Java 2 Standard Edition 5.0 Tiger
拡張された言語仕様について

『プログラミング言語 Java 第 3 版』補遺

初版第 7 刷

柴田 芳樹 著

2013 年 10 月 25 日

まえがき

Tiger, Tiger burning bright
Like a geek who works all night
What new-fangled bit or byte
Could ease the hacker's weary plight?
– Joshua Bloch

Tiger と呼ばれて開発された Java 2 Standard Edition 5.0 では、プログラミング言語 Java に対する言語仕様の拡張が行われています。その仕様拡張は、プログラミング言語としての表現力の改善を行っており、Java 言語でソフトウェアを開発する上で、その仕様を理解することは重要です。

言語仕様の拡張は、プログラムの書き方が変わるだけでなく、多くのツールが影響を受けます。そのため、主要なリリースごとに言語仕様が変更になるべきではありません。1.1 でインナークラスが導入され、1.4 で `assert` が導入されたのですが、5.0 では多くの言語仕様の拡張が行われています。Java 言語の最初のリリース以来、5.0 はおそらく最も大規模な言語仕様の拡張であると言えます。そのため、ツールが対応して、新たな言語仕様が広く使用されるには時間を要していますし、今後これほどの言語仕様の拡張は行われないと思われます。

本書は、1.4 までの Java 言語に習熟した人を対象として、5.0 で追加された機能を解説するものです。したがって、Java 言語の初心者を対象としたものではありませんので注意してください。本文では、1.4 をリリース 1.4 と呼び、5.0 をリリース 5.0 と呼んで記述してあります。

本書で解説されるリリース 5.0 の Java 言語仕様拡張は、以下の通りです。

- **ジェネリックス**： クラスやインタフェースに**型パラメータ**（*type parameter*）を導入する機能です。
- **ボクシング/アンボクシング**： 基本データ型と対応するラッパークラス間の変換を自動的に行う機能を提供します。
- **拡張 `for` 文**： `for` 文に新たなシンタックスが導入されます。
- **`enum` 型**： C 言語/C++言語にある `enum` に似ていますが、さらにオブジェクトとしての機能を提供します。
- **static インポート**： static メンバーをインポートするための機能を提供します。
- **可変長パラメータ**： 可変長のパラメータを宣言する機能を提供します。
- **アノテーション**: クラス、メソッド、フィールドなどの要素にアノテーション（注釈）を付ける機能を提供します。

本書は、『プログラミング言語 Java 第 3 版』[Arnold00] を補完することを目的としています。第 3 版は、リリース 1.3 に基づいていますので、本書では、リリース 5.0 での言語仕様の拡張を中心に、リリース 1.4/5.0 の差分を説明します。本書は以下の章から構成されています。

第 1 章「ジェネリックス」では、ジェネリックスに関して、文法を説明するだけでなく、コンパイラがどのようなことを行っているかを説明することで、ジェネリックスを説明します。

第 2 章「ボクシング/アンボクシング」では、基本データ型と対応するラッパークラス間の変換を行うボクシング/アンボクシングを説明します。

第 3 章「拡張 `for` 文」では、より簡単なループの書き方ができる拡張 `for` 文を説明します。

第 4 章「`enum` 型」では、リリース 1.4 までの手作業によるタイプセーフ enum と比較しながら、`enum` 型を説明します。

第 5 章「static インポート」では、static メンバーをインポートするための static インポートを説明します。

第 6 章「可変長パラメータ」では、可変長のパラメータを説明します。

第 7 章「アノテーション」では、標準のアノテーション、アノテーション型の宣言方法、および、アノテーションを定義する場合に使用するメタアノテーションを説明します。

第 8 章「アノテーション処理プログラミング」では、ソースコードに記述されたアノテーションを処理するためのリフレクション API と JDK（*Java Development Kit*） に含まれる apt（*Annotation Processor Tool*）コマンド の簡単な使用方法を説明します。

第 9 章「基本パッケージ」では、基本パッケージに関して、リリース 1.4 およびリリース 5.0 で変更になった部分を説明します。基本的に、『プログラミング言語 Java 第 3 版』の章立てに従って、各節が構成されています。

第 1 章から第 8 章までは、（株）技術評論社の書籍『Java PRESS, Vol.37』および『Java PRESS, Vol.38』に執筆した記事を基にして、追加および修正を行ったものです。

コンパイルオプションについて

コンパイラの次の 2 つのオプションについて注意が必要です。

`-source` ソースコードの言語仕様の J2SE バージョンを指定します。
`-target` コンパイル結果のクラスファイル仕様の J2SE バージョンを指定します。

デフォルトでは、`-source 1.5` と`-target 1.5` を指定したことになります。 `-target 1.5` が指定されていますので、コンパイル結果のクラスファイルを、リリース 1.4 上で動作させることはできません。したがって、リリース 1.4 までのプログラムをリリース 1.4 上で動作させる必要がある場合には、`-source 1.4` と`-target 1.4` の両方を指定してコンパイルする必要があります。

謝辞

新たな言語仕様や技術的不明点に関して、いつも迅速で丁寧な回答を頂いた Joshua Bloch、Neal Gafter、David Holmes の各氏に深く感謝します。 Joshua Bloch 氏には、CAP [*1] へ私を紹介して頂き、微力ながら一人の技術者として J2SE 5.0 へ貢献する機会を与えて頂いたことに感謝します。さらに、Joshua Bloch 氏には、詩の引用およびサンプルコードの引用を快諾してくださったことに感謝します。

本書をレビューして助言を頂いた荒井玲子、木南英夫、黒川裕之、相馬純平、中村正、難波亮丞、皆本房幸の各氏に深く感謝します。また、Joshua Bloch 氏の詩[*2] を翻訳してくださった野崎梨絵さんに感謝します。

1999 年暮れに Java に関する記事を執筆する機会を与えて頂き、その後も、様々な執筆の機会を与えて頂いている（株）技術評論社の谷戸伸好、伴達也、鈴木秀光の各氏に感謝します。『プログラミング言語 Java 第 3 版』および『Effective Java プログラミング言語ガイド』を翻訳する機会を与えて頂き、今回は本書を出版する機会を与えて頂いた（株）ピアソン・エデュケーションの藤村行俊氏に深く感謝します。

最後に、校正を手伝ってくれた私の妻恵美子に感謝します。

柴田 芳樹

2005 年 3 月

Tiger、*Tiger*、あかあかと燃える
夜通し働き詰めのコンピュータ狂のように。
新奇のビットやバイトのどんなものが
ハッカーのうんざりを和らげてくれるだろうか？
– *Joshua Bloch*

[*1] J2SE Compatibility and Performance Program

[*2] Tiger と呼ばれた J2SE 5.0 に関して、2003 年に Joshua Bloch 氏が書いた詩です。この詩は、英国人の詩人である William Blake（1757-1827）の “The Tyger” に基づいています。第 6 章「可変長パラメータ」の詩は、2003 年に書かれた Joshua Bloch 氏のオリジナルの詩には含まれていませんでしたが、2004 年 6 月に特別に作成してもらいました。第 1 章から第 7 章までの各章の初めにオリジナルの英語の詩を掲載し、章末に翻訳を掲載しています。

目次

第 1 章

ジェネリックス

To the most despised collections' cast
We'll bid a fond farewell at last
With generics' burning spear
The need for cast will disappear
– Joshua Bloch

総称性（*genericity*）あるいは**ジェネリックス**（*generics*）と呼ばれる機能は、**型パラメータ**（*type parameter*）を明示的に示して、型のキャストをコンパイラにより安全に行うことで、Java 言語での表現力と安全性を向上させるものです。特に、コレクションなどのクラスライブラリを柔軟に、かつ、安全に使用するためには、重要な機能です。

Java 言語の生みの親である James Gosling 氏は、ジェネリックスに関して次のように述べています。

This has been high on my wish list since before the first release of Java.
最初に Java がリリースされる前から、私が欲しい機能のリストの上位に、ジェネリックスはあった。
– James Gosling

ジェネリックスに関しては、Java 言語が登場して以来、様々な議論が行われて、Gilad Bracha 氏 を中心として言語仕様が決まり、Joshua Bloch 氏 が中心となって策定した他の言語仕様と上手く調和が取られて仕様が出来上がっています。

1.1 コレクションの操作

Java 言語で、ある型の集合を扱うには、いくつかの方法があります。１つは、単純にその型の配列を宣言する方法です。しかし、基本的にすべてのプログラムが必要とする処理を、配列だけで実現することはできません。そのために、**コレクションフレームワーク**（*Collection Framework*）が用意されています。コレクションフレームワークでは、コレクション内に保存されたオブジェクトを取り出す場合、リリース 1.4 までは `Object` クラスのインスタンスとして取り出されるため、必ずキャストする必要がありました。次のコードは、`LinkedList` クラスを使用した簡単なテストプログラムです。

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
```

```
class LinkedListTest { // 1.4 version
    public static void main(String[] args) {
        List list = new LinkedList();
        list.add(new Integer(10));
        list.add(new Integer(20));

        int total = 0;
        for (Iterator i = list.iterator(); i.hasNext(); ) {
            total += ((Integer) i.next()).intValue(); // キャストが必要
        }
        System.out.println("total = " + total);
    }
}
```

このプログラムでは、リストからイテレータを使用して要素を取り出す場合に、キャストを必要としています。仮に、誤ったキャストを行っても、コンパイル時には何も警告がでることなくコンパイルできます。しかし、実行時に ClassCastException がスローされます。

このプログラムをジェネリックスを使用して書き直したものが、次のコードです。

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

class LinkedListTest { // 5.0 version
    public static void main(String[] args) {
        List<Integer> list = new LinkedList<Integer>(); // *
        list.add(new Integer(10));
        list.add(new Integer(20));

        int total = 0;
        for (Iterator<Integer> i = list.iterator(); i.hasNext(); ) { // *
            total += i.next().intValue(); // キャストが不要
        }
        System.out.println("total = " + total);
    }
}
```

ローカル変数 list は、任意のオブジェクトを取り扱う List ではなく、Integer を取り扱うリストであるとして、List<Integer>と宣言されています。イテレータから要素を取り出す場合に、キャストすることなく取り出していることにも注目してください。for 文内で、Iterator<Integer>と宣言していますが、これを Iterator<String>とすると、i = list.iterator() の代入がコンパイルエラーになります。なぜなら、list は Integer のリストなので、その iterator メソッドで返されるイテレータは、Iterator<Integer>型だからです。

ボクシング/アンボクシング（第 2 章）および拡張 for 文（第 3 章）を使用すると、Iterator を使用することなく、次の通り、さらに簡潔に書くことができます。

```
import java.util.LinkedList;
import java.util.List;

class LinkedListTest { // 5.0 version
    public static void main(String[] args) {
        List<Integer> list = new LinkedList<Integer>();

        list.add(10); // ボクシング
        list.add(20); // ボクシング

        int total = 0;
        for (Integer value: list)  // 拡張 for 文
            total += value; // アンボクシング
        System.out.println("total = " + total);
    }
}
```

このようにジェネリックスを使用して、コレクションが取り扱う型を明示的に指定することで、**実行時ではなく、コンパイル時に安全に型の検査が可能となり、コードも簡潔になります**。誤った型操作を行うプログラムを書いた場合、リリース 1.4 では実行するまでは誤った型操作を行っていることが分からない場合が多かったのですが、ジェネリックスを使用することで、コンパイル時にそのほとんどがコンパイラにより発見され、ソフトウェアの生産性を向上させることになります。

ジェネリックスにより、コレクションフレームワークの使用方法がどのように変わるかについて簡単に説明します。コレクションを生成する場合には、どのような型のオブジェクトを扱うコレクションかを指定して生成することになります。ほとんどのプログラムでは、コレクションに複数の型のオブジェクトを入れるのではなく、特定の型のオブジェクトしか入れないことを考慮すれば、その特定の型を明示的に指定することは、プログラムの可読性を向上させます。たとえば、Integer 型の ArrayList であれば、次のように生成します。

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

このリストは、Integer 型のオブジェクトを扱いますので、要素を取り出す際に、キャストすることなく、次のように要素を取り出すことができます。

```
Integer i = list.get(0);
```

あるいは、生成したコレクションを ArrayList ではなく、List インタフェースとして扱いたければ、次のように書くこともできます。

```
List<Integer> list =  new ArrayList<Integer>();
```

このように生成した list から取り出すイテレータは、次のように取り出します。

```
Iterator<Integer> iter = list.iterator();
```

取り出したイテレータの next() メソッドは、Integer 型の参照を返しますので、キャストすることなく、次のように使用することができます。

```
Integer i = iter.next();
```

あるいは、元のリストから次のように直接取り出すことも可能です（ただし、要素が一つは入っていることが前提です）。

```
Integer i = list.iterator().next();
```

1.2 ジェネリック化された型と型変数

ジェネリックスでは、新たな型の形式として、**ジェネリック化された型**（*generic type*） と**型変数**（*type variable*） が導入されます。ジェネリック化された型とは、**型パラメータ**（*type parameter*） リストを持つクラスやインタフェースを指します。型パラメータリストは、クラス宣言やインタフェース宣言において、クラス名あるいはインタフェース名の後に次のように型変数を指定することで表現されます。

```
class クラス名<型変数> { /* ... */ }
interface インタフェース名<型変数> { /* ... */ }
```

あるいは、型変数をカンマ（,）で区切ることで、複数指定することも可能です。

```
class クラス名<型変数 1, 型変数 2> { /* ... */ }
interface インタフェース名<型変数 1, 型変数 2> { /* ... */ }
```

ジェネリック化されたクラスとインタフェースは、**ジェネリック**（*generic*）であると言い、そのようなクラスやインタフェースを、**ジェネリッククラス**（*generic class*） および**ジェネリックインタフェース**（*generic interface*）と呼びます。 ただし、Throwable クラスを直接あるいは間接的に継承しているクラスを、ジェネリック化してジェネリック例外を作成することはできません。

では、簡単な Stack クラスを作成してみます。まず、型変数を使用しないで普通に実装したのが、次のコードです。StackTest クラスは、テストのためのクラスです。

```
import java.util.EmptyStackException;

class Stack {
    private Object[] elements;
    private int size = 0;

    public Stack(int initialCapacity) {
        elements = new Object[initialCapacity];
    }
```

```
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null;
        return result;
    }

    private void ensureCapacity() {
        if (elements.length == size) {
            Object[] oldElements = elements;
            elements = new Object[2 * elements.length + 1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}

class StackTest {
    public static void main(String[] args) {
        Stack s = new Stack(args.length);

        for (int i = 0; i < args.length; i++)
            s.push(new Integer(args[i]));

        int total = 0;
        for (int i = 0; i < args.length; i++)
            total += ((Integer) s.pop()).intValue();

        System.out.println("total = " + total);
    }
}
```

任意の型を扱うために、Object への参照型を使用しています。このクラスを、型変数を使用して書き直したものが、次のコードです。

```
import java.util.EmptyStackException;

class Stack<E> {                // [1]
    private E[] elements;       // [2]
    private int size = 0;

    public Stack(int initialCapacity) {
```

```
            elements = (E[]) new Object[initialCapacity]; // [3]
        }

        public void push(E e) {     // [4]
            ensureCapacity();
            elements[size++] = e;
        }

        public E pop() {            // [5]
            if (size == 0)
                throw new EmptyStackException();
            E e = elements[--size];    // [6]
            elements[size] = null;
            return e;
        }

        private void ensureCapacity() {
            if (elements.length == size) {
                E[] oldElements = elements;
                elements = (E[]) new Object[2 * elements.length + 1]; // [7]
                System.arraycopy(oldElements, 0, elements, 0, size);
            }
        }
    }

    class StackTest {
        public static void main(String[] args) {
            Stack<Integer> s = new Stack<Integer>(args.length);  // [8]

            for (int i = 0; i < args.length; i++)
                s.push(new Integer(args[i]));

            int total = 0;
            for (int i = 0; i < args.length; i++)
                total += s.pop(); // [9]

            System.out.println("total = " + total);
        }
    }
```

このプログラムの重要な箇所に番号を付けてありますので、それらの箇所について解説します。

[1] クラスの型変数 E を宣言しています。

[2] 配列の要素を保持する elements を、型変数 E を使用して宣言しています。

[3] 型変数 E を使用して new E[initialCapacity] と書いて配列を生成することはできません (1.2.2 節参照)。したがって、Object クラスの配列を生成して、E[] へキャストしています。

[4] push メソッドの引数の型として、型変数 E が指定されています。
[5] pop メソッドの戻り値型として、型変数 E が指定されています。
[6] 一時変数である e の型として、型変数 E が指定されています。
[7] ensureCapacity メソッド内で、配列を大きくする処理で、Object クラスの配列を生成して、E[] へキャストしています。
[8] Stack クラスで使用する型変数 E に対して、Integer クラスを指定して生成しています。
[9] pop メソッドで Integer クラスのインスタンスを取り出して、直接 total に加算しています。実際にはコンパイラが Integer クラスの intValue メソッドを呼び出して、その結果を加算するバイトコードを生成します（アンボクシング）。

このようにリリース 5.0 からは、取り扱う型を型変数として宣言するプログラミングが可能となります。なお、E[] へのキャストを行うと、コンパイラにより警告メッセージが表示されます。[*1]

[8] で、ローカル変数 s の型として、ジェネリッククラスである Stack の型変数を指定した Stack<Integer>は、**パラメータ化された型**（*parameterized type*）と呼びます。

この例のように、ジェネリッククラスやジェネリックインタフェースで定義されている型変数は、フィールド宣言、メソッドの戻り値型、メソッドやコンストラクタのパラメータ宣言、ローカル変数宣言、ネストした型宣言内で使用することができます。

1.2.1 型変数の命名規約

型変数には、大文字や小文字、複数文字から構成される単語など何を使用しても構わないのですが、一般的な命名規約として、以下の規約があります。

- 可能なら一文字の大文字を使用し、小文字は避ける。
- コレクション内の要素型を表す場合には、element の意味で、E を使用する。
- マップ内のキー型と値型を表す場合には、key と value の意味で、それぞれ、K と V を使用する。
- 一般的な型の場合には、type の意味で、T を使用する。さらに必要な場合には、アルファベットで T に隣接する S などを用いる。

Java 言語には 6 つの名前空間[*2]があります。型変数は、型を必要とする場所で使用しますので、**型名**の名前空間に属します。したがって、型変数名として既存の型、たとえば、Integer を使用すると、それは、java.lang.Integer を指すのではなく、単なる型変数名として扱われますので注意してください。

[*1] 型変数へのキャストは、警告メッセージが出力されます。@SuppressWarnings アノテーション（7.1.3 節参照）を用いて警告メッセージの出力を抑制できます。
[*2] パッケージ名、型名、フィールド名、メソッド名、ローカル変数名（パラメータを含む）、ラベル

1.2.2 型変数に対する制約

実際に生成されるバイトコードに関しては、Stack クラス（5 頁）の型変数 E は、Object としてコンパイルされています。ジェネリックスを用いて作成した Stack クラスのコンパイル結果を、javap コマンドで調べてみると、次の結果となり、Object としてコンパイルされていることが分かります。[*3]

```
D:\Tiger\example\generics>javap -private Stack
Compiled from "Stack.java"
class Stack extends java.lang.Object{
    private java.lang.Object[] elements;
    private int size;
    public Stack(int);
    public void push(java.lang.Object);
    public java.lang.Object pop();
    private void ensureCapacity();
}
```

型変数 E を用いた StackTest クラスでは、キャストは必要としていませんが、実際には、キャストを行うバイトコードをコンパイラが生成します。手作業でキャストするコードを入れる場合には、間違ったキャストを書く可能性が残ってしまうのですが、ジェネリックスを使用すると、実行時エラーにならないことが保証されたキャストを行うバイトコードを、コンパイラが自動的に生成します。

ジェネリッククラスやジェネリックインタフェースであっても、このように、コンパイラによって型パラメータは削除されて、必要なキャストの挿入と、型が Object になってしまうような処理を**イレイジャ**（*erasure*）と呼びます。

しかし、制約もあります。それらの制約は、ジェネリックスを用いて書かれたプログラムが、どのようにコンパイルされるかを知っていないと理解できない場合があります。Stack クラスでの型変数 E は、Object としてコンパイルされると説明しましたが、その結果、次の制約が出てきます。

【制約 1】型変数 E に対して、基本データ型を指定することはできない。[*4]
【制約 2】型変数 E のオブジェクトを、直接生成することはできない。
【制約 3】型変数 E の配列を生成することはできない。
【制約 4】メソッドの戻り値として型変数 E の配列を返すようにしても、Object の配列となる。
【制約 5】型変数は、static メンバーと static 初期化子には適用できない。
【制約 6】イレイジャ処理の結果、メソッドのシグニチャが同じになるメソッドは定義できない。

制約 1 は、Stack クラスの例でいえば、次のようなコードは書けないということです。

```
Stack<int> s = new Stack<int>(10); // NG! コンパイルエラー
```

[*3] C++言語のテンプレート機能のように、型変数に対して、実際の型が指定されるごとに、その型を適用したコードが生成されることはありません。
[*4] 配列は参照型ですので、基本データ型の配列は、Stack<int[]>のように指定することができます。

しかし、実際には、ボクシングとアンボクシング機能により、次のようにあたかも int のスタックかのごとく使用することが可能です。

```
Stack<Integer> s = new Stack<Integer>(10);
s.push(10);
int y = s.pop();
```

制約 2 は、型変数 E を宣言している Stack クラス内で、つぎのように型変数 E 型のオブジェクトを直接生成できないということです。

```
E e = new E(); // NG! コンパイルエラー
```

イレイジャ処理の結果、型変数 E は、Stack クラスの例では、すべて Object に置き換えられています。仮に、new E() ができるとしたら、Stack<Integer>型であっても、Integer クラスのインスタンスが生成されるのではなく、Object クラスのインスタンスが生成されてしまうことになります。

制約 3 は、型変数 E の配列を直接生成できないことを意味します。もし、直接生成できるとしたら、次のようなコードを書くことができてしまいます。

```
public E[] toArray() {
    E[] array = new E[size]; // 実際にはできないが
    for (int i = 0; i < size; i++)
       array[i] = elements[i];
    return array;
}
```

このようにコードが書けるとしたら、次のようなコードを書いて実行すると正しく実行できないといけないことになってしまいます。

```
Stack<Integer> s = new Stack<Integer>(10);
// ...
Integer[] array = s.toArray();
```

この場合、制約 4 にあるように、実際に toArray メソッドが返すのは、Interger[] ではなく、Object[] ですので、コンパイラは次のようにキャストを行うコードを生成することになります。

```
Integer[] array = (Integer[])s.toArray();
```

しかし、仮に new E[size] が許されたとしても、生成されるのは Object[] 型であり、コンパイラが挿入した Integer[] 型へのキャストは正しくないことになり、実行時に ClassCastException がスローされることになります。これでは、ジェネリックスにおける「コンパイラが安全なキャストを自動的に挿入するという方針」に反してしまいます。そのため、型変数 E の配列を生成することはできないという制約 3 が課されています。

したがって、Stack クラス内で型変数 E の配列と宣言されている elements を、E[] 型としてクラスの外に返さないようにしなければなりません。どうしても内部で保持している要素を配列として返したければ、次のように toArray メソッドを実装する必要があります。

```
public E[] toArray(E[] a) {
    if (a.length < size) {
        a = (E[]) java.lang.reflect.Array.newInstance(
                        a.getClass().getComponentType(),
                        size);
    }
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

引数として、E[] 型の配列への参照を渡すように修正されています。その配列は、Object[] 型ではなく、正しい型の配列が渡されます。たとえば、次のようなコードになります。

```
Integer[] stackContents = s.toArray(new Integer[0]);
```

渡された配列 a の大きさが、スタックの大きさよりも小さい場合には、新たに正しい型の配列を生成して、要素をコピーしています。渡された配列 a の方が、スタックよりも大きければ、要素をコピーした後に、null を代入することで、要素の終わりを表しています。このような toArray メソッドを追加するのであれば、null を受け付けないことになりますので、push メソッドで null が渡されたら、NullPointerException をスローするように修正する必要があります。

制約 5 は、あるクラスの 2 つのインスタンスは、型変数として異なる型が指定されているかもしれないので、同一の static メンバーに対するアクセスは、異なる型へのアクセスとなり、禁止されているということです。たとえば、次のようなクラスが定義できたとします。

```
class GlobalStack<E> {

    private static List<E> stack = new ArrayList<E>();

    public void push(E e) { stack.add(e); }
    public E pop()        { return stack.remove(0); }
}
```

もし、このようなコードが書けたとしたら、次のようなコードも書けてしまいます。

```
class Foo {
    GlobalStack<Foo> fooStack = new GlobalStack<Foo>();
    // ...
}

class Bar {
    GlobalStack<Bar> barStack = new GlobalStack<Bar>();
    // ...
}
```

あるクラスの static フィールドは、そのクラスの Class オブジェクト内に存在します。一方、GlobalStack<Foo>と GlobalStack<Bar>の Class オブジェクトは同一（1.8.3 節参照）であり、stack フィールドはシステム内に、1 つしか存在しません。その結果、GlobalStack クラスの stack フィールドが参照しているリストには、異なった 2 つの型のインスタンスが混在して入ることになってしまい、型の安全性が失われることになってしまいます。

制約 6 は、次のようなコードを書くことができないということです。

```
class Foo<T, S> {
    public void add(T t) { /* ... */ }  // NG! コンパイルエラー
    public void add(S s) { /* ... */ }  // NG! コンパイルエラー
}
```

この場合、2 つの add メソッドは、イレイジャ処理により、同一のシグニチャのメソッドとなり、コンパイルエラーとなります。このように、ジェネリック型（ジェネリッククラスまたはジェネリックインタフェース）のメソッドがオーバーロードやオーバーライドが可能か否かは、イレイジャ処理後のメソッドのシグニチャによって決まることに注意してください。なお、オーバーロードとオーバーライドに関しては、1.7 節で説明します。

1.2.3 リリース 1.4 との互換性

型変数 E を用いて書かれた Stack クラスをコンパイルした結果のクラスファイルは、リリース 5.0 より前のリリース 1.4 などでは実行できませんし、javac コマンドのオプションである-source 1.5 と -target 1.4 を一緒に用いてコンパイルすることもできません。ジェネリックスを使用したプログラムは、リリース 1.4 ではサポートされていないクラスファイル形式になるため、リリース 1.4 では動作させることができません。

1.2.4 ジェネリック化されたコンストラクタ宣言

クラス全体ではなく、コンストラクタだけをジェネリック化することもできます。コンストラクタの場合には、コンストラクタ名の前で、**<型変数>**の宣言を行います。

```
public class Foo {
    public <T> Foo(T x) { /* ... */ }
}
```

このようにジェネリック化されたコンストラクタを**ジェネリックコンストラクタ**（*generic constructor*）と呼びます。

1.2.5 ジェネリック化されたメソッド宣言

クラス全体ではなく、メソッドだけをジェネリック化することもできます。メソッドの場合には、メソッドの戻り値型の前で**<型変数>**の宣言を行います。たとえば、配列の特定の要素を入れ替えるメソッ

ドと配列を表示するメソッドを実装したのが、次のプログラムです。

```
class Util {

    static <T> void swap(T[] a, int i, int j) {
        T temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

    static <S> void print(S[] a) {
        for (S s : a)
            System.out.println(s + " ");
    }

    public static void main(String[] args) {
        Integer[] intArray = new Integer[args.length];

        for (int i = 0; i < args.length; i++)
            intArray[i] = new Integer(args[i]);
        swap(intArray, 0, intArray.length - 1);
        print(intArray);
    }
}
```

swap メソッドと print メソッドの呼び出しでは、型変数 T と S は、Integer が指定されたと見なされます。このようにジェネリック化されたメソッドを**ジェネリックメソッド**（*generic method*） と呼びます。

1.2.6 ジェネリック化されたインタフェース宣言

インタフェースのジェネリック化は、クラスと同様に、インタフェース名の後に型変数宣言を行います。たとえば、Comparable インタフェースの宣言は、リリース 5.0 では次のようになっています。

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

ジェネリック化されたインタフェースを実装する場合には、型変数を指定する必要があります。次のプログラムは、このジェネリック化された Comparable インタフェースの実装例です。

```
import java.util.Arrays;

class EmployeeName implements Comparable<EmployeeName> { // [1]
    public final String firstName;
    public final String lastName;
```

```
    public EmployeeName(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public int compareTo(EmployeeName that) {          // [2]
        if (lastName.compareTo(that.lastName) != 0)
            return lastName.compareTo(that.lastName);
        return firstName.compareTo(that.firstName);
    }

    public String toString() {
        return firstName + " " + lastName;
    }

    public static void main(String[] args) {
        EmployeeName[] names = {
            new EmployeeName("Yoshiki", "Shibata"),
            new EmployeeName("David", "Nesbitt"),
            new EmployeeName("Steve", "Bartlett") };

        Arrays.sort(names);
        for (int i = 0; i < names.length; i++)
            System.out.println(names[i]);

    }
}
```

番号が付けられている重要な箇所について解説します。

[1] Comparable インタフェースに対する型変数として、自分自身のクラス名を指定しています。

[2] 自分自身のクラス名を Comparable インタフェースの型変数として指定していますので、Comparable インタフェースで定義された compareTo メソッドのパラメータ型として、自分自身のクラス名を指定しています。

このプログラムでは、Arrays.sort メソッドを使用してソートを行っています。ここで注意してほしいのは、Arrays.sort メソッドが引数で受け取る配列のオブジェクトは、従来の Comparable インタフェースのメソッドである、次のメソッドを実装したオブジェクトでなければなりません。

```
public int compareTo(Object o);
```

EmployeeName クラスでは、このようなメソッドを定義していませんが、コンパイルして実行してみると、正しく動作します。EmployeeName クラスがどのような仕組みでうまく実行されるかについては、1.3 節で説明します。

1.2.7 型変数のメソッド呼び出し

型変数を使用して宣言されたパラメータ変数に対して、メソッドを呼び出すためには、その型変数が明らかにそのメソッドを持っていることが、コンパイル時に判断できなければなりません。そのためには、その型変数が何らかのクラスを継承しているか、何らかのインタフェースを実装しているかを明示することが必要で、それによって、それらのクラスやインタフェースで定義されているメソッドを呼び出すことが可能となります。

型変数<T>の宣言は、暗黙に<T extends Object>と宣言しているのと同じなので、Object クラスで定義されている public のメソッドを呼び出すことができます。

型変数に対しては、より具体的に継承しているクラスや実装しているインタフェースを指定することが可能です。次のプログラムでは、max メソッドを定義しています。

```
import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;

class BoundTest {

    public static <T extends Comparable<T>> //[1]
      T max(Collection<T> coll) {  // [2]
        Iterator<T> i = coll.iterator();
        T candidate = i.next();

        while(i.hasNext()) {
            T next = i.next();
            if (next.compareTo(candidate) > 0) // [3]
                candidate = next;
        }
        return candidate;
    }


    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();

        for (int i = 0; i < args.length; i++)
            list.add(new Integer(args[i]));

        System.out.println("max = " + max(list)); // [4]
    }
}
```

番号が付けられている重要な箇所について解説します。

[1] 型変数として T を宣言していますが、それ自身は Comparable<T>を実装している[*5]と宣言しています。
[2] max メソッドの戻り値型を T と宣言し、引数の型を Collection<T>と宣言しています。
[3] 型変数 T は、Comparable<T>を実装していると宣言されていますので、compareTo メソッドを呼び出して、比較を行っています。
[4] max メソッドの呼び出しで、型変数 T が何になるかは、max メソッドの引数により決定されます。

型変数に対して、extends を使用して、どのクラスから継承されているか、あるいはどのインタフェースを実装しているかを指定した場合には、指定された型を扱うクラスとしてコンパイルされます。したがって、max メソッドの場合には、イレイジャ処理の結果として、型変数が Object に置き換えられるのではなく、Comparable に置き換えられます。つまり、あたかも次のプログラムを書いたかのようにコンパイルされます。

```
import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;

class BoundTest {

    public static Comparable max(Collection coll) {
        Iterator i = coll.iterator();
        Comparable candidate = (Comparable)i.next();

        while(i.hasNext()) {
            Comparable next = (Comparable)i.next();
            if (next.compareTo(candidate) > 0)
                candidate = next;
        }
        return candidate;
    }

    public static void main(String[] args) {
        ArrayList list = new ArrayList();

        for (int i = 0; i < args.length; i++)
            list.add(new Integer(args[i]));

        System.out.println("max = " + ((Integer)max(list)));
    }
}
```

[*5] インタフェースであっても、extends と書きます。たとえば、class Foo<T1, T2 extends T1>と宣言されたら、実際に T1 として指定される型がインタフェースなのか、クラスなのかは分かりません。したがって、すべて extends と書き、implements は使用されません。

実際にメソッドのシグニチャを javap コマンドで、調べてみると次のような結果となります。

```
D:\Tiger\example\generics>javap BoundTest
Compiled from "BoundTest.java"
class BoundTest extends java.lang.Object{
    BoundTest();
    public static java.lang.Comparable max(java.util.Collection);
    public static void main(java.lang.String[]);
}
```

実は、このジェネリック化した max メソッドには問題があります。それは、実際に生成された max の戻り値型が、Comparable というのは正しくありません。コレクションからの最大値の要素を探して返すのですから、その型が Comparable では意味的におかしいことになります。正しくは、Object を返す必要があります。そのためには、次のように型変数の宣言を変更します。

```
public static <T extends Object & Comparable<T>>
      T max(Collection<T> coll) { /* ... */ }
```

この変更により、実際にコンパイラにより生成される max メソッドの戻り値型は、イレイジャ処理の結果、Object になります。仮に、max メソッドの戻り値型を型変数 T ではなく、直接 Object 型と宣言すると、max メソッドを使用する場合には、常に明示的なキャストが必要になってしまうことに注意してください。

このように、型変数に対して、&を使用して、複数のインタフェースを実装していることを明示することができます。

```
<T extends A & B & C>
```

これは、型変数 T は、A を継承もしくは実装し、B と C を実装していることを示します。もちろんインタフェースはいくつも指定できますが、クラスは 1 つしか指定できませんし、最初に書かれる A がクラスでなければなりません。このように複数の型が指定されている場合には、実際に型変数 T が使用されている部分は、A 型としてコンパイルされます。残りの B 型や C 型のメソッド呼び出しが書かれた場合には、コンパイラがキャストを自動的に挿入して、それらの型固有のメソッドを呼び出すバイトコードを生成します。

もし、A、B、C がすべてインタフェースの場合には、どれを A として最初に書くかについては注意が必要です。決して Serializable インタフェースなどのマーカーインタフェースを最初に書かないように注意してください。マーカーインタフェースを最初に書くと、後に書かれたインタフェースのメソッドを呼び出す部分では、すべてキャストが挿入されることになります。

既存のクラスやインタフェースをジェネリック化する場合には、イレイジャ処理の結果のメソッドのシグニチャと戻り値型 が、ジェネリック化する前と同じになるようにしなければなりません。そうしないと、それらのクラスやインタフェースを使用しているクライアントのコードのリコンパイルが必要となります。

1.2.8 型を明示したジェネリックメソッド/コンストラクタの呼び出し

ジェネリックメソッドでは、明示的に型パラメータの型を指定する必要がある場合があります。たとえば、次のコードを見てください。

```
class MethodInvocation {

    public <T extends Comparable<T>> void foo(T f) {
        System.out.println("the first foo is invoked");
    }

    public <T> void foo(T f) {
        System.out.println("the second foo is invoked");
    }

    public static void main(String[] args) {
        MethodInvocation m = new MethodInvocation();
        m.foo(1);
        m.<Object>foo(1);
    }
}
```

この場合、2 つの foo メソッドは、イレイジャ処理の結果、異なったシグニチャになります。最初の foo メソッドの引数の型は Comparable であり、2 つ目の foo メソッドでは Object となります。

m.foo(1) による foo メソッドの呼び出しでは、値 1 をボクシングにより Integer クラスのインスタンスへ変換後、最初の foo メソッドが呼び出されるべきメソッドとして選択されます。ここで、2 つ目のメソッドを選択して呼ぶには、明示的に型パラメータの型を指定する必要があります。型の明示は、メソッド名の前に<型名>と指定します。この場合、ジェネリックメソッド宣言で宣言された型パラメータに対応した型を、すべて指定する必要があります。なお、メソッド内から、同じクラスあるいはスーパークラスのジェネリックメソッド宣言されているメソッドを呼び出す場合に、<型名>を指定する際には、その前に必ず this. あるいは super. を付ける必要があります。

ジェネリックメソッドと同様に、ジェネリックコンストラクタを呼び出す場合には、明示的に型パラメータの型を指定することができます。

```
public class Foo {
    public <T> Foo(T x) { /* ... */ }

    public static void main(String[] args) {
        Foo f = new <String>Foo("Hello");
    }
}
```

この例では、型変数 T に対して、明示的に String を指定して、Foo クラスのコンストラクタを呼び出しています。

1.2.9 例外

型変数として宣言された例外は、catch 節には書くことはできません。catch 節には、必ず明示的に例外を記述しなければなりません。一方で、throws リストには、型変数として宣言された例外を記述することができます。たとえば、次のように Job インタフェースが定義されていたとします。

```
interface Job<E extends Exception> {
    void run() throws E;
}
```

run メソッドは、Exception クラスのサブタイプである、型変数 E で表される例外をスローすると定義されています。さらに、この Job インタフェースを実装したオブジェクトを実行するためのクラスとして、次の JobExecutor クラスが定義されていたとします。

```
class JobExecutor {
    public static <E extends Exception> void execute(Job<E> job) throws E {
        job.run();
    }
}
```

static のジェネリックメソッド execute が、特定の例外ではなく、型変数 E で表される例外をスローすると宣言されていることに注意してください。

ここで次のようなテストコードを書いたとします。

```
import java.io.FileNotFoundException;

class Test {
    public static void main(String[] args) {
        JobExecutor.execute(new Job<FileNotFoundException>() {
            public void run() throws FileNotFoundException {
                /* ... */
            }
        });
    }
}
```

このコードをコンパイルすると、次のようにコンパイルエラーとなります。

```
D:\Tiger\example\generics>javac Test.java
Test.java:6: 例外 java.io.FileNotFoundException は報告されません。スローするには
キャッチまたは、スロー宣言をしなければなりません。
        JobExecutor.execute(new Job<FileNotFoundException>() {
                            ^
エラー 1 個
```

これは、execute メソッドがチェックされる例外である FileNotFoundException をスローするので、キャッチしなければならないという意味のエラーです。execute メソッドの宣言では、型変数で表さ

れる例外をスローすると宣言しているだけなのですが、実際の例外の型が明示された場合に、その例外の型をスローすると宣言されたかのように見なされて、catch 節を記述することが強制されることになります。

この機能は、一見すると便利なように見えますが、実際には run メソッドでスローできる例外を 1 つに限定してしまうことになってしまうため、汎用的なインタフェースを設計する上では、あまり実用的でないかもしれません。

1.3 ブリッジメソッド

ジェネリック化された Comparable インタフェースは、コンパイルされると、イレイジャ処理の結果、実際には次のコードと同等になります。

```
public interface Comparable {
    public int compareTo(Object o);
}
```

したがって、ジェネリック化された Comparable インタフェースを実装するということは、実際には引数が Object 型の compareTo メソッドも生成されなければならないことになります。12 頁の EmployeeName クラスでは、そのような compareTo メソッドを明示的には記述していませんが、コンパイラによって、次のようなメソッドが自動的に挿入されます。

```
public int compareTo(Object that) {
    return compareTo((EmployeeName) that);
}
```

javap コマンドを用いて、EmployeeName.class ファイルを解析してみると次のようになります。

```
D:\Tiger\example\generics>javap EmployeeName
Compiled from "EmployeeName.java"
class EmployeeName extends java.lang.Object implements java.lang.Comparable{
    public final java.lang.String firstName;
    public final java.lang.String lastName;
    public EmployeeName(java.lang.String, java.lang.String);
    public int compareTo(EmployeeName);
    public java.lang.String toString();
    public static void main(java.lang.String[]);
    public int compareTo(java.lang.Object);
}
```

引数の型が、Object と EmployeeName である 2 つの compareTo メソッドが存在していることが分かります。つまり、従来の compareTo メソッドの契約を守ったメソッドが自動的に生成されていることになります。このようにコンパイラによって自動生成されるメソッドを、**ブリッジメソッド**（*bridge method*）と呼びます。コンパイラは、ブリッジメソッドを含むソースコードを生成してから、それをコンパイルしているのではありません。直接、ブリッジメソッドを含むバイトコードを生成しています。

このことから分かるように、ジェネリック化されたインタフェースに関しては、次の制約があります。

【制約 7】型変数が異なる同じインタフェースを複数実装することはできない。

つまり、次のような宣言はできないことになります。

```
class Foo implements Comparable<Integer>, Comparable<String>
```

この場合、コンパイラはブリッジメソッドを挿入しなければなりませんが、そのブリッジメソッドは、Integer 型を引数に取る compareTo メソッドを呼び出すのか、String 型を引数に取る compareTo メソッドを呼び出すのかを決定することはできません。したがって、型変数が異なる同じインタフェースを複数実装することはできないわけです。

ブリッジメソッドはコンパイラが自動的に生成するメソッドですので、ソースコードからドキュメントを生成する javadoc コマンドでは、ブリッジメソッドに関する説明は一切含まれないことにも注意してください。たとえば、リリース 1.4 までは、Short クラスには以下の 2 つの compareTo メソッドがソースコード上に定義され、オンラインドキュメントにどちらも表示されています。

```
public final class Short extends Number implements Comparable {
    // ...
    public int compareTo(Object o) {
        return compareTo((Short)o);
    }
    public int compareTo(Short anotherShort) {
        return this.value - anotherShort.value;
    }
    // ...
}
```

しかし、リリース 5.0 では、ソースコードが以下のように修正されたため、compareTo メソッドはソースコード上には 1 つしかありませんし、javadoc コマンドで生成されたオンラインドキュメントにも 1 つしか表示されません。

```
public final class Short extends Number implements Comparable<Short> {
    // ...
    public int compareTo(Short anotherShort) {
        return this.value - anotherShort.value;
    }
    // ...
}
```

1.3.1 Iterator インタフェースとブリッジメソッド

コレクションフレームワークの中でジェネリック化されたインタフェースの 1 つとして、Iterator インタフェースがあります。その定義は、次のようになっています。

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

次のプログラムは、Stack クラスに、スタックの先頭から値を取り出すイテレータを返す iterator メソッドを追加したものです。なお、java.lang.Iterable インタフェースを実装していますが、Iterable インタフェースについては、3.2 節で説明します。

```
import java.util.EmptyStackException;
import java.util.Iterator;
import java.util.NoSuchElementException;

class Stack<E> implements Iterable<E> {
    private E[] elements;
    private int size = 0;

    public Stack(int initialCapacity) {
        elements = (E[]) new Object[initialCapacity];
    }

    public void push(E e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public E pop() {
        if (size == 0)
            throw new EmptyStackException();
        E e = elements[--size];
        elements[size] = null;
        return e;
    }

    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private int pos = 0;

            public boolean hasNext() {
                return (pos < size);
            }

            public E next() {
                if (pos >= size)
                    throw new NoSuchElementException();
                return elements[pos++];
```

```
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }

    private void ensureCapacity() {
        if (elements.length == size) {
            E[] oldElements = elements;
            elements = (E[]) new Object[2 * elements.length + 1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}

class StackTest {

    public static void main(String[] args) {
        Stack<Integer> s = new Stack<Integer>(args.length);

        for (int i = 0; i < args.length; i++)
            s.push(new Integer(args[i]));

        int total = 0;
        for (Integer i: s)
            total += i;

        System.out.println("total = " + total);
    }
}
```

next メソッドが次のように型変数 E を用いて宣言されていることに注意してください。

```
public E next()
```

この場合、イテレータそのものが Stack クラス内の無名クラスとして生成されているので、実際に生成される無名クラスの next メソッドのシグニチャは、次のようになります。

```
public Object next()
```

この例からも分かるように、Stack クラスの型変数 E は、無名クラス内でも有効です。

では、次のプログラムを見てください。

```
import java.util.NoSuchElementException;
import java.util.Iterator;
```

```
class IteratorTest {

    public static Iterator<Integer> walkThrough(final Integer[] objs) {
        class Iter implements Iterator<Integer> {
            private int pos = 0;
            public boolean hasNext() {
                return (pos < objs.length);
            }
            public Integer next() {      // *
                if (pos >= objs.length)
                    throw new NoSuchElementException();
                return objs[pos++];
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        }

        return new Iter();
    }

    public static void main(String[] args) {
        Integer[] objs = new Integer[args.length];

        for (int i = 0; i < args.length; i++)
            objs[i] = new Integer(args[i]);

        int total = 0;
        for (Iterator<Integer> i = walkThrough(objs); i.hasNext(); )
            total += i.next();

        System.out.println("total = " + total);
    }
}
```

walkThrough メソッドでは、Integer の配列を受け取って、Iterator<Integer>を返します。walkThrough メソッド内では、イテレータを実装しているローカルクラス Iter が定義されており、そのインスタンスが返されています。

Iter クラスは、Iterator<Integer>を実装すると宣言されていますので、next メソッドが次のように宣言されていることに注目してください。

```
public Integer next()
```

ジェネリック化された Iterator インタフェースであっても、next メソッドとしては、実際には戻り値型が Object の next メソッドが必要となります。つまり、コンパイラは、戻り値型が Object であるブリッジメソッドを、Comparable インタフェースの例と同様に、自動的に挿入します。その結果、

Iter クラスのオブジェクトは、次の 2 つの next メソッドを持つことになります。

```
public Integer next()
public Object next()
```

Java 言語仕様では、戻り値型だけが異なるメソッドをオーバーロードすることはできません。ただし、それはソースコードレベルで書いた場合の話です。つまり、戻り値型だけが異なるメソッドをオーバーロードしたプログラムをコンパイルすると、コンパイルエラーになります。しかし、バイトコードレベルで戻り値型が異なるメソッドを、コンパイラが生成することは可能です。

ところで、「戻り値型が Object の next メソッドは、どうやって、もう 1 つの next メソッドを呼び出すのだろう？」と疑問に思うかもしれません。コンパイラが自動生成したブリッジメソッドは、プログラマが定義した next メソッドを呼び出して、その値を返さなければなりません。そのため、ブリッジメソッドの本体には、プログラマが定義した next メソッドを呼び出すコードが生成されます。

呼び出されるメソッドの形式（戻り値型を含む）が決定されるのは、コンパイル時であって、実行時ではありません。したがって、ジェネリックスに対応したコンパイラは、プログラマが定義した next メソッドを呼び出すブリッジメソッドのバイトコードを生成します。[*6]

1.4 共変戻り値型（*covariant return type*）

リリース 1.4 までの Java 言語では、スーパークラスのメソッドを、サブクラスがオーバーライドする場合には、メソッドのシグニチャだけでなく、戻り値型が同じでなければなりませんでした。次のプログラムでは、クラス C のサブクラス D で、メソッド dup を定義していますが、戻り値型がクラス C ではなく、クラス D と定義されていることに注意してください。

```
class C {
    C dup() {
        return new C();
    }
    public String toString() {
        return "instance of class C";
    }
}

class D extends C {
    D dup() { // 共変戻り値型
        return new D();
    }
    public String toString() {
        return "instance of class D";
    }
}
```

[*6] コンパイル時に決定することに関しては、『プログラミング言語 Java 第 3 版』の 6.9.1 節「正しいメソッドを見つける」を参照してください（『プログラミング言語 Java 第 4 版』の 9.6.1 節「正しいメソッドを見つける」）。

```
class CovariantReturnType {
    public static void main(String[] args) {
        C c = new D();
        C dup = c.dup();
        System.out.println(dup);
    }
}
```

このプログラムをリリース 1.4 でコンパイルすると、「D の dup は C の dup をオーバーライドできません」というエラーメッセージが表示されて、コンパイルエラーになります。リリース 5.0 でコンパイルするとコンパイルできて、CovariantReturnType クラスを指定して実行すると、クラス D の dup メソッドが呼び出されて、instance of class D と表示されます。

このように、ジェネリックスの仕様の一部である**共変戻り値型**（*covariant return type*）のサポートにより、メソッドのオーバーライドの定義が変更になります。**メソッドのシグニチャが一致していて、戻り値型が参照型で、かつ、スーパークラスで定義されたメソッドの戻り値型のサブタイプであれば、スーパークラスのメソッドをオーバーライドすることができるようになります。**

この場合でも、実際には次のようなブリッジメソッドがクラス D に、コンパイラにより挿入されます。

```
C dup()
```

このメソッドが実際にスーパークラス C の dup メソッドを、オーバーライドしており、その中で、戻り値型が D と定義された dup メソッドを呼び出して、その結果を返しているのです。

クラス D のコンパイル結果を javap コマンドで調べると次の結果になります。

```
D:\Tiger\example\generics>javap D
Compiled from "D.java"
class D extends C{
    D();
    D dup();
    public java.lang.String toString();
    C dup();
}
```

クラス D に、戻り値型が C である dup メソッドがブリッジメソッドとして定義されていることが分かります。つまり、クラス C の dup メソッドを実際にオーバーライドしているのは、このブリッジメソッドです。ここで、次のようにクラス E を定義したとします。

```
class E extends D {
    E dup() {
        return new E();
    }
    public String toString() {
        return "instance of class E";
    }
}
```

クラス E のコンパイル結果を javap コマンドで調べると次の結果になります。

```
D:\Tiger\example\generics>javap E
Compiled from "E.java"
class E extends D{
    E();
    E dup();
    public java.lang.String toString();
    D dup();
    C dup();
}
```

3 個の dup メソッドが定義されていることが分かります。最初の dup メソッドは、ソースコードに書かれたメソッドです。2 番目の dup メソッドは、クラス D のソースコードで書かれた dup メソッドをオーバーライドするためのブリッジメソッドです。3 番目の dup メソッドは、クラス C で定義された dup メソッドをオーバーライドするためのブリッジメソッドです。

共変戻り値型は、このような方法で実現されていますのでリリース 1.4 を使用して作成された既存のクラスを、共変戻り値型を使用して書き直す場合には注意が必要です。たとえば、クラス D と E で、dup メソッドの戻り値型が C となっていた既存のクラスのソースコードとコンパイル結果のクラスファイルが存在したとします。そして、クラス D の dup メソッドの戻り値型を D に変更して、クラス D だけを再コンパイルしたとします。そうすると、クラス E にはブリッジメソッドは存在しませんので、戻り値型が D であるクラス D の dup メソッドはオーバーライドされていません。さらに、クラス E を何も修正することなく再コンパイルすると、コンパイルエラーになってしまいます。なぜならば、ソースコード上では、戻り値型が C であるクラス E の dup メソッドが、戻り値型が D であるスーパークラスの dup メソッドをオーバーライドしようとしているからです。

1.4.1 インタフェースの実装

複数のインタフェースを実装した場合に、シグニチャが同じで、戻り値型が異なっているメソッドがインタフェースで定義されている場合には注意が必要です。たとえば、次のようなインタフェース X、Y、Z があったとします。

```
interface X {
    Object foo();
}

interface Y {
    Number foo();
}

interface Z {
    Integer foo();
}
```

すべて同じシグニチャの foo メソッドを定義していますが、戻り値型が異なります。このような場合、あるインタフェースで定義されたメソッドの戻り値型が、他のすべてのインタフェースで定義されているメソッドの戻り値型のサブタイプであれば、すべてのインタフェースを継承することができます。Z インタフェースの foo メソッドの戻り値型である Integer 型は、X インタフェースおよび Y インタフェースの foo メソッドの戻り値型のサブタイプとなっていますので、この 3 つのインタフェースを実装することが可能です。

このような複数のインタフェースを実装する場合には、次のように、戻り値型がもっともサブタイプになっている（継承関係の末端になっている）型のメソッドだけを実装します。つまり、Z インタフェースの foo メソッドだけを実装します。

```
class XYZ implements X, Y, Z {

    public Integer foo() {
        return new Integer(0);
    }
}
```

コンパイラは、自動的に X インタフェースと Y インタフェースの foo メソッドを生成します。

1.4.2 clone メソッド

共変戻り値型を使用した例としては、clone メソッドの実装があります。Object クラスの clone メソッドはジェネリックスを使用するようには変更されていませんが、共変戻り値型の導入により、clone メソッドの戻り値型を Object 型ではなく、そのクラス自身の型に定義できます。

```
public class MyClass implements Cloneable {
    public MyClass clone() throws CloneNotSupportedException {
        return (MyClass) super.clone();
    }
    // ...
}
```

さらに、配列に関する言語仕様が変更になっています。配列は、Cloneable インタフェースと Serializable インタフェースを実装しており、リリース 1.4 までは、配列に対する clone メソッドの呼び出しでは、その戻り値型は Object 型でした。リリース 5.0 からは、T 型の配列である T[] の配列の clone メソッドの戻り値型は、T[] となります。その結果、配列に対して clone メソッドを呼び出して、同じ型の配列参照へ代入する場合には、キャストは必要ありません。

```
int[] intArray = new int[10];
int[] x = intArray.clone(); // キャストは不要
```

1.4.3 注意事項

共変戻り値型が許されるのは、参照型だけです。基本データ型には適用されません。short 型の値はキャストなしで、int 型に代入可能ですが、次のようなオーバーライドはできません。

```
class Super {
    int foo() { /* ... */ }
}

class Sub extends Super {
    short foo() { /* ... */ } // NG! コンパイルエラー
}
```

1.5 型の継承関係とワイルドカード

ArrayList<Integer>は、次の通り、List<Integer>に代入可能です。

```
List<Integer> list = new ArrayList<Integer>();
```

Integer を要素として持つ ArrayList は、Integer を要素に持つ List であることは明らかです。List<Integer>として取り扱って、Integer クラスのインスタンスを追加しても、Integer インスタンスの配列リストであるという不変式を破ることになりません。つまり、List<Integer>は、ArrayList<Integer>のスーパータイプとなります。

では、List<Integer>と List<Object>ではどうなるでしょうか。List<Integer>は、Integer クラスのインスタンスを持つリストであり、List<Object>は、Object を要素に持つリストです。その意味で、List<Object>は、List<Integer>のスーパータイプのように見えます。しかし、実際には、その 2 つには、スーパータイプ/サブタイプの関係は存在しません。次のコードを見てください。

```
List<Integer> iList = new ArrayList<Integer>();
List<Object> oList = iList; // NG! コンパイルエラー
oList.add(new Double(0.0));
```

もし、この例で、oList = iList の代入が許されるとしたら、次の行の oList に対する add メソッドの呼び出しにより、Double クラスのインスタンスが追加されてしまいます。そうすると、実際に生成されているのは、Integer クラスのインスタンスを要素に持つ配列リストであるはずにもかかわらず、不正な Double クラスのインスタンスが挿入されてしまうことになってしまいます。そのため、List<Object>と List<Integer>間には、スーパタイプ/サブタイプの関係は存在しません。

1.5.1 ワイルドカード

List<Integer>と List<String>は、異なる型ですが、そのどちらも取り扱いたいメソッドを作ることを考えてみます。たとえば、リスト中の要素をすべて出力するようなメソッドを考えてみてくださ

い。ジェネリックスを使用しない従来の書き方ですと、次のように書けます。

```
static void printList(List list) {
    for (Iterator i = list.iterator(); i.hasNext(); )
        System.out.println(i.next());
}
```

このメソッドをジェネリックメソッドとして書き直すと次のようになります。

```
static <E> void printList(List<E> list) {
    for (E e: list)
        System.out.println(e);
}
```

この場合、型変数 E に対して、型変数固有のメソッドを呼び出していません。したがって、特に型変数を指定するのではなく、次のようにさらに書き直すことができます。

```
static void printList(List<?> list) {
    for (Object e: list)
        System.out.println(e);
}
```

List<?>はすべての List のスーパータイプを表し、?は**ワイルドカード**（*wildcard*）[*7]と呼ばれます。List<?>は**不明な型のリスト**であり、List<Integer>や List<String>は、そのサブタイプとして扱われます。

List<?>はそのリストが取り扱っている型が不明ですので、そのリストに対して要素の追加はできません。たとえば、List インタフェースの add メソッドは、次のように定義されています。

```
public interface List<E> extends Collection<E> {
    // ...
    boolean add(E o);
    // ...
}
```

型変数 E を用いて、add メソッドは定義されていますので、次のようなコードは書けません。

```
List<?> list = new ArrayList<Integer>(); // OK
list.add(new Integer(10)); // NG! コンパイルエラー
list.add(new Object()); // NG! コンパイルエラー
list.add(null); // OK
```

これは、List<?>は、不明な型のリストなので、不明な型のサブタイプしか渡せませんが、Object クラスであっても、不明な型のサブタイプとしては取り扱われません。そのため、add メソッドの引数として渡すことはできません。唯一の例外は、null です。

[*7] ワイルドカードは、初期のジェネリックスの仕様には入っていませんでした。ワイルドカードの最終的な仕様策定は、サンマイクロシステムズ社とデンマークの Aarhus 大学との共同作業として行われたようです。

不明な型のリストであっても、その中に含まれている要素は、オブジェクトであることは明白ですので、その要素を printList メソッドの例のように Object クラスのインスタンスとして取り扱うことは可能です。

配列の要素として不明な型のリストを取るように宣言することもできます。次のコードは、そのような配列の宣言と配列の生成を行います。

```
List<?>[] arrayOfLists = new List<?>[64];
```

ジェネリッククラスやジェネリックインタフェースを設計する際に注意しなければならないのは、型変数が指定されて生成されたインスタンスが持つ型変数に関連した不変式を破壊しないように、メソッドを設計する必要があることです。次のコードは、データを保持するだけの簡単なクラスです。

```
class DataHolder<E> {
    private E e;

    public void set(E e) { this.e = e; }
    public E get()       { return e; }
    public void loophole(Object o) {
        e = (E) o;
    }
}
```

この DataHolder クラスの問題点は、loophole メソッドです。これは、E 型の要素を持つという不変式を破壊しており、ワイルドカードを用いても次のようなコードが書けてしまいます。

```
DataHolder<?> dh = new DataHolder<Integer>();
dh.loophole(new Double(0.0));
```

Integer クラスを保持するインスタンスであるはずなのに、loophole メソッドにより、他の型のインスタンスを設定できてしまいます。ジェネリッククラスやジェネリックインタフェースは、このような不整合が起きないように設計する必要があります。

1.5.2 有界ワイルドカード

任意のリストを扱うために、List<?>の使用が可能であることを説明しました。Number クラスを継承した、Integer クラスや Float クラスなどのリストの要素を出力するメソッドを、次のように書いたとします。

```
static void printNumberList(List<Number> list) {
    for (Number n: list)
        System.out.println(n);
}
```

残念ながら、この printNumberList メソッドには、List<Integer>型や List<Float>型のインスタンスを渡すことはできません。なぜなら、28 頁で説明したように、List<Integer>型と List<Number>型

にはスーパータイプ/サブタイプの関係は存在しないからです。そこで、次のように書き直す必要があります。

```
static <T extends Number> void printNumberList(List<T> list) {
    for (Number n: list)
        System.out.println(n);
}
```

この場合、**型変数 T は、メソッド内では一切使用されていません**ので、次のようにワイルドカードを使用してさらに書き直すことができます。

```
static void printNumberList(List<? extends Number> list) {
    for (Number n: list)
        System.out.println(n);
}
```

List<? extends Number>は、不明な型のリストであることに変わりはないのですが、その不明な型は、Number のサブタイプであることを示しています。しかし、実際の型は不明ですので、要素の追加などはできません。つまり、次のようになります。

```
Integer[] ia = new Integer[] {1, 2, 3, 4};
List<? extends Number> list = new ArrayList<Integer>(Arrays.asList(ia));

Number num = list.get(0); // OK
list.add(new Double(0.0)); // NG! コンパイルエラー
```

extends を用いた境界[*8] は、ワイルドカードに対する**上限境界**（*upper bounds*）を表します。一方、ワイルドカードに対しては、super を用いて**下限境界**（*lower bounds*）を表すことができます。たとえば、String クラスのスーパータイプを要素として持つリストは、List<? super String>と表します。この場合、List<? super String>は、List<String>と List<Object>のスーパタイプとなります。

```
List<? super String> list = new ArrayList<String>(); // OK
list = new ArrayList<Object>(); // OK
list = new ArrayList<Integer>(); // NG! コンパイルエラー
```

super を用いた下限境界指定の場合には、上限境界と異なり、次のようなコードを書くことができます。

```
List<? super String> list = new ArrayList<String>();
list.add("Hello"); // OK
```

ワイルドカードが使用されていますが、list は、String および Object のリストですから、add メソッドで String クラスのインスタンスを追加することができます。

[*8] ワイルドカードでは、複数の境界を指定することはできません。たとえば List<? extends Number & Serializable>と書くことはできません。

ワイルドカードを用いていない List<Integer>型と List<Number>型には、スーパータイプ/サブタイプの関係は存在しませんが、ワイルドカードを用いた型同士には、スーパータイプ/サブタイプの関係は存在します。Number クラスは、Integer クラスのスーパタイプですので、extends を用いた上限境界では、List<? extends Number>は List<? extends Integer>のスーパタイプです。一方、super を用いた下限境界では、List<? super Number>は、List<? super Integer>のサブタイプです。

下限境界を用いて定義された例として、TreeSet クラスを用いて説明します。TreeSet クラスは、Comparator インタフェースを実装したオブジェクトを引数に取るコンストラクタを持っています。

```
public class TreeSet<E>
    extends AbstractSet<E>
    implements SortedSet<E>, Cloneable, java.io.Serializable {
    // ...

    public TreeSet(Comparator<? super E> c) { /* ... */ }
    // ...
}
```

仮に、このコンストラクタが次のように定義されていたとします。

```
public TreeSet(Comparator<E> c)
```

そうすると、TreeSet<String>を作成しようとすると、必ず、Comparator<String>を実装したコンパレータを作成して渡さなければなりません。しかし、必ずしも、そこまで制約を課す必要もなく、Comparator<Object>であっても、TreeSet の実装としては構わないことを示すために、Comparator<? super E>と宣言されています。

もう少し複雑な例を考えてみます。1.2.7 節で説明した、BoundTest クラスの max メソッドでは、次のように定義しました。

```
public static <T extends Object & Comparable<T>> T
       max(Collection<T> coll)
```

ここで、次のようなクラスを定義したとします。

```
class Foo implements Comparable<Object> { /* ... */ }
```

この Foo クラスでは、Comparable インタフェースを実装しているのですが、型パラメータとして Object 型を指定していますので、Foo クラスのインスタンスを max メソッドに渡すことができません。

```
List<Foo> list = new ArrayList<Foo>();
// ...
Foo maxFoo = BoundTest.max(list); // NG! コンパイルエラー
```

この問題を解決するためには、次のように max メソッドの定義を変更する必要があります。

```
public static <T extends Object & Comparable<? super T>> T
       max(Collection<T> coll)
```

さらに、この max メソッドは、渡されたコレクションに対して修正操作をすることはなく、読み取り操作しかしないため、次のように書き直すことで、T 型だけでなく、T 型のサブタイプのコレクションも取り扱うことができるようになります。

```
public static <T extends Object & Comparable<? super T>> T
                max(Collection<? extends T> coll)
```

なお、super を用いた下限境界の指定では、ワイルドカードしかスーパタイプ部分に指定できないことに注意してください。つまり、次のようなジェネリックメソッドを書くことはできません。

```
public <T> void bar(List<T super Integer> list) { // NG! コンパイルエラー
    // ...
}
```

1.5.3 ワイルドカードキャプチャー

リストに要素を追加する次のジェネリックメソッドがあったとします。

```
public static <E> void addToList(List<E> list, E e) {
    list.add(e);
}
```

このジェネリックメソッドに対して次のようなコードは、コンパイルエラーとなります。

```
List<?> list = new ArrayList<Integer>();
addToList(list, new Integer(0)); // コンパイルエラー
```

実際には、Integer のリストですが、list は不明な型のリストと宣言されています。したがって、addToList メソッドの呼び出しは、第 2 引数が正しい型のオブジェクトであってもコンパイルエラーとなります。

たとえば、リスト内の要素を逆順にして返すジェネリックメソッドを定義したいとします。リスト内の型は何であるかは特に関係ありませんので、次のようにワイルドカードを使用して定義したとします。

```
public static List<?> reverse(List<?> list) { /* ... */ }
```

しかし、この宣言では、次のようなコードはコンパイルエラーになります。

```
List<Integer> list = new ArrayList<Integer>();
// ...
list = reverse(list); // コンパイルエラー
```

この場合、reverse メソッドの戻り値型が不明な型のリストなので、戻り値を Integer のリストへ代入することはできません。そこで、再度、reverse を次のように定義しなおしたとします。

```
public static <T> List<T> reverse(List<T> list) { /* ... */ }
```

これで先ほどのコンパイルエラーはなくなります。さらに、次のようなコードもコンパイルできます。

```
List<?> list = new ArrayList<Integer>();
// ...
list = reverse(list); // OK
```

reverse メソッドには、不明な型のリストを渡していますが、型変数がメソッドのパラメータに 1 度しか使用されていなければ、コンパイルできます。これは、不明な型の実行時の本当の型をメソッドの呼び出しの際に得る（キャプチャー）ということで、**ワイルドカードキャプチャー**（*wildcard capture*）と呼ばれます。最初の addToList メソッドの例では、型変数がメソッドパラメータの中で 2 度使用されており、ワイルドカードキャプチャーは行われませんので、コンパイルエラーとなります。また、メソッドのパラメータで 1 度しか使用されていなくても、ジェネリッククラスの型変数として指定されている場合（たとえば、List<T>）はよいのですが、そうでない場合（たとえば、List<List<T>>）には、ワイルドカードキャプチャーは行われません。つまり、次のようなジェネリックメソッドに対するメソッド呼び出しでは、ワイルドカードキャプチャーは行われません。

```
public static <T> List<T> getFirst(List<List<T>> list) {
    return list.get(0);
}
```

現実的な例として、Collections クラスの static メソッドである synchronizedList メソッドを見ると、次のように定義されています。

```
public static <T> List<T> synchronizedList(List<T> list)
```

ワイルドカードキャプチャーにより、次のように不明な型のリストを受け取って、同期されたリストへ変換することが可能です。

```
List<?> list = new ArrayList<Integer>();
// ...
list = Collections.synchronizedList(list);
```

1.5.4 instanceof 演算子とジェネリック化された型

instanceof 演算子の後には参照型を指定しなければなりませんが、ジェネリックスの導入により仕様が変更になり、次の参照型の指定が可能となります。

- ジェネリック化されていない型
- ジェネリック化されている型の原型（1.6 節参照）
- すべての型引数に、境界なしのワイルドカードが指定されてジェネリック化されている型
- 上記の型の配列

最初の項目は、従来通りですが、ジェネリック化されている List インタフェースなどでは、次のように書くことができます。

```
if (o instanceof List) // 原型
    // ...

if (o instanceof List<?>) // 型引数が境界なしのワイルドカード
    // ...

if (o instanceof List<?>[]) // 配列
    // ...
```

しかし、次のようなコードは、すべてコンパイルエラーです。

```
if (o instanceof List<Integer>) // NG! コンパイルエラー
   // ...

if (o instanceof List<? extends Number>) // NG! コンパイルエラー
   // ...
```

さらに、型変数に対しても、次のように適用できません。

```
class Foo<T> {
    boolean instanceOf(Object o) {
        return o instanceof T; // NG! コンパイルエラー
    }
}
```

1.5.5 パラメータ化された型の配列

パラメータ化された型の配列の生成に関して制約があります。次のようなコードは、コンパイルエラーとなります。

```
List<Integer>[] list = new List<Integer>[10];  // NG! コンパイルエラー
```

しかし、次のコードのように、ワイルドカードを使用した配列の生成はできます。

```
List<?>[] list = new List<?>[10];
```

1.6 原型（*raw type*）

ジェネリックスはリリース 5.0 で導入された言語仕様ですから、当然、リリース 1.4 までのジェネリックスを全く使用していない既存のプログラムが膨大に存在します。ジェネリッククラスやジェネリックインタフェースに対して、型変数を指定しないで従来通りの宣言をしたものは、**原型**（*raw type*）と呼ばれます。たとえば、次のコードを見てください。

```
List list = new ArrayList<Integer>(); // OK
List<Integer> iList = list; // OK ただし、無検査警告（unchecked warning）
```

このように list は、List<Integer>ではなく、単に List と宣言されていますので、原型となります。パラメータ化されて型変数が指定されたインスタンスを原型に代入することは、特に問題ありません。一方で、原型を型変数が指定された参照に代入することは、一見すると安全ではないので、コンパイルエラーにすべきのように見えます。

しかし、現実問題として、ジェネリックスと既存のプログラムが今後混在していくことを考慮すると、このような代入でコンパイルエラーにすると、多くのプログラムがコンパイルできなくなってしまいます。したがって、コンパイルエラーになるのではなく、**無検査警告**（*unchecked warning*）メッセージがコンパイラにより表示されます。つまり、ジェネリックスにより、型の安全性をコンパイラがさらにチェックするようになったのですが、この場合には、そのような代入が型の安全性の観点から保証できないと警告しています。

実際にプログラムのどの部分に対して、そのメッセージが表示されているかを知るには、コンパイルオプションとして、-Xlint:unchecked（9.15 節参照）を指定しなければなりません。たとえば、1 頁で示した LinkedListTest クラスを、リリース 1.4 のコンパイラでコンパイルすると、何も警告メッセージが表示されることなくコンパイルできます。しかし、リリース 5.0 のコンパイラでコンパイルすると、次のようにメッセージが表示されます。

```
D:\Tiger\example\generics>javac LinkedListTest.java
注: LinkedListTest.java の操作は、未チェックまたは安全ではありません。
注: 詳細については、-Xlint:unchecked オプションを指定して再コンパイルしてください。
```

そこで、-Xlint:unchecked を付けてコンパイルしてみると、次のように詳細が表示されます。

```
D:\Tiger\example\generics>javac -Xlint:unchecked LinkedListTest.java
LinkedListTest.java:9: 警告: [unchecked] raw 型 java.util.List のメンバとしての
add(E) への無検査呼び出しです。
        list.add(new Integer(10));
                ^
LinkedListTest.java:10: 警告: [unchecked] raw 型 java.util.List のメンバとしての
 add(E) への無検査呼び出しです。
        list.add(new Integer(20));
                ^
警告 2 個
```

このような警告の表示を抑制するために、@SuppressWarnings アノテーション（7.1.3 節参照）が、リリース 5.0 からは導入されています。[*9]

1.7 オーバーロードとオーバーライド

ジェネリックスの導入により、メソッドの**シグニチャ**（*signature*）、および、メソッドのオーバーロードとオーバーライドには、注意が必要です。

[*9] @SuppressWarnings("unchecked") とアノテーションを指定します。リリース 5.0 が最初にリリースされてから一年程度は、このアノテーションは意図的にサポートされていませんでした。

型変数がパラメータとして使用されているメソッドやコンストラクタは、以下の条件が成り立つ場合に、**同じシグニチャ**（*same signature*）と見なされます。

- 同じ個数の型変数を持ち、各型変数の境界が同じである。
- 2 番目のメソッドの型変数名を、1 番目のメソッドの型変数名で置き換えた場合に、**仮パラメータ**（*formal parameter*）の型が同じである。

2 つのメソッドのシグニチャに関しては、**サブシグニチャ**（*subsignature*）という関係が存在する場合があります。m1 メソッドと m2 メソッドで、以下の条件が成り立てば、m1 メソッドのシグニチャは、m2 メソッドのサブシグニチャとなります。

- m1 メソッドが m2 と同じシグニチャを持っている。あるいは、
- m2 メソッドに対してイレイジャ処理を行った結果のシグニチャが、m1 メソッドと同じである。

そして、2 つの m1 メソッドおよび m2 メソッドは、m1 メソッドのシグニチャが m2 メソッドのサブシグニチャか、あるいは逆に、m2 メソッドのシグニチャが m1 メソッドのサブシグニチャであれば、**オーバーライド等価**（*override-equivalent*）なシグニチャを持つメソッドだと見なされます。

メソッドがオーバーロードされるのは、名前が同じで、オーバーライド等価では ない シグニチャを持つ場合だけです。言い換えると、イレイジャ処理を行った結果のシグニチャが同じであれば、オーバーロードすることはできません。

「オーバーライド等価」という言葉は、紛らわしい言葉です。これは、メソッドをオーバーライドできるということを意味しているのではありません。**スーパークラスのメソッドをサブクラスでオーバーライドする場合は、サブクラスのメソッドのシグニチャは、スーパークラスのメソッドのサブシグニチャでなければなりません。**たとえば、次のようなクラス A と B がある場合を考えてみます。

```
class A {
    <T> void foo(T t) { }
    void bar(Object o) { }
}

class B extends A {
    void foo(Object o) { }  // OK!
    <T> void bar(T t) { }   // NG! コンパイルエラー
}
```

B クラスの foo メソッドのシグニチャは、A クラスの foo メソッドに対して、イレイジャ処理を行った結果と同じなので、A クラスの foo メソッドのサブシグニチャと見なされます。しかし、B クラスの bar メソッドのシグニチャは、A クラスの bar メソッドのサブシグニチャではありません。その結果、コンパイルエラーとなります。

本来ならば、B クラスの foo メソッドもコンパイルエラーとすべきかもしれませんが、コンパイルエラーにしてしまうと、現実の世界では不都合が発生してしまいます。つまり、既存のベースクラスに対して型変数を導入してジェネリッククラスにしてしまうと、すべてのサブクラスを修正しなければなら

なくなってしまいます。そうなると、既存のライブラリーをジェネリックスに対応させるための障害となり、ジェネリックスを普及させる足かせになってしまうため、このような仕様となっています。

1.8 Class クラス

java.lang.Class クラスは、リリース 5.0 ではジェネリック化されており、以下のようにクラス宣言されています。

```
public final class Class<T> implements java.io.Serializable,
                  java.lang.reflect.GenericDeclaration,
                  java.lang.reflect.Type,
                  java.lang.reflect.AnnotatedElement
```

そして、この型変数 T を使用して newInstance メソッドも次のように宣言されています。

```
public T newInstance() throws InstantiationException, IllegalAccessException
```

したがって、次のようにコードを書くことができるようになります。

```
Class<Date> c = Date.class;
Date d = c.newInstance();
```

ここで、Date.class は、Class<Date>型であるとして扱われていることに注意してください。その結果、newInstance メソッドの戻り値をキャストすることなく、Date 型の参照である d に代入できています。

このように参照型 W のクラスリテラルである W.class は、Class<W>型と見なされます。一方、基本データ型の int のクラスリテラル int.class の型は、Class<Integer>型となります。つまり、基本データ型のクラスリテラルは、対応するラッパークラスを X とすると Class<X>型になり、int.class と Integer.class の型は、どちらも Class<Integer>型となります。ただし、int.class と Integer.class は、全く別のオブジェクトです。また、void.class の型は、Class<Void>型です。

なお、ジェネリック化された型に、型パラメータを指定してクラスリテラルを得ることはできません。たとえば、ArrayList<Integer>.class とは書けません。

1.8.1 cast メソッド

オブジェクトを受け取り、指定された型へ変換して返すような次のメソッドを作成したとします。

```
public <T> T narrow(Object o) {
    return (T) o;
}
```

このメソッドの実装では、T 型へキャストすることに対して、無検査警告メッセージがコンパイラから出されます。これを回避するには、Class クラスの cast メソッドを使用して、次のように変更します。

```
public static <T> T narrow(Object o, Class<T> cl) {
    return cl.cast(o);
}
```

このように、オブジェクトをキャストしたい型を表している Class オブジェクトを使用することで、明示的にキャストすることなく変換することができます。ただし、渡されたオブジェクトが指定された T 型への代入ができない場合には、実行時に ClassCastException がスローされます。

1.8.2 asSubclass メソッド

Class クラスには、Class クラスのインスタンスを、特定の型のサブクラスの型へ変換するための asSubclass メソッドが用意されています。asSubclass メソッドの定義は、次のようになっています。

```
public <U> Class<? extends U> asSubclass(Class<U> clazz)
```

たとえば、Runnable インタフェースを実装しているはずであるクラスを、Class クラスの forName メソッドを用いてロードする場合を考えてみます。forName メソッドの戻り値型は、Class<?>と宣言されています。したがって、クラスをロードして、Class オブジェクトを取得するためのコードは、次のようになります。

```
Class<?> cl = Class.forName( /* ... */ ); // クラス名を指定してロードする
```

Runnable インタフェースを実装したクラスの Class オブジェクトは、本来なら、Class<? extends Runnable>ですので、次のようにキャストしたとします。

```
Class<? extends Runnable> runnableCl = (Class<? extends Runnable>) cl;
```

このキャストに対しては、コンパイラは、無検査警告メッセージを表示します。その警告メッセージを出さないようにするには、次のように asSubclass メソッドを使用します。

```
Class<? extends Runnable> runnableCl = cl.asSubclass(Runnable.class);
```

もし、cl が表している本当の型が、asSubclass メソッドに渡された型のサブタイプでなかった場合には、実行時に ClassCastException がスローされます。このようにして取得した runnableCl を使用して、警告メッセージなしに、次のようにインスタンスを生成することができます。

```
Runnable r = runnableCl.newInstance();
```

1.8.3 Object クラスの getClass メソッド

Object クラスの getClass メソッドは、そのメソッドが呼び出されたオブジェクトのクラスを表す Class オブジェクトを返します。getClass メソッドが呼び出された参照の型が T だとすると、getClass メソッドが返す値の型は、Class<? extends T>と見なされます。

```
String s = "Hello";
Class<? extends String> sClass = s.getClass();
```

s は String 型なので、getClass メソッドの戻り値型は、Class<? extends String>型となります。なお、この例で、sClass の型を Class<String>と宣言するとコンパイルエラーになります。

ジェネリッククラスで型パラメータを指定したクラスリテラルを、ArrayList<Integer>.class などと記述して取得することはできません。ジェネリッククラスでも、そのクラスの Class オブジェクトは 1 つしかなく、ArrayList.class とクラスリテラルは記述します。そのため、型パラメータとして異なった型が指定されたインスタンスであっても、getClass() メソッドで得られる Class オブジェクトは、同一のオブジェクトになります。次のような 2 つの ArrayList クラスのインスタンスがあったとします。

```
ArrayList<Integer> iList = new ArrayList<Integer>();
ArrayList<String>  sList = new ArrayList<String>();
```

この場合には、iList.getClass() == sList.getClass() が成り立ちます。

1.9 Collections クラス

ジェネリックスの導入により、java.util.Collections クラスには、新たな static メソッドが追加されています。

1.9.1 チェックされるコレクション

ジェネリックスが導入される前の既存のコードに対して、コレクションを渡す場合には、型の安全性が失われます。たとえば、Integer のリストである ArrayList<Integer>のインスタンスを、既存のコードに渡した場合には、Integer クラス以外のインスタンスを不正に挿入される可能性があります。そのような不正な挿入を防ぐために、java.util.Collections クラスに新たな static ファクトリーメソッドが追加されています。それらの static ファクトリーメソッドでは、不正な型のインスタンスの挿入を検出するためのチェック機能を持つ新たなコレクションのインスタンスを生成して返します。

たとえば、Integer クラスのインスタンスを持つチェックされるリストは、次のように生成します。

```
List list = Collections.checkedList(new ArrayList<Integer>, Integer.class);
```

こうして作成した list を既存のコードに渡すことで、既存のコード内で Integer クラス以外のクラスのインスタンスを挿入しようとすると、ClassCastException がスローされます。

各種コレクションに対応したチェックされるコレクションを生成するために、checkedCollection メソッド、checkedSet メソッド、checkedSortedSet メソッド、checkedMap メソッド、checkedSortedMap メソッドが用意されています。

1.9.2 空コレクション

リリース 1.4 までは、空のコレクションとして、Collections クラスに、EMPTY_SET、EMPTY_LIST、EMPTY_MAP が定義されていました。しかし、ジェネリックスの導入により、これらは原型として宣言さ

れていると見なされます。そのため、次のような代入を行うと、無検査警告メッセージが出ます。

```
List<Integer> list = Collections.EMPTY_LIST; // 無検査警告
```

この問題を解決するために、Collections クラスには、ジェネリックメソッドとして emptySet メソッド、emptyList メソッド、emptyMap メソッドが追加されており、次のように書き直すことができます。

```
List<Integer> list = Collections.emptyList();
```

1.10 リフレクション API

リリース 5.0 では、ジェネリックスに対応して、java.lang.reflect パッケージに新たなインタフェースが追加されています。

Type インタフェース
: Java 言語におけるすべての型を表すためのマーカーインタフェースです。このマーカーインタフェースを直接継承しているインタフェースは、GenericArrayType インタフェース、ParameterizedType インタフェース、TypeVariable インタフェース、WildcardType インタフェースです。直接実装しているクラスは、Class クラスです。

TypeVariable インタフェース
: すべての型変数を表すためのインタフェースです。型変数の名前を取得するためのメソッドやその境界（*bounds*）を取得するためのメソッドが用意されています。

WildcardType インタフェース
: ワイルドカードによる型を表すためのインタフェースです。ワイルドカードの上限境界と下限境界を取得するためのメソッドが用意されています。

GenericArrayType インタフェース
: T[] などのジェネリックの型の配列を表すインタフェースです。配列の要素の型を取得するためのメソッドが用意されています。

ParameterizedType インタフェース
: Collection<String>などのパラメータ化された型を表すインタフェースです。ジェネリック化された型での実際の型引数（Collection<String>であれば、String）を取得するためのメソッド、型引数に対してイレイジャ処理を行った場合の原型（Collection<T>であれば、Object）を取得するためのメソッドが用意されています。また、O<T>.I<S>として参照される I<S>にはエンクロージングクラスである O<T>がありますので、エンクロージングクラスの型を取得するためのメソッドが用意されています。

GenericDeclaration インタフェース
: 型変数を宣言できるものを表すインタフェースであり、Class クラス、Constructor クラス、Method クラスが実装しています。型変数を取得するためのメソッドだけが用意されています。

1.11 まとめ

ジェネリックスにより、コレクションを取り扱う際には、キャストすることなく使用できるようになります。一方、ジェネリックスを用いた API を設計する場合には、その API がどのように使用されるか、どの程度の柔軟性を与えるべきか、どのような制約を与えるべきかを注意深く検討しなければなりません。**ジェネリックスは型の安全性を格段に向上させたと言えますが、一方で、API 設計者には、深い洞察を必要とする高い能力を要求することになります。**

一番の嫌われ者のコレクションのキャストにも
最後にはやさしくさよならを言おう
ジェネリックスの火のついた槍をもってすれば
キャストは要らなくなってしまうだろう
– *Joshua Bloch*

第 2 章

ボクシング / アンボクシング

When from the collections ints are drawn
Wrapper classes make us mourn
When Tiger comes, we'll shed no tears
We'll autobox them in the ears
– Joshua Bloch

リリース 1.4 までは、int や long などの**基本データ型**（*primitive type*）と Integer クラスや Long クラスなどの**参照型**（*reference type*）間で、明示的にデータ変換を行う必要がありました。リリース 5.0 からは、基本データ型から参照型への自動変換である**ボクシング**（*boxing*）と、参照型から基本データ型への自動変換である**アンボクシング**（*unboxing*）がサポートされます。

2.1 簡単な例

ボクシング/アンボクシングの機能を説明するために、int 型のデータを扱うスタックを ArrayList クラスを用いて実装した次のプログラムを見てください。

```
import java.util.*;

class IntStack { // 1.4 version
    private List elements;

    public IntStack(int initialCapacity) {
        elements = new ArrayList(initialCapacity);
    }

    public void push(int e) {
        elements.add(new Integer(e));
    }

    public int pop() {
        if (elements.size() == 0)
            throw new EmptyStackException();
        Integer e = (Integer) elements.remove(elements.size() - 1);
        return e.intValue();
    }
}
```

push メソッドでは、リストに挿入するために Integer クラスのインスタンスを生成して追加しています。pop メソッドでは、リストから取り除いたオブジェクトを Integer へキャストした後、intValue メソッドを使用して値を取り出して返しています。このようにリリース 1.4 までは、基本データ型をコレクションで取り扱うには、ラッパークラスを明示的に使用したコードを書く必要がありました。

リリース 5.0 では、ジェネリックスと組み合わせることで、次のようにすっきりと書き直すことができます。

```
import java.util.*;

class IntStack { // 5.0 version
    private List<Integer> elements;

    public IntStack(int initialCapacity) {
        elements = new ArrayList<Integer>(initialCapacity);
    }

    public void push(int e) {
        elements.add(e);
    }

    public int pop() {
        if (elements.size() == 0)
            throw new EmptyStackException();
        return elements.remove(elements.size() - 1);
    }
}
```

elements フィールドが、List<Integer>と宣言されていることに注意してください。つまり、要素として、Integer クラスのインスタンスを持つリストであると宣言されています。push メソッドでは、リストに対する add メソッドにより、int 型のパラメータである e を elements に追加しようとします。そのために、int 型から Integer 型への変換を行う Integer.valueOf(e) が自動的に行われて(ボクシング)、その結果のインスタンスが追加されます。pop メソッドでは、elements から要素を取り除きますが、その要素は Integer クラスのインスタンスであることは elements フィールドの宣言から明らかですから、そのインスタンスに対して、Integer クラスの intValue メソッドが自動的に呼び出されて、その結果が返されます。

ジェネリックスでは、ジェネリッククラスやジェネリックインタフェースの型変数に対して、基本データ型を指定できません。しかし、ボクシング/アンボクシングにより、Integer などのラッパークラスを型変数として指定して生成したコレクションクラスのインスタンスに対しては、あたかも対応する基本データ型のコレクションのように取り扱うことができます。

2.2 ボクシング変換

ボクシング変換では、基本データ型（boolean、byte、char、short、int、long、float、double）が、対応するラッパークラス（Boolean、Byte、Character、Short、Integer、Long、Float、Double）のインスタンスに自動変換されます。ボクシング変換では、変換対象の値の型に対応したラッパークラスの参照型へ変換されます。

変数宣言と同時に行われている初期化のコードを見てください。

```
Integer x = 1;
```

ここでは、Integer クラスのインスタンスを明示的に生成しなくても、右辺の式の型は int 型ですので、自動的に値として 1 を持つ Integer クラスのインスタンスが生成されて、変数 x に代入されます。しかし、次のコードでは、右辺の式の型は short 型ですので、ボクシング変換では、Integer ではなく Short へ変換されます。その結果は、Integer へ代入できません。

```
short s = 1;
Integer x = s; // NG! コンパイルエラー
```

同様に、次のコードでも代入ができません。

```
int i = 1;
Short x = i; // NG! コンパイルエラー
```

右辺の型は int 型なので Integer へ変換され、Short への代入はコンパイルエラーとなります。しかし、次のようにキャストすることで代入可能となります。

```
int i = 1;
Short x = (short)i;
```

整数リテラルの場合には、代入先のラッパークラスに対応する基本データ型に代入可能であれば、ボクシングにより代入されます。

```
Short s = 20; // 暗黙に int から short へ、そして、Short へ変換
```

最後に、ボクシング変換により、どんな式の値であっても、Object 型の参照へ代入できることになることにも注意してください。

```
Object o = 1 + 2 + 3;
o = 1.25 * 3.0;
```

2.3 アンボクシング変換

アンボクシング変換では、ラッパークラス（Boolean、Byte、Character、Short、Integer、Long、Float、Double）への参照が、対応する基本データ型（boolean、byte、char、short、int、long、float、double）の値へと自動的に変換されます。

次のコードでは、Integer 型への参照である x を int 型の変数 y へ代入しています。

```
Integer x = 1; // ボクシング
int y = x;     // アンボクシング
```

この場合、特にキャストすることなく、y に代入することが可能です。アンボクシングでは、参照型の値が null である場合には、NullPointerException がスローされます。

2.4 メソッド呼び出し

メソッド呼び出しの際のパラメータに関しても、ボクシングが行われます。次のプログラムを見てください。

```
class BoxingParameter {
    public static void main(String[] args) {
        showInteger(10);
        showInteger(20);
    }
    public static void showInteger(Integer i) {
        System.out.println("value of integer = " + i);
    }
}
```

showInteger メソッドのパラメータは、Integer クラスへの参照型ですので、メソッド呼び出しの際に、ボクシングが行われ、実行結果は次のようになります。

```
value of integer = 10
value of integer = 20
```

メソッド呼び出しの際に、アンボクシングも行われます。次のプログラムを見てください。

```
class UnboxingParameter {
    public static void main(String[] args) }
        showInt(new Integer(10));
        showInt(new Integer(20));
    }
    public static void showInt(int i) }
        System.out.println("value of int = " + i);
    }
}
```

showInt メソッドのパラメータは int 型ですので、メソッド呼び出しの際にアンボクシングが行われます。実行結果は次のようになります。

```
value of int = 10
value of int = 20
```

2.4.1 オーバーロードされたメソッドの呼び出し

どのメソッドを呼び出すかをコンパイラが決定する場合、まず、ボクシングもアンボクシングも行わずに呼び出せるメソッドを検索します。該当するメソッドが定義されていない場合には、ボクシングやアンボクシングを行った結果で、呼び出せるメソッドを検索します。その結果、該当するメソッドが定義されていれば、ボクシングやアンボクシングを行って該当メソッドを呼び出すようにコンパイルされます。次のコードを見てください。

```
class Overload {

    static void foo(int x, int y) {
        System.out.println("foo(int,int) is invoked");
    }

    static void foo(Integer x, Integer y) {
        System.out.println("foo(Integer,Integer) is invoked");
    }

    public static void main(String[] args) {
        foo(1, 1);
        foo(new Integer(1), new Integer(1));
    }
}
```

このプログラムでは、foo(1,1) および foo(new Integer(1), new Integer(1)) の呼び出しでは、ボクシングやアンボクシングを行わなくても、引数の型が一致するメソッドが見つかりますので、実行結果は次のようになります。

```
foo(int,int) is invoked
foo(Integer,Integer) is invoked
```

もし、foo(1, new Integer(1)) とすると、第 1 引数をボクシングすることで呼び出し可能なメソッドと、第 2 引数をアンボクシングすることで呼び出し可能なメソッドの両方が存在します。その結果、呼び出すメソッドが決定できないため、コンパイルエラーとなります。

2.5 演算子との関係

従来、Integer などのラッパークラスはあくまでもラッパーであり、+や-などの演算子を直接適用することはできませんでした。しかし、ボクシング/アンボクシングの機能導入により、基本データ型と同じように演算子が適用できるようになります。

```
Integer x = 1;
x++;
++x;
```

```
x = x * x;
x = x / 3;
x--;
--x;
x = -x;
x = x << 3;
x = x >> 3;
x = x >>> 1;
x = x & 0xff;
x = (x == 0) ? 1 : 2;
```

このようにあたかも基本データ型のように扱えますが、乱用するのは良くありません。なぜなら、演算結果を元の変数 x へ代入するごとに、Integer クラスのインスタンスが新たに生成されるからです。そのため、処理時間も要しますし、当然、多くのインスタンスが生成され、不要になったインスタンスが回収されなければなりません。

Boolean クラスのインスタンスを除くラッパークラスのインスタンスは、関係演算子（>、>=、<、<=）のオペランドとして使用された場合には、アンボクシングが行われて基本データ型のように扱えます。

```
Integer x = 1;
Integer y = 2;

if (x < y)
    // ...

if (x <= y)
    // ...

if (x > y)
    // ...

if (x >= y)
    // ...
```

Boolean クラスのインスタンスに関しては、論理演算子（&、|、&&、^、!、||）、および?:演算子[*1]のオペランドとしても使用可能です。

```
Boolean b = /* ... */ ;
Boolean c = /* ... */ ;

if (b & c)
    // ...

if (b && c)
    // ...
```

[*1] ?:演算子については、仕様が変更されていますので、詳しくは、9.1.4 節を参照してください。

```
if (b || c)
    // ...

int x = b ? 1 : 0;
```

注意しなければならない演算子は、等価演算子（==、!=）です。等価演算子の左右のオペランドのどちらかが数値リテラルか基本データ型であり、もう片方がラッパークラスのインスタンスであれば、アンボクシングが行われて、値が等しい（*value equality*）か否かが検査されます。しかし、両方のオペランドがラッパークラスのインスタンスであると、それらのインスタンスへの参照が等しい（*reference identity*）か否かが検査されます。

リリース 1.4 までは関係演算子や論理演算子のオペランドとして、参照型のインスタンスを使用することは文法的に禁止されていました。つまり、リリース 1.4 までは関係演算子や論理演算子に対して参照型のインスタンスを指定しているコードは存在しなかったので、リリース 5.0 ではアンボクシングの導入により許可されるようになりました。しかし、リリース 1.4 までは等価演算子のオペランドとして参照型のインスタンスを使用できましたので、リリース 5.0 でもその仕様は踏襲されています。

つまり、次のコードでは、左右のオペランドが参照型ですので、リリース 1.4 でもリリース 5.0 以降でも同じ動作をする必要があります。

```
Integer x = new Integer(500);
Integer y = new Integer(500);

if (x == y) // 常に false
  // ...
```

ボクシングに関してはさらに注意が必要です。たとえば、次の x、y、z があったとします。

```
Integer x = 100;
int     y = 100;
Integer z = 100;
```

この場合、x == y では、x がアンボクシング変換により基本データ型へ変換されてから比較されるので true です。さらに、x == z も true となります。なぜ、そうなるのでしょうか。

言語仕様では、基本データ型によっては、値をボクシングにより対応するラッパークラスのインスタンスへ変換した場合に、ある範囲内の値であれば常に同じ参照が返されると規定されています。boolean 型と byte 型のデータは、ボクシング変換された場合には、同じ値に対して同じインスタンスの参照を返します。char 型であれば、\u0000 から \u007F の範囲で同じインスタンスの参照を返します。int 型と short 型であれば、-128 から 127 の範囲で同じインスタンスの参照を返します。基本データ型の値をボクシングにより参照型へ変換した場合には、同じ値に対して同じインスタンスの参照が返される仕様が理想ですが、現実的ではありません。その結果、このような実用的な仕様になっています。したがって、上記の例で、100 ではなく、300 が x、y、z に代入されるとボクシングにより新たなインスタンスが生成されますので、x == z が false となります。

2.6 フロー制御文

従来は boolean 型の値しか指定できなかったフロー制御文の次に示す *boolean-expression* 部分に、Boolean クラスのインスタンスを指定することが可能となります。

```
if (boolean-expression)
    statement1
else
    statement2

while (boolean-expression)
    statement

do
    statement
while (boolean-expression)

for (init-expression; boolean-expression; inc-expression)
    statement
```

2.7 switch 文

switch 文では、Integer、Short、Byte、Character の各クラスのインスタンスも使用できます。

```
Integer i = /* ... */ ;

switch (i) {
  case 0:
      // ...
  case 1;
      // ...
}
```

2.8 ラッパークラスの valueOf メソッド

ボクシング/アンボクシングは、コンパイラによって自動的に行われます。ボクシングの際に、ラッパークラスのインスタンスが自動的に生成されます。従来、Long クラス、Integer クラス、Short クラス、Byte クラス、Character クラス、Float クラス、Double クラスでは、対応する基本データ型の値を指定してインスタンスを生成するには、コンストラクタを呼び出すしか方法がありませんでした。しかし、リリース 5.0 からは、static ファクトリーメソッドである valueOf メソッドが追加されていま

す。valueOf メソッドは、Boolean クラスにはリリース 1.4 から入っていましたので、すべてのラッパークラスで統一的に追加されたことになります。

ラッパークラスの中で、Boolean クラスと Byte クラスの valueOf メソッドは、事前に用意されたインスタンスを返します。Character クラスの valueOf メソッドは、\u0000 から \u007F の範囲であれば事前に用意されたインスタンスを返します。Integer クラスと Short クラスの valueOf メソッドは、値が-128 から 127 の範囲であれば事前に用意されたインスタンスを返します。

このように事前に用意されたインスタンスを返すことで、ボクシング/アンボクシングで頻繁に、インスタンスが生成・回収されることを回避しています。もちろん、範囲外の値であれば、その都度インスタンスが生成されます。なお、Float クラス、Double クラス、Long クラスの valueOf メソッドは、どのような値であっても新たなインスタンスを生成して返します。

コレクションから int が引かれたら
ラッパークラスにほとほと嘆きたくなる
でも *Tiger* がくれば、もう涙は流さない
散々殴りつけてやる
– *Joshua Bloch*

第 3 章

拡張 for 文

While Iterators have their uses
They sometimes strangle us like nooses
With enhanced-for's deadly ray
Iterator's kept at bay
– Joshua Bloch

イテレータを使用して、コレクションから要素を参照する場合には、`while` 文あるいは `for` 文を使用して書きます。しかし、要素を削除することなく参照するだけであれば、どちらの書き方も面倒ですし、間違いやすいです。リリース 5.0 では、それらを解決するために `for` 文が拡張されています。

3.1 概要

次のメソッドは、`String` クラスのインスタンスを要素に持つコレクションを受け取り、文字列長を合計して返すメソッドです。

```
int totalLengthOfStrings(Collection coll) {
    int result = 0;

    for (Iterator i = coll.iterator(); i.hasNext();) {
        String s = (String) i.next();
        result += s.length();
    }
    return result;
}
```

ジェネリックスを用いれば、`String` クラスのインスタンスのコレクションであることを明示的に書くことができて、キャストも不要となります。

```
int totalLengthOfStrings(Collection<String> coll) {
    int result = 0;

    for (Iterator<String> i = coll.iterator(); i.hasNext();) {
        result += i.next().length();  // キャストは不要
    }
    return result;
}
```

リリース 5.0 では、for 文が拡張されて、次のシンタックスの for 文がサポートされます。

```
for ( FormalParameter : Expression )
    Statement
```

この新たな for 文を用いて書き直すと、次の通り、より簡潔になります。

```
int totalLengthOfStrings(Collection<String> coll) {
    int result = 0;

    for (String s : coll) {
        result += s.length();
    }
    return result;
}
```

コレクションだけでなく、普通の配列についても拡張 for 文を使用することができます。その場合、配列要素をインデックス 0 から順番にループします。配列に適用したのが次のコードです。

```
int totalLengthOfStrings(String[] strings) {
    int result = 0;

    for (String s : strings) {
        result += s.length();
    }
    return result;
}
```

配列の配列の場合も、次のコードのようにループすることができます。

```
class ArrayOfArray {
    public static void main(String[] args) {
        int[][] intTable = {{0},
                            {0, 1},
                            {0, 1, 2},
                            {0, 1, 2, 3}};

        for (int[] ia : intTable) {
            for (int i : ia) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
}
```

このコードの実行結果は、次の通りです。

```
0
0 1
0 1 2
0 1 2 3
```

拡張 for 文内では、ラベル付き continue 文や break 文が従来通り使用できます。次のような for 文があったとします。

```
int[] intArray = /* ... */ ;

Label1:
for (final int intValue: intArray) {
    // ...
    continue Label1;
}
```

これは、コンパイラにより、次のように変換されます。

```
int[] intArray = /* ... */ ;

Label1:
for (int #i = 0; #i < intArray.lengh; #i++) {
    final int intValue = intArray[#i];
    // ...
    continue Label1;
}
```

ここで、#i は、コンパイラにより自動生成される変数であり、プログラム内の変数と名前が衝突しないように決められます。配列の場合だけでなく、Iterable インタフェースを実装した場合であっても、*FormalParameter* はループ内で宣言されたのと同じように扱われますので、final 修飾子を付けることが可能です。

3.1.1 注意事項

拡張 for 文では、コレクション内の各要素を順に 1 つずつ参照することしかできません。そのため、イテレーション中にイテレータの remove メソッドを用いてコレクションから要素を削除するコードを、拡張 for 文で書き直すことはできません。その場合には、従来通りの for 文を使用する必要があります。また、リストや配列などで、ループ中に要素のインデックスを取得することもできません。[*1]

3.2 Iterable インタフェース

拡張 for 文のシンタックスで、*Expression* 部分に指定できるのは、java.lang.Iterable インタフェースを実装したオブジェクトか配列です。Iterable インタフェースの定義は次のようになってい

[*1] すべてのケースを網羅するのではなく、多くのケースを単純な構文で網羅できるような仕様策定の結果です。

ます。

```
package java.lang;
import java.util.Iterator;

/** Implementing this interface allows an object to be the target of
 *  the "foreach" statement.
 */
public interface Iterable<T> {
    /**
     * Returns an iterator over a set of elements of type T.
     *
     * @return an Iterator.
     */
    Iterator<T> iterator();
}
```

リリース 5.0 では、java.util.Collection インタフェースが、Iterable インタフェースを拡張するように修正されています。当然、その結果として、Collection インタフェースおよび、それを拡張したインタフェースを実装しているクラスのインスタンスは、拡張 for 文で使用できることになります。

自分で作った何らかのコレクションであっても、直接 Iterable インタフェースを実装し、その要素を取り出す iterator メソッドを定義することで、拡張 for 文で使用できるようになります。その例が、1.3.1 節 (21 頁) の Stack クラスです。Stack クラスは Iterable インタフェースを実装しており、StackTest クラスでは、拡張 for 文を使用してスタックの内容の合計を計算しています。

イテレータが幅をきかしていた頃は
縄でぐるぐるに縛られてしまうようなこともあった
でも拡張された for の痛烈な光線があれば
イテレータも寄せ付けない
– *Joshua Bloch*

第 4 章

enum 型

The int-enum will soon be gone
Like a foe we've known too long.
With type safe-enum's mighty power
Our foe will bother us no more
– Joshua Bloch

Java 言語には、C/C++言語が提供している enum、すなわち**列挙型**（*enumerated type*）は、提供されていませんでした。しかし、C/C++言語が提供している enum の欠点を持たない**タイプセーフ enum**（*typesafe enum*）を、Joshua Bloch 氏は『Effective Java』[Bloch01] の項目 21「enum 構文をクラスで置き換える」で示しています。その Joshua Bloch 氏が中心となって JSR201 としての仕様策定活動を経て、新たな言語仕様として **enum 型**（*enum type*）がリリース 5.0 から導入されています。

4.1 単純な enum 型

リリース 5.0 で導入される enum 型を説明する前に、タイプセーフ enum を説明し、それらがリリース 5.0 ではどのように簡単に書けるようになるかを説明します。なお、タイプセーフ enum に関するコードは、『Effective Java』[Bloch01] から引用しています。タイプセーフ enum をより深く理解するには、併せて『Effective Java』[Bloch01] を読まれることをお勧めします。

タイプセーフ enum が紹介されるまでは、Java 言語では、定数を列挙するために次のようなコードが書かれていました。

```
// int enum パターン - 問題がある!
public class PlayingCard {
    public static final int CLUBS    = 0;
    public static final int DIAMONDS = 1;
    public static final int HEARTS   = 2;
    public static final int SPADES   = 3;
}
```

このコードで定義されている定数（CLUBS、DIAMONDS、HEARTS、SPADES）は int 型であり、それらの定数を引数に取るメソッドでは、パラメータの型を int と宣言することになります。

次のコードは、PlayingCard クラスの定数を受け取って、その定数が表すカードの種類を文字列に

変換するメソッドです。

```
String cardToString(int card) {
    switch (card) {
        case PlayingCard.CLUBS:
            return "clubs";
        case PlayingCard.DIAMONDS:
            return "diamonds";
        case PlayingCard.HEARTS:
            return "hearts";
        case PlayingCard.SPADES:
            return "spades";
        default:
            throw new IllegalArgumentException();
    }
}
```

この方法だと cardToString メソッドに任意の int 値を渡すことが可能なため、CLUBS、DIAMONDS、HEARTS、SPADES 以外のプログラムが想定していない値は、実行時にしか検出されません。また、PlayingCard クラスの定数値を参照しているすべてのクラスに、その定数値がコンパイル時に埋め込まれてしまいますので、定数値を変更した場合には参照しているすべてのクラスをリコンパイルする必要があります。

トランプのスーツだけを表すクラスを、タイプセーフ enum を使用して書くと次のようなコードになります。

```
// タイプセーフ enum パターン
public class Suit {
    public static final Suit CLUBS    = new Suit();
    public static final Suit DIAMONDS = new Suit();
    public static final Suit HEARTS   = new Suit();
    public static final Suit SPADES   = new Suit();
}
```

こうすることで、Suit クラスの定数を引数に取るメソッドでは、int 型ではなく、次のように Suit 型で受け取ることができるようになります。

```
String cardToString(Suit suit) { /* ... */ }
```

これで、cardToString メソッドには Suit 型のオブジェクトしか渡せませんので、誤った型のオブジェクトを渡そうとするとコンパイル時にエラーとなります。また、定数の並びを変更したり、新たな定数を追加しても、既存の定数が削除されない限り、すべてのクラスをリコンパイルする必要はありません。

リリース 5.0 では、予約語として enum が追加されていますので、先ほどの Suit クラスは、次のように書き換えることができます。

```
public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

この場合、public 宣言されていますので Suit はトップレベルのクラスと同じ扱いとなり、Suit.java ファイルに書かなければなりません。また、最後の定数の後に、カンマ（,）を書くことも許されています。

Suit.java ファイルをコンパイルすると Suit.class ファイルが生成されます。それを javap コマンドで調べると次の結果となります。

```
D:\Tiger\example\enum>javap Suit
Compiled from "Suit.java"
public final class Suit extends java.lang.Enum{
    public static final Suit CLUBS;
    public static final Suit DIAMONDS;
    public static final Suit HEARTS;
    public static final Suit SPADES;
    public static final Suit[] values();
    public static Suit valueOf(java.lang.String);
    static {};
}
```

enum 型は、コンパイラによって、タイプセーフ enum に似た形式のクラスに変換されていることが分かります。つまり、Java VM から見れば単なるクラスにしか過ぎません。この結果からも、次のことが分かります。

- コンパイルした結果のクラスは、java.lang.Enum クラスを継承している。
- 単純な enum をコンパイルした結果のクラスは、final と宣言されている。
- static メソッドである values メソッドと valueOf メソッドがある。

values メソッドは、宣言されているすべての **enum 定数**（*enum constant*）を要素として持つ配列を返します。この配列は、values メソッドが呼び出されるごとに生成されて返されます。Suit のすべての enum 定数を出力するコードは、次のように書くことができます。

```
for (Suit s: Suit.values())
    System.out.println(s);
```

valueOf メソッドは、引数として enum 定数名を受け取り、対応する enum 定数を返します。Enum クラスと enum 定数[*1]については、次に説明します。

4.2 java.lang.Enum クラス

java.lang.Enum クラスは、リリース 5.0 から導入された抽象クラスであり、次のように定義されています。

[*1] enum 定数のインポートについては、5.3 節で説明します。

```
package java.lang;
import java.io.Serializable;

public abstract class Enum<E extends Enum<E>>
        implements Comparable<E>, Serializable {

    private final String name;
    public final String name() { return name; }

    private final int ordinal;
    public final int ordinal() { return ordinal; }

    protected Enum(String name, int ordinal) {
        this.name = name;
        this.ordinal = ordinal;
    }

    public String toString() { return name; }

    public final boolean equals(Object other) {
        return this==other;
    }

    public final int hashCode() {
        return System.identityHashCode(this);
    }

    protected final Object clone()
                throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }

    public final int compareTo(E o) { /* ... */ }

    public final Class<E> getDeclaringClass() { /* ... */ }

    public static <T extends Enum<T>> T valueOf(
                        Class<T> enumType,
                        String name) { /* ... */ }
}
```

すべてのenum定数は、名前（name）フィールドを持っており、実際には、ソースコードで記述された名前と同じになります。たとえば、Suitで定義した、Suit.CLUBSは、"CLUBS"をnameフィールドの値として持ちます。nameメソッドとtoStringメソッドは、nameフィールドの値を返しています。nameメソッドは、finalと宣言されており、オーバーライドできませんが、toStringメソッドはオーバーライドはできます。

すべての enum 定数は ordinal メソッドを持っており、enum 型内の enum 定数の数値位置を返します。たとえば、次の enum 型を考えてみてください。

```
public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

Suit 型の定数値である CLUBS、DIAMONDS、HEARTS、SPADES に対して ordinal メソッドを呼び出すと、それぞれ順に、0、1、2、3 を返します。このメソッドは、EnumMap（4.8.1 節）や EnumSet（4.8.2 節）などの汎用の enum に基づくデータ構造が使用するように設計されていますので、そのようなデータ構造を設計するのでなければ、ordinal メソッドを使用することは推奨されていません。[*2]

equals メソッドは、final と宣言されて、自分自身と同じ参照値の場合に true となっていますので、列挙型のすべての同値のオブジェクトは、すべて同一であること（a==b が true ならば、a.equals(b) は true）が保証されます。同様に、hashCode メソッドもオーバーライドできないように final と宣言されています。さらに、clone メソッドでは、CloneNotSupportedException をスローし、オーバーライドされないように final と宣言されることで、enum 定数であるオブジェクトを複製できないようになっています。

言語仕様として、enum 型の宣言では、Object クラスの finalize メソッドをオーバーライドすることもできないと定義されています。

Comparable インタフェースを実装していますが、ソースコードで記述した順序になるように ordinal フィールドがコンパイラによって設定されます。

getDeclaringClass メソッドは、enum 定数であるオブジェクトに対して、それを宣言しているクラスの Class オブジェクトを返します。valueOf メソッドは static メソッドであり、enum 型の Class オブジェクトと enum 定数名を渡すことで、該当する enum 定数を表すオブジェクトを返します。

4.3 enum 定数の名前空間

enum 宣言はクラスに変換されるため、enum で定義される定数は C/C++言語での enum と異なり個別の名前空間を持ちます。たとえば、次のような宣言では定数として同じ名前（BLOOD）が登場しますが、全く別物として取り扱われるため C/C++言語のように名前の衝突は発生しません。

```
enum Orange { NAVEL, TEMPLE, BLOOD }
enum Fluid { BLOOD, SWEAT, TEARS }
```

4.4 定数固有の振舞い

タイプセーフ enum では、メソッドを追加することが可能でした。たとえば、次のコードは toString メソッドを実装したタイプセーフ enum です。

[*2] 『Effective Java 第 2 版』[Bloch08] の項目 31「序数の代わりにインスタンスフィールドを使用する」参照。

```
// タイプセーフ enum パターン
public class Suit {
    private final String name;

    private Suit(String name) { this.name = name; }

    public String toString()  { return name; }

    public static final Suit CLUBS    = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS   = new Suit("hearts");
    public static final Suit SPADES   = new Suit("spades");
}
```

定数であってもSuitクラスのインスタンスですので、toStringメソッドを持っていることになります。Java言語でのタイプセーフenumとC/C++言語でのenumは、この点が大きく異なります。さらに、リリース5.0のenum型を使用すると次のようにコードを書くことができます。

```
// リリース 5.0 の enum 型を使用
public enum Suit {
    CLUBS("clubs"), DIAMONDS("diamonds"), HEARTS("hearts"), SPADES("spades");

    private final String name;

    Suit(String name) { this.name = name; }
    public String toString()  { return name; }
}
```

enum定数宣言をセミコロン（;）で終わらせたあと、通常のクラスと同様に、フィールド、コンストラクタ、メソッドを書くことができます。コンストラクタは暗黙にprivate宣言されており、publicやprotectedのアクセス修飾子を付けることはできません。コンストラクタは、必要に応じて複数宣言することも可能です。フィールドやメソッドには、普通のクラスと同様にアクセス修飾子を付けることができます。static宣言されたフィールドやメソッドを定義することもできます。

『Effective Java』[Bloch01]には、タイプセーフenumに、**定数固有の振舞い**（*constant-specific behavior*）を実装させた例として、次のコードが掲載されています。

```
// 定数に振舞を持たせたタイプセーフ enum
public abstract class Operation {
    private final String name;

    Operation(String name)   { this.name = name; }

    public String toString() { return this.name; }

    // 定数で表された算術操作を行う
    abstract double eval(double x, double y);
```

```
    public static final Operation PLUS = new Operation("+") {
        double eval(double x, double y) { return x + y; }
    };

    public static final Operation MINUS = new Operation("-") {
        double eval(double x, double y) { return x - y; }
    };

    public static final Operation TIMES = new Operation("*") {
        double eval(double x, double y) { return x * y; }
    };

    public static final Operation DIVIDED_BY = new Operation("/") {
        double eval(double x, double y) { return x / y; }
    };
}
```

リリース 5.0 の enum 型を使用すると、このタイプセーフ enum を使用したコードは、よりすっきりと次のように書き直すことができます。

```
// リリース 5.0 の enum を使用
public enum Operation {
    PLUS("+")       { double eval(double x, double y) { return x + y; }},
    MINUS("-")      { double eval(double x, double y) { return x - y; }},
    TIMES("*")      { double eval(double x, double y) { return x * y; }},
    DIVIDED_BY("/") { double eval(double x, double y) { return x / y; }};

    private final String name;
    Operation(String name)   { this.name = name; }
    public String toString() { return this.name; }

    // 定数で表された算術操作を行う
    abstract double eval(double x, double y);
}
```

ここでは、enum 型として定義された Operation 内に抽象メソッドである eval を定義しています。この例で分かるように、通常のクラス宣言と異なり、enum 型そのものを abstract と宣言する必要はありませんし宣言できません。この例では、eval メソッドは、enum 定数ごとにその実装が定義されています。もちろん、抽象メソッドだけでなく、普通のメソッドもオーバーライドすることができます。このように、enum 定数の個々に固有の振る舞いを定義した場合、PLUS などのオブジェクトは、Operation のインスタンスではなく、Operation を継承したサブクラスのインスタンスです。そのサブクラスは、コンパイラにより自動的に生成されます。

したがって、Operation.java ファイルをコンパイルすると、Operation.class ファイルだけでなく、それらのサブクラス用.class ファイルも同時に生成されます。この場合、それらのサブクラスは

final と宣言されます。一方、Operation を直接継承したクラスを作成することはできません。enum 型で定義されたコンストラクタは、暗黙に private 宣言されたものと見なされますので、サブクラス化することはできません。

enum 型の宣言では、インタフェースを実装することも可能です。たとえば、次のようなインタフェース定義があったとします。

```
public interface Operative {
    double eval(double x, double y);
}
```

このインタフェースを実装する形式で Operation を再度定義すると次のようになります。

```
public enum Operation implements Operative {
    PLUS("+")      { public double eval(double x, double y){ return x + y; }},
    MINUS("-")     { public double eval(double x, double y){ return x - y; }},
    TIMES("*")     { public double eval(double x, double y){ return x * y; }},
    DIVIDED_BY("/"){ public double eval(double x, double y){ return x / y; }};

    private final String name;
    Operation(String name)   { this.name = name; }
    public String toString() { return this.name; }
}
```

implements で指定したインタフェースが定義したメソッドは、直接実装することもできますし、この例のように、enum 定数の個々で実装する場合には、abstract の eval メソッドを宣言する必要はありません。また、個々の定数の定義では、abstract メソッドをオーバーライドする例だけを示してきましたが、必要に応じて個々の enum 定数で、toString メソッドをオーバーライドすることも可能です。

4.5 enum 型と switch 文

C/C++言語では、enum の値は整数値として取り扱えるため、switch 文の case ラベルで使用することができます。Java 言語の enum 定数は実際にはオブジェクトへの参照ですが、switch 文で使用できます。次のコードは、Suit を受け取りその色を返すものです。

```
public class SuitColor {
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    public static String suitColor(Suit suit) {
        switch (suit) {
            case CLUBS:
            case SPADES:
                return "black";
            case DIAMONDS:
            case HEARTS:
                return "red";
```

```
            default:
                throw new AssertionError("Unknown " + suit);
        }
    }

    public static void main(String[] args) {
        for (Suit s: Suit.values())
            System.out.println(s + " is " + suitColor(s));
    }
}
```

この例では、Suit は SuitColor クラス内に定義されており、static と宣言されていなくても、SuitColor クラス内で static 宣言された内部クラスと同じ扱いとなります。

switch 文の case ラベルでは、Suit.CLUBS と書かずに CLUBS と書いています。switch 文の case ラベルだけが、このように enum 定数を直接書くことができます。他の場合、特定の enum 定数を指定するには、Suit.CLUBS などのように書かなければなりません。つまり、switch 文ではなく if 文で書くと次のように書かなければなりません。

```
public class SuitColor {
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    public static String suitColor(Suit suit) {
        if (suit == Suit.CLUBS || suit == Suit.SPADES)
            return "black";
        else if (suit == Suit.DIAMONDS || suit == Suit.HEARTS)
            return "red";
        else
            throw new AssertionError("Unknown " + suit);
        }
    }

    public static void main(String[] args) {
        for (Suit s: Suit.values())
            System.out.println(s + " is " + suitColor(s));
    }
}
```

enum 定数は、実際にはオブジェクトですが、case ラベルとして null を書くことはできません。さらに、switch 文が実行される際に enum 値が null だった場合には、NullPointerException がスローされます。

switch 文を使用した例では、default ラベルを書いて AssertionError をスローしていることに注意してください。すべての定数が列挙されているので不要に見えますが、将来、新たな定数が追加された場合を考慮すると必要となります。

4.6 enum 型の制約

ここで、enum 型に関するいくつかの制約をまとめておきます。

【制約 1】enum 型の宣言では、クラス宣言と異なり、クラス修飾子である abstract と final は使用できません。
【制約 2】enum 型のインスタンスを、直接 new オペレータで生成することはできません。
【制約 3】enum 型を、直接サブクラス化することはできません。
【制約 4】enum 型の宣言で、コンパイラが自動生成する values メソッド、valueOf メソッド、および enum 定数と衝突するようなメンバーは宣言できません。
【制約 5】enum 型の宣言で、java.lang.Enum クラスで final 宣言されているメソッドである equals(Object)、hashCode()、clone()、compareTo(Object)、name()、ordinal()、getDeclaringClass() をオーバーライドすることはできません。また、Object クラスの finalize メソッドをオーバーライドすることもできません。
【制約 6】enum 型の宣言内のコンストラクタ、初期化ブロック、および、インスタンスフィールドの初期化子から、enum 定数を含む static のフィールドをアクセスすることはできません。
【制約 7】enum 定数に付けることができる修飾子は、アノテーション（第 7 章参照）だけです。
【制約 8】enum 型の中で宣言されているコンストラクタでは、super を使用してスーパークラスのコンストラクタを明示的に呼び出すことはできません。
【制約 9】リフレクション API での enum 定数の生成は禁止されています。

制約 6 は、enum 定数を表すインスタンスが生成される際にコンストラクタが呼ばれ、そのコンストラクタが static フィールドにアクセスすると依存関係が循環してしまうため、そのような循環を禁止するための制約となっています。

クラス内に宣言された enum は、そのクラスの static メンバーと同じ扱いになりますが、そのために発生する制約があります。それは、内部クラス（*inner class*）内では、enum 宣言をすることができないということです。内部クラス宣言では、final static フィールド以外の static メンバーを持つことはできません[*3]ので、内部クラス内で enum 宣言をすることはできません。つまり、次のようなコードは、書けません。

```
class Outer {
    class Inner {
        enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES } // NG! コンパイルエラー
    }
}
```

このコードをコンパイルすると、次のようにコンパイルエラーとなります。

[*3] 『プログラミング言語 Java 第 3 版』の 125 頁参照（『プログラミング言語 Java 第 4 版』の 119 頁参照）。

```
D:\Tiger\example\enum>javac Outer.java
Outer.java:3: 修飾子 enum をここで使うことはできません。
        enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
        ^
エラー 1 個
```

制約 9 は、リフレクション API を使用しても特定の enum 型のインスタンスを生成できないようにするための制約です。通常は、java.lang.reflect パッケージの Constructor クラスの newInstance メソッドを用いて、クラスのインスタンスを生成することができますが、enum 型のインスタンスを生成しようとすると、IllegalArgumentException がスローされます。この禁止に関しては、newInstance メソッドの契約となる Javadoc コメント内の IllegalArgumentException を記述した@exception タグ部分に、次の一行が書き加えられています。

if this constructor pertains to an enum type.

つまり、enum 型のコンストラクタであれば、例外をスローすると記述されています。

4.7 enum 定数のシリアライズ

java.lang.Enum クラスは Serializable インタフェースを実装していますが、readResolve メソッドは定義されていません。従来のタイプセーフ enum では、readResolve メソッドを実装しなければ、正しくディシリアライズできませんでした。[*4]

リリース 5.0 からは、シリアライズ/ディシリアライズの仕様が変更になっており、enum 定数は特別に処理されるようになっています。enum 定数をシリアライズする場合には、name フィールドだけがシリアライズされます。他のフィールドは一切シリアライズされません。そして、ディシリアライズでは、name フィールドの値を用いて、Enum.valueOf メソッドを使用して値オブジェクトを取得するようになります。

enum 型の宣言で、シリアライズ/ディシリアライズのためのカスタマイズ用メソッド（writeObject、readObject、readObjectNoData、writeReplace、readResolve）を定義しても、すべて無視されます。同様に、serialPersistentFields と serialVersionUID もすべて無視され、serialVersionUID は、常に 0L となります。

4.8 EnumMap クラスと EnumSet クラス

リリース 5.0 では、enum 定数をキーとしたマップやセットとして、最適な実装を提供する EnumMap クラスと EnumSet クラスがコレクションフレームワークに追加されています。

[*4] 『Effective Java』[Bloch01] の項目 21 参照

4.8.1 EnumMap クラス

EnumMap クラスはジェネリッククラスであり、特定の enum 型の enum 定数をキーとして受け付けるマップです。内部では指定された enum 型で定義されている enum 定数の個数分の配列を生成して、その配列でキーと値の組を管理しています。各 enum 定数の ordinal フィールド値を配列のインデックスとして使用しますので、他のマップ実装よりも高速に動作します。

EnumMap クラスを生成する場合には、コンストラクタの引数としてキーとなっている enum 型の Class オブジェクトを渡す必要があります。次の例では、Suit をキーとして、スーツの色を表す文字列を値とするマップを生成しています。

```
import java.util.EnumMap;

class EnumMapSample {
    enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    public static void main(String[] args) {
        EnumMap<Suit, String> map = new EnumMap<Suit, String>(Suit.class);

        map.put(Suit.CLUBS, "black");
        map.put(Suit.DIAMONDS, "red");
        map.put(Suit.HEARTS, "red");
        map.put(Suit.SPADES, "black");
    }
}
```

4.8.2 EnumSet クラス

EnumSet クラスはジェネリッククラスであり、特定の enum 型の enum 定数を要素として受け付けるセットです。内部では、指定された enum 型で定義されている enum 定数をすべて表現するのに十分な大きさのビットベクターを生成します。そのビットベクター[*5]で要素の集まりを管理していますので、コンパクトで高速な処理を提供します。

EnumSet クラスのインスタンスを new オペレータで直接生成することはできません。すべて static ファクトリーメソッドを使用して生成します。

`public static <E extends Enum<E>> EnumSet<E>` **noneOf(Class⟨E⟩ elementType)**

引数で指定された要素型の空の enum セットを返します。

`public static <E extends Enum<E>> EnumSet<E>` **allOf(Class⟨E⟩ elementType)**

引数で指定された要素型のすべての要素を持つ enum セットを返します。

[*5] enum 定数が 64 個までしか定義されていなければ、64 ビット長である long 型の変数をビットベクターとして使用します。64 個より多い場合には、long[] 型の配列をビットベクターとして使用します。

`public static <E extends Enum<E>> EnumSet<E>` **copyOf(EnumSet⟨E⟩ s)**
引数で指定された enum セットと同じ要素型の enum セットを返します。引数の enum セットと同じ要素が含まれる enum セットを返します。

`public static <E extends Enum<E>> EnumSet<E>` **copyOf(Collection⟨E⟩ c)**
引数で指定されたコレクションに含まれる要素を含むように初期化された enum セットを返します。

`public static <E extends Enum<E>> EnumSet<E>` **complementOf(EnumSet⟨E⟩ s)**
引数で指定された enum セットと同じ要素型の enum セットを返します。引数の enum セットに含まれていない要素を含む enum セットを返します。

`public static <E extends Enum<E>> EnumSet<E>` **of(E e)**
引数で指定された要素を持つ enum セットを返します。

`public static <E extends Enum<E>> EnumSet<E>` **of(E e1, E e2)**
引数で指定された要素を持つ enum セットを返します。

`public static <E extends Enum<E>> EnumSet<E>` **of(E e1, E e2, E e3)**
引数で指定された要素を持つ enum セットを返します。

`public static <E extends Enum<E>> EnumSet<E>` **of(E e1, E e2, E e3, E e4)**
引数で指定された要素を持つ enum セットを返します。

`public static <E extends Enum<E>> EnumSet<E>` **of(E e1, E e2, E e3, E e4, E e5)**
引数で指定された要素を持つ enum セットを返します。

`public static <E extends Enum<E>> EnumSet<E>` **of(E first, E... rest)**
引数で指定された要素を持つ enum セットを返します。任意の要素数の enum セットを生成できますが、引数が 5 個を超えない場合には、他の `of` メソッドが呼び出されます（6.5.2 節参照）。

`public static <E extends Enum<E>> EnumSet<E>` **range(E from, E to)**
引数で指定された要素と、その要素間のすべての要素を含む enum セットを返します。

`EnumSet` クラスは `Set` インタフェースを実装していますので、普通のセットとしての各種メソッドも提供されています。

C/C++言語の `enum` とは異なり、ある `enum` 型が定義している定数の範囲をイテレートすることができます。すべての定数をイテレートするためには、`values` メソッドを使用します。

```
public class Iteration1 {
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    public static void main(String[] args) {
        for (Suit s: Suit.values())
            System.out.println(s);
    }
}
```

ある特定の定数間のみをイテレートする場合には、`EnumSet.range` メソッドを使用します。

```
import java.util.EnumSet;

public class Iteration2 {
    public enum Day {
             SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }

    public static void main(String[] args) {
        System.out.print("Weekdays:");
        for (Day d: EnumSet.range(Day.MONDAY, Day.FRIDAY))
            System.out.print(" " + d);
        System.out.println();
    }
}
```

EnumSet.range メソッドを使用して、Day.MONDAY から Day.FRIDAY までのすべての enum 定数を要素として持つセットを生成していますので、その結果、MONDAY から FRIDAY までをイテレートして、表示することになります。

4.9 リフレクション API

java.lang.Enum クラスは、getDeclaringClass メソッドを定義しています。getDeclaringClass メソッドを enum 定数に対して呼び出すことで、その定数を定義している enum 型の Class オブジェクトが返されます。したがって、2 つの enum 定数 e1 と e2 は、次の条件が成り立てば、同一の enum 型で定義された定数であると見なされます。

```
e1.getDeclaringClass() == e2.getDeclaringClass()
```

getDeclaringClass メソッドで返される Class オブジェクトは、getClass メソッドで返される Class オブジェクトと同一であるとは限りません。たとえば、次の単純な例を考えてみます。

```
public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

この例では、次の条件は成り立ちます。

```
Suit.CLUBS.getClass() == Suit.CLUBS.getDeclaringClass()
```

この場合、クラスリテラルである Suit.class と同じものが、どちらも返されています。

次の例ではどうでしょうか。

```
public enum Suit {
    CLUBS("clubs")       { String color() { return "black"; }},
    DIAMONDS("diamonds") { String color() { return "red";   }},
    HEARTS("hearts")     { String color() { return "red";   }},
    SPADES("spades")     { String color() { return "black"; }};

    private final String name;
```

```
        Suit(String name) { this.name = name; }
        public String toString()  { return name; }
        abstract String color();
    }
```

この場合には、Suit.CLUBS.getClass() が Suit.CLUBS.getDeclaringClass() と同じであるという条件は成り立ちません。なぜならば、CLUBS は Suit のサブクラスのインスタンスだからです。しかし、以下の条件は成り立ちます。

```
Suit.class == Suit.CLUBS.getDeclaringClass()
```

enum に関連して、java.lang.Class クラスには isEnum メソッドと getEnumConstants メソッドが追加されています。isEnum メソッドは、そのクラスオブジェクトが表しているクラスが enum 型であるかどうかを返します。getEnumConstants メソッドは、クラスオブジェクトが表しているクラスが enum 型である場合に、定義されているすべての enum 定数の配列を返します。そうでない場合には null を返します。あるクラスオブジェクトが enum 型を表していると分かった場合に、そのオブジェクトにあるフィールドが enum 定数なのか、あるいは通常のフィールドであるのかを判別するために、java.lang.reflect.Field クラスに isEnumConstant メソッドが追加されています。

int-enum はそのうち消える
長きにわたり知りすぎていた敵のように。
タイプセーフ enum の偉大な力をもってすれば
敵はもう我々をてこずらさない
– *Joshua Bloch*

第 5 章

static インポート

And from the constant interface
We shall inherit no disgrace
With static import at our side
Our joy will be unqualified
– Joshua Bloch

あるクラスの public 宣言された static メンバー（フィールド、メソッド、内部クラス）を外部から参照する場合には、リリース 1.4 までは、そのクラス名を指定して、static メンバーを参照する必要がありました。たとえば、Math クラスの abs メソッドを使用するには、次のように書かなければなりませんでした。

```
y = Math.abs(x);
```

Math クラスは、java.lang パッケージに属していますので、import 文を書く必要はありませんが、abs メソッドを使用するために、毎回 Math.abs と書かなければなりません。リリース 5.0 からは、この煩わしさを解消するために、**static インポート**（*static import*）が新たに導入されています。

5.1 シンタックス

static メンバーをインポートするためのシンタックスは、次の通りです。

```
import static TypeName.Identfier;
```

あるいは、

```
import static TypeName.*;
```

たとえば、Math クラスのすべての static メンバーをインポートするためには、次のように宣言します。

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

こうすることで、Math クラスの abs メソッドへアクセスするために、単純に abs(x) と書くことができます。あるいは、Math クラスで定義されている定数である E や PI も Math. で修飾する必要がなくなります。

5.2 定数インタフェースの排除

『Effective Java』の項目 17「型を定義するためだけにインタフェースを使用する」では、いわゆる**定数インタフェース**（*constant interface*）の使用を強く否定しています。定数インタフェースは、定数を外部へ提供する static final のフィールドだけから構成されています。それらの定数を使用するクラスは、クラス名で定数名を修飾する面倒さを回避するために、そのインタフェースを実装したりします。『Effective Java』[Bloch01] の項目 17（第 2 版では項目 19）では、定数インタフェースの例として、次のコードが掲載されています。

```
// 定数インタフェースパターン - 使用してはいけない!
public interface PhysicalConstants {
    // アヴォガドロ数 (1/mol)
    static final double AVOGADROS_NUMBER   = 6.02214199e23;

    // ボルツマン定数 (J/K)
    static final double BOLTZMANN_CONSTANT = 1.3806503e-23;

    // 電子の質量 (kg)
    static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

このようなインタフェースは使用すべきではなく、クラス名で定数を修飾する手間を要しても、次に示されるように定数ユーティリティクラスを使用するようにと説明されています。

```
// 定数ユーティリティクラス
public class PhysicalConstants {
  private PhysicalConstants() { }  // インスタンス化を防ぐ

  public static final double AVOGADROS_NUMBER   = 6.02214199e23;
  public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
  public static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

static インポートにより、このような定数ユーティリティクラスを定義しても、次のように定数名だけで使用することが可能となります。

```
import static PhyscialConstants.*;

class Guacamole {
    public static void main(String[] args) {
        double moles = /* ... */ ;
        double molecules = AVOGADROS_NUMBER * moles;
        // ...
    }
}
```

5.3 enum 定数のインポート

enum 定数も実際には static フィールドですので、同様に static インポートを適用することが可能です。ただし、enum 型が無名パッケージではなく、特定のパッケージに属していなければ、static インポートすることはできません。たとえば、Suit.java ファイルに次のように無名パッケージとして定義したとします。

```
public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

この場合、他のソースコードから、この Suit を次のように、static インポートすることはできません。

```
import static Suit.*;  // NG! コンパイルエラー
```

もし、次のように Suit が宣言されていたとします。

```
package mypackage;

public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

この場合には、次のように static インポートすることができます。

```
import static mypackage.Suit.*;  // OK
```

次に、クラス内に次のように定義された Suit について説明します。

```
package mypackage;

public class Foo {
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
    public static void main(String[] args) {
        for (Suit s: Suit.values())
            System.out.println(s);
    }
}
```

この Foo クラス内の CLUBS を、外部から参照するには以下の方法があります。

- import mypackage.Foo; とインポートして、Foo.Suit.CLUBS と参照する。
- import mypackage.Foo.Suit とインポートして、Suit.CLUBS と参照する。
- import static mypackage.Foo.Suit.*とインポートして、CLUBS と参照する。

残念ながら、Suit を定義している Foo クラス内では、switch 文の case ラベルを除いて、Suit.CLUBS と参照するしかありません。

5.4 ネストしたインタフェースとネストしたクラス

static インポートは、static メンバーのインポートを行います。したがって、static メンバーとして宣言されているネストしたインタフェースとネストしたクラスもインポートできます。クラス内で宣言されたネストしたインタフェースは、static 宣言されていなくても static メンバーです。同様に、インタフェース内に定義されたクラスやインタフェースも static メンバーです。たとえば、java.util.Map インタフェースの内で定義されている Entry インタフェースは、従来は次のようにしてインポートしていました。

```
import java.util.Map.Entry;
```

従来通りインポートすることも可能ですが、static インポートすることも可能です。

```
import static java.util.Map.Entry;
```

5.5 メソッドの static インポート

static インポートで、メソッドをインポートする場合に、オーバーロードされた複数のメソッドが存在すると、そのすべてのメソッドがインポート対象と見なされます。たとえば、java.util.Arrays クラスには、sort メソッドが 18 個あります。したがって、次のような import 文では、それらすべてが対象となります。

```
import static java.util.Arrays.sort;
```

sort メソッドが呼び出された際に、そのパラメータからどのメソッドが呼び出されるかが決まります。そのため、同じ名前のメソッドであっても、シグニチャが異なっていれば、再度 static インポートすることができます。たとえば、java.util.Collections クラスは、Arrays クラスで定義されている sort メソッドとシグニチャが異なる sort メソッドを 2 つ定義しています。したがって、次のように両方 static インポートすることが可能です。

```
import static java.util.Arrays.sort;
import static java.util.Collections.sort;
```

このように同一名のメソッドを複数のクラスから static インポートした場合、呼び出されたメソッドのシグニチャと同じメソッドが 2 つ以上存在しなければ、問題ありません。もし、2 つ以上存在するとコンパイルエラーになります。

メソッドの static インポートに関しては、もう 1 つ注意点があります。クラス名で修飾されることなく呼び出されているメソッドと同じ名前のメソッドが、クラスですでに定義されている場合には、メソッド検索において、static インポートによるメソッドは対象とされません。次の例では、private の引数なしの exp メソッドが定義されていますが、一方で、Math クラスの exp メソッドを static インポートしようとしています。

```
import static java.lang.Math.exp;

class ImportTest {

    private void exp() { /* ... */ }

    public double tanh(double x) {
        return (exp(x) - exp(-x)) / (exp(x) + exp(-x)); // NG! コンパイルエラー
    }
}
```

tanh メソッド内で使用されている exp メソッドは、double 型の引数を受け取りますので、Math クラスの exp(double) メソッドが期待されるのですが、ImportTest クラスではすでに同じ名前のメソッドが定義されています。このように、(継承したメソッドも含めて) すでに同じ名前のメソッドが定義されている場合、static インポートによるメソッドのインポートは行われません。ImportTest クラスの例では、引数なしの exp メソッドが存在するため、Math クラスからの static インポートは行われずに、次のようにコンパイルエラーとなります。

```
ImportTest.java:8: exp() (ImportTest 内) を (double) に適用できません
        return (exp(x) - exp(-x)) / (exp(x) + exp(-x));
```

5.6 注意事項

static インポートの使用は強く推奨されるものではなく、定数インタフェースを避けたり、定数をローカル変数へコピーするのを避けたりする場合などに限定されるべきです。そうでなければ、ソースコード上に現れるフィールド名やメソッド名などが、そのソースコードで定義されているクラスのフィールドやメソッドなのか、あるいは、static インポートされた他のクラスのメンバーなのかの区別がつかなくなってしまい、ソースコードの可読性が低下してしまう結果となります。

また、コンパイル時に値が決定できる定数式で初期化された基本データ型、あるいは、null ではない定数式で初期化された String 型の static final フィールドを static インポートした場合には、その定数値がコンパイル時に埋め込まれてしまいますので、static インポートした定数値が変更された場合にはリコンパイルする必要があります。

定数インタフェースから
不名誉までも引き継ぐことなかれ
static インポートを味方につければ
喜びも限りなく広がる
– *Joshua Bloch*

第 6 章

可変長パラメータ

O joyless nights, o joyless days
Our programs cluttered with arrays
With varargs here, we needn't whine;
We'll simply put the args inline
– Joshua Bloch

リリース 5.0 からは、コンストラクタやメソッドの引数を任意の個数指定できる**可変長パラメータ**（*variable arity parameter*）[*1]が導入されています。また、既存クラスの中には、可変長パラメータを用いて定義が書き換えられたものや、可変長パラメータを持つ新たなメソッドが追加されたものもあります。

6.1 シンタックス

可変長パラメータは、コンストラクタやメソッドの最後のパラメータに指定でき、次のように書きます。

```
type ... variableName
```

type の前に `final` 修飾子を付けることもできます。ただし、*type* として、パラメータ化された型を指定することはできません。型名と変数の間に、`...` を書くことで、可変長パラメータであることを示します。このパラメータは、プログラム上は、次のように宣言されたのと同じ扱い[*2]がされます。

```
type[] variableName
```

つまり、*variableName* は、配列型として取り扱うプログラミングをします。

次の例は、引数で渡された値の合計を返すメソッドです。

[*1] arity とは、コンピュータ用語で、メソッドや演算子が取る引数の数を意味します。つまり、可変数のパラメータという意味になります。

[*2] Java 言語仕様 [JLS05] では、`T...` 型と宣言された**仮パラメータ**（*formal parameter*）の型は、`T[]` となると定義されています。

```
class Sum {
    static int sum(int ... numbers) {
        int total = 0;
        for (int i: numbers)
            total += i;
        return total;
    }

    public static void main(String[] args) {
        System.out.printf("%d%n", sum(1, 2, 3, 4, 5));
    }
}
```

この例では、sum メソッドは引数の和を返します。numbers は配列として扱われますので、その要素が単純に加算されて返されています。

もし、可変長パラメータ部分が全く省略されてメソッドが呼び出された場合には、長さ 0 の配列が渡されます。つまり、引数なしで sum() と呼び出すと、合計値は 0 が返されます。

では、リリース 5.0 で可変長パラメータを使用しているクラスやメソッドを次に紹介します。

6.2 使用例：PrintStream.printf メソッド

PrintStream クラスには、可変長パラメータを使用した以下の printf メソッドが追加されています。

```
public PrintStream printf(String format, Object ... args)
public PrintStream printf(Locale l, String format, Object ... args)
```

printf メソッドは、C 言語での printf 関数のように書式（9.10.3 節参照）を指定して出力することができます。[*3] 2 番目の形式は、ロケールを指定してフォーマットします。たとえば、次のように書くことができます。

```
class PrintfTest {
    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int sum = x + y;
        System.out.printf("%d + %d = %d%n", x, y, sum);
    }
}
```

[*3] C/C++言語で printf 関数を使用したことがある人であれば、サンプルコードで、改行するのに \n ではなく、%n が使用されていることに気づかれるかと思います。println メソッドで改行を行った場合には、一文字の'\n' で改行するのではなく、システムプロパティの line.separator で返される文字列が使用されています。したがって、システムに依存した行区切りを使用して改行するには、%n を使用します。

printf メソッドの可変長パラメータの型は、Object 型と宣言されていますが、ボクシング（第 2 章）により、x、y、sum の各値は、対応するラッパークラスのインスタンス、つまり、Integer クラスのインスタンスに変換されてから渡されます。

printf と同様に、format メソッドもあり、どちらも同じ処理を行います。また、フォーマットされた文字列を生成するために、String クラスに static の format メソッドも追加されています。

6.3 使用例：リフレクション API

次のようなクラスがあったとして、そのメソッドである aMethod を、リフレクションを使用して呼び出すことを考えてみます。

```
class AClass {
    public void aMethod(String strArg, int intArg) {
        System.out.printf("strArg = %s, intArg = %d%n", strArg, intArg);
    }
}
```

このメソッドを従来のリフレクションを使用して呼び出すには、次のようなコードを書かなければなりませんでした。

```
import java.lang.reflect.*;

class AClassInvoker {
    public static void main(String[] args) throws Exception {
        Class c = Class.forName("AClass");
        Method m = c.getMethod("aMethod",
                        new Class[] {String.class, int.class});
        m.invoke(c.newInstance(), new Object[] {"Hello", new Integer(10)});
    }
}
```

リリース 5.0 では、Class クラスの getMethod メソッドや Method クラスの invoke メソッドは、可変長パラメータを使用して書き直されていますので、次のように書き直すことができます。

```
import java.lang.reflect.*;

class AClassInvoker {
    public static void main(String[] args) throws Exception {
        Class c = Class.forName("AClass");
        Method m = c.getMethod("aMethod", String.class, int.class);
        m.invoke(c.newInstance(), "Hello", 10);
    }
}
```

getMethod メソッドと invoke メソッドのメソッド呼び出しは、従来と比較すると簡潔に記述できていることに注目してください。

6.4 可変長パラメータと配列パラメータの互換性

可変長パラメータは、実質的には配列と同じ扱いになりますので、可変長パラメータとして定義されているパラメータに対して、実際に配列を渡すことはできます。たとえば、次のコードを見てください。

```
class ArrayPassingTest {
    public static void main(String[] args) {
        Object[] msg = new String[] {"How", "are", "you?"};
        System.out.printf("%s %s %s%n", msg);
    }
}
```

printf メソッドの定義は、printf(String format, Object ... args) となっています。したがって、String クラスの配列を Object[] 型の msg に代入して、printf メソッドに渡すことで、配列の最初の要素がフォーマットに対応する最初の引数と見なされ、2 番目の要素が第 2 引数、3 番目の要素が第 3 引数として処理されます。コンパイルして実行すると次のようになります。

```
D:\Tiger\example\var>javac ArrayPassingTest.java

D:\Tiger\example\var>java ArrayPassingTest
How are you?
```

可変長パラメータとして配列を渡すと、配列の要素自身が可変長パラメータの個々のパラメータと見なされることになります。一方、配列を可変長パラメータの 1 つとして渡すには、次の例のように Object にキャストする必要があります。

```
class ArrayPassingTest2 {
    public static void main(String[] args) {
        Object[] msg = new String[] {"How", "are", "you?"};
        System.out.printf("%s%n", (Object)msg);
    }
}
```

このコードをコンパイルして実行すると次のようになります。

```
D:\Tiger\example\var>javac ArrayPassingTest2.java

D:\Tiger\example\var>java ArrayPassingTest2
[Ljava.lang.String;@10b30a7
```

可変長パラメータの 1 つとして配列のオブジェクトそのものが渡されますので、単純に Object クラスの toString メソッドが呼び出された結果となっています。

パラメータとして配列が宣言されているメソッドに対して、可変長パラメータが宣言されたような呼び出しはできません。つまり、次のようなコードは書けません。

```
class Sum {
    static int sum(int[]  numbers) { // 配列パラメータ
        int total = 0;
        for (int i: numbers)
            total += i;
        return total;
    }

    public static void main(String[] args) {
        System.out.printf("%d%n",
               sum(1, 2, 3, 4, 5)); // NG! コンパイルエラー
    }
}
```

メソッドのオーバーロードに関して、注意する必要があります。可変長パラメータは、指定された型の配列としてコンパイルされます。したがって、可変長パラメータでパラメータを定義しているメソッドを、同じ型の配列をパラメータとして持つメソッドでオーバーロードすることはできません。

```
class Overload {
    void foo(String... x) { /* ... */ }
    void foo(String[] x)  { /* ... */ } // NG! コンパイルエラー
}
```

2 つ目の foo メソッドは、1 つ目の foo メソッドと同じシグニチャとしてコンパイルされますので、コンパイルエラーになります。

6.5 注意事項

6.5.1 可変長パラメータと null

メソッドの可変長パラメータ宣言されたパラメータに対して null を渡す場合には、注意が必要です。明示的にキャストしない限り、長さが 1 の配列が生成されて、その要素が null となります。もし、配列の参照として null を渡したい場合には、型の配列としてキャストしなければなりません。たとえば、可変長パラメータの宣言されている型が Integer とすると、何もキャストしないで null を渡すと、new Integer[] {null} を渡したのと同じになります。配列の参照として null を渡したければ (Integer[])null とします。

したがって、可変長パラメータとして宣言したのであっても、null がその参照値として渡されてくることもありますので、必要ならばパラメータが null であるかを検査する必要があります。

6.5.2 可変長パラメータとメソッドのオーバーロード

可変長パラメータを使用したメソッドを設計する場合には、本当に可変長パラメータが必要か否かを注意深く検討する必要があります。また、可変長パラメータを使用したメソッドをオーバーロードする

ようなメソッドを定義することは避けるべきです。なぜならば、どちらのメソッドを呼び出しているのかを判断するのが困難になるからです。

メソッド呼び出しにおいて、オーバーロードしているどのメソッドが呼び出されるかに関しては、まずは、可変長パラメータを使用していないメソッドが優先的に選択されます。その選択には、引数をボクシングあるいはアンボクシングした結果呼び出すことができるメソッド（2.4.1 節参照）も含まれます。該当するメソッドがない場合にのみ、可変長パラメータを使用しているメソッドが選択されます。たとえば、次のようなメソッド定義があったとします。

```
public void foo(int i) { /* ... */ }
public void foo(int i, int j) { /* ... */ }
public void foo(int... ints) { /* ... */ }
```

`foo(1)` と呼び出すと、最初の `foo` メソッドが呼び出されます。`foo(1,1)` では、2 番目 `foo` メソッドが呼び出されます。決して、可変長パラメータを使用した 3 番目の `foo` メソッドが呼び出されることはありません。4.8.2 節で説明した `EnumSet` クラスには、可変長パラメータの `of` メソッドだけでは非効率なため、パラメータ数が固定のオーバーロードされた `of` メソッドが意図的に定義されています。

あぁ寝ても覚めてもつまらない
配列でプログラムは散らかり放題
でも可変長パラメータがあれば愚痴もなくなる
ただ引数を並べればいいだけ
– *Joshua Bloch*

第 7 章
アノテーション

As for noble metadata
I'll have to sing its praises later
Its uses are so numerous
To give their due, I'd miss the bus
– Joshua Bloch

リリース 5.0 からは、プログラムの動的振る舞いに影響を与えることなく、フィールド、メソッド、クラスなどに**アノテーション**（*annotation*:注釈）を付けることができるようになります。アノテーションは、それ自身が何かを行うのではなく、他のツールがアノテーションを読み取って、何らかの処理を行うためのものです。アノテーション機能は、様々なアノテーションとそれに関連したツールにより、ソフトウェア開発の負荷を減らすことを目的としており、そのための基盤を提供します。

しかし、逆に言えば、アノテーションは、それを処理する何らかのツールが無いと何の役にも立たないことになります。本章では、Java コンパイラがサポートしている標準アノテーションを中心に説明します。なお、サンプルコードの多くは、JSR175 の "Public Draft Specification" [*1] から引用しています。

アノテーション型は、型（インタフェース）[*2] を定義します。`java.lang` パッケージに宣言されている標準アノテーション型を使用する場合には、インポートする必要がありません。しかし、`java.lang` パッケージ以外で定義されているアノテーション型は、インポートする必要があります。また、アノテーション型の名前空間は、型の名前空間に属しますので、通常のクラス、インタフェース、`enum` により定義される型名と衝突しないように注意する必要があります。

7.1 標準アノテーション型

`java.lang` パッケージに定義されている標準アノテーション型としては、次の 3 つがあります。

- `Override`
- `Deprecated`
- `SuppressWarnings`

[*1] http://jcp.org/aboutJava/communityprocess/review/jsr175/index.html
[*2] アノテーション型は、`.java` ファイルに記述し、コンパイルされて `.class` ファイルが生成されます。そのクラスファイルにアノテーション型に関する情報が記録されます。

7.1.1 @Override アノテーション

@Override アノテーションは、メソッドがスーパークラスのメソッドをオーバーライドしているはずであることを明示するのに使用します。したがって、このアノテーションはメソッドにしか付けることができません。

このアノテーションが付けられているメソッドをコンパイルする際に、コンパイラはスーパークラスのメソッドをオーバーライドしているかを検査します。もし、オーバーライドしていなければ、コンパイルエラーとなります。次のコードを見てください。

```
class Foo {
    public void foo() { }
}

class Bar extends Foo {
    @Override public void bar() { } // NG! コンパイルエラー
}
```

bar メソッドの前に、アノテーション（@Override）が書かれていますので、bar メソッドは、スーパークラスのメソッドをオーバーライドしているはずだと宣言しています。しかし、実際には、オーバーライドしていませんので、コンパイルエラーとなります。また、次の例を見てください。

```
class Foo {
    @Override public boolean equals(Foo o) { // NG! コンパイルエラー
        // ...
    }
}
```

この例では、Object クラスの equals メソッドをオーバーライドして定義しているつもりになっていますが、実際には、オーバーロードしている[*3] だけであり、@Override アノテーションが付いていなければ、コンパイルエラーになりませんので、気づかないことになります。このように、@Override アノテーションは、タイプミスや引数の型宣言の誤りでオーバーライドしていないのに、オーバーライドしていると勘違いしてしまうような間違いを防止するのに使用することができます。

これらの例から分かるようにアノテーションの記述は、@の後にアノテーション型（ここでは、Override）を書くことで行います。**アノテーション型**（*annotation type*）の宣言方法については、7.2 節で説明します。また、同一のアノテーション型のアノテーションを 2 つ以上付けることはできません。

[*3] equals メソッドのパラメータの型が Object 型ではなく、Foo 型と宣言されています。したがって、Object クラスの equals メソッドとは異なるシグニチャとなっていることに注意してください。

7.1.2 @Deprecated アノテーション

@Deprecated アノテーションは、それが付けられている要素の使用が推奨されないことを示します。たとえば、メソッドの使用が推奨されてないことは、@deprecated Javadoc タグで示していましたが、アノテーションでも指定できるようになります。

```
public class Foo {
    @Deprecated public void foo() { }
}
```

@deprecated Javadoc タグの代わりに、@Deprecated アノテーションを使用しても、javadoc コマンドは認識して処理します。なぜなら、Deprecated アノテーション型の宣言では、（後述する）@Documented アノテーションが付けられており、javadoc コマンドなどのドキュメンテーションツールが認識して処理すべきアノテーション型と宣言されているからです。

しかし、表示方法が若干異なっています。@deprecated Javadoc タグをメソッドに使用すると、そのメソッドの説明部分に “推奨されていません” と表示されます。しかし、@Deprecated アノテーションは、メソッドの定義の一部として表示されます。たとえば、上記の Foo クラスの場合には、foo メソッドの定義の一部として、@Deprecated アノテーションが、次のように表示されます。

```
@Deprecated
public void foo()
```

@Deprecated アノテーションの目的は、将来的に、@deprecated Javadoc タグを置き換えることです。つまり、コンパイラがソースコード上のコメントを解析して何らかの振る舞いを行うこと自体が、コンパイラが行うべき処理ではないということです。

7.1.3 @SuppressWarnings アノテーション

@SuppressWarnings アノテーションは、コンパイラが出す警告メッセージを抑制するためのアノテーションです。言語仕様の拡張に伴い、コンパイラは様々な警告メッセージ（9.15 節参照）を出力するようになってきています。通常は、コンパイル時の警告メッセージに対しては、ソースコードを修正して、取り除くことが望ましいです。しかし、ソースコードを修正して取り除くことができず、かつ、プログラマが警告メッセージの原因となっている箇所のコードの正しさに確信しているような場合には、@SuppressWarnings アノテーションを使用して警告メッセージ出力を抑制するようにします。たとえば、無検査警告メッセージを取り除くには、@SuppressWarnings("unchecked") アノテーションを使用します。

@SuppressWarnings アノテーションは、型（クラスやインタフェース）、フィールド、メソッド、パラメータ、コンストラクタ、ローカル変数で使用できます。そのために、クラス全体に@SuppressWarnings アノテーションを使用することで、警告メッセージを安易に抑制できますが、そのような使用は好ましくありません。@SuppressWarnings アノテーションを使用する場合には、その抑制範囲を限定するよ

うにしてください。[*4]

7.1.4 契約の一部としてのアノテーション

@Deprecated アノテーションは、javadoc コマンドから生成されるオンラインドキュメンテーションに反映されますが、@Override アノテーションと@SuppressWarnings アノテーションは、反映されません。これらの 2 つのアノテーションは、コンパイル時のコンパイラの動作を指示したものであり、それらのアノテーションは、メソッドやフィールドなどの契約の一部ではないからです。@Override アノテーションと@SuppressWarnings アノテーションが、契約の一部ではないと見なされるのは、それらのアノテーション型の宣言において、（後述する）@Documented アノテーションが指定されていないからです。

7.1.5 アノテーションは修飾子

アノテーションは修飾子であり、public や private などのアクセス修飾子や他の修飾子より前に書かなければならないという規則はありません。しかし、アノテーションを書く場合には、一般に最初の修飾子として書きます。

7.2 アノテーション型宣言

アノテーション型の宣言は、通常の interface 宣言と似ており、interface の前に@を付けることで行います。[*5]アノテーション型の宣言は、インタフェースを宣言できる場所であれば、宣言できますし、インタフェースと同様の可視性を持たせることができます。また、その名前空間は、インタフェースと同じ名前空間に属します。本章では、トップレベルでの宣言だけを紹介します。

7.2.1 マーカーアノテーション型

最も簡単なアノテーション型宣言は、次のような、何も本体がない**マーカーアノテーション**（*marker annotation*）型の宣言です。

```
/**
 * この型のアノテーションは、API 要素が暫定であり、変更されることがある
 * ことを示す。
 */
public @interface Preliminary { }
```

この Preliminary アノテーション型は、指定されたメソッドなどが、まだ最終版ではなく、変更になるかもしれないという意味のアノテーションを定義しています。したがって、次のように foo メソッ

[*4] 『Effective Java 第 2 版』[Bloch08] の項目 24「無検査警告を取り除く」を参照。

[*5] @と interface は、厳密には別のトークンとして取り扱われますので、その間に空白を入れても構いません。しかし、空白を入れることは推奨されていません。

ドに付けることにより、まだ変更になるかもしれないメソッドであることを示します。

```
public class Foo {
    @Preliminary public void foo() { }
}
```

この場合、foo メソッドには、@Preliminary アノテーションが付けられていることが、Foo.class に記録されます。ここで注意しなければならないのは、@Preliminary アノテーションを使用しても、その意味を理解して処理するツールやソフトウェアがなければ何も意味を持たないということです。

7.2.2 単一要素アノテーション型

アノテーション型では、メソッド宣言もできます。メソッド宣言は、アノテーション型の**要素**（*element*）を定義することになります。次の Copyright アノテーション型を見てください。

```
/**
 * API 要素に著作権表示を関連付ける
 */
public @interface Copyright {
    String value();
}
```

戻り値型が String である value 要素を定義しています。このように、要素が 1 つしかないアノテーションは、**単一要素アノテーション**（*single-element annotation*）と呼ばれ、要素名は value を用います。@Copyright アノテーションは、次のように使用できます。

```
@Copyright("2004 by Yoshiki Shibata")
public class Foo {
    @Preliminary public void foo() { }
}
```

これは、Foo クラスに@Copyright アノテーションを付けたことになります。そして、そのアノテーションの value メソッドで返される文字列が"2004 by Yoshiki Shibata"ということになります。

7.2.3 複数要素を持つアノテーション型

要素を 2 つ以上定義する場合には、任意の名前を使用することができます。次の例は、4 つの要素を定義したものです。

```
/**
 * API 要素が作られることになったエンハンス要求（RFE: request-for-enhancement）
 * を記述する。
 */
public @interface RequestForEnhancement {
    int    id();       // RFE に関連付けされている一意の ID
    String synopsis(); // RFE の概要
```

```
    String engineer(); // RFE を実装したエンジニア名
    String date();     // RFE が実装された日付
}
```

このように複数の要素を定義しているアノテーション型を使用して、その要素値を指定する場合には、要素名の後に=を書くことで次のように指定します。

```
@RequestForEnhancement(
    id       = 286874,
    synopsis = "Provide time-travel functionality",
    engineer = "Mr. Peabody",
    date     = "4/1/2004"
)
public static void travelThroughTime(Date destination) { /* ... */ }
```

また、次のように順番が異なっていてもよいです。

```
@RequestForEnhancement(
    date     = "4/1/2004",
    engineer = "Mr. Peabody",
    synopsis = "Provide time-travel functionality",
    id       = 286874)
public static void travelThroughTime(Date destination) { /* ... */ }
```

単一要素アノテーション型の場合には、要素名の指定が省略されていることになります。また、マーカーアノテーション型を使用した場合には、要素が無いため () も省略されていることになります。したがって、次のように書いても間違いではありません。

```
@Copyright(value = "2004 by Yoshiki Shibata") // 要素名を明示
public class Foo {
    @Preliminary() public void foo() { } // () を付けている
}
```

指定する要素が 1 つしかなく、その要素名が value である場合だけ、要素名を明示する必要がありません。つまり、単一要素アノテーション型でも、要素名を value 以外にすることができますが、その場合には、必ず要素名を指定して要素値を明示しなければなりません。

7.2.4 配列型の戻り値型

要素として定義されているメソッドの戻り値型に配列を指定することもできます。次の、Authors アノテーション型を見てください。

```
public @interface Authors {
    String[] value();
}
```

value メソッドの戻り値型は配列ですが、要素値として 1 つしか指定しない場合には、次のように書けます。

```
@Authors("Christie Golden")
public class HomeComing {
    public void read() { }
}
```

配列の要素として複数指定する場合には、次のように書きます。

```
@Authors({"Tom DeMarco", "Timothy Lister"})
public class Peopleware {
    public void read() { }
}
```

配列に関する制限事項として、ネストした配列、つまり、配列の配列は戻り値型としては指定できません。

```
public @interface Authors {
    String[][] value(); // NG! コンパイルエラー
}
```

7.2.5 Class 型の戻り値型

メソッドの戻り値型に Class 型を宣言することも可能です。次のような、フォーマットをするための Formatter インタフェースが定義されたとします。

```
public interface Formatter { /* ... */ }
```

この Formatter インタフェースを、アノテーション型の宣言で、次のように[*6]メソッドの戻り値型として使用することができます。

```
/**
 * アノテーションが付けられたクラスをプリティプリントする
 * ためのフォーマッタを指定する。
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface PrettyPrinter {
    Class<? extends Formatter> value();
}
```

ここで、次のように Formatter インタフェースを実装した BeautyPrinter クラスが用意されているとします。

```
public class BeautyPrinter implements Formatter { /* ... */ }
```

そうすると、次のように、この BeautyPrinter クラスのクラスリテラルを、@PrettyPrinter アノテーションの値として指定することができます。

[*6] @Retention アノテーションが付けられていますが、@Retention アノテーションについては、7.4.2 節で説明します。

```
@PrettyPrinter(BeautyPrinter.class)
class UglyCode { /* ... */ }
```

この UglyCode クラスをプリティプリントするには、BeautyPrinter クラスを使用することをアノテーションとして記述していることになります。

7.2.6 アノテーション型の戻り値型

メソッドの戻り値型にアノテーション型を宣言することも可能です。次のような Name アノテーション型が定義されたとします。

```
/**
 * 人名。このアノテーション型は、プログラム要素に直接注釈を付けるように
 * 設計されているのではなく、他のアノテーション型で要素を定義するために
 * 設計されている。
 */
public @interface Name {
    String first();
    String last();
}
```

この Name アノテーション型をメソッドの戻り値型に持つ、次のような Author アノテーション型を定義することができます。

```
/**
 * プログラム要素の著者を示す。
 */
public @interface Author {
    Name value();
}
```

この場合には、@Author アノテーションは、次のように記述します。

```
@Author(@Name(first = "Christie", last = "Golden"))
public class HomeComing {
    public void read() { }
}
```

7.2.7 デフォルト値の指定

アノテーション型の宣言においては、各要素のデフォルト値を宣言することが可能です。デフォルト値は、default 予約語を用いて次のように記述します。

```
// デフォルト値が定義された要素を持つアノテーション型宣言
public @interface RequestForEnhancement {
    int    id();       // デフォルト値なし。アノテーションごとに値を指定する。
```

```
    String synopsis(); // デフォルト値なし。アノテーションごとに値を指定する。
    String engineer()  default "[unassigned]";
    String date()      default "[unimplemented]";
}
```

この場合、engineer と date はデフォルト値が指定されていますので、次のように値が指定されていない場合には、デフォルト値が使用されます。

```
@RequestForEnhancement(
    id       = 286874,
    synopsis = "Provide time-travel functionality"
)
public static void travelThroughTime(Date destination) { /* ... */ }
```

もし、メソッドとして定義されたすべての要素にデフォルト値が定義されていると、そのアノテーション型は、マーカーアノテーションとしても使用可能です。また、複数のメソッドが定義され、デフォルト値が定義されていないメソッドが 1 つだけで、かつ、そのメソッド名が value ならば、単一要素アノテーションとしても使用可能です。

値が指定されず、デフォルト値が使用される場合、コンパイルされた時点で、そのデフォルト値があたかも書かれているようにはコンパイルされません。つまり、上記の travelThroughTime メソッドを含むソースコードをコンパイルすると、「RequestForEnhancement 型で定義されたデフォルト値が使用される」と記録されるだけであり、デフォルト値そのものがコピーされるのではありません。

7.3 制約事項

アノテーション型宣言は、コンパイルされると、通常のインタフェースとしてコンパイルされますが、次の制約事項があります。

- アノテーション型は、暗黙に java.lang.annotation.Annotation インタフェースを拡張していますので、extends を書くことはできません。
- メソッドは、パラメータを持つことはできません。
- アノテーション型は、ジェネリックスを使用して、ジェネリック化することはできません。
- 型パラメータを持つメソッドを定義することはできません。
- メソッドには、throws 節を書けません。
- メソッドの戻り値型は、基本データ型（char, byte, short, int, long, float, double）、String、Class、enum 型、アノテーション型か、これらの型の配列でなければなりません。

要素値およびデフォルト値の指定に関しては、以下の制約事項があります。

- メソッドの戻り値型が、基本データ型か String 型なら、定数しか指定できません。
- メソッドの戻り値型が、Class 型なら、クラスリテラルしか指定できません。

- メソッドの戻り値型が、enum 型なら、その enum 型の enum 定数しか指定できません。
- null は、指定できません。

Annotation インタフェースでは、いくつかのメソッドを定義していますが、アノテーション型の要素とは見なされません。また、Annotation インタフェース自身は、アノテーション型とは見なされませんし、Annotation インタフェースやアノテーション型を明示的に実装したクラスや拡張したインタフェースもアノテーション型とは見なされません。

メソッドの戻り値型に、アノテーション型を指定することができますが、定義しようとしている型を、直接的あるいは間接的に要素の戻り値型として指定することはできません。たとえば、次のように、定義しようとしているアノテーション型を、その定義内で使用することはできません。

```
@interface SelfRef {
    SelfRef value(); // NG! コンパイルエラー
}
```

さらに、次のような循環した定義もできません。

```
@interface Ping {
    Pong value(); // NG! コンパイルエラー
}

@interface Pong {
    Ping value();
}
```

7.4 標準メタアノテーション型

アノテーション型を定義する際に、そのアノテーション型の定義に使用可能な**標準メタアノテーション型**（*standard meta annotation type*）があります。それらは、java.lang.annotation パッケージに定義されており、以下のものがあります。

- Target
- Retention
- Documented
- Inherited

7.4.1 @Target アノテーション

@Target アノテーションは、アノテーション型の宣言において、宣言されているアノテーションが、どの要素に適用できるかを記述するためのものです。たとえば、Copyright アノテーション型の宣言で、そのアノテーションがメソッドや、フィールドには適用できず、クラス、インタフェース、enum 型、およびアノテーション型にしか適用できないことを示すには、次のように宣言します。

```
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Target(ElementType.TYPE)
public @interface Copyright {
    String value();
}
```

@Target アノテーションが付けられて、その値として ElementType.TYPE が指定されていますので、Copyright アノテーション型は、クラス宣言などの型宣言にしか使用できないことになります。そのため、@Copyright アノテーションを、メソッドに次のように適用するとコンパイルエラーとなります。

```
public class Foo {
    @Copyright("2004 by Y.Shibata") // NG! コンパイルエラー
    public void foo() { }
}
```

では、Target アノテーション型がどのように定義されているかを見ていきます。定義は次の通りです。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    ElementType[] value();
}
```

@Documented アノテーション、@Retention アノテーション、@Target アノテーションの 3 つが付けられていることが分かります。最初の 2 つは、この後説明します。注意して欲しいのは、Target アノテーション型宣言で、そのアノテーション自身を使用していることです。@Target アノテーションの値として ElementType.ANNOTATION_TYPE が指定されていますので、@Target アノテーションは、アノテーション型にしか付けることができないことになります。

@Target アノテーションで指定できる値は、ElementType の配列とされており、ElementType は、次の enum 型です。

```
package java.lang.annotation;

public enum ElementType {
    /** Class, interface (including annotation type), or enum declaration */
    TYPE,
    /** Field declaration (inlcudes enum constants) */
    FIELD,
    /** Method declaration */
    METHOD,
    /** Parameter declaration */
    PARAMETER,
```

```
        /** Constructor declaration */
        CONSTRUCTOR,
        /** Local variable declaration */
        LOCAL_VARIABLE,
        /** Annotation type declaration */
        ANNOTATION_TYPE,
        /** Package declaration */
        PACKAGE
    }
```

ElementType.TYPE は、型（*type*）、すなわち、クラス、インタフェース、アノテーション型、enum の宣言に付けるアノテーション型であることを示します。ElementType.FIELD、ElementType.METHOD、ElementType.PARAMETER、ElementType.CONSTRUCTOR、ElementType.LOCAL_VARIABLE は、それぞれ、フィールド、メソッド、パラメータ、コンストラクタ、ローカル変数に付けるアノテーション型であることを示しています。

@Target **アノテーション**として、ElementType.PARAMETER が含まれるアノテーション型のアノテーションは、コンストラクタやメソッドのパラメータだけでなく、例外をキャッチする catch 節にも使用できます。つまり、catch 節でキャッチされるべき例外は、パラメータですので、final 宣言することもできますし、アノテーションを付けることもできます。

最後の定数である ElementType.PACKAGE は、パッケージ宣言に付けるアノテーション型であることを示します。リリース 5.0 からは、package-info.java ファイル（9.9.2 節）[*7]が、パッケージに対する情報（アノテーションによるメタ情報やドキュメンテーション）を記述するためのファイルとなります。パッケージ宣言にアノテーションを付ける場合には、package-info.java ファイルにパッケージ宣言だけを行い、そのパッケージ宣言の前に付けます。また、そのパッケージ宣言の前にドキュメントコメントを書くことで、javadoc コマンドは、そのパッケージのためのドキュメンテーションを生成してくれます。したがって、従来の package.html ファイルは不要となります。

アノテーション型の宣言において、@Target アノテーションが明示されていない場合には、そのアノテーション型はすべての要素に適用できるという意味になります。

7.4.2 @Retention アノテーション

Retention アノテーション型は、あるアノテーション型で記述されたアノテーション情報がどこまで反映されるかを示すために使用されます。7.1.1 節で説明した@Override アノテーションは、メソッドがオーバーライドされていることを規定しているだけであり、オーバーライドされているかどうかは、コンパイル時に判定されます。したがって、@Override アノテーションは、ソースコード上の記述だけであり、クラスファイルに記録する必要は全くありません。Override アノテーション型は、次

[*7] package-info.java ファイルは、コンパイルすることができます。ただし、パッケージ宣言されているだけの場合には、コンパイルしても何も生成されません。パッケージ宣言の前にアノテーションが付いていれば、コンパイルした結果として、package-info.class ファイルが生成されます。public ではないクラスやインタフェースを package-info.java ファイル内に記述することも可能ですが、記述するのは避けるべきです。

のように定義されています。

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override { }
```

@Retention アノテーションの値として、RetentionPolicy.SOURCE が指定されていますので、@Override アノテーションは、ソースコード上だけのアノテーションであることを示しています。

Retention アノテーション型は、次のように定義されています。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    RetentionPolicy value();
}
```

前述の@Target アノテーションの値として、ElementType.ANNOTATION_TYPE が指定されていますので、Retention アノテーション型のアノテーションは、アノテーション型の宣言にしか付けられないことを示しています。

@Retention アノテーションで指定できる値は、次の RetentionPolicy 型の enum 定数です。

```
package java.lang.annotation;

public enum RetentionPolicy {
    SOURCE,
    CLASS,
    RUNTIME
}
```

RetentionPolicy.SOURCE は、ソースコード上だけのアノテーションであることを示します。RetentionPolicy.CLASS は、クラスファイルに記録されますが、JVM にはロードされないアノテーションであることを示します。RetentionPolicy.RUNTIME は、クラスファイルに記録されると同時に、JVM にもロードされて、リフレクション API（8.1 節参照）を使用して読み取り可能なアノテーションであることを示します。@Retention アノテーションが付けられていないアノテーション型は、デフォルトで RetentionPolicy.CLASS が指定されたと見なされます。

ローカル変数に付けられたアノテーションは、そのアノテーション型の宣言で RetentionPolicy として何が指定されていたとしても、クラスファイルには記録されません。

7.4.3 @Documented アノテーション

@Deprecated アノテーションで説明したように、アノテーションそのものがメソッドやクラスの契約の一部であり、javadoc コマンドなどで処理された際に反映されなければならないアノテーション型を明示するのに、@Documented アノテーションを付けます。Documented アノテーション型は、マーカーアノテーションであり、次のように定義されています。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Documented { }
```

たとえば、Copyright アノテーション型を次のように定義したとします。

```
import java.lang.annotation.*;

@Documented
@Target(ElementType.TYPE)
public @interface Copyright {
    String value();
}
```

そして、次のように Foo クラスの定義で使用したとします。

```
@Copyright("2004 by Y.Shibata")
public class Foo {
    public void foo() { }
}
```

javadoc コマンドを使用して、Foo クラスのオンラインドキュメンテーションを生成すると、Foo クラスに関する説明部分が次のように生成されます。

```
@Copyright(value="2004 by Y.Shibata")
public class Foo
extends java.lang.Object
```

7.4.4 @Inherited アノテーション

クラス宣言に付けられたアノテーションが、サブクラスに自動的に継承されるアノテーション型であることを示すのが@Inherited アノテーションです。Inherited アノテーション型は、次の定義に示されるマーカーアノテーションです。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Inherited { }
```

@Inherited アノテーションを付けて宣言されたアノテーション型が、クラス宣言にアノテーションを付けるために使用された場合だけ有効となります。そのアノテーションをインタフェース宣言に付けても、クラス内のフィールド宣言やメソッド宣言に付けても継承されません。

たとえば、Copyright アノテーション型を次のように宣言したとします。

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface Copyright {
    String value();
}
```

@Target アノテーションが指定されていませんので、すべての要素に適用できるアノテーションとなります。このアノテーションを次のように Super クラスに適用し、さらに、Super クラスを継承した Sub クラスを作成したとします。

```
@Copyright("2004 by Yoshiki Shibata")  // 継承される
class Super {
    @Copyright("2004") // 継承されない
    public void foo() { /* ... */ }
}

class Sub extends Super {
    public void foo() { /* ... */ }
}
```

8.1.2 節で説明するリフレクション API を使用して、Sub クラスにアノテーションが付けられているかを調べると、継承されたアノテーションの値を取得することができます。一方、オーバーライドしている foo メソッドに対してはアノテーションは継承されませんので、値を取得することはできません。

7.5 アノテーションの読み込み

最初に述べたように、アノテーションをソースコード上に記述しても、それを読み込んで処理するツールが必要となります。そのようなツールを作成するためには、アノテーションを読み込むための API が必要となります。

アノテーションのために、リフレクション API が拡張されています。しかし、リフレクション API は、@Retention アノテーションの値として、RetentionPolicy.RUNTIME が付けられたアノテーション型の情報しか読み取れません。なぜならば、リフレクション API は、JVM 上にロードされたクラス情報を参照するからです。

一方で、RetentionPolicy が SOURCE や CLASS となっているアノテーションに関しては、どのようにして読み取ったら良いかという問題が残ります。そのためのツールとして、apt（*Annotation Processor Tool*）コマンド が、リリース 5.0 からは含まれています。apt コマンドは、アノテーションのパーサーの役割を果たしてくれますので、アノテーションに応じた処理を自分で書くことができます。たとえば、アノテーションが記述された Java のソースコードから別の Java のソースコードを生成する処理を書くことができます。ただし、apt コマンドでも、ローカル変数に付けられたアノテーションを読み取ることはできません。ローカル変数に付けられたアノテーションを読み取る必要がある場合

には、読み取るためのツールを自分で作成しなければなりません。

7.6　アノテーションの今後

今後、アノテーションを使用して、コードの自動生成や各種ファイルを自動的に生成するために、様々なライブラリの仕様拡張が行われるでしょう。たとえば、「Web Services Metadata for the Java Platform」（JSR181）、「Enterprise JavaBeans 3.0」（JSR220）、「JDBC 4.0 API Specification」（JSR221）などです。これらの新たな仕様では、**開発の容易性**（*Ease of Development*）を目的として、アノテーションが導入されます。

気高いメタデータに関して言えば
あとで賛辞をもって礼賛しないと。
使い道があまりに多岐にわたるものだから
きちんと説明しようとしているとバスにも乗り遅れてしまいそう
– *Joshua Bloch*

第 8 章

アノテーション処理プログラミング

アノテーションは、そのアノテーション型の定義において、@Retention アノテーションの値として指定される RetentionPolicy の値に応じて、JVM にロードされたり、ロードされないがクラスファイルに記録されるだけであったり、あるいは、ソースコード上に記述されるだけだったりします。アノテーションは、それを読み取って何らかの処理をするプログラムがなければ意味がありません。

8.1 リフレクション API

@Retention アノテーションの値として RetentionPolicy.RUNTIME が指定されたアノテーション型を用いたアノテーションは、JVM にロードされます。そのようにロードされたアノテーションは、リフレクション API を使用して読み取ることができます。そのようなアノテーションを読み取るために、次の java.lang.reflect.AnnotatedElement インタフェースが追加されています。

```
package java.lang.reflect;
import java.lang.annotation.Annotation;

public interface AnnotatedElement {
    boolean isAnnotationPresent(Class<? extends Annotation> annotationType);
    <T extends Annotation> T getAnnotation(Class<T> annotationType);
    Annotation[] getAnnotations();
    Annotation[] getDeclaredAnnotations();
}
```

リフレクション API で、AnnotatedElement インタフェースを直接あるいは間接的に実装しているクラスは、java.lang.Class、java.lang.Package、java.lang.reflect.Constructor、java.lang.reflect.Field、java.lang.reflect.Method です。これらのクラスにより、パッケージ、クラス、コンストラクタ、フィールド、enum 定数[*1]、メソッドに付けられたアノテーションを読み取れることになります。

AnnotatedElement インタフェースの各メソッドの説明は、次の通りです。

boolean isAnnotationPresent(Class⟨? extends Annotation⟩ annotationType)

パラメータで指定されたアノテーション型のアノテーションが付けられていれば、true を返しま

[*1] enum 定数は実際にはフィールドであり、Field クラスに追加された isEnumConstant メソッドで確認します。

す。このメソッドは、主にマーカーアノテーションが付けられているかどうかを検査するのに使用します。

`<T extends Annotation> T` **getAnnotation(Class⟨T⟩ annotationType)**
パラメータで指定されたアノテーション型のアノテーションが付けられていれば、そのアノテーションを返します。アノテーションが付けられていなければ、`null` が返されます。

`Annotation[]` **getAnnotations()**
付けられているすべてのアノテーションの配列が返されます。何もアノテーションが付けられていなければ、長さが 0 の配列が返されます。

`Annotation[]` **getDeclaredAnnotations()**
継承することなく、直接付けられているアノテーションの配列が返されます。直接付けられているアノテーションがなければ、長さが 0 の配列が返されます。

コンストラクタおよびメソッドのパラメータに付けられたアノテーションは、`Constructor` クラスと `Method` クラスに追加された次のメソッドにより読み取れます。

`public Annotation[][]` **getParameterAnnotations()**
パラメータに付けられたアノテーションの配列の配列を返します。配列の並びは、パラメータの宣言順です。パラメータを持たない場合には、長さが 0 の配列が返されます。

また、`Class` クラスに次のメソッドも追加されています。

`public boolean` **isAnnotation()**
この `Class` で表されるオブジェクトがアノテーション型であれば、`true` を返します。その場合、`isInterface` メソッドも `true` を返します。なぜなら、アノテーション型は、インタフェースでもあるからです。

8.1.1 マーカーアノテーションの読み取り

マーカーアノテーションの場合には、そのアノテーションが付けられているかどうかだけを判断できれば良いです。次のマーカーアノテーション `Preliminary` があるとします。

```
import java.lang.annotation.*;

/**
 * この型のアノテーションは、API 要素が暫定であり、変更されることがある
 * ことを示す。
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface Preliminary { }
```

この `Preliminary` アノテーション型の宣言には、`@Retention` アノテーションの値として、`RetentionPolicy.RUNTIME` が付けられていることに注意してください。`@Retention` アノテーショ

ンが明示的に付けられていないアノテーション型は、RetentionPolicy.CLASS となりますので、リフレクション API を使用して読み取ることはできません。

@Preliminary アノテーションを使用した次の Foo クラスがあったとします。

```
public class Foo {
    @Preliminary public void foo() { }
}
```

Foo クラスの foo メソッドに、@Preliminary アノテーションが付けられているかどうかは、次のように判定することができます。

```
Method m = Foo.class.getMethod("foo");
boolean isPreliminary = m.isAnnotationPresent(Preliminary.class);
```

8.1.2 単一要素アノテーションの読み取り

単一要素アノテーションの読み取りでは、値の読み取りは、value メソッドを通して行うことができます。次の単一要素アノテーションがあるとします。

```
import java.lang.annotation.*;

/**
 * API 要素に著作権表示を関連付ける
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface Copyright {
    String value();
}
```

@Copyright アノテーションを付けた次の Foo クラスがあったとします。

```
@Copyright("2004 by Yoshiki Shibata")
public class Foo {
    public void foo() { }
}
```

Foo クラスに付けられている@Copyright アノテーションの読み取りは、次のコードで行うことができます。

```
Copyright copyright = Foo.class.getAnnotation(Copyright.class);
if (copyright == null)
    System.out.println("No @Copyright");
else
    System.out.println("@Copyright = " + copyright.value());
```

このように、Foo クラスに付けられた@Copyright アノテーションは、Copyright インタフェース型で受けて、その要素である value メソッドにより値を読み取ることができます。

8.1.3 複数要素アノテーションの読み取り

複数要素アノテーションの読み取りも、単一要素アノテーションと同様に行うことができます。次の RequestForEnhancement アノテーション型が定義されていたとします。

```
import java.lang.annotation.*;

/**
 * API 要素が作られることになったエンハンス要求（RFE: request-for-enhancement）
 * を記述する。
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface RequestForEnhancement {
    int    id();       // RFE に関連付けされている一意の ID
    String synopsis(); // RFE の概要
    String engineer(); // RFE を実装したエンジニア名
    String date();     // RFE が実装された日付
}
```

@RequestForEnhancement アノテーションを付けた次の TimeTravel クラスがあったとします。

```
import java.util.Date;

class TimeTravel {

    @RequestForEnhancement(
        id       = 286874,
        synopsis = "Provide time-travel functionality",
        engineer = "Mr. Peabody",
        date     = "4/1/2004")
    public void travelThroughTime(Date destination) {  }
}
```

TimeTravel クラスの travelThroughTime メソッドに付けられている@RequestForEnhancement アノテーションの読み取りは、次のコードで行うことができます。

```
Method m = TimeTravel.class.getMethod("travelThroughTime", Date.class);
RequestForEnhancement rfe = m.getAnnotation(RequestForEnhancement.class);
if (rfe == null)
    System.out.println("No @RequestForEnhancement");
else
    System.out.printf("id = %d, synopsys = %s, engineer = %s, date = %s%n",
                      rfe.id(), rfe.synopsis(), rfe.engineer(), rfe.date());
```

8.1.4 Class 型の戻り値型の読み取り

Class 型を要素の戻り値型として持つアノテーションも、読み取ることができます。92 頁で示した以下のインタフェースとクラスで説明します。

```
public interface Formatter { /* ... */ }

/**
 * アノテーションが付けられたクラスをプリティプリントする
 * ためのフォーマッタを指定する。
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface PrettyPrinter {
    Class<? extends Formatter> value();
}

public class BeautyPrinter implements Formatter { /* ... */ }

@PrettyPrinter(BeautyPrinter.class)
class UglyCode { /* ... */ }
```

ここで、最後の UglyCode クラスに付けられている@PrettyPrinter アノテーションの読み取りは、次のコードで行うことができます。

```
PrettyPrinter printer = UglyCode.class.getAnnotation(PrettyPrinter.class);
if (printer == null)
    System.out.println("No @PrettyPrinter");
else {
    Class<? extends Formatter> formatterClass = printer.value();
}
```

このコードで、UglyCode クラスに付けられている@PrettyPrinter アノテーションの値を読み取ろうとして、BeautyPrinter クラスがロードできない場合には、一行目の getAnnotation メソッドの呼び出しで、ClassNotFoundException ではなく、TypeNotPresentException がスローされます。

8.1.5 アノテーション型の戻り値型の読み取り

アノテーション型を要素の戻り値型として持つアノテーションも、読み取ることができます。まず、戻り値型として使用される次のアノテーション型を見てください。

```
import java.lang.annotation.*;

/**
 * 人名。このアノテーション型は、プログラム要素に直接注釈を付けるように
 * 設計されているのではなく、他のアノテーション型で要素を定義するために
```

```
     * 設計されている。
     */
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Name {
        String first();
        String last();
    }
```

さらに Name アノテーション型を使用した次のアノテーション型があったとします。

```
    import java.lang.annotation.*;

    /**
     * プログラム要素の著者を示す。
     */
    @Retention(RetentionPolicy.RUNTIME)
    public @interface Author {
        Name value();
    }
```

ここで、@Author アノテーションを使用して、次のように HomeComing クラスを定義したとします。

```
    @Author(@Name(first = "Christie", last = "Golden"))
    public class HomeComing {
        public void read() { }
    }
```

HomeComing クラスに付けられている@Author アノテーションの読み取りは、次のコードで行うことができます。

```
    Author author = HomeComing.class.getAnnotation(Author.class);
    if (author == null)
        System.out.println("No @Author");
    else {
        Name name = author.value();
        System.out.printf("%s %s%n", name.first(), name.last());
    }
```

8.2　アノテーション処理ツール (apt)

ソースコードに書かれたアノテーションを処理するために、apt（*Annotation Processor Tool*）コマンドが提供されています。apt コマンドは、ソースコードを解析して、使用されているアノテーションを検出し、それらのアノテーションを処理するための**アノテーション・プロセッサ**（*annotation processor*）を**アノテーション・プロセッサ・ファクトリー**（*annotation processor factory*）に対して要求します。つまり、アノテーションを処理するために、アノテーション・プロセッサ・ファクトリー

とアノテーション・プロセッサを作成する必要があります。なお、以降では、それぞれ、ファクトリーとプロセッサと呼びます。

apt コマンドは、プロセッサによるアノテーション処理が終了した時点で、そのプロセッサが新たな Java のソースコードを生成していれば、その新たなソースコードに対して、再度、アノテーション処理を行います。この処理は、新たなソースコードの生成が行われなくなるまで、繰り返されます。最後に、最初のソースコードと新たに生成されたソースコードすべてが、実際にコンパイルされます。

8.2.1 ミラー API パッケージ

アノテーションを処理するための API として、以下の 4 つのパッケージが用意されています。

com.sun.mirror.apt パッケージ
: プロセッサとファクトリーが、apt コマンドとやり取りするためのインタフェースが定義されたパッケージです。

com.sun.mirror.declaration パッケージ
: ソースコード上のフィールド、メソッド、クラスなどの宣言をモデル化したインタフェースが定義されたパッケージです。

com.sun.mirror.type パッケージ
: ソースコード上の型をモデル化したインタフェースが定義されたパッケージです。

com.sun.mirror.util パッケージ
: 型や宣言を処理するための各種ユーティリティを定義したパッケージです。

これらのミラー API は、Java 言語のタイプシステムをモデル化するように設計されています。

8.2.2 ファクトリーの作成

アノテーションを処理するためには、com.sun.mirror.apt パッケージに定義されている AnnotationProcessorFactory インタフェースを実装したファクトリーを作成しなければなりません。AnnotationProcessorFactory インタフェースの定義は、以下の通りです。

```
public interface AnnotationProcessorFactory {
    Collection<String> supportedOptions();
    Collection<String> supportedAnnotationTypes();
    AnnotationProcessor getProcessorFor(Set<AnnotationTypeDeclaration> atds,
                                        AnnotationProcessorEnvironment env);
}
```

apt コマンドは、supportedAnnotationTypes メソッドを通して、ファクトリーに対してサポートするアノテーション名を問い合わせます。したがって、supportedAnnotationTypes メソッドの実装では、サポートするアノテーション型の名前のコレクションを返さなければなりません。名前の指定方法としては、次の 3 通りがあります。

- アノテーション型の完全修飾名を指定します。たとえば、"java.lang.annotation.Retention"と指定します。
- パッケージ名を指定します。たとえば、"java.lang.annotation.*"は、指定されたパッケージおよびそのサブパッケージ内で定義されているすべてのアノテーションを指定したことになります。
- すべてのアノテーション型を指定するということで、"*"を指定します。

たとえば、すべてのアノテーション型を処理するファクトリーであることを示すには、次のように supportedAnnotationTypes メソッドを定義します。

```
private static final Collection<String> supportedAnnotations
    = Collections.unmodifiableCollection(Arrays.asList("*"));

public Collection<String> supportedAnnotationTypes() {
    return supportedAnnotations;
}
```

ファクトリーやそれから生成されるプロセッサが受け付けるオプションとして何があるかを、apt コマンドは、supportedOptions メソッドを通して問い合わせます。ファクトリーやプロセッサが定義するコマンドラインのオプションは、必ず、-A で始まっていなければなりませんので、-Adebug や -Aloglevel=3 などのように指定することになります。この場合、supportedOption メソッドとしては、次のコードのように、"-Adebug と"-Aloglevel"を返します。

```
private static final Collection<String> supportedOptions
    = Collections.unmodifiableCollection(
                  Arrays.asList("-Adebug", "-Aloglevel"));

public Collection<String> supportedOptions() {
    return supportedOptions;
}
```

アノテーションを処理するためのプロセッサを、getProcessorFor メソッドを通して、apt コマンドは取得しようとします。そのため、ファクトリーの実装は、getProcessorFor メソッドで、プロセッサを返すようにしなければなりません。引数の atds は、プロセッサを生成すべきアノテーション型のセットが渡されます。env は、アノテーション処理のための環境情報を提供します。

次の AptFactory クラスは、ファクトリーを実装したクラスです。

```
import java.util.Arrays;
import java.util.Collection;
import static java.util.Collections.*;
import java.util.Set;

import com.sun.mirror.apt.AnnotationProcessor;
import com.sun.mirror.apt.AnnotationProcessorEnvironment;
import com.sun.mirror.apt.AnnotationProcessorFactory;
import com.sun.mirror.declaration.AnnotationTypeDeclaration;
```

```
import com.sun.mirror.declaration.TypeDeclaration;

public class AptFactory implements AnnotationProcessorFactory {
    private static final Collection<String> supportedAnnotations
        = unmodifiableCollection(Arrays.asList("*"));
    private static final Set<String> supportedOptions
        = emptySet();

    public Collection<String> supportedAnnotationTypes() {
        return supportedAnnotations;
    }

    public Collection<String> supportedOptions() {
        return supportedOptions;
    }

    public AnnotationProcessor getProcessorFor(
            Set<AnnotationTypeDeclaration> atds,
            AnnotationProcessorEnvironment env) {

        for (AnnotationTypeDeclaration atd: atds)
            System.out.println("Annotation Type = " + atd.getQualifiedName());

        for (TypeDeclaration typeDecl : env.getSpecifiedTypeDeclarations())
            System.out.println("Specified Type = " + typeDecl);

        return null;
    }
}
```

supportedAnnotationTypes メソッドでは、すべてのアノテーション型をサポートすることを示すコレクションを返しています。supportedOptions メソッドでは、オプションを何も定義しないので、空のセットが返されています。getProcessorFor メソッドでは、atds として渡されたアノテーション型の表示と、env で渡された、対象となるアノテーションが使用されている型の一覧を表示しています。

ミラー API は、標準のコアのクラスライブラリには含まれていませんので、JDK に含まれる tools.jar を明示してコンパイルする必要があります。

```
D:\Tiger\example\apt>javac -cp C:\jdk1.5.0\lib\tools.jar AptFactory.java
```

では、ここで、次の Authors アノテーション型を定義した Authors.java ファイルとそれを使用した Peopleware.java ファイルがあるとします。

```
// File: Authors.java
public @interface Authors {
    String[] value();
}
```

```
// File: Peopleware.java
@Authors({"Tom DeMarco", "Timothy Lister"})
public class Peopleware {
    public void read() { }
}
```

この Peopleware.java に対して、AptFactory クラスをファクトリーとして、apt コマンドに指定して実行してみます。

```
D:\Tiger\example\apt>apt -factory AptFactory Peopleware.java
Annotation Type = Authors
Specified Type = Peopleware
警告: ファクトリ AptFactory が注釈プロセッサに null を返しました。
警告 1 個
```

ファクトリーのクラスを-factory オプションで指定しています。Peopleware クラスでは、@Authors アノテーションが使用されているので、その旨が表示されています。また、処理の対象としている型は、Peopleware クラスですので、その旨も表示されています。プロセッサとしては、null を返していますので、警告が表示されています。

8.2.3 プロセッサの作成

アノテーションを読み取るには、実際にプロセッサを生成して返す必要があります。プロセッサは、com.sun.mirror.apt.AnnotationProcessor インタフェースを実装しなければなりません。AnnotationProcessor インタフェースの定義は次の通りです。

```
public interface AnnotationProcessor {
    void process();
}
```

引数もなく、戻り値型も void の process メソッドが定義されているだけですので、process メソッドを実装するには、ファクトリーの getProcessorFor で渡された env への参照を渡す必要があります。

次のコードは、ソースコード内でのクラス宣言、フィールド宣言、メソッド宣言におけるアノテーションを取り出して表示するファクトリーとプロセッサです。

```
import static java.util.Collections.*;
import java.util.*;

import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.type.*;
import com.sun.mirror.util.*;

public class AptFactory2 implements AnnotationProcessorFactory {
    private static final Collection<String> supportedAnnotations
        = unmodifiableCollection(Arrays.asList("*"));
```

```
    private static final Set<String> supportedOptions = emptySet();

    public Collection<String> supportedAnnotationTypes() {
        return supportedAnnotations;
    }

    public Collection<String> supportedOptions() {
        return supportedOptions;
    }

    public AnnotationProcessor getProcessorFor(
            Set<AnnotationTypeDeclaration> atds,
            AnnotationProcessorEnvironment env) {
        return new AptProcessor(env);
    }

    // A Simple Annotation Processor
    private static class AptProcessor implements AnnotationProcessor {
        private final AnnotationProcessorEnvironment env;

        public AptProcessor(AnnotationProcessorEnvironment env) {
            this.env = env;
        }

        public void process() {
            for (TypeDeclaration td: env.getSpecifiedTypeDeclarations()) {
                System.out.printf("Processing %s.java ...%n", td);

                printTypeDeclarationAnnotation(td);

                for (FieldDeclaration fd: td.getFields())
                    printFieldAnnotation(fd);

                for (MethodDeclaration md: td.getMethods())
                    printMethodAnnotation(md);
            }
        }

        private void printTypeDeclarationAnnotation(TypeDeclaration td) {
            System.out.println("class or interface : " + td);
            for (AnnotationMirror am: td.getAnnotationMirrors())
                    printAnnotationMirror(am);
        }

        private void printFieldAnnotation(FieldDeclaration fd) {
            System.out.println("field: " + fd);
            for (AnnotationMirror am: fd.getAnnotationMirrors())
```

```
                printAnnotationMirror(am);
        }

        private void printMethodAnnotation(MethodDeclaration md) {
            System.out.println("method: " + md);
            for (AnnotationMirror am: md.getAnnotationMirrors())
                    printAnnotationMirror(am);
        }

        private void printAnnotationMirror(AnnotationMirror am) {
            System.out.println(" type : " + am.getAnnotationType());
            for (Map.Entry<AnnotationTypeElementDeclaration,AnnotationValue>
                        e: am.getElementValues().entrySet()) {
                AnnotationTypeElementDeclaration name = e.getKey();
                AnnotationValue value = e.getValue();
                System.out.println(" "+ name.getSimpleName() +" : " +value);
            }
        }
    }
}
```

プロセッサの process メソッド内では、クラス宣言、フィールド宣言、メソッド宣言に付けられたアノテーションを取り出して、アノテーションの型名と値を表示しています。AptFactory2 クラスをコンパイルして、Peopleware.java ファイルに対して apt コマンドで実行すると次のようになります。

```
D:\Tiger\example\apt>javac -cp C:\jdk1.5.0\lib\tools.jar AptFactory2.java

D:\Tiger\example\apt>apt -factory AptFactory2 Peopleware.java
Processing Peopleware.java ...
class or interface : Peopleware
 type : Authors
 value : {"Tom DeMarco", "Timothy Lister"}
method: read()
```

Peopleware クラスには@Authors アノテーションが付けられているので、その型名と値が表示されています。read メソッドにはアノテーションが付けられていませんので、メソッド名だけが表示されています。

8.2.4 ファイル出力

アノテーションを解析してファイルを生成するには、ファクトリーの getProcessorFor メソッドの第 2 引数として渡される AnnotationProcessorEnvironment インタフェース型である env を使用します。AnnotationProcessorEnvironment インタフェースは、次のメソッドを定義しています。

Filer getFiler()

com.sun.mirror.apt.Filer インタフェースを実装したインスタンスを返します。

このgetFilerメソッドで返されるインスタンスが実装しているFilerインタフェースは、次のメソッドを定義しています。これらのメソッドは、すべてIOExceptionをスローすると宣言されています。

PrintWriter **createSourceFile(String name)**
: Javaソースコードを生成するためのPrintWriterインスタンスを返します。nameは、そのJavaソースコード内で定義される型の完全修飾名を指定します。

OutputStream **createClassFile(String name)**
: バイトコードを生成するためのOutputStreamインスタンスを返します。nameは、そのバイトコードが生成される型の完全修飾名を指定します。

PrintWriter **createTextFile(Location loc, String pkg, File relPath, String charsetName)**
: テキストファイルを生成するためのPrintWriterインスタンスを返します。出力先を示すLocationはenum型であり、ソースディレクトリを指すSOURCE_TREE、クラスファイルのディレクトリを指すCLASS_TREEが定義されています。pkgは、テキストファイルを出力するパッケージ名を指定します。relPathは、ファイル名を指定します。charsetNameは、文字セットを指定します。特に指定しない場合には、nullを渡します。

OutputStream **createBinaryFile(Location loc, String pkg, File relPath)**
: バイナリファイルを生成するためのOutputStreamインスタンスを返します。引数の意味は、createTextFileメソッドと同じです。

AnnotationProcessorEnvironmentインタフェースは、アノテーション処理中にエラーを発見した場合に、エラーメッセージ等を表示するためのcom.sun.mirror.apt.Messagerインタフェースのインスタンスを返すgetMessagerメソッドを提供しています。Messagerインタフェースは、メッセージ出力用のprintErrorメソッド、printWarningメソッド、printNoticeメソッドを提供しています。

aptコマンドの簡単な使用方法について説明しました。ミラーAPIは、きめ細かにアノテーションを読み取るためのAPIを提供していますが、詳細については、API仕様を参照してください。

8.3 クラスファイルからの読み込み

Retentionアノテーションの値がRetentionPolicy.CLASSのアノテーションは、クラスファイル内にアノテーションが保存されます。しかし、クラスファイルから、リフレクションAPIを使用して保存されているアノテーションを読み取ることはできませんし、aptコマンドを使用して読み取ることもできません。残念ながら、クラスファイル内のアノテーションを読み取るには、クラスファイルを直接解析するしかありません。しかし、クラスファイルのフォーマットは複雑であり、自分で解析するのは容易ではありません。

クラスファイルを解析するためのライブラリーとしては、BCEL（*Bytecode Engineering Library*）

[*2]があります。BCEL を用いたアノテーションの読み取り方法については、本書の範疇外であり詳細は説明しませんが、興味があれば、*Core Java, Seventh Edition, Volume II – Advanced Features*[Horstmann04b] の第 13 章 *Annotations* の *Bytecode Engineering* を参照されると良いでしょう。

[*2] `http://jakarta.apache.org/bcel`

第 9 章

基本パッケージ

本章では、リリース 1.3 に基づいて記述されている『プログラミング言語 Java 第 3 版』に沿って、リリース 1.4 およびリリース 5.0 までの差分を説明していきます。

9.1 トークン、演算子、式

9.1.1 文字セット

リリース 1.4 では、使用される文字セットが Unicode 3.0 [*1]になりましたが、リリース 5.0 からは、Unicode 4.0 になりました。Unicode 標準[*2] は、最初は 16 ビット長固定の文字エンコーディングとして規定されていましたが、多くの文字をサポートするために、16 ビット以上に拡張されています。

Unicode 4.0 では、有効な**コードポイント**（*code point*）は、`U+0000` から `U+10FFFF` までの 21 ビットとされています。`U+`*n* は、Unicode 標準における表記法です。`U+0000` から `U+FFFF` までは、**基本多言語プレーン**（*Basic Multilingual Plane*）と呼ばれます。`U+FFFF` を超える文字セットは、**補助文字**（*supplementary character*）と呼ばれます。

`String` クラスや `StringBuffer` クラスでは、UTF-16 形式で文字を保持します。UTF-16 では、`U+FFFF` を超える補助文字は 2 つの `char` の組で表現され、それぞれ、**上位代理**（*high-surrogate*）（`\uD800-\uDBFF`）範囲と**下位代理**（*low-surrogate*）（`\uDC00-\uDFFF`）範囲に変換されます。また、`UTF-16` 内の各 16 ビットの文字は、**コードユニット**（*code unit*）と呼ばれます。

`char` 型は、16 ビットですので、基本多言語プレーン内の文字か、代理文字しか表現できません。一方、`int` 型は 32 ビットですので、21 ビットのコードポイントを保持できることになります。`Character` クラスなどで、メソッドのパラメータとして文字を受け取る場合に、その型が `int` と宣言されているパラメータはコードポイントを表しますし、パラメータ名も `codePoint` となっています。`Character` クラスには、`int` 型のコードポイントを取り扱うための static メソッド（9.7.4 節）が用意されています。なお、16 ビットを超える補助文字を表現するための文字リテラル表現形式はありません。

[*1] JDK1.1 より前のバージョンで Unicode 1.1.5 をサポートし、JDK1.1 で Unicode 2.0、JDK1.1.7 で Unicode 2.1 をサポート。

[*2] `http://www.unicode.org`

9.1.2 予約語

リリース 1.4 で assert、リリース 5.0 で enum が、新たな予約語（*keyword*）として加えられています。

予約語

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

const と goto は予約はされていますが、従来通り使用されていません。

9.1.3 浮動小数点リテラル

リリース 5.0 からは、10 進数だけでなく、16 進数で浮動少数点リテラルを記述できます。0x または 0X で始まる 16 進数字で表される値と、それに対する 2 のべき乗の指数部分から構成されます。2 のべき乗の指数部分は、p または P で区切り、その後に符号付 10 進数で指定します。最後に、単精度定数を示すために f または F を、倍精度定数を表すために d または D を付けることができます。次のリテラルは、すべて同じ値を表しています。

```
1.25 0x1.4p0 0x.aP1 0x2.8p-1
```

0x1.4p0、0x.aP1、0x2.8p-1 は、それぞれ次の計算により、1.25 であることが分かります。

$$\texttt{0x1.4p0} = (1 \times 16^0 + 4 \times 16^{-1}) \times 2^0 = (1 \times 1 + 4 \times 0.0625) \times 1 = 1.25$$

$$\texttt{0x.aP1} = (10 \times 16^{-1}) \times 2^1 = (10 \times 0.0625) \times 2 = 1.25$$

$$\texttt{0x2.8p-1} = (2 \times 16^0 + 8 \times 16^{-1}) \times 2^{-1} = (2 \times 1 + 8 \times 0.0625) \times 0.5 = 1.25$$

浮動小数点リテラルの型は、0x1.4p0f のように最後に float 定数にするための f または F が付いていれば、float 型です。最後何も付いていないか、d または D が付いていれば double 型です。

9.1.4 条件演算子?:

リリース 1.4 までは、条件演算子?:の第 2 オペランドと第 3 オペランドの式を評価した結果は、代入可能な互換性のある型でなければなりませんでした。つまり、どちらかのオペランドの式を評価した結果は、明示的なキャストなしで、他方のオペランドの型に代入可能でなければなりませんでした。その結果、リリース 1.4 までは、次のようなコードはコンパイルエラーとなります。

```
interface I { }
class X implements I { }
class Y implements I { }

class Tenary {
    public static void main(String[] args) {
        X x = new X();
        Y y = new Y();

        I i = (args.length) == 0 ? x : y; // NG for 1.4, OK for 5.0
        System.out.print("object = " + i);
    }
}
```

X クラスと Y クラスは、相互に代入可能ではありませんので、リリース 1.4 まではコンパイルエラーとなります。リリース 5.0 からは、両方に共通な型が、条件演算子?:の左辺の変数の型に代入可能であればよいことになります。

第 2 オペランドと第 3 オペランドのどちらかが基本データ型で、もう片方が参照型でかつアンボクシングにより互換性のある基本データ型に変換できない場合には、基本データ型のオペランドは、ボクシングにより参照型に変換されます。つまり、どちらも参照型となります。最初から第 2 オペランドと第 3 オペランドの両方が参照型の場合も含めて、それらの 2 つの参照型に共通な型が、条件演算子?:の式の型となります。つまり、Tenary クラスの例では、インタフェース I がその共通な型と見なされます。たとえば、第 2 オペランドの式の型が int 型で、第 3 オペランドの式の型が String 型だとすると、第 2 オペランドの式の値はボクシングにより Integer 型となります。そして、Integer 型と String 型の共通の型は、Object 型ですので、条件演算子?:の式の型は、Object 型となります。

9.2 例外

9.2.1 例外連鎖

例外は、他の例外により引き起こされる場合があります。『プログラミング言語 Java 第 3 版』の 1.13 節「例外」[*3]には、次のサンプルコードが掲載されています。

[*3] 『プログラミング言語第 4 版』の 1.14 節「例外」

```
class BadDataSetException extends Exception {}

class MyUtilities {
    public double [] getDataSet(String setName)
        throws BadDataSetException
    {
        String file = setName + ".dset";
        FileInputStream in = null;
        try {
            in = new FileInputStream(file);
            return readDataSet(in);
        } catch (IOException e) {
            throw new BadDataSetException();
        } finally {
            try {
                if (in != null)
                    in.close();
            } catch (IOException e) {
                ; // 無視：データの読み込みは成功している
            }
        }
    }

    // ... readDataSet の定義 ...
}
```

このように、ある API の実装において、低レベルの例外を、その API が提供する抽象概念の例外へ変換することは**例外翻訳**（*exception translation*）[*4]と呼ばれます。

この getDataSet メソッドの問題点は、BadDataSetException 例外の原因となった IOException 例外に関する情報が失われることです。このためリリース 1.4 からは、Throwable クラスに例外の**原因**（*cause*）を保持する**例外連鎖**（*exception chaining*）のための拡張が行われ、以下のコンストラクタが追加されています。

`public` **Throwable(String message, Throwable cause)**
: 指定された詳細メッセージと原因（cause）を持つ例外を生成します。ただし、原因は、自動的に詳細メッセージに含まれたりしません。

`public` **Throwable(Throwable cause)**
: 指定された原因（cause）を持つ例外を生成します。詳細メッセージとしては、cause.toString() 呼び出しの結果を保持します。

すべての例外は、Throwable クラスを継承した Error クラスか Exception クラスのサブクラスですので、Error クラスと Exception クラスにも同様にコンストラクタが追加されています。

[*4] 『Effective Java』[Bloch01]、項目 43「抽象概念に適した例外をスローする」（第 2 版は、項目 61）。

例外連鎖のために、次のメソッドが Throwable クラスに追加されています。

public Throwable **getCause()**
: この例外がスローされる原因となった例外を返します。原因となる例外が存在しない場合には、null が返されます。

public Throwable **initCause(Throwable cause)**
: この例外の原因として、指定された cause を設定します。このメソッドは、たかだか一回しか呼び出すことができません。Throwable を cause として受け取るコンストラクタを使用してインスタンスが生成された場合には、このメソッドは、一回も呼び出すことができません。すでに、原因が設定されている状態でこのメソッドを呼び出すと IllegalStateException がスローされます。

initCause メソッドにより、既存の例外に対して、その原因を設定できます。前節の例では、次のように書き直すことで、IOException を原因として設定してからスローすることができます。

```
BadDataSetException bdse = new BadDataSetException();
bdse.initCause(e);
throw bdse;
```

もちろん、BadDataSetException クラスを書き直して、原因（cause）を受け取るコンストラクタを追加することで対処することも可能です。しかし、常にすべての既存の例外クラスを書き直すことができるとは限りません。たとえば、サードパーティのクラスライブラリでソースコードが無い場合には、書き直せません。その場合でも、すべての例外クラスは Throwable クラスを継承していますので、initCause メソッドで原因となる例外を設定することが可能です。

9.2.2 スタックトレース

Throwable クラスを継承した例外クラスのインスタンスが生成された時点で、そのインスタンスはスタックトレースを保持します。リリース 1.3 までは、printStackTrace メソッドで出力するしかできませんでした。リリース 1.4 からは、スタックトレースを取り出すための getStackTrace メソッドが Throwable クラスに追加されています。

public StackTraceElement[] **getStackTrace()**
: スタックトレースを、StackTraceElement の配列として返します。インデックスが 0 の要素が、この例外を生成した時点のスタックのトップであり、例外を発生させたメソッドを通常は表しています。

スタックトレースが StackTraceElement の配列として返されますので、その配列の要素を調べることで、スタックトレースの内容を知ることができます。調べるためのメソッドとして、StackTraceElement クラスは、getFileName メソッド、getLineNumber メソッド、getClassName メソッド、getMethodName メソッド、isNativeMethod メソッドを提供しています。

次のコードは、Throwable クラスのインスタンスを直接生成して、スタックトレースを表示するサンプルコードです。

```
class StackTraceTest {
    public static void main(String[] args) {

        StackTraceElement[] ste = createThrowable().getStackTrace();

        for (StackTraceElement e: ste) {
            System.out.printf("%s: line %2d: %s#%s%n",
                e.getFileName(), e.getLineNumber(),
                e.getClassName(), e.getMethodName());
        }
    }

    private static Throwable createThrowable() {
        return new Throwable();
    }
}
```

コンパイルして実行すると次の結果となります。

```
StackTraceTest.java: line 14: StackTraceTest#createThrowable
StackTraceTest.java: line  4: StackTraceTest#main
```

main メソッドから呼ばれた createThrowable メソッド内で作成された例外であることを示しています。

9.3 アサーション

防御的プログラミング（*defensive programming*）を行うことは、ソフトウェア開発では重要なことです。たとえば、メソッドのパラメータが正しいか否かを判断して、不正ならば適切な例外をスローします。一方で、メソッドの処理を開始した時点で、メソッドが呼ばれているオブジェクトの状態が正しい（事前条件）とか、メソッドの処理が完了した時点でもオブジェクトの状態が正しい（事後条件）かを検査するには、リリース 1.4 で導入された assert を使用します。

9.3.1 シンタックス

assert 文のシンタックスは、次の通りです。

```
assert eval-expr [: detail-expr];
```

eval-expr はブール式であり、式を評価した結果は、boolean 型か Boolean 型でなければなりません。*detail-expr* は、オプションであり、その型が、void（すなわち、戻り値型が void であるメソッド呼び出し）の場合には、コンパイルエラーとなります。

デフォルトでは、assert 文は評価されません。つまり、ソースコード上に、その assert 文が書かれていないかのように実行されます。注意しなければならないのは、assert 文が評価されないだけで

あり、その assert 文自身はコンパイルされたバイトコードには含まれます。これは、C 言語/C++言語で使用されている関数マクロである assert と異なる点です。assert 文の評価は、仮想マシン内のすべてのクラスに対して有効にすることもできますし、パッケージ単位、クラス単位、あるいはクラスローダ単位に有効にすることもできます。

assert 文が有効になっている場合には、*eval-expr* が評価されて、その値が true であれば、assert 文は終了します。*eval-expr* の評価結果が false であり、オプションの *detail-expr* が指定されていなければ、AssertionError がスローされます。*eval-expr* の評価結果が false であり、オプションである *detail-expr* が指定されている場合には、その式が評価されます。評価された結果が基本データ型であれば、そのまま文字列に変換されます。オブジェクト参照であれば、toString メソッドを呼び出して文字列に変換されます。そして、その文字列が、詳細メッセージとして AssertionError のコンストラクタに渡されて、AssertionError がスローされます。さらに、その評価した結果が Throwable クラスのサブクラスであれば、AssertionError の原因として設定され、AssertionError の getCause メソッドで返される値となります。

9.3.2 コマンドラインからの制御

assert 文の評価は、デフォルトで無効になっています。java コマンドを使用して仮想マシンを起動する場合には、標準のコマンドラインオプションを使用して、有効にすることができます。

-enableassertions/-ea[*descriptor*]
: *descriptor* で定義されている assert 文の評価を有効にします。*descriptor* が指定されていなければ、システムクラスローダでロードされたクラス以外のすべてのクラスに対して assert 文が有効になります。

-disableassertions/-da[*descriptor*]
: *descriptor* で定義されている assert 文の評価を無効にします。*descriptor* が指定されていなければ、すべてのクラスに対して assert 文が無効になります。

オプションである *descriptor* が指定されていなければ、すべてのシステムクラスではないクラス（すなわち、システムクラスローダ以外のクラスローダによりロードされたクラス）に対して適用されます。オプションである *descriptor* を明示した場合には、システムクラスも指定することができます。

descriptor の指定形式としては、次の 3 通りがあります。

packageName...
: *packageName* で指定されたパッケージ内とそのサブパッケージ内のすべてのクラスで assert 文を有効、あるいは、無効にすることを指定します。

...
: 現在のワーキングディレクトリ下にある無名パッケージ内のすべてのクラスで assert 文を有効、あるいは、無効にすることを指定します。

className
: *className* で指定されたクラス内で assert 文を有効、あるいは、無効にすることを指定し

ます。*className* としては、クラスの完全修飾名を指定します。

実際の指定例を次に示します。

`-ea:com.starship...`
: com.starship パッケージ内とそのすべてのサブパッケージ内のすべてのクラスで、assert 文を有効にします。

`-ea:com.starship.Enterprise`
: com.starship.Enterprise クラスに対する assert 文を有効にします。

`-ea:com.starship... -da:com.starship.Voyager`
: com.starship パッケージ内とそのすべてのサブパッケージ内のすべてのクラスで、assert 文を有効にしますが、com.starship.Voyager クラスだけは無効にします。

`-da:com.starship... -ea:com.starship.Voyager`
: com.starship パッケージ内とそのすべてのサブパッケージ内のすべてのクラスで、assert 文を無効にしますが、com.starship.Voyager クラスだけは有効にします。

このように、複数のオプションは、コマンドラインに書かれた順序で評価されます。

descriptor を指定することで、システムクラスに対しても assert 文を個別に有効にしたり、無効にしたりすることができますが、システムクラス全体で assert 文を有効にしたり、無効にしたりするには、システムクラス用のオプション-enablesystemassertions/-esa と -disablesystemassertions/-dsa を使用します。

9.3.3 プログラミングによる制御

次に示す ClassLoader クラスのメソッドを使用して、実行しているコードから assert 文の評価を制御することができます。

`public void` **setDefaultAssertionStatus(boolean enabled)**
: このローダで、将来ロードされて初期化される、すべてのクラスに対してデフォルトのアサーションステータスを設定します。クラスローダに対するデフォルトのアサーションステータスの初期値は、false です。

`public void` **setPackageAssertionStatus(String packageName, boolean enable)**
: このローダで、将来ロードされて初期化される、指定されたパッケージ内とそのサブパッケージ内の、すべてのクラスとそのすべてのネストクラスに対するデフォルトのアサーションステータスを設定します。パッケージ名として null が渡されると、カレントの無名パッケージを意味します。

`public void` **setClassAssertionStatus(String className, boolean enable)**
: このローダでロードされる際に、指定されたトップレベルのクラスとそのすべてのネストしたクラスに対するデフォルトのアサーションステータスを設定します。このメソッドは、まだ初期化されていないクラスに対して影響します。つまり、一旦、クラスが初期化された場合には、そのアサーションステータスを変更することはできません。

public void **clearAssertionStatus()**

このクラスローダに対するデフォルトのアサーションステータスを false にし、パッケージあるいはクラスに関連付けられていた設定をすべて破棄します。

どのメソッドで指定した設定でも、これからロードされて初期化されるクラスに対してだけ適用されます。一旦ロードされたクラスに対するアサーションステータスを変更することはできません。

9.4 文字列

リリース 5.0 からは、java.lang.StringBuilder クラスが導入されています。StringBuilder クラスは、StringBuffer クラスと同じメソッドを持ちますが、StringBuffer クラスがスレッドセーフであるのに対して、StringBuilder クラスはスレッドセーフではありません。

リリース 1.4 までのコンパイラは、文字列の結合を行う場合に、StringBuffer を使用するコードを生成していました。たとえば、次の式をコンパイラがどのようにコンパイルするかについて説明します。

```
String fullName(String firstName, String lastName) {
    return firstName + ' ' + lastName;
}
```

リリース 1.4 までは、この return 文の式に対して、コンパイラは次のような変換を行っています。

```
new StringBuffer(firstName).append(' ').append(lastName).toString();
```

一方、リリース 5.0 では、次のような変換を行います。

```
new StringBuilder(firstName).append(' ').append(lastName).toString();
```

このようにリリース 5.0 では、文字列結合の効率を上げるために、StringBuilder クラスが使用されます。その結果、リリース 5.0 で導入された新たな言語仕様を一切使用していないプログラムであっても、-target 1.4 のコンパイルオプションを指定しない限り、リリース 5.0 以降のコンパイラでコンパイルされると、リリース 1.4 までの動作環境では動作しないことになります。

既存のコードで、StringBuffer への操作そのものがスレッドセーフであることを必要としている箇所はほとんど無いでしょうから、該当箇所を単純に StringBuilder に置換することが可能です。

9.4.1 CharSequence インタフェース

リリース 1.4 からは、char 型のシーケンス（ひと続き）であることを表すための CharSequence インタフェースが導入されています。定義は、次のようになっています。

```
package java.lang;

public interface CharSequence {
    int length();
```

```
    char charAt(int index);
    CharSequence subSequence(int start, int end);
    public String toString();
}
```

length メソッドは、char 型の文字が何文字あるかを返します。コードポイントでないことに注意してください。charAt メソッドは、インデックスで指定された文字を返します。subSequence メソッドは、インデックスが start から、end - 1 までのサブシーケンスを返します。

CharSequence インタフェースを実装しているのは、String クラス、StringBuffer クラス、java.nio.CharBuffer クラス、StringBuilder クラスです。

9.4.2 Appendable インタフェース

リリース 5.0 からは、char もしくは、CharSequence で表される文字列を追加可能であることを示す Appendable インタフェースが追加されています。Appendable インタフェースは、Formatter クラス（9.10.3 節）の出力を受け付けるクラスが実装しなければならなく、その定義は次のようになっています。

```
package java.lang;

import java.io.IOException;

public interface Appendable {
    Appendable append(CharSequence csq) throws IOException;
    Appendable append(CharSequence csq, int start, int end)
                                                throws IOException;
    Appendable append(char c) throws IOException;
}
```

2 つ目の append メソッドは、インデックスが start から end - 1 までの文字を追加します。

Appendable インタフェースを実装しているのは、すべてのライタークラス（BufferedWriter、CharArrayWriter、FileWriter、FilterWriter、OutputStreamWriter、PipedWriter、PrintWriter、StringWriter、Writer）、PrintStream クラス、StringBuffer クラス、StringBuilder クラスです。

9.4.3 Readable インタフェース

リリース 5.0 からは、java.nio.CharBuffer クラスへ文字の読み込みが可能である Readable インタフェースが追加されています。その定義は、次の通りです。

```
package java.lang;

import java.io.IOException;
```

```
public interface Readable {
    public int read(java.nio.CharBuffer cb) throws IOException;
}
```

Readable インタフェースは、すべてのリーダークラス（BufferedReader、CharArrayReader、FileReader、FilterReader、InputStreamReader、LineNumberReader、PipedReader、PushbackReader、Reader、StringReader）、CharBuffer クラスが実装しています。

9.4.4 String クラス

String クラスには、新たに次のコンストラクタが追加されています。

public **String(int[] codePoints, int offset, int count)**
: 配列引数 codePoints のインデックス offset から、count 数分のコードポイントを char へ変換した内容を持つ String クラスのインスタンスを構築します。offset + count が codePoints の配列の長さよりも大きい場合には、IndexOutOfBoundsException がスローされます。

public **String(StringBuilder builder)**
: 引数 builder に含まれる文字列を内容として持つ String クラスのインスタンスを構築します。

新たなメソッドとしては、次のインスタンスメソッドが追加されています。

public int **codePointAt(int index)**
: index で指定されたインデックス位置の文字（Unicode のコードポイント）を返します。index は、0 から length() - 1 の範囲でなければなりません。index で指定された位置の char が上位代理範囲内であり、index + 1 が length() - 1 以下であり、次の位置にある char が下位代理範囲内であれば、それらの 2 つの char で表される補助文字のコードポイントを返します。そうでなければ、index で指定された位置の char だけが返されます。

public int **codePointBefore(int index)**
: index で指定されたインデックス位置の前の文字（Unicode のコードポイント）を返します。index は、1 から length() の範囲でなければなりません。codePointAt(index - 1) と同じです。

public int **codePointCount(int beginIndex, int endIndex)**
: この String 内の beginIndex から endIndex - 1 の範囲にあるコードポイントの数を返します。上位代理範囲と下位代理範囲の対となっていない代理範囲の char は、1 個のコードポイントとして数えられます。

public int **offsetByCodePoints(int index, int codePointOffset)**
: index で指定された位置から codePointOffset 数のコードポイントを表す char の次の位置が、インデックスとして返されます。上位代理範囲と下位代理範囲の対となっていない代理範囲の char は、1 個のコードポイントとして数えられます。

public boolean **contentEquals(StringBuffer sb)**
: sb 内の文字列と等しいか否かを返します。sb の内容を一度 String へ変換することなく、直接内

部バッファを比較します。したがって、equals(sb.toString()) で比較するよりも速いです。

public boolean **contentEquals(CharSequence cs)**

cs で表される文字シーケンスと等しいかを返します。contentEquals メソッド呼び出しで、String クラスあるいは StringBuilder クラスのインスタンスが渡されたコードが書かれた場合には、このメソッドが呼び出されます。String クラス、StringBuffer クラス、StringBuilder クラスは、すべて CharSequence インタフェースを実装していることに注意してください。

public CharSequence **subSequence(int beginIndex, int endIndex)**

beginIndex から endIndex - 1 までのインデックス位置の文字から構成される CharSequence を返します。これは、次の呼び出しと同じような振る舞いとなります。

```
subString(beginIndex, endIndex)
```

public boolean **matches(String regex)**

この文字列が、regex で指定された正規表現（9.12.5 参照）と一致するかどうかを返します。一致した場合には true が返され、一致しなければ false が返されます。matches メソッドが呼び出された文字列を str とすると、次の呼び出しと同じです。

```
Pattern.matches(regex, str)
```

public boolean **contains(CharSequence s)**

この文字列が、s で指定された文字シーケンスを含むかどうかを返します。含んでいれば true が返され、含まれていなければ false が返されます。

public String **replaceFirst(String regex, String replacement)**

regex で指定された正規表現に最初に一致する部分を、replacement で置換した文字列を返します。matches メソッドが呼び出された文字列を str とすると、次の呼び出しと同じです。

```
Pattern.compile(regex).matcher(str).replaceFirst(replacement)
```

public String **replaceAll(String regex, String replacement)**

regex で指定された正規表現に一致する部分をすべて、replacement で置換した文字列を返します。matches メソッドが呼び出された文字列を str とすると、次の呼び出しと同じです。

```
Pattern.compile(regex).matcher(str).replaceAll(replacement)
```

public String **replace(CharSequence target, CharSequence replacement)**

target で指定された文字シーケンスと一致する部分をすべて、replacement で指定された文字シーケンスで置換した文字列を返します。

public String[] **split(String regex, int limit)**

この文字列を regex で指定された正規表現と一致した文字列で区切った文字列の配列を返します。正規表現と一致した部分は、取り除かれます。limit は、指定された正規表現が、この文字列に何回適用されるかを指定します。limit が 0 であれば、可能な限り何度も適用します。limit が正の値であれば、limit - 1 回適用し、残りの文字列は、返される配列の最後の要素となります。limit が負の値であれば、その絶対値の回数分適用し、残りの文字列は返されません。たとえば、文字列"x--y--z--v"に split("--",0) が呼び出されると、次のように配列の要素が返されます。

```
0: x
1: y
2: z
3: v
```

split メソッドが呼び出された文字列を str とすると、次の呼び出しと同じです。

```
Pattern.complie(regex).split(str,limit)
```

public String[] **split(String regex)**

split(regex,0) と同じです。

正規表現を引数として受け取るメソッドは便利ですが、その都度正規表現をコンパイルします。頻繁に使用される正規表現であれば、一度だけコンパイルしてから、メソッドを呼び出す方が効率的です。

既存のメソッドで、文字として int 型の引数を受け取る indexOf メソッドと lastIndexOf メソッドでは、int 型として渡された文字 ch が 0x0000 から 0xFFFF の範囲であれば、charAt(*k*) == ch となるインデックス *k* が返されますし、0xFFFF より大きければ、Unicode のコードポイントとして扱われ、codePointAt(*k*) == ch となるインデックス *k* が返されます。

新たな static メソッドとしては、以下のメソッドが追加されています。

public static String **format(String format, Object ... args)**

format で指定された形式（9.10.3 節参照）で、引数をフォーマットした結果の文字列を返します。

public static String **format(Locale locale, String format, Object ... args)**

local で指定されたロケールで、format に従って引数をフォーマットした結果の文字列を返します。

9.4.5 StringBuffer クラスと StringBuilder クラス

リリース 5.0 では、スレッドセーフな設計がされている StringBuffer クラスに対して、スレッドセーフではない StringBuilder クラスが導入されていますが、どちらも AbstractStringBuilder クラスを継承しています。StringBuilder クラスは、StringBuffer クラスと同じパラメータを取るコンストラクタと、同じシグニチャのメソッドを提供していますので、ここでは、StringBuffer クラスの変更点について説明します。

新たに以下のコンストラクタが追加されています。

public **StringBuffer(CharSequence seq)**

seq で指定された文字シーケンスを含むインスタンスを生成します。内部バッファの容量の初期値は、seq が含む文字数に 16 を足した大きさとなります。CharSequence インタフェースは、String クラス、StringBuffer クラス、StringBuilder クラスが実装していますので、それらのクラスのインスタンスの内容をコピーすることができます。

新たなメソッドとしては、String クラスに追加されたコードポイント関連と同じメソッド（codePointAt、codePointBefore、codePointCount、offsetByCodePoints、subSequence）、リ

リース 1.4 で String クラスと同じになるように追加されたメソッド（indexOf、lastIndexOf）に加えて以下のメソッドが追加されています。

public void **trimToSize()**
　内部で保持しているバッファの大きさを、実際に含まれている文字を保持するだけに必要な大きさに小さくします。
public StringBuffer **append(StringBuffer sb)**
　sb で指定された文字シーケンスを追加します。
public StringBuffer **append(CharSequence s)**
　s で指定された文字シーケンスを追加します。
public StringBuffer **append(CharSequence s, int start, int end)**
　s で指定された文字シーケンスのインデックス位置が start から end - 1 までの文字を追加します。
public StringBuffer **appendCodePoint(int codePoint)**
　指定されたコードポイントを追加します。
public StringBuffer **insert(int dstOffset, CharSequence s)**
　dstOffset で指定されるオフセット位置に、s で指定された文字シーケンスを挿入します。
public StringBuffer **insert(int dstOffset, CharSequence s, int start, int end)**
　dstOffset で指定されるオフセット位置に、s で指定された文字シーケンスのインデックス位置が start から end - 1 までの文字を挿入します。

9.4.6 Charset クラスと文字エンコーディング

文字エンコーディングは、通常は、"US-ASCII"などの文字列で表現されます。リリース 1.4 からは、文字セットと文字エンコーディングを表すクラスが、java.nio.charset パッケージで提供されています。文字セットを表す Charset クラスがあり、文字エンコーディング/デコーディングを行うための CharsetEncoder クラスと CharsetDecoder クラスがあります。

Charset クラスのインスタンスの生成は、以下の static ファクトリーメソッドを使用して行います。

public static Charset **forName(String charsetName)**
　指定された文字セット名を表す Charset クラスのインスタンスを返します。

動作している Java 仮想マシンでのロケールと OS に依存したデフォルトの文字セットを返す defaultCharset メソッド、すべての利用可能な文字セットを返す availableCharsets メソッドもあります。また、文字セットをエンコード/デコードするための CharsetEncoder クラスおよび CharsetDecoder クラスのインスタンスを返す newEncoder メソッドと newDecoder メソッドもあります。

java.io パッケージの InputStreamReader クラスには、Charset クラス、あるいは、CharsetDecoder クラスのインスタンスを引数として受け取るコンストラクタが追加されてい

ます。OutputStreamWriter クラスには、Charset クラス、あるいは、CharsetEncoder クラスのインスタンスを引数として受け取るコンストラクタが追加されています。

9.5 スレッド

9.5.1 Thread クラスと ThreadGroup クラス

Thread クラスには、リリース 1.4 で新たに次のコンストラクターが追加されています。

public **Thread(ThreadGroup group, Runnable target, String name, long stackSize)**
: 指定された ThreadGroup に属し、指定された名前を持ち、指定されたスタックサイズで、指定された target の run メソッドを使用する新たな Thread を生成します。

他のコンストラクタと異なり、スレッドのスタックサイズが指定できるようになっています。しかし、指定できるスタックサイズは、あくまでも Java 仮想マシン（JVM）に対するヒント程度のものであり、JVM が実際にどれだけの大きさのスタックを割り当てるかを、保証するものではありません。したがって、同じ値が指定されたとしても、JVM の実装や動作する OS ごとに割り当てるスタックサイズが異なったりします。

リリース 5.0 では、スレッドのスタック情報を取得するために、次のメソッドが追加されています。

public StackTraceElement[] **getStackTrace()**
: このスレッドのスタックトレースを返します。スレッドが開始されていなかったり、すでに終了している場合には、長さが 0 の配列が返されます。

また、すべてのスレッドのスタックトレースを取得するための static メソッドとして、getAllStackTraces メソッドも追加されています。

リリース 5.0 からは、スレッドが例外をキャッチしなかった場合に、どのような処理を行うかを実装したハンドラーを設定できるようになっています。

public UncaughtExceptionHandler **getUncaughtExceptionHandler()**
: キャッチされなかった例外により、スレッドが終了した場合に呼び出されるハンドラーを返します。ハンドラーが明示的に設定されていなければ、このスレッドが属する ThreadGroup クラスのインスタンスが返されます。もし、ハンドラーが明示的に設定されていなくて、スレッドがすでに終了している場合には、null が返されます。

public void **setUncaughtExceptionHandler(UncaughtExceptionHandler eh)**
: キャッチされなかった例外により、スレッドが終了した場合に呼び出されるハンドラーを設定します。

処理を行うハンドラーが実装すべき UncaughtExceptionHandler インタフェースは、Thread クラス内に定義されているインタフェースであり、次のように定義されています。

```
public interface UncaughtExceptionHandler {
    void uncaughtException(Thread t, Throwable e);
}
```

ThreadGroup クラスは、このインタフェースを実装するように、変更されています。スレッドごとにハンドラーを登録しなかった場合には、従来通り ThreadGroup クラスの uncaughtException メソッドが呼び出されます。特定のスレッドではなく、システム全体のスレッドに対して、ハンドラーを 1 つ登録することも可能です。そのようなハンドラーの設定と問い合わせは、ThreadGroup クラスの static メソッドである setDefaultUncaughtExceptionHandler メソッドと getDefaultUncaughtExceptionHandler メソッドで行います。まとめると、リリース 5.0 からは、キャッチされなかった例外が発生し、スレッドが終了した場合には、次の順序で処理が行われます。

- 終了したスレッドに、(setUncaughtExceptionHandler メソッドにより) 事前にハンドラーが設定されていれば、そのハンドラーの uncaughtException メソッドを呼び出して処理を終了します。
- ThreadGroup クラスの setDefaultUncaughtExceptionHandler メソッドによりグローバルなハンドラーが設定されていれば、そのハンドラーの uncaughtException メソッドを呼び出して処理を終了します。
- どちらのハンドラーも設定されていなくて、キャッチされなかった例外が ThreadDeath でなければ、その例外のスタックトレースを表示して処理を終了します。

例外がスローされスタックトレースが表示されたとしても、その表示が必ずしも、ThreadGroup クラスの uncaughtException メソッドによって行われているとは限らないことに注意してください。たとえば、AWT や Swing を使用して、GUI を表示している場合に、ボタンを押した場合の処理中に例外がスローされると、AWT のイベントディスパッチスレッドがその例外をキャッチして、スタックトレースを表示します。この場合、スレッドが例外により終了したのではありませんので、ThreadGroup クラスの setDefaultUncaughtExceptionHandler メソッドによりグローバルなハンドラーを設定していても、そのハンドラーが呼び出されることはありません。

Thread クラスに追加された他のメソッドは、次の通りです。

`public long` **getId()**
: このスレッドの識別子番号を返します。識別子 ID は、Thread クラスのインスタンスが生成された時点で割り当てられます。

`public State` **getState()**
: スレッドの状態を返します。State は、enum 型であり、定数値としては、NEW (インスタンスは生成されたが、まだ開始していない)、RUNNABLE (スレッドが実行されている)、BLOCKED (スレッドがモニターロック獲得待ちとなっている)、WAITING (スレッドが他のスレッドからのアクション待ちとなっている)、TIMED_WAITING (スレッドが他のスレッドからのアクションを時間指定で待っている)、TERMINATED (スレッドの実行が終了している) のどれかの値です。

`public static native boolean` **holdsLock(Object obj)**
: カレントスレッドが、指定されたオブジェクトに対してモニターロックを獲得していれば true を

返します。獲得していなければ、false を返します。

holdsLock メソッドは、カレントスレッドが特定のロックを保持しているか検査することを、プログラムから可能にするために追加されています。デッドロックを回避するための「リソースの順序付け」をプログラム的に検査するために使用することができます。リリース 1.3 までは、リソースの順序付けについては、ソースコード上でコメントとして、「ここではすでに *XXX* のロックを獲得しておくこと」と書くことで注意することしかできませんでした。リリース 1.4 以降では、holdsLock メソッドにより、assert を使用して、次のように事前条件を明示的に検査するためのプログラミングが可能となっています。

```
assert Thread.holdsLock(obj);
```

9.5.2 ThreadLocal クラス

ThreadLocal クラスは、ジェネリッククラスとしてパラメータ化されています。さらに、リリース 5.0 からは、remove メソッドが追加されています。

public void **remove()**

このスレッドローカル変数から、カレントスレッドに対して保持している値を取り除きます。取り除かれた後に、set メソッドで何も値が設定されていない状態で、get メソッドにより取り出されると、initialValue メソッドで初期化された値が返されます。

9.6 メモリモデル

あるスレッドが設定したフィールドの値が、他のスレッドから取得できることを正しく保証するためには同期が必要です。同期なしでは、設定された値が正しく取得できる保証はありません。たとえば、『Effective Java』[Bloch01] の項目 48 には、次のサンプルコードが示されています。

```
// 不完全 - 同期が必要!
private static int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

1 つのスレッドが繰り返し generateSerialNumber メソッドを呼び出して、ゼロから n までのシリアル番号列を得た後に、他のスレッドが generateSerialNumber メソッドを呼び出してゼロのシリアル番号を得る可能性もあります。同期なしでは、2 つ目のスレッドは最初のスレッドによる変更を**何も**取得できないかもしれません。この問題を解決するには、generateSerialNumber メソッドを synchronized 宣言して同期することです。同期されたメソッド内で設定され、返されるフィールドの値は、すべてのスレッドが取得できることが保証されます。

volatile 宣言されたフィールドでは、その値を読み出すと最後に設定された値が読み出されることが保証されています。しかし、オブジェクト参照自身が volatile 宣言されていても、その参照を通して、同期なしで、オブジェクトのフィールドの最新の値が読み出されることは保証されません。

リリース 5.0 では、メモリモデルの変更が行われており、その詳細は高度な事柄ですが、本章では、基本的な事柄だけを説明します。

9.6.1 同期アクション

あるスレッドによるフィールドの変更内容が、正しく他のスレッドからも取得できるようにするには、リリース 1.4 まではモニタのロックの獲得と解放しかありませんでした。すなわち、モニタのロックを獲得した場合には、キャッシュを破棄して、スレッドは最新のフィールドの値をメインメモリから読み出すことは保証されます。モニタのロックを解放する場合には、キャッシュの内容がメインメモリへ書き出されることが保証されます。また、volatile 宣言されたフィールドの読み出しは、常にその前に書かれた最新の値が読み出されることが保証されていました。

リリース 5.0 からは、モニタのロック獲得/解放以外に、**同期アクション**（*synchronization action*）として以下のものがあります。

- volatile 宣言されたフィールドへの書き込みは、そのフィールドの読み出しと同期します。
- スレッドを開始する処理は、スレッドが行う最初の処理[*5]と同期します。
- スレッドが終了する前の最後の処理[*6]は、そのスレッドの終了を検出する処理と同期します。
- スレッドへの割り込みは、そのスレッドが割り込まれたことを検出する処理と同期します。
- オブジェクト生成時のフィールドへのデフォルト値の書き込みは、スレッドが行う最初の処理と同期します。

あるスレッドが、volatile 宣言されている x フィールドへの書き込み以前に、volatile 宣言されて**いない** y フィールドへの書き込みを行ったとします。この場合、他のスレッドが y フィールドを単純に読み出すと、最後に書き込まれた値が見えることは保証されません。しかし、volatile 宣言されている x フィールドを先に読み出すことで、その後に y フィールドを読み出すと、最後に書き込まれた値を取得できることが保証されます。

スレッドを生成して実行する場合には、start メソッドによるスレッドの開始とそのスレッドでの処理の開始で同期処理が行われることが保証されますので、新たに生成されたスレッドから、生成前に設定されていたオブジェクトのフィールドの値が正しく取得できることが保証されます。また、スレッドが終了した場合には、そのスレッドの終了を join メソッドあるいは isAlive メソッドで検出した場合も同期処理が行われることが保証されます。次のコードを見てください。

[*5] ソースコード上の run メソッド内の最初の処理を指しているのではなく、それ以前に、JVM 内でスレッドが生成された直後に行われるソースコード上にはない処理を指しています。

[*6] ソースコード上の run メソッド内の最後の処理を指しているのではなく、その後で、JVM 内でスレッド終了する直前に行われるソースコード上にはない処理を指しています。

```
class ThreadSynchronization extends Thread {
    private int x;

    public ThreadSynchronization(int x) { this.x = x; }

    public void run() {
        System.out.println("run: x = " + x);
        x = 2;
    }

    public int getX() { return x; }

public static void main(String[] args) throws InterruptedException  {
        ThreadSynchronization ts = new ThreadSynchronization(1);
        ts.start();
        ts.join();
        System.out.println("main: x = " + ts.getX());
    }
}
```

この ThreadSynchronization クラスの x フィールドは、volatile とは宣言されていないことに注意してください。main メソッド内で、ThreadSynchronization クラスのインスタンスを生成した際に、x フィールドには 1 が設定されます。その設定は、main メソッドを呼び出している main スレッドによって行われています。次に、start メソッドにより新たなスレッドを生成して開始します。run メソッドの先頭で、x フィールドの値を表示します。スレッドの生成とスレッドの処理の開始で同期処理が行われますので、x フィールドの値として 1 が必ず表示されます。次に、run メソッド内で、x フィールドに 2 を設定して、処理を終了しています。main メソッドでは、スレッドの終了を join メソッドで待っていますので、そこで同期処理が行われることが保証されます。その結果、getX メソッドは、必ず 2 を返します。

あるスレッド T へ interrupt メソッドにより割り込みを行った場合には、InterruptedException がスローされたり、Thread クラスの interrupted メソッドあるいは isInterrupted メソッドで割り込みを検出すると同期処理が行われることが保証されます。また、スレッド T 以外のスレッドが、スレッド T への割り込みを isInterrupted メソッドで検出しても同期処理が行われることが保証されます。たとえば、次のコードを見てください。

```
class InterruptSynchronization extends Thread {
    private int x;

    public void run() {
        while (true) {
            try {
                Thread.sleep(3600*1000);
            } catch (InterruptedException e) {
                System.out.println("run: x = " + x);
```

```
                }
            }
        }

        public void setX(int x) {
            this.x = x;
            interrupt();
        }

        public static void main(String[] args) throws Exception {
            InterruptSynchronization is = new InterruptSynchronization();
            is.start();
            for (int i = 0; i < 10; i++) {
                is.setX(i);
                Thread.sleep(1000);
            }
        }
    }
```

InterruptSynchronization クラスの x フィールドは、volatile 宣言されていません。run メソッドと setX メソッドでは、明示的に synchronized 文などで同期を取ることなく x フィールドにアクセスしています。main スレッドが setX メソッドを通して x フィールドを更新して、もう 1 つのスレッドに interrupt メソッドにより割り込みを行っています。その結果、sleep メソッドで眠っているスレッドでは、InterruptedException がスローされます。そして、その例外をキャッチしてから、x フィールドの値を表示しています。この場合、setX メソッドで設定された値が、run メソッドを実行しているスレッドからも正しく取得できることが保証されますので、最後に設定された値が表示されます。

オブジェクトが生成された場合には、通常ソースコード上で初期化子による初期化より以前に、概念的にはすべてのフィールドにデフォルト値が書き込まれます。このデフォルト値の書き込みは、すべてのスレッドから取得できることが保証されます。スレッドの実行が開始された時の最初の処理で同期処理が保証されますので、そのスレッドは、オブジェクトのフィールドの値に関しては、デフォルトの値か、他のスレッドなどによりオブジェクト生成後に設定された値のどちらかしか取得できないことが保証されます。つまり、それ以外の任意の値を取得することはないということです。

9.6.2 final 宣言されたフィールド

オブジェクトのフィールドが final 宣言されている場合のセマンティックスも変更になっています。

> コンストラクタが終了した時点で、オブジェクトは**完全に初期化された**と見なされます。オブジェクトが完全に初期化された後に、そのオブジェクトの参照を得たスレッドは、そのオブジェクトの final 宣言されているフィールドの正しい値を取得できることが保証されます。しかし、final 宣言されていないフィールドの値は、同期なしでは、正しい値を取得できることは保証されません。

また、final 宣言されたフィールドがオブジェクトや配列への参照の場合、そのフィールドに参照の値が書き込まれた時点の参照されているオブジェクトや配列の内容を取得できることが保証されます。

コンストラクタ呼び出しが完了して、生成されたオブジェクトの参照が、オブジェクト生成を行ったスレッド（スレッド A）に返された場合を考えてみてください。そして、その返された参照が、他のスレッド（スレッド B）により単純に参照されたとします。この場合、そのオブジェクトのフィールドで final 宣言されていないフィールドに関しては、スレッド B が正しい値を取得できることは保証されません。次のコードを見てください。

```
public class FinalFieldExample {
    public final int x;
    public int y;
    public static FinalFieldExample f;

    public FinalFieldExample() {
        x = 3;
        y = 4;
    }

    public static void writer() {
        f = new FinalFieldExample();
    }

    public static void reader() {
        if (f != null) {
            int i = f.x; // 正しい値が保証される
            int j = f.y; // 正しい値は保証されない
        }
    }
}
```

スレッド A が writer メソッドを呼び出して、FinalFieldExample クラスのインスタンスを生成したとします。その後に、スレッド B が reader メソッドを呼び出して、そのインスタンスのフィールドをアクセスしたとします。この場合、f.x は、final 宣言されたフィールドですので、i に 3 が代入されることは保証されます。しかし、f.y は、final 宣言されていないフィールドですので、j に 4 が代入されることは保証されません。

つまり、コンストラクタの処理が完了した時点で、final 宣言されたフィールドの値は、その後で、そのオブジェクトの参照を入手したすべてのスレッドに対して同じ値であることが保証されます。もし、コンストラクタの処理の中で、this で表されているそのオブジェクト自身の参照を他のスレッドに渡した場合には、参照を取得したスレッドから、final 宣言されたフィールドの値が正しく取得できることは保証されません。

したがって、不変オブジェクト（*immutable object*）を設計する際には、注意が必要です。次の単純な Point クラスを見てください。

```
// 不変オブジェクトとならない Point クラス
public final class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int getX() { return x; }
    int getY() { return y; }
}
```

この Point クラスは、final 宣言されていますので、サブクラスが作成される心配はありません。したがって、一旦インスタンスが生成されたら内容が変更にならない不変オブジェクトを生成できるように見えます。そのため、getX メソッドも getY メソッドも、synchronized 宣言されていません。しかし、実際には、x フィールドと y フィールドが final 宣言されていませんので、複数のスレッドから getX メソッドおよび getY メソッドが呼び出されたとしたら、すべてのスレッドがコンストラクタで設定された値を取得できることは保証されません。この場合、不変オブジェクトにするためには、フィールドを final 宣言する[*7]必要があります。

String クラスのインスタンスは、一度生成されれば、不変オブジェクトであると考えられてきましたが、実際には、いくつかの private フィールドがリリース 1.4 までは final 宣言されておらず、リリース 5.0 で修正されています。

9.7 型によるプログラミング

基本データ型に対応するラッパークラスには、ラッパークラス間での様々な不整合を取り除くためにメソッドが追加されています。

- 対応する基本データ型をパラメータとして受け取り、ラッパークラスのインスタンスを返す valueOf メソッド（2.8 節参照）がすべてのクラスに用意されました。
- Boolean クラスを除いて、リリース 5.0 で、対応する基本データ型が何バイトであるかを表す SIZE 定数が追加されています。

9.7.1 Boolean クラス

Boolean クラスには、次の static メソッドが追加されています。

public static boolean parseBoolean(String s)

指定された文字列が null ではなく、かつ、大文字と小文字の区別なしに"true"と同じであれば、

[*7] 『Effective Java』[Bloch01]、項目 13「不変性を選ぶ」（第 2 版、項目 15「可変性を最小限にする」）

true を返します。そうでなければ、false を返します。

public static Boolean **valueOf(boolean b)**

指定された boolean 値を表す Boolean クラスのインスタンスを返します。指定された値が true であれば、Boolean.TRUE が返され、そうでなければ Boolean.FALSE が返されます。新たなオブジェクトが生成されて返されることはありません。

public static String **toString(boolean b)**

指定された値が true であれば、"true"を返します。そうでなければ、"false"を返します。

また、リリース 1.4 までは、Boolean クラスは、Comparable インタフェースを実装していませんでしたが、リリース 5.0 からは Comparable<Boolean>として実装するようになりましたので、次のメソッドが追加されています。

public int **compareTo(Boolean b)**

指定された他の Boolean インスタンスと大きさを比較します。どちらも true か false で等しければ、0 を返します。等しくなく、かつ、このインスタンスが保持する値が true なら 1 を返します。そうでなければ、-1 を返します。

この変更により、Boolean クラスのインスタンスをソートされたマップ（たとえば、TreeMap クラス）などのキーとして使用することが可能となります。[*8]

9.7.2 Integer クラスと Long クラス

リリース 5.0 からは、Integer クラスと Long クラスにビットやバイトに関するメソッドが追加されています。以下の説明は、Integer クラスのメソッドですが、同じメソッドが Long クラスにも用意されています。

public static int **highestOneBit(int i)**

指定された int 値の 2 の補数 2 進表現において、1 となっている最上位ビット位置の 1 だけを残した値を返します。たとえば、int 値が 16 進表現で 0x0FFFFFFF だとすると、0x08000000 となります。指定された int 値が 0 であれば、0 が返されます。

public static int **lowestOneBit(int i)**

指定された int 値の 2 の補数 2 進表現において、1 となっている最下位ビット位置の 1 だけを残した値を返します。たとえば、int 値が 16 進表現で 0xFFFFFFF0 だとすると、0x00000010 となります。指定された int 値が 0 であれば、0 が返されます。

public static int **numberOfLeadingZeros(int i)**

指定された int 値の 2 の補数 2 進表現において、1 となっている最上位ビット位置の 1 の前に、何

[*8] Boolean クラスのインスタンス同士が比較できることは、直感的ではないかと思います。つまり、基本データ型である boolean の値同士には、大小比較は適用できません。そのため、長い間、Boolean クラスは、Comparable インタフェースを実装していませんでした。しかし、Swing の JTable を使用して表を作成した場合には、どこかの列で保持されている値が、Boolean クラスのインスタンスということもあり得ます。表ですので、その列の値でソートしたくなる必要がある場合もあり、リリース 1.2 の頃から Comparable インタフェースを実装するように要求が出ていました。

ビットの 0 があるかを返します。たとえば、int 値が 16 進表現で 0x12345678 だとすると、3 となります。指定された int 値が 0 であれば、32 が返されます。

`public static int` **numberOfTrailingZeros(int i)**

指定された int 値の 2 の補数 2 進表現において、1 となっている最下位ビット位置の 1 の後に、何ビットの 0 があるかを返します。たとえば、int 値が 16 進表現で 0x12345670 だとすると、4 となります。指定された int 値が 0 であれば、0 が返されます。

`public static int` **bitCount(int i)**

指定された int 値の 2 の補数 2 進表現において、1 となっているビット数を返します。たとえば、int 値が 16 進表現で 0x12345678 だとすると、13 となります。

`public static int` **rotateLeft(int i, int distance)**

指定された int 値の 2 の補数 2 進表現において、distance で指定されたビット数分だけ、循環的に左にビットシフトします。たとえば、int 値が 16 進表現で 0x12345678 で、distance が 4 だとすると、0x23456781 となります。distance が負の値であれば、rotateRight(i, -distance) の呼び出しと同じになります。

`public static int` **rotateRight(int i, int distance)**

指定された int 値の 2 の補数 2 進表現において、distance で指定されたビット数分だけ、循環的に右にビットシフトします。たとえば、int 値が 16 進表現で 0x12345678 で、distance が 4 だとすると、0x81234567 となります。distance が負の値であれば、rotateLeft(i, -distance) の呼び出しと同じになります。

`public static int` **reverse(int i)**

指定された int 値の 2 の補数 2 進表現において、ビット列を逆順に並べたビット列に変換した結果を返します。たとえば、int 値が 16 進表現で 0x12345678 だとすると、0x1e6a2c48 となります。

`public static int` **signum(int i)**

指定された int 値の値が負であれば-1 を、0 であれば 0 を、正であれば 1 を返します。

`public static int` **reverseBytes(int i)**

指定された int 値の 2 の補数 2 進表現において、バイト単位で、逆順に並べたバイト列に変換した結果を返します。たとえば、int 値が 16 進表現で 0x12345678 だとすると、0x78563412 となります。

最後の reverseBytes は、Short クラスと Character クラスにも用意されています。

これらのメソッドの実装を見ると、何をやっているのかさっぱり分からないようなコードが書かれています。たとえば、reverse メソッドの実装は、以下の通りです。

```
public static int reverse(int i) {
    // HD, Figure 7-1
    i = (i & 0x55555555) << 1 | (i >>> 1) & 0x55555555;
    i = (i & 0x33333333) << 2 | (i >>> 2) & 0x33333333;
    i = (i & 0x0f0f0f0f) << 4 | (i >>> 4) & 0x0f0f0f0f;
    i = (i << 24) | ((i & 0xff00) << 8) |
        ((i >>> 8) & 0xff00) | (i >>> 24);
    return i;
}
```

コメントを見ると HD, Figure 7-1 と書かれています。この HD とは、書籍『ハッカーのたのしみ』(“*Hacker's Delight*”) [Warren02] [*9] を指しています。この本には、この一見すると難解なアルゴリズムの解説が書かれています。

9.7.3 Float クラスと Double クラス

Float オブジェクト同士の比較は、float 値同士の比較と異なった振る舞いをします。Double オブジェクトと double 値に関しても同じです。すなわち、ラッパークラスのオブジェクトの compareTo メソッドで比較した場合には、-0.0 は+0.0 より小さく、NaN は（$+\infty$ を含む）すべての値より大きく、すべての NaN はお互いに等しいと扱われます。リリース 1.3 までは、float 値に対して、このような比較を行うためには、Float オブジェクトへ変換してから行う必要がありましたが、リリース 1.4 からは次のメソッドが追加されています。同様のメソッドは、Double クラスにも追加されています。

`public static int` **compare(float f1, float f2)**

2 つの float 値を比較して、その結果を返します。比較された結果の符号は、次の呼び出しで返される結果の符号と同じです。

```
new Float(f1).compareTo(new Float(f2))
```

リリース 5.0 からは浮動小数点リテラルの 16 進数表現（9.1.3 節）が導入されましたので、Float クラスには、16 進数表現に変換するための次のメソッドが追加されています。同様のメソッドは、Double クラスにも追加されています。

`public static String` **toHexString(float f)**

指定された float 値を、16 進数表現の文字列に変換します。たとえば、1.25f は、"0x1.4p0"に変換されます。負の値の場合には、先頭に"-"が付きますので、-1.25f は"-0x1.4p0"に変換されます。正の無限大と負の無限大は、それぞれ、"Infinity"と"-Infinity"に変換されます。また、NaN は、"NaN"に変換されます。変換についてのさらなる詳細については、toHexString メソッドのオンラインドキュメントを参照してください。

9.7.4 Character クラス

リリース 1.4 では、Unicode3.0 に対応して、次の 2 つのメソッドが追加されています。

`public static byte` **getDirectionality(char ch)**

指定された文字の Unicode における方向性を返します。文字の方向性は、テキストの視覚的な順序を計算するのに使用されます。未定義の文字に対する方向性としては、DIRECTIONALITY_UNDEFINED が返されます。このメソッドは、補助文字を正しく処理しませんので、その場合には、getDirectionality(int codePoint) メソッドを使用する必要があります。

[*9] 2002 年 12 月に、Joshua Bloch 氏を、彼のオフィスに訪ねた時に紹介された本です。英語版には、彼や Guy Steele 氏のコメントが裏カバーに印刷されています。日本語版では、カバーにコメントの訳が印刷されています。

`public static boolean` **isMirrored(char ch)**

指定された文字が Unicode 仕様に従った鏡文字であるか否かを返します。鏡文字は、右から左に書かれた場合に、垂直方向に鏡像となる文字があるものです。たとえば、'\u0028' LEFT PARENTHESIS は、意味的には、開き括弧として定義されています。この文字は、左から右に書かれた場合には、'(' となりますが、右から左に書かれた場合には、')' となります。

詳細な使用方法は、本書の範疇外ですが、方向性に関しては、次の定数が定義されています。

```
DIRECTIONALITY_UNDEFINED
DIRECTIONALITY_LEFT_TO_RIGHT
DIRECTIONALITY_RIGHT_TO_LEFT
DIRECTIONALITY_RIGHT_TO_LEFT_ARABIC
DIRECTIONALITY_EUROPEAN_NUMBER
DIRECTIONALITY_EUROPEAN_NUMBER_SEPARATOR
DIRECTIONALITY_EUROPEAN_NUMBER_TERMINATOR
DIRECTIONALITY_ARABIC_NUMBER
DIRECTIONALITY_COMMON_NUMBER_SEPARATOR
DIRECTIONALITY_NONSPACING_MARK
DIRECTIONALITY_BOUNDARY_NEUTRAL
DIRECTIONALITY_PARAGRAPH_SEPARATOR
DIRECTIONALITY_SEGMENT_SEPARATOR
DIRECTIONALITY_WHITESPACE
DIRECTIONALITY_OTHER_NEUTRALS
DIRECTIONALITY_LEFT_TO_RIGHT_EMBEDDING
DIRECTIONALITY_LEFT_TO_RIGHT_OVERRIDE
DIRECTIONALITY_RIGHT_TO_LEFT_EMBEDDING
DIRECTIONALITY_RIGHT_TO_LEFT_OVERRIDE
DIRECTIONALITY_POP_DIRECTIONAL_FORMAT
```

9.1.1 節で説明したように、Unicode 4.0 に対応するために、char 型では保持できないコードポイントを int 型で表現するようになりました。それに対応して、int 型のコードポイントを引数に取るように、以下の static メソッドがオーバーロードする形式で追加されています。

```
digit(int codePoint, int radix)
getDirectionality(int codePoint)
getNumericValue(int codePoint)
getType(int codePoint)
isDefined(int codePoint)
isDigit(int codePoint)
isIdentifierIgnorable(int codePoint)
isISOControl(int codePoint)
isJavaIdentifierPart(int codePoint)
isJavaIdentifierStart(int codePoint)
isLetter(int codePoint)
isLetterOrDigit(int codePoint)
isLowerCase(int codePoint)
isMirrored(int codePoint)
isSpaceChar(int codePoint)
isTitleCase(int codePoint)
isUnicodeIdentifierPart(int codePoint)
isUnicodeIdentifierStart(int codePoint)
isUpperCase(int codePoint)
isWhitespace(int codePoint)
toLowerCase(int codePoint)
toTitleCase(int codePoint)
toUpperCase(int codePoint)
```

コードポイントに関連して追加された static メソッドは、次の通りです。

`public static boolean` **isValidCodePoint(int codePoint)**
指定されたコードポイントが、コードポイントとしての有効な範囲（0x0000 から 0x10FFFF）の値かを調べた結果を返します。

`public static boolean` **isSupplementaryCodePoint(int codePoint)**
指定されたコードポイントが、補助文字としての範囲（0x10000 から 0x10FFFF）のコードポイントかを調べた結果を返します。

`public static boolean` **isHighSurrogate(char ch)**
指定された char の値が、上位代理範囲（\uD800 から \uDBFF）の値かを調べた結果を返します。この範囲の値は、それ自身では文字を表していませんが、UTF-16 エンコーディングにおいて補助文字を表現するのに使用されます。

`public static boolean` **isLowSurrogate(char ch)**
指定された char の値が、下位代理範囲（\uDC00 から \uDFFF）の値かを調べた結果を返します。この範囲の値は、それ自身では文字を表していませんが、UTF-16 エンコーディングにおいて補助文字を表現するのに使用されます。

`public static boolean` **isSurrogatePair(char high, char low)**
指定された 2 つの char の値の組み合わせが、有効な代理組合わせを表しているかを調べて返します。

`public static int` **charCount(int codePoint)**
指定されたコードポイントを、char の値で表現するのに必要な char 数を返します。コードポイントとして指定された文字が、0x10000 以上であれば、このメソッドは 2 を返します。そうでなければ、1 を返します。

`public static int` **toCodePoint(char high, char low)**
指定された代理組合せの 2 つの char を、コードポイントへ変換します。正しい代理組合せであるかは検査しませんので、呼び出し側で isSurrogatePair メソッドを使用して検査する必要があります。

`public static int` **codePointAt(CharSequence seq, int index)**
CharSequence 内の index で指定されたインデックス位置の文字（Unicode のコードポイント）を返します。index は、0 から seq.length() - 1 の範囲でなければなりません。index で指定された位置の char が上位代理範囲内であり、index + 1 が seq.length() - 1 以下であり、次の位置にある char が下位代理範囲内であれば、それらの 2 つの char で表される補助文字のコードポイントを返します。そうでなければ、index で指定された位置の char だけが返されます。

`public static int` **codePointAt(char[] a, int index)**
char 配列内の index で指定されたインデックス位置の文字（Unicode のコードポイント）を返します。index は、0 から a.length - 1 の範囲でなければなりません。index で指定された位置の char が上位代理範囲内であり、index + 1 が a.length - 1 以下であり、次の位置にある char が下位代理範囲内であれば、それらの 2 つの char で表される補助文字のコードポイントを返します。そうでなければ、index で指定された位置の char だけが返されます。

`public static int` **codePointAt(char[] a, int index, int limit)**

char 配列内の index で指定されたインデックス位置の文字（Unicode のコードポイント）を返します。index は、0 から limit - 1 の範囲でなければなりません。index で指定された位置の char が上位代理範囲内であり、index + 1 が limit - 1 以下であり、次の位置にある char が下位代理範囲内であれば、それらの 2 つの char で表される補助文字のコードポイントを返します。そうでなければ、index で指定された位置の char だけが返されます。

`public static int` **codePointBefore(CharSequence seq, int index)**

CharSequence 内の index で指定されたインデックス位置の前の文字（Unicode のコードポイント）を返します。index は、1 から seq.length() - 1 の範囲でなければなりません。index - 1 の位置にある char が下位代理範囲内であり、かつ、index - 2 が 0 以上で、その位置にある char が上位代理範囲内であれば、それらの 2 つの char で表される補助文字のコードポイントを返します。そうでなければ、index - 1 の位置にある char だけが返されます。

`public static int` **codePointBefore(char[] a, int index)**

char 配列内の index で指定されたインデックス位置の前の文字（Unicode のコードポイント）を返します。index は、1 から a.length - 1 の範囲でなければなりません。index - 1 の位置にある char が下位代理範囲内であり、かつ、index - 2 が 0 以上で、その位置にある char が上位代理範囲内であれば、それらの 2 つの char で表される補助文字のコードポイントを返します。そうでなければ、index - 1 の位置にある char だけが返されます。

`public static int` **codePointBefore(char[] a, int index, int start)**

char 配列内の index で指定されたインデックス位置の前の文字（Unicode のコードポイント）を返します。index は、start から a.length - 1 の範囲でなければなりません。index - 1 の位置にある char が下位代理範囲内であり、かつ、index - 2 が 0 以上で、その位置にある char が上位代理範囲内であれば、それらの 2 つの char で表される補助文字のコードポイントを返します。そうでなければ、index - 1 の位置にある char だけが返されます。

`public static int` **toChars(int codePoint, char[] dst, int dstIndex)**

指定されたコードポイントを UTF-16 形式に変換します。コードポイントが、基本多言語プレーンの文字であれば、dst[dstIndex] に同じ値が設定されて 1 が返されます。補助文字であれば、代理組合せに変換されて、上位代理範囲の文字が dst[dstIndex] に設定され、下位代理範囲の文字が dst[dstIndex + 1] に設定され、2 が返されます。

`public static char[]` **toChars(int codePoint)**

指定されたコードポイントを UTF-16 形式に変換した結果の char の配列を返します。

`public static int` **codePointCount(CharSequence seq, int beginIndex, int endIndex)**

指定された CharSequence 内の beginIndex から endIndex - 1 の範囲にあるコードポイント数を返します。上位代理範囲と下位代理範囲の対となっていない代理範囲の char は、1 個のコードポイントとして数えられます。

`public static int` **codePointCount(char[] a, int offset, int count)**

指定された配列のインデックス位置が offset から offset + count - 1 までの範囲にあるコードポイント数を返します。上位代理範囲と下位代理範囲の対となっていない代理範囲の char は、

1 個のコードポイントとして数えられます。

`public static int` **offsetByCodePoints(CharSequence seq,int index,int codePointOffset)**

指定された CharSequence 内の index で指定された位置から codePointOffset 数のコードポイントを表す char の次の位置が、インデックスとして返されます。上位代理範囲と下位代理範囲の対となっていない代理範囲の char は、1 個のコードポイントとして数えられます。

`public static int` **offsetByCodePoints(char[] a, int start,int count,int index,int cpOffset)**

指定された配列のインデックス位置が start から start + count - 1 までの範囲で、index で指定された位置から cpOffset 数のコードポイントを表す char の次の位置が、インデックスとして返されます。上位代理範囲と下位代理範囲の対となっていない代理範囲の char は、1 個のコードポイントとして数えられます。

コードポイントに関して、以下の定数も追加されています。

```
MIN_HIGH_SURROGATE  MAX_LOW_SURROGATE  MIN_SUPPLEMENTARY_CODE_POINT
MAX_HIGH_SURROGATE  MIN_SURROGATE      MIN_CODE_POINT
MIN_LOW_SURROGATE   MAX_SURROGATE      MAX_CODE_POINT
```

9.7.5 Class クラス

Class クラスには、リリース 5.0 で多くのメソッドが追加されています。

`public boolean` **isAnnotation()**

この Class オブジェクトがアノテーション型を表していれば true を返します。

`public boolean` **isSynthetic()**

Java 言語仕様に定義されている**合成**（*synthetic*）[*10]であるか否かを返します。

`public TypeVariable<Class<T>>[]` **getTypeParameters()**

この Class オブジェクトがジェネリック宣言されている型を表している場合、その型変数（TypeVariable）の配列を返します。配列の内容は、型変数が宣言されている順番に格納されています。ジェネリック宣言されていない場合には、長さが 0 の配列が返されます。

`public Type` **getGenericSuperclass()**

この Class オブジェクトで表されている型の直接のスーパークラスを表す Type を返します。

`public Type[]` **getGenericInterfaces()**

この Class オブジェクトで表されている型が直接実装しているインタフェースを表す Type の配列を返します。

`public Method` **getEnclosingMethod()**

この Class オブジェクトが、メソッド内で宣言されたローカルクラスあるいは無名クラスを表している場合には、そのメソッドを返します。そうでなければ、null が返されます。

[*10] デフォルトのコンストラクタとクラス初期化メソッドを除いて、コンパイラによって導入された、ソースコード中に対応する構造のない構造を指します。たとえば、コンパイラが生成するブリッジメソッド（1.3 節参照）も合成です。

`public Constructor<?>` **getEnclosingConstructor()**

この Class オブジェクトが、コンストラクタ内で宣言されたローカルクラスあるいは無名クラスを表している場合には、そのコンストラクタを返します。そうでなければ、null が返されます。

`public Class<?>` **getEnclosingClass()**

この Class オブジェクトが表す型の直接のエンクロージングクラスを返します。もし、この Class オブジェクトで表される型がトップレベルである場合には、null が返されます。

`public String` **getSimpleName()**

ソースコードに書かれたクラス名を返します。無名クラスの場合には、長さ 0 の文字列が返されます。

`public String` **getCanonicalName()**

Java 言語仕様で定義されている**正準名** (*canonical name*) を返します。もし、正準名が存在しない場合には、null が返されます。

`public boolean` **isAnonymousClass()**

この Class オブジェクトが、無名クラスを表していれば、true を返します。

`public boolean` **isLocalClass()**

この Class オブジェクトが、ローカルクラス[*11]を表していれば、true を返します。

`public boolean` **isMemberClass()**

この Class オブジェクトが、メンバークラス[*12]を表していれば、true を返します。

`public boolean` **isEnum()**

この Class オブジェクトが、enum を表していれば、true を返します。

`public T[]` **getEnumConstants()**

この Class オブジェクトが、enum を表していれば、定義されている enum 定数の配列を返します。enum を表していなければ、null を返します。

`public T` **cast(Object obj)**

1.8.1 節を参照。

`public <U> Class<? extends U>` **asSubclass(Class⟨U⟩ clazz)**

1.8.2 節を参照。

Class クラスは、java.lang.reflect.AnnotatedElement インタフェースを実装したメソッドを提供していますが、それらについては、第 8.1 節を参照してください。

9.7.6 Field クラス

Field クラスには、リリース 5.0 で以下のメソッドが追加されています。

`public boolean` **isEnumConstant()**

この Field オブジェクトが、enum 定数であるフィールドを表していれば、true を返します。

[*11] メソッド本体、コンストラクタ、初期化ブロックなどのコードブロックの中で定義されているクラス。
[*12] クラスのメンバーとして宣言されているクラス。

`public boolean` **isSynthetic()**
Java 言語仕様に定義されている**合成**（*synthetic*）であるか否かを返します。

`public Type` **getGenericType()**
この `Field` オブジェクトが表しているフィールドの宣言された型を表す `Type` オブジェクトを返します。返された `Type` オブジェクトがパラーメータ化された型を表している場合には、ソースコード上の実際の型パラメータを反映しています。

`public String` **toGenericString()**
ジェネリック型を含むこのフィールドを表す文字列を返します。

9.7.7 `Method` クラス

`Method` クラスには、リリース 5.0 で以下のメソッドが追加されています。

`public TypeVariable<Method>[]` **getTypeParameters()**
この `Method` オブジェクトで表されるメソッドの宣言において宣言された型変数を表す `TypeVariable` オブジェクトの配列を返します。型変数が宣言されていなければ、長さ 0 の配列が返されます。

`public Type` **getGenericReturnType()**
この `Method` オブジェクトで表されるメソッドの正式な戻り値型を表す `Type` オブジェクトを返します。返された `Type` オブジェクトがパラーメータ化された型を表している場合には、ソースコード上の実際の型パラメータを反映しています。

`public Type[]` **getGenericParameterTypes()**
この `Method` オブジェクトで表されるメソッドのパラメータの正式な型を、宣言されている順に並んでいる `Type` オブジェクトの配列として返します。パラメータを何も取らないメソッドであれば、長さ 0 の配列が返されます。返された配列内の `Type` オブジェクトがパラーメータ化された型を表している場合には、ソースコード上の実際の型パラメータを反映しています。

`public Type[]` **getGenericExceptionTypes()**
この `Method` オブジェクトで表されるメソッドがスローすると宣言している例外を表す `Type` オブジェクトの配列を返します。何もスローすると宣言されていなければ、長さ 0 の配列が返されます。返された配列内の `Type` オブジェクトがパラーメータ化された型を表している場合には、ソースコード上の実際の型パラメータを反映しています。

`public String` **toGenericString()**
ジェネリック型を含むこのメソッドを表す文字列を返します。

`public boolean` **isBridge()**
この `Method` オブジェクトで表されるメソッドが、ブリッジメソッド（1.3 参照）ならば `true` を返します。

`public boolean` **isVarArgs()**
この `Method` オブジェクトで表されるメソッドが、可変長のパラメータを取るメソッドとして宣言

されていれば true を返します。

public boolean **isSynthetic()**

Java 言語仕様に定義されている**合成**（*synthetic*）であるか否かを返します。

public Object **getDefaultValue()**

この Method オブジェクトで表されるメソッドが、アノテーション型のメンバーを表している場合に、デフォルト値を返します。メンバーの型が基本データ型ならば、ラッパークラスのオブジェクトとして返されます。デフォルト値が宣言されていなかったり、アノテーション型のメンバーを表していなかったりした場合には、null が返されます。

public Annotation[][] **getParameterAnnotations()**

この Method オブジェクトで表されるメソッドのパラメータに付けられているアノテーションを、Annotation オブジェクトの配列の配列として返します。パラメータが何も宣言されていなければ、長さ 0 の配列が返されます。各パラメータにアノテーションが付けられていなければ、ネストした配列の長さは 0 となります。

9.7.8 Constructor クラス

Constrcutor クラスには、リリース 5.0 で以下のメソッドが追加されています。

public TypeVariable<Constructor<T>>[] **getTypeParameters()**

この Constructor オブジェクトで表されるコンストラクタの宣言において宣言された型変数を表す TypeVariable オブジェクトの配列を返します。型変数が宣言されていなければ、長さ 0 の配列が返されます。

public Type[] **getGenericParameterTypes()**

この Constructor オブジェクトで表されるコンストラクタのパラメータの正式な型を、宣言されている順に並んでいる Type オブジェクトの配列として返します。パラメータを何も取らないコンストラクタであれば、長さ 0 の配列が返されます。返された配列内の Type オブジェクトがパラーメータ化された型を表している場合には、ソースコード上の実際の型パラメータを反映しています。

public Type[] **getGenericExceptionTypes()**

この Constructor オブジェクトで表されるコンストラクタがスローすると宣言している例外を表す Type オブジェクトの配列を返します。何もスローすると宣言されていなければ、長さ 0 の配列が返されます。返された配列内の Type オブジェクトがパラーメータ化された型を表している場合には、ソースコード上の実際の型パラメータを反映しています。

public String **toGenericString()**

ジェネリック型を含むこのコンストラクタを表す文字列を返します。

public boolean **isVarArgs()**

この Constructor オブジェクトで表されるコンストラクタが、可変長のパラメータを取るコンストラクタとして宣言されていれば true を返します。

`public boolean` **isSynthetic()**

Java 言語仕様に定義されている**合成**（*synthetic*）であるか否かを返します。

`public Annotation[][]` **getParameterAnnotations()**

この `Constructor` オブジェクトで表されるコンストラクタのパラメータに付けられているアノテーションを、`Annotation` オブジェクトの配列の配列として返します。パラメータが何も宣言されていなければ、長さ 0 の配列が返されます。各パラメータにアノテーションが付けられていなければ、ネストした配列の長さは 0 となります。

9.8 ガーベッジコレクションとメモリ

9.8.1 ガーベッジコレクタとのやり取り

リリース 1.4 からは、`Runtime` クラスに `maxMemory` メソッドが追加されています。

`public long` **maxMemory()**

Java 仮想マシンが利用可能なメモリの最大値を返します。単位はバイトです。もし、上限値がない場合には、`Long.MAX_VALUE` が返されます。

次のプログラムは、`maxMemory` メソッド、`totalMemory` メソッド、`freeMemory` メソッドの結果を出力します。

```
public class TestMemory {
    public static void main(String[] args) {
        Runtime rt = Runtime.getRuntime();
        System.out.println("max memory   = " + rt.maxMemory());
        System.out.println("total memory = " + rt.totalMemory());
        System.out.println("free memory  = " + rt.freeMemory());
    }
}
```

J2SE 5.0 の java コマンドのデフォルト設定のまま、Windows XP 上で実行すると、結果は次のようになりました。

```
max memory   = 66650112
total memory = 2031616
free memory  = 1826176
```

実行時に、これらの値を調べることで、メモリの使用状況をある程度把握することが可能となります。

9.9 ドキュメンテーションコメント

9.9.1 新たなタグ

新たなタグとしては、以下のタグが追加されています。

{@linkplain}
: {@link} タグと同じですが、リンク文字がコードフォントではなく、通常フォントで表示されます。

{@inheritdoc}
: スーパータイプのドキュメンテーションコメントをコピーしてきます。リリース 1.3 までは、スーパータイプのドキュメンテーションコメントをすべて継承するか、あるいは、すべて書き換えるかしかできませんでしたが、このタグにより、スーパータイプのドキュメンテーションコメントを新たなコメント内の一部にコピーすることが可能となります。

@serial *field-description* | include | exclude
: 「直列化された形式」のページに表示される、デフォルトでシリアライズされるフィールドにドキュメンテーションコメントを付けるのに使用します。*field-description* は、オプションであり、フィールドに対する説明を記述します。

  また、@serial タグは、パッケージやクラスが、「直列化された形式」のページに表示されるかどうかを指定するのにも使用します。デフォルトでは、Serializable インタフェースを実装した public と protected のクラスが表示され、Serializable インタフェースを実装したパッケージプライベート[*13]と private のクラスは表示されません。このデフォルトの振る舞いを@serial include タグ、あるいは、@serial exclude タグで変更することができます。なお、クラスに対する@serial タグの指定は、パッケージに対する@serial タグよりも優先されます。

{@value *package.class#field*}
: オプションなしで、{@value} として、static のフィールドに付けられたコメント内で使用されると、そのフィールドの値を表示します。表示される値は、「定数フィールド値」のページに表示される値です。また、オプションとして、{@value *package.class#field*} として、特定のクラスのフィールドを指定することで、任意のドキュメンテーションコメント内で、そのフィールドの定数値を表示することができます。

{@literal}
: {@literal} タグは、リテラルテキストを表します。このタグに含まれるテキストは、HTML タグやネストした javadoc タグとは解釈されません。たとえば、{@literal a<B>c} は、生成された HTML では、a<B>c と表示されて、<B>がボールドタグとは解釈されません。

{@code}
: {@code} タグは、リテラルテキストをコードフォントで表示します。{@code abc} は、<code>{@literal abc}</code>と同じです。

[*13] public、protected、private のどのアクセス修飾子も指定されていない場合のデフォルトアクセス。

また、7.1.2 節で説明したように@deprecated タグの代わりに、@Deprecated アノテーションを使用することが可能です。

9.9.2 package-info.java ファイル

リリース 1.4 までは、パッケージに関するドキュメンテーションコメントは、package.html ファイルに記述しなければなりませんでした。リリース 5.0 からは、package-info.java ファイルにパッケージに関するドキュメンテーションコメントやパッケージに対するアノテーションを記述することができます。package-info.java ファイル内には、該当するパッケージ宣言だけを行います。そして、その宣言の前に、通常のドキュメンテーションコメントの形式でコメントを追加したり、アノテーションを記述したりします。

9.10 I/O

9.10.1 java.nio パッケージ

本書では詳細は説明しませんが、リリース 1.4 から java.nio パッケージとそれに関連したサブパッケージが追加されています。java.io パッケージのクラスが**ストリーム**（*stream*）として I/O を取り扱うのに対して、java.nio パッケージは、**バッファ**（*buffer*）と**チャネル**（*channel*）という新たな概念で I/O を取り扱います。バッファに対してはデータの読み書きができ、チャネルは I/O 操作が可能なバッファ、ファイル、ソケットなどを表します。また、java.nio パッケージは、ブロックしない I/O 操作も提供します。

9.10.2 Closeable インタフェースと Flushable インタフェース

リリース 5.0 からは、java.io パッケージに、Closeable インタフェースと Flushable インタフェースが追加されています。Closeable インタフェースには close メソッド、Flushable インタフェースには flush メソッドが、次のように定義されているだけです。

```
public interface Closeable {
    public void close() throws IOException;
}

public interface Flushable {
    void flush() throws IOException;
}
```

これらのインタフェースは、Formatter クラスと Scanner クラスを実装するために導入されています。close メソッドを持っていた入出力関連のクラスやインタフェースは、Closeable インタフェースを実装あるいは継承するように変更されていますし、flush メソッドを持っていた入出力関連のクラスやインタフェースも同様に Flushable インタフェースを実装あるいは継承するように変更されてい

ます。

9.10.3 printf メソッド

PrintStream クラスには、可変長パラメータを使用した以下の printf メソッドが追加されています。

```
public PrintStream printf(String format, Object ... args)
public PrintStream printf(Locale l, String format, Object ... args)
```

printf と同様に、format メソッドもあり、どちらも同じ処理を行います。また、フォーマットされた文字列を生成するために、String クラスに、static の format メソッドも追加されています。実際のフォーマット処理は、java.util.Formatter クラスで実装されており、その実装を呼び出すようになっています。

printf メソッドの format 引数の文字列内で、%を用いて指定するフォーマット形式は以下の通りです。

%[*argument_index*$][*flags*][*width*][.*precision*]*conversion*

- オプションである *argument_index* は、引数リスト内の引数位置を表す 10 進数です。最初の引数は"1$"、第 2 引数は"2$"と指定します。
- オプションである *flags* は、出力形式に対する詳細な指示をするための文字の集まりです。指定可能な有効文字は、*conversion* に依存します。詳細については、java.util.Formatter クラスのドキュメンテーションを参照してください。
- オプションである *width* は、出力する文字列の最小文字数を示す負でない 10 進数です。
- オプションである *precision* は、基本的には出力文字数を制限するための負でない 10 進数です。このオプションの振舞いについては、*conversion* に依存します。このオプションは、日時をフォーマットする場合には、指定できません。
- 必須である *conversion* は、引数のフォーマット形式を指定する 1 つの文字です。ただし、日時をフォーマットする場合には、2 文字となります。引数に対して指定可能な文字は、引数の型に依存します。

conversion として指定できる文字は、以下の通りです。大文字、小文字のどちらでも指定可能なものに関しては、両方示してあります。大文字が指定されている場合には、出力文字は、locale に従って、String クラスの toUpperCase メソッドを用いて大文字に変換されたものと同じ結果になります。

b, B ブール値としてフォーマットします。もし、引数が null ならば false となります。引数が boolean もしくは Boolean ならば、それらの値に応じて true または false となります。それ以外の場合には、true となります。

h, H ハッシュコードを 16 進で表示します。

s, S 引数が null の場合には、"null"となります。引数が Formattable インタフェースを実装していれば、その formatTo メソッドが呼び出された結果となります。そうでなければ、引数の toString メソッドが呼び出された結果となります。

c, C Unicode 文字となります。

d 10 進数としてフォーマットします。

o 8 進数としてフォーマットします。

x, X 16 進数としてフォーマットします。

e, E **コンピュータ化された科学的記数法**（*computerized scientific notation*）で浮動小数点としてフォーマットします。

f 10 進形式で浮動小数点としてフォーマットします。

g, G 精度に応じて、コンピュータ化された科学的記数法または 10 進形式で、浮動小数点をフォーマットします。

a, A 16 進形式で浮動小数点としてフォーマットします。

t*X* 日時を、指定された *X* に従って変換します。*X* として指定できる形式の詳細については、java.util.Formatter クラスのドキュメンテーションを参照してください。

% 結果は、'%'（'\u0025'）です。

n 結果は、プラットフォームごとの行区切り文字となります。

conversion として s が指定された場合、通常は、対象となるオブジェクトに対して toString メソッドを呼び出すことで、フォーマットされます。しかし、対象となるオブジェクトが java.util.Formattable インタフェースを実装していれば、Formattable インタフェースで定義されている formatTo メソッドが呼び出されます。formatTo メソッドの定義は、以下の通りです。

```
void formatTo(Formatter formatter,
              int flags,
              int width,
              int precision)
```

conversion として s が指定された場合の他のオプションである *flags*、*width*、*precision* が引数として渡されてきますので、それらを自分で解釈して、formatter に対して文字列出力をすることで、独自の出力ができるようになります。

出力に関して、printf で簡単になったのと同様に、入力に関しても、より簡単に処理できるようにするために、Scanner クラスが用意されています。Scanner クラスに関しては、9.12.1 節を参照してください。

9.10.4 オブジェクトのシリアライズ

オブジェクトのシリアライズでは、オブジェクトのグラフをそっくりそのまま再現できるように、オブジェクトストリーム内でオブジェクトの参照をキャッシュします。したがって、従来では、ObjectOutputStream インスタンスへ書き込んだオブジェクトに関する情報をクリアーするために

reset メソッドを使用する必要がありました。リリース 1.4 からは、オブジェクトの参照をキャッシュしない書き込みである writeUnshared メソッドが ObjectOutputStream クラスに追加されています。

public void **writeUnshared(Object obj)** throws IOException
: オブジェクトストリームに対して、新たなオブジェクトとして書き込みます。書き込まれようとしているオブジェクトが、すでに writeObject メソッドで以前に書き込まれていても、それへの参照を書き込むのではなく、新たなオブジェクトとして書き込まれます。writeUnshared メソッドで書き込まれたオブジェクトと同一のオブジェクトが writeObject メソッドにより、後で書き込まれたとしても、writeUnshared メソッドですでに書き込まれたオブジェクトを参照することなく、新たなオブジェクトとして書き込まれます。つまり、writeUnshared メソッドで書き込まれたオブジェクトは、全く独立したオブジェクトとして書き込まれることになります。

writeUnshared メソッドに対応して、ObjectInputStream クラスには、次のメソッドが追加されています。

public Object **readUnshared()** throws IOException, ClassNotFoundException
: オブジェクトストリームから、共有されていないオブジェクトを読み込みます。このメソッド呼び出しで読み込もうとしているオブジェクトが、このメソッド以前に読み出されたオブジェクトを参照している場合には、ObjectStreamException がスローされます。また、このメソッド呼び出しで読み込んだオブジェクトが、以降の readObject メソッドあるいは readUnshared メソッドの呼び出しにおいて、読み出そうとするオブジェクトにより参照されていた場合にも、ObjectStreamException がスローされます。

また、ObjectStreamField クラスには、次のコンストラクタとメソッドが追加されています。

public **ObjectStreamField(String name, Class⟨?⟩ type, boolean unshared)**
: unshared が true ならば、この ObjectStreamField クラスのインスタンスで表されるシリアライズ可能なフィールドが、writeUnshared メソッドおよび readUnshared メソッドで読み書きしたのと同様にシリアライズ/ディシリアライズされることを示します。

public boolean **isUnshared()**
: この ObjectStreamField クラスのインスタンスで表されるシリアライズ可能なフィールドが、共有されているか否かを返します。

リリース 1.4 からは、シリアライズ可能なクラスで定義する private の readObject メソッドおよび writeObject メソッドに加えて、readObjectNoData メソッドが追加されています。readObjectNoData メソッドを定義する場合には、そのシグニチャは、以下の通りです。

```
private void readObjectNoData() throws ObjectStreamException;
```

クラス階層において、あるバージョンから、クラス階層途中に別のクラスが追加されたりすることがあります。たとえば、クラス C にはスーパークラスがなかったとして、クラス C のインスタンスがシリアライズされて保存されていたとします。その後、クラス C のスーパークラスとしてクラス S が追

加されたとします。この場合に、古いクラス C の保存されていたシリアライズされたデータを、ディシリアライズしようとしたとします。古いシリアライズされたデータには、スーパークラスであるクラス S の情報は当然保存されていません。

クラス S が、直接あるいは間接的に Serializable インタフェースを実装していない場合には、クラス S のフィールドに対しては、初期化子や引数なしコンストラクタによる初期化が行われます。しかし、Serializable インタフェースを実装している場合に、readObject メソッドが定義されていなければ、すべてのフィールドは、そのフィールドの型に応じた初期値になります。つまり、int などの整数型であれば 0 ですし、参照型であれば null です。[*14] このような場合に、クラス S は readObjectNoData メソッドを定義しておくことで、自分のフィールドに関する初期化を行うことが可能となります。

クラス S の readObjectNoData メソッドが呼び出される条件は、以下の通りです。

- クラス S が、直接あるいは間接的に Serializable インタフェースを実装している。
- クラス S が、readObjectNoData メソッドを、前述のシグニチャで定義している。
- シリアライズストリーム中に、クラス C のスーパークラスとして、クラス S のクラス記述子が含まれていない。

なお、クラス S に呼び出し可能な readObject メソッドが定義されている場合には、readObjectNoData メソッドは呼び出されません。

9.11 コレクション

9.11.1 RandomAccess インタフェース

リリース 1.4 から java.util パッケージに、RandomAccess インタフェースが追加されています。RandomAccess インタフェースは、List インタフェースを実装しているインスタンスが、ランダムにアクセス可能か否かを表すマーカーインタフェースです。すなわち、RandomAccess インタフェースが実装されていれば、get メソッドでインデックスを指定して、個々の要素にアクセスする時間が $O(1)$ であることを示しています。たとえば、ArrayList クラスがこのインタフェースを実装しています。

9.11.2 Queue インタフェース

リリース 5.0 から java.util パッケージには、Queue インタフェースが追加されています。Queue インタフェースの定義は、次の通りです。

[*14] オブジェクトのディシリアライズにおける動作については、『プログラミング言語 Java 第 3 版』の 15.7.3 節「シリアライズとディシリアライズの順序」を参照してください。

```
public interface Queue<E> extends Collection<E> {
    boolean offer(E o);
    E poll();
    E remove();
    E peek();
    E element();
}
```

各メソッドの説明は以下の通りです。

boolean offer(E o)
: キューに指定された要素を挿入します。キューに挿入できた場合には、true を返します。キューに容量制限などがあり、挿入できなかった場合には、false を返します。Collection インタフェースの add メソッドは、挿入できなかった場合には例外をスローする点が、offer メソッドと異なることに注意してください。

E poll()
: キューの先頭から要素を取り除いて返します。キューが空なら null が返されます。

E remove()
: キューの先頭から要素を取り除いて返します。キューが空なら NoSuchElementException がスローされます。

E peek()
: キューの先頭の要素を返しますが、その要素はキューから取り除かれません。キューが空なら null が返されます。

E element()
: キューの先頭の要素を返しますが、その要素はキューから取り除かれません。キューが空なら NoSuchElementException がスローされます。

Queue インタフェースを実装するための骨格実装である AbstractQueue クラスも提供されています。また、LinkedList クラスは、この Queue インタフェースを実装するように変更されていますし、Queue インタフェースを実装した具象クラスとして、優先順位順に要素をキューイングする PriorityQueue クラスも用意されています。PriorityQueue クラスは、スレッドセーフでないことに注意してください。

9.11.3 IdentityHashMap クラス

Map インタフェースの定義では、キーを比較するには、equals メソッドを使用して行うことになっています。それに対して、この契約をあえて破ってキーの参照値だけで比較を行うのが IdentityHashMap です。キーとして k1 および k2 が使用された場合、HashMap クラスでは、キーが同一かの判定は、(k1==null ? k2==null : k1.equals(k2)) で行われます。一方、IdentityHashMap クラスでは、(k1==k2) で行われることになります。このように、Map インタフェースの契約を破っていますので、汎用的なマップとして使用することはできません。

9.11.4 LinkedHashMap クラス

HashMap クラスに対して、キーと値の組をイテレートすると、キーと値の組を入れた順序とは関係なく、ランダムな順序となります。それに対して、キーと値の組を入れた順にイテレートすることを保証しているのが、LinkedHashMap クラスです。

次のテストコードは、0 から 9 までを値として持つ Integer クラスのインスタンスをキーとしています。

```
import java.util.*;

class MapDemo {
    public static void main(String[] args) {
        Map<Integer, Integer> map = new HashMap<Integer,Integer>();

        for (int i = 0; i < 10; i++ )
            map.put(i, i*i);

        for (Map.Entry<Integer, Integer> entry: map.entrySet()) {
            System.out.printf("%d ", entry.getKey());
        }
        System.out.println();
    }
}
```

これをコンパイルして実行すると、次のように、マップに入れた順序とは関係ない順序となります。

```
2 4 9 8 6 1 3 7 5 0
```

次に、HashMap クラスのインスタンスを生成している部分を、次のように LinkedHashMap クラスに変更します。

```
Map<Integer, Integer> map = new LinkedHashMap<Integer,Integer>();
```

再度コンパイルして実行すると、次のように、マップに入れた順序となります。

```
0 1 2 3 4 5 6 7 8 9
```

このように LinkedHashMap クラスを使用すると、入れた順にイテレートすることができます。また、すでにキーが存在する場合に、put メソッドで同じキーに対する値の変更を行った場合には、順序は変更されないことにも注意してください。

LinkedHashMap クラスのコンストラクタの 1 つは、パラメータとして他の Map を受け取ります。その場合は、渡されたマップ内のキーと値の組の順序と全く同じインスタンスが生成されます。

LinkedHashMap クラスのコンストラクタの 1 つに、boolean 型の accessOrder をパラメータとして受け取るものがあります。accessOrder として、true を指定すると、イテレーションの順序が、マップへの put メソッドおよび get メソッドでアクセスした順序となります。最も最初にアクセスさ

れたキーと値の組からイテレートし、最後にアクセスされたキーと値の組がイテレーションの最後となる順序になります。次のコードでは、accessOrder として true を設定しています。

```
import java.util.*;

class MapDemo2 {
    public static void main(String[] args) {
        Map<Integer, Integer> map = new LinkedHashMap<Integer,Integer>(
                                                        16, 0.75f, true);

        for (int i = 0; i < 10; i++ )
            map.put(i, i*i);
        showMap(map);

        map.get(7);
        map.get(5);
        map.get(3);

        showMap(map);
    }

    private static void showMap(Map<Integer, Integer> map) {
        for (Map.Entry<Integer, Integer> entry: map.entrySet()) {
            System.out.printf("%d ", entry.getKey());
        }
        System.out.println();
    }
}
```

最初に LinkedHashMap クラスのインスタンスを生成して、値が 0 から 9 までを持つ Integer クラスのインスタンスをキーとして、要素を追加しています。その後、マップの内容を表示しています。さらに、キーが 7、5、3 の順にアクセスして、再度、マップの内容を表示しています。このコードを実行すると、次の結果となります。

```
0 1 2 3 4 5 6 7 8 9
0 1 2 4 6 8 9 7 5 3
```

順序が入れ替わっていることが分かります。このように LinkedHashMap クラスは、アクセスされた順序で保持する機能も提供されていますので、いわゆる LRU（*least recently used*）キャッシュを作る必要がある場合には役立ちます。

9.11.5 LinkedHashSet クラス

LinkedHashMap クラスと同様に、要素が入れられた順序でイテレートすることが可能となっているセットが LinkedHashSet クラスです。ただし、LinkedHashMap クラスのように、アクセスされた順序でイテレートする機能は提供されていません。

9.11.6 Arrays ユーティリティクラス

Arrays クラスには、新たな static メソッドである、配列の内容を文字列に変換する toString メソッド、配列の配列の内容を文字列に変換するための deepToString メソッド、配列の配列を比較するための deepEquals メソッド、配列のハッシュコードを計算するための hashCode メソッド、配列の配列のハッシュコードを計算するための deepHashCode メソッドが追加されています。

9.11.7 Collections ユーティリティクラス

Collections クラスには、以下のメソッドが追加されています。

public static <T> **addAll(Collection⟨? super T⟩ c, T... a)**
: 指定されたコレクションに、指定された要素を追加します。可変長パラメータを使用して定義されていることに注意してください。

public static boolean **disjoint(Collection⟨?⟩ c1, Collection⟨?⟩ c2)**
: コレクション c1 および c2 間で共通の要素が 1 つもなければ true を返します。共通の要素が存在する場合には、false を返します。Collection インタフェースの一般契約に従っていない SortedSet インタフェースを実装しているコレクションや IdentityHashMap クラスのインスタンスが渡された場合には、結果の正しさは保証されません。

public static int **indexOfSubList(List⟨?⟩ source, List⟨?⟩ target)**
: source で指定されているリスト内の要素列内で、target で指定されているリストの要素列が一致する最初のインデックスを返します。一致しなければ、-1 が返されます。

public static int **frequency(Collection⟨?⟩ c, Object o)**
: コレクション c 内にある、オブジェクト o と同じオブジェクトの個数を返します。同じかどうかは、コレクション内の要素を e とすると、(o == null ? e == null : o.equals(e)) で検査されます。

public static int **lastIndexOfSubList(List⟨?⟩ source, List⟨?⟩ target)**
: source で指定されているリスト内の要素列内で、target で指定されているリストの要素列が一致する最後のインデックスを返します。一致しなければ、-1 が返されます。

public static <T> boolean **replaceAll(List⟨T⟩ list, T oldVal, T newVal)**
: リスト list 内の要素で、oldVal と同じ要素を、newVal で置き換えます。リスト内の要素を e とすると、oldVal と同じかどうかは、(oldVal == null ? e == null : oldVal.equals(e)) で検査されます。

public static void **rotate(List⟨?⟩ list, int distance)**
: リスト list 内の要素を、distance で指定された個数分だけ、循環的に移動させます。たとえば、list が、[t, a, n, k, s] の順序で要素を含んでいたとします。これに対して、Collections.rotate(list, 1) と行うと、[s, t, a, n, k] となります。負の値を指定すると、逆方向に循環しますので、Collections.rotate(list, -2) と行うと、[n, k, s, t, a]

となります。

public static void **swap(List⟨?⟩ list, int i, int j)**

リスト list 内のインデックスが i と j の位置の要素を入れ替えます。

9.11.8 コンカレントコレクション

リリース 5.0 からは、java.util.concurrent パッケージが追加されています。このパッケージは、マルチスレッド間で使用することを想定したコレクションが用意されています。本書では、その詳細については説明しませんが、簡単に概要だけを説明します。まず、以下の 2 つのインタフェースが用意されています。

BlockingQueue インタフェース

Queue インタフェースを拡張しており、キューが空の場合に要素を取り出そうとすると、要素が入れられるまでブロックしてしまう put メソッドや、キューが一杯の場合に要素を入れようとすると、キューに空きができるまでブロックする take メソッドが追加されています。また、キューの大きさに関連するメソッドも追加されています。

ConcurrentMap インタフェース

Map インタフェースを拡張しており、アトミックな操作を提供する putIfAbsent メソッド、remove メソッド、replace メソッドを追加しています。

Queue インタフェースを実装した具象クラスとしては、キューをリンクリストで実装し、スレッドセーフな ConcurrentLinkedQueue クラスが用意されています。BlockingQueue インタフェースを実装した具象クラスとしては、LinkedBlockingQueue クラス、ArrayBlockingQueue クラス、PriorityBlockingQueue クラス、DelayQueue クラス、SynchronousQueue クラスが用意されています。ConcurrentMap インタフェースを実装した具象クラスとしては、ConcurrentHashMap クラスが用意されています。

ConcurrentHashMap クラス

ハッシュテーブルに基づくコンカレントでスレッドセーフな ConcurrentMap インタフェースの実装です。マップからの取り出しでは、決してブロックしないので、高速です。また、Hashtable クラスのすべてのメソッドを実装していますので、単純に ConcurrentHashMap クラスで安全に書き換えることができます。

また、List インタフェースと Set インタフェースを実装した以下のクラスも用意されています。

CopyOnWriteArrayList クラス

配列による List インタフェースの実装です。add メソッド、set メソッド、remove メソッドなどのリストの内容を変更するような操作は、その配列の新たなコピーを作ることで実装されています。そのため、同期は必要としませんし、イテレータを使用してイテレートしている間に、リストへの変更が行われても、イテレータを取り出した時点での要素をイテレートしますので、決して

ConcurrentModificationExceptionをスローすることはありません。このリストは、変更の頻度よりも読み出しの頻度が高い場合に有用です。

CopyOnWriteArraySet クラス

配列によるSetインタフェースの実装です。setメソッドは、その配列の新たなコピーを作ることで実装されていますので、その処理時間は、他のSetインタフェースの実装クラスと異なり、$O(\mathrm{n})$になります。

9.12 その他のユーティリティ

9.12.1 Scanner クラス

文字列で表現された数値を入力するには、従来であれば、数値部分の文字列を取り出して、Integer.parseIntメソッドを使用して、int型の値に変換する必要がありました。より入力処理を簡単にするためにリリース5.0からは、java.utilパッケージに、Scannerクラスが用意されています。

Scannerクラスには、8個のコンストラクタが用意されており、様々な入力ソースに対応しています。Scannerクラスは、入力ソースから文字列を読み込んで、**トークン**（*token*）に分解します。 たとえば、標準入力からトークンに分解して、最初のトークンを数値として読み込むには、次のようにScannerクラスのインスタンスを生成して、nextIntメソッドを呼び出します。

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

デフォルトでは、空白を区切り文字として扱います。区切り文字の設定は、useDelimiterメソッドを使用して行います。指定方法としては、正規表現も使用できますので、複雑な区切り文字を指定することも可能です。さらに、ロケールをuseLocaleメソッドで指定することで、数値の3桁ごとの区切り文字を何にするかも指定することができます。詳しくは、Scannerクラスのドキュメンテーションを参照してください。

next*XXX*メソッドとして用意されているメソッドは、以下の通りです。

```
public String nextLine()
public boolean nextBoolean()
public byte nextByte()
public byte nextByte(int radix)
public short nextShort()
public short nextShort(int radix)
public int nextInt()
public int nextInt(int radix)
public long nextLong()
public long nextLong(int radix)
public float nextFloat()
public double nextDouble()
```

```
public BigInteger nextBigInteger()
public BigDecimal nextBigDecimal()
```

これらのメソッドは、指定された型に変換できない文字列の場合には、チェックされない例外、すなわち、実行時例外である InputMismatchException をスローします。また、次のトークンが無い場合には、NoSuchElementException がスローされます。

実行時例外ですので、特に try-catch を使用する必要はないのですが、指定した型に変換できない場合には、InputMismatchException がスローされます。そのため、次のトークンが変換可能かを検査するための hasNext*XXX* メソッドが、next*XXX* メソッドに対応して用意されています。

9.12.2 BitSet クラス

BitSet クラスには、リリース 1.4 で次のメソッドが追加されています。

public int **cardinality()**
: この BitSet 内で、値が true であるビットの数を返します。

public void **clear(int fromIndex, int toIndex)**
: fromIndex で指定されるインデックス位置のビットから、toIndex - 1 のインデックス位置までのビットに false を設定します。

public void **clear()**
: この BitSet 内のすべてのビットを false に設定します。

public void **flip(int bitIndex)**
: 指定されたインデックス位置のビットの値を反転させます。

public void **flip(int fromIndex, int toIndex)**
: fromIndex で指定されるインデックス位置のビットから、toIndex - 1 のインデックス位置までのビットを反転させます。

public boolean **intersects(BitSet set)**
: この BitSet 内のビットで、渡された set 内の true となっているすべてのビット位置に対応するビットが、true ならば true を返します。

public boolean **isEmpty()**
: この BitSet 内で、値が true であるビットが存在しなければ、true を返します。

public void **set(int bitIndex, boolean value)**
: 指定されたインデックス位置のビットを、指定された値に設定します。

public void **set(int fromIndex, int toIndex)**
: fromIndex で指定されるインデックス位置のビットから、toIndex - 1 のインデックス位置までのビットに true を設定します。

public void **set(int fromIndex, int toIndex, boolean value)**
: fromIndex で指定されるインデックス位置のビットから、toIndex - 1 のインデックス位置までのビットに、指定された値を設定します。

public BitSet **get(int fromIndex, int toIndex)**

fromIndex で指定されるインデックス位置のビットから、toIndex - 1 のインデックス位置までのビットから構成される新たな BitSet クラスのインスタンスを生成して返します。

public int **nextClearBit(int fromIndex)**

fromIndex で指定されたインデックス位置から始まって、値が false となっている最初のビットのインデックスを返します。

public int **nextSetBit(int fromIndex)**

fromIndex で指定されたインデックス位置から始まって、値が true となっている最初のビットのインデックスを返します。そのようなビットが存在しなければ、-1 が返されます。

9.12.3 UUID クラス

リリース 5.0 からは、**UUID**（*Universally Unique IDentifier*）を生成するための UUID クラスが java.util パッケージに追加されています。UUID の詳細については、Internet-Draft である *A UUID URN Namespace*[*15]、もしくは、ISO 定義「ISO/IEC 11578:1996」[*16]を参照してください。

9.12.4 Math クラスと StrictMath クラス

リリース 5.0 では、Math クラスと StrictMath クラスに、次の static メソッドが追加されています。

機能	値
`double log10(double x)`	$\log_{10} x$ (10 を基底とする対数)
`double cbrt(double x)`	x の立方根
`double ulp(double x)`	x の ulp(*Units in the Last Place*) の値
`float ulp(float x)`	x の ulp(*Units in the Last Place*) の値
`double signum(double x)`	x がゼロならゼロ、正なら 1.0、負なら-1.0 を返す
`float signum(float x)`	x がゼロならゼロ、正なら 1.0f、負なら-1.0f を返す
`double sinh(double x)`	x の双曲線正弦
`double cosh(double x)`	x の双曲線余弦
`double tanh(double x)`	x の双曲線正接
`double hypot(double x, double y)`	$\sqrt{x^2+y^2}$
`double expm1(double x)`	$e^x - 1$
`double log1p(double x)`	$(x+1)$ の自然対数

[*15] http://www.ietf.org/internet-drafts/draft-mealling-uuid-urn-05.txt

[*16] http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=2229

9.12.5 他のパッケージ

本書では、詳細は説明しませんが、以下のパッケージが追加されています。

`java.util.regex` **パッケージ**
: リリース 1.4 で追加されたパッケージであり、正規表現によるパターンマッチングを行うために、`MatchResult` インタフェース、`Pattern` クラス、`Matcher` クラスが提供されています。

`java.util.logging` **パッケージ**
: アプリケーションが記録を残して、後で調査を行ったり、レポートを出したりするのに使用する、ロギングのためのインタフェースとクラスを提供しています。

`java.util.prefs` **パッケージ**
: アプリケーションがユーザ固有の情報やコンフィグレーション情報を保存したり取り出したりするためのインタフェースとクラスを提供しています。

`java.util.concurrent` **パッケージ**
: 9.11.8 節で説明したコンカレントコレクションの他に、**タスク**（*task*）の様々なスケジューリングのための機構である**タスク・スケジューリング・フレームワーク**（*Task Scheduling Framework*）、スレッド間の同期のための新たな機構として**セマフォ**（*semaphore*）、**ミューテックス**（*mutex*）、**バリアー**（*barrier*）、**ラッチ**（*latch*）、**イクスチェンジャ**（*exchanger*）が提供されています。また、サブパッケージとして、新たなロックのためのインタフェースとクラスが提供されている `java.util.concurrent.locks` パッケージ、および、同期なしで値を更新するためのアトミック変数を提供している `java.util.concurrent.atomic` パッケージが含まれています。

9.13 システムプログラミング

9.13.1 System クラス

`java.lang.System` クラスには、以下のメソッドが追加されています。

`public static Channel` **inheritedChannel()** `throws IOException`
: Java 仮想マシンを起動した実体から継承したチャネルを返します。このメソッドは、単に `SelectorProvider.provider().inheritedChannel();` を呼び出してその値を返しています。Solaris などの Unix 環境で、inetd から Java 仮想マシンが起動された場合に、継承されたソケットを取得するために使用します。詳しくは、inetd から起動するためのガイド[*17] を参照してください。

`public static long` **nanoTime()**
: システムタイマーの現在の値を、ナノ秒単位として返します。この値は、時間の経過を測定するの

[*17] `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/inetd/launch-service.html`

に使用しますが、日時とは全く関係のない値です。ある固定された起点からの経過時間を返しますが、その起点が未来の場合には、負の値が返されます。このメソッドは、ナノ秒を単位とした値を返しますが、ナノ秒の正確さは保証してはいません。たとえば、次のように経過時間を測定します。

```
long startTime = System.nanoTime();
// ... 測定するコード ...
long elapsedTime = System.nanoTime() - startTime;
```

`public static String` **clearProperty(String key)**

指定されたキーのシステムプロパティを削除します。指定されたキーが存在していれば、その値を返します。存在していなければ、`null` を返します。

`public static java.util.Map<String,String>` **getenv()**

環境変数のマップを返します。システムが環境変数をサポートしていなければ、空のマップが返されます。返されたマップは、キーおよび値として `null` を含みませんので、`null` を検索しようとすると `NullPointerException` がスローされます。

なお、`System.getenv(String)` メソッドは、リリース 1.4 では推奨されないメソッドになっており、実装されていませんでした。しかし、リリース 5.0 でサポートされるメソッドに変更されています。

9.13.2 プロセッサ数

リリース 1.4 から `Runtime` クラスに `availableProcessors` メソッドが追加されています。このメソッドは、Java 仮想マシンが利用可能なプロセッサ数を返します。返される数は、Java 仮想マシンの実行中に変わることがありますので、プロセッサ数に依存するようなアルゴリズムを実行しているアプリケーションは定期的にプロセッサ数をチェックする必要があります。

9.13.3 `ProcessBuilder` クラス

リリース 1.4 までは、`Runtime.exec` メソッドを使用して、プロセスを生成していました。リリース 5.0 からは、より細かな指定が可能な `java.lang.ProcessBuilder` クラスが用意されています。コンストラクタとしては、次の 2 つが用意されています。

`public` **ProcessBuilder(List⟨String⟩ command)**

指定されたコマンドを持つインスタンスを生成します。引数で渡されたコマンドは、内部でコピーされないことに注意してください。つまり、インスタンスを生成後に、`command` の内容を変更すると、起動されるプロセスのコマンドあるいは引数を変更することになります。

`public` **ProcessBuilder(String... command)**

指定されたコマンドを持つインスタンスを生成します。

プロセスの生成は、次の `start` メソッドを用いて行います。

`public Process` **start()** `throws IOException`

プロセスの生成を行います。プロセスの生成に使用されるコマンド、ワーキングディレクトリ、お

よび環境変数は、それぞれ、command() メソッド、directory() メソッド、および environment() メソッドで返される値が使用されます。

デフォルトのワーキングディレクトリおよび環境変数でプロセスを起動するのは、次のように簡単です。

```
Process p = new ProcessBuilder("myCommand", "myArg").start();
```

ProcessBuilder クラスが提供している他のメソッドは、以下の通りです。

public ProcessBuilder **command(List⟨String⟩ command)**
引数で指定されたコマンドを設定します。引数で渡されたリストは、内部でコピーされないことに注意してください。戻り値としては、このメソッドが呼び出されたインスタンスが返されます。

public ProcessBuilder **command(String... command)**
引数で指定されたコマンドを設定します。戻り値としては、このメソッドが呼び出されたインスタンスが返されます。

public List<String> **command()**
設定されているコマンドを返します。返されるリストは、コピーされたものではないことに注意してください。つまり、その返されたリストへの変更は、プロセスの起動に使用されるコマンドを変更することになります。

public Map<String,String> **environment()**
起動されるプロセスの環境変数をマップとして返します。返されたマップは、コピーされたものではないことに注意してください。つまり、そのマップへの変更は、プロセスの起動に使用される環境変数の変更となります。ただし、start メソッドでプロセスを起動後に、このマップを変更しても、すでに起動されたプロセスの環境変数には反映されません。環境変数をサポートしていない場合には、空のマップが返されます。

public File **directory()**
起動されるプロセスのワーキングディレクトリを返します。返された値が null であれば、現在の Java プロセスのワーキングディレクトリであることを示します。現在の Java プロセスのワーキングディレクトリは、システムプロパティの"user.dir"で設定されているディレクトリです。

public ProcessBuilder **directory(File directory)**
起動されるプロセスのワーキングディレクトリを設定します。null を指定すると、現在の Java プロセスのワーキングディレクトリを指定することになります。戻り値としては、このメソッドが呼び出されたインスタンスが返されます。

public boolean **redirectErrorStream()**
起動されるプロセスの標準エラーを標準出力に出力されるように設定されているか否かを返します。設定されていれば true を返し、そうでなければ、false を返します。

public ProcessBuilder **redirectErrorStream(boolean redirectErrorStream)**
起動されるプロセスの標準エラーを標準出力に出力するかを設定します。true なら、標準エラーを標準出力に出力するように設定します。戻り値としては、このメソッドが呼び出されたインスタ

ンスが返されます。

次のコードは、プロセスを起動するためのメソッドの実装例です。

```
public Process invoke(List<String> command,
        File workingDirectory,
        Map<String,String> env,
        boolean redirectErrorStream) throws IOException  {
    ProcessBuilder pb = new ProcessBuilder(command);
    pb.directory(workingDirectory);
    pb.environment().putAll(env);
    pb.redirectErrorStream(redirectErrorStream);
    return pb.start();
}
```

command は、起動するプログラム名と引数を指定します。workingDirectory は、起動するプロセスのワーキングディレクトリを指定します。env は、環境変数を書き換えるべき環境変数を指定しています。環境変数を書き換えるために、environment メソッドで返されたマップを直接書き換えていることに注意してください。

9.13.4 java.lang の新たなサブパッケージ

第 7 章で説明した java.lang.annotation パッケージに加えて、本書の範疇外ですが、リリース 5.0 では、以下のサブパッケージが java.lang パッケージに追加されています。

java.lang.instrument パッケージ
クラスをロードする際にメソッドのバイトコードを変更するための仕組みを提供しています。

java.lang.management パッケージ
Java 仮想マシンをモニタリングして管理するための管理 API を提供しています。

9.14 国際化とローカリゼーション

9.14.1 Currency クラス

リリース 1.4 からは、通貨に関する情報を表す java.util.Currency クラスが追加されています。Currency クラスのインスタンスの取得は、すべて static ファクトリーメソッドを使用して行います。

public static Currency getInstance(String currencyCode)
指定された通貨コード（ISO 4217 コード[*18] ）に対応するインスタンスを返します。

public static Currency getInstance(Locale locale)
ロケールで指定された国の通貨に対応するインスタンスを返します。

[*18] http://www.bsi-global.com/

static ファクトリーメソッドで取得したインスタンスに対して、以下のメソッドを呼び出して、通貨に関する情報を取得することができます。

`public String` **getCurrencyCode()**
: ISO4 217 コードを返します。

`public String` **getSymbol()**
: デフォルトのロケールにおける通貨記号を返します。

`public String` **getSymbol(Locale locale)**
: 指定されたロケールにおける通貨記号を返します。たとえば、米国ドルの場合、ロケールが US ならば、`"$"`ですが、他のロケールに対しては、`"US$"`となります。

`public int` **getDefaultFractionDigits()**
: 金額を表現する場合に、小数点以下が何桁かを返します。たとえば、ユーロであれば 2 であり、日本円であれば 0 が返されます。

次のプログラムは、デフォルトのロケールが表している国の通貨に関する情報を出力します。

```
import java.util.Currency;
import java.util.Locale;

public class CurrencyTest {
    public static void main(String[] args) {
        Currency cr = Currency.getInstance(Locale.getDefault());
        System.out.printf("Currency Code = %s, Symbol = %s, "
                          + "DefaultFractionDigits = %d%n",
            cr.getCurrencyCode(),
            cr.getSymbol(),
            cr.getDefaultFractionDigits());
    }
}
```

日本語環境で実行すると、次の出力結果となります。

```
Currency Code = JPY, Symbol = ￥, DefaultFractionDigits = 0
```

9.15 -Xlint コンパイルオプション

非標準ですが、リリース 5.0 からは javac コマンドのオプションとして、`-Xlint` オプションが追加されています。非標準ですので、将来のリリースで変更になる可能性がありますが、簡単に紹介しておきます。

-Xlint
: すべての推奨される警告メッセージを有効にします。

-Xlint:none
言語仕様で必須とされている警告メッセージ以外を無効にします。

-Xlint:-*xxx*
xxx で指定されている警告メッセージを無効にします。xxx で指定する警告メッセージ名は、以下の-Xlint:xxx の xxx 部分です。

-Xlint:unchecked
言語仕様で必須とされている無検査警告メッセージに対する詳細を表示します。

-Xlint:path
存在しないパス（クラスパス、ソースパス）ディレクトリに対する警告メッセージを表示します。

-Xlint:serial
シリアライズ可能なクラスに serialVersionUID フィールドが定義されていなければ警告メッセージを表示します。

-Xlint:finally
正常に終了することができない finally 節に対する警告メッセージを表示します。

-Xlint:fallthrough
switch 文の case ラベルで指定された処理が break 文で終わることなく、次の case ラベルへ落ちていく場合に、警告メッセージを表示します。

-Xlint:unchecked は、第 1 章で説明しました。次のコードは、-Xlint:finally オプションと-Xlint:fallthrough をテストするためのプログラムラムです。

```
class XlintTest {

    void foo() {
        try {
        } finally {
            throw new AssertionError();
        }
    }

    void bar(int x) {
        switch (x) {
            case 0: System.out.println("0");
            case 1: System.out.println("1");
                    break;
            case 2:
            case 3:
                    System.out.println("2 or 3");
                    break;
        }
    }
}
```

foo メソッドでは、finally 節内で、AssertionError をスローすることで、正常に終了させていません。また、bar メソッドでは、case 0:の処理で break 文で抜けるようになっていません。このプログラムを、-Xlint オプションでコンパイルすると、次の結果となります。

```
D:\Tiger\example\javac>javac -Xlint XLintTest.java
XLintTest.java:7: 警告: [finally] finally 節が正常に完了できません。
        }
        ^
XLintTest.java:13: 警告: [fallthrough] case に fall-through する可能性があります。
            case 1: System.out.println("1");
            ^
警告 2 個
```

もちろん、-Xlint オプションではなく、-Xlint:finally オプションや-Xlint:fallthrough オプションで個別に指定することもできます。あるいは、次のように一部を無効にすることも可能です。

```
D:\Tiger\example\javac>javac -Xlint -Xlint:-fallthrough XLintTest.java
XLintTest.java:7: 警告: [finally] finally 節が正常に完了できません。
        }
        ^
警告 1 個
```

-Xlint オプションで、すべての警告を一旦有効として指定した後、-Xlint:-fallthrough オプションにより、fallthrough だけを無効にしています。

参考文献

[Arnold00] Arnold, Ken, James Gosling, David Holmes. *The Java™ Programming Language, Third Edition*. Addison-Wesley, Boston, 2000. ISBN: 0201704331.
柴田芳樹 訳『プログラミング言語 Java 第 3 版』、ピアソン・エデュケーション、ISBN: 4894713438。

[Arnold05] Arnold, Ken, James Gosling, David Holmes. *The Java™ Programming Language, Fourth Edition*. Addison-Wesley, Boston, 2005. ISBN: 0321349806.
柴田芳樹 訳『プログラミング言語 Java 第 4 版』、ピアソン・エデュケーション、ISBN: 978-4-89471-716-9。

[Bloch01] Bloch, Joshua. *Effective Java™ Programming Language Guide*. Addison-Wesley, Boston, 2001. ISBN: 0201310058.
柴田芳樹 訳『Effectiv Java™ プログラミング言語ガイド』、ピアソン・エデュケーション、ISBN: 4894714361。

[Bloch05] Bloch, Joshua, Neal Gafter. *Java Puzzlers : Traps, Pitfalls, and Corner Cases*. Addison-Wesley, Boston, 2005. ISBN: 032133678X.
柴田芳樹 訳『Java™ Puzzlers – 罠、落とし穴、コーナーケース』、ピアソン・エデュケーション、ISBN: 4894716895。

[Bloch08] Bloch, Joshua. *Effective Java™, Second Edition*. Addison-Wesley, Boston, 2008. ISBN: 0321356683.
柴田芳樹 訳『Effectiv Java™ 第 2 版』、ピアソン・エデュケーション、ISBN: 978-4-89471-499-1。

[Bracha04a] Bracha, Gilad. *Generics in the Java Programming Language*. June, 2004. <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>

[Bracha04b] Bracha, Gilad, David Ungar. *Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages*. Proceeding of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, October 2004. <http://www.bracha.org/mirrors.pdf>

[Horstmann04a] Horstmann, Cay S., Gary Cornell. *Core Java 2, Seventh Edition, Volume I – Fundamentals*. Prentice Hall, 2004. ISBN: 0131482025.

[Horstmann04b] Horstmann, Cay S., Gary Cornell. *Core Java 2, Seventh Edition, Volume II – Advanced Features*. Prentice Hall, 2004. ISBN: 0131118269.

[J2SE-APIs] *Java™ 2 Platform, Standard Edition, v 5.0 API Specification*. Sun Microsystems. October 2004. <http://java.sun.com/j2se/1.5.0/docs/api/overview-summary.html>.

[JLS00] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha. *The Java™ Language Specification, Second Edition*, Addison-Wesley, Boston, 2000. ISBN: 0201310082.
村上雅章 訳『Java™言語仕様第 2 版』、ピアソン・エデュケーション、ISBN: 4894713063。

[JLS05] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha. *The Java™ Language Specification, Third Edition*, Addison-Wesley, Boston, 2005. ISBN: 0321246780.
村上雅章 訳『Java™言語仕様第 3 版』、ピアソン・エデュケーション、ISBN: 4-89471-715-8。

[Lea00] Lea, Doug. *Concurrent Programming in Java™: Design Principles and Patterns, Second Edition*, Addison-Wesley, Boston, 2000. ISBN: 0201310090.
松野良蔵 監訳『Java スレッドプログラミング』、翔泳社、ISBN: 4881359185。

[Linden04] Linden, Peter van der. *Just Java 2 Sixth Edition*, Prentice Hall, 2004. ISBN: 0131482114.

[McLaughlin04] McLaughlin, Brett, David Flanagan. *Java 1.5 Tiger: A Developer's Notebook*, O Reilly, 2004. ISBN: 0596007388.
菅野 良二訳『Java 5.0 Tiger』、オライリー・ジャパン、ISBN: 4873112141。

[Torgersen04] Torgersen, Mads, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, Neal Gafter. *Adding Wildcards to the Java Programming Language*, Proceedings of SAC 2004. <http://www.bracha.org/wildcards.pdf>.

[Warren02] Warren, Henry S. Jr. *Hacker's Delight*, Addison-Wesley, 2002. ISBN: 0201914654.
滝沢徹、赤池英夫、藤波順久、鈴木貢、葛毅、玉井浩 訳『ハッカーのたのしみ—本物のプログラマはいかにして問題を解くか』、エスアイビーアクセス、ISBN: 4434046683。

索引

■ J

■ L

■ M

■ T

■ 著者紹介

柴田　芳樹（しばた　よしき）：
1959 年生まれ。九州工業大学情報工学科で情報工学を学び、1984 年同大学大学院で情報工学修士課程を修了し、以来、様々なソフトウェア開発に従事。
ゼロックス社のパロアルト研究所を含め、5 年間米国に駐在してソフトウェア開発に従事。
現在は、ソフトウェア開発、教育、コンサルテーション等に従事している。

訳書:『Objective-C 明解プログラミング』『プログラミング言語 Go フレーズブック』『Android SDK 開発クックブック』『プログラミング原論』『Effective Java 第 2 版』『プログラミング言語 Java 第 4 版』『Java Puzzlers 罠、落とし穴、コーナーケース』『Google Web Toolkit ソリューション』『Java リアルタイム仕様』（以上、ピアソン桐原）、『アプレンティスシップ・パターン』（オライリー・ジャパン）

著書:『Java 2 Standard Edition 5.0 Tiger 拡張された言語仕様について』（ピアソン桐原）、『プログラマー “まだまだ” 現役続行』『ソフトウェア開発の名著を読む【第二版】』（以上、技術評論社）