情報工学実験第二 課題 B1
Javaによるオブジェクト指向プログラミング
(OOP コース)

～第2回目：スタックの実装・例外処理・端末I/O～

目次

First Java Version by K. Tokuyi, Oct. 1997

第5章

スタックの実装

5.1 Stack クラスの定義

Java のクラス定義を利用して、整数スタックのクラス Stack を記述し、実装してみましょう。整数型の配列を使ってプログラムすることにします。スタックには、データを格納しておく int 型配列 data[] と、データ数を記憶する変数 count が必要ですね。これらはプライベート変数とし、クラスの外部から直接参照できないように保護しましょう。

スタックのメソッドは、次の5種類を用意します。

`void init()` スタックを初期化する。

`void push(int item)` スタックに整数 item を入れる。

`int pop()` スタックから取り出す。

`int top()` 一番上の要素を見る。

`boolean isEmpty()` スタックが空であるかどうか調べる

クラス定義は次のようになります。メソッドの中身はまだ書いていません。

```
1     public class Stack {
2       private final static int STACK_SIZE = 100;
3       private int count;
4       private int data[];
5       public void init() { . . . }
6       public void push(int item) { . . . }
7       public int pop() { . . . }
8       public int top() { . . . }
9       boolean isEmpty() { . . . }
10    }
```


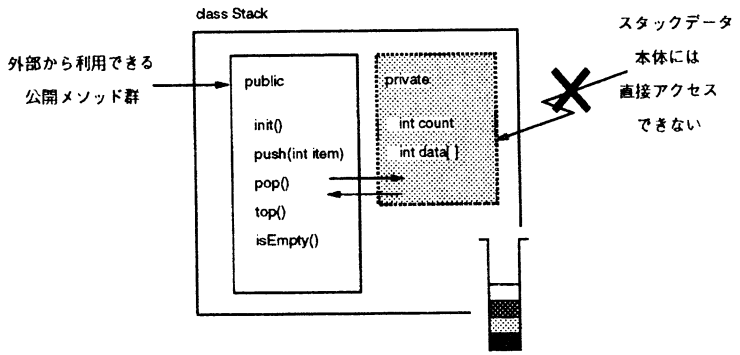

図 5.1: スタック・クラス。スタックのデータ本体は非公開にして保護する。

5.1.1 配列

配列の生成

1行目ではスタックの実体として int 型の配列を宣言しています。Java の配列には次のような特徴があります。

- 配列は参照で扱われます。
- new を使って動的に生成されます。
- 参照されなくなると自動的にカーベッジ・コレクションの対象となります。

Java で配列を生成する方法は2つあります。1つは new を使い、割り当てる配列の大きさを指定します。

```
int int_array[] = new int[1024];    // int 型の大きさ 1024 の配列
char char_array[] = new char[16];   // char 型の大きさ 16 の配列
Stack stacks[] = new Stack[10];     // Stack オブジェクトの大きさ 10 の配列
```

このようにして生成された配列の各要素は、その型の省略値 (デフォルト値) で初期化されます。たとえば、int 型の配列の要素は 0 に、オブジェクト型の配列の要素は null に初期化されます。

2つ目の方法は、配列の初期値を使います。これは C のやり方と同様です。

```
int lookup_table[] = {1,2,4,8,16,32,64,128};
```

この構文により、動的に配列が生成され、その各要素は指定された値に初期化されます。初期値で指定する要素は、C では定数しか指定できないのと異なり、どのような式でもかまいません。

配列宣言に関するもう一つのポイントは、配列のサイズは C のように型に含まれないということです。たとえば、変数を String[] 型と宣言すると、配列の長さに関わらずどのような String 配列も代入できます。

```
String[] strings;
strings = new String[10];  //10 個の String を含むものを参照したり
strings = new String[20];  //20 個の String を含むものを参照したりできる
```

配列要素へのアクセス

Java での配列のアクセスは C と同様です。つまり、整数値となる式をかぎ括弧の中に入れて配列名の後に続けることで、配列の要素へとアクセスできます。

```
int a[] = new int[128];
a[0] = 0;
for(int i=1; i<a.length; i++) {
    a[i] = i+a[i-1];
}
```

配列の要素数の求め方に注意してください。つまり、配列の length フィールドにアクセスすればいいのです。配列でサポートされているのはこのフィールドだけで、読み出し専用です。Java の配列参照では、添字が小さ過ぎたり大き過ぎたりしないようにチェックを行なっています。添字が範囲外であれば、ArrayIndexOutOfBoundsException という型の例外 (後述) が発生します。これは Java のバグとセキュリティ問題を防ぐための手段の 1 つです。

5.1.2 定数の定義

Java には C におけるプリプロセッサはありません。つまり、#define、#include、#ifdef などは使えません。定数はどのように宣言したらよいのでしょうか?

Java では final 宣言された変数は全て定数になります。final という修飾子は、その変数の値が変更できないことを示します。これと static という修飾子を使えば、定数に名前をつけることができます。static は変数が「クラス変数」であること、つまりクラスのインスタンスがいくつ生成されても、この変数のインスタンスはただ 1 つであることを宣言します。

Stack クラスでは、配列のサイズを次のように定義しています。定数名には大文字を使うという C の慣習が、Java でも踏襲されています。

```
2         private final static int STACK_SIZE = 100;
```


5.2 メソッドの実装

さて、メソッドの中身を記述しましょう。まず、初期化のための init() では、サイズが STACK_SIZE である整数配列を生成して、data に入れ、要素数 count を 0 に初期化します。

```
public void init() {
  data = new int[STACK_SIZE];
  count = 0;
}
```

push() は、整数型の引数を 1 つ受けとって、その値を配列の最後の要素の次の要素に格納します。C でファイル操作のライブラリ関数を呼ぶときにファイルハンドルを指定するのとは違って、引数にスタック自身を指定する必要がないことに注意してください。pop() は、要素数を 1 つ減じ、配列の最後の要素を戻り値として返します。isEmpty() では、要素数が 0 であるかどうか調べればいいですね。

```
public void push(int item) {
  data[count] = item;
  count++;
}
public int pop() {
  count--;
  return data[count];
}
boolean isEmpty() {
  if (count <= 0) {
    return true;
  } else {
    return false;
  }
}
```

現在のクラス定義では、スタックが溢れたり、空のスタックから取り出そうとしたときの対策がありません。エラー処理はあとで追加するとして、とりあえずこのクラス使ってみたのが下のプログラムです。

```
1       public class Main {
2         public static void main(String argv[]) {
3           Stack stack;
```

```
4           stack = new Stack();     // インスタンス生成
5           stack.init();             // 初期化
6           stack.push(1);            // stack = 1
7           stack.push(2);            // stack = 2 1
8           stack.push(3);            // stack = 3 2 1
9           while (!stack.isEmpty()) {   // 空になるまでpop
10             System.out.println(stack.pop());
11          }
12        }
13      }
```

5.3 オブジェクトの生成

スタックオブジェクトをどのようにして生成したか、もう一度見てみましょう。

```
4           stack = new Stack();     // インスタンス生成
```

この括弧はなにを意味するのでしょうか? 関数呼び出しをしているように見えます。事実、まさにここでStack() という関数を呼び出しているのです。Javaでは、各クラスにはクラスと同じ名前の関数が少なくとも1つあり、ここで新しいオブジェクトに必要な初期化を行なっているのです。このStackクラスのように、明示的に定義していない場合は、引数が無く、特別な初期化はなにも行なわないStack() という関数をJavaが自動的に作ってくれます。

5.3.1 コンストラクタ

そのような関数を**コンストラクタ (constructor、構築子)** といいます。コンストラクタはクラス名と同じ名前を持ちます。もちろん、あらかじめ明示的に定義しておけば、その中でプログラマが必要な初期化処理をすることができます。

一般にオブジェクトの生成は次の手順で行なわれます。

1. キーワードnewでクラスの新しいインスタンスを動的に生成(メモリ割り当て)。
2. コンストラクタが呼び出され、暗黙にその新オブジェクトが渡される。同時に、括弧内に明示的に指定した引数も渡される。
3. コンストラクタが記述された初期化処理を実行。

コンストラクタを用いて、自動的にスタックを初期化することにしましょう。init関数の代わりに、コンストラクタStackを定義します。コンストラクタは必ずpublic関数として定義します。


```
    public Stack() {
      data = new int[STACK_SIZE];
      count = 0;
    }
```

コンストラクタの宣言で、戻り値voidの型の指定がないことに気付きましたか? コンストラクタは戻り値を返すことはないので、わざわざvoidと書く必要はないのです。

コンストラクタを定義したので、メインプログラムは簡単になります。

```
Stack stack;
stack = new Stack();
stack.init();
```

と書く代わりに、

```
Stack stack;
stack = new Stack();  // (^^)あとは使うだけ
```

と書けばスタックを使う準備ができます。

5.3.2 複数のコンストラクタ

オブジェクトの初期化は、その時の状況によって最も便利な形式を選び、様々な方法で行なった方がよい場合があります。たとえば、スタックで扱うデータ量の上限があらかじめわかっているのなら、大きさを指定して初期化したいということがあります。クラスはコンストラクタを複数持つことができるので、これには何の問題もありません。

Stack.java:

```
    . . .
    public Stack() {   // コンストラクタ1
      data = new int[STACK_SIZE];
      count = 0;
    }
    public Stack(int size) { // コンストラクタ2:大きさを指定する
      data = new int[size];
      count = 0;
    }
```

Main.java:

```
    public class Main {
```

```
public static void main(String argv[]) {
  Stack stack1, stack2;
  stack1 = new Stack();      // コンストラクタ1が起動
  stack2 = new Stack(1000); // コンストラクタ2が起動
  . . .
```

5.3.3 メソッドのオーバーロード

この例で注目すべき点は、コンストラクタが全て同じ名前であるということです。コンパイラは、メソッド名、リターン名、引数の数とその型、それらの位置によってメソッドを区別してくれます。これはコンストラクタに限ったことではなく、普通のメソッドも同様のメカニズムで区別されます。あるメソッドを呼び出したとき、それと同じ名前のメソッドが複数ある場合は、そのとき与えられた引数の形式に一致するものをコンパイラが自動的に選択してくれます。

同じ名前で、異なる引数を持つメソッドを定義することを、メソッドのオーバーロードといいます。少しだけ形式の異なる入力データに対して同じ処理を行なう場合のみに、同名のメソッドを定義すれば、これは便利な機能といえます。

☺

第 6 章

例外処理

第5章のスタッククラス定義では、スタックが溢れたり、空のスタックに対して pop しようとしたときのエラー処理がありませんでした。とりあえず、次のような対策が考えられます。push() の場合は、まずスタック要素の数をカウントしている変数 count の値が STACK_SIZE 超えているかどうか、pop() の場合は、count の値が 0 より大きいかどうかでチェックできます。

```
public void push(int item) {
  if (STACK_SIZE <= count) {
    System.err.println("Error: Push overflows stack");
  } else {
    data[count] = item;
    count++;
  }
}
public int pop() {
  if (count <= 0) {
    System.err.println("Error: Pop causes stack underflow");
    return 0;
  } else {
    count--;
    return data[count];
  }
}
```

C では、上の例のようにプログラマーが自分でエラーを検出して、処理するコードを書かなければなりません。しかし Java では、ある種の例外的な状態 (エラーなど) が生じたときに、それをに対して必要ななんらかの処理が行なえるような仕組みが言語レベルでサポートされています。たとえ

ば、5.1.1節で触れたように、Java では、実行時に配列の添字値の範囲についてのチェックが行なわれ、不適当な値が与えられたらそのことが通知されるので、その通知を受けて適切な処理を施すことができます。

6.1 例外と例外処理

例外処理は、Java の新しい機能です (といっても C++にも似たような機能があります)。話を進める前に、用語の定義をしておきます。

例外 (exeption) ある種の例外的状態 (エラーなど) が生じたことを示すシグナル。

例外をあげる (throw) 例外的状態のシグナルを発すること。

例外を受ける (catch) 例外状態から回復するために必要ななんらかのアクションをとること。

例外処理は図 6.1のようなメカニズムで処理されます。

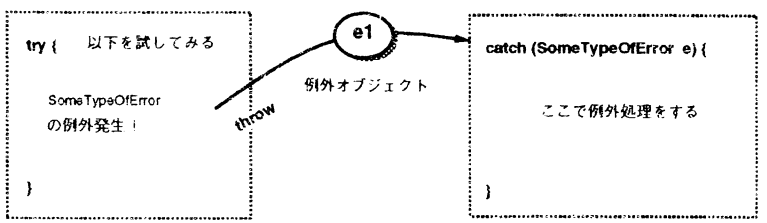

図 6.1: 例外処理 : try — throw — catch

6.1.1 例外オブジェクト

プログラムの実行中に例外が発生すると、**例外オブジェクト** という特別なオブジェクトが生成されます。これは、java.lang.Throwable というクラスのインスタンスのようなオブジェクトです。

Throwable には java.lang.Error と java.lang.Exception という標準的な 2 つのサブクラスがあります。**Error** は、メモリ不足のような Java 仮想マシンでの回復不能な問題で、普通 catch することはできません。インタプリタはエラーメッセージをプリントして終了します。**Exception** に属する例外には、ファイルの終わりを知らせる java.io.EOFException、配列の添字の範囲外を知らせる java.lang.ArrayIndexOutOfBoundsException などが含まれます。この種の例外は、catch して処理することができます。

例外はオブジェクトであるため、データを含み、メソッドを定義することができます。Throwable オブジェクトには String オブジェクトが含まれているので、これを受けてメッセージをプリントすることができます。例外オブジェクトは、プログラマが新しく定義して生成することもできます。

6.1.2 try

try は例外を扱うコード・ブロックを設定します。すなわち、エラーが起こる可能性のあるコードを try に続く中括弧で囲っておきます。try 節自体は、特別なことはしません。

6.1.3 catch

try ブロックの後には、様々な型の例外を処理するコードを指定した catch 節を書きます。catch 節の構文は図 6.1のような変わった形式のもので、その節で処理したい例外オブジェクトの種類と、オブジェクトを引数に持ちます。この catch 引数はその catch 節の中だけで有効で、あげられた実際の例外オブジェクトを参照しています。catch 節の中では、その例外的状態の対処に必要な処理を行ないます。

Stack.push() メソッドのエラー処理を try/catch を使って書くと、次のようになります。

```
public void push(int item) {
  try {
    data[count] = item;
    count++;
  }
  catch (ArrayIndexOutOfBoundsException e) {
     System.err.println("Error: Push overflows stack");
  }
}
```

6.1.4 finally

finally 節は一般に、ファイルのクローズ、リソースの解放などの try 節の後始末をします。finally 節の便利な点は、try 節のコードが一部でも実行されれば、その try 節からどのように出ようとも、必ず finally 節内のコードが実行されることが保証されているという点です。普通の場合は、制御が try 節の最後に到達すると、次に finally 節に進み必要な後始末を行ないます。

Java 言語での finally 節も含めた例外処理の枠組は、次のようになります。finally 節は必ずしも必要ではありません。また、catch 節を省略して、try と finally だけを一緒に使うこともできます。

```
try {
    通常の処理;
    . . .
}
catch(例外クラス型 1 変数) {
    例外処理;
}
catch(例外クラス型 2 変数) {
    例外処理;
}
. . .
catch(例外クラス型 n 変数) {
    例外処理;
}
finally {
    後始末;
}
```

☺

第7章

練習問題2

今日の練習問題では、いよいよクラスのプログラミングを行ないます。クラスとインスタンス（オブジェクト）の違い、メソッド・演算子のオーバーロード、例外処理の仕組みに注意しましょう。

練習問題2-1：キュー

キュー(queue)クラスをプログラムしてください。キュー要素の型はintとします。キューの構造や操作はスタックに類似しているので、スタッククラスを改造すれば実現できそうですね。なお、キューが溢れた場合の例外処理も忘れずに。

適当なテストプログラムを作って、キューとしての機能がちゃんと実現されているかどうか、エラー対策は大丈夫かどうかなどについてもテストしてください。

練習問題2-2：長方形クラス

長方形(rectangle)を表現するクラス"Rect"を作ってください。オブジェクト変数としては、どんなものが必要でしょうか？向かい合う頂点の座標(x1,y1,x2,y2)があれば、長方形を表現できますね。Rectクラスは、これらの座標を操作する次のメソッドを持っています。

`Rect(int x1, int y1, int x2, int y2)`：コンストラクタ

`Rect(int xo, int yo, int width, int height)`：別のコンストラクタ

`move(int deltax, int deltay)`：指定した値で矩形を移動する。

`boolean isInside(int x, int y)`：点(x、y)が長方形の内部にあるかどうか検査する。

`String toString()`：長方形の情報を文字列にして返す。これは、スーパークラスであるObjectクラスのメソッド(付録A.4を参照)ですが、これをオーバーライドします。

(返り値の文字列書式例)

`[0,0,3,5]` ... 頂点が(0,0),(3,5)である長方形。

適当なテストプログラムを作って、各メソッドの動作を確認してください。

練習問題2-3：円クラス

Rectクラスと類似している、円を表現する"Circle"クラスを書いてください。move()メソッドとisInside()メソッドを定義してください。適当なテストプログラムを作って、各メソッドの動作を確認してください。

実習

1. 練習問題2-1は全員必ずやって下さい。
2. 次に、学籍番号の末尾の桁が奇数の人は練習問題2-2を、偶数の人は練習問題2-3をやってください。

written by K. Takagi, Ver1.0 Oct. 1, 1998

第 8 章

カウンタの作成

これまで Java を用いた簡単なクラスの定義と実装のしかたを学んできました。今日は、カウンタ (計数器) のプログラムを例にとって、少し複雑なクラスの作りかたを勉強します。今日作るカウンタは、ユーザーが任意の桁数、基数を選ぶことができ、カウントアップとカウントダウンの両方の機能を持つものです。

8.1 クラスの部品としての利用

オブジェクト指向プログラミングでは、どちらかというと部品を先に作り、最後に全体をまとめるというボトムアップ的アプローチをとります。例題では、カウンタを構成する部品を先に作り、それらを組み合わせてカウンタを作ることにします。すなわち、すでに定義したクラスのインスタンスを部品として用いて、組み合わせ、より大きなクラスを定義します。

交通量調査などで使われている機械式カウンタを組み立てることを想像して、同じような手順でプログラムすることにします (図 8.1)。具体的には、まず 1 桁分だけのカウントができる部品を作ります (UnitCounter クラス)。次に、その部品に、桁上がりを上位の桁に伝達する歯車をつけます (LinkCounter クラス)。最後に LinkCounter を必要な桁数だけ接続して、カウントアップ、ダウンするボタンを付けてケースをかぶせて完成です (Counter クラス)

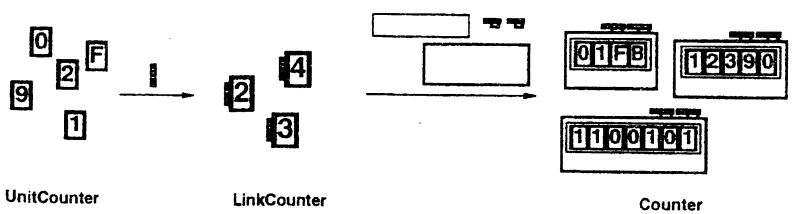

図 8.1: 部品を組み上げてカウンタを作る

8.2 UnitCounter

まず、カウンタの各桁に対応するクラスを、UnitCounter という名前で作ります (図 8.2)。クラス UnitCounter は、その桁の現在の値を状態として持ちます。あとで、2 進数、16 進数などにも対応したいので、基数も覚えておくことにしましょう。メソッドとしては、次のものを用意します。カウントアップとダウンのメソッドは、桁上がりあるいは桁下がりをメソッドの値として返す必要があります。

UnitCounter(rdx) 基数 rdx のカウンタを生成

inc() 1 つカウントアップする

dec() 1 つカウントダウンする

getValue() 値を返す

getRadix() 基数を返す

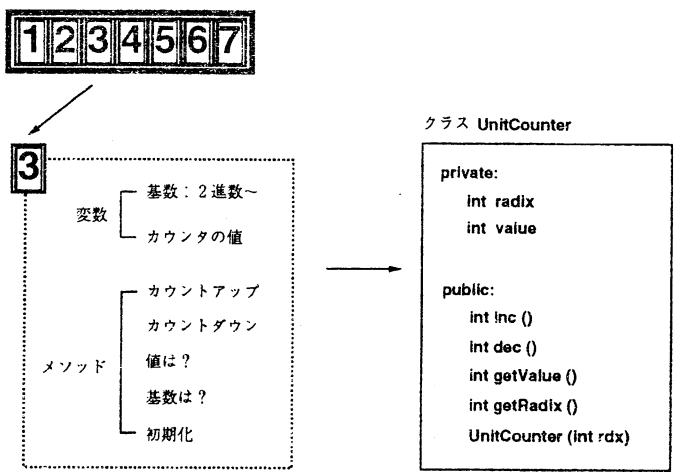

図 8.2: クラス UnitCounter

メソッドの中身は簡単ですね。下のようになります。

```
public class UnitCounter {
  private int value;
  private int radix;

  public UnitCounter(int rdx) {    // コンストラクタ
    value = 0;
```

```
    radix = rdx;
  }
  public int inc() {              // カウントアップ
    value++;
    if (radix <= value) {         // 桁上がり
      value = 0;
      return 1;
    } else {
      return 0;
    }
  }
  public int dec() {              // カウントダウン
    value--;
    if (value < 0) {              // 桁下がり
      value = radix - 1;
      return 1;
    } else {
      return 0;
    }
  }
  public int getValue() {         // 値を返す
    return value;
  }
  public int getRadix() {         // 基数を返す
    return radix;
  }
}
```

コンストラクタは、カウンタ値をリセットして、基数をオブジェクト変数にセットします。inc() は、現在の値 value を 1 だけ増加させて、繰り上がりが生じたら、すなわち value が基数 radix 以上になったら、value を 0 に戻して、桁上がり値 1 を返します。dec() でも同様の処理をしています。getValue()、getRadix() はそれぞれ、value と radix の値をメソッドの値として返すだけです。

8.3 LinkCounter

つぎに、UnitCounter を隣の UnitCounter オブジェクトと接続する機能を追加した新しいクラス LinkCounter を作ります (図 8.3)。既に定義されているクラスの性質を引き継いで、別のクラスを作るためにはクラスの「継承」というオブジェクト指向言語の機能を使うこともできますが、ここでは UnitCounter を部品として使うことにします。継承については課題 6 で学びます。

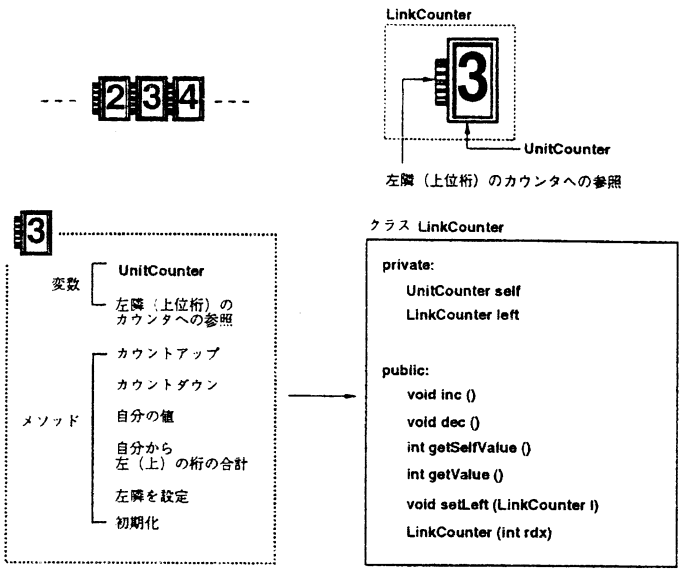

図 8.3: クラス LinkCounter

8.3.1 オブジェクト変数

LinkCounter のクラスには、どのような変数を与えればいいでしょうか。まず、自分自身 1 桁分のカウンタを持っています。前節で UnitCounter を定義しておきましたから、これを利用しましょう。さらに、左隣のカウンタに桁上がりを渡さなければなりませんから、左隣のカウンタへの参照が必要です。したがって、オブジェクト変数の定義は、次のようにしておけばよいでしょう。

```
private UnitCounter self;       // 自分自身
private LinkCounter left;       // 左隣
```

8.3.2 コンストラクタ

LinkCounter のコンストラクタも、基数を表す引数を持つ必要があります。なぜならば、その基数を用いて自分の中の UnitCounter のオブジェクトを生成する必要があるからです。コンストラクタの役目は、自分が指定された基数を使って、UnitCounter のインスタンス self を生成することです。

```
public LinkCounter(int rdx) {
    self = new UnitCounter(rdx);   // 基数 rdx の 1 桁カウンタ
    left = null;                   // 隣はまだわからない
}
```

8.3.3 桁上がり・桁下がり

メンバー関数 inc() はどのように実現したらいいでしょうか。inc() メソッドが実行されたら、まず、自分の UnitCounter クラスのオブジェクト self の inc() メソッドを呼びます。実行の結果、繰り上がりがあり、かつ自分よりも左 (上位) の桁があれば、左の LinkCounter の inc() メソッドを呼んで、桁上がりを上の桁に加算すればいいですね。この仕組みをコードにすると次のようになります。dec() も同じようにします。

```
public void inc() {
    int carry = self.inc();
    if (省略) {
        ... 省略 ...;
    }
}
```

LinkCounter にも UnitCounter のように、自分の値を返すメソッドが必要ですね。このメソッドを getSelfValue() とします。

```
public int getSelfValue() {
    return self.getValue();
}
```

カウンタ値を計算するメソッドをもう 1 種類作ります。自分から左 (上位) の桁のカウンタの合計値を返すものです。このメソッドを getValue() としましょう。自分の左の LinkCounter があれば、その getValue() 関数を実行し、戻り値として上の桁の値をもらいます。その値に基数を掛けて、自分の値を足したものがその桁の getValue() の返り値です (図 8.4)。

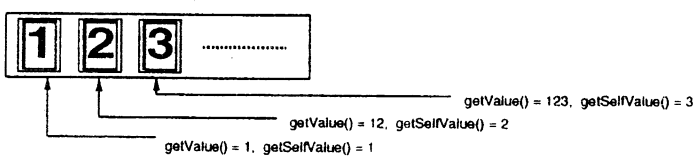

図 8.4: LinkCounter.getValue() の値

```
public int getValue() {
    int sum = ... 省略 ...
    if(left != null) {
        sum += ... 省略 ...
    }
    return sum;
}
```


8.4 Counter

最後に LinkCounter を必要な桁数だけ接続して、カウンタ全体をモデル化したクラス Counter(図 8.4) を組み立てます。LinkCounter のインスタンスを指定された桁数 size だけ生成して、全体が size 桁、基数 rdx のカウンタとして働くようにするのが Counter クラスの機能です。

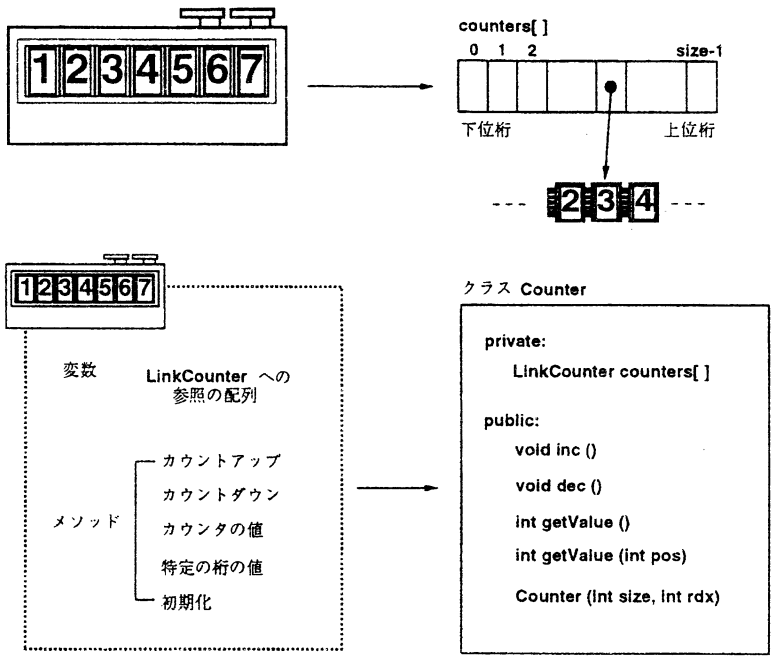

図 8.5: クラス Counter

8.4.1 クラスの定義

クラス Counter の定義を見てみましょう。オブジェクト変数としては、LinkCounter の配列が必要です。メソッドは、コンストラクタ、カウントアップ、カウントダウンの他に、カウンタ全体の値を返すメソッドを定義します。端末にカウンタを表示するために、特定の桁の値だけを返せるようにもしておきましょう。

```
public class Counter {
  private LinkCounter counters[];
  public Counter(int size,int rdx) { . . . } // コンストラクタ
  public void inc() { . . . }                // カウントアップ
```

```
public void dec() { . . . }          // カウントダウン
public getValue() { . . . }          // カウンタ全体の値
public getValue(int pos) { . . . }   // 特定の桁の値
```

8.4.2 コンストラクタ

Counter クラスの主要な機能は、指定された桁数分の LinkCounter オブジェクトを生成し、それらを接続することです。それはコンストラクタで行ないます。コンストラクタの処理の枠組は、次のようになります。

```
public Counter(int size,int rdx) {
    counters = ... 省略...  // size 個の要素を持つ LinkCounter 配列を生成

    ... 省略...      // size 個の LinkCounter オブジェクトを生成して
                     // counters に入れる。

    for (int i = 0; i < size-1; i++) {
        ... 省略...          // i 桁目の左に i+1 桁目を接続
    }
}
```

LinkCounter をつなぐためには、LinkCounter のオブジェクト変数 left に隣の LinkCounter への参照をセットする必要があります。したがって、LinkCounter にこの仕事をするメソッドを追加しましょう。

```
public void setLeft(LinkCounter l) {
    ... 省略 ...
}
```

8.4.3 カウント機能

Counter クラスの inc()、dec() の中ではどのような仕事をしたらよいでしょうか。UnitCounter、LinkCouner と組み上げてきて、結局 Counter クラスは図 8.6 のようになっています。それぞれの 1 桁表示カウンタは桁上がり機構で連結されていますから、カウンタ全体をカウントアップするには、再下位の桁をカウントアップすればよいことになります。カウントダウンも同様に考えれば実装できます。

getValue()、getValue(int pos) は、それほど難しくはありませんね。各 LinkCounter オブジェクトは、自分から上位の桁の合計を計算できますから (図 8.4)、それを利用します。

8.5 メインルーチン

カウンタの機能をテストするプログラムを作りましょう。ここでは、キーボードから入力された文字数を数えるプログラムを作ります (リターンキーも数えます)。カウンタの桁数、基数、カウントの方向は、この順にコマンドライン引数として指定します。コマンドライン引数は全て省略可能にし

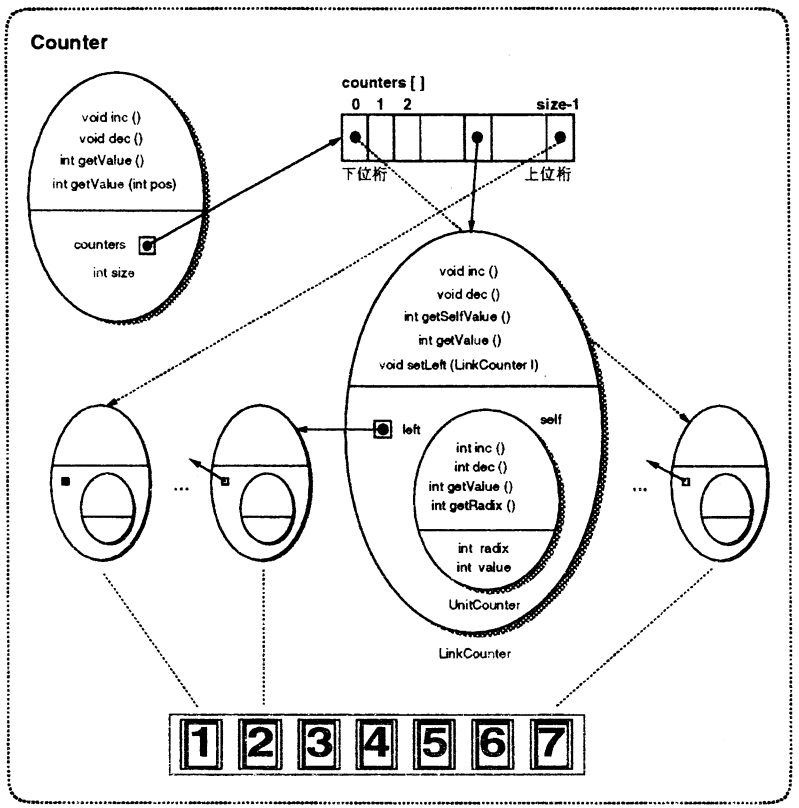

図 8.6: カウンタのオブジェクト関係図

ておきます。ただし、指定する場合は必ずこの順に指定することとします。全て省略された場合は、3 桁、10 進数、カウントアップに設定されます。実行結果としては、たとえば以下のようなものを期待しています。

```
% java Count          // 3 桁,10 進数, カウントアップ
me<Return>           // 入力文字 'm','e','\n'
0 0 1 (1)            // 1 文字目のカウント結果
0 0 2 (2)            // 2 文字目
0 0 3 (3)            // 3 文字目
love<Return>         // 入力文字 'l','o','v','e','\n'
0 0 4 (4)
0 0 5 (5)
0 0 6 (6)
0 0 7 (7)
```

```
<Ctrl-D>              // 終了
%

% java Count 5 2 -1   // 5桁,2進数, カウントダウン
me<Return>
1 1 1 1 1 (31)        // 最大値からカウントダウンし始める
1 1 1 1 0 (30)
1 1 1 0 1 (29)
love<Return>
1 1 1 0 0 (28)
. . .
0 0 0 0 0 (0)         // 0に戻ったら
<Return>
1 1 1 1 1 (31)        // つぎはまた最大値からカウントダウンし始める
                      // ... カウントアップの場合も同様
<Ctrl-D>              // 終了
```

メインプログラムはどのような構成になるでしょうか。処理としては、まずコマンドライン引数があれば、それにしたがって希望のカウンタを生成します。なければ省略時のカウンタ構成にします。次に、端末からの入力を待ち、一行入力されたら改行文字も含めて文字数を計算し、その数だけ1つずつカウンタを動かし(アップまたはダウン)途中経過を端末に表示することを繰り返せばよいでしょう。最後は<Ctrl-D>で強制終了させます。

さて、Javaで端末からデータを読んだり、端末にデータを書き出したりするにはどうしたらよいのでしょうか。第9章で説明します。

☺

第9章

端末I/O

9.1 コマンド引数

プログラムのコマンドライン引数を処理するためには、C/C++では下のコードのようにしました。実行開始時に main が呼び出されたときに、2つの引数が渡されます。最初の引数 argc は引数の個数、2番目の引数 argv は引数を含む文字列の配列です。

```
/* C/C++におけるコマンド引数の使用 */
main(int argc, char *argv[]) {
    int i;
    for(i=0; i<argc; i++) {      /* argc は引数の個数 */
        /* argv[i] の処理 ... argv[0] はプログラム名   */
    }
```

Javaでも同じことができますが、形式は多少違っています。Javaでは main メソッドの引数は1つだけで、引数の個数は String 型の配列 argv の長さで調べられます。Java の配列は要素の個数も保持しているのでしたね。なお、Java の場合は、argv[0] には第1番目のコマンド引数が入ります。したがって、引数になにも指定しない場合は、argv.length = 0 で、この点も C/C++とは異ります。

```
public class Main {
  public static void main(String argv[]) {  // argv だけ
        int i;
        for(i=0; i<argv.length; i++) {
            // argv[i] の処理                // argv[0] は第1引数
        }
  }
```

9.2 文字型から数値型への変換

いま作成中のカウンタプログラムでは、カウンタの桁数、基数、カウントの方向をコマンド引数として受けとります。コマンド引数 argv[i] は String 型のオブジェクトなので、使うためには整数型に変換する必要があります。

C/C++言語では、atoi() または sscanf() を使えば文字列を数値型に変換できますが、Java では、少し面倒くさいやり方をします。

整数型　たとえば、整数型への変換は次のようにします。

```
int size = Integer.valueOf(argv[0]).intValue()
```

まず String 型を整数の文字表記を解釈して、Integer 型のオブジェクトを生成し、その内容を int 型として得るというものです。

なお、この Integer クラスのメソッドである parseInt を使うと、もう少し簡単に整数値を得ることができます。

```
int size = Integer.parseInt(argv[0]);
```

valueOf() と parseInt() というメソッドは、引数の文字列が整数として解釈できない場合、NumberFormatException という例外型を投げます。したがって、実際のプログラムでは例外処理も一緒に書いておくべきです。

```
int size;
try {
    size = Integer.parseInt(argv[0]);
}
catch(NumberFormatException e) {
    System.err.println("parse error: "+argv[0])
    size = 0;
}
```

なお、parseInt() メソッドは、引数を2つとることもできます。その場合は、第2引数として基数を指定します。この形を使えば、たとえば16進数を解釈して int 型の変数に入れることができます。

```
String s = "FF";                    // "ff"でも OK
int i;
try {
    i = Integer.parseInt(s,16);
}
catch(NumberFormatException e) {
    System.err.println(e)
    i = 0;
}
```

long 型　long 型に変換する場合は、Integer を Long に、intValue の代わりに parseLong を使えばできます。

float 型・double 型　float 型と double 型の場合は parseInt() に相当する方法がないので、次のようにします。文字列を float 型の文字表現と解釈して、その値を持つ Float クラスのオブジェクトを生成し、その値を得るという方法です。次に float 型の場合を示します。

```
String s = "1.2";
float f;
try {
    f = new Float(s).floatValue();
}
catch(NumberFormatException e) {
    System.err.println(e)
    f = 0;
}
```

9.3 ストリーム

Java はストリーム (stream) に基づいて、基本的な入出力を行ないます。ストリームは byte のデータの流れです。まず 1 バイト目、次に 2 バイト目というように順にデータが流れます。ストリームの一方の端にデータの源があり、もう一方にデータの行き先があるといった構造になります。

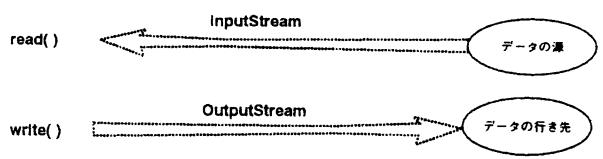

図 9.1: InputStream と OutputStream の基本的機能

9.3.1 入出力ストリーム

ストリームには、入力ストリームと出力ストリームがあり、Java ではそれぞれに対して、入力ストリーム (InputStream) クラス、出力ストリーム (OutStream) クラスを定義しています。入力ストリームに対しては、InputStream オブジェクトの read() メソッドを呼び出すと、流れてきたデータを取り出すことができます。出力ストリームの場合は、OutputStream オブジェクトの write() を呼び出してデータを流します。

Java では入出力の API は、java.io というパッケージにまとめられています。ここには、かなり多くのクラスが含まれていますが、きれいに階層化された構造をしていて、その大部分は InputStream や OutputStream のサブクラスです。

InputStream と OutputStream には、様々な入出力先に対してのデータの読み書きを行なうストリームが定義されています。入力には 4 種類、出力には 3 種類用意されています。

表 9.1: ストリームクラス

入力ストリーム	出力ストリーム	データの源、行き先
FileInputStream	FileOutputStream	ファイル
ByteArrayInputStream	ByteArrayOutputStream	byte の配列
StringBufferInputStream	StringBufferOutputStream	文字列

青柳、「Java API プログラミングガイド」、p. 66 より転載

9.3.2 端末 I/O

InputStream オブジェクトの代表的な例は、Java アプリケーションの標準入力です。C の stdin や C++の cin と同じように、このオブジェクトもプログラム環境からデータを読み込みます。この環境は普通、端末ウインドウまたは (コマンドの) パイプです。

Java には java.io の他にも、いろいろな API のパッケージがあり、その中の java.lang.System クラスには、in という標準入力を参照する InputStream 型のクラス変数が定義されています。このクラスには、out 変数 (標準出力)、err 変数 (標準エラー出力) も定義されています。皆さんはすでにプログラムの中で使っていますね。ちなみに、java.lang.System では次のように定義されています。

```
public final class System extends Object {
    . . .
    //Class Variables
    public static PrintStream err;
    public static InputStream in;
    public static PrintStream out;
    //Class Methods
    ....
}
```

InputStream の read() を使用すると、次のようにして標準入力から 1 バイトずつ読み込むことができます。

```
try {
    int val = System.in.read();
    . . .
}
catch (IOException e) {
    . . .
}
```

read() の返り値は int 型です。返り値が -1 の場合は、正常にストリームの終わりに達したことを表します。読み込み中にエラーが発生すると IOException という例外を投げますから、その処理のための catch 節を書いておく必要があります。

たとえば、標準入力から 1 バイト読むたびに、なにか処理を行なうとすれば、次のようなコードになるでしょう。

```
try {
    while (System.in.read() >= 0) {
        //ここでなにかする
    }
}
catch (IOException e) {
  System.err.println("io error");
  return;
}
```

9.3.3 フィルタ型ストリーム

バイトの読み書き以外の処理は、どうすればいいのでしょうか。ストリームから流れて来たデータに何か処理を加えて流すというストリームをフィルタ型ストリームといいます。java.io には FilterInputStream と FilterOutputStream クラスがあります。それぞれ InputStream と OutputStream のサブクラスです。

表 9.2: フィルタ型ストリーム

入力ストリーム	出力ストリーム	処理
BufferedInputStream	BufferedOutputStream	バッファリングする。
LineNumberInputStream		行番号を数える。
PushbackInputStream		1 バイト先読み (読んだ 1 バイトを元に戻す)。
SequenceInputStream		2 本の入力ストリームを 1 本にする。
	PrintStream	toString() で文字列に直してプリントする。

青柳、「Java API プログラミングガイド」、p. 67 より転載

9.3.4 Print ストリーム

フィルター型ストリームの中でも、PrintStream はよく使うものの一つです。このクラスが提供する print() メソッドを使うと、引数を文字列に変換しストリームに流すことができます。println() は文字列の終端に改行を追加するメソッドです。9.3.2節で示した System クラスの定義にあるように、System.out と System.err は PrintStream オブジェクトです。このようなストリームは、プログラムの処理結果を表示するためによく使われます。PrintStream の定義をみれば、print()、println() が大抵のものならなんでもプリントできることがわかります。

```
public class PrintStream extends FilterOutputStream {
    . . .
    //Public Instance Methods
    . . .
```



```
    public void print(Object obj);
    public synchronized void print(String s);
    public synchronized void print(char[] s);
    public void print(char c);
    public void print(int i);
    public void print(long l);
    public void print(float f);
    public void print(double d);
    public void print(boolean b);
    public void println()
    public synchronized void println(Object obj);
    ...
}
```

9.4 カウンタプログラム

この章では、課題5で利用する端末入出力の機能について説明してきました。それを使って、早速カウンタのテストプログラムを書いてみましょう。処理の大枠は次のようになります。

```
import java.io.*;          // java.io.*のクラスを省略名で
                           // 使用するために必要。

public class Count {
    private final static int DEFAULT_SIZE = 3;
    private final static int DEFAULT_RADIX = 10;
    private final static int DEFAULT_DIR = 1; //カウントアップ
    public static void main(String argv[]) {
        int size = DEFAULT_SIZE;        // 桁数の省略値
        int radix = DEFAULT_RADIX;      // 基数の省略値
        int dir = DEFAULT_DIR;          // カウント方向の省略値

        if(argv.length > 0) {
            ... 省略 ... // 桁数読み込み、セット
        }
        if (argv.length > 1) {
            ... 省略 ... // 基数読み込み、セット
        }
        if(argv.length > 2) {
            ... 省略 ... // カウント方向読み込み、セット
        }

        // カウンタオブジェクト生成
        Counter counter = new Counter(size,radix);

        try {
```

```
            while ( ... 省略 ... //1 バイト読むたびに) {
                if (0 < dir) {
                    counter.inc();      //カウントアップ
                } else {
                    counter.dec();      //カウントダウン
                }
                ... 省略 ... // まず各桁を空白で区切って表示
                ... 省略 ... // そのあとの () にカウンタの値を表示
            }
        }
        catch (IOException e) {
            System.err.println("io error");
            return;
        }
    }
```

第 10 章

練習問題 3

カウンタクラスの構造は把握できたでしょうか？機能の割には複雑に作り過ぎではないかと思った人もいるかもしれません。テキストの例では UnitCounter は一桁でしたが、これを 4 桁や 8 桁に拡張して大きな整数のカウントができるようにするのは容易なことです。動的データ構造の実現の仕方も示唆していますから、カウンタクラスの仕様、カウンタオブジェクトがどのように生成されるかについては、しっかり理解しておいてください。今日は、さらに、端末入出力、コマンドライン引数の使用の仕方についても留意して練習問題をやりましょう。

練習問題 2 - 4: Counter クラス

テキストの説明で省略されているコードを追加し、さらに 16 進数にまで対応できるように変更して、カウンタプログラムを完成してください。全ての機能が一通り確かめられるようなテストケース (テスト項目と予想される結果一覧) を考え、機能を確かめましょう。

整数値を 16 進表記に変換する方法としては、いくつかの方法が考えられます。

- 16 進数文字を格納する char 型の配列を用意する。
- 整数値を直接文字コードに変換する。
- java.lang.Character クラスのメソッドを利用。
- java.lang.Integer クラスのメソッドを利用。