordIndex = 0;
305:         }
306:         break;
307:       case DOWN : //下に移動
308:         nextRecordIndex = currentRecordIndex + 1;
309:         if (nextRecordIndex >= getDirectoryRecordCount()) {
310:           nextRecordIndex = getDirectoryRecordCount() - 1;
311:         }
312:         break;
313:       default :
314:         break;
315:     }
316:
317:     //次のカーソル位置を設定する
318:     currentRecordIndex = nextRecordIndex;
319:   }
320:
321:   /**
322:    * 名簿データのセーブを行う.
323:    */
324:   void save() {
325:
326:     //ファイル名を入力する
327:     showPrompt("セーブするファイル名を入力してください"); //プロンプト
328:     String fileName = Input.getString();
329:
330:     //前処理
```

```
331:     PrintStream writer = FileIO.openForWrite(fileName); //ファイルオープン
332:
333:     //書き込み
334:     for (int i = 0; i < getDirectoryRecordCount(); i++) {
335:       Employee employee = employees.get(i);
336:       //CSV 形式
337:       writer.println(
338:         employee.getId()
339:           + ","
340:           + employee.getSection()
341:           + ","
342:           + employee.getName()
343:           + ","
344:           + employee.getAddress()
345:           + ","
346:           + employee.getBirthYear());
347:     }
348:
349:     //後処理
350:     writer.close(); //ファイルクローズ
351:     System.out.println("正常にセーブされました");
352:   }
353:
354:   /**
355:    * 名簿データのロードを行う.
356:    */
357:   void load() {
358:
359:     //ファイル名を入力する
360:     showPrompt("ロードするファイル名を入力してください");
361:     String fileName = Input.getString();
362:
363:     //前処理
364:     ReadStream reader = FileIO.openForRead(fileName); //ファイルオープン
365:
366:     employees.removeAll(); //名簿を初期化
367:
368:     //読み込み
369:     while (!reader.isEnd()) {
370:       //CSV 形式のデータを分割する
371:       String line = reader.readLine();
372:       String[] elements = line.split(",");
373:
374:       //データを追加する
375:       Employee employee =
376:         new Employee(
377:           Integer.parseInt(elements[0]),
378:           elements[1],
379:           elements[2],
380:           elements[3],
381:           Integer.parseInt(elements[4]));
382:
383:       //社員を名簿に追加する
384:       employees.add(employee);
```

```
385:     }
386:
387:     //後処理
388:     reader.close(); //ファイルクローズ
389:     currentRecordIndex = 0; //カーソルの初期化
390:     System.out.println("正常にロードされました");
391:   }
392:
393:   /*****************************************************
394:   /* コマンド用部品
395:   *****************************************************/
396:
397:   /**
398:    * 数値の入力を受け取る
399:    * 入力が正しく無い場合は再入力を促す
400:    */
401:   int getValidInt() {
402:     while (true) {
403:       //入力する
404:       String input = Input.getString();
405:
406:       //入力が数字に変換可能なら入力された文字列を返す，そうでなければ再入力
407:       if (Input.isInteger(input)) {
408:         return Integer.parseInt(input);
409:       } else {
410:         showPrompt(INCORRECT_INPUT_ERROR_MESSAGE);
411:       }
412:     }
413:   }
414:
415:   /**
416:    * 指定されたコマンドが正しいコマンドかどうかを調べる
417:    */
418:   boolean isCorrectCommand(String command) {
419:     for (int i = 0; i < COMMANDS.length; i++) {
420:       if (COMMANDS[i].equals(command)) { //妥当なコマンドならば
421:         return true;
422:       }
423:     }
424:     return false;
425:   }
426:
427:   /*****************************************************
428:   /* 画面表示用部品
429:   *****************************************************/
430:
431:   /**
432:    * 指定されたエラーメッセージを表示する
433:    */
434:   void showError(String errorMessage) {
435:     System.out.print(errorMessage);
436:   }
437:
438:   /**
```

```
439:    * 指定されたメッセージを持つプロンプトを表示する
440:    */
441:   void showPrompt(String message) {
442:     System.out.println();
443:     System.out.print(message);
444:     System.out.print(COMMAND_PROMPT);
445:     System.out.flush();
446:   }
447:
448:   /**
449:    * 表示を更新する
450:    */
451:   void updateView() {
452:     showTitle();
453:     showDirectory();
454:   }
455:
456:   /**
457:    * タイトルを表示する
458:    */
459:   void showTitle() {
460:     System.out.println(COMPANY_NAME + "社員管理システム");
461:   }
462:
463:   /**
464:    * 社員の情報を 1 画面に書く。
465:    */
466:   void showDirectory() {
467:     for (int i = 0; i < getDirectoryRecordCount(); i++) {
468:       showDirectoryRecord(i);
469:     }
470:   }
471:
472:   /**
473:    * 社員一人分の情報を表示する
474:    */
475:   void showDirectoryRecord(int recordIndex) {
476:
477:     setMark(recordIndex);
478:
479:     Employee employee = employees.get(recordIndex);
480:     System.out.println(
481:       "\t"
482:         + (recordIndex + 1)
483:         + "\t"
484:         + employee.getId()
485:         + "\t"
486:         + employee.getSection()
487:         + "\t"
488:         + employee.getName()
489:         + "\t"
490:         + employee.getAddress());
491:     System.out.println(
492:       "\t"
```

```
493:         + "\t"
494:         + employee.getBirthYear()
495:         + "年生まれ\t"
496:         + employee.getAge()
497:         + "才");
498:
499:     System.out.println(LINE);
500:   }
501:
502:   /**
503:    * 注目行マークをつける
504:    */
505:   void setMark(int recordIndex) {
506:     if (recordIndex == currentRecordIndex) {
507:       System.out.print("*");
508:     } else {
509:       System.out.print("  ");
510:     }
511:   }
512:
513:   /**
514:    * 名簿の要素数を取得する
515:    */
516:   int getDirectoryRecordCount() {
517:     return employees.size();
518:   }
519: }
```

考えてみよう

コンストラクタを定義することで、どのようにプログラムが書きやすくなるでしょう？

9.3　意味のまとまりとしてのオブジェクト

9.3.1　アルゴリズムとデータ構造を結合する意義

9.3.1.1　アルゴリズムとデータ構造の結合

7, 8, 9 章では、配列を管理する社員リストクラスと、社員の情報を管理する社員クラスを導入してきました。このようなクラスの導入を行うことで、一度に管理する範囲が狭められ、それぞれ個別に管理できるために、全体として複雑なアプリケーションを管理することが容易になります。

しかし、ただ単にクラスを導入しただけでは、個別に管理できるようになるとは限りません。7 章で登場したインスタンスメソッドを持たない社員クラスを思い出してみましょう。この段階の社員クラスでは、年齢などを計算で求めるような変更を行うと、アプリケーションのクラスまで全て変更する必要がでてしまいます。

それぞれのクラスに、管理したいデータ構造と、それを操作するアルゴリズムをまとめることで、データ構造を隠蔽することができます。さらに、オブジェクトに対して明確なインターフェイスを定義することで、オブジェクトを使う人はオブジェクトの持つ意味のあるデータにだけ注目することができます。このようなことがオブジェクト指向の最も重要な意義です。

9.3.1.2　クラスの責任

データ構造とアルゴリズムを結合し、それをまとめたクラスをつくると、バグが発生したり、変更の必要がでたりしたときにどこのクラスをみればいいのかが明らかになります。これは、クラスが一つの意味について責任を持ち、それについての処理はすべてその責任を持つクラスに書かれているからです。

考えてみよう

今までに社員名簿アプリケーションに登場したそれぞれのクラスの責任を考えてみましょう。

9.3.2 クラスが持つ責任とテスト

9.3.2.1 単体テスト

データ抽象の考え方を用い、クラスの責任を明らかにすると、そのクラスがどのようなことを保証しているのかを決めることができます。それぞれのクラスが保証していることをテストすることで、 より信頼性の高いソフトウェアをつくることができます。

リスト 78: EmployeeList のテスト

```
 1: /**
 2:  * 社員リストをテストするテストクラス
 3:  *
 4:  * 社員リストクラスは、社員の管理に責任を持つ
 5:  * このクラスでは以下の公開メソッドをテストする
 6:  *   add()
 7:  *   get()
 8:  *   remove()
 9:  *   removeAll();
10:  *   isEmpty();
11:  *
12:  * @author bam
13:  * @version $Id: EmployeeListTest.java,v 1.2 2003/05/04 12:10:30 macchan Exp $
14:  */
15: public class EmployeeListTest {
16:
17:   public static void main(String[] args) {
18:     EmployeeListTest employeeListTest = new EmployeeListTest();
19:     employeeListTest.test();
20:   }
21:
22:   EmployeeList employees;
23:   Employee testEmployee1;
24:   Employee testEmployee2;
25:   Employee testEmployee3;
26:
27:   private void test() {
28:
29:     //初期設定を行う
30:     employees = new EmployeeList();
31:     testEmployee1 = new Employee(1, "", "", "", 1);
32:     testEmployee2 = new Employee(2, "", "", "", 2);
33:
34:     //isEmpty() のテスト
35:     if (!employees.isEmpty()) {
36:       System.out.println("isEmpty が間違っています。");
37:     }
38:
39:     //add() のテスト
40:     employees.add(testEmployee1); //1 人目
41:     if (employees.size() != 1) {
```

```
42:       System.out.println("add が間違っています。");
43:     }
44:     employees.add(testEmployee2); //２人目
45:     if (employees.size() != 2) {
46:       System.out.println("add が間違っています。");
47:     }
48:
49:     //get() のテスト
50:     if (!(employees.get(0) == testEmployee1)) {
51:       System.out.println("get が間違っています。");
52:     }
53:
54:     //remove() のテスト
55:     employees.remove(0);
56:     if (employees.size() != 1) {
57:       System.out.println("remove が間違っています。");
58:     }
59:     if (!(employees.get(0) == testEmployee2)) {
60:       System.out.println("remove が間違っています。");
61:     }
62:
63:     //removeAll() のテスト
64:     employees.removeAll();
65:     if (employees.size() != 0) {
66:       System.out.println("removeAll が間違っています。");
67:     }
68:   }
69:
70: }
```

9.3.2.2 何をテストするか

テストは対象となるクラスの外部に公開された public なメソッドについてテストします。

9.3.3 スタック

データ構造とアルゴリズムの結合について学んだところで、その考え方をつかって、もう少し複雑な構造とアルゴリズムを持つデータ構造に挑戦してみましょう。

9.3.3.1 スタックとは

現状の社員名簿は一度削除したら削除したデータは復旧することができません。しかし、一般的なアプリケーションでは、削除などの重要な操作は、取り消すことができるのが一般的です。社員名簿に Undo 機能を実装してみましょう。

Undo 機能を実装するためには、社員名簿のデータを管理する EmployeeList とは別に、削除された社員をとっておくためのクラスが必要になります。

また、このクラスは、今までの「空いてるところに追加して、どのデータでもインデックスで取得できる」というものではなく、「削除したものを順番に追加しておいて、Undo のときには一番最後に削除したものを取得する」というクラスである必要があります。このような、追加するときは最後に追加して、取得するときには一番最後に入っているデータを取得するというデータ構造をスタックといいます。

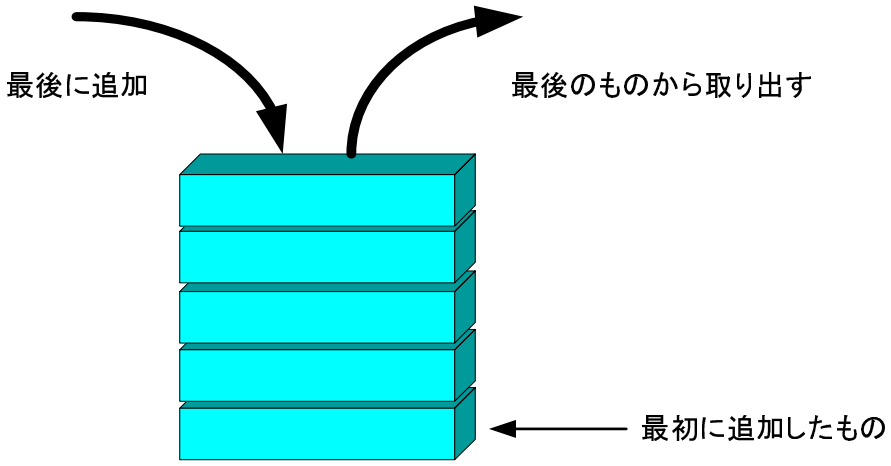


図 9.4: スタック

考えてみよう

身の回りにあるスタックをつかったシステム（情報システムでなくてもかまいません）をあげてみましょう。

9.3.3.2 スタックの実装

ここでは配列をつかってスタックを実装します。スタックを配列で実装するには、要素を格納する配列と、次に要素を追加するカーソルを表す整数の値が必要になります。

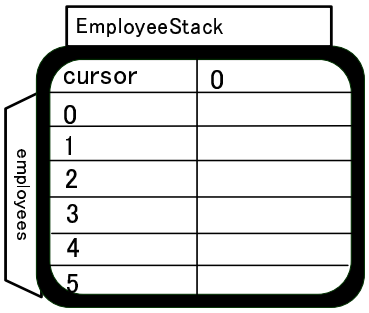


図 9.5: スタックオブジェクト

リスト 79: 社員スタッククラス

```
 1: /**
 2:  * 社員スタッククラス
 3:  * 一度削除された社員を削除された逆順に保管しておく
 4:  *
 5:  * @author bam
 6:  * @version $Id: EmployeeStack.java,v 1.2 2003/05/08 16:54:53 bam Exp $
 7:  */
 8: public class EmployeeStack {
 9:
10:   private final int ARRAY_SIZE = 6; //配列の大きさ
11:
12:   private Employee[] employees = new Employee[ARRAY_SIZE]; //社員を保存する配列
13:   private int cursor = 0; //追加・削除する際に使うカーソル
14:
15:   /**
16:    * 社員をスタックに積む
17:    */
18:   public void push(Employee employee) {
19:     employees[cursor] = employee;
20:     cursor++;
21:   }
22:
23:   /**
24:    * スタックから社員を取り出す
25:    */
26:   public Employee pop() {
27:
28:     if (cursor <= 0) { //取り出す社員がない場合
29:       return null;
30:     }
```

```
31:
32:     Employee employee = employees[--cursor];
33:     return employee;
34:   }
35:
36:   /**
37:    * スタックが空かどうかを調べる
38:    */
39:   public boolean isEmpty() {
40:     return cursor == 0;
41:   }
42: }
```

9.3.3.3　要素の追加

スタックへの要素の追加は配列の指定された追加カーソルの位置に要素を追加するだけです。また、次に要素を追加する時のために追加カーソルを一つ進める必要があります。

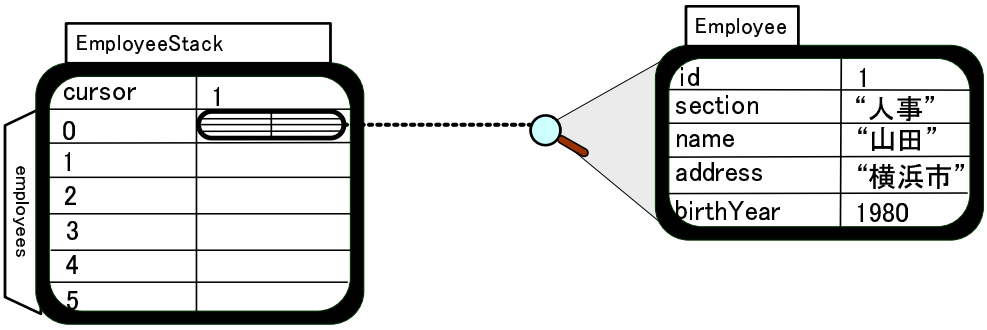
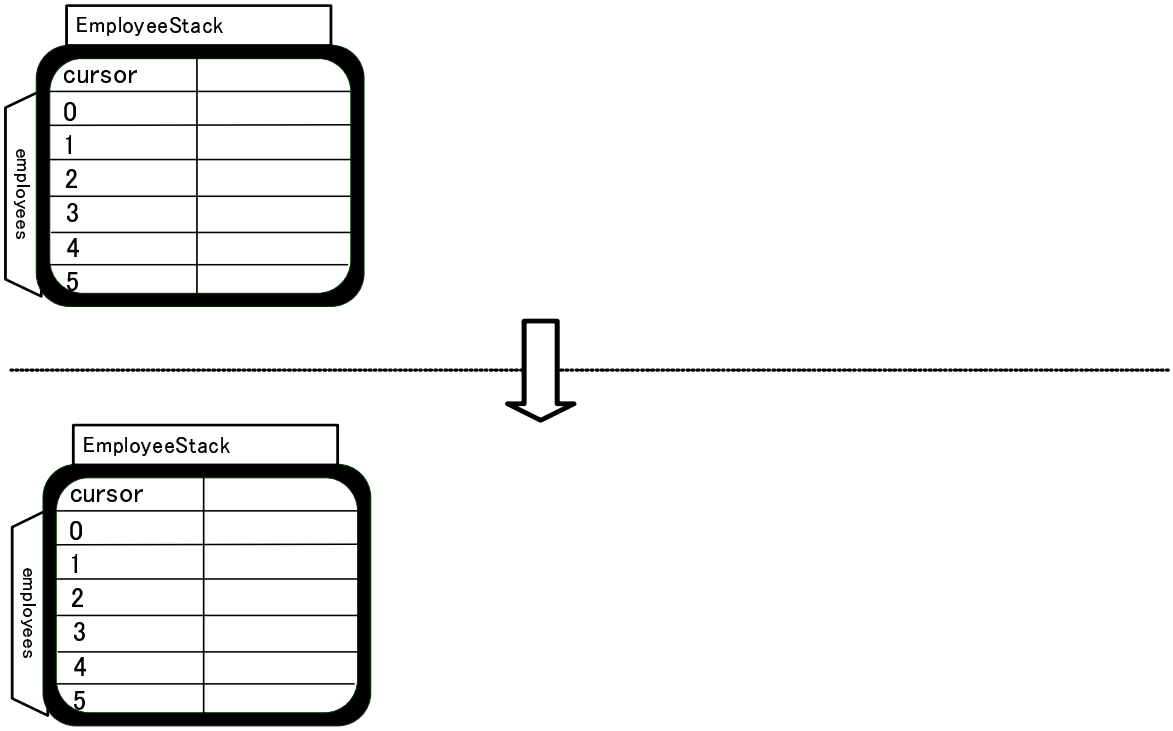


図 9.6: スタックへの要素の追加

要素を一つずつ追加していくとどのように変化するか記入してみましょう。



9.3.3.4　要素の取得

要素の取得では、追加カーソルの一つ前の位置にある配列の要素を返します。まず追加カーソルをひとつ戻し、その位置にある要素を返すと同時に返した要素を配列から削除する必要があります。

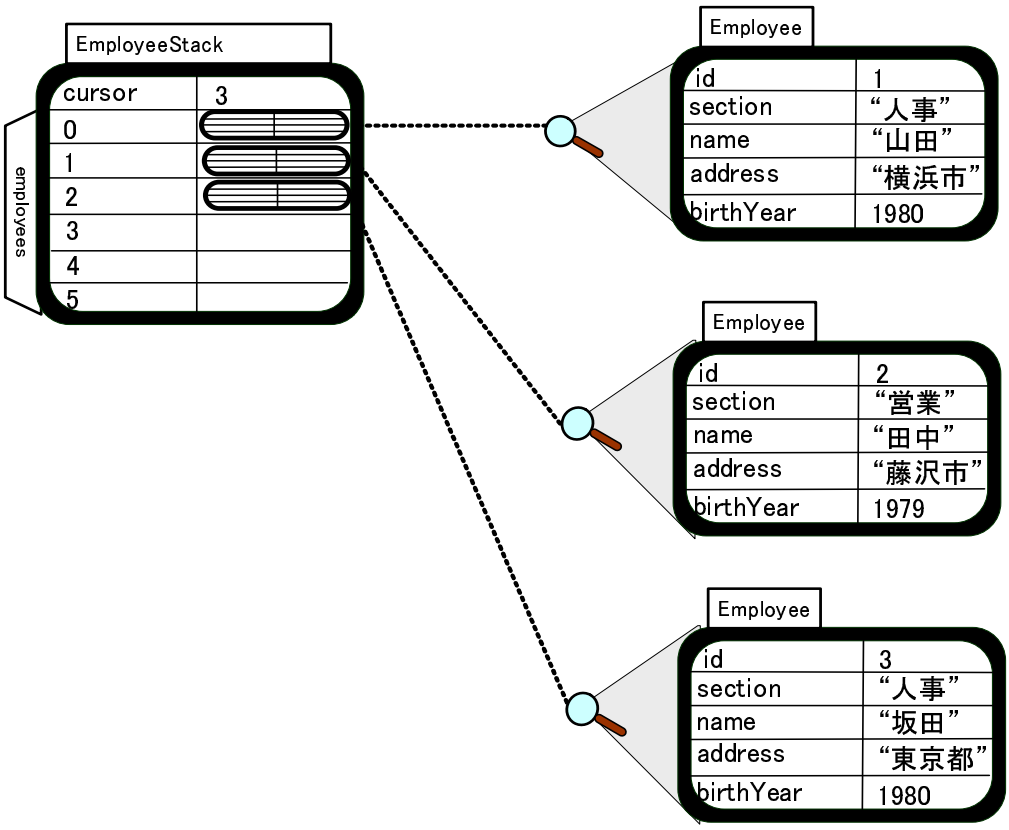


図 9.7: 取得前のスタック

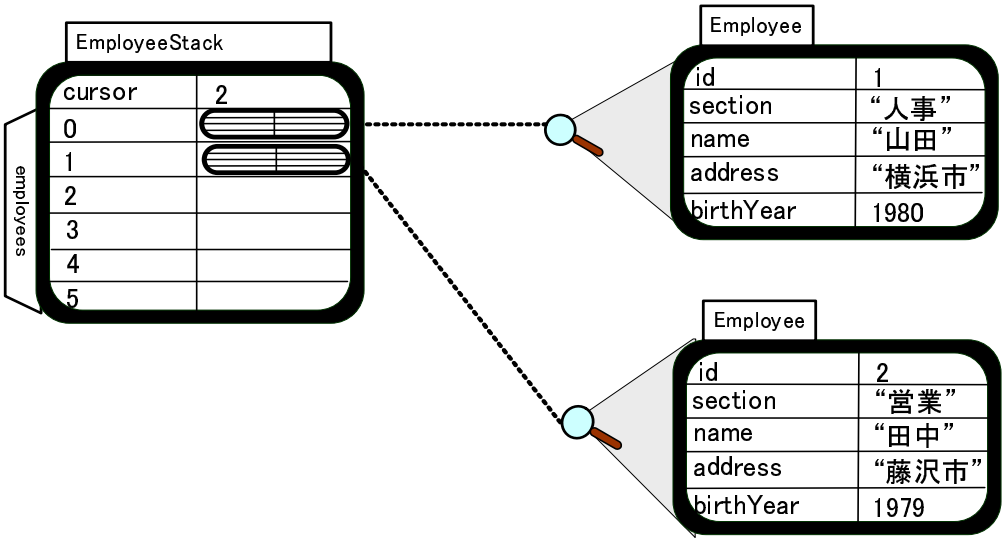
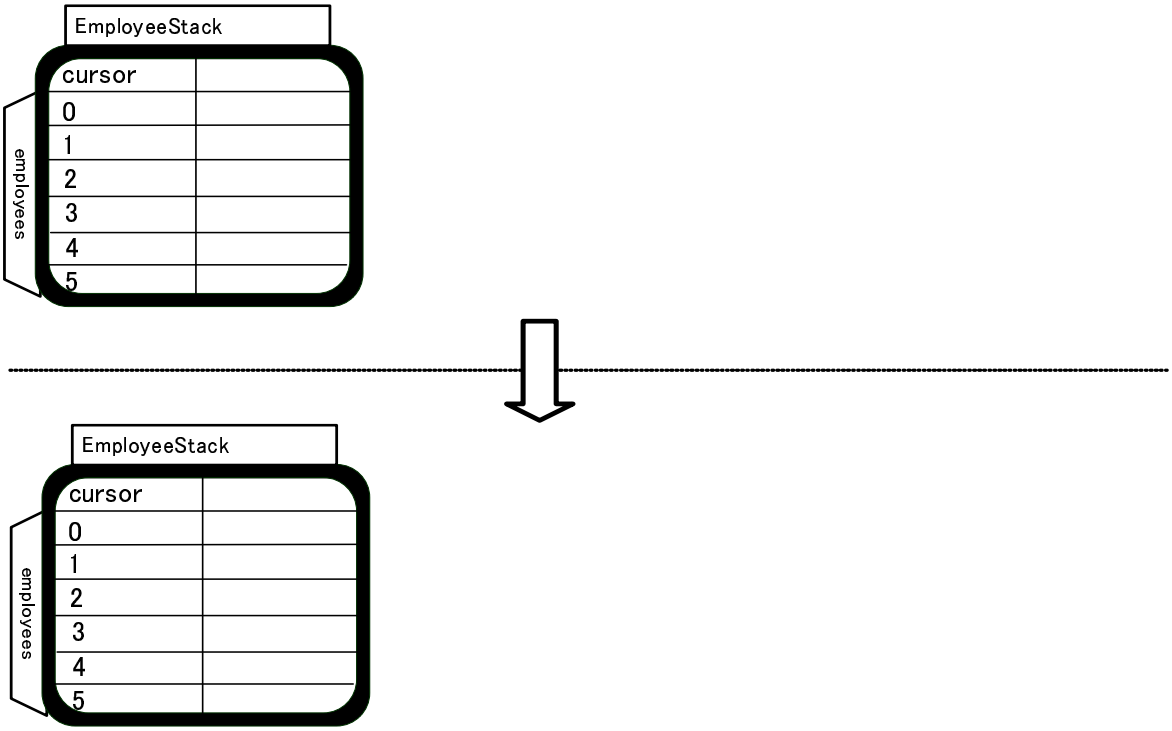


図 9.8: スタックからの要素の取得

要素を一つずつ取得していくとどのように変化するか記入してみましょう。



9.3.4　キュー

9.3.4.1　キューとは

スタックは最後に追加して最後のデータから取得するというものでしたが、これとは逆に、データの最後に追加して取得するときには最初から取り出すというデータ構造もあります。このようなデータ構造のことをキューといいます。

キューを使ったシステムの例として、自動販売機アプリケーションを実装することを考えて見ましょう。自動販売機は入荷した商品を在庫の一番最後に追加して、ジュースが買われたときには、在庫の最初に入っている一番古い商品を出すキューをつかった典型的なシステムです。

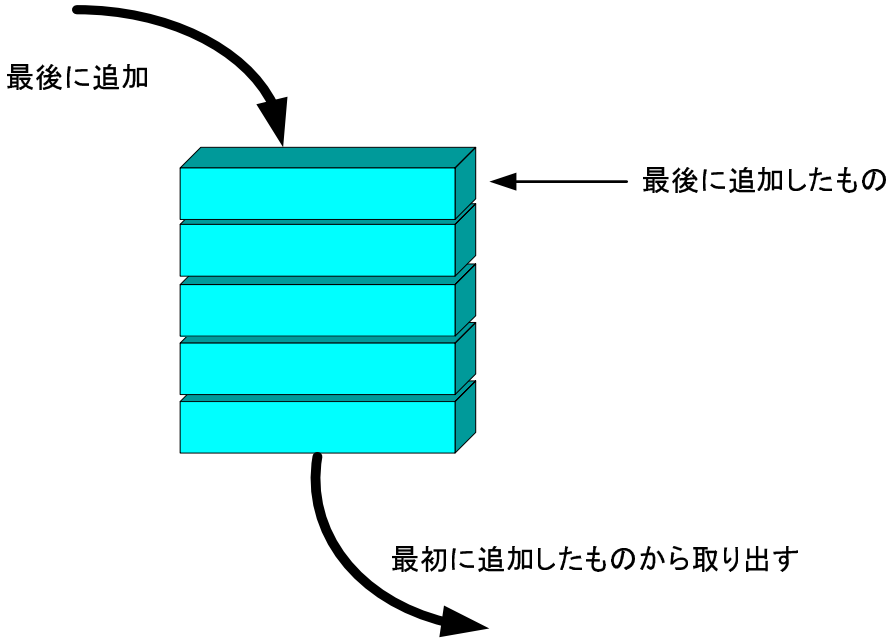


図 9.9: キュー

考えてみよう

身の回りにあるキューをつかったシステム（情報システムでなくてもかまいません）をあげてみましょう。

9.3.4.2 キューの実装

ここでは配列を使ってキューを実装します。キューを実装するには、要素を格納する配列と、次に要素を追加するカーソルを表す整数の値が必要になります。

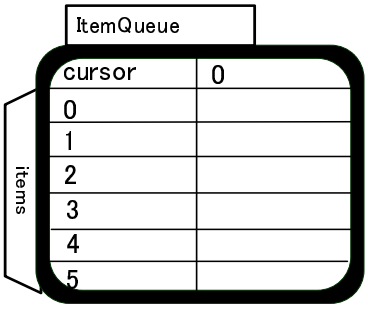


図 9.10: キューのオブジェクト

リスト 80: 商品キュークラス

```
 1: /**
 2:  * 商品キュークラス
 3:  * 商品を日付の若い順に保管しておく
 4:  *
 5:  * @author kei
 6:  * @version $Id: ItemQueue.java,v 1.2 2003/05/06 09:23:57 bam Exp $
 7:  */
 8: public class ItemQueue {
 9:
10:   private final int ARRAY_SIZE = 6; //配列のサイズ
11:
12:   private Item[] items = new Item[ARRAY_SIZE]; //商品を格納する配列
13:   private int cursor = 0; //追加カーソル
14:
15:   /**
16:    * 商品を補充する
17:    */
18:   public void enqueue(Item item) {
19:     items[cursor] = item;
20:     cursor++;
21:   }
22:
23:   /**
24:    * 商品を取り出す
25:    */
26:   public Item dequeue() {
27:
28:     //最初に補充されたものを取り出す
29:     Item item = items[0];
30:     cursor--;
```

```
31:
32:     //配列を詰める
33:     for (int i = 0; i < cursor; i++) {
34:       items[i] = items[i + 1];
35:     }
36:
37:     return item;
38:   }
39:
40:   /**
41:    * キューが満杯かどうか調べる
42:    */
43:   public boolean isFull() {
44:     return cursor >= ARRAY_SIZE;
45:   }
46:
47:   /**
48:    * キューが空かどうか調べる
49:    */
50:   public boolean isEmpty() {
51:     return cursor == 0;
52:   }
53: }
```

9.3.4.3 要素の追加

キューへの要素の追加は配列の指定された追加カーソルの位置に要素を追加するだけです。また次に要素が追加されたときのために、追加カーソルを一つ進めます。

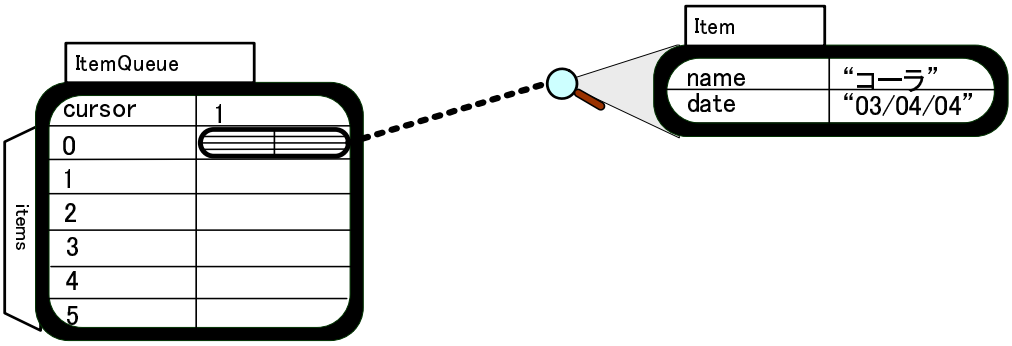
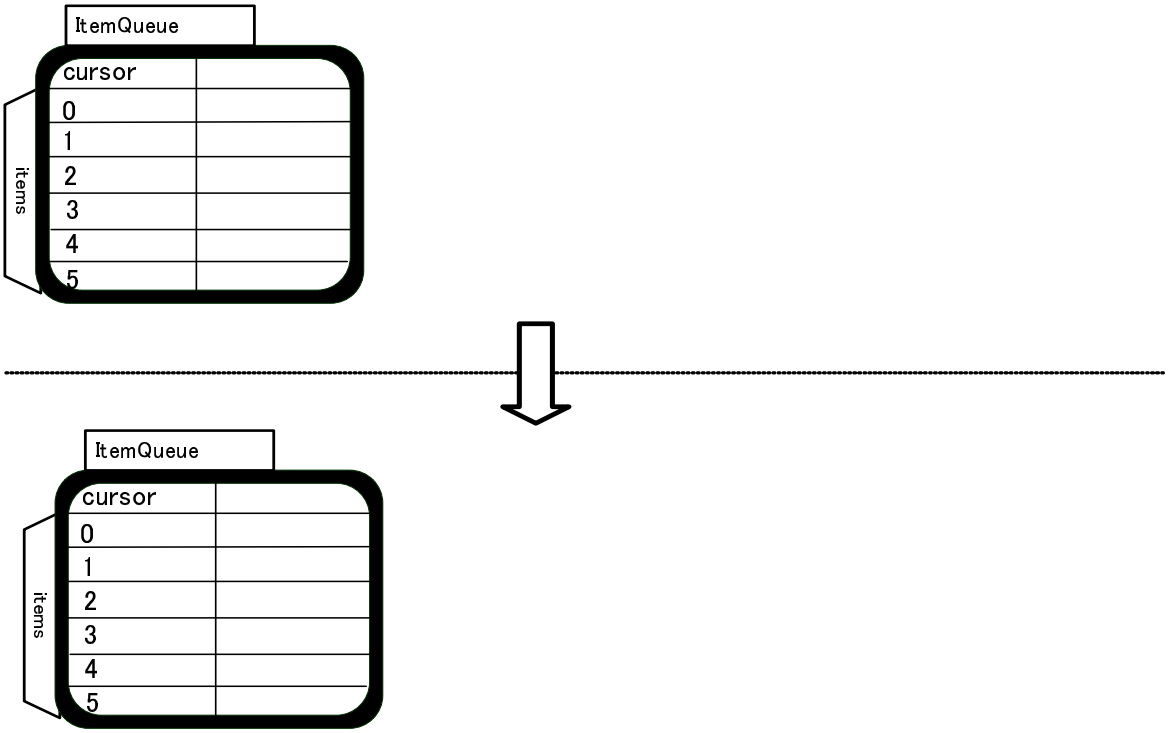


図 9.11: キューへの要素の追加

要素を一つずつ追加していくとどのように変化するか記入してみましょう。



9.3.4.4 要素の取得

要素の取得では、配列の先頭の要素を返します。また要素を返すと同時に返した要素を配列から削除し、残りの要素をつめる必要があります。

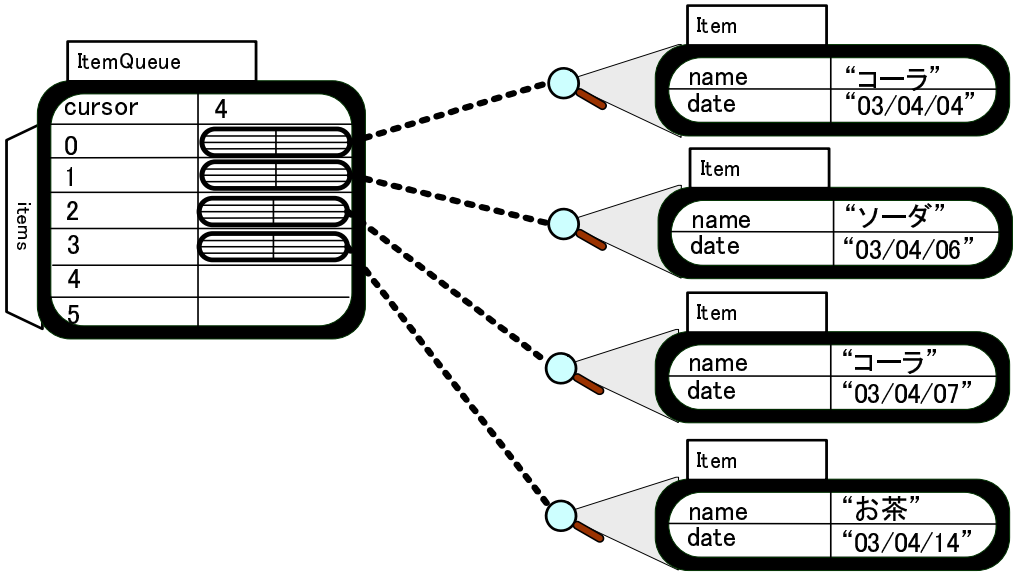


図 9.12: 取得前のキュー

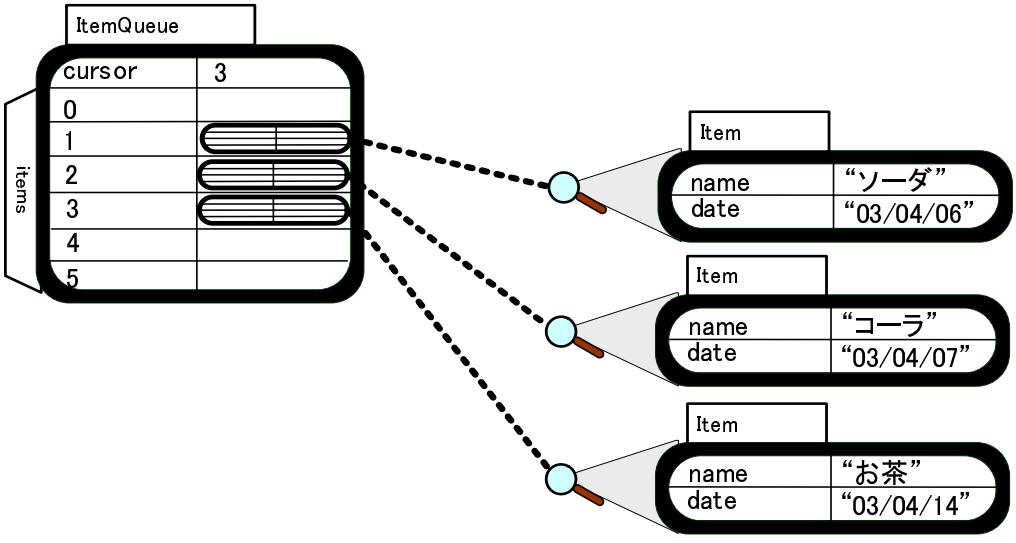


図 9.13: キューからの要素の取得 (1)

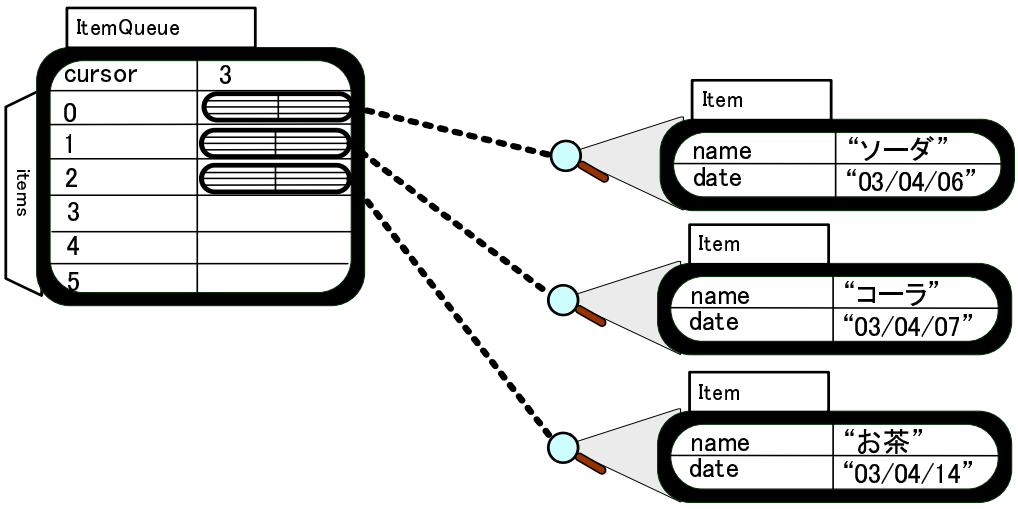
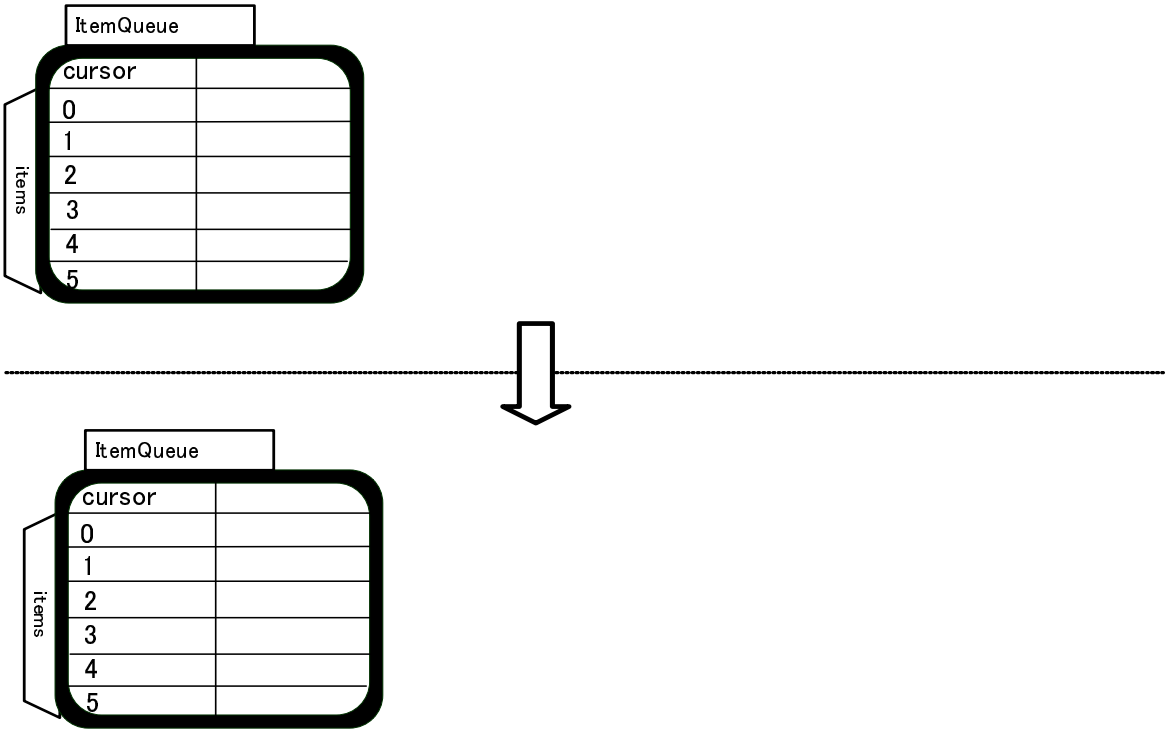


図 9.14: キューからの要素の取得 (2)

要素を一つずつ取得していくとどのように変化するか記入してみましょう。



9.4 練習問題

練習問題 1

8 章で作成した辞書アプリケーション（DictionaryApplication.java）をカプセル化し、コンストラクタを導入してください。

練習問題 2

EmployeeStackTest.java を実装して、スタックのテストプログラムを作ってください。

練習問題 3

ItemQueueTest.java を実装して、キューのテストプログラムを作ってください。

第 10 章

オブジェクトのネットワーク構造 (1)

この章で学習すること

基本データ型と参照型の違いを説明できる
インスタンスの関係をインスタンスの参照モデルで図に書くことができる

10.1 参照

10.1.1 インスタンスの参照モデル

10.1.1.1 部署の管理をする社員名簿アプリケーション

これまで、社員名簿アプリケーションは社員の情報だけを管理するアプリケーションでした。しかし、実際に企業の名簿管理をするには社員それぞれの管理をする以外にも必要なことがあります。そのひとつが部署の管理です。今までの社員管理アプリケーションは部署をユーザーそれぞれが持つ値として管理してきましたが、実際には部署の名前変更など、部署単位で管理する必要があるはずです。今回は今まで作ってきた社員名簿アプリケーションに機能の追加を行い、部署の管理をできるようにします。

10.1.1.2 部署の名前を変更する

部署の名前が変更になったときに、部署に所属する社員の情報を一人一人すべて変更していくのは大変です。この章では社員名簿管理アプリケーションを、部署の名前を一括で変更することができるようにしていきます。

考えてみよう

現状の社員名簿アプリケーションの構造で、「人材部」に所属する社員を全員「人事部」に変更する、というような変更を行う機能を追加するためには、どのようなプログラムを書く必要があるでしょう。

10.1.1.3 同じものはひとつのオブジェクトに

部署の変更が容易でないのは同じ部署に所属する社員がそれぞれ違う String のオブジェクトを持っているからです。本来同じものをあらわすはずの部署をひとつのオブジェクトであらわすことはできないのでしょうか？そうすることで変更の個所もひとつにできるはずです。

考えてみよう

複数の社員クラスのインスタンスが同じ部署クラスのインスタンスを持つ、というのはどのようなことなのでしょうか?入れ子モデルで表してみましょう。

10.1.1.4　入れ子モデルの矛盾

二つのインスタンスが同じインスタンスを持っている。という状態は、入れ子モデルで表すことはできません。なぜなら、実際にプログラムの中ではインスタンスは他のインスタンスを内部に持っているわけではないからです。

10.1.1.5　オブジェクトの参照モデル

実際にはインスタンスは他のインスタンスを内部に持っているのではなく、他のインスタンスのメモリ内での所在地を”指している”のです。この”指している”ことを参照を持つといいます。オブジェクトを持っているとは実はオブジェクトへの参照を持っているということなのです。これをインスタンスの参照モデル [1] [2]という図を使って考えてみましょう。

参照モデルでは他のインスタンスへの参照を持っていることを矢印で表します。それ以外のことは入れ子モデルと同じです。

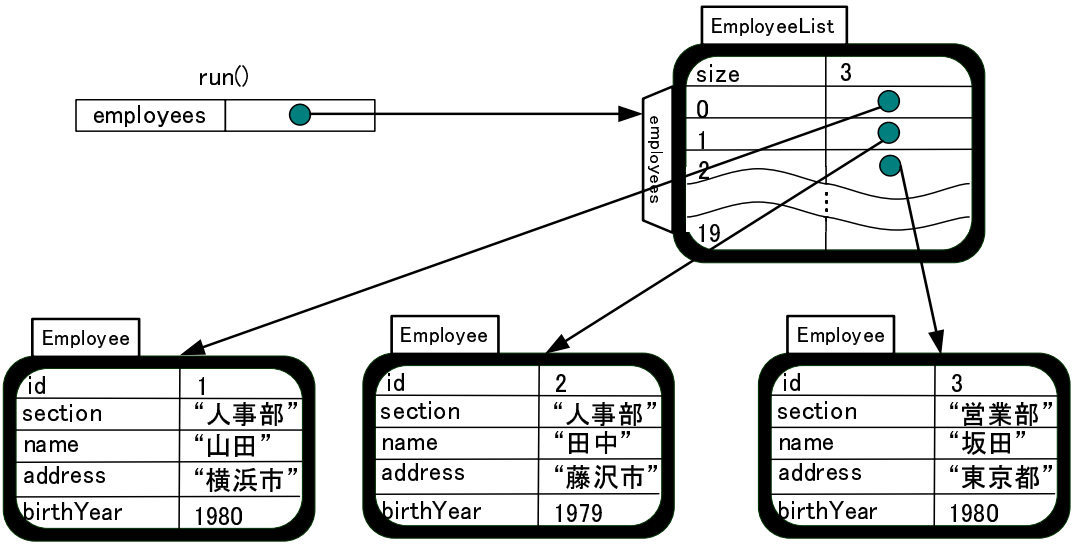


図 10.1: 参照モデル例

[1] このモデルにより 4.2.2.3 節で配列が同期していた、という説明をした訳が明らかになるでしょう。正確には、同期していたのではなく、同じオブジェクトを指していたのです。

[2] Java では配列や文字列もオブジェクトですので、正確な参照モデルを書こうとすると、配列や文字列も矢印で示す必要があります。しかし、そのような記述では図が複雑になってしまうので、本テキストでは、配列と文字列は、あたかも参照ではないように記述しています。

10.1.1.6 部署クラス

参照の考え方を使うとひとつの部署を複数の社員が指しているという構造を作ることができます。

リスト 81: 社員クラス (参照を使った場合)

```
 1: import java.io.Serializable;
 2:
 3: /**
 4:  * 社員クラス
 5:  *
 6:  * @author bam
 7:  * @version $Id: Employee.java,v 1.3 2003/05/07 03:51:55 macchan Exp $
 8:  */
 9: public class Employee implements Serializable {
10:
11:   private int id; //社員 ID
12:   private Section section; //部署
13:   private String name; //漢字の名前
14:   private String address; //住所
15:   private int birthYear; //誕生
16:
17:   /**
18:    * コンストラクタ
19:    */
20:   public Employee(
21:     int id,
22:     Section section,
23:     String name,
24:     String address,
25:     int birthYear) {
26:     this.id = id;
27:     this.section = section;
28:     this.name = name;
29:     this.address = address;
30:     this.birthYear = birthYear;
31:   }
32:
33:   /**
34:    * 社員 ID を取得する
35:    */
36:   public int getId() {
37:     return id;
38:   }
39:
40:   /**
41:    * 部署を取得する
42:    */
43:   public Section getSection() {
44:     return section;
```

```
45:   }
46:
47:   /**
48:    * 部署を設定する
49:    */
50:   public void setSection(Section section) {
51:     this.section = section;
52:   }
53:
54:   /**
55:    * 名前を取得する
56:    */
57:   public String getName() {
58:     return name;
59:   }
60:
61:   /**
62:    * 名前を設定する
63:    */
64:   public void setName(String name) {
65:     this.name = name;
66:   }
67:
68:   /**
69:    * 住所を取得する
70:    */
71:   public String getAddress() {
72:     return address;
73:   }
74:
75:   /**
76:    * 住所を設定する
77:    */
78:   public void setAddress(String address) {
79:     this.address = address;
80:   }
81:
82:   /**
83:    * 誕生年を取得する
84:    */
85:   public int getBirthYear() {
86:     return birthYear;
87:   }
88:
89:   /**
90:    * 誕生年を設定する
91:    */
92:   public void setBirthYear(int birthYear) {
93:     this.birthYear = birthYear;
94:   }
95:
96:   /**
97:    * 年齢を取得する
98:    */
```

```
 99:   public int getAge() {
100:     int age = Calendar.getCurrentYear() - birthYear;
101:     return age;
102:   }
103: }
```

リスト 82: 部署クラス (参照を使った場合)

```
 1: import java.io.Serializable;
 2:
 3: /**
 4:  * 部署クラス
 5:  *
 6:  * @author bam
 7:  * @version $Id: Section.java,v 1.3 2003/05/07 03:51:55 macchan Exp $
 8:  */
 9: public class Section implements Serializable {
10:
11:   private int id; //部署 ID
12:   private String name; //部署名
13:
14:   /**
15:    * コンストラクタ
16:    */
17:   public Section(int id, String name) {
18:     this.id = id;
19:     this.name = name;
20:   }
21:
22:   /**
23:    * 部署の ID を取得する
24:    */
25:   public int getId() {
26:     return id;
27:   }
28:
29:   /**
30:    * 部署の名前を取得する
31:    */
32:   public String getName() {
33:     return name;
34:   }
35:
36:   /**
37:    * 部署の名前を設定する
38:    */
39:   public void setName(String name) {
40:     this.name = name;
41:   }
42:
43: }
```

また変更などを行うためにはアプリケーションのクラスが、全ての部署への参照を持つ必要があります。これを管理するために社員のときと同じように部署リストクラスを導入します。

リスト 83: 部署リストクラス

```
 1: import java.io.Serializable;
 2:
 3: /**
 4:  * 部署を格納するリストクラス
 5:  *
 6:  * @author bam
 7:  * @version $Id: SectionList.java,v 1.3 2003/05/07 03:51:55 macchan Exp $
 8:  */
 9: class SectionList implements Serializable {
10:
11:   private final int ARRAY_SIZE = 100; //配列の大きさ
12:
13:   private Section[] sections; //部署を保存する配列
14:   private int size; //要素数
15:
16:   /**
17:    * コンストラクタ
18:    */
19:   public SectionList() {
20:     sections = new Section[ARRAY_SIZE];
21:   }
22:
23:   /**
24:    * 部署を追加する
25:    */
26:   public void add(Section section) {
27:     sections[size] = section;
28:     size++;
29:   }
30:
31:   /**
32:    * 指定されたリストの部署全員を追加する
33:    */
34:   public void addAll(SectionList sections) {
35:     for (int i = 0; i < sections.size(); i++) {
36:       add(sections.get(i));
37:     }
38:   }
39:
40:   /**
41:    * 部署を削除する
42:    */
43:   public void remove(Section section) {
44:
```

```
45:     int i; //ループ用
46:
47:     //削除対象の部署を見つける
48:     for (i = 0; i < size; i++) {
49:       if (sections[i] == section) { //見つかった
50:         sections[i] = null; //見つかったら、削除する（実は不要）
51:         break;
52:       }
53:     }
54:
55:     //削除する
56:     for (; i < size - 1; i++) { //要素を詰める
57:       sections[i] = sections[i + 1];
58:     }
59:     size--;
60:   }
61:
62:   /**
63:    * 指定された番地の部署を削除する
64:    */
65:   public void remove(int index) {
66:     for (int i = index; i < size - 1; i++) { //要素を詰める
67:       sections[i] = sections[i + 1];
68:     }
69:     size--;
70:   }
71:
72:   /**
73:    * 全ての要素を削除する
74:    */
75:   public void removeAll() {
76:     sections = new Section[ARRAY_SIZE];
77:     size = 0;
78:   }
79:
80:   /**
81:    * 指定された番地の部署を取得する
82:    */
83:   public Section get(int index) {
84:     return sections[index];
85:   }
86:
87:   /**
88:    * 部署数を取得する
89:    */
90:   public int size() {
91:     return size;
92:   }
93:
94:   /**
95:    * 部署が空かどうか調べる
96:    */
97:   public boolean isEmpty() {
98:     return size == 0;
```

```
 99:   }
100:
101:   /**
102:    * 指定されたＩＤの部署を検索する
103:    */
104:   public Section search(int id) {
105:     for (int i = 0; i < size; i++) {
106:       if (sections[i].getId() == id) {
107:         return sections[i];
108:       }
109:     }
110:     return null;
111:   }
112:
113: }
```

このような構造を定義し、複数の社員のインスタンスが同じ部署への参照を持つようなプログラムを記述します。

リスト 84: 部署を参照としてもつサンプルプログラム

```
 1: /**
 2:  * 部署クラスを用いたサンプルプログラム
 3:  *
 4:  * @author bam
 5:  * @version $Id: SectionSample.java,v 1.1 2003/05/07 06:16:51 macchan Exp $
 6:  */
 7: public class SectionSample {
 8:
 9:   public static void main(String[] args) {
10:     SectionSample sectionSample = new SectionSample();
11:     sectionSample.run();
12:   }
13:
14:   private void run() {
15:
16:     SectionList sections = new SectionList();
17:     EmployeeList employees = new EmployeeList();
18:
19:     //部署を生成
20:     sections.add(new Section(1, "人事部"));
21:     sections.add(new Section(2, "営業部"));
22:
23:     //社員を生成
24:     employees.add(new Employee(1, sections.search(1), "山田", "横浜市", 1980));
25:     employees.add(new Employee(2, sections.search(1), "田中", "横浜市", 1981));
26:     employees.add(new Employee(3, sections.search(2), "坂田", "東京都", 1980));
27:
28:     //社員リストを表示
29:     showEmployeeList(employees);
30:   }
```

```
31:
32:   /**
33:    * 社員リストを表示する
34:    */
35:   private void showEmployeeList(EmployeeList employees) {
36:     for (int i = 0; i < employees.size(); i++) {
37:       Employee employee = employees.get(i);
38:       showEmployee(employee);
39:     }
40:   }
41:
42:   /**
43:    * 社員一人分の情報を表示する
44:    */
45:   private void showEmployee(Employee employee) {
46:     System.out.println(
47:       "\t"
48:         + employee.getId()
49:         + "\t"
50:         + employee.getSection().getName()
51:         + "\t"
52:         + employee.getName()
53:         + "\t"
54:         + employee.getAddress()
55:         + "\t"
56:         + employee.getBirthYear()
57:         + "年生まれ\t"
58:         + employee.getAge()
59:         + "才");
60:   }
61: }
```

このサンプルプログラムが生成するオブジェクトの参照モデルは図 10.2 のようになります。

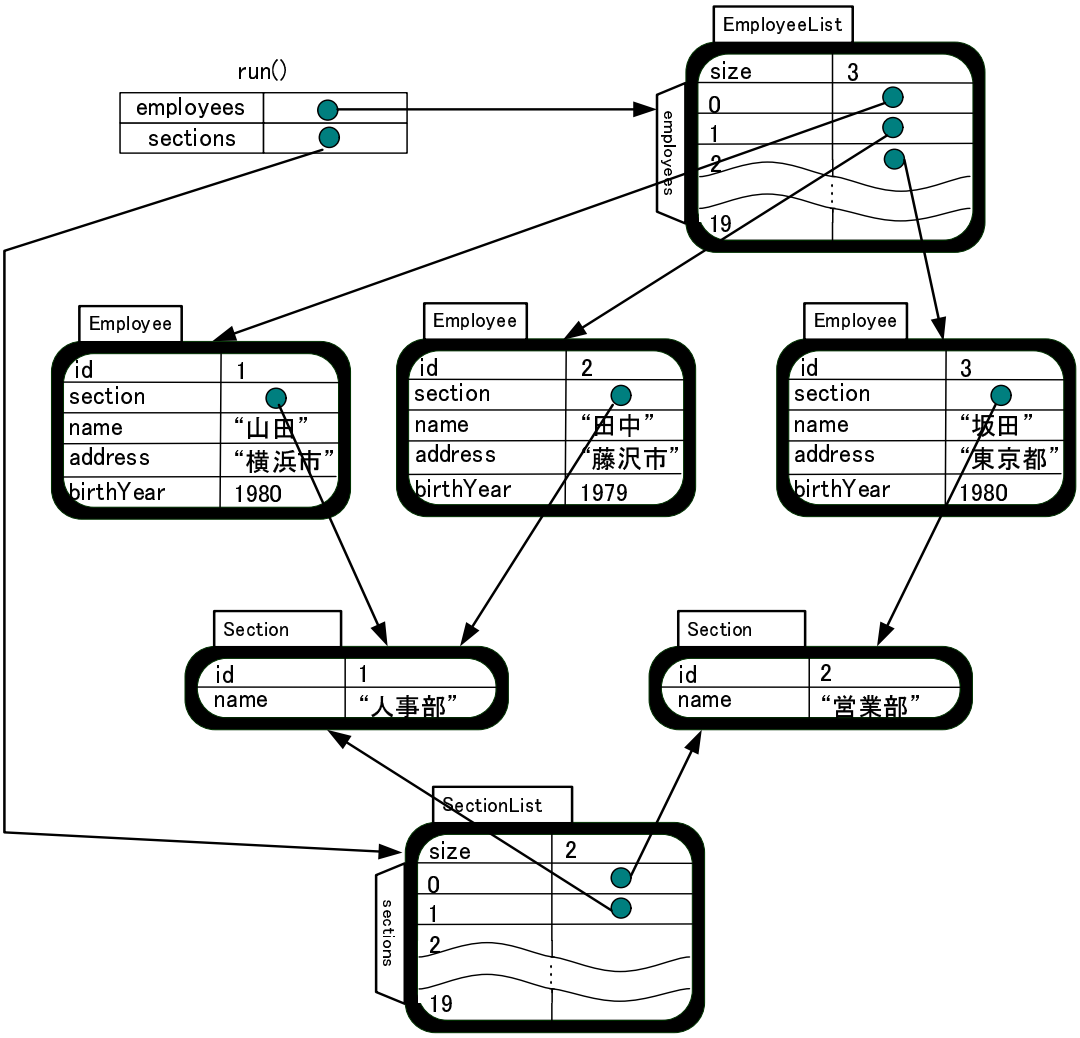


図 10.2: 社員名簿参照モデル

10.1.2 同じとはどういうことか

10.1.2.1 値と参照

これまで”同じ”という言葉をいろいろなところで使ってきました。しかし、プログラムでいう”同じ”ということには 2 種類の意味があります。

考えてみよう

二人の社員の入社年が同じなのと、部署が同じなのでは、”同じ”という意味がどのように違うか考えてみましょう。

10.1.2.2　基本データ型と参照型

int, boolean, double などは Java ではオブジェクトではありません。このような型を扱う場合、変数には直接それぞれの値が対応しています。ですから例え入社年が同じ社員うち片方の入社年を変更したとしても、もう一方の入社年が同じように変更されるわけではありません。このような型のことを基本データ型といいます。[3]

一方、String、Employee などはオブジェクトです。オブジェクトを扱う場合、変数に対応しているのは実際の値ではなく、オブジェクトへの参照です。[4]ですから、同じ部署への参照を持つ社員がいた場合に、その部署を変更すると両方の社員の部署が変更されたようにみえることになります。このような型のことを参照型といいます。

10.1.2.3

今まで int を比較するときには「==」演算子、String 型のオブジェクトを比較するときには.equals() メソッドを使ってきました。この二つは同じような使い方をしますが、なぜ使い分ける必要があるのでしょうか。

「==」演算子は、その変数に対応している実際の値を比較する演算子です。ですから基本データ型の比較にはこの演算子を使います。しかし、String のような参照型の変数を比較する場合、たとえ内部で同じ値を持つオブジェクトでも、(例えば、2 つの文字列オブジェクトの内容”田中”が同じだった場合など) 違うオブジェクトを参照していれば「==」演算子の評価の値は false となってしまいます。

このように、参照型のオブジェクト「==」演算子を使って比較をすると、意図した評価値にならない場合があります。

そのような場合に対応するため、String クラスに実装されているメソッドが.equals() メソッドです。このメソッドは変数の値ではなく、参照先のオブジェクトの値を比較し、真偽値を返します。[5]

考えてみよう

次のプログラムリスト 85 を実行すると、どのような結果になるか考えてみましょう。

リスト 85: 「==」演算子と .equals() メソッド

```
1: /**
2:  * 「==」演算子と .equals() メソッドの違いを確かめるサンプルプログラム
```

[3] 基本データ型の一覧は「Java 言語プログラミングレッスン (上)」の付録 H を参照してください。
[4] 正確にはオブジェクトが存在しているメモリのアドレスが値として入っています。
[5] そうなるように equals() メソッドがオーバーライドされています。

```
 3:  *
 4:  * @author kei
 5:  * @version $Id: EqualSample.java,v 1.5 2003/05/08 16:54:53 bam Exp $
 6:  */
 7: public class EqualSample {
 8:
 9:   public static void main(String[] args) {
10:     EqualSample equalSample = new EqualSample();
11:     equalSample.run();
12:   }
13:
14:   private void run() {
15:     String str1 = new String("文字");
16:     String str2 = new String("文字");
17:
18:     if (str1 == str2) {
19:       System.out.println("str1 == str2 である");
20:     } else {
21:       System.out.println("str1 == str2 ではない");
22:     }
23:
24:     if (str1.equals(str2)) {
25:       System.out.println("str1.equals(str2) である");
26:     } else {
27:       System.out.println("str1.equals(str2) ではない");
28:     }
29:   }
30: }
```

10.2　オブジェクトの構造とナビゲーション

10.2.1　参照の方向

10.2.1.1　逆方向からの検索

ここで、現在の社員名簿アプリケーションの構造を使って、部署に所属する社員の一覧を表示するプログラムを書くことを考えてみましょう。

リスト 86: 部署の一覧表示をするサンプルプログラム

```
 1: /**
 2:  * 部署に所属する社員を表示するサンプルプログラム
 3:  *
 4:  * @author bam
 5:  * @version $Id: SectionViewSample.java,v 1.5 2003/05/07 08:08:32 macchan Exp $
 6:  */
 7: public class SectionViewSample {
 8:
 9:   public static void main(String[] args) {
10:     SectionViewSample sectionViewSample = new SectionViewSample();
11:     sectionViewSample.run();
12:   }
13:
14:   private void run() {
15:
16:     SectionList sections = new SectionList();
17:     EmployeeList employees = new EmployeeList();
18:
19:     //部署を生成
20:     sections.add(new Section(1, "人事部"));
21:     sections.add(new Section(2, "営業部"));
22:
23:     //社員を生成
24:     employees.add(new Employee(1, sections.search(1), "山田", "横浜市", 1980));
25:     employees.add(new Employee(2, sections.search(1), "田中", "藤沢市", 1979));
26:     employees.add(new Employee(3, sections.search(2), "坂田", "東京都", 1980));
27:
28:     //社員リストを表示
29:     showSection(employees, sections.search(1)); //人事部
30:     showSection(employees, sections.search(2)); //営業部
31:   }
32:
33:   /**
34:    * 部署一つ分の所属社員を表示する
35:    */
36:   private void showSection(EmployeeList employees, Section section) {
37:
38:     //部署のタイトルを出力
39:     System.out.println("---" + section.getName() + "---");
40:
```

```
41:     //部署に所属する社員を表示
42:     for (int i = 0; i < employees.size(); i++) {
43:       Employee employee = employees.get(i);
44:       if (employee.getSection() == section) { //対象となる部署に所属する社員である場合
45:         showEmployee(employee);
46:       }
47:     }
48:   }
49:
50:   /**
51:    * 社員一人分の情報を表示する
52:    */
53:   private void showEmployee(Employee employee) {
54:     System.out.println(
55:       "\t"
56:         + employee.getId()
57:         + "\t"
58:         + employee.getSection().getName()
59:         + "\t"
60:         + employee.getName()
61:         + "\t"
62:         + employee.getAddress()
63:         + "\t"
64:         + employee.getBirthYear()
65:         + "年生まれ\t"
66:         + employee.getAge()
67:         + "才");
68:   }
69:
70: }
```

10.2.1.2 相互参照

リスト 86 のようなプログラムで部署に所属する社員を調べようとすると、全ての社員から部署に所属する社員を検索する必要があります。そこでプログラムを部署に所属する社員を調べられるように変更してみましょう。部署クラスを以下のように変更し、部署はそれぞれその部署に所属する社員のリストを持つようにします。

リスト 87: 部署クラス (相互参照を使う)

```
1: import java.io.Serializable;
2:
3: /**
4:  * 部署クラス
5:  *
6:  * 所属する社員の参照を持つ
7:  *
8:  * @author bam
9:  * @version $Id: Section.java,v 1.4 2003/05/07 04:07:22 macchan Exp $
```

```
10:  */
11: public class Section implements Serializable {
12:
13:   private int id; //部署 ID
14:   private String name; //部署名
15:
16:   //  社員の管理
17:   private EmployeeList employees = new EmployeeList(); //部署に所属する社員
18:
19:   /**
20:    * コンストラクタ
21:    */
22:   public Section(int id, String name) {
23:     this.id = id;
24:     this.name = name;
25:   }
26:
27:   /**
28:    * 部署の ID を取得する
29:    */
30:   public int getId() {
31:     return id;
32:   }
33:
34:   /**
35:    * 部署の名前を取得する
36:    */
37:   public String getName() {
38:     return name;
39:   }
40:
41:   /**
42:    * 部署の名前を設定する
43:    */
44:   public void setName(String name) {
45:     this.name = name;
46:   }
47:
48:   /*****************************************************
49:   /* この部署に所属する社員の管理
50:   /*****************************************************/
51:
52:   /**
53:    * 部署に所属する社員を追加する
54:    */
55:   public void addEmployee(Employee employee) {
56:     employees.add(employee);
57:   }
58:
59:   /**
60:    * 部署に所属する社員を削除する
61:    */
62:   public void removeEmployee(Employee employee) {
63:     employees.remove(employee);
```

```
64:   }
65:
66:   /**
67:    * 部署に所属する社員を index で指定して取得する
68:    */
69:   public Employee getEmployee(int index) {
70:     return employees.get(index);
71:   }
72:
73:   /**
74:    * 部署に所属する社員のリストを取得する
75:    */
76:   public EmployeeList getEmployees() {
77:     return employees;
78:   }
79:
80:   /**
81:    * この部署に所属する社員がいるかどうか調べる
82:    */
83:   public boolean hasEmployee() {
84:     return !(employees.size() == 0);
85:   }
86:
87: }
```

リスト 88: 部署の一覧表示をするサンプルプログラム (相互参照)

```
 1: /**
 2:  * 部署に所属する社員を表示するサンプルプログラム
 3:  * (相互参照バージョン)
 4:  *
 5:  * @author bam
 6:  * @version $Id: SectionViewSample.java,v 1.4 2003/05/07 07:19:46 bam Exp $
 7:  */
 8: public class SectionViewSample {
 9:
10:   public static void main(String[] args) {
11:     SectionViewSample sectionViewSample = new SectionViewSample();
12:     sectionViewSample.run();
13:   }
14:
15:   private void run() {
16:
17:     SectionList sections = new SectionList();
18:     EmployeeList employees = new EmployeeList();
19:
20:     //部署を生成
21:     sections.add(new Section(1, "人事部"));
22:     sections.add(new Section(2, "営業部"));
23:
24:     //社員を生成
```

```
25:     addEmployee(1, sections.search(1), "山田", "横浜市", 1980, employees);
26:     addEmployee(2, sections.search(1), "田中", "藤沢市", 1979, employees);
27:     addEmployee(3, sections.search(2), "坂田", "東京都", 1980, employees);
28:
29:     //社員リストを表示
30:     showSection(sections.search(1)); //人事部
31:     showSection(sections.search(2)); //営業部
32:   }
33:
34:   /**
35:    * 社員を一人追加する
36:    */
37:   private void addEmployee(
38:     int id,
39:     Section section,
40:     String name,
41:     String address,
42:     int birthYear,
43:     EmployeeList employees) {
44:
45:     Employee employee = new Employee(id, null, name, address, birthYear);
46:     employees.add(employee);
47:
48:     employee.setSection(section);
49:     section.addEmployee(employee);
50:   }
51:
52:   /**
53:    * 部署一つ分の所属社員を表示する
54:    */
55:   private void showSection(Section section) {
56:
57:     //部署のタイトルを出力
58:     System.out.println("---" + section.getName() + "---");
59:
60:     //部署に所属する社員を表示
61:     EmployeeList employees = section.getEmployees();
62:     for (int i = 0; i < employees.size(); i++) {
63:       Employee employee = employees.get(i);
64:       showEmployee(employee);
65:     }
66:   }
67:
68:   /**
69:    * 社員一人分の情報を表示する
70:    */
71:   private void showEmployee(Employee employee) {
72:     System.out.println(
73:       "\t"
74:         + employee.getId()
75:         + "\t"
76:         + employee.getSection().getName()
77:         + "\t"
78:         + employee.getName()
```

```
79:         + "\t"
80:         + employee.getAddress()
81:         + "\t"
82:         + employee.getBirthYear()
83:         + "年生まれ\t"
84:         + employee.getAge()
85:         + "才");
86:   }
87:
88: }
```

リスト 88 が生成するオブジェクトの参照モデルは図 10.3 のようになります。

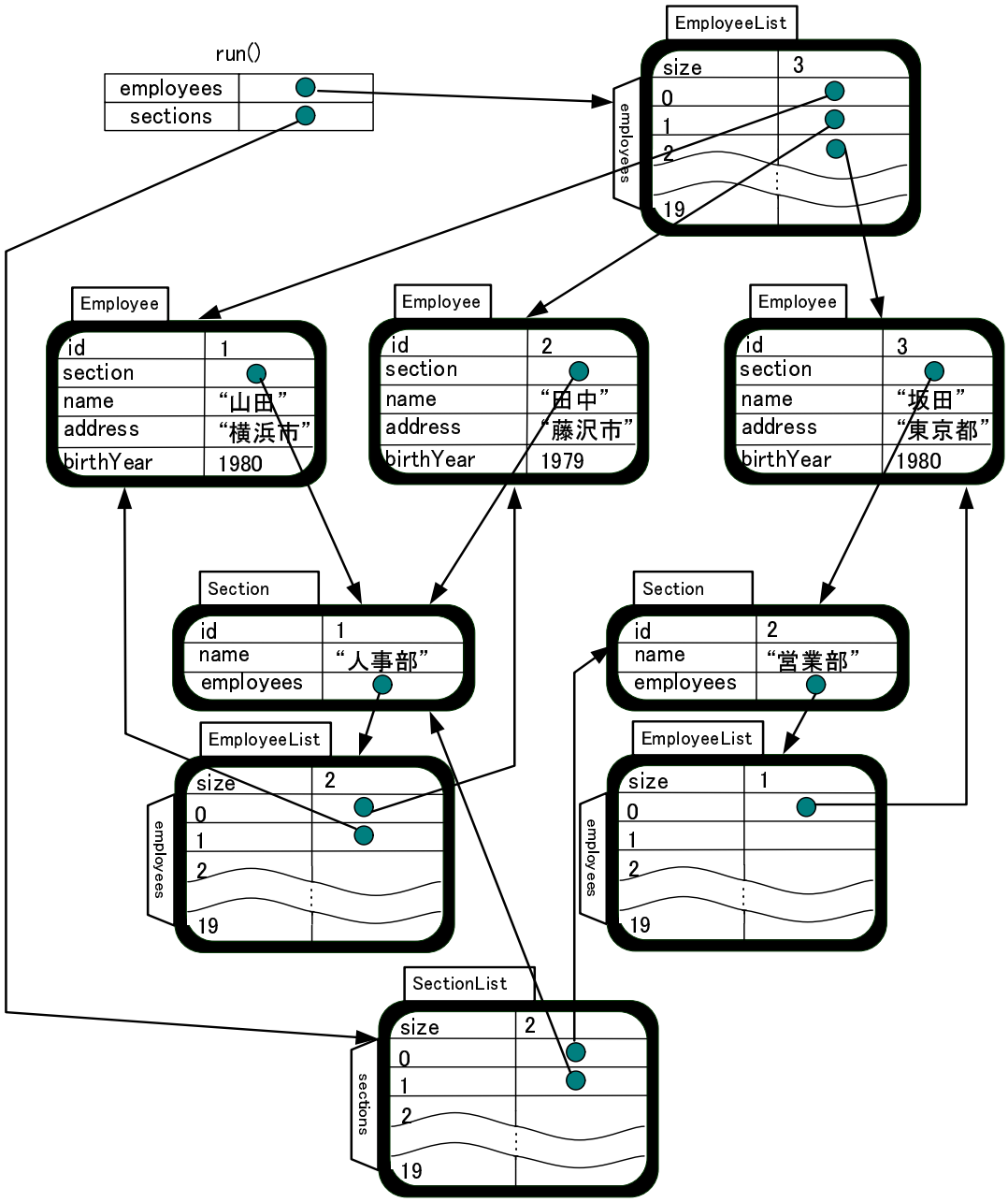


図 10.3: 相互参照を使った部署と社員

このような変更を加えると、部署はその部署に所属する社員への参照を持ち、社員はその社員が所属する部署への参照を持つという構造 [6]ができあがります。

このようなお互いに参照を持ち合うような関係を相互参照と言います。

[6] 部署は直接社員を持っているわけではないですが、実際には配列で持っているのと同じです。なぜなら、EmployeeList は配列を管理するだけのクラスであるからです。

10.2.1.3　参照の方向とナビゲーション

部署と社員には何らかの関係が存在します。本章では、その関係をプログラムとして実装する方法として、参照モデルを説明してきました。

参照の方向はどちらか一方向の参照があれば、リスト 86 の例のように実装することができる場合が多くあります。しかし、リスト 88 のように相互参照を利用し、必要な情報を効率よくわかりやすく取得できる場合も多くあります。そのときどきに応じてどちらにするか検討する必要があります。

今までの社員システムは、社員が部署を持っているだけの構造でした。このように片方からしか参照を持っていないことを単方向の参照といいます。これに対して相互参照のことを双方向の参照といいます。

考えてみよう

リスト 86 とリスト 88 を比較して、単方向参照と双方向参照の利点と欠点を議論してみましょう。

10.2.2　ネットワーク構造の構築

10.2.2.1　ネットワーク構造の構築

参照モデルを導入すると、オブジェクトは入れ子構造ではなく ネットワーク構造になっていることがわかります。オブジェクトは包含関係になっているのではなく、お互い関係を持ち合いながら参照でつながっているのです。

またこのようなネットワーク構造を構築するためには、オブジェクトを生成したとき、もしくは後から他のオブジェクトへの参照を設定する必要があります。

リスト 89: 社員追加メソッド (相互参照)

```
37:   private void addEmployee(
38:     int id,
39:     Section section,
40:     String name,
41:     String address,
42:     int birthYear,
43:     EmployeeList employees) {
44:
45:     Employee employee = new Employee(id, null, name, address, birthYear);
46:     employees.add(employee);
47:
48:     employee.setSection(section);
49:     section.addEmployee(employee);
50:   }
```

社員名簿の例では社員を追加する際に、社員リストと社員が所属する部署の両方に追加したい部署への参照を渡す必要があります。

10.2.2.2　ネットワーク構造の永続化

この節では、社員名簿システムに部署クラスを導入して、オブジェクトのネットワーク構造を構築してきました。しかし、オブジェクトのネットワーク構造は今までのように単純にテキスト形式でファイルに書き出すことができません。このような問題を解決するためには、オブジェクトの参照情報を含めたネットワーク構造そのものを保存する必要があります。

Topics オブジェクトの保存

人にやさしいプログラミングの哲学では、オブジェクトを参照ごと保存するためのライブラリが用意されています。オブジェクトを保存するためには以下のようなプログラムを書きます。

リスト 90: オブジェクトの保存

```
119:   private void save() {
120:
121:     //ファイル名入力
122:     System.out.println("セーブするファイル名を入力してください");
123:     String fileName = Input.getString();
124:
125:     //記録係の生成
126:     Recorder recorder = new Recorder();
127:     recorder.employees = employees;
128:     recorder.sections = sections;
129:
130:     //前処理（ファイルオープン）
131:     WriteObjectStream writer = FileIO.openObjectStreamForWrite(fileName);
132:
133:     //書き出し
134:     writer.write(recorder);
135:
136:     //後処理（ファイルクローズ）
137:     writer.close();
138:   }
```

ファイルからオブジェクトを取り出すためには、以下のようなプログラムを書きます。このとき、必ず読み込みたいクラスのオブジェクトにキャスト [7]する必要があります。

リスト 91: オブジェクトの読み込み

```
143:   private void load() {
144:
145:     //ファイル名入力
146:     System.out.println("ロードするファイル名を入力してください");
147:     String fileName = Input.getString();
148:
149:     //前処理（ファイルオープン）
150:     ReadObjectStream reader = FileIO.openObjectStreamForRead(fileName);
151:
152:     //読み込み
153:     Recorder recorder = (Recorder) reader.read();
154:
155:     //後処理（ファイルクローズ）
156:     reader.close();
157:
158:     //読み込んだデータを設定
159:     sections = recorder.sections;
160:     employees = recorder.employees;
161:   }
```

[7] オブジェクトのキャストについては、12.1.2.1 節を参照のこと。ここでは int 型などをキャストするときと同じように、クラスの型でキャストすると考えてください。

オブジェクトを保存したときに保存されるのは、保存するオブジェクトから参照をたどれるものだけです。必ず保存したい全てのオブジェクトへの参照をたどることができるオブジェクトを指定する必要があります。そのため、Recorder という記録専用のオブジェクトを定義しそれを保存するようにします。

リスト 92: 記録係

```
168: class Recorder implements Serializable {
169:   SectionList sections;
170:   EmployeeList employees;
171: }
```

10.2.3　オブジェクトの導出

10.2.3.1　2 箇所に追加するプログラム

ここでもう一度相互参照を使った社員追加メソッドを見てみましょう。

リスト 93: 社員追加メソッド (相互参照)

```
37:   private void addEmployee(
38:     int id,
39:     Section section,
40:     String name,
41:     String address,
42:     int birthYear,
43:     EmployeeList employees) {
44:
45:     Employee employee = new Employee(id, null, name, address, birthYear);
46:     employees.add(employee);
47:
48:     employee.setSection(section);
49:     section.addEmployee(employee);
50:   }
```

現状では、社員の追加を行うときに、アプリケーションのクラスが持つ社員のリストと、社員を新たに追加する部署が持つ社員リストの両方に社員を追加する必要があります。

削除のときにも同様に、アプリケーションのクラスが持つ社員のリストと、社員を新たに追加する部署が持つ社員リストの両方から社員を削除する必要があります。

このような「社員を追加するときには部署とアプリケーションのそれぞれが持つ社員リストに社員を追加しなければならない」というような約束事のあるプログラムは、あまりわかりやすいプログラムとは言えません。この約束事を忘れてしまった、もしくは知らない人がプログラムを変更するとバグが発生する可能性があります。

考えてみよう

社員の新規追加が起きたとき社員を部署が持つ社員リストだけに追加するようにプログラムを変更しようと思います。アプリケーションのクラスが名簿に登録された全社員のリストを直接持つという現状の解決策以外で、アプリケーションのクラスから全ての社員を取得するにはどどうすればいいでしょうか。

10.2.3.2　社員リストの導出

上のような問題点を解決するために全社員のリストを、全部署のリストから導出するように変更します。導出を導入すると社員の追加は部署にのみに対して行えばいいようになります。

リスト 94: 部署の一覧表示をするサンプルプログラム (導出)

```
 1: /**
 2:  * 部署クラスを用いたサンプルプログラム
 3:  *（社員リスト導出バージョン）
 4:  *
 5:  * @author bam
 6:  * @version $Id: SectionViewSample.java,v 1.2 2003/05/07 07:19:46 bam Exp $
 7:  */
 8: public class SectionViewSample {
 9:
10:   public static void main(String[] args) {
11:     SectionViewSample sectionViewSample = new SectionViewSample();
12:     sectionViewSample.run();
13:   }
14:
15:   private void run() {
16:
17:     SectionList sections = new SectionList();
18:
19:     //部署を生成
20:     sections.add(new Section(1, "人事部"));
21:     sections.add(new Section(2, "営業部"));
22:
23:     //社員を生成
24:     addEmployee(1, sections.search(1), "山田", "横浜市", 1980);
25:     addEmployee(2, sections.search(1), "田中", "藤沢市", 1979);
26:     addEmployee(3, sections.search(2), "坂田", "東京都", 1980);
27:
28:     //社員リストを表示
29:     showSection(sections.search(1)); //人事部
30:
31:     //全ての社員リストを表示
32:     showAllEmployees(sections);
33:   }
34:
35:   /**
36:    * 社員を一人追加する
37:    */
38:   private void addEmployee(
39:     int id,
40:     Section section,
41:     String name,
42:     String address,
```

```
43:     int birthYear) {
44:
45:     Employee employee = new Employee(id, section, name, address, birthYear);
46:     section.addEmployee(employee);
47:   }
48:
49:   /**
50:    * 全ての社員を導出する
51:    */
52:   private EmployeeList navigateAllEmployees(SectionList sections) {
53:
54:     EmployeeList navigatedEmployees = new EmployeeList(); //導出される社員リスト
55:
56:     //全ての部署の社員をリストに加える
57:     for (int i = 0; i < sections.size(); i++) {
58:       navigatedEmployees.addAll(sections.get(i).getEmployees());
59:     }
60:
61:     return navigatedEmployees;
62:   }
63:
64:   /**
65:    * 全ての社員を表示する
66:    */
67:   private void showAllEmployees(SectionList sections) {
68:
69:     //タイトルを出力
70:     System.out.println("---すべての社員---");
71:
72:     //全ての社員リストを導出して表示
73:     showEmployeeList(navigateAllEmployees(sections));
74:   }
75:
76:   /**
77:    * 部署一つ分の所属社員を表示する
78:    */
79:   private void showSection(Section section) {
80:
81:     //部署のタイトルを出力
82:     System.out.println("---" + section.getName() + "---");
83:
84:     //部署に所属する社員を表示
85:     showEmployeeList(section.getEmployees());
86:   }
87:
88:   /**
89:    * 社員リストを表示する
90:    */
91:   private void showEmployeeList(EmployeeList employees) {
92:     for (int i = 0; i < employees.size(); i++) {
93:       Employee employee = employees.get(i);
94:       showEmployee(employee);
95:     }
96:   }
```

```
 97:
 98:   /**
 99:    * 社員一人分の情報を表示する
100:    */
101:   private void showEmployee(Employee employee) {
102:     System.out.println(
103:       "\t"
104:         + employee.getId()
105:         + "\t"
106:         + employee.getSection().getName()
107:         + "\t"
108:         + employee.getName()
109:         + "\t"
110:         + employee.getAddress()
111:         + "\t"
112:         + employee.getBirthYear()
113:         + "年生まれ\t"
114:         + employee.getAge()
115:         + "才");
116:   }
117:
118: }
```

このプログラムが生成するオブジェクトの参照モデルは図 10.4 のようになります。

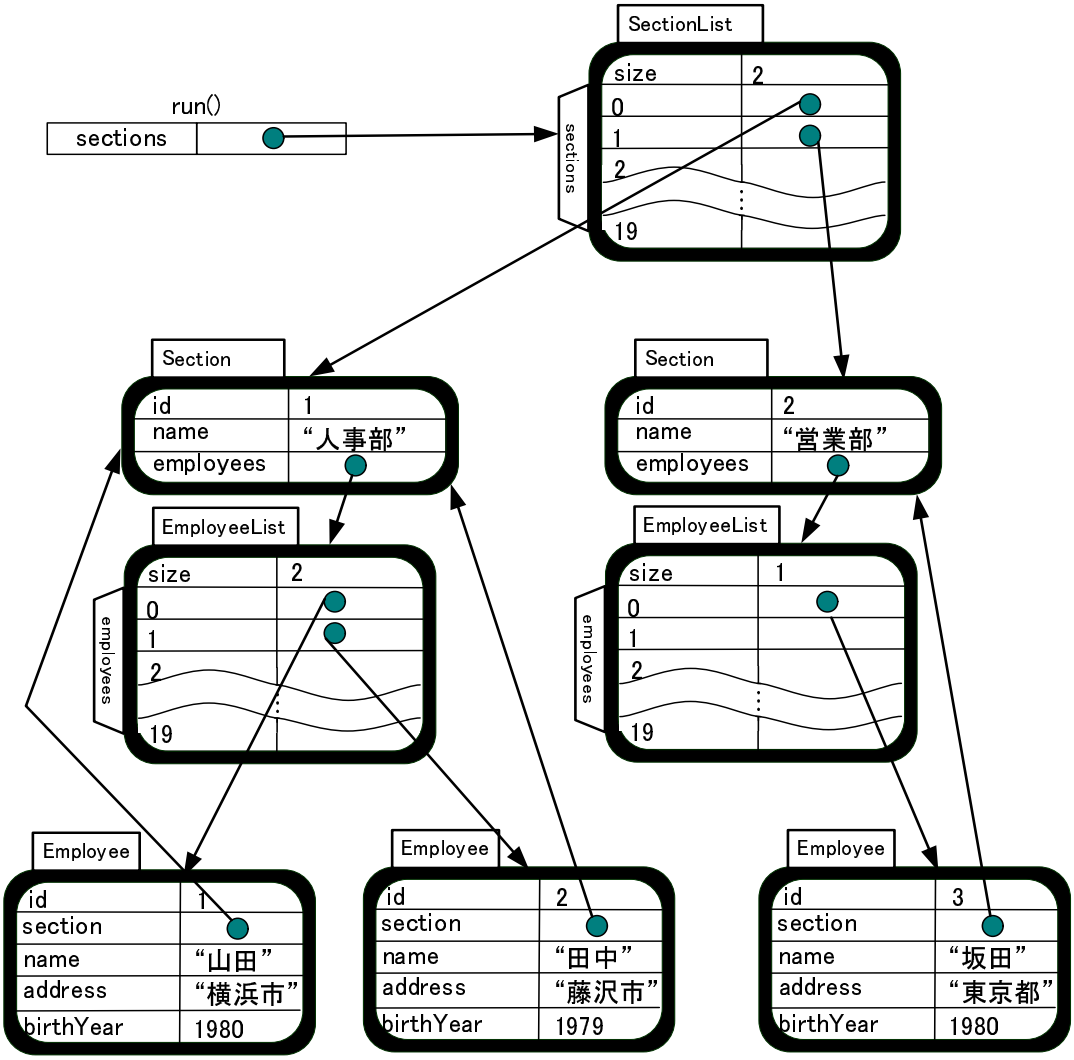


図 10.4: 導出を使った社員名簿

10.3　練習問題

練習問題 1

以下のバス運行管理シミュレーション (リスト 95-リスト 98) を読んで、コメントが抜けている部分を補い、さらに、(1) から (6) の時点での参照モデルを記述してください。

リスト 95: バス運行管理シミュレーション (BusSimulatorSample.java)

```
 1: /**
 2:  * バスの運行をシミュレートするサンプルプログラム
 3:  * （参照モデル版）
 4:  *
 5:  * @author bam
 6:  * @version $Id: BusSimulatorSample.java,v 1.2 2003/05/07 10:03:42 macchan Exp $
 7:  */
 8: public class BusSimulatorSample {
 9:
10:   public static void main(String[] args) {
11:     BusSimulatorSample busTest = new BusSimulatorSample();
12:     busTest.main();
13:   }
14:
15:   private void main() {
16:
17:     //
18:     BusStop daiba = new BusStop();
19:     BusStop teleport = new BusStop();
20:     BusStop hamamatsutyo = new BusStop();
21:     daiba.setName("台場");
22:     teleport.setName("テレポート");
23:     hamamatsutyo.setName("浜松町");
24:
25:     //
26:     Bus bus = new Bus();
27:     bus.setDestination(hamamatsutyo);
28:
29:     //
30:     Passenger yamamoto = new Passenger();
31:     Passenger tanaka = new Passenger();
32:     yamamoto.setName("山本");
33:     tanaka.setName("田中");
34:
35:     showBusStopInfo(daiba);
36:
37:     //
38:     daiba.addWaitingPassenger(yamamoto);
39:     daiba.addWaitingPassenger(tanaka);
40:     showBusStopInfo(daiba);
41:     //----- (1) -----
42:
```

```
43:     //
44:     daiba.setStoppingBus(bus);
45:     showBusStopInfo(daiba);
46:     //----- (2) -----
47:
48:     //
49:     daiba.removeWaitingPassenger(yamamoto);
50:     daiba.removeWaitingPassenger(tanaka);
51:     showBusStopInfo(daiba);
52:
53:     //
54:     bus.addRidingPassenger(yamamoto);
55:     bus.addRidingPassenger(tanaka);
56:     showBusInfo(bus);
57:     //----- (3) -----
58:
59:     //
60:     daiba.setStoppingBus(null);
61:     showBusStopInfo(daiba);
62:     //----- (4) -----
63:
64:     //
65:     teleport.setStoppingBus(bus);
66:     showBusStopInfo(teleport);
67:
68:     //
69:     bus.removeRidingPassenger(yamamoto);
70:     showBusInfo(bus);
71:
72:     //
73:     teleport.setStoppingBus(null);
74:     showBusStopInfo(teleport);
75:
76:     //
77:     hamamatsutyo.setStoppingBus(bus);
78:     showBusStopInfo(hamamatsutyo);
79:     //----- (5) -----
80:
81:     //
82:     bus.removeRidingPassenger(tanaka);
83:     showBusInfo(bus);
84:
85:     //
86:     bus.setDestination(daiba);
87:     showBusStopInfo(hamamatsutyo);
88:     //----- (6) -----
89:   }
90:
91:   /**
92:    * バスの状況を表示する
93:    */
94:   private void showBusInfo(Bus bus) {
95:
96:     Passenger[] passengers = bus.getRidingPassengers();
```

```
 97:     int count = 0;
 98:
 99:     //タイトル
100:     System.out.println("現 在 の" + bus.getDestination().getName() + "行 き バ ス の 状
況");
101:
102:     //待っている人のリストを表示する
103:     for (int i = 0; i < passengers.length; i++) {
104:       if (passengers[i] != null) {
105:         System.out.println("乗客:" + passengers[i].getName());
106:         count++;
107:       }
108:     }
109:
110:     //待っている人の数を表示する
111:     System.out.println("現在乗っている客は:" + count + "人です");
112:
113:     //ユーザの指示を待つ
114:     waitSimulator();
115:   }
116:
117:   /**
118:    * バス停の状況を表示する
119:    */
120:   private void showBusStopInfo(BusStop busStop) {
121:
122:     Passenger[] passengers = busStop.getPassengers();
123:     int count = 0;
124:
125:     //タイトル
126:     System.out.println("現在の" + busStop.getName() + "の状況");
127:
128:     //待っている人のリストを表示する
129:     for (int i = 0; i < passengers.length; i++) {
130:       if (passengers[i] != null) {
131:         System.out.println("バス待ちの客:" + passengers[i].getName());
132:         count++;
133:       }
134:     }
135:
136:     //待っている人の数を表示する
137:     System.out.println(
138:       "現在" + busStop.getName() + "で待っている客は:" + count + "人です");
139:
140:     //停車中のバスの情報を表示する
141:     if (busStop.getStoppingBus() == null) {
142:       System.out.println("現在" + busStop.getName() + "にバスは停まっていません");
143:     } else if (busStop.getStoppingBus().getDestination() != null) {
144:       System.out.println(
145:         "現在"
146:           + busStop.getStoppingBus().getDestination().getName()
147:           + "行きのバスが停車中です");
148:     } else {
149:       System.out.println("現在行き先未定のバスが停車中です");
```

```
150:     }
151:
152:     //ユーザの指示を待つ
153:     waitSimulator();
154:   }
155:
156:   /**
157:    * シミュレータを一旦止めて、ユーザからの指示を待つ
158:    */
159:   private void waitSimulator() {
160:     System.out.println("バス運行シミュレータを進めるには Enter を押してください");
161:     Input.getString();
162:   }
163: }
```

リスト 96: バス運行管理シミュレーション (Bus.java)

```
 1: /**
 2:  * バスクラス
 3:  * 目的地に向かって移動する
 4:  * 乗客を乗せることができる
 5:  *
 6:  * @author N.Aoyama
 7:  * @version $Id: Bus.java,v 1.5 2003/05/07 10:03:42 macchan Exp $
 8:  */
 9: class Bus {
10:
11:   private BusStop destination;
12:   private Passenger[] ridingPassengers = new Passenger[100];
13:
14:   /**
15:    * バスの行き先を設定する
16:    */
17:   public void setDestination(BusStop destination) {
18:     this.destination = destination;
19:   }
20:
21:   /**
22:    * バスの乗客を追加する
23:    */
24:   public void addRidingPassenger(Passenger passenger) {
25:     for (int i = 0; i < this.ridingPassengers.length; i++) {
26:       if (ridingPassengers[i] == null) {
27:         ridingPassengers[i] = passenger;
28:         return;
29:       }
30:     }
31:   }
32:
33:   /**
34:    * バスの乗客を削除する
```

```
35:    */
36:   public void removeRidingPassenger(Passenger passenger) {
37:     for (int i = 0; i < this.ridingPassengers.length; i++) {
38:       if (ridingPassengers[i] == passenger) {
39:         ridingPassengers[i] = null;
40:         return;
41:       }
42:     }
43:   }
44:
45:   /**
46:    * バスの目的地を取得する
47:    */
48:   public BusStop getDestination() {
49:     return destination;
50:   }
51:
52:   /**
53:    * バスの乗客を取得する
54:    */
55:   public Passenger[] getRidingPassengers() {
56:     return ridingPassengers;
57:   }
58:
59: }
```

リスト 97: バス運行管理シミュレーション (BusStop.java)

```
 1: /**
 2:  * バス停クラス
 3:  * バスが停車して、待っている客が乗ることが出来る
 4:  *
 5:  * @author N.Aoyama
 6:  * @version $Id: BusStop.java,v 1.4 2003/05/07 10:03:42 macchan Exp $
 7:  */
 8: class BusStop {
 9:
10:   private String name;
11:   private Bus stoppingBus;
12:   private Passenger[] waitingPassengers = new Passenger[10];
13:
14:   /**
15:    * 名前を設定する
16:    */
17:   public void setName(String name) {
18:     this.name = name;
19:   }
20:
21:   /**
22:    * 名前を取得する
23:    */
```

```
24:   public String getName() {
25:     return name;
26:   }
27:
28:   /**
29:    * バス停にとまっているバスを取得する
30:    */
31:   public Bus getStoppingBus() {
32:     return stoppingBus;
33:   }
34:
35:   /**
36:    * バス停に止まっているバスを設定する
37:    */
38:   public void setStoppingBus(Bus stoppingBus) {
39:     this.stoppingBus = stoppingBus;
40:   }
41:
42:   /**
43:    * バス停に待っている乗客を追加する
44:    */
45:   public void addWaitingPassenger(Passenger passenger) {
46:     for (int i = 0; i < waitingPassengers.length; i++) {
47:       if (waitingPassengers[i] == null) {
48:         waitingPassengers[i] = passenger;
49:         return;
50:       }
51:     }
52:   }
53:
54:   /**
55:    * バスの乗客を削除する
56:    */
57:   public void removeWaitingPassenger(Passenger passenger) {
58:     for (int i = 0; i < waitingPassengers.length; i++) {
59:       if (waitingPassengers[i] == passenger) {
60:         waitingPassengers[i] = null;
61:         return;
62:       }
63:     }
64:   }
65:
66:   /**
67:    * バスの乗客を取得する
68:    */
69:   public Passenger[] getPassengers() {
70:     return waitingPassengers;
71:   }
72: }
```

リスト 98: バス運行管理シミュレーション (Passenger.java)

```
 1: /**
 2:  * 客クラス
 3:  * 停留所で待ち、バスで移動する
 4:  *
 5:  * @author N.Aoyama
 6:  * @version $Id: Passenger.java,v 1.3 2003/05/07 10:03:42 macchan Exp $
 7:  */
 8: class Passenger {
 9:
10:   private String name;
11:
12:   /**
13:    * 乗客の名前を取得する
14:    */
15:   public String getName() {
16:     return this.name;
17:   }
18:
19:   /**
20:    * 乗客の名前を設定する
21:    */
22:   public void setName(String name) {
23:     this.name = name;
24:   }
25: }
```

練習問題 2

履修登録プログラムの（SubjectManagementApplication.java, Professor.java, ProfessorList.java, StudentList.java, Subject.java, SubjectList.java）うち、Professor.java, Student.java, Subject.java を実装してプログラムを完成させてください。

練習問題 3

自動販売機プログラム（VendingMachineApplication.java, Item.java, ItemQueue.java, ItemType.java, ItemTypeList.java）のうち、Item.java, ItemType.java を実装してプログラムを完成させてください。

第 11 章

オブジェクトのネットワーク構造 (2)

この章で学習すること

クラス図をもとに参照モデル図を書くことができる
クラスの再帰構造を発見することができる
連結リストを使ったプログラムが書ける

11.1 クラス構造の図解

11.1.1 クラス図

ここまで、社員名簿アプリケーションに機能を追加し、それらを管理するためにいくつかのクラスを導入してきました。このようにクラスの構造が複雑になってくると、構造を理解するために参照モデルの図を書こうとしても、あまりにオブジェクトが多すぎて構造が分かりにくくなってきます。このような問題を解決するため、インスタンスの構造ではなく、クラスの構造をあらわす図を導入しましょう。

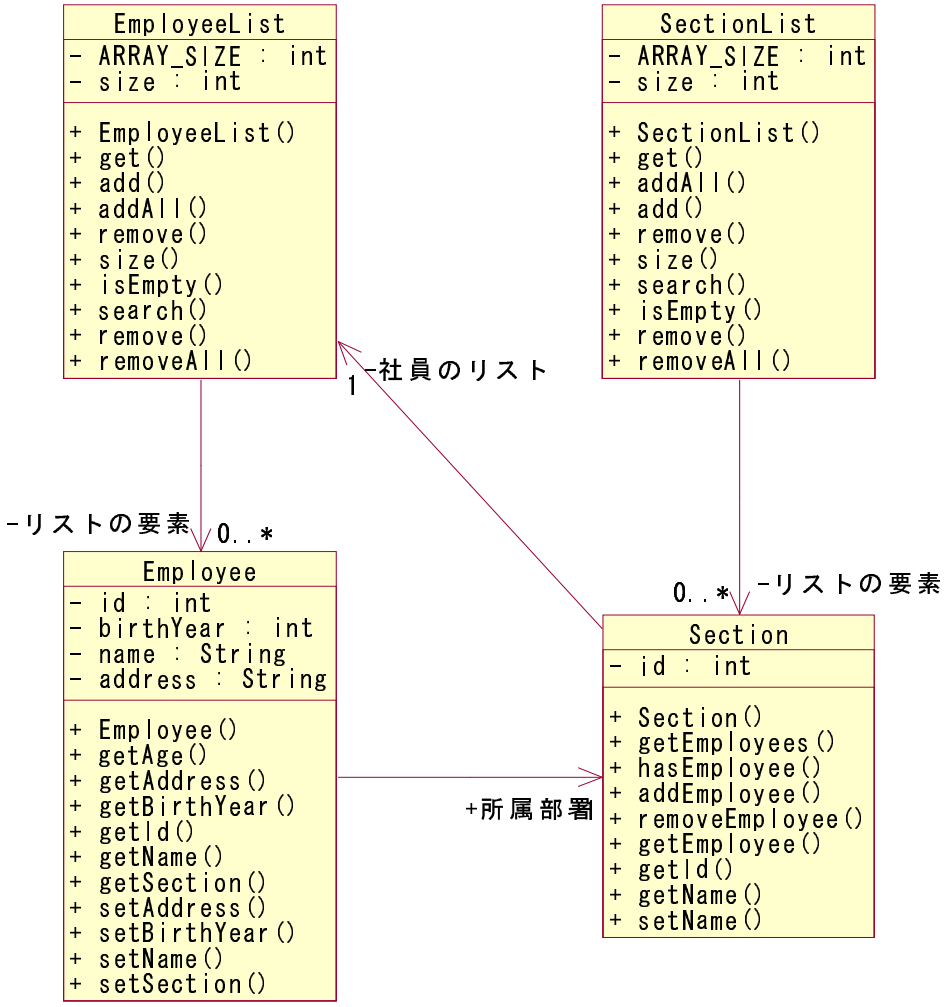


図 11.1: 社員名簿クラス図

このような図のことをクラス図といいます。

11.1.2　クラスの表現

クラス図でクラスを表現するには、四角形にクラス名、メソッド、フィールドとそれに関するアクセス修飾子を記述します。アクセス修飾子は public を「+」、private を「-」として表します。

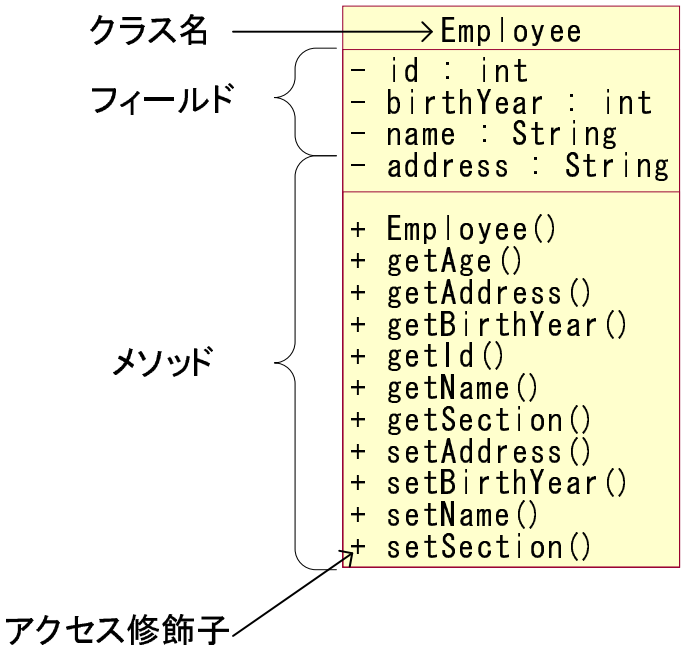


図 11.2: 社員クラス

11.1.3　関連の表現

クラスとクラスの間に何らかの参照を持つ関係があるときはクラスの間に線を引いて表現します。この線のことを関連と呼び、必要な情報を記述します。

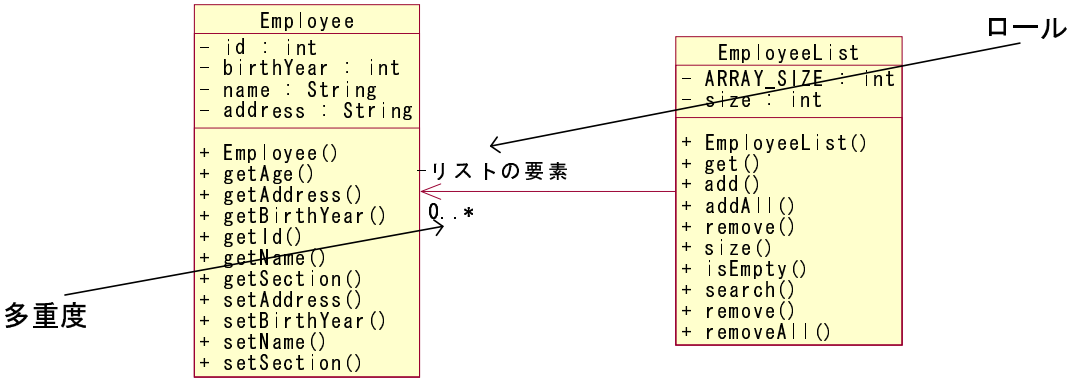


図 11.3: 社員クラスと社員リストクラス

関連の方向　クラス図では関連がどちらのクラスからどちらのクラスへのものなのかということを矢印であらわします。この例では EmployeeList からの関連なので、Employee は EmployeeList への参照を持ちません。

多重度　多重度とは、一方のクラスのインスタンスが、関連先のクラスのインスタンスをいくつ参照として持つのかということをあらわす数字です。線の終点に 1, 3, 0..* のように記述します。この例では EmployeeList のインスタンスは Employee のインスタンスへの参照を 0 個以上持つ可能性があるということを表します。

ロール　ロールは、一方のクラスのインスタンスから見て、関連先のクラスのインスタンスはどのような意味のオブジェクトなのかということを表します。

11.1.4　クラス図の曖昧さ

この節ではクラス構造を図で表す方法が新たに学びました。しかし、クラスの構造が記述できるからといって、今までの参照モデルの図が必要なくなるわけではありません。

例としてポーカーを行うアプリケーションを考えてみましょう。クラス設計は図 11.4 のようにしてみました。

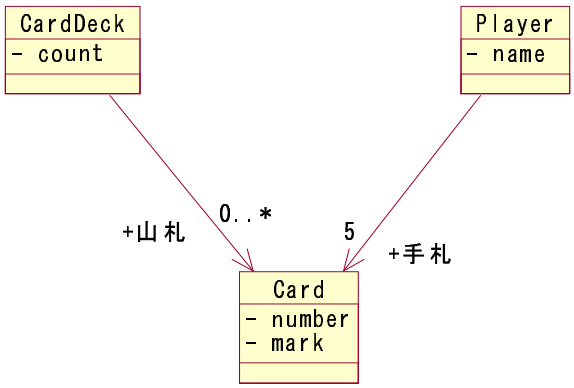


図 11.4: ポーカー

このクラス構造から想像できるアプリケーションのある状態を参照モデルを図 11.5 に示します。

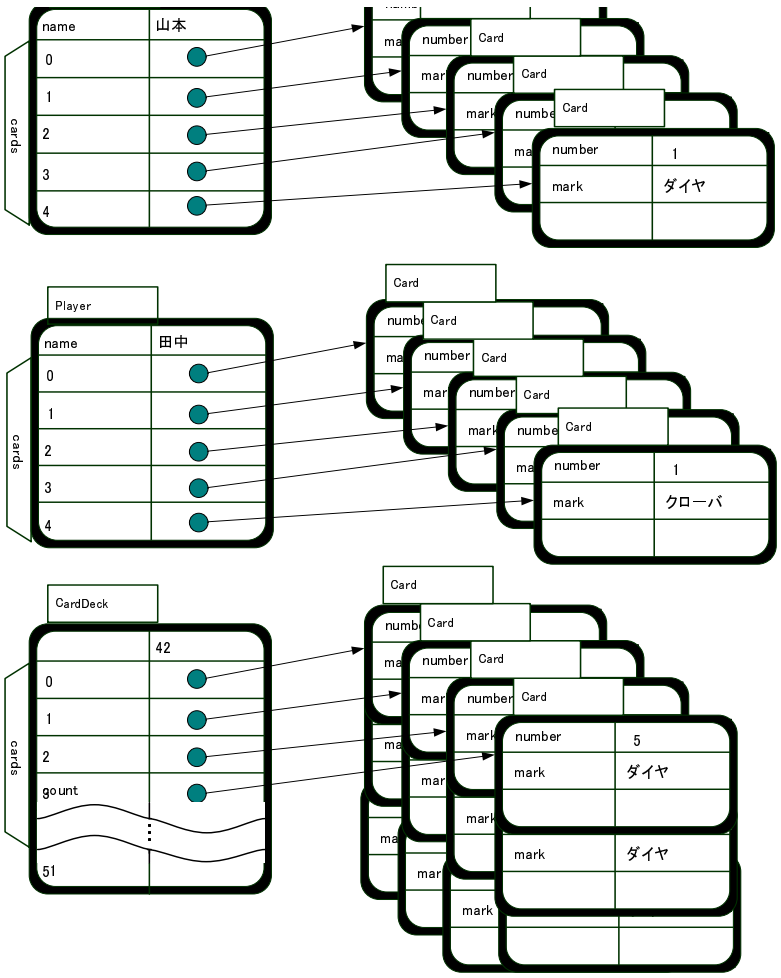


図 11.5: ポーカー参照モデル

このように、ここまで扱ったクラス図の記法では、アプリケーションの状態を完全に表現できません。[1]必ず参照モデルも書く必要があります。

考えてみよう

クラス図と参照モデル図の利点と欠点、類似点と相違点について議論してみましょう。

[1] このような問題を解決するために、クラス図には制約という記述方法があります。

11.2　クラスの再帰構造

11.2.1　階層構造の表現

11.2.1.1　部署の階層構造

前の章までで登場した部署クラスは、「人事部」「庶務課」といったようなそれぞれが独立した部署を表すクラスでした。しかし、現実の部署というものはそのような構造ではありません。「人事部」の中に、「販売人事課」「研修課」といったようなより小さな部署が含まれています。また、販売人事課はさらに「–係」といった、より小さな部署に分かれている場合もあります。図に表すと以下のようになります。このような構造を階層構造といいます。

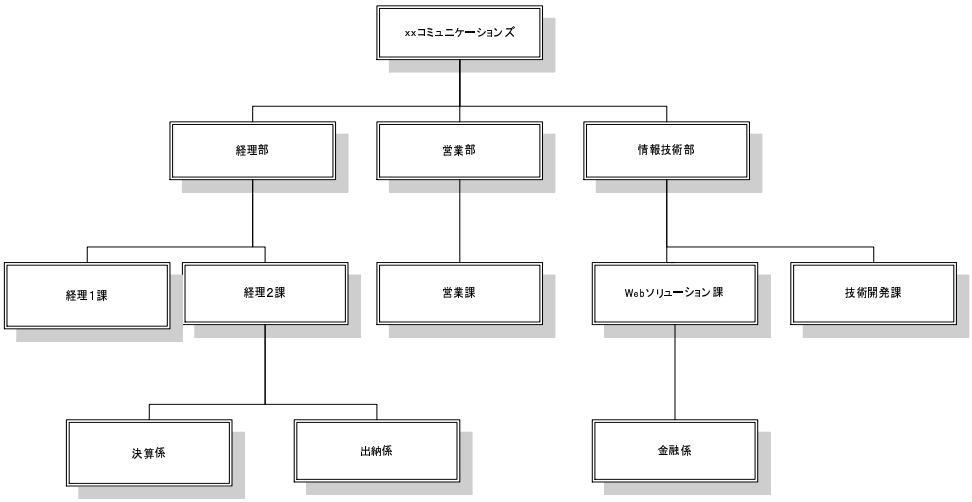


図 11.6: 部署の階層構造

考えてみよう

現実世界にある階層構造にはどのようなものがあるでしょう。考えてみましょう。

11.2.1.2　クラスの再帰構造

部署のように、ひとつのクラスのインスタンスが、さらにいくつかの同じクラスのインスタンスを持っているような階層構造をクラス構造で表すにはどのようにすればよいでしょう。このとき必要になる考え方が、構造化プログラミング編で登場した再帰の考え方です。

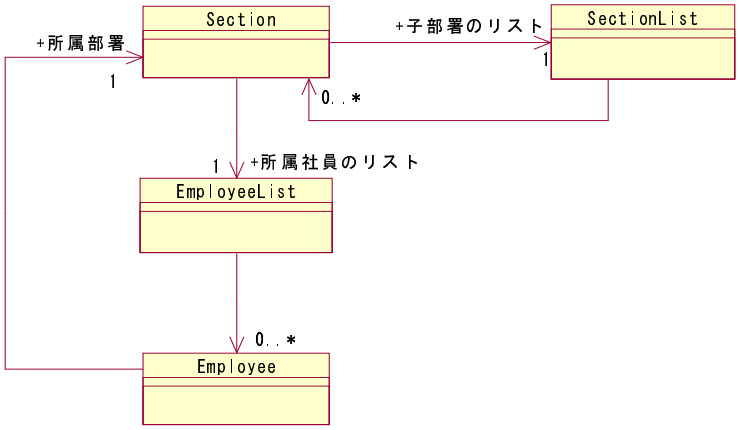


図 11.7: 部署の再帰構造クラス図

このクラス図はクラスの再帰構造をつかった部署クラスを導入したものです。部署は自分の子となる部署として他の部署への参照を幾つか持ちます。部署は部署リストを通して他の部署を持ちますが、これは配列を通して直接持っているのと同じように考えることができます。

再帰構造を使って実装した新しい部署クラスをリスト 99 に示します。再帰構造の探索に手続きの再帰呼び出しを使っています。

リスト 99: 部署クラス (再帰を使った)

```
 1: import java.io.Serializable;
 2:
 3: /**
 4:  * 部署クラス
 5:  *
 6:  * 所属する社員の参照を持つ
 7:  * 親部署、子部署の階層構造を持つ
 8:  *
 9:  * @author bam
10:  * @version $Id: Section.java,v 1.4 2003/05/07 04:07:22 macchan Exp $
11:  */
12: public class Section implements Serializable {
13:
```

```
14:   private int id; //部署 ID
15:   private String name; //部署名
16:
17:   //  社員の管理
18:   private EmployeeList employees = new EmployeeList(); //部署に所属する社員
19:
20:   //  部署の階層構造の管理
21:   private SectionList children = new SectionList(); //この部署の子となる部署のリスト
22:
23:   /**
24:    * コンストラクタ
25:    */
26:   public Section(int id, String name) {
27:     this.id = id;
28:     this.name = name;
29:   }
30:
31:   /**
32:    * 部署の ID を取得する
33:    */
34:   public int getId() {
35:     return id;
36:   }
37:
38:   /**
39:    * 部署の名前を取得する
40:    */
41:   public String getName() {
42:     return name;
43:   }
44:
45:   /**
46:    * 部署の名前を設定する
47:    */
48:   public void setName(String name) {
49:     this.name = name;
50:   }
51:
52:   /*****************************************************
53:   /* この部署に所属する社員の管理
54:   /*****************************************************/
55:
56:   /**
57:    * 部署に所属する社員を追加する
58:    */
59:   public void addEmployee(Employee employee) {
60:     employees.add(employee);
61:   }
62:
63:   /**
64:    * 部署に所属する社員を削除する
65:    */
66:   public void removeEmployee(Employee employee) {
67:     employees.remove(employee);
```

```
 68:   }
 69:
 70:   /**
 71:    * 部署に所属する社員を index で指定して取得する
 72:    */
 73:   public Employee getEmployee(int index) {
 74:     return employees.get(index);
 75:   }
 76:
 77:   /**
 78:    * 部署に所属する社員のリストを取得する
 79:    */
 80:   public EmployeeList getEmployees() {
 81:     return employees;
 82:   }
 83:
 84:   /**
 85:    * この部署に所属する社員がいるかどうか調べる
 86:    */
 87:   public boolean hasEmployee() {
 88:     return !(employees.size() == 0);
 89:   }
 90:
 91:   /*****************************************************
 92:   /* この部署にふくまれる部署の管理
 93:   /*****************************************************/
 94:
 95:   /**
 96:    * この部署の子となる部署を追加する
 97:    */
 98:   public void addChild(Section section) {
 99:     children.add(section);
100:   }
101:
102:   /**
103:    * この部署の子となる部署を index を指定して削除する
104:    */
105:   public void removeChild(int index) {
106:     children.remove(index);
107:   }
108:
109:   /**
110:    * この部署の子となる部署を index で指定して取得する
111:    */
112:   public Section getChild(int index) {
113:     return (Section) children.get(index);
114:   }
115:
116:   /**
117:    * この部署の子となる部署のリストを取得する
118:    * (再帰的に取得する)
119:    */
120:   public SectionList getChildren() {
121:     return children;
```

```
122:   }
123:
124:   /**
125:    * この部署に含まれる部署が存在するか調べる
126:    */
127:   public boolean hasChild() {
128:     return !(children.size() == 0);
129:   }
130:
131:   /*****************************************************
132:   /* 再帰導出処理
133:   /*****************************************************/
134:
135:   /**
136:    * 部署に所属する社員のリストを再帰的に導出する
137:    */
138:   public EmployeeList navigateAllEmployees() {
139:     EmployeeList navigatedEmployees = new EmployeeList(); //導出された社員のリスト
140:
141:     //この部署の社員をリストに加える
142:     navigatedEmployees.addAll(employees);
143:
144:     //全ての子の部署の社員をリストに加える
145:     for (int i = 0; i < children.size(); i++) {
146:       Section section = (Section) children.get(i);
147:       navigatedEmployees.addAll(section.navigateAllEmployees());
148:     }
149:
150:     return navigatedEmployees;
151:   }
152:
153:   /**
154:    * この部署の子となる部署のリストを再帰的に導出する
155:    */
156:   public SectionList navigateAllSections() {
157:     SectionList navigatedSections = new SectionList(); //導出された社員のリスト
158:
159:     //この部署の部署をリストに加える
160:     navigatedSections.addAll(children);
161:
162:     //全ての子の部署の部署をリストに加える
163:     for (int i = 0; i < children.size(); i++) {
164:       Section section = (Section) children.get(i);
165:       navigatedSections.addAll(section.navigateAllSections());
166:     }
167:
168:     return navigatedSections;
169:   }
170:
171: }
```

このようなクラス構造を使ったサンプルプログラムをリスト 100 に示します。

リスト 100: 部署の一覧表示をするサンプルプログラム (再帰)

```
 1: /**
 2:  * 部署クラスを用いたサンプルプログラム
 3:  *（再帰バージョン）
 4:  *
 5:  * @author bam
 6:  * @version $Id: SectionViewSample.java,v 1.2 2003/05/07 08:08:32 macchan Exp $
 7:  */
 8: public class SectionViewSample {
 9:
10:   public static void main(String[] args) {
11:     SectionViewSample sectionViewSample = new SectionViewSample();
12:     sectionViewSample.run();
13:   }
14:
15:   private void run() {
16:
17:     //Top セクションを生成
18:     Section top = new Section(0, "xx 株式会社");
19:
20:     //部署を生成する
21:     Section infotec = new Section(1, "情報技術部");
22:     Section business = new Section(2, "営業部");
23:     Section account = new Section(3, "経理部");
24:     Section account2 = new Section(4, "経理 2 課");
25:     Section account1 = new Section(5, "経理 1 課");
26:     Section web = new Section(6, "Web 開発課");
27:     Section casher = new Section(7, "出納係");
28:     Section settle = new Section(8, "決算係");
29:
30:     //部署の階層構造を構築
31:     top.addChild(infotec);
32:     top.addChild(business);
33:     top.addChild(account);
34:
35:     account.addChild(account2);
36:     account.addChild(account1);
37:
38:     infotec.addChild(web);
39:
40:     account1.addChild(casher);
41:     account1.addChild(settle);
42:
43:     //全ての部署/社員リストを表示
44:     showSection(top, 0);
45:   }
46:
47:   /**
48:    * 社員を一人追加する
49:    */
50:   private void addEmployee(
51:     int id,
52:     Section section,
53:     String name,
```

```
54:     String address,
55:     int birthYear) {
56:
57:     Employee employee = new Employee(id, section, name, address, birthYear);
58:     section.addEmployee(employee);
59:   }
60:
61:   /**
62:    * 部署一つ分の所属社員を表示する
63:    * (再帰的に表示する)
64:    */
65:   private void showSection(Section section, int depth) {
66:
67:     //部署のタイトルを出力
68:     indent(depth); //インデント
69:     System.out.println("---" + section.getName() + "---");
70:
71:     //部署に所属する社員を表示
72:     showEmployeeList(section.getEmployees(), depth);
73:
74:     //子部署を表示
75:     SectionList children = section.getChildren();
76:     for (int i = 0; i < children.size(); i++) {
77:       showSection(children.get(i), depth + 1);
78:     }
79:   }
80:
81:   /**
82:    * 社員リストを表示する
83:    */
84:   private void showEmployeeList(EmployeeList employees, int depth) {
85:     for (int i = 0; i < employees.size(); i++) {
86:       Employee employee = employees.get(i);
87:       indent(depth);
88:       showEmployee(employee);
89:     }
90:   }
91:
92:   /**
93:    * 社員一人分の情報を表示する
94:    */
95:   private void showEmployee(Employee employee) {
96:     System.out.println(
97:       "\t"
98:         + employee.getId()
99:         + "\t"
100:        + employee.getSection().getName()
101:        + "\t"
102:        + employee.getName()
103:        + "\t"
104:        + employee.getAddress()
105:        + "\t"
106:        + employee.getBirthYear()
107:        + "年生まれ\t"
```

```
108:         + employee.getAge()
109:         + "才");
110:   }
111:
112:   /**
113:    * インデントをつける
114:    */
115:   private void indent(int depth) {
116:     for (int i = 0; i < depth; i++) {
117:       System.out.print("\t");
118:     }
119:   }
120:
121: }
```

このサンプルプログラムが生成するオブジェクトの参照モデルは図 11.8 のようになります。[2]

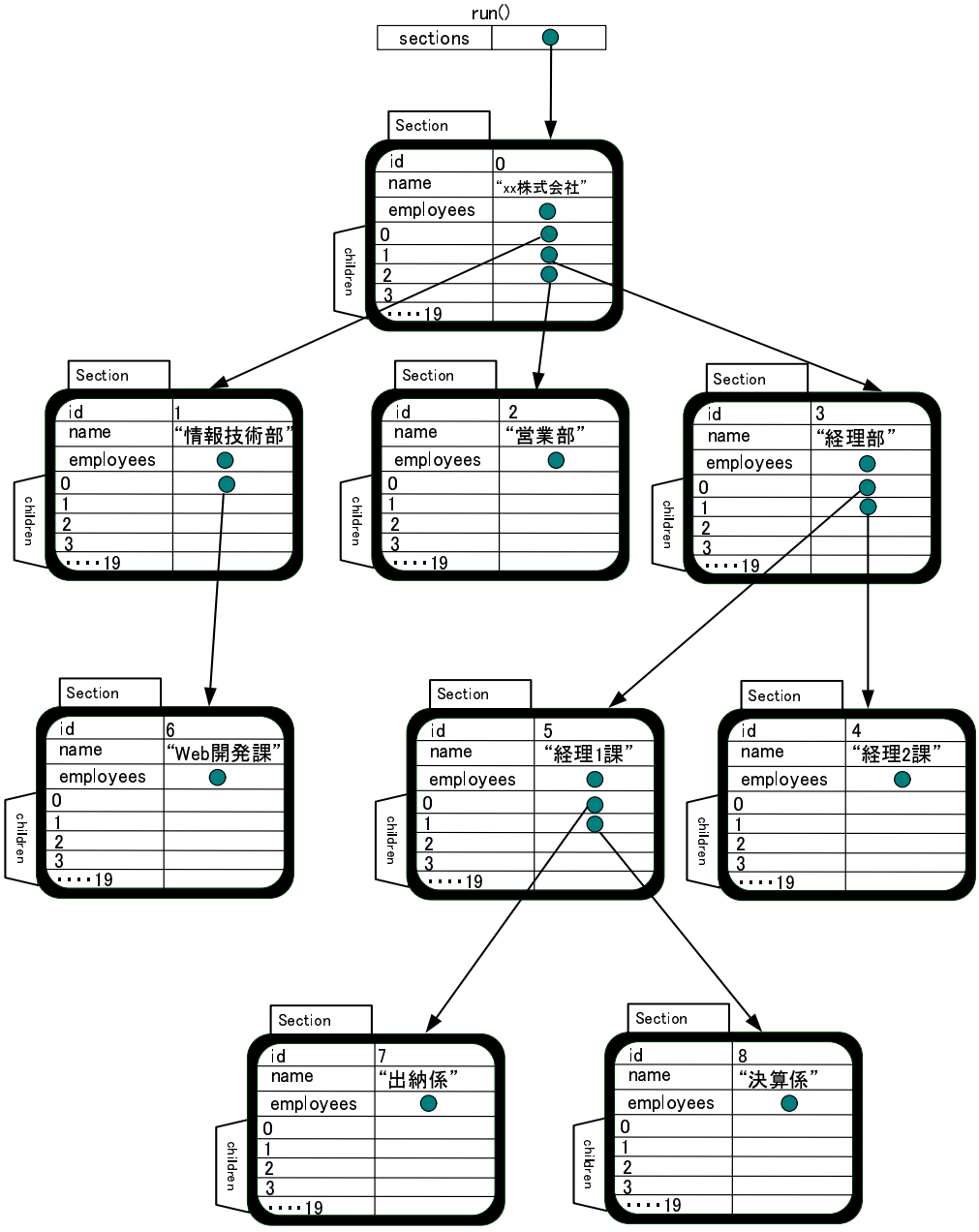


図 11.8: 部署の再帰構造参照モデル

このように再帰構造を用いれば、部署の階層構造を事実上無限に表現することができます。階層を深くしたければ、一番下の階層の部署に子の部署をどんどん足していけばよい

[2] 実際には Section のインスタンスは SectionList のインスタンスを持ち、SectionList のインスタンスが、Section のインスタンスを複数持っています。しかしここでは図をわかりやすくするため SectionList を省略し、Section のインスタンスが子部署を表す Section のインスタンスの配列を持っているような図を書いています。

のです。

再帰構造で階層構造を表現した場合、アプリケーションのクラスは、階層構造の一番上にある要素への参照を持つ必要があります。逆にそれをもっていればその子となる全ての要素への参照は階層構造を下ることで得ることができます。以下は EmployeeDirectoryApplication の持つ最上位部署への参照です。

リスト 101: 最上位部署への参照

```
18:     Section top = new Section(0, "xx 株式会社");
```

11.2.1.3 再帰構造の例

現実に存在するシステムの中には、再帰構造をつかって、階層構造を表現しているものがたくさんあります。その最も身近な例はコンピュータ上で扱われるディレクトリです。

図 11.9: ディレクトリクラス図

ディレクトリの階層構造を扱うときに、必ず必要となるのはハードディスクドライブへの参照です。現実でもディレクトリ内を探すときには windows ではｃやｄドライブを開く必要がありますね。ここでいう c ドライブというのが階層構造の最上位の要素となります。

考えてみよう

上のクラス図を見て、このクラス図が表している構造のある瞬間の参照モデルを書いてみてください。

11.2.2　連結リスト

11.2.2.1　配列を使わないデータ構造

ここまで、いろいろな場面で配列をつかってデータを格納してきました。リストはもちろん、スタックやキューも最終的には配列を使って、オブジェクトを管理していました。しかし、データを格納する方法は配列をつかったものだけではありません。この節で学んだ再帰構造の考え方を使った、連結リストという新しいデータ構造を使って社員リストを実装してみましょう。

11.2.2.2　連結リストのデータ構造

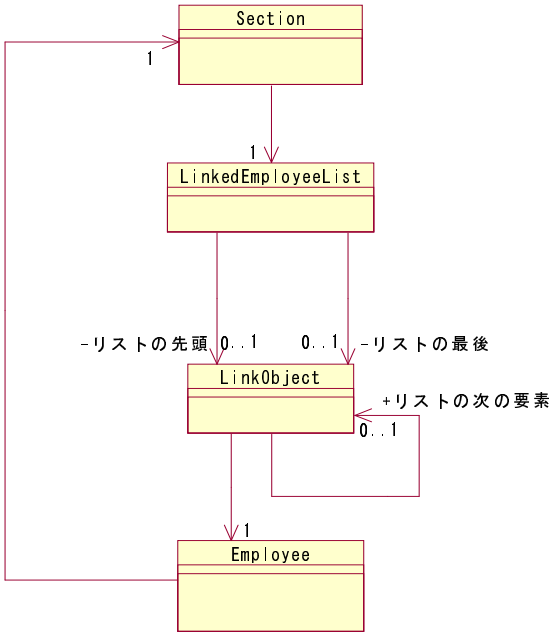


図 11.10: 連結リストクラス図

連結リストを実装するには 1 要素を格納するためのオブジェクトが必要です。ここではそれに LinkObject という名前を付け、クラスを定義します。連結リストクラスではリストの先頭となるリンクオブジェクトへの参照だけを持ちます。

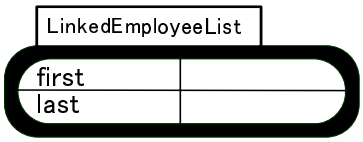


図 11.11: 空の状態の連結リスト

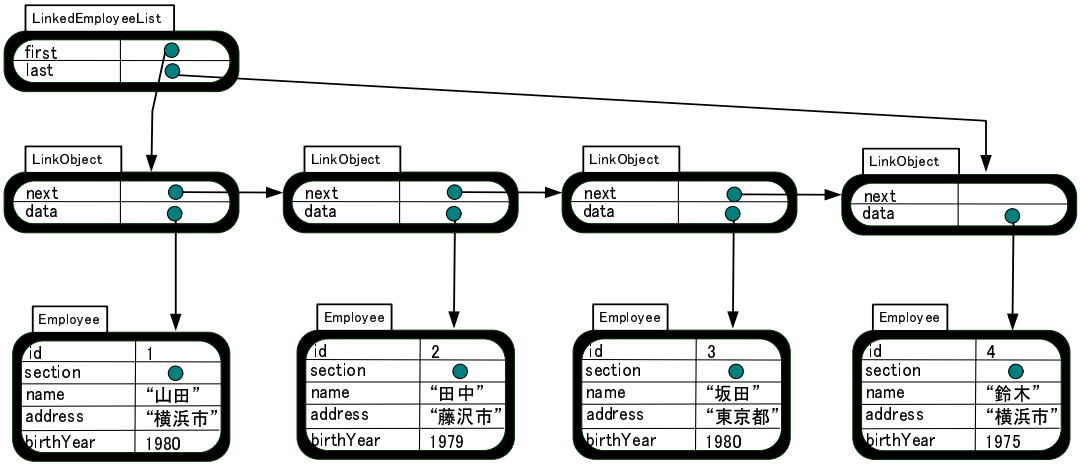


図 11.12: 連結リスト

リンクオブジェクトは、要素として格納する社員への参照を一つ持つと同時に、次のリンクオブジェクトへの参照を持ちます。このリンクオブジェクトの連鎖をリストとして扱うのが連結リストです。

11.2.2.3　連結リストの追加アルゴリズム

連結リストが空のときは、連結リストが持つ先頭のリンクオブジェクトと最後のリンクオブジェクトへの参照を、追加するリンクオブジェクトに設定します。

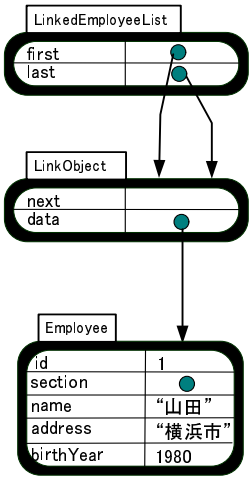


図 11.13: 連結リストへの要素の追加 (1)

それ以外のときは連結リストの最後にあるリンクオブジェクトに新たなリンクオブジェクトをつなぎます。その際連結リストが持つ最後のリンクオブジェクトへの参照を、追加するリンクオブジェクトに設定します。

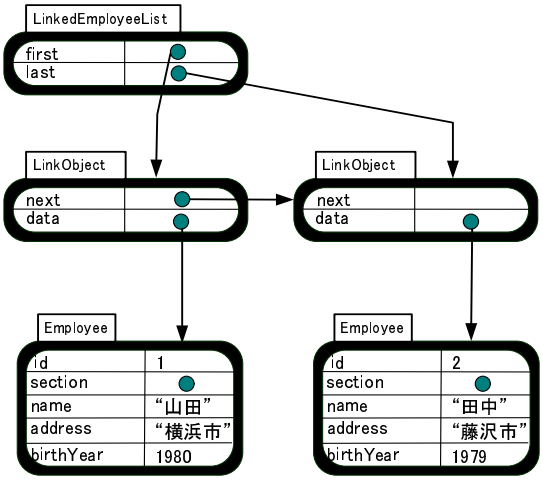


図 11.14: 連結リストへの要素の追加 (2)

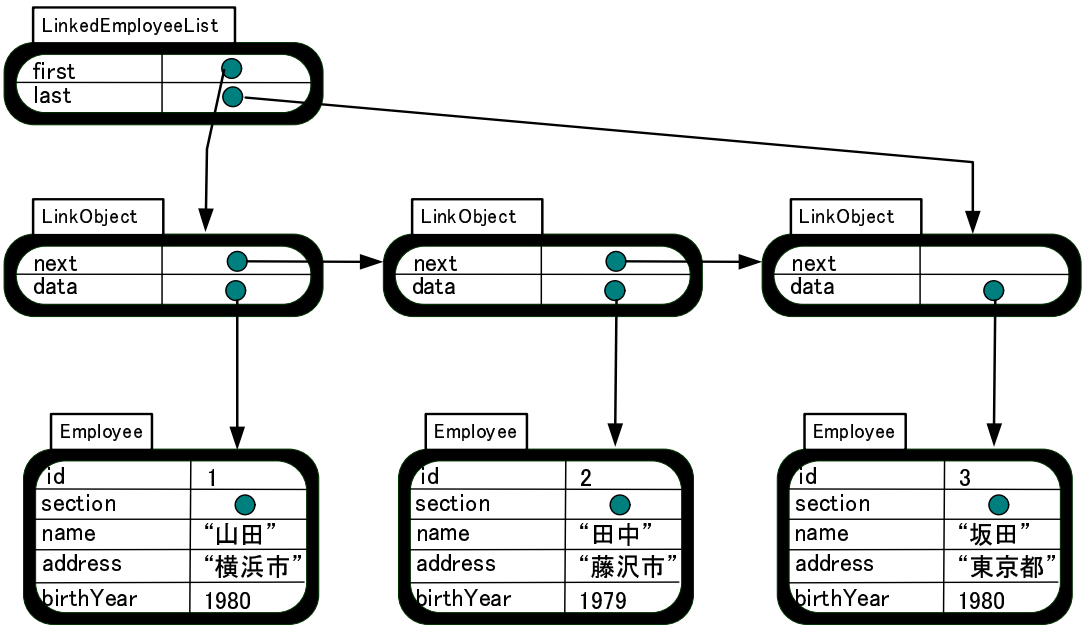


図 11.15: 連結リストへの要素の追加 (3)

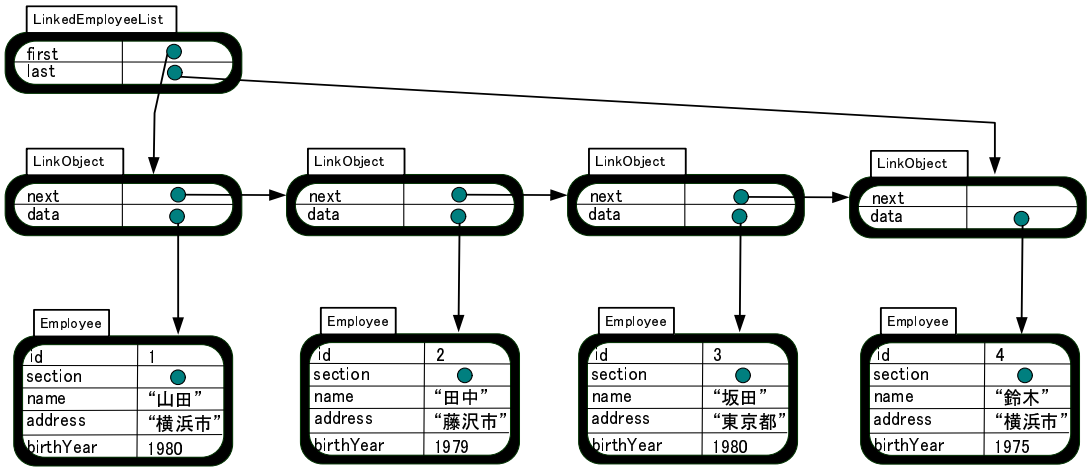


図 11.16: 連結リストへの要素の追加 (4)

11.2.2.4　連結リストの検索アルゴリズム

検索を行う際にはリストが持っている先頭のリンクオブジェクトから順にリンクオブジェクトをたどっていき、指定された要素を返します。

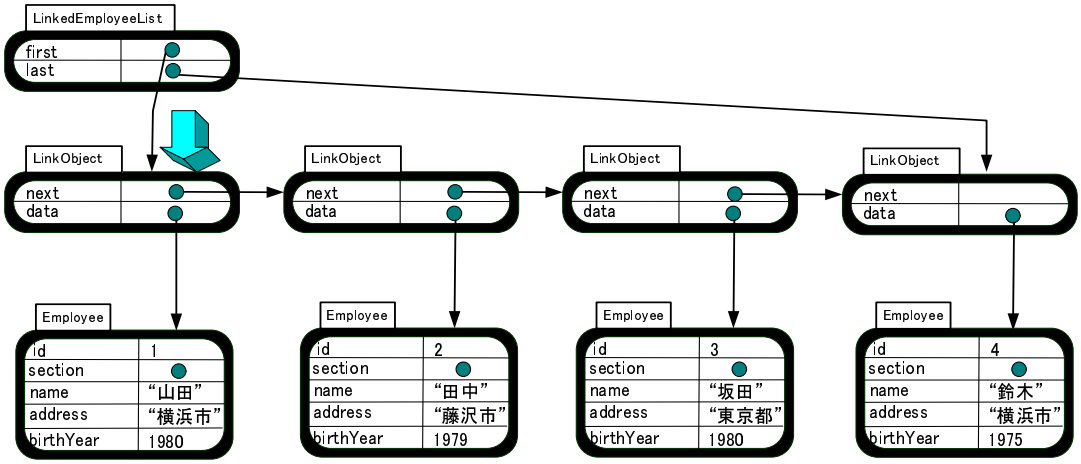


図 11.17: 連結リストでの要素の検索 (1)

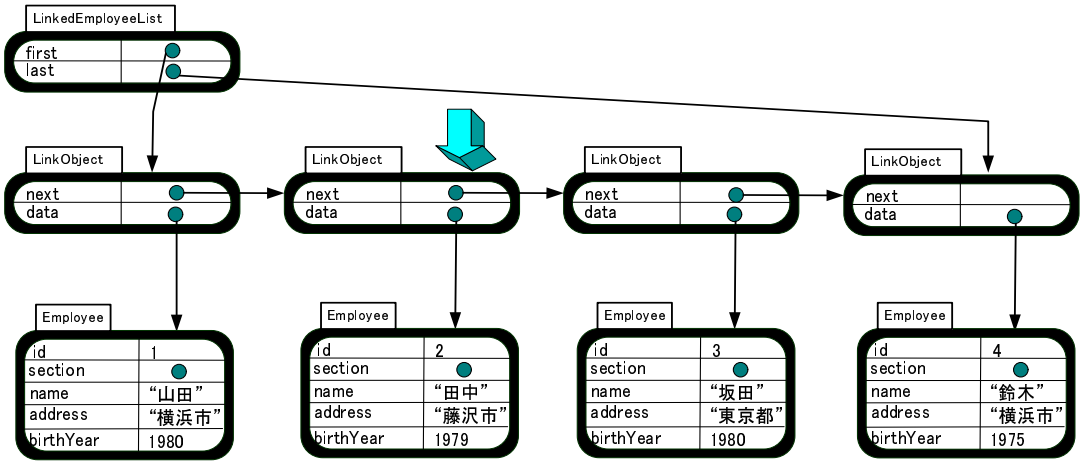


図 11.18: 連結リストでの要素の検索 (2)

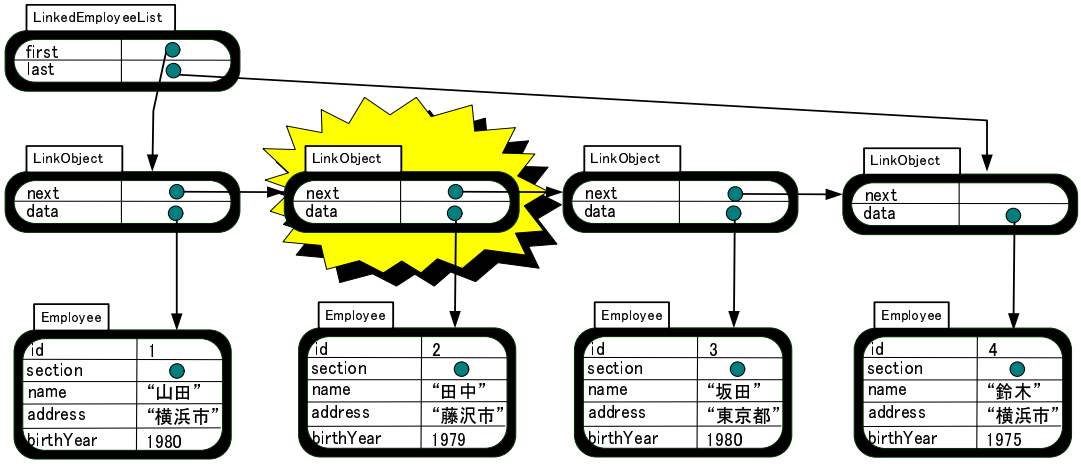


図 11.19: 連結リストでの要素の検索 (3)

もし最後まで参照をたどっても目的のオブジェクトなければ検索対象のオブジェクトが存在しないということになります。

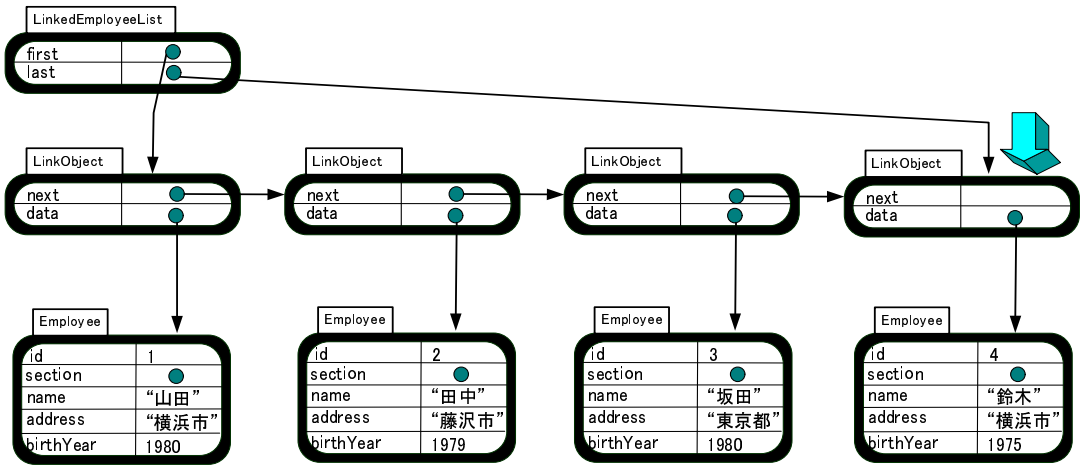


図 11.20: 連結リストでの要素の検索 (4)

11.2.2.5　連結リストの削除アルゴリズム

連結リストの削除を行う際には、まず指定されたリンクオブジェクトを検索します。連結リストの先頭に削除するオブジェクトがあったときは、連結リストオブジェクトが持つ先頭のリンクオブジェクトへの参照を変更します。

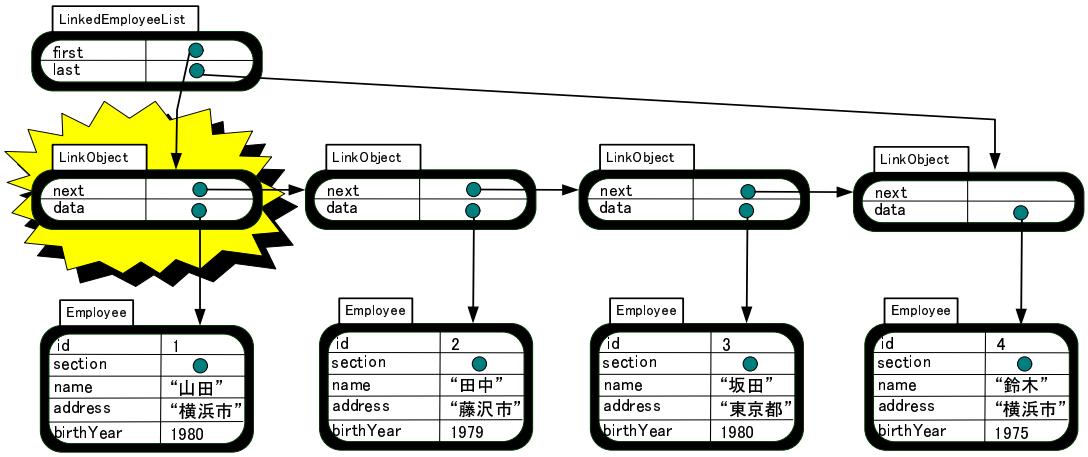


図 11.21: 先頭の要素を削除 (1)

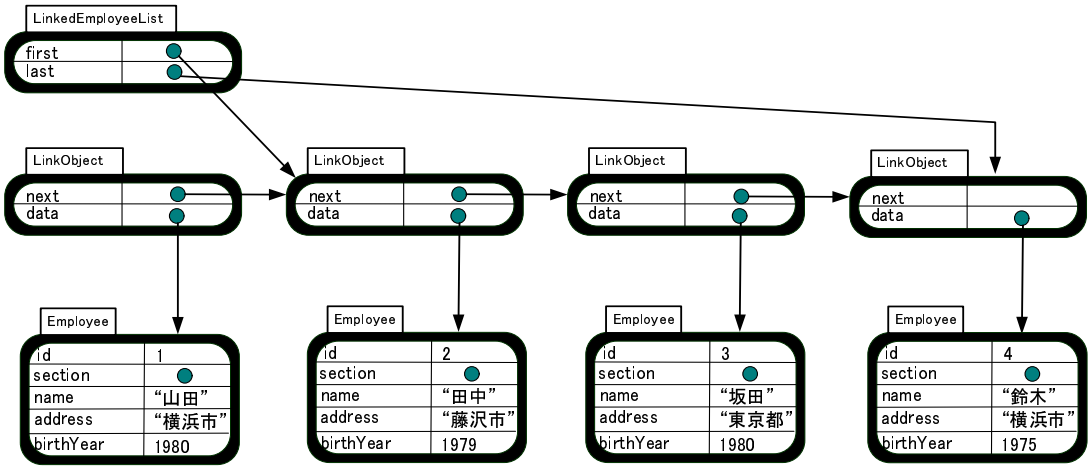


図 11.22: 先頭の要素を削除 (2)

連結リストの最後に削除するオブジェクトがあったときは、最後から 2 番目にあるリンクオブジェクトの次のリンクオブジェクトへの参照を null に設定し、連結リストオブジェクトが持つ最後のリンクオブジェクトへの参照を変更します。

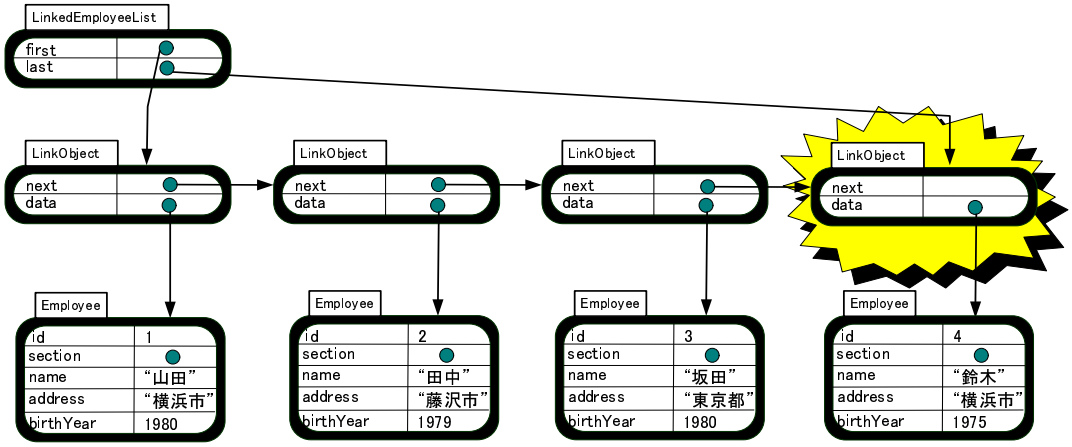


図 11.23: 最後の要素を削除 (1)

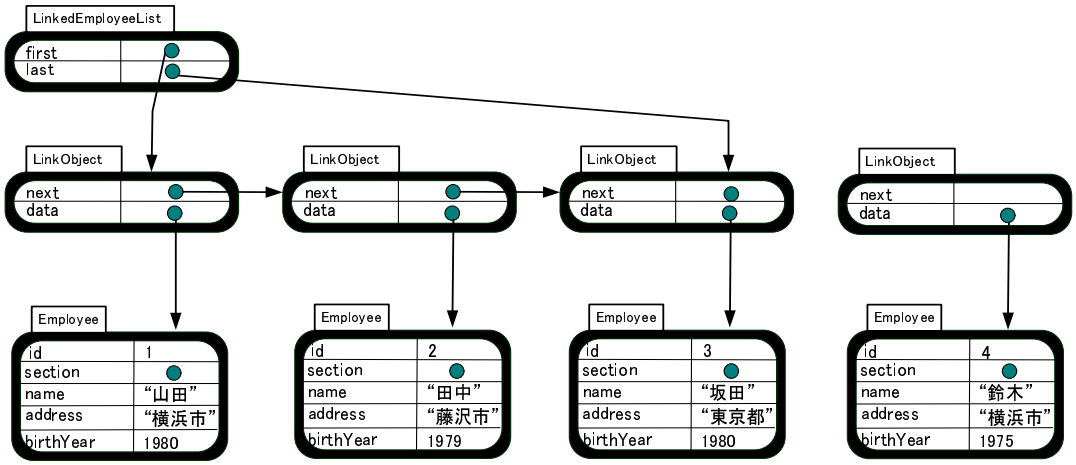


図 11.24: 最後の要素を削除 (2)

それ以外、つまり連結リストの途中に削除するオブジェクトがあったときは、削除するオブジェクトの一つ手前にあるリンクオブジェクトの次のリンクオブジェクトへの参照を削除したいリンクオブジェクトの次のリンクオブジェクトに設定します。

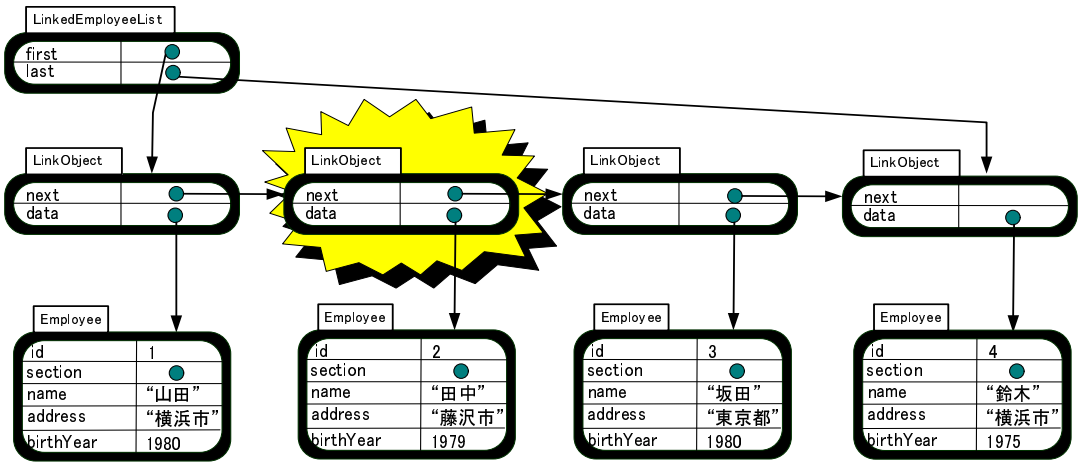


図 11.25: 中間の要素を削除 (1)

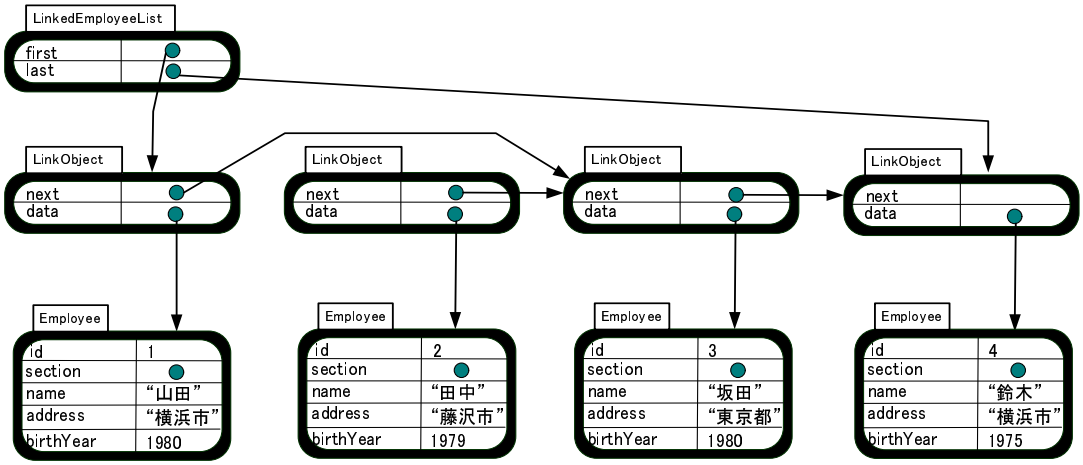


図 11.26: 中間の要素を削除 (2)

11.3　人にやさしいクラス構造の設計

11.3.1　現実世界に即したクラス構造の設計

11.3.1.1　上司をどう扱うか

社員名簿アプリケーションに直属の上司を扱うフィールドを扱うような変更を行うことを考えます。これを表現するにはどのようなクラス構造が必要でしょうか。上司も社員なのでどのように上司という役割を持つ社員を扱うかということを考えてみます。

二つの設計

まず最初に思いつくのは、社員それぞれが上司への参照を持つという構造でしょう。このような構造をつくれば上司をクラス構造で表現することができます。(図 11.27)

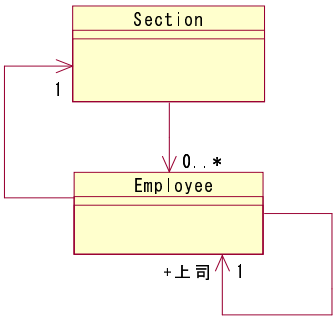


図 11.27: 上司を直接持つ

さらに、ある社員にとって上司という存在が、実際は社員が所属する部署の責任者であるということを考えると、次のような構造 (図 11.28) も考えられます。社員は、自分が所属する部署の責任者のことを上司と捉えるのです。

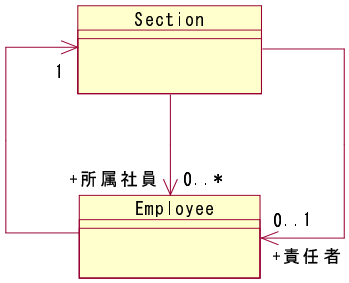


図 11.28: 部署が責任者を持つ

考えてみよう

2 つの設計におけるそれぞれ利点、欠点を挙げ、どちらがより適切かを考えてみましょう。

クラス構造の設計

現実世界は、非常に複雑な構造で成り立ったいます。オブジェクトとなりうる多くの人や物があり、それぞれの間にたくさんの関連があります。しかし、この全てをプログラムで表現することは無理ですし、その必要もありません。プログラムの目的に対して必要なオブジェクトを抽出する作業がアプリケーション開発の第一歩です。

さらに抽出したオブジェクトを、プログラムで表現するために、クラス構造に落とす作業が必要になります。これが設計と呼ばれる作業です。設計をおこなう際には、できるだけ現実世界にあてはめやすい構造を考える必要があります。そうすることによってより人間にとってわかりやすい人にやさしいクラス構造を構築することができます。

現実世界を表現するためのクラス構造にひとつの正解はありません。いくつもの答えが考えられるのです。しかし、実際に実装を行う段階ではその選択肢のうちのひとつに決定する必要があります。設計を行うという作業は、その選択肢を数多く挙げること、その選択肢それぞれの利点欠点を考え吟味すること、吟味した結果から選択肢のうちのひとつを選ぶことの 3 つで成り立っていると言えます。

11.4　練習問題

練習問題 1

空港管制塔システムのクラス図 (図 11.29) を見て、ある時点（任意でよい）での参照モデルの図を記述してください。

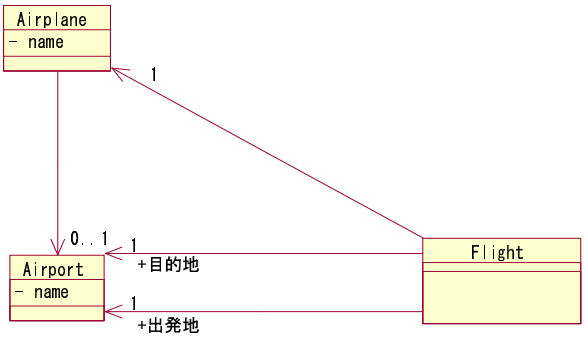


図 11.29: 空港管制塔システム

練習問題 2

レンタルビデオ屋システムのある時点での参照モデルの図 (図 11.29) を見て、クラス図を記述してください。

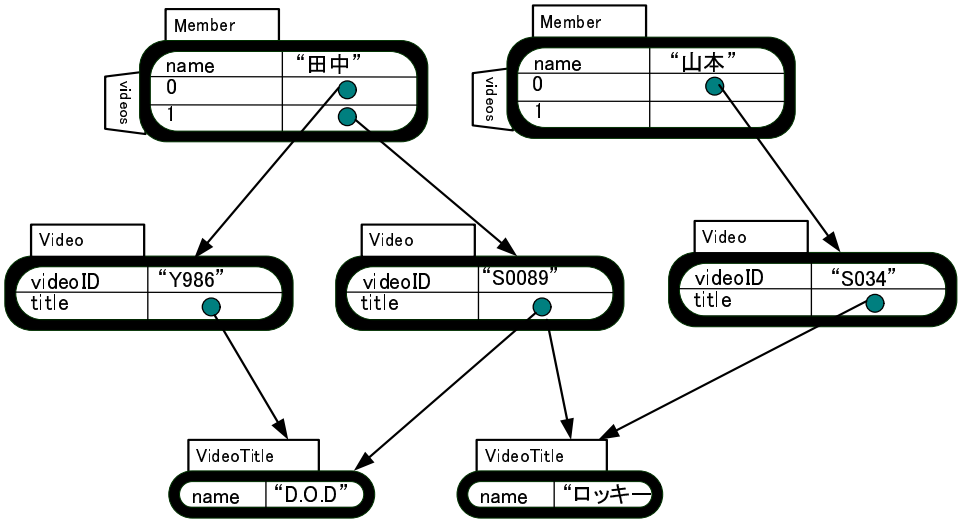


図 11.30: レンタルビデオ屋システム

練習問題 3

与えられたファイルシステム管理アプリケーション（DirectoryApplication.java がアプリケーションの本体）のうち、Directory.java を実装してプログラムを完成させてください。

練習問題 4*

与えられた社員名簿アプリケーション（EmployeeDirectoryApplication.java がアプリケーションの本体）において、LinkedEmployeeList.java を実装してプログラムを完成させてください。

第 12 章

継承を使った抽象化

この章で学習すること

継承を使ったプログラムを書くことができる
継承を使ってクラスを抽象化することの利点を説明できる
JavaAPI をつかってプログラムを書くことができる

12.1 クラスの抽象化

12.1.1 汎用的なリスト

12.1.1.1 社員リストと部署リスト

ここまでいろいろな考え方を導入して重複コードをなくし、わかりやすいソースコードを目指してきました。しかし、社員名簿アプリケーションだけではなく、今まで例題や課題で登場した多くのアプリケーションで頻繁に登場するリストの重複コードがはそのままになっています。この章ではこの重複コードをなくす方法について考えていきます。

考えてみよう

今までメソッドや、クラスなどを使っていろいろな方法で重複コードをなくすように努力してきましたが、なぜ今までの方法では SectionList と EmployeeList はひとつにまとめることができないのでしょうか。

12.1.1.2　リストの要素

SectionList と EmployeeList は引数や戻り値の型が違うだけでほとんど同じメソッドを持つクラスです。ということは Section や、Employee といったクラスは、リストにとっては「リストの要素」という意味で同じ意味のものと考えることができます。この章では部署と社員をリストの要素という意味で抽象化し、同じものとして扱うことを考えていきます。

12.1.1.3　継承

社員と部署を、リストの要素という意味でひとつのものとして扱うために、クラスの抽象化を行います。そうすることでリストのクラスの中では「リストの要素」という抽象的なものに対して処理を行い、実際に社員クラスのインスタンスなのか、部署クラスのインスタンスなのかに関しては意識する必要をなくすことができます。このようなことをクラスの抽象化といい、「部署クラスと社員クラスは、リストの要素クラスを継承している」と言います。また継承をクラス図に表すと図 12.1 のようになります。

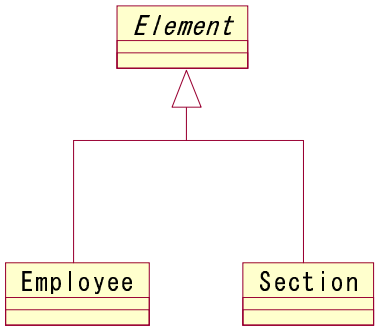


図 12.1: 継承をクラス図で表す

12.1.1.4　継承を使ったプログラム

実際にプログラムで継承を使ってみましょう。社員名簿アプリケーションの例では、まず Element というクラスをリストの要素をあらわす抽象的なクラスとして定義します。さらにこの Element を継承した、Section と Employee を定義します。

リスト 102: スーパークラスとしてのリストの要素クラス

```
1: /**
2:  * 社員と部署のスーパークラス
```

```
 3:  *
 4:  * @author bam
 5:  * @version $Id: Element.java,v 1.2 2003/05/04 21:15:54 macchan Exp $
 6:  */
 7: public class Element {
 8: }
```

リスト 103: サブクラスとしての社員クラス (定義部分のみ)

```
public class Employee extends Element implements Serializable{
```

リスト 104: サブクラスとしての部署クラス (定義部分のみ)

```
public class Section extends Element implements Serializable {
```

このように継承を適用したときに、継承をされる側のクラスをスーパークラスといい、継承する側のクラスをサブクラスといいます。

今までリストは社員クラスのインスタンスか、部署クラスのインスタンスのどちらか一方を入れるためのものでした。このような抽象化を行うことで、リストの要素クラスのインスタンスを格納するためのリスト、つまり部署と社員両方のクラスのインスタンスを格納するためのリストを定義することができます。

リスト 105: リストの要素を格納するリスト

```
 1: import java.io.Serializable;
 2:
 3: /**
 4:  * 要素を格納するリストクラス
 5:  *
 6:  * @author bam
 7:  * @version $Id: ElementList.java,v 1.3 2003/05/07 04:07:22 macchan Exp $
 8:  */
 9: class ElementList implements Serializable {
10:
11:   private final int ARRAY_SIZE = 100; //配列の大きさ
12:
13:   private Element[] elements; //要素を保存する配列
14:   private int size; //要素数
15:
16:   /**
17:    * コンストラクタ
18:    */
```

```
19:   public ElementList() {
20:     elements = new Element[ARRAY_SIZE];
21:   }
22:
23:   /**
24:    * 要素を追加する
25:    */
26:   public void add(Element element) {
27:     elements[size] = element;
28:     size++;
29:   }
30:
31:   /**
32:    * 指定されたリストの要素全員を追加する
33:    */
34:   public void addAll(ElementList elements) {
35:     for (int i = 0; i < elements.size(); i++) {
36:       add(elements.get(i));
37:     }
38:   }
39:
40:   /**
41:    * 要素を削除する
42:    */
43:   public void remove(Element element) {
44:
45:     int i; //ループ用
46:
47:     //削除対象の要素を見つける
48:     for (i = 0; i < size; i++) {
49:       if (elements[i] == element) { //見つかった
50:         elements[i] = null; //見つかったら、削除する（実は不要）
51:         break;
52:       }
53:     }
54:
55:     //削除する
56:     for (; i < size - 1; i++) { //要素を詰める
57:       elements[i] = elements[i + 1];
58:     }
59:     size--;
60:   }
61:
62:   /**
63:    * 指定された番地の要素を削除する
64:    */
65:   public void remove(int index) {
66:     for (int i = index; i < size - 1; i++) { //要素を詰める
67:       elements[i] = elements[i + 1];
68:     }
69:     size--;
70:   }
71:
72:   /**
```

```
 73:     * 全ての要素を削除する
 74:     */
 75:   public void removeAll() {
 76:     elements = new Element[ARRAY_SIZE];
 77:     size = 0;
 78:   }
 79:
 80:   /**
 81:    * 指定された番地の要素を取得する
 82:    */
 83:   public Element get(int index) {
 84:     return elements[index];
 85:   }
 86:
 87:   /**
 88:    * 要素数を取得する
 89:    */
 90:   public int size() {
 91:     return size;
 92:   }
 93:
 94:   /**
 95:    * 要素が空かどうか調べる
 96:    */
 97:   public boolean isEmpty() {
 98:     return size == 0;
 99:   }
100:
101: }
```

これらのクラスの構造をクラス図に表すと図 12.2 のようになります。

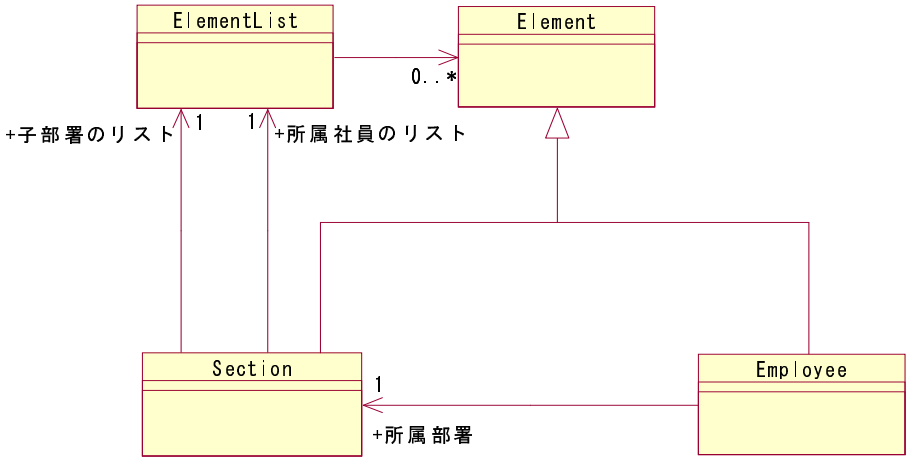


図 12.2: 汎用的なリスト

リストを使う他のクラスはどのように変わるのかを見ていきましょう。
ElementList を導入した場合のサンプルプログラムをリスト 106 に示します。

リスト 106: 部署の一覧表示をするサンプルプログラム (ElementList 導入)

```
 1: /**
 2:  * 部署クラスを用いたサンプルプログラム
 3:  * (ElementList バージョン)
 4:  *
 5:  * @author bam
 6:  * @version $Id: SectionViewSample.java,v 1.2 2003/05/07 08:08:32 macchan Exp $
 7:  */
 8: public class SectionViewSample {
 9:
10:   public static void main(String[] args) {
11:     SectionViewSample sectionViewSample = new SectionViewSample();
12:     sectionViewSample.run();
13:   }
14:
15:   private void run() {
16:
17:     //Top セクションを生成
18:     Section top = new Section(0, "xx 株式会社");
19:
20:     //部署を生成する
21:     Section infotec = new Section(1, "情報技術部");
22:     Section business = new Section(2, "営業部");
23:     Section account = new Section(3, "経理部");
24:     Section account2 = new Section(4, "経理 2 課");
25:     Section account1 = new Section(5, "経理 1 課");
26:     Section web = new Section(6, "Web 開発課");
```

```
27:     Section casher = new Section(7, "出納係");
28:     Section settle = new Section(8, "決算係");
29:
30:     //部署の階層構造を構築
31:     top.addChild(infotec);
32:     top.addChild(business);
33:     top.addChild(account);
34:
35:     account.addChild(account2);
36:     account.addChild(account1);
37:
38:     infotec.addChild(web);
39:
40:     account1.addChild(casher);
41:     account1.addChild(settle);
42:
43:     //全ての部署/社員リストを表示
44:     showSection(top, 0);
45:   }
46:
47:   /**
48:    * 社員を一人追加する
49:    */
50:   private void addEmployee(
51:     int id,
52:     Section section,
53:     String name,
54:     String address,
55:     int birthYear) {
56:
57:     Employee employee = new Employee(id, section, name, address, birthYear);
58:     section.addEmployee(employee);
59:   }
60:
61:   /**
62:    * 部署一つ分の所属社員を表示する
63:    * (再帰的に表示する)
64:    */
65:   private void showSection(Section section, int depth) {
66:
67:     //部署のタイトルを出力
68:     indent(depth); //インデント
69:     System.out.println("---" + section.getName() + "---");
70:
71:     //部署に所属する社員を表示
72:     showEmployeeList(section.getEmployees(), depth);
73:
74:     //子部署を表示
75:     ElementList children = section.getChildren();
76:     for (int i = 0; i < children.size(); i++) {
77:       showSection((Section) children.get(i), depth + 1);
78:     }
79:   }
80:
```

```
 81:   /**
 82:    * 社員リストを表示する
 83:    */
 84:   private void showEmployeeList(ElementList employees, int depth) {
 85:     for (int i = 0; i < employees.size(); i++) {
 86:       Employee employee = (Employee) employees.get(i);
 87:       indent(depth);
 88:       showEmployee(employee);
 89:     }
 90:   }
 91:
 92:   /**
 93:    * 社員一人分の情報を表示する
 94:    */
 95:   private void showEmployee(Employee employee) {
 96:     System.out.println(
 97:       "\t"
 98:         + employee.getId()
 99:         + "\t"
100:         + employee.getSection().getName()
101:         + "\t"
102:         + employee.getName()
103:         + "\t"
104:         + employee.getAddress()
105:         + "\t"
106:         + employee.getBirthYear()
107:         + "年生まれ\t"
108:         + employee.getAge()
109:         + "才");
110:   }
111:
112:   /**
113:    * インデントをつける
114:    */
115:   private void indent(int depth) {
116:     for (int i = 0; i < depth; i++) {
117:       System.out.print("\t");
118:     }
119:   }
120:
121: }
```

社員追加のメソッドの中では、Employee クラスのインスタンスを生成してリストに追加しています。しかし、ElementList の add メソッドの引数は Element になっているはずです。これまで、引数の違うメソッドは呼ぶことができないと学んできました。変数にインスタンスを代入する場合も同様に、同じクラスのインスタンスしか代入することができないと学んできました。では、なぜこのようなことが可能なのでしょうか？

Employee は Element を継承しています。言い換えると Employee は、ある種の Element であるということができます。Employee は抽象的な定義である Element を具体化した一つのかたちなのです。Section も同様です。この考え方にたつと、Employee のイ

ンスタンスは Employee のインスタンスであると同時に Element のインスタンスとして扱うことができます。

　このようにサブクラスのインスタンスは、そのスーパークラスのインスタンスとして扱うことができます。ですから、スーパークラスの変数にサブクラスのインスタンスを代入したり、スーパークラスを引数にとるメソッドにサブクラスのインスタンスを引数として渡したりすることができます。

12.1.2　オブジェクトと型

それでは、このような何でも入るリストを使う側のプログラムを書くことを考えてみましょう。

考えてみよう

リスト 107: キャストをするサンプルプログラム (コンパイルエラー)

```
Section section = elements.get(0);
```

このプログラムはコンパイルエラーがでてしまいます。なぜコンパイルできないのか考えてみましょう。

12.1.2.1　型の変換

このように Element のようなスーパークラスのインスタンスとして扱っていたオブジェクトを、もとのサブクラスのインスタンスとして扱いたいときには型の変換を行う必要があります。この型変換のことをキャストといいます。

リスト 108: キャストをするサンプルプログラム

```
Section section = (Section)elements.get(0);
```

12.2 コレクション API

12.2.1 JavaAPI とクラスライブラリ

12.2.1.1 再利用可能なクラス

今まで多くのプログラムで見てきたように、配列とそれを管理するクラスは多くのアプリケーションで使われています。しかし、そのたびにリストのクラスをいちいちプログラムすることは労力の無駄です。

そのような問題を解決するために、この節で学んできた考え方を使って**何でも入るリスト**をつくれば、それを再利用することができます。

12.2.1.2 Object クラス

前の節でつくった ElementList クラスは、部署も社員も入れることのできるリストです。しかし、このままではこの ElementList にいれるクラスは全て、Element を継承しなければなりません。このような問題を解決するために、Java には**何でも**をあらわすクラスが導入されています。このクラスのことを Object クラスと言います。

それでは何でもをあらわすクラスとはどのようなことなのでしょうか。Object クラスは、全てのクラスのスーパークラスです。今まで書いてきたプログラムのように、何も継承していないクラスを書いたとしても、実際には全てのクラスは Object クラスを継承しています。何も継承していないように見えるのは、「extends Object」という記述が省略されているためです。Object クラスを格納できるリストを作ることにより、何でもはいるリストをつくることができます。

12.2.1.3 JavaAPI

実は、このような何でも入るリストは、Java ではすでにライブラリとして用意されています。このライブラリのことを JavaAPI と呼び、プログラムを書くときに適宜使うことができます。JavaAPI には他にも、ファイル入出力や、文字列の操作などプログラムを書くときによくでてくる機能が用意されています。今まで当たり前のように使ってきた String もこの javaAPI の一部です。

12.2.2　コレクションフレームワーク

12.2.2.1　コレクションフレームワークとは

今までつかってきたリストのような、オブジェクトの集合を扱うことはプログラムを書く上で最もよくでてくる処理の一つです。

このようなオブジェクトの集合を扱うライブラリとして、Java にはコレクションフレームワークというものが用意されています。

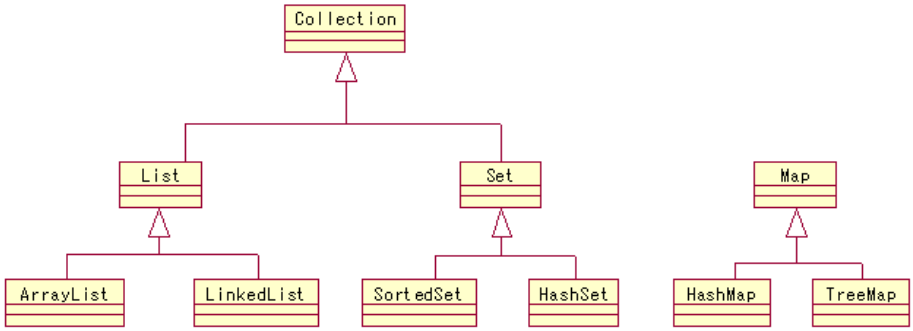


図 12.3: コレクションフレームワーク

12.2.2.2　JavaAPI を使ったプログラム

それでは実際に配列で実装されたリストを表す ArrayList を社員名簿プログラムの中で使ってみましょう。

リスト 109: 部署クラス (ArrayList を使う)

```
 1: import java.io.Serializable;
 2: import java.util.ArrayList;
 3:
 4: /**
 5:  * 部署クラス
 6:  *
 7:  * 所属する社員の参照を持つ
 8:  * 親部署、子部署の階層構造を持つ
 9:  *
10:  * @author bam
11:  * @version $Id: Section.java,v 1.3 2003/05/07 08:21:50 rocky Exp $
12:  */
13: public class Section implements Serializable {
14:
15:   private int id; //部署 ID
16:   private String name; //部署名
```

```
17:
18:   //  社員の管理
19:   private ArrayList employees = new ArrayList(); //部署に所属する社員
20:
21:   //  部署の階層構造の管理
22:   private ArrayList children = new ArrayList(); //この部署の子となる部署のリスト
23:
24:   /**
25:    * コンストラクタ
26:    */
27:   public Section(int id, String name) {
28:     this.id = id;
29:     this.name = name;
30:   }
31:
32:   /**
33:    * 部署の ID を取得する
34:    */
35:   public int getId() {
36:     return id;
37:   }
38:
39:   /**
40:    * 部署の名前を取得する
41:    */
42:   public String getName() {
43:     return name;
44:   }
45:
46:   /**
47:    * 部署の名前を設定する
48:    */
49:   public void setName(String name) {
50:     this.name = name;
51:   }
52:
53:   /*****************************************************
54:   /* この部署に所属する社員の管理
55:   /*****************************************************/
56:
57:   /**
58:    * 部署に所属する社員を追加する
59:    */
60:   public void addEmployee(Employee employee) {
61:     employees.add(employee);
62:   }
63:
64:   /**
65:    * 部署に所属する社員を削除する
66:    */
67:   public void removeEmployee(Employee employee) {
68:     employees.remove(employee);
69:   }
70:
```

```
 71:   /**
 72:    * 部署に所属する社員を index で指定して取得する
 73:    */
 74:   public Employee getEmployee(int index) {
 75:     return (Employee)employees.get(index);
 76:   }
 77:
 78:   /**
 79:    * 部署に所属する社員のリストを取得する
 80:    */
 81:   public ArrayList getEmployees() {
 82:     return employees;
 83:   }
 84:
 85:   /**
 86:    * この部署に所属する社員がいるかどうか調べる
 87:    */
 88:   public boolean hasEmployee() {
 89:     return !(employees.size() == 0);
 90:   }
 91:
 92:   /*****************************************************
 93:   /* この部署にふくまれる部署の管理
 94:   /*****************************************************/
 95:
 96:   /**
 97:    * この部署の子となる部署を追加する
 98:    */
 99:   public void addChild(Section section) {
100:     children.add(section);
101:   }
102:
103:   /**
104:    * この部署の子となる部署を index を指定して削除する
105:    */
106:   public void removeChild(int index) {
107:     children.remove(index);
108:   }
109:
110:   /**
111:    * この部署の子となる部署を index で指定して取得する
112:    */
113:   public Section getChild(int index) {
114:     return (Section) children.get(index);
115:   }
116:
117:   /**
118:    * この部署の子となる部署のリストを取得する
119:    * (再帰的に取得する)
120:    */
121:   public ArrayList getChildren() {
122:     return children;
123:   }
124:
```

```
125:   /**
126:    * この部署に含まれる部署が存在するか調べる
127:    */
128:   public boolean hasChild() {
129:     return !(children.size() == 0);
130:   }
131:
132:   /*****************************************************
133:   /* 再帰導出処理
134:   /*****************************************************/
135:
136:   /**
137:    * 部署に所属する社員のリストを再帰的に導出する
138:    */
139:   public ArrayList navigateAllEmployees() {
140:     ArrayList navigatedEmployees = new ArrayList(); //導出された社員のリスト
141:
142:     //この部署の社員をリストに加える
143:     navigatedEmployees.addAll(employees);
144:
145:     //全ての子の部署の社員をリストに加える
146:     for (int i = 0; i < children.size(); i++) {
147:       Section section = (Section) children.get(i);
148:       navigatedEmployees.addAll(section.navigateAllEmployees());
149:     }
150:
151:     return navigatedEmployees;
152:   }
153:
154:   /**
155:    * この部署の子となる部署のリストを再帰的に導出する
156:    */
157:   public ArrayList navigateAllSections() {
158:     ArrayList navigatedSections = new ArrayList(); //導出された社員のリスト
159:
160:     //この部署の部署をリストに加える
161:     navigatedSections.addAll(children);
162:
163:     //全ての子の部署の部署をリストに加える
164:     for (int i = 0; i < children.size(); i++) {
165:       Section section = (Section) children.get(i);
166:       navigatedSections.addAll(section.navigateAllSections());
167:     }
168:
169:     return navigatedSections;
170:   }
171:
172: }
```

JavaAPI のような外部のライブラリをプログラムの中で使うためには、プログラムの先頭に import 文を書く必要があります。これは、そのライブラリを使うことを明示的に宣言するためのものです。

12.2.3　JavaAPI ドキュメント

12.2.3.1　ドキュメントを読む

JavaAPI [1] には、膨大な数のクラスがライブラリとして用意されています。これらのクラスの使い方を全て覚えることはできません。このようなときのために、Web に JavaAPI の仕様が公開されています。ここから必要な機能を探し出し、適宜使うようにしましょう。

[1] http://java.sun.com/j2se/1.4/ja/docs/ja/index.html

12.3 練習問題

練習問題 1

これまでに実装した辞書アプリケーション（DictionaryApplication.java, Word.java, WordList.java）のうち、WordList.java は ArrayList を使うように変更してください。

練習問題 2

これまでに実装した自動販売機アプリケーション（VendingMachineApplication.java, Item.java, ItemStockQueue.java, ItemType.java, ItemTypeList.java）のうち、ItemStockQueue.java は LinkedList を使い、ItemTypeList.java は ArrayList を使うように変更してください。

第 III 部

付録

付録 A

ミニプロジェクト

A.1 第 I 部

A.1.1 概要

ミニプロジェクトでは、小さくてもすこし実用的なプログラムを作ってみようと思います。ユーザは、あなた自身なので、どう作ってもらおうと構いませんが、手を抜いたばかりに使用するときに面倒な思いをするのも、あなたです。

課題は、**金銭出納帳**です。

まず、金銭出納帳のフォーマット（どのように罫線が引かれていて、どんなふうに項目がならんでいるか）を思い出してみてください。これをコンピュータの画面に再現するのです。

金銭出納帳の一行には、

年月日
摘要あるいは項目
収入
支出
（その時点での）残高

が少なくとも書かれているはずです。これが一度に入力する情報となるでしょう。但し繰越金を前もって入力していれば、残高は毎回計算で求められるので、あえて入力しなくてもプログラムで計算すればよいでしょう。

さらに、後で項目ごとに集計できるように

分類

も入力するようにしておくと、プログラミングするだけの意味が、十分に満喫できると思います。この程度の問題ならば、プログラムのテクニックとしては今まで習ってきたことにプラス α するだけで、かなりの部分がうまくいくと思います。むしろプログラムをどう作るか、といった設計のほうが重要になってくると思います。

A.1.2　例題: 社員住所録管理プログラム

今回、例題として住所録プログラムを用意しました。実行してみれば分かるように、今までの例題と比べ、アプリケーションプログラムとしての見栄えが少しだけよくなっている気がします。

この住所録管理プログラムをよく読んでみましょう。このプログラムの中には金銭出納帳を作成するための情報が隠されています。ミニプロジェクトでは、利用できるものは最大限に利用することを心がけてください。

A.1.3　画面制御用メソッド

例題の住所録管理プログラムでは、ミニプロ専用の画面制御用メソッドがいくつか使われています。

ConsoleWindow.[メソッド名]　と記述されている計 4 種類のメソッドです。以下にメソッドの説明を載せました。

void initializeWindow(int x, int y, int width, int height)　画面（ウインドウ）を引数で指定された大きさ、位置に初期化して、画面を表示させます。一度しか呼ばないように。呼ばれた後、System.out.print 類での出力が、その画面上に表示されるようになります。

void locateCursor(int row, int col)　引数で与えられた位置 (row:行 col:列) にカーソルを移動します。カーソルの移動した位置に System.out.print 類が出力されます。

void clearLine()　現在カーソルがある位置の行にある文字を全て消去します。

void clearScreen()　画面を全て消去します

A.1.4　演習問題

A.1.4.1　プロトタイプ

住所録管理プログラムプロトタイプ (リスト 110) を参考にして以下の演習を行ってください。

1. 金銭出納帳プログラムの仕様を考えてください。
   最初に繰り越し金額を入力できることが必要
   データを追加するコマンドが必要
   データが多くなると一ページに表示できなくなる。ページングするコマンドが必要
2. 画面表示をどのようにするか考えてください。
3. 金銭出納帳プログラムのプロトタイプを作ってください。

A.1.4.2 ファイル機能

住所録管理プログラムファイル機能付き (リスト 111) を参考にして以下の演習を行ってください。

1. 金銭出納帳プログラムの仕様に新しい機能を追加してください。
   データをファイルに保存するためにセーブコマンドが必要
   セーブしたデータを読み込むためにロードコマンドが必要
2. ファイルのフォーマットを考えてください。
3. ファイル機能付き金銭出納帳プログラムを作ってください。

A.1.4.3 編集機能

住所録管理プログラム編集機能付き (リスト 112) を参考にして以下の演習を行ってください。

1. 金銭出納帳プログラムの仕様に新しい機能を追加してください。
   過去のデータを修正するために編集コマンドが必要
   必要のないデータを消去するために削除コマンドが必要
2. 金額の再計算について考えてください。
3. 編集機能付き金銭出納帳プログラムを作ってください。

A.1.4.4 独自機能

以下の演習を行ってください。

1. 金銭出納帳プログラムの新しい仕様を考えてください。例えば、次のようなものが考えられます。
   検索（項目名で）
   分類別集計
   エンゲル係数計算
2. 独自機能の設計してください。
3. 独自機能付き金銭出納帳プログラムを作ってください。

A.1.5 例題ソース

A.1.5.1 プロトタイプ

リスト 110: 社員住所録管理アプリケーション (プロトタイプ)

```
 1: /**
 2:  * 社員名簿管理アプリケーション（プロトタイプ）
 3:  *
 4:  * 本アプリケーションの機能
 5:  * ○名簿の管理
 6:  *      ・社員の追加
 7:  * ○名簿の閲覧
 8:  *      ・ページの移動
 9:  *      ・カーソルの移動
10:  *
11:  * @author Manabu Sugiura
12:  * @version $Id: EmployeeDirectoryApplication.java,v 1.5 2003/05/20 15:10:48 gackt Exp $
13:  */
14: public class EmployeeDirectoryApplication {
15:
16:   public static void main(String[] args) {
17:     EmployeeDirectoryApplication employeeDirectoryApplication =
18:       new EmployeeDirectoryApplication();
19:     employeeDirectoryApplication.main();
20:   }
21:
22:   /*****************************************************
23:   /* 定数
24:   /*****************************************************/
25:
26:   //データ関連の定数
27:   final int RECORD_MAX = 6; //保存できる最大データの数
28:   final int FIELD_COUNT = 9; //一件当たりのデータフィールドの数
29:
30:   //画面制御用の定数
31:   final int DISPLAY_RECORD_MAX = 5; //1 ページに書くデータ件数
32:   final int TITLE_LINE = 1; //タイトル行の位置
33:   final int TITLE_POSITION = 25; //タイトルの位置
34:   final int DATA_START_LINE = 3; //データ行の先頭位置
35:   final int LINES_FOR_A_RECORD = 4; //1 レコードの表示に必要な行数
36:   //  メッセージ行の先頭位置
37:   final int MESSAGE_START_LINE =
38:     DATA_START_LINE + LINES_FOR_A_RECORD * DISPLAY_RECORD_MAX;
39:   //コマンドエラーメッセージの行の位置
40:   final int COMMAND_ERROR_LINE = MESSAGE_START_LINE;
41:   //  コマンドメッセージの行の位置
42:   final int COMMAND_INFO_LINE = MESSAGE_START_LINE + 1;
43:   //コマンドプロンプトの行の位置
44:   final int COMMAND_INPUT_LINE = COMMAND_INFO_LINE + 1;
45:   //  追加メッセージ行の位置
```

```
 46:   final int APPEND_LINE = MESSAGE_START_LINE;
 47:   // メッセージフィールドの最終行の位置
 48:   final int MESSAGEAREA_TERMINAL_LINE = MESSAGE_START_LINE + 20;
 49:
 50:   //コマンド類
 51:   final String ADD = "A"; //データの追加
 52:   final String PREVIOUS_PAGE = "P"; //前ページへ
 53:   final String NEXT_PAGE = "N"; //次ページへ
 54:   final String UP_CURSOR = "U"; //カーソルを上へ
 55:   final String DOWN_CURSOR = "D"; //カーソルを下へ
 56:   final String QUIT = "Q"; //終了
 57:   final String[] COMMANDS =
 58:     { ADD, PREVIOUS_PAGE, NEXT_PAGE, UP_CURSOR, DOWN_CURSOR, QUIT };
 59:
 60:   //表示文字列類
 61:   final String LINE =
 62:     "------------------------------------------------------------------"
 63:       + "------------------------------------------------------------------"
 64:       + "------------------------------------------------------------------";
 65:   final String SPACE = "  ";
 66:   final String SC = ":";
 67:   final String ADD_MSG = ADD + SC + "追加" + SPACE;
 68:   final String PP_MSG = PREVIOUS_PAGE + SC + "前ページへ" + SPACE;
 69:   final String NP_MSG = NEXT_PAGE + SC + "次ページへ" + SPACE;
 70:   final String UC_MSG = UP_CURSOR + SC + "前のデータへ" + SPACE;
 71:   final String DC_MSG = DOWN_CURSOR + SC + "次のデータへ" + SPACE;
 72:   final String QUIT_MSG = QUIT + SC + "終了" + SPACE;
 73:   final String COMMAND_INFO_MESSAGE =
 74:     "コマンド"
 75:       + "("
 76:       + SPACE
 77:       + ADD_MSG
 78:       + PP_MSG
 79:       + NP_MSG
 80:       + UC_MSG
 81:       + DC_MSG
 82:       + QUIT_MSG
 83:       + ")";
 84:   final String COMMAND_PROMPT = " >> ";
 85:   final String COMPANY_NAME = "○×コミュニケーションズ";
 86:   final String REMOVE_CONFIRM_MESSAGE = "本当に削除してもいいですか？ y/n";
 87:   final String YES = "Y";
 88:
 89:   final String COMMAND_ERROR_MESSAGE = "不適当なコマンドでした";
 90:   final String INCORRECT_INPUT_ERROR_MESSAGE = "不正な入力です　数字を入力してくださ
い";
 91:   final String DATA_OVERFLOW_ERROR_MESSAGE = "データがいっぱいで、もう追加できません";
 92:   final String NO_DATA_UNDELETALBLE_ERROR_MESSAGE = "削除可能なデータがありません";
 93:   final String NO_DATA_UNEDITABLE_ERROR_MESSAGE = "編集可能なデータがありません";
 94:
 95:   //カーソル移動用
 96:   final int UP = 1;
 97:   final int DOWN = 2;
 98:
```

```
 99:   //画面の大きさ用
100:   final int X = 100;
101:   final int Y = 100;
102:   final int WIDTH = 850;
103:   final int HEIGHT = 600;
104:
105:   /*****************************************************
106:   /* インスタンス変数
107:   /*****************************************************/
108:
109:   int currentRecordIndex = -1; //現在カーソルがあるインデックス。何もないときは-1
110:   int recordCount; //現在の要素数
111:
112:   //社員の情報を保存しておくための配列群
113:   int[] id = new int[RECORD_MAX]; //社員 ID
114:   String[] section = new String[RECORD_MAX]; //部署
115:   String[] name = new String[RECORD_MAX]; //漢字の名前
116:   String[] kana = new String[RECORD_MAX]; //名前のふりがな
117:   String[] zipCode = new String[RECORD_MAX]; //郵便番号
118:   String[] address = new String[RECORD_MAX]; //住所
119:   String[] telephoneNumber = new String[RECORD_MAX]; //電話番号
120:   int[] birthYear = new int[RECORD_MAX]; //生まれた年
121:   int[] entranceYear = new int[RECORD_MAX]; //入社年度
122:   int[] age = new int[RECORD_MAX]; //年齢
123:   int[] serviceLength = new int[RECORD_MAX]; //勤続年数
124:
125:   /*****************************************************
126:   /* メイン
127:   /*****************************************************/
128:
129:   void main() {
130:
131:     String command; //入力されたコマンド
132:
133:     //初期化
134:     ConsoleWindow.initializeWindow(X, Y, WIDTH, HEIGHT);
135:     initializeTitle();
136:
137:     //コマンドの入力と実行
138:     while (!((command = inputCommand()).equals(QUIT))) {
139:       executeCommand(command);
140:       showOneDirectoryPage();
141:     }
142:
143:     //終了処理
144:     ConsoleWindow.clearScreen();
145:     System.exit(0);
146:   }
147:
148:   /*****************************************************
149:   /* コマンド処理
150:   /*****************************************************/
151:
152:   /**
```

```
153:    * 社員名簿のコマンドを入力し、妥当なコマンドであれば、入力されたコマンドを返す。
154:    * もしも、不適当なコマンドであれば、再入力を要求する。
155:    */
156:   String inputCommand() {
157:
158:     String command = null; //入力されたコマンド
159:
160:     //メニューを出す
161:     showCommandMenu();
162:
163:     while (true) {
164:
165:       //プロンプトを出す
166:       showCommandPrompt();
167:
168:       //コマンドを入力する
169:       command = Input.getString();
170:       command = command.toUpperCase(); //入力されたコマンドを大文字に
171:
172:       //正しいコマンドならば終了して入力コマンドを返す。
173:       //正しくなければエラーを表示して再入力
174:       if (isCorrectCommand(command)) {
175:         break;
176:       } else {
177:         showError(COMMAND_ERROR_MESSAGE);
178:       }
179:     }
180:
181:     //画面の後始末
182:     clearMessageArea();
183:
184:     return command;
185:   }
186:
187:   /**
188:    * 指定された社員名簿のコマンドを実行する
189:    */
190:   void executeCommand(String command) {
191:
192:     if (command.equals(ADD)) { //追加
193:       addEmployee();
194:     } else if (command.equals(PREVIOUS_PAGE)) { //前ページ
195:       paging(UP);
196:     } else if (command.equals(NEXT_PAGE)) { //次ページ
197:       paging(DOWN);
198:     } else if (command.equals(UP_CURSOR)) { //前のデータ
199:       moveCursor(UP);
200:     } else if (command.equals(DOWN_CURSOR)) { //次のデータ
201:       moveCursor(DOWN);
202:     }
203:   }
204:
205:   /*****************************************************
206:   /* 名簿管理コマンド群
```

```
207:  /*****************************************************/
208:
209:  /**
210:   * 社員名簿データに、新しい社員を追加する
211:   */
212:  void addEmployee() {
213:
214:    //データが満杯だったらエラーを表示して終了する
215:    if (recordCount >= RECORD_MAX) {
216:      showError(DATA_OVERFLOW_ERROR_MESSAGE);
217:      return;
218:    }
219:
220:    //データを入力する
221:    showPrompt("社員 ID");
222:    int inputId = getValidInt();
223:    showPrompt("所属部署 ");
224:    String inputSection = Input.getString();
225:    showPrompt("名前 ");
226:    String inputName = Input.getString();
227:    showPrompt("ふりがな ");
228:    String inputKana = Input.getString();
229:    showPrompt("郵便番号 ");
230:    String inputZipCode = Input.getString();
231:    showPrompt("住所 ");
232:    String inputAddress = Input.getString();
233:    showPrompt("電話番号 ");
234:    String inputTelephoneNumber = Input.getString();
235:    showPrompt("生まれた年 ");
236:    int inputPirthYear = getValidInt();
237:    showPrompt("入社年度 ");
238:    int inputEntranceYear = getValidInt();
239:    showPrompt("年齢 ");
240:    int inputAge = getValidInt();
241:    showPrompt("勤続年数 ");
242:    int inputServiceLength = getValidInt();
243:
244:    //配列に格納する
245:    id[recordCount] = inputId;
246:    section[recordCount] = inputSection;
247:    name[recordCount] = inputName;
248:    kana[recordCount] = inputKana;
249:    zipCode[recordCount] = inputZipCode;
250:    address[recordCount] = inputAddress;
251:    telephoneNumber[recordCount] = inputTelephoneNumber;
252:    birthYear[recordCount] = inputPirthYear;
253:    entranceYear[recordCount] = inputEntranceYear;
254:    age[recordCount] = inputAge;
255:    serviceLength[recordCount] = inputServiceLength;
256:
257:    //追加の後処理
258:    recordCount++;
259:    currentRecordIndex = recordCount - 1;
260:
```

```
261:     //画面の後始末
262:     clearMessageArea();
263:   }
264:
265:   /**
266:    * 注目行を前、あるいは次のページに移動する。移動できない時は、そのまま。
267:    */
268:   void paging(int direction) {
269:
270:     int nextRecordIndex = 0; //次のカーソル位置
271:
272:     //データが存在しないとき
273:     if (recordCount == 0) {
274:       return;
275:     }
276:
277:     //次のカーソル位置を計算する
278:     switch (direction) {
279:       case UP : //上に移動
280:         nextRecordIndex = currentRecordIndex - DISPLAY_RECORD_MAX;
281:         if (nextRecordIndex < 0) {
282:           nextRecordIndex = 0;
283:         }
284:         break;
285:       case DOWN : //下に移動
286:         nextRecordIndex = currentRecordIndex + DISPLAY_RECORD_MAX;
287:         if (nextRecordIndex >= recordCount) {
288:           nextRecordIndex = recordCount - 1;
289:         }
290:         break;
291:       default :
292:         break;
293:     }
294:
295:     //次のカーソル位置を設定する
296:     currentRecordIndex = nextRecordIndex;
297:   }
298:
299:   /**
300:    * データ 1 つ分のカーソルの移動を行う
301:    */
302:   void moveCursor(int direction) {
303:
304:     int nextRecordIndex = 0; //次のカーソル位置
305:
306:     //データが存在しないとき
307:     if (recordCount == 0) {
308:       return;
309:     }
310:
311:     //次のカーソル位置を計算する
312:     switch (direction) {
313:       case UP : //上に移動
314:         nextRecordIndex = currentRecordIndex - 1;
```

```
315:         if (nextRecordIndex < 0) {
316:           nextRecordIndex = 0;
317:         }
318:         break;
319:       case DOWN : //下に移動
320:         nextRecordIndex = currentRecordIndex + 1;
321:         if (nextRecordIndex >= recordCount) {
322:           nextRecordIndex = recordCount - 1;
323:         }
324:         break;
325:       default :
326:         break;
327:     }
328:
329:     //次のカーソル位置を設定する
330:     currentRecordIndex = nextRecordIndex;
331:   }
332:
333:   /*****************************************************
334:   /* コマンド用部品
335:   /*****************************************************/
336:
337:   /**
338:    * 数値の入力を受け取る
339:    * 入力が正しく無い場合は再入力を促す
340:    */
341:   int getValidInt() {
342:     while (true) {
343:       //入力する
344:       String input = Input.getString();
345:
346:       //入力が数字に変換可能なら入力された文字列を返す，そうでなければ再入力
347:       if (Input.isInteger(input)) {
348:         return Integer.parseInt(input);
349:       } else {
350:         showPrompt(INCORRECT_INPUT_ERROR_MESSAGE);
351:       }
352:     }
353:   }
354:
355:   /**
356:    * 指定されたコマンドが正しいコマンドかどうかを調べる
357:    */
358:   boolean isCorrectCommand(String command) {
359:     for (int i = 0; i < COMMANDS.length; i++) {
360:       if (COMMANDS[i].equals(command)) { //妥当なコマンドならば
361:         return true;
362:       }
363:     }
364:     return false;
365:   }
366:
367:   /*****************************************************
368:   /* 画面表示用部品
```

```
369:   /*************************************************/
370:
371:   /**
372:    * 画面を消去し、第 1 行目にタイトルを書いておく
373:    */
374:   void initializeTitle() {
375:     ConsoleWindow.clearScreen();
376:     ConsoleWindow.locateCursor(TITLE_LINE, TITLE_POSITION);
377:     System.out.println(COMPANY_NAME + "社員管理システム");
378:   }
379:
380:   /**
381:    * 指定されたエラーメッセージを表示する
382:    */
383:   void showError(String errorMessage) {
384:     ConsoleWindow.locateCursor(COMMAND_ERROR_LINE, 1);
385:     ConsoleWindow.clearLine();
386:     System.out.println("エラー: " + errorMessage);
387:   }
388:
389:   /**
390:    * メッセージエリアを消去する
391:    */
392:   void clearMessageArea() {
393:     ConsoleWindow.locateCursor(MESSAGE_START_LINE, 1);
394:     for (int i = 0; i < MESSAGEAREA_TERMINAL_LINE; i++) {
395:       ConsoleWindow.clearLine();
396:       System.out.print("");
397:     }
398:   }
399:
400:   /**
401:    * ページエリアを消去する
402:    */
403:   void clearPageArea() {
404:     ConsoleWindow.locateCursor(DATA_START_LINE, 1);
405:     for (int i = DATA_START_LINE; i < MESSAGE_START_LINE; i++) {
406:       ConsoleWindow.clearLine();
407:     }
408:   }
409:
410:   /**
411:    * コマンドメニューを表示する
412:    */
413:   void showCommandMenu() {
414:     ConsoleWindow.locateCursor(COMMAND_INFO_LINE, 1);
415:     ConsoleWindow.clearLine();
416:     System.out.print(COMMAND_INFO_MESSAGE);
417:   }
418:
419:   /**
420:    * コマンドプロンプトを表示する
421:    */
422:   void showCommandPrompt() {
```

```
423:     ConsoleWindow.locateCursor(COMMAND_INPUT_LINE, 1);
424:     ConsoleWindow.clearLine();
425:     System.out.print(COMMAND_PROMPT);
426:   }
427:
428:   /**
429:    * 指定されたメッセージを持つプロンプトを表示する
430:    */
431:   void showPrompt(String message) {
432:     System.out.println();
433:     System.out.print(message);
434:     System.out.print(COMMAND_PROMPT);
435:     System.out.flush();
436:   }
437:
438:   /**
439:    * 1 ページ分の社員の情報を 1 画面に書く。
440:    * 1 ページには、指定されたレコード数のデータを書く。
441:    */
442:   void showOneDirectoryPage() {
443:
444:     int start; //表示開始レコード番号
445:     int end; //  表示終了レコード番号
446:     int linePosition; //表示位置
447:
448:     //表示位置を計算する
449:     start =
450:       (((currentRecordIndex) / DISPLAY_RECORD_MAX) * DISPLAY_RECORD_MAX);
451:     end = start + DISPLAY_RECORD_MAX;
452:     if (end > recordCount) {
453:       end = recordCount;
454:     }
455:     linePosition = DATA_START_LINE;
456:
457:     //ページを初期化する
458:     clearPageArea();
459:
460:     //指定された件数分のデータを書く
461:     int count = 0;
462:     for (int i = start; i < end; i++) {
463:       showDirectoryRecord(linePosition, i);
464:       linePosition += LINES_FOR_A_RECORD;
465:       count++;
466:     }
467:     //件数が満たなければ空白で埋める
468:     while (count < DISPLAY_RECORD_MAX) {
469:       for (int i = 0; i < LINES_FOR_A_RECORD; i++) {
470:         System.out.println();
471:       }
472:       count++;
473:     }
474:   }
475:
476:   /**
```

```
477:     * 社員一人分の情報を表示する
478:     */
479:   void showDirectoryRecord(int position, int recordIndex) {
480:
481:     //前処理（移動）
482:     ConsoleWindow.locateCursor(position, 1);
483:
484:     //注目行マークをつける
485:     setMark(recordIndex);
486:
487:     //データを書き込む
488:     System.out.println(
489:       "\t"
490:         + (recordIndex + 1)
491:         + "\t"
492:         + id[recordIndex]
493:         + "\t"
494:         + section[recordIndex]
495:         + "\t"
496:         + name[recordIndex]
497:         + "\t"
498:         + kana[recordIndex]);
499:     System.out.println(
500:       "\t"
501:         + "\t"
502:         + "\t"
503:         + address[recordIndex]
504:         + "\t"
505:         + telephoneNumber[recordIndex]
506:         + "\t"
507:         + "〒"
508:         + zipCode[recordIndex]);
509:     System.out.println(
510:       "\t"
511:         + "\t"
512:         + birthYear[recordIndex]
513:         + "年生まれ \t"
514:         + age[recordIndex]
515:         + "才 \t"
516:         + entranceYear[recordIndex]
517:         + "年入社 \t 勤続"
518:         + serviceLength[recordIndex]
519:         + "年");
520:
521:     //後処理（境界線）
522:     System.out.println(LINE);
523:   }
524:
525:   /**
526:     * 注目行マークをつける
527:     */
528:   void setMark(int recordIndex) {
529:     if (recordIndex == currentRecordIndex) {
530:       System.out.print("*");
```

```
531:     } else {
532:       System.out.print("  ");
533:     }
534:   }
535:
536: }
```

A.1.5.2 ファイル機能付き

リスト 111: 社員住所録管理アプリケーション (ファイル機能付き)

```
 1: import java.io.PrintStream;
 2:
 3: /**
 4:  * 社員名簿管理アプリケーション（ファイル機能付き）
 5:  *
 6:  * 本アプリケーションの機能
 7:  * ○名簿の管理
 8:  *      ・社員の追加
 9:  * ○名簿の閲覧
10:  *      ・ページの移動
11:  *      ・カーソルの移動
12:  * ○名簿の保存
13:  *      ・セーブ
14:  *      ・ロード
15:  *
16:  * @author Manabu Sugiura
17:  * @version $Id: EmployeeDirectoryApplication.java,v 1.2 2003/05/04 10:19:33 macchan Exp $
18:  */
19: public class EmployeeDirectoryApplication {
20:
21:   public static void main(String[] args) {
22:     EmployeeDirectoryApplication employeeDirectoryApplication =
23:       new EmployeeDirectoryApplication();
24:     employeeDirectoryApplication.main();
25:   }
26:
27:   /*****************************************************
28:   /* 定数
29:   /*****************************************************/
30:
31:   //データ関連の定数
32:   final int RECORD_MAX = 6; //保存できる最大データの数
33:   final int FIELD_COUNT = 9; //一件当たりのデータフィールドの数
34:
35:   //画面制御用の定数
36:   final int DISPLAY_RECORD_MAX = 5; //1 ページに書くデータ件数
37:   final int TITLE_LINE = 1; //タイトル行の位置
38:   final int TITLE_POSITION = 25; //タイトルの位置
39:   final int DATA_START_LINE = 3; //データ行の先頭位置
40:   final int LINES_FOR_A_RECORD = 4; //1 レコードの表示に必要な行数
41:   //  メッセージ行の先頭位置
42:   final int MESSAGE_START_LINE =
43:     DATA_START_LINE + LINES_FOR_A_RECORD * DISPLAY_RECORD_MAX;
44:   //コマンドエラーメッセージの行の位置
45:   final int COMMAND_ERROR_LINE = MESSAGE_START_LINE;
46:   //  コマンドメッセージの行の位置
47:   final int COMMAND_INFO_LINE = MESSAGE_START_LINE + 1;
```

```
48:   //コマンドプロンプトの行の位置
49:   final int COMMAND_INPUT_LINE = COMMAND_INFO_LINE + 1;
50:   //  追加メッセージ行の位置
51:   final int APPEND_LINE = MESSAGE_START_LINE;
52:   // メッセージフィールドの最終行の位置
53:   final int MESSAGEAREA_TERMINAL_LINE = MESSAGE_START_LINE + 20;
54:
55:   //コマンド類
56:   final String ADD = "A"; //データの追加
57:   final String PREVIOUS_PAGE = "P"; //前ページへ
58:   final String NEXT_PAGE = "N"; //次ページへ
59:   final String UP_CURSOR = "U"; //カーソルを上へ
60:   final String DOWN_CURSOR = "D"; //カーソルを下へ
61:   final String SAVE = "S"; //セーブ
62:   final String LOAD = "L"; //ロード
63:   final String QUIT = "Q"; //終了
64:   final String[] COMMANDS =
65:     {
66:       ADD,
67:       PREVIOUS_PAGE,
68:       NEXT_PAGE,
69:       UP_CURSOR,
70:       DOWN_CURSOR,
71:       SAVE,
72:       LOAD,
73:       QUIT };
74:
75:   //表示文字列類
76:   final String LINE =
77:     "------------------------------------------------------------------------"
78:       + "------------------------------------------------------------------------"
79:       + "------------------------------------------------------------------------";
80:   final String SPACE = "  ";
81:   final String SC = ":";
82:   final String ADD_MSG = ADD + SC + "追加" + SPACE;
83:   final String PP_MSG = PREVIOUS_PAGE + SC + "前ページへ" + SPACE;
84:   final String NP_MSG = NEXT_PAGE + SC + "次ページへ" + SPACE;
85:   final String UC_MSG = UP_CURSOR + SC + "前のデータへ" + SPACE;
86:   final String DC_MSG = DOWN_CURSOR + SC + "次のデータへ" + SPACE;
87:   final String SAVE_MSG = SAVE + SC + "セーブ" + SPACE;
88:   final String LOAD_MSG = LOAD + SC + "ロード" + SPACE;
89:   final String QUIT_MSG = QUIT + SC + "終了" + SPACE;
90:   final String COMMAND_INFO_MESSAGE =
91:     "コマンド"
92:       + "("
93:       + SPACE
94:       + ADD_MSG
95:       + PP_MSG
96:       + NP_MSG
97:       + UC_MSG
98:       + DC_MSG
99:       + SAVE_MSG
100:      + LOAD_MSG
101:      + QUIT_MSG
```

```
102:       + ")";
103:   final String COMMAND_PROMPT = " >> ";
104:   final String COMPANY_NAME = "○×コミュニケーションズ";
105:   final String REMOVE_CONFIRM_MESSAGE = "本当に削除してもいいですか？ y/n";
106:   final String YES = "Y";
107:
108:   final String COMMAND_ERROR_MESSAGE = "不適当なコマンドでした";
109:   final String INCORRECT_INPUT_ERROR_MESSAGE = "不正な入力です　数字を入力してくださ
い";
110:   final String DATA_OVERFLOW_ERROR_MESSAGE = "データがいっぱいで、もう追加できません";
111:   final String NO_DATA_UNDELETALBLE_ERROR_MESSAGE = "削除可能なデータがありません";
112:   final String NO_DATA_UNEDITABLE_ERROR_MESSAGE = "編集可能なデータがありません";
113:
114:   //カーソル移動用
115:   final int UP = 1;
116:   final int DOWN = 2;
117:
118:   //画面の大きさ用
119:   final int X = 100;
120:   final int Y = 100;
121:   final int WIDTH = 850;
122:   final int HEIGHT = 600;
123:
124:   /*****************************************************
125:   /* インスタンス変数
126:   /*****************************************************/
127:
128:   int currentRecordIndex = -1; //現在カーソルがあるインデックス。何もないときは-1
129:   int recordCount; //現在の要素数
130:
131:   //社員の情報を保存しておくための配列群
132:   int[] id = new int[RECORD_MAX]; //社員 ID
133:   String[] section = new String[RECORD_MAX]; //部署
134:   String[] name = new String[RECORD_MAX]; //漢字の名前
135:   String[] kana = new String[RECORD_MAX]; //名前のふりがな
136:   String[] zipCode = new String[RECORD_MAX]; //郵便番号
137:   String[] address = new String[RECORD_MAX]; //住所
138:   String[] telephoneNumber = new String[RECORD_MAX]; //電話番号
139:   int[] birthYear = new int[RECORD_MAX]; //生まれた年
140:   int[] entranceYear = new int[RECORD_MAX]; //入社年度
141:   int[] age = new int[RECORD_MAX]; //年齢
142:   int[] serviceLength = new int[RECORD_MAX]; //勤続年数
143:
144:   /*****************************************************
145:   /* メイン
146:   /*****************************************************/
147:
148:   void main() {
149:
150:     String command; //入力されたコマンド
151:
152:     //初期化
153:     ConsoleWindow.initializeWindow(X, Y, WIDTH, HEIGHT);
154:     initializeTitle();
```

```
155:
156:     //コマンドの入力と実行
157:     while (!((command = inputCommand()).equals(QUIT))) {
158:       executeCommand(command);
159:       showOneDirectoryPage();
160:     }
161:
162:     //終了処理
163:     ConsoleWindow.clearScreen();
164:     System.exit(0);
165:   }
166:
167:   /*****************************************************
168:   /* コマンド処理
169:   /*****************************************************/
170:
171:   /**
172:    * 社員名簿のコマンドを入力し、妥当なコマンドであれば、入力されたコマンドを返す。
173:    * もしも、不適当なコマンドであれば、再入力を要求する。
174:    */
175:   String inputCommand() {
176:
177:     String command = null; //入力されたコマンド
178:
179:     //メニューを出す
180:     showCommandMenu();
181:
182:     while (true) {
183:
184:       //プロンプトを出す
185:       showCommandPrompt();
186:
187:       //コマンドを入力する
188:       command = Input.getString();
189:       command = command.toUpperCase(); //入力されたコマンドを大文字に
190:
191:       //正しいコマンドならば終了して入力コマンドを返す。
192:       //正しくなければエラーを表示して再入力
193:       if (isCorrectCommand(command)) {
194:         break;
195:       } else {
196:         showError(COMMAND_ERROR_MESSAGE);
197:       }
198:     }
199:
200:     //画面の後始末
201:     clearMessageArea();
202:
203:     return command;
204:   }
205:
206:   /**
207:    * 指定された社員名簿のコマンドを実行する
208:    */
```

```
209:   void executeCommand(String command) {
210:
211:     if (command.equals(ADD)) { //追加
212:       addEmployee();
213:     } else if (command.equals(PREVIOUS_PAGE)) { //前ページ
214:       paging(UP);
215:     } else if (command.equals(NEXT_PAGE)) { //次ページ
216:       paging(DOWN);
217:     } else if (command.equals(UP_CURSOR)) { //前のデータ
218:       moveCursor(UP);
219:     } else if (command.equals(DOWN_CURSOR)) { //次のデータ
220:       moveCursor(DOWN);
221:     } else if (command.equals(SAVE)) { //セーブ
222:       save();
223:     } else if (command.equals(LOAD)) { //ロード
224:       load();
225:     }
226:
227:   }
228:
229:   /****************************************************
230:   /* 名簿管理コマンド群
231:   ****************************************************/
232:
233:   /**
234:    * 社員名簿データに、新しい社員を追加する
235:    */
236:   void addEmployee() {
237:
238:     //データが満杯だったらエラーを表示して終了する
239:     if (recordCount >= RECORD_MAX) {
240:       showError(DATA_OVERFLOW_ERROR_MESSAGE);
241:       return;
242:     }
243:
244:     //データを入力する
245:     showPrompt("社員 ID");
246:     int inputId = getValidInt();
247:     showPrompt("所属部署 ");
248:     String inputSection = Input.getString();
249:     showPrompt("名前 ");
250:     String inputName = Input.getString();
251:     showPrompt("ふりがな ");
252:     String inputKana = Input.getString();
253:     showPrompt("郵便番号 ");
254:     String inputZipCode = Input.getString();
255:     showPrompt("住所 ");
256:     String inputAddress = Input.getString();
257:     showPrompt("電話番号 ");
258:     String inputTelephoneNumber = Input.getString();
259:     showPrompt("生まれた年 ");
260:     int inputPirthYear = getValidInt();
261:     showPrompt("入社年度 ");
262:     int inputEntranceYear = getValidInt();
```

```
263:     showPrompt("年齢 ");
264:     int inputAge = getValidInt();
265:     showPrompt("勤続年数 ");
266:     int inputServiceLength = getValidInt();
267:
268:     //配列に格納する
269:     id[recordCount] = inputId;
270:     section[recordCount] = inputSection;
271:     name[recordCount] = inputName;
272:     kana[recordCount] = inputKana;
273:     zipCode[recordCount] = inputZipCode;
274:     address[recordCount] = inputAddress;
275:     telephoneNumber[recordCount] = inputTelephoneNumber;
276:     birthYear[recordCount] = inputPirthYear;
277:     entranceYear[recordCount] = inputEntranceYear;
278:     age[recordCount] = inputAge;
279:     serviceLength[recordCount] = inputServiceLength;
280:
281:     //追加の後処理
282:     recordCount++;
283:     currentRecordIndex = recordCount - 1;
284:
285:     //画面の後始末
286:     clearMessageArea();
287:   }
288:
289:   /**
290:    * 注目行を前、あるいは次のページに移動する。移動できない時は、そのまま。
291:    */
292:   void paging(int direction) {
293:
294:     int nextRecordIndex = 0; //次のカーソル位置
295:
296:     //データが存在しないとき
297:     if (recordCount == 0) {
298:       return;
299:     }
300:
301:     //次のカーソル位置を計算する
302:     switch (direction) {
303:       case UP : //上に移動
304:         nextRecordIndex = currentRecordIndex - DISPLAY_RECORD_MAX;
305:         if (nextRecordIndex < 0) {
306:           nextRecordIndex = 0;
307:         }
308:         break;
309:       case DOWN : //下に移動
310:         nextRecordIndex = currentRecordIndex + DISPLAY_RECORD_MAX;
311:         if (nextRecordIndex >= recordCount) {
312:           nextRecordIndex = recordCount - 1;
313:         }
314:         break;
315:       default :
316:         break;
```

```
317:     }
318:
319:     //次のカーソル位置を設定する
320:     currentRecordIndex = nextRecordIndex;
321:   }
322:
323:   /**
324:    * データ 1 つ分のカーソルの移動を行う
325:    */
326:   void moveCursor(int direction) {
327:
328:     int nextRecordIndex = 0; //次のカーソル位置
329:
330:     //データが存在しないとき
331:     if (recordCount == 0) {
332:       return;
333:     }
334:
335:     //次のカーソル位置を計算する
336:     switch (direction) {
337:       case UP : //上に移動
338:         nextRecordIndex = currentRecordIndex - 1;
339:         if (nextRecordIndex < 0) {
340:           nextRecordIndex = 0;
341:         }
342:         break;
343:       case DOWN : //下に移動
344:         nextRecordIndex = currentRecordIndex + 1;
345:         if (nextRecordIndex >= recordCount) {
346:           nextRecordIndex = recordCount - 1;
347:         }
348:         break;
349:       default :
350:         break;
351:     }
352:
353:     //次のカーソル位置を設定する
354:     currentRecordIndex = nextRecordIndex;
355:   }
356:
357:   /**
358:    * 名簿データのセーブを行う.
359:    */
360:   void save() {
361:
362:     //ファイル名を入力する
363:     showPrompt("セーブするファイル名を入力してください"); //プロンプト
364:     String fileName = Input.getString();
365:
366:     //前処理
367:     PrintStream writer = FileIO.openForWrite(fileName); //ファイルオープン
368:
369:     //書き込み
370:     for (int i = 0; i < recordCount; i++) {
```

```
371:       //CSV 形式
372:       writer.println(
373:         id[i]
374:           + ","
375:           + section[i]
376:           + ","
377:           + name[i]
378:           + ","
379:           + kana[i]
380:           + ","
381:           + zipCode[i]
382:           + ","
383:           + address[i]
384:           + ","
385:           + telephoneNumber[i]
386:           + ","
387:           + birthYear[i]
388:           + ","
389:           + entranceYear[i]
390:           + ","
391:           + age[i]
392:           + ","
393:           + serviceLength[i]);
394:     }
395:
396:     //後処理
397:     writer.close(); //ファイルクローズ
398:
399:   }
400:
401:   /**
402:    * 名簿データのロードを行う.
403:    */
404:   void load() {
405:
406:     //ファイル名を入力する
407:     showPrompt("ロードするファイル名を入力してください");
408:     String fileName = Input.getString();
409:
410:     //前処理
411:     ReadStream reader = FileIO.openForRead(fileName); //ファイルオープン
412:
413:     //読み込み
414:     for (recordCount = 0; !reader.isEnd(); recordCount++) {
415:       //CSV 形式のデータを分割する
416:       String line = reader.readLine();
417:       String[] elements = line.split(",");
418:
419:       //データを追加する
420:       id[recordCount] = Integer.parseInt(elements[0]);
421:       section[recordCount] = elements[1];
422:       name[recordCount] = elements[2];
423:       kana[recordCount] = elements[3];
424:       zipCode[recordCount] = elements[4];
```

```
425:       address[recordCount] = elements[5];
426:       telephoneNumber[recordCount] = elements[6];
427:       birthYear[recordCount] = Integer.parseInt(elements[7]);
428:       entranceYear[recordCount] = Integer.parseInt(elements[8]);
429:       age[recordCount] = Integer.parseInt(elements[9]);
430:       serviceLength[recordCount] = Integer.parseInt(elements[10]);
431:     }
432:
433:     //後処理
434:     reader.close(); //ファイルクローズ
435:     currentRecordIndex = 0; //カーソルの初期化
436:   }
437:
438:   /*****************************************************
439:   /* コマンド用部品
440:   /*****************************************************/
441:
442:   /**
443:    * 数値の入力を受け取る
444:    * 入力が正しく無い場合は再入力を促す
445:    */
446:   int getValidInt() {
447:     while (true) {
448:       //入力する
449:       String input = Input.getString();
450:
451:       //入力が数字に変換可能なら入力された文字列を返す，そうでなければ再入力
452:       if (Input.isInteger(input)) {
453:         return Integer.parseInt(input);
454:       } else {
455:         showPrompt(INCORRECT_INPUT_ERROR_MESSAGE);
456:       }
457:     }
458:   }
459:
460:   /**
461:    * 指定されたコマンドが正しいコマンドかどうかを調べる
462:    */
463:   boolean isCorrectCommand(String command) {
464:     for (int i = 0; i < COMMANDS.length; i++) {
465:       if (COMMANDS[i].equals(command)) { //妥当なコマンドならば
466:         return true;
467:       }
468:     }
469:     return false;
470:   }
471:
472:   /*****************************************************
473:   /* 画面表示用部品
474:   /*****************************************************/
475:
476:   /**
477:    * 画面を消去し、第 1 行目にタイトルを書いておく
478:    */
```

```
479:   void initializeTitle() {
480:     ConsoleWindow.clearScreen();
481:     ConsoleWindow.locateCursor(TITLE_LINE, TITLE_POSITION);
482:     System.out.println(COMPANY_NAME + "社員管理システム");
483:   }
484:
485:   /**
486:    * 指定されたエラーメッセージを表示する
487:    */
488:   void showError(String errorMessage) {
489:     ConsoleWindow.locateCursor(COMMAND_ERROR_LINE, 1);
490:     ConsoleWindow.clearLine();
491:     System.out.println("エラー: " + errorMessage);
492:   }
493:
494:   /**
495:    * メッセージエリアを消去する
496:    */
497:   void clearMessageArea() {
498:     ConsoleWindow.locateCursor(MESSAGE_START_LINE, 1);
499:     for (int i = 0; i < MESSAGEAREA_TERMINAL_LINE; i++) {
500:       ConsoleWindow.clearLine();
501:       System.out.print("");
502:     }
503:   }
504:
505:   /**
506:    * ページエリアを消去する
507:    */
508:   void clearPageArea() {
509:     ConsoleWindow.locateCursor(DATA_START_LINE, 1);
510:     for (int i = DATA_START_LINE; i < MESSAGE_START_LINE; i++) {
511:       ConsoleWindow.clearLine();
512:     }
513:   }
514:
515:   /**
516:    * コマンドメニューを表示する
517:    */
518:   void showCommandMenu() {
519:     ConsoleWindow.locateCursor(COMMAND_INFO_LINE, 1);
520:     ConsoleWindow.clearLine();
521:     System.out.print(COMMAND_INFO_MESSAGE);
522:   }
523:
524:   /**
525:    * コマンドプロンプトを表示する
526:    */
527:   void showCommandPrompt() {
528:     ConsoleWindow.locateCursor(COMMAND_INPUT_LINE, 1);
529:     ConsoleWindow.clearLine();
530:     System.out.print(COMMAND_PROMPT);
531:   }
532:
```

```
533:   /**
534:    * 指定されたメッセージを持つプロンプトを表示する
535:    */
536:   void showPrompt(String message) {
537:     System.out.println();
538:     System.out.print(message);
539:     System.out.print(COMMAND_PROMPT);
540:     System.out.flush();
541:   }
542:
543:   /**
544:    * 1 ページ分の社員の情報を 1 画面に書く。
545:    * 1 ページには、指定されたレコード数のデータを書く。
546:    */
547:   void showOneDirectoryPage() {
548:
549:     int start; //表示開始レコード番号
550:     int end; //  表示終了レコード番号
551:     int linePosition; //表示位置
552:
553:     //表示位置を計算する
554:     start =
555:       (((currentRecordIndex) / DISPLAY_RECORD_MAX) * DISPLAY_RECORD_MAX);
556:     end = start + DISPLAY_RECORD_MAX;
557:     if (end > recordCount) {
558:       end = recordCount;
559:     }
560:     linePosition = DATA_START_LINE;
561:
562:     //ページを初期化する
563:     clearPageArea();
564:
565:     //指定された件数分のデータを書く
566:     int count = 0;
567:     for (int i = start; i < end; i++) {
568:       showDirectoryRecord(linePosition, i);
569:       linePosition += LINES_FOR_A_RECORD;
570:       count++;
571:     }
572:     //件数が満たなければ空白で埋める
573:     while (count < DISPLAY_RECORD_MAX) {
574:       for (int i = 0; i < LINES_FOR_A_RECORD; i++) {
575:         System.out.println();
576:       }
577:       count++;
578:     }
579:   }
580:
581:   /**
582:    * 社員一人分の情報を表示する
583:    */
584:   void showDirectoryRecord(int position, int recordIndex) {
585:
586:     //前処理（移動）
```

```
587:     ConsoleWindow.locateCursor(position, 1);
588:
589:     //注目行マークをつける
590:     setMark(recordIndex);
591:
592:     //データを書き込む
593:     System.out.println(
594:       "\t"
595:         + (recordIndex + 1)
596:         + "\t"
597:         + id[recordIndex]
598:         + "\t"
599:         + section[recordIndex]
600:         + "\t"
601:         + name[recordIndex]
602:         + "\t"
603:         + kana[recordIndex]);
604:     System.out.println(
605:       "\t"
606:         + "\t"
607:         + "\t"
608:         + address[recordIndex]
609:         + "\t"
610:         + telephoneNumber[recordIndex]
611:         + "\t"
612:         + "〒"
613:         + zipCode[recordIndex]);
614:     System.out.println(
615:       "\t"
616:         + "\t"
617:         + birthYear[recordIndex]
618:         + "年生まれ \t"
619:         + age[recordIndex]
620:         + "才 \t"
621:         + entranceYear[recordIndex]
622:         + "年入社 \t 勤続"
623:         + serviceLength[recordIndex]
624:         + "年");
625:
626:     //後処理（境界線）
627:     System.out.println(LINE);
628:   }
629:
630:   /**
631:    * 注目行マークをつける
632:    */
633:   void setMark(int recordIndex) {
634:     if (recordIndex == currentRecordIndex) {
635:       System.out.print("*");
636:     } else {
637:       System.out.print("  ");
638:     }
639:   }
640:
```

```
641: }
```

A.1.5.3 編集機能付き

リスト 112: 社員住所録管理アプリケーション (編集機能付き)

```
 1: import java.io.PrintStream;
 2:
 3: /**
 4:  * 社員名簿管理アプリケーション（編集機能付き）
 5:  *
 6:  * 本アプリケーションの機能
 7:  * ○名簿の管理
 8:  *      ・社員の追加
 9:  *      ・社員の削除
10:  *      ・社員の編集
11:  * ○名簿の閲覧
12:  *      ・ページの移動
13:  *      ・カーソルの移動
14:  * ○名簿の保存
15:  *      ・セーブ
16:  *      ・ロード
17:  *
18:  * @author Manabu Sugiura
19:  * @version $Id: EmployeeDirectoryApplication.java,v 1.2 2003/05/04 10:19:33 macchan Exp $
20:  */
21: public class EmployeeDirectoryApplication {
22:
23:   public static void main(String[] args) {
24:     EmployeeDirectoryApplication employeeDirectoryApplication =
25:       new EmployeeDirectoryApplication();
26:     employeeDirectoryApplication.main();
27:   }
28:
29:   /*****************************************************
30:   /* 定数
31:   /*****************************************************/
32:
33:   //データ関連の定数
34:   final int RECORD_MAX = 6; //保存できる最大データの数
35:   final int FIELD_COUNT = 9; //一件当たりのデータフィールドの数
36:
37:   //画面制御用の定数
38:   final int DISPLAY_RECORD_MAX = 5; //1 ページに書くデータ件数
39:   final int TITLE_LINE = 1; //タイトル行の位置
40:   final int TITLE_POSITION = 25; //タイトルの位置
41:   final int DATA_START_LINE = 3; //データ行の先頭位置
42:   final int LINES_FOR_A_RECORD = 4; //1 レコードの表示に必要な行数
43:   //  メッセージ行の先頭位置
44:   final int MESSAGE_START_LINE =
45:     DATA_START_LINE + LINES_FOR_A_RECORD * DISPLAY_RECORD_MAX;
46:   //コマンドエラーメッセージの行の位置
47:   final int COMMAND_ERROR_LINE = MESSAGE_START_LINE;
```

```
48:   //  コマンドメッセージの行の位置
49:   final int COMMAND_INFO_LINE = MESSAGE_START_LINE + 1;
50:   //コマンドプロンプトの行の位置
51:   final int COMMAND_INPUT_LINE = COMMAND_INFO_LINE + 1;
52:   //  追加メッセージ行の位置
53:   final int APPEND_LINE = MESSAGE_START_LINE;
54:   // メッセージフィールドの最終行の位置
55:   final int MESSAGEAREA_TERMINAL_LINE = MESSAGE_START_LINE + 20;
56:
57:   //コマンド類
58:   final String ADD = "A"; //データの追加
59:   final String REMOVE = "R"; //データの削除
60:   final String EDIT = "E"; //データの編集
61:   final String PREVIOUS_PAGE = "P"; //前ページへ
62:   final String NEXT_PAGE = "N"; //次ページへ
63:   final String UP_CURSOR = "U"; //カーソルを上へ
64:   final String DOWN_CURSOR = "D"; //カーソルを下へ
65:   final String SAVE = "S"; //セーブ
66:   final String LOAD = "L"; //ロード
67:   final String QUIT = "Q"; //終了
68:   final String[] COMMANDS =
69:     {
70:       ADD,
71:       REMOVE,
72:       EDIT,
73:       PREVIOUS_PAGE,
74:       NEXT_PAGE,
75:       UP_CURSOR,
76:       DOWN_CURSOR,
77:       SAVE,
78:       LOAD,
79:       QUIT };
80:
81:   //表示文字列類
82:   final String LINE =
83:     "---------------------------------------------------------------------"
84:       + "---------------------------------------------------------------------"
85:       + "---------------------------------------------------------------------";
86:   final String SPACE = "  ";
87:   final String SC = ":";
88:   final String ADD_MSG = ADD + SC + "追加" + SPACE;
89:   final String RM_MSG = REMOVE + SC + "削除" + SPACE;
90:   final String EDIT_MSG = EDIT + SC + "編集" + SPACE;
91:   final String PP_MSG = PREVIOUS_PAGE + SC + "前ページへ" + SPACE;
92:   final String NP_MSG = NEXT_PAGE + SC + "次ページへ" + SPACE;
93:   final String UC_MSG = UP_CURSOR + SC + "前のデータへ" + SPACE;
94:   final String DC_MSG = DOWN_CURSOR + SC + "次のデータへ" + SPACE;
95:   final String SAVE_MSG = SAVE + SC + "セーブ" + SPACE;
96:   final String LOAD_MSG = LOAD + SC + "ロード" + SPACE;
97:   final String QUIT_MSG = QUIT + SC + "終了" + SPACE;
98:   final String COMMAND_INFO_MESSAGE =
99:     "コマンド"
100:      + "("
101:      + SPACE
```

```
102:        + ADD_MSG
103:        + RM_MSG
104:        + EDIT_MSG
105:        + PP_MSG
106:        + NP_MSG
107:        + UC_MSG
108:        + DC_MSG
109:        + SAVE_MSG
110:        + LOAD_MSG
111:        + QUIT_MSG
112:        + ")";
113:    final String COMMAND_PROMPT = " >> ";
114:    final String COMPANY_NAME = "○×コミュニケーションズ";
115:    final String REMOVE_CONFIRM_MESSAGE = "本当に削除してもいいですか？ y/n";
116:    final String YES = "Y";
117:
118:    final String COMMAND_ERROR_MESSAGE = "不適当なコマンドでした";
119:    final String INCORRECT_INPUT_ERROR_MESSAGE = "不正な入力です　数字を入力してくださ
い";
120:    final String DATA_OVERFLOW_ERROR_MESSAGE = "データがいっぱいで、もう追加できません";
121:    final String NO_DATA_UNDELETALBLE_ERROR_MESSAGE = "削除可能なデータがありません";
122:    final String NO_DATA_UNEDITABLE_ERROR_MESSAGE = "編集可能なデータがありません";
123:
124:    //カーソル移動用
125:    final int UP = 1;
126:    final int DOWN = 2;
127:
128:    //画面の大きさ用
129:    final int X = 100;
130:    final int Y = 100;
131:    final int WIDTH = 850;
132:    final int HEIGHT = 600;
133:
134:    /*****************************************************
135:    /* インスタンス変数
136:    /*****************************************************/
137:
138:    int currentRecordIndex = -1; //現在カーソルがあるインデックス。何もないときは-1
139:    int recordCount; //現在の要素数
140:
141:    //社員の情報を保存しておくための配列群
142:    int[] id = new int[RECORD_MAX]; //社員 ID
143:    String[] section = new String[RECORD_MAX]; //部署
144:    String[] name = new String[RECORD_MAX]; //漢字の名前
145:    String[] kana = new String[RECORD_MAX]; //名前のふりがな
146:    String[] zipCode = new String[RECORD_MAX]; //郵便番号
147:    String[] address = new String[RECORD_MAX]; //住所
148:    String[] telephoneNumber = new String[RECORD_MAX]; //電話番号
149:    int[] birthYear = new int[RECORD_MAX]; //生まれた年
150:    int[] entranceYear = new int[RECORD_MAX]; //入社年度
151:    int[] age = new int[RECORD_MAX]; //年齢
152:    int[] serviceLength = new int[RECORD_MAX]; //勤続年数
153:
154:    /*****************************************************
```

```
155:   /* メイン
156:   /*****************************************************/
157:
158:   void main() {
159:
160:     String command; //入力されたコマンド
161:
162:     //初期化
163:     ConsoleWindow.initializeWindow(X, Y, WIDTH, HEIGHT);
164:     initializeTitle();
165:
166:     //コマンドの入力と実行
167:     while (!((command = inputCommand()).equals(QUIT))) {
168:       executeCommand(command);
169:       showOneDirectoryPage();
170:     }
171:
172:     //終了処理
173:     ConsoleWindow.clearScreen();
174:     System.exit(0);
175:   }
176:
177:   /*****************************************************
178:   /* コマンド処理
179:   /*****************************************************/
180:
181:   /**
182:    * 社員名簿のコマンドを入力し、妥当なコマンドであれば、入力されたコマンドを返す。
183:    * もしも、不適当なコマンドであれば、再入力を要求する。
184:    */
185:   String inputCommand() {
186:
187:     String command = null; //入力されたコマンド
188:
189:     //メニューを出す
190:     showCommandMenu();
191:
192:     while (true) {
193:
194:       //プロンプトを出す
195:       showCommandPrompt();
196:
197:       //コマンドを入力する
198:       command = Input.getString();
199:       command = command.toUpperCase(); //入力されたコマンドを大文字に
200:
201:       //正しいコマンドならば終了して入力コマンドを返す。
202:       //正しくなければエラーを表示して再入力
203:       if (isCorrectCommand(command)) {
204:         break;
205:       } else {
206:         showError(COMMAND_ERROR_MESSAGE);
207:       }
208:     }
```

```
209:
210:     //画面の後始末
211:     clearMessageArea();
212:
213:     return command;
214:   }
215:
216:   /**
217:    * 指定された社員名簿のコマンドを実行する
218:    */
219:   void executeCommand(String command) {
220:
221:     if (command.equals(ADD)) { //追加
222:       addEmployee();
223:     } else if (command.equals(REMOVE)) { //削除
224:       removeEmployee();
225:     } else if (command.equals(EDIT)) { //編集
226:       editEmployee();
227:     } else if (command.equals(PREVIOUS_PAGE)) { //前ページ
228:       paging(UP);
229:     } else if (command.equals(NEXT_PAGE)) { //次ページ
230:       paging(DOWN);
231:     } else if (command.equals(UP_CURSOR)) { //前のデータ
232:       moveCursor(UP);
233:     } else if (command.equals(DOWN_CURSOR)) { //次のデータ
234:       moveCursor(DOWN);
235:     } else if (command.equals(SAVE)) { //セーブ
236:       save();
237:     } else if (command.equals(LOAD)) { //ロード
238:       load();
239:     }
240:
241:   }
242:
243:   /*****************************************************
244:   /* 名簿管理コマンド群
245:   *****************************************************/
246:
247:   /**
248:    * 社員名簿データに、新しい社員を追加する
249:    */
250:   void addEmployee() {
251:
252:     //データが満杯だったらエラーを表示して終了する
253:     if (recordCount >= RECORD_MAX) {
254:       showError(DATA_OVERFLOW_ERROR_MESSAGE);
255:       return;
256:     }
257:
258:     //データを入力する
259:     showPrompt("社員 ID");
260:     int inputId = getValidInt();
261:     showPrompt("所属部署 ");
262:     String inputSection = Input.getString();
```

```
263:     showPrompt("名前 ");
264:     String inputName = Input.getString();
265:     showPrompt("ふりがな ");
266:     String inputKana = Input.getString();
267:     showPrompt("郵便番号 ");
268:     String inputZipCode = Input.getString();
269:     showPrompt("住所 ");
270:     String inputAddress = Input.getString();
271:     showPrompt("電話番号 ");
272:     String inputTelephoneNumber = Input.getString();
273:     showPrompt("生まれた年 ");
274:     int inputPirthYear = getValidInt();
275:     showPrompt("入社年度 ");
276:     int inputEntranceYear = getValidInt();
277:     showPrompt("年齢 ");
278:     int inputAge = getValidInt();
279:     showPrompt("勤続年数 ");
280:     int inputServiceLength = getValidInt();
281:
282:     //配列に格納する
283:     id[recordCount] = inputId;
284:     section[recordCount] = inputSection;
285:     name[recordCount] = inputName;
286:     kana[recordCount] = inputKana;
287:     zipCode[recordCount] = inputZipCode;
288:     address[recordCount] = inputAddress;
289:     telephoneNumber[recordCount] = inputTelephoneNumber;
290:     birthYear[recordCount] = inputPirthYear;
291:     entranceYear[recordCount] = inputEntranceYear;
292:     age[recordCount] = inputAge;
293:     serviceLength[recordCount] = inputServiceLength;
294:
295:     //追加の後処理
296:     recordCount++;
297:     currentRecordIndex = recordCount - 1;
298:
299:     //画面の後始末
300:     clearMessageArea();
301:   }
302:
303:   /**
304:    * 名簿データの削除を行う
305:    */
306:   void removeEmployee() {
307:
308:     //社員が存在しないとき
309:     if (recordCount == 0) {
310:       showError(NO_DATA_UNDELETALBLE_ERROR_MESSAGE);
311:       return;
312:     }
313:
314:     //削除確認をする
315:     System.out.print(REMOVE_CONFIRM_MESSAGE + COMMAND_PROMPT);
316:     String input = Input.getString();
```

```
317:
318:     //削除確認がとれなかったときはなにもしない
319:     if (!input.toUpperCase().equals(YES)) {
320:       return;
321:     }
322:
323:     //削除する（要素をつめる）
324:     for (int i = currentRecordIndex; i < recordCount - 1; i++) {
325:       id[i] = id[i + 1];
326:       section[i] = section[i + 1];
327:       name[i] = name[i + 1];
328:       kana[i] = kana[i + 1];
329:       zipCode[i] = zipCode[i + 1];
330:       address[i] = address[i + 1];
331:       telephoneNumber[i] = telephoneNumber[i + 1];
332:       birthYear[i] = birthYear[i + 1];
333:       entranceYear[i] = entranceYear[i + 1];
334:       age[i] = age[i];
335:       serviceLength[i] = serviceLength[i + 1];
336:     }
337:
338:     //削除の後始末
339:     recordCount--;
340:     if (recordCount == currentRecordIndex) { //最後の要素が削除された場合
341:       currentRecordIndex--;
342:     }
343:     clearMessageArea();
344:   }
345:
346:   /**
347:    * 社員の編集を行う
348:    */
349:   void editEmployee() {
350:
351:     //社員が存在しないとき
352:     if (recordCount == 0) {
353:       showError(NO_DATA_UNEDITABLE_ERROR_MESSAGE);
354:       return;
355:     }
356:
357:     //変更入力する
358:
359:     showPrompt("所属部署 " + section[currentRecordIndex]);
360:     String inputSection = Input.getString();
361:     if (inputSection.equals("")) { //空白なら変更しないとする
362:       inputSection = section[currentRecordIndex];
363:     }
364:
365:     showPrompt("名前 " + name[currentRecordIndex]);
366:     String inputName = Input.getString();
367:     if (inputName.equals("")) { //空白なら変更しないとする
368:       inputName = name[currentRecordIndex];
369:     }
370:
```

```
371:     showPrompt("ふりがな " + kana[currentRecordIndex]);
372:     String inputKana = Input.getString();
373:     if (inputKana.equals("")) { //空白なら変更しないとする
374:       inputKana = kana[currentRecordIndex];
375:     }
376:
377:     showPrompt("郵便番号 " + zipCode[currentRecordIndex]);
378:     String inputZipCode = Input.getString();
379:     if (inputZipCode.equals("")) { //空白なら変更しないとする
380:       inputZipCode = zipCode[currentRecordIndex];
381:     }
382:
383:     showPrompt("住所 " + address[currentRecordIndex]);
384:     String inputAddress = Input.getString();
385:     if (inputAddress.equals("")) { //空白なら変更しないとする
386:       inputAddress = address[currentRecordIndex];
387:     }
388:
389:     showPrompt("電話番号 " + telephoneNumber[currentRecordIndex]);
390:     String inputTelephoneNumber = Input.getString();
391:     if (inputTelephoneNumber.equals("")) { //空白なら変更しないとする
392:       inputTelephoneNumber = telephoneNumber[currentRecordIndex];
393:     }
394:
395:     showPrompt("生まれた年 " + birthYear[currentRecordIndex]);
396:     int inputBirthYear = getValidInt();
397:
398:     showPrompt("入社年度 " + entranceYear[currentRecordIndex]);
399:     int inputEntranceYear = getValidInt();
400:
401:     showPrompt("年齢 " + age[currentRecordIndex]);
402:     int inputAge = getValidInt();
403:
404:     showPrompt("勤続年数 " + serviceLength[currentRecordIndex]);
405:     int inputServiceLength = getValidInt();
406:
407:     //データの変更を行う
408:     section[currentRecordIndex] = inputSection;
409:     name[currentRecordIndex] = inputName;
410:     kana[currentRecordIndex] = inputKana;
411:     zipCode[currentRecordIndex] = inputZipCode;
412:     address[currentRecordIndex] = inputAddress;
413:     telephoneNumber[currentRecordIndex] = inputTelephoneNumber;
414:     birthYear[currentRecordIndex] = inputBirthYear;
415:     entranceYear[currentRecordIndex] = inputEntranceYear;
416:     age[currentRecordIndex] = inputAge;
417:     serviceLength[currentRecordIndex] = inputServiceLength;
418:
419:     //画面の後始末
420:     clearMessageArea();
421:   }
422:
423:   /**
424:    * 注目行を前、あるいは次のページに移動する。移動できない時は、そのまま。
```

```
425:    */
426:   void paging(int direction) {
427:
428:     int nextRecordIndex = 0; //次のカーソル位置
429:
430:     //データが存在しないとき
431:     if (recordCount == 0) {
432:       return;
433:     }
434:
435:     //次のカーソル位置を計算する
436:     switch (direction) {
437:       case UP : //上に移動
438:         nextRecordIndex = currentRecordIndex - DISPLAY_RECORD_MAX;
439:         if (nextRecordIndex < 0) {
440:           nextRecordIndex = 0;
441:         }
442:         break;
443:       case DOWN : //下に移動
444:         nextRecordIndex = currentRecordIndex + DISPLAY_RECORD_MAX;
445:         if (nextRecordIndex >= recordCount) {
446:           nextRecordIndex = recordCount - 1;
447:         }
448:         break;
449:       default :
450:         break;
451:     }
452:
453:     //次のカーソル位置を設定する
454:     currentRecordIndex = nextRecordIndex;
455:   }
456:
457:   /**
458:    * データ 1 つ分のカーソルの移動を行う
459:    */
460:   void moveCursor(int direction) {
461:
462:     int nextRecordIndex = 0; //次のカーソル位置
463:
464:     //データが存在しないとき
465:     if (recordCount == 0) {
466:       return;
467:     }
468:
469:     //次のカーソル位置を計算する
470:     switch (direction) {
471:       case UP : //上に移動
472:         nextRecordIndex = currentRecordIndex - 1;
473:         if (nextRecordIndex < 0) {
474:           nextRecordIndex = 0;
475:         }
476:         break;
477:       case DOWN : //下に移動
478:         nextRecordIndex = currentRecordIndex + 1;
```

```
479:         if (nextRecordIndex >= recordCount) {
480:           nextRecordIndex = recordCount - 1;
481:         }
482:         break;
483:       default :
484:         break;
485:     }
486:
487:     //次のカーソル位置を設定する
488:     currentRecordIndex = nextRecordIndex;
489:   }
490:
491:   /**
492:    * 名簿データのセーブを行う.
493:    */
494:   void save() {
495:
496:     //ファイル名を入力する
497:     showPrompt("セーブするファイル名を入力してください"); //プロンプト
498:     String fileName = Input.getString();
499:
500:     //前処理
501:     PrintStream writer = FileIO.openForWrite(fileName); //ファイルオープン
502:
503:     //書き込み
504:     for (int i = 0; i < recordCount; i++) {
505:       //CSV 形式
506:       writer.println(
507:         id[i]
508:           + ","
509:           + section[i]
510:           + ","
511:           + name[i]
512:           + ","
513:           + kana[i]
514:           + ","
515:           + zipCode[i]
516:           + ","
517:           + address[i]
518:           + ","
519:           + telephoneNumber[i]
520:           + ","
521:           + birthYear[i]
522:           + ","
523:           + entranceYear[i]
524:           + ","
525:           + age[i]
526:           + ","
527:           + serviceLength[i]);
528:     }
529:
530:     //後処理
531:     writer.close(); //ファイルクローズ
532:
```

```
533:   }
534:
535:   /**
536:    * 名簿データのロードを行う.
537:    */
538:   void load() {
539:
540:     //ファイル名を入力する
541:     showPrompt("ロードするファイル名を入力してください");
542:     String fileName = Input.getString();
543:
544:     //前処理
545:     ReadStream reader = FileIO.openForRead(fileName); //ファイルオープン
546:
547:     //読み込み
548:     for (recordCount = 0; !reader.isEnd(); recordCount++) {
549:       //CSV 形式のデータを分割する
550:       String line = reader.readLine();
551:       String[] elements = line.split(",");
552:
553:       //データを追加する
554:       id[recordCount] = Integer.parseInt(elements[0]);
555:       section[recordCount] = elements[1];
556:       name[recordCount] = elements[2];
557:       kana[recordCount] = elements[3];
558:       zipCode[recordCount] = elements[4];
559:       address[recordCount] = elements[5];
560:       telephoneNumber[recordCount] = elements[6];
561:       birthYear[recordCount] = Integer.parseInt(elements[7]);
562:       entranceYear[recordCount] = Integer.parseInt(elements[8]);
563:       age[recordCount] = Integer.parseInt(elements[9]);
564:       serviceLength[recordCount] = Integer.parseInt(elements[10]);
565:     }
566:
567:     //後処理
568:     reader.close(); //ファイルクローズ
569:     currentRecordIndex = 0; //カーソルの初期化
570:   }
571:
572:   /******************************************************
573:   /* コマンド用部品
574:   /******************************************************/
575:
576:   /**
577:    * 数値の入力を受け取る
578:    * 入力が正しく無い場合は再入力を促す
579:    */
580:   int getValidInt() {
581:     while (true) {
582:       //入力する
583:       String input = Input.getString();
584:
585:       //入力が数字に変換可能なら入力された文字列を返す，そうでなければ再入力
586:       if (Input.isInteger(input)) {
```

```
587:         return Integer.parseInt(input);
588:       } else {
589:         showPrompt(INCORRECT_INPUT_ERROR_MESSAGE);
590:       }
591:     }
592:   }
593:
594:   /**
595:    * 指定されたコマンドが正しいコマンドかどうかを調べる
596:    */
597:   boolean isCorrectCommand(String command) {
598:     for (int i = 0; i < COMMANDS.length; i++) {
599:       if (COMMANDS[i].equals(command)) { //妥当なコマンドならば
600:         return true;
601:       }
602:     }
603:     return false;
604:   }
605:
606:   /*****************************************************
607:   /* 画面表示用部品
608:   *****************************************************/
609:
610:   /**
611:    * 画面を消去し、第 1 行目にタイトルを書いておく
612:    */
613:   void initializeTitle() {
614:     ConsoleWindow.clearScreen();
615:     ConsoleWindow.locateCursor(TITLE_LINE, TITLE_POSITION);
616:     System.out.println(COMPANY_NAME + "社員管理システム");
617:   }
618:
619:   /**
620:    * 指定されたエラーメッセージを表示する
621:    */
622:   void showError(String errorMessage) {
623:     ConsoleWindow.locateCursor(COMMAND_ERROR_LINE, 1);
624:     ConsoleWindow.clearLine();
625:     System.out.println("エラー: " + errorMessage);
626:   }
627:
628:   /**
629:    * メッセージエリアを消去する
630:    */
631:   void clearMessageArea() {
632:     ConsoleWindow.locateCursor(MESSAGE_START_LINE, 1);
633:     for (int i = 0; i < MESSAGEAREA_TERMINAL_LINE; i++) {
634:       ConsoleWindow.clearLine();
635:       System.out.print("");
636:     }
637:   }
638:
639:   /**
640:    * ページエリアを消去する
```

```
641:    */
642:   void clearPageArea() {
643:     ConsoleWindow.locateCursor(DATA_START_LINE, 1);
644:     for (int i = DATA_START_LINE; i < MESSAGE_START_LINE; i++) {
645:       ConsoleWindow.clearLine();
646:     }
647:   }
648:
649:   /**
650:    * コマンドメニューを表示する
651:    */
652:   void showCommandMenu() {
653:     ConsoleWindow.locateCursor(COMMAND_INFO_LINE, 1);
654:     ConsoleWindow.clearLine();
655:     System.out.print(COMMAND_INFO_MESSAGE);
656:   }
657:
658:   /**
659:    * コマンドプロンプトを表示する
660:    */
661:   void showCommandPrompt() {
662:     ConsoleWindow.locateCursor(COMMAND_INPUT_LINE, 1);
663:     ConsoleWindow.clearLine();
664:     System.out.print(COMMAND_PROMPT);
665:   }
666:
667:   /**
668:    * 指定されたメッセージを持つプロンプトを表示する
669:    */
670:   void showPrompt(String message) {
671:     System.out.println();
672:     System.out.print(message);
673:     System.out.print(COMMAND_PROMPT);
674:     System.out.flush();
675:   }
676:
677:   /**
678:    * 1 ページ分の社員の情報を 1 画面に書く。
679:    * 1 ページには、指定されたレコード数のデータを書く。
680:    */
681:   void showOneDirectoryPage() {
682:
683:     int start; //表示開始レコード番号
684:     int end; //  表示終了レコード番号
685:     int linePosition; //表示位置
686:
687:     //表示位置を計算する
688:     start =
689:       (((currentRecordIndex) / DISPLAY_RECORD_MAX) * DISPLAY_RECORD_MAX);
690:     end = start + DISPLAY_RECORD_MAX;
691:     if (end > recordCount) {
692:       end = recordCount;
693:     }
694:     linePosition = DATA_START_LINE;
```

```
695:
696:     //ページを初期化する
697:     clearPageArea();
698:
699:     //指定された件数分のデータを書く
700:     int count = 0;
701:     for (int i = start; i < end; i++) {
702:       showDirectoryRecord(linePosition, i);
703:       linePosition += LINES_FOR_A_RECORD;
704:       count++;
705:     }
706:     //件数が満たなければ空白で埋める
707:     while (count < DISPLAY_RECORD_MAX) {
708:       for (int i = 0; i < LINES_FOR_A_RECORD; i++) {
709:         System.out.println();
710:       }
711:       count++;
712:     }
713:   }
714:
715:   /**
716:    * 社員一人分の情報を表示する
717:    */
718:   void showDirectoryRecord(int position, int recordIndex) {
719:
720:     //前処理（移動）
721:     ConsoleWindow.locateCursor(position, 1);
722:
723:     //注目行マークをつける
724:     setMark(recordIndex);
725:
726:     //データを書き込む
727:     System.out.println(
728:       "\t"
729:         + (recordIndex + 1)
730:         + "\t"
731:         + id[recordIndex]
732:         + "\t"
733:         + section[recordIndex]
734:         + "\t"
735:         + name[recordIndex]
736:         + "\t"
737:         + kana[recordIndex]);
738:     System.out.println(
739:       "\t"
740:         + "\t"
741:         + "\t"
742:         + address[recordIndex]
743:         + "\t"
744:         + telephoneNumber[recordIndex]
745:         + "\t"
746:         + "〒"
747:         + zipCode[recordIndex]);
748:     System.out.println(
```

```
749:       "\t"
750:         + "\t"
751:         + birthYear[recordIndex]
752:         + "年生まれ\t"
753:         + age[recordIndex]
754:         + "才\t"
755:         + entranceYear[recordIndex]
756:         + "年入社\t勤続"
757:         + serviceLength[recordIndex]
758:         + "年");
759:
760:     //後処理（境界線）
761:     System.out.println(LINE);
762:   }
763:
764:   /**
765:    * 注目行マークをつける
766:    */
767:   void setMark(int recordIndex) {
768:     if (recordIndex == currentRecordIndex) {
769:       System.out.print("*");
770:     } else {
771:       System.out.print("  ");
772:     }
773:   }
774:
775: }
```

A.2 第 II 部

A.2.1 概要

第 II 部でのミニプロジェクトは、第 I 部で作った金銭出納帳の続きです。オブジェクト指向プログラミングを適用して、より美しいプログラムに変身させて下さい。

A.2.2 演習問題

A.2.2.1 オブジェクト指向の適用

第 I 部のミニプロで作った金銭出納帳プログラムをオブジェクト指向で書き直してみましょう。

1. 金銭出納帳のデータ構造をクラス図、参照モデルを使って設計してください。
2. 金銭出納帳プログラムのオブジェクト指向版を作ってください。

A.2.2.2 独自拡張

A.1.4.4 節で示したように、オブジェクト指向に直したプログラムに対して独自拡張を行ってください。

非オブジェクト指向のプログラムとどちらが拡張しやすいか考察してみましょう。

付録 B

HCP チャートの記法

B.1 設計例

[1]

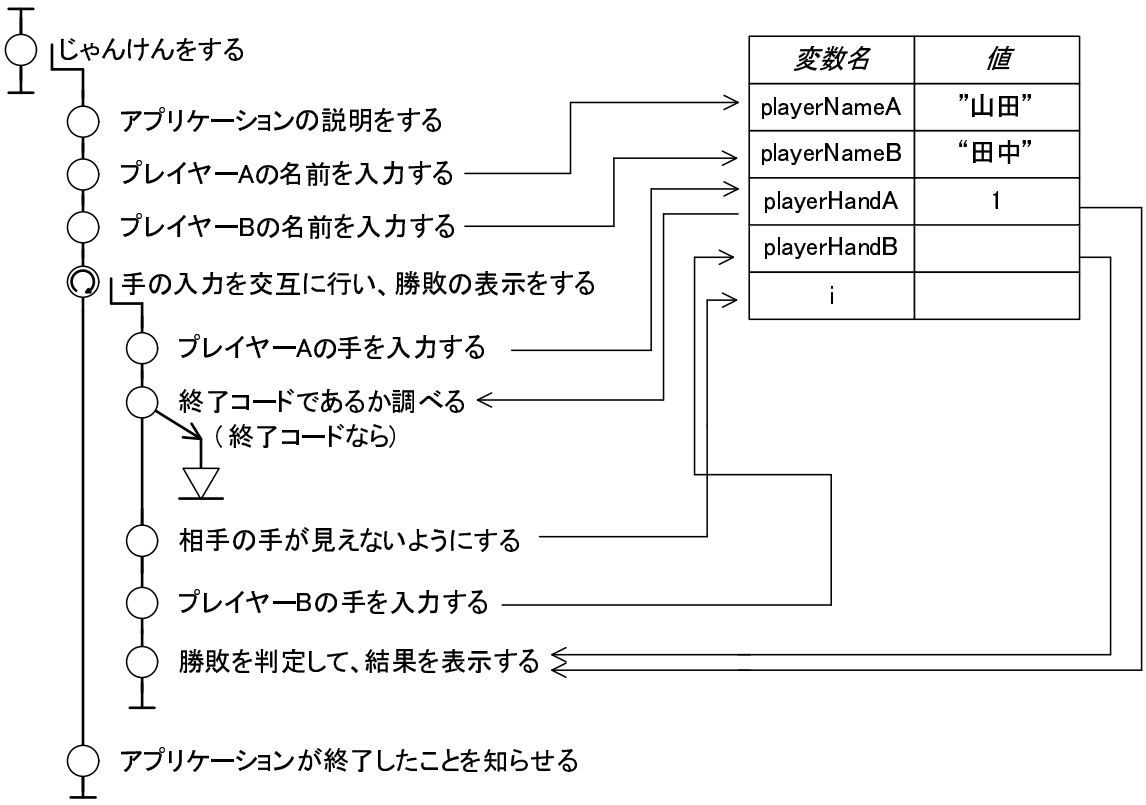


図 B.1: 「じゃんけんアプリケーション」の HCP チャート

[1] 図 2.3 と同じものです。

B.2　記法

B.2.1　繰り返し

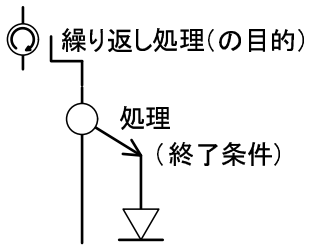


図 B.2: 繰り返しの HCP チャート記法

B.2.2　分岐

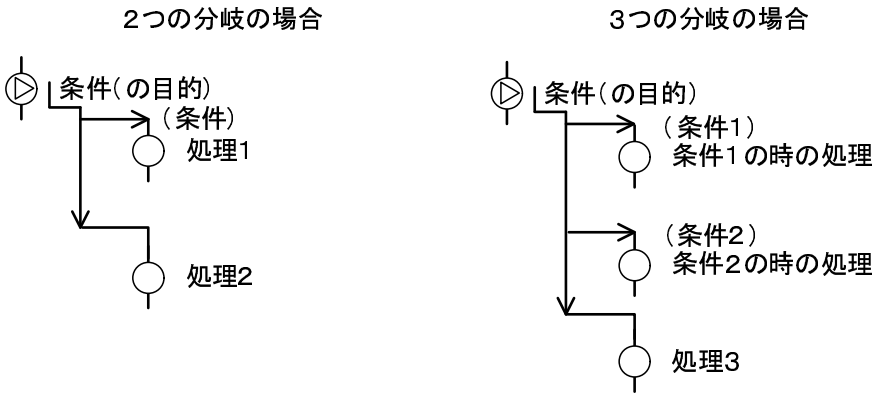


図 B.3: 分岐の HCP チャート記法

B.2.3 手続き

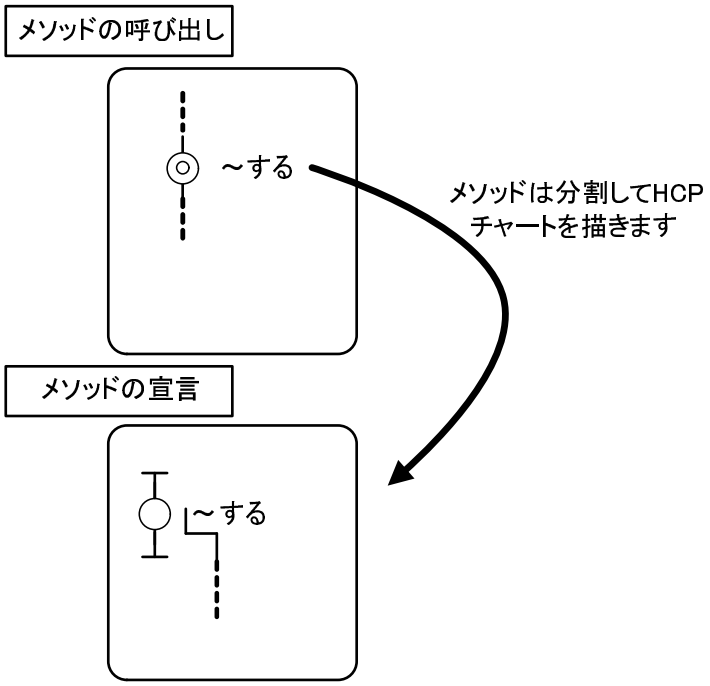


図 B.4: 手続きの HCP チャート記法