```
26      }
27      if(!title.equals(b.title)) {
28        return false;
29      }
30      return true;
31    }
32    public int compareTo(Book o) {
33      return this.publishDate.compareTo(o.publishDate);
34    }
35    public Book clone() {
36      Book b = new Book();
37      b.title = this.title;
38      b.comment = this.comment;
39      b.publishDate = (Date) this.publishDate.clone();
40      return b;
41    }
42    :     // フィールド宣言、getter/setterの宣言は省略
43  }
```


第5章

さまざまな種類のクラス

Javaでは一般的なクラスはもちろん、さまざまな種類の「クラスのようなもの」を利用して便利に開発を進めることができます。たとえばインタフェースがそれにあたりますが、数あるものの1つに過ぎません。

この章では、新しい種類のクラスについて学ぶとともに、クラスや型について理解を深めましょう。

5.1 型安全という価値

5.1.1 もし型がなかったら

この章では、さまざまな種類のクラスや型について紹介していくよ。でもその前に、「そもそもなぜ型というものがあるのか？　型にはどんなメリットがあるのか？」を考えてみないかい。

えっ…。型を使うのが当たり前だと思っていたから、急に聞かれると…。どうしてなのかしら？

Java では変数を用いるとき、必ず**型**(type)を指定します。変数には、その指定した型の情報(数値やインスタンス)しか格納することはできません。int 型の変数には int 型の整数だけが、String 型の変数には文字列だけが格納できることはご存じのとおりです。つまり、**型とは「格納するデータに制約をかけるしくみ」**と言うことができます。

ではなぜ、そんな「わざわざ不自由になる道具」を使う必要があるのでしょうか？　もし Java に型のしくみがなくて、「変数には数字でも文字列でも勇者でもなんでも自由に代入できる」としたら、そのほうが便利だと感じませんか？

そっか。それなら、いいかげんに何でもデータを放り込めるんだよね。確かに、型なんてないほうが便利じゃないか！

湊の楽天的な発言のせいかしら。私はなんだか危うさというか、怖さを感じるんだけど…。

もし、あなたが朝香さんのように「怖い」という感覚を覚えるようであれば、型の役割とメリットに多少なりとも気付いています。

実際にその感覚を試すために、Object 型(＝何でも入る型)だけを使い、擬似的に実現した「型のしくみがない Java プログラム」を見てみましょう(リスト 5-1)。

リスト 5-1　Object 型を利用した型のしくみがない Java プログラム

Main.java

```
1  public class Main {
2    // printsメソッド
3    // 第1引数の文字列を第2引数の回数だけ表示します
4    // 第1引数には文字列情報を、第2引数には整数を指定してください
5    public static void prints(Object a, Object b) {
6      for(int i = 0; i < (Integer)b; i++) {
7        System.out.println(a);
8      }
9    }
10   public static void main(String[] args) {
11     Object s = "こんにちは";
12     s = new Hero();
13     Object n = 1;
14     prints(s, n);
15   }
16 }
```

s には勇者も文字列も格納可能

注意深く呼び出す必要がある

このプログラムは問題なく動作します。確かに 11 ～ 12 行目では湊くんの言うように、「いろいろな種類の値を型を意識せず変数に入れられる」という便利さはあるかもしれません。

一方、14 行目では、prints() メソッドを呼び出すときには**かなり神経を使う必要がある**ことに気付きませんか？　うっかりすると次ページの図 5-1 にある(b)や(c)のような間違いをしてしまいます。

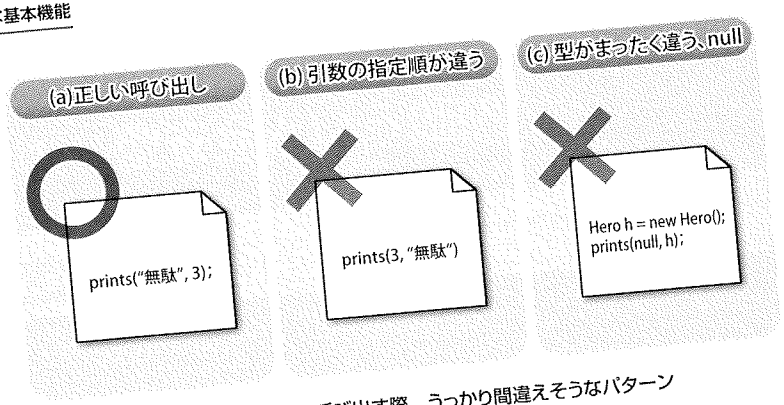

図 5-1　prints() を呼び出す際、うっかり間違えそうなパターン

(b)や(c)のような**単純なミスにもかかわらず、コンパイル時にはエラーが出ないためミスに気付きません**。実際にプログラムを動かし、処理が問題の箇所にさしかかったときに実行時エラーが起きて初めてミスに気付きます。

たとえば「絶対止めてはならない超重要システム」のあるメソッド内で上記のような単純なミスをしたとしよう。本番稼働から３か月過ぎて初めてそのメソッドが動作→緊急停止→大事故になるのはいやだよね。

5.1.2　型を用いるメリット

一方、型を適切に使った prints() メソッドの宣言を見てみましょう。

```
public static void prints(String msg, int num) { … }
```

この宣言では「第１引数は String、第２引数は int しか受け付けない」という制約がかけられています。だからこそ、もし図 5-1 の(b)や(c)のようなミスをしたら、コンパイル時のエラーメッセージですぐミスに気付きます。

つまり型というしくみは**「変数に予期しない種類の情報が入ってしまうことを未然に防ぐための安全装置」**の役割を果たしていることがわかります。

安全性のためにわざと private や protected で制限をかける「カプセル化」と、ちょっと似てるわね。

型によって担保される安全性のことを**型安全**(type safe)といいます。型安全のメリットを享受するために、**変数には可能な限り厳密な型を指定しましょう**。なぜなら厳密な型に絞り込むほど、意図しない不正な値が入る余地を小さくできるからです。

コンパイラにデバッグさせる

コンパイルエラーは重大事故を未然に防ぐ安全装置。
型安全を活用して、どんどんコンパイラにミスを探させよう。

Java には型安全のメリットをより享受するために、「ジェネリクス」や「列挙型」というしくみが準備されています。次節以降で順に見ていきましょう。

静的型付けと動的型付け

Java のように、コンパイル時に型を決定し、型安全を積極活用していく言語が採用している方針を**静的型付け**(static typing)といいます。一方、変数などに型の指定をせず、コンパイル時に型チェックを行わない**動的型付け**(dynamic typing)の考え方もスクリプト言語などで広く使われています。

動的型付けの場合、期待しないオブジェクトが代入され実行時エラーが出たり、予期しない変換がなされたりなどの懸念がありますが、手軽である、柔軟に変数を利用できるといった長所もあります。第 14 章で紹介する自動テストなどを組み合わせることで、リスクを抑えながらメリットを享受することもできます。

5.2 ジェネリクス

5.2.1 ひと昔前の ArrayList

第3章では ArrayList を初めとするコレクションクラスの使い方を学びました。実はこのコレクション、昔（Java1.4 以前）は少し異なる記述をしなければなりませんでした。

```
// Java5以降の場合
ArrayList<String> list = new ArrayList<String>();
// Java5より古い場合　(Java5以降でも可能だが非推奨)
ArrayList list = new ArrayList();
```

昔は <String> という部分の指定をしなくてよかったんですね！

でも、それがどういうことを意味するのか、「型安全」のことを思い出しながら考えてごらん。

ArrayList<String> という型は「文字列が入る ArrayList 型」という意味です。よって、文字列以外の要素を格納することはできません。一方、**ただの ArrayList 型では格納可能な要素の型を特に制限していません**。そのため、文字列でも勇者でも何でも型を問わず格納できてしまいます（図 5-2）。

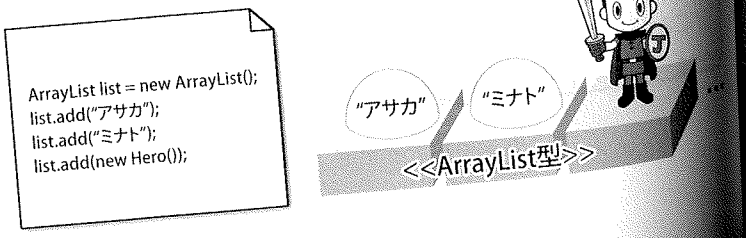

図 5-2
さまざまな種類の要素を格納できる ArrayList

型安全のメリットを理解した皆さんであれば、**古い ArrayList の使い方を「怖い」と感じる**ことができるはずです。たとえば、文字列の要素しか格納されていないはずのリストに、いつのまにか数値が格納されてしまい、それが原因で予期しない実行時エラーが発生するかもしれません。

コレクションクラスの型安全
型安全のために、コレクションクラスは常に＜～＞を付けて利用する。

5.2.2 ジェネリクス

API にはたくさんクラスがありますけど、どうしてコレクションクラスだけ＜～＞が使えるんですか？

実はコレクションクラスは、普通のクラスとは少し違った「特殊な型」として宣言されているんだ。

API に準備された String 型や、私たちが作った Hero 型などでは、＜～＞記法は使えません。たとえば String<Character> や Hero<Sword> のような記述はコンパイルエラーになります。

ArrayList や HashMap にだけ＜～＞記法が許されているのは、それらが普通のクラスとしてではなく、**ジェネリクス**（generics）といわれる特別なしくみを使って定義されているからです。

ジェネリクス
ジェネリクスを使って宣言されたクラスは、＜～＞記法を利用できる。

ちなみに、ジェネリクスという用語に代えて「総称型」や「テンプレート」という言葉が使われることもあるよ。

ジェネリクスが利用されているクラスは、API リファレンスにおいても＜～＞という表記が示されますので、あるクラスについて＜～＞記法が可能か不可能かを簡単に判断することができます（図 5-3）。

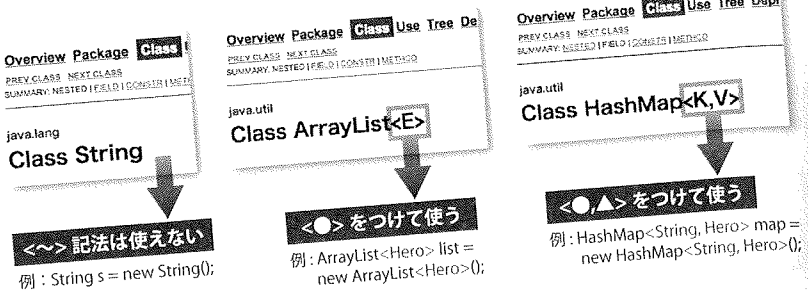

図 5-3　API リファレンスにおけるジェネリクス利用の表記

図 5-3 中の ArrayList <E> という表記は「ArrayList を実際に使うときには、**＜～＞記法で型を 1 つ指定できます**」という意味でしかなく、E というアルファベット自体には特に意味はありません。Z でも ZZ でも何でもよかったのですが、おそらく ArrayList クラスを作った人は要素（element）の頭文字を取って E を選んだのでしょう。

5.2.3　ジェネリクス活用の前知識

ひょっとして、この「ジェネリクス」というしくみは、私たちが自分のクラスを作るときにも利用できるんですか？

そうなんだ。では簡単な例で紹介しよう。

ジェネリクスはコレクションのためだけのしくみではありません。Java においてクラスを作るときに誰でも自由に使える汎用的な文法です。そこで、今回は「インスタンスを 1 つだけ格納できる」という機能を持った独自のコレクションクラス（Pocket）を作りながら、ジェネリクスの使い方を学んでいきましょう。

Pocket クラスの仕様（1）

- どんな型のインスタンスでも格納できる。
- 格納するための put() メソッド、取り出すための get() メソッドがある。

まず、あえてジェネリクスを使わず実現したものが次のリスト 5-2 です。

リスト 5-2　ジェネリクスを使わず実現した Pocket クラス（ver.1）

Pocket.java

```
1  public class Pocket {
2      private Object data;     // 格納用の変数
3      public void put(Object d) { this.data = d; }
4      public Object get() { return this.data; }
5  }
```

利用時に何型のインスタンスを入れるかわからないから、Object 型を使って宣言してありますね。

Java 1.4 までは、ArrayList や HashMap もこんな感じで定義してあったんだよ。

これでも一応、先述の仕様（1）を満たしています。しかし、Pocket を利用する側のプログラム（次ページのリスト 5-3）を見ると、Pocket がいかに不便なクラスであるかがわかります。

リスト 5-3　Pocket クラス(ver.1)を利用するプログラム

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Pocket p = new Pocket();
4      p.put("1192");
5      String s = (String) p.get();
6      System.out.println(s);
7    }
8  }
```

文字列を格納

取り出すときにキャストが必要

先輩に「危険だから滅多なことでは使うな」とたしなめられた「キャスト」が使ってありますね。

格納した文字列を単に取り出しているだけですが、5 行目では String 型へのキャストが必要です。なぜなら、Pocket クラス内部の Object 型変数 data に値が格納された瞬間、その情報がもともと String インスタンスであったことが忘れ去られ、以降 Object の一種の何かとしてザックリと捉えられるからです。

しかし、「なかば無理矢理に型変換を試みる」キャストという道具は可能な限り利用を避けるべきです。たとえば、何らかの原因で文字列以外のインスタンスを格納するようなミスを犯した場合も、コンパイル時ではなく、実行時に ClassCastException が起きて初めて気付くことになるからです(図 5-4)。

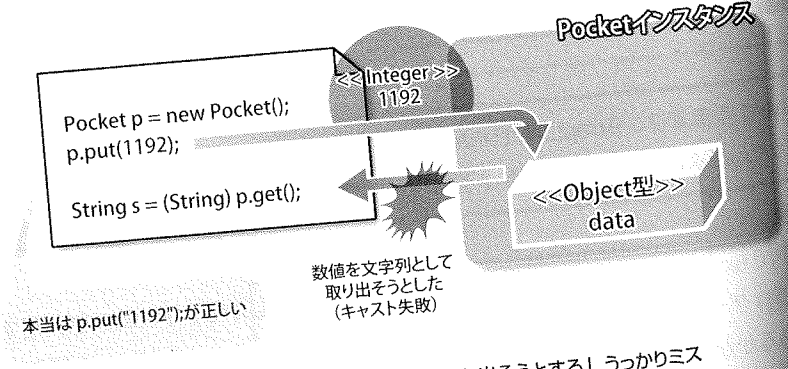

図 5-4　「数字を入れて文字列として取り出そうとする」うっかりミス

5.2.4　ジェネリクスの活用

それでは次に、ジェネリクスを活用して定義された Pocket を見てみましょう(リスト 5-4)。

リスト 5-4　ジェネリクスを使った Pocket クラス(ver.2)

Pocket.java

```
1  public class Pocket<E> {
2    private E data;
3    public void put(E d) { this.data = d; }
4    public E get() { return this.data; }
5  }
```

仮型引数を伴うクラス宣言

仮型引数 E を利用したメンバ宣言

不思議なコードですね…。でも、よく見たらリスト 5-2 と基本構造は同じよね？

1 行目の「Pocket<E>」の E は**仮型引数**(formal type parameter)と呼ばれ、クラス内のフィールドやメソッドの定義に広く利用できます。しかし、**E を使って定義されたこのクラスは、それ自体ではまだ未完成品です。なぜなら、E の部分が実際にどのような型になるかは、この時点ではまだ決まっていないからです。**

E が実際にどのような型になるかは、Pocket クラスを利用する際に**実型引数**(actual type parameter)を指定して決定します。

たとえば、プログラム中で「Pocket<String> s;」という変数宣言をすると、コンパイラは裏でこっそり次のような Pocket<String> クラスを生成して利用します。

リスト 5-5　「Pocket<E>」から裏で作られる「Pocket<String> クラス」

Pocket.java

```
1  public class Pocket<String> {
```