第13章

多態性

前章では抽象クラスやインタフェースを用いて、現実世界にある
あいまいな物事を Java のコードとして表現できることを学びました。
私たちは日常生活でも、無意識に物事を
「あいまいに捉える」「あいまいに使う」ことにより、
さまざまなメリットを享受しています。
本章で学ぶ最後のオブジェクト指向の 3 大要素の 1 つ「多態性」とは、
Java 仮想世界でも物事を「あいまいに捉える」ための機能です。

CONTENTS

13.1 多態性とは

13.1.1 開発をラクにする多態性

多態性（polymorphism）はオブジェクト指向プログラミングを支える３大機能の１つで、多様性やポリモーフィズムと呼ばれることもあります。

また、難しそうな名前だなぁ…。

だけど、この多態性をマスターすると、湊くんの RPG 作りは何倍もラクになるんだよ。

そもそもオブジェクト指向は、「ラクしてよいものを実現する」ためのものでした。特に、多態性を上手に活用すると、とても効率よく楽しく開発できると聞けば、がんばって覚えようという気になりませんか？

カプセル化や継承と同じく、多態性の学習にも「コツ」があります。実は、多態性を定義や文法規則から学び始めると、大変難しく感じます。しかし、イメージを理解してから文法や定義を学べばそれほど難しいものではありません。

この章では、多くのイメージ図を示しながら、よりやさしくマスターできるように解説を進めていきます。ぜひ頭の中にイメージを広げながら、気楽に読み進めてください。

13.1.2 多態性のイメージ

多態性の定義は章の最後に紹介します。また、「多態」の意味についても、今は考えないでください。現時点では、次のような理解で十分です。

多態性のあいまいなイメージ

「あるものを、あえてザックリ捉える」ことで、さまざまなメリットを享受しようという機能。

この章を学ぶにあたっての大事なキーワードは「ザックリ」です。ザックリ捉えるとは、たとえば以下のような考え方です。

厳密に言えば SuperHero なんだけど、まぁザックリいえば Hero だよね。
厳密に言えば GreatWizard なんだけど、まぁザックリいえば Wizard だよね。
厳密に言えば Slime なんだけど、まぁザックリいえば Monster だよね。

このような捉え方をして、さまざまなメリットを享受しようというのが多態性という機能なのです。

13.1.3 ザックリ捉えるメリット

ザックリ捉える、というのはなんとなくわかりましたけど、そんなことでメリットなんてあるんですか？

もちろんさ。ザックリなしでは、人間は生きていけないよ。

ザックリ捉えることによるメリットは、私たちの日常生活にも多く見ることができます。たとえば、レンタカーを借りて車を運転するときのことを考えてみましょう。厳密にいえば初めて乗る車であるにも関わらず、多くの人は問題なく運転できます。なぜ、初めての車なのに運転できるのかとドライバーに聞けば、おそらく次のような答えが返ってくるでしょう。

「まぁハンドルは同じだし、右ペダルがアクセル、左がブレーキ。
細かいところはあれこれ違うけど、まぁザックリみれば、どの車も同じだよ。」

この人は高級車や軽自動車、ライトバン、さらには来年発売の新車（現時点では未知の車）でも問題なく運転できるでしょう。

「そんなの当たり前じゃないか」と思うかもしれませんが、もし運転するのがロボットだとしたら、こうはいきません。ロボットに内蔵される運転制御プログラムは、たとえば以下のように無数の細かい設定が必要と考えられます。

> もし 1997 年式のプリウスなら、ハンドルはシートから○ cm の高さにあり、それを 10 度回すとタイヤが○度曲がり…。
> もし 1998 年式のインプレッサなら、…（以下、延々と続く）

ロボットは、それぞれの車種について細かい条件を完全に把握している必要があり、かつ「把握してない車」は操作できません。

明らかに人間のほうが「ラク」をして同じ成果が得られていますが、それは、**車の厳密な車種についてはあまり考えていない＝ザックリと「車」としか捉えていない**からです。

車に限らず、私たち人間は、世の中にあるさまざまな複雑なものをザックリ捉えることによって、厳密には違うものも「似たようなもの」として上手に利用しています。この「私たちが現実世界でラクするために用いている方法」をプログラムでも実現する機能こそ、オブジェクト指向３大機能における最後の１つ、「**多態性**」なのです。

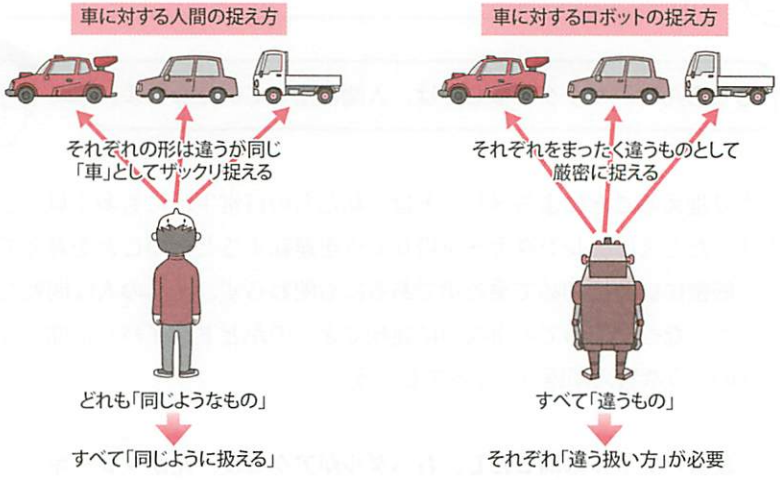


図 13-1　ザックリ捉える人間、厳密に捉えるロボット

13.2 ザックリ捉える方法

13.2.1 ザックリ捉えるための文法

確かに、Java の世界でも「ザックリ」を実践できたら便利そうね。

ザックリ捉えるには、どんな文法やキーワードを使えばいいんですか？

実はザックリ捉えるための特別な文法はないんだよ。

今まで学習したオブジェクト指向の機能には、特有のキーワードが登場しました。カプセル化といえば「private」や「public」、継承といえば「extends」がすぐに思い浮かぶでしょう。

そこで多態性も**何か専用の文法があって、それを書けば利用できると思い込んでしまう**人がいますが、**多態性には専用の文法はありません**。

実は、今まで何度も用いてきた「代入の文法」を使えば、ザックリ捉えることができるようになっています。

それではさっそく、SuperHero クラスを用いて解説していきましょう。まず前提として SuperHero は親として Hero クラスを、さらにその親として Character クラスを持っています（p.437 の図 11-14）。

通常、SuperHero のインスタンスを生成して利用するには、次のような文を記述するのでしたね。

```
SuperHero h = new SuperHero();
```

このときの状態をイメージで表すと、次ページの図 13-2 のようになります。ここで、箱に書かれた「<<SuperHero です >>」という文字は、変数 h の型を表しています。

SuperHero 型の変数に SuperHero のインスタンスが入っているわけですね。通常、変数にはその型と同じ型の内容が入りますから、これはごく当たり前の状態です。

図 13-2　SuperHero 型の変数 h に格納された SuperHero インスタンス

次に、SuperHero を「ザックリ Character として捉える」書き方です。

```
Character c = new SuperHero();
```

第 8 章では「new をするときは左辺と右辺に同じ型を書く」と紹介しました（8.4.2 項）が、このように左辺と右辺の型を変えることも実は可能です。今回の場合、左辺の型が Character になっただけですが、イメージ図は図 13-3 のように変わります。

箱の中身のインスタンスは正真正銘の SuperHero ですが、箱の表面には「Character です」と書かれています。よって、以後この変数 c を利用するときは、**本当は SuperHero インスタンスなのですが、あくまで Character として捉えて利用する**ことになります。

図 13-3　Character 型の変数 c に格納された SuperHero インスタンス

このように、多態性を活用するためには「箱の型」と「中身の型」という異なる 2 つの型が関係してきます。そして**あるインスタンスをどのように捉えるかは、どの型の変数に代入するか（箱の型）で決まります。**

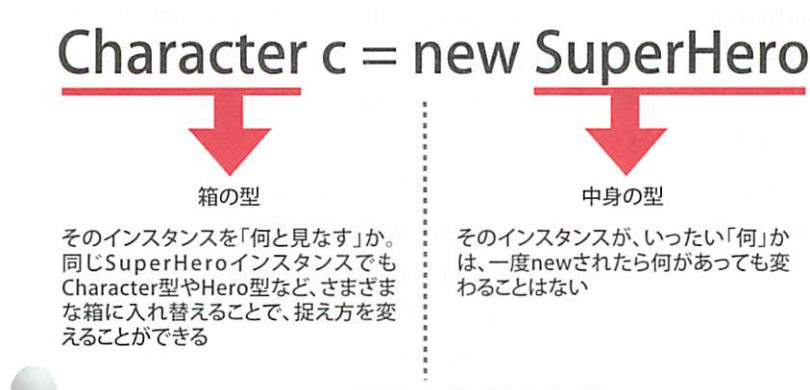


図 13-4　箱の型と中身の型

13.2.2 できる代入、できない代入

new をするときの左辺と右辺は、別に同じ型でなくてもいいんですね。

そうだよ。ただし、どんな型でもいいわけではないんだ。

次のコードを見てください。1 行目はエラーになりませんが、2 ～ 4 行目はすべてエラーになります。

```
Character c = new SuperHero();     // OK!
Sword s = new Hero();              // エラー
Flower f = new Fish();             // エラー
Phone p = new Coffee();            // エラー
```

第 1 章の 1.3.1 項で解説したように、代入式は基本的に「左辺の型と右辺の型が異なる場合はエラー」になります。たとえば「String str = 1;」がエラーになるのは当然ですね。しかし、この原則でいえば 1 行目のコードも「左辺は Character 型、

右辺はSuperHero型」なのでエラーになるはずです。なぜ1行目だけがエラーにならず、特例として許されているのでしょうか？

その「代入が許される判断基準」は絵を描いてみればわかります。先ほどの図13-3をもう一度見てください。

箱には「Characterです」と書いてあり、中身にはSuperHeroが入っています。そしてこの**絵の内容は（厳密ではありませんが）嘘ではありません**。スーパーヒーローもキャラクターの一種（SuperHero is-a Character）ですから、「キャラクターが入っています」という箱に入っていても別に矛盾はないのです。

Javaではこのように、**絵に描いてみて嘘にならないインスタンスの代入は許されます**。

しかし、次のような代入はエラーになります。

```
float f = new Hero();
Item i = new Hero();
SuperHero sh = new Hero();
```

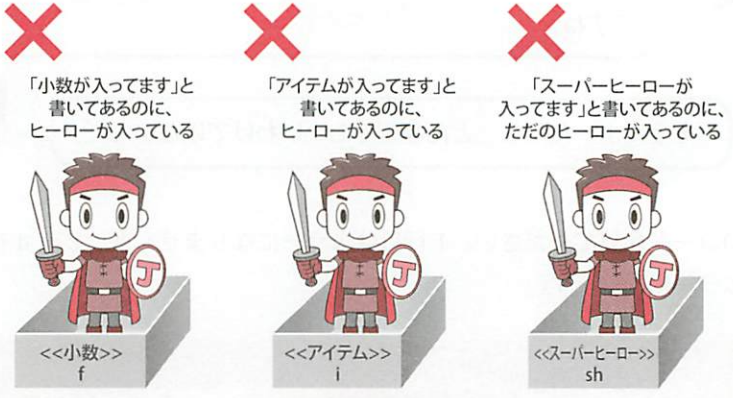


図13-5　代入ができない例

絵で描いてみればわかるように、これらはいずれも嘘になってしまうからです。

よく「子クラスのインスタンスは親クラスの型に代入可能」などという定義を丸暗記しようとして混乱する人がいる。だが、「イメージ図を描いてみる」ほうが忘れにくいしオススメだよ。

やっぱりオブジェクト指向ってイメージが大事なんですね。

13.2.3 継承のもう 1 つの役割

前節で「絵に描いて嘘がないならば代入可能」なことはわかりました。ただし 1 つだけ注意点があります。

そもそも「絵に嘘が含まれるか」を判断するには、「～は～の一種である」という前提知識が必要です。たとえば私たちは「ヒーローの中で特に選ばれた者がスーパーヒーローである（つまり、スーパーヒーローはヒーローの一種である）」という前提知識があるからこそ、図 13-3 が「嘘ではない」と判断できました。

私たち人間には「常識」がありますので、「魔術師が生き物の一種であること」や「剣が武器の一種であること」を知っています。しかし、Java には「何が何の一種か」という一般常識は備わっていません。

そこで Java は、extends や implements を用いた継承関係にあるクラス同士について、「片方のクラスは、他方のクラスの一種（is-a の関係）」であると考えます。言い換えれば、**extends や implements はプログラマが「is-a の関係」を Java に伝える手段**でもあるのです。

そのため、いくら私たち人間が考えた一般常識では is-a の関係であっても、Java の継承関係で 2 つのクラスがつながっていなければ代入はできません。

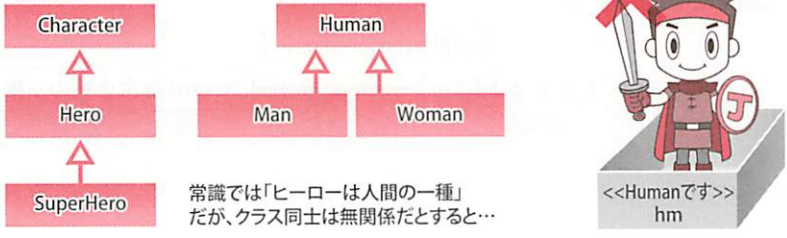


図 13-6　extends で is-a 関係が結ばれていないと代入できない

第 11 章の最後でも触れたけど、「is-a の関係（抽象的・具体的の関係）を Java に知らせる」というのも継承の重要な役割なんだ。

だから「仮にラクをできるとしても、is-a の関係でないなら継承を使ってはならない」と教えられたわけですね（11.4.2 項）。

13.2.4 箱の型に抽象クラスを使う

ここで、あなたが命あるあらゆるものの親として、新たに Life インタフェースを作ったと考えましょう。Character も Life を実装するように変更したと仮定します。第 12 章で「抽象クラスやインタフェースはインスタンス化できない」と学びました。ですからインタフェース Life が定義されている場合、「new Life()」とすることはできません。

しかし、第 8 章で学んだ「クラス定義によって可能になる 2 つのこと（8.3.1 項）」を思い出してください。インタフェースである Life はインスタンス生成のためには利用できませんが、変数の型（表面に「Life が入っています」と書かれた箱）として使うことはできます。たとえばリスト 13-1 のように、Life 型の変数に Wizard インスタンスを入れることができます。

リスト 13-1

Life.java

```
1  public interface Life { … }
```

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Life lf = new Wizard();
4    }
5  }
```

Wizard は生き物の一種

抽象クラスやインタフェースの型

抽象クラスやインタフェースからインスタンスを生み出すことはできないが、型を利用することは可能。

13.3 ザックリ捉えたものに命令を送る

13.3.1 捉え方の違いは使い方の違い

前節では、どのようにして「ザックリ捉えるか」を紹介しました。この節では、「あるインスタンスをザックリ捉えるのと、厳密に捉えるのでは、その利用にどのような違いが出てくるか」を見ていきます。

多態性の説明を一休みして、1つたとえ話をしましょう。

> この紙切れを原始人に渡すと「絵」と捉えるでしょう。そして原始人は、これを見て楽しむことはあっても、何かに使えるとは思わないでしょう。
>
> 一方、同じものを湊くんに渡すと、それはもう大喜びします。絵の人物が「福沢諭吉」であり、その紙切れが厳密には「紙幣」だということを知っているからです。彼はその紙切れを眺めることもできますが、どこかの店に持ち込み、何か商品と交換してしまうでしょう。

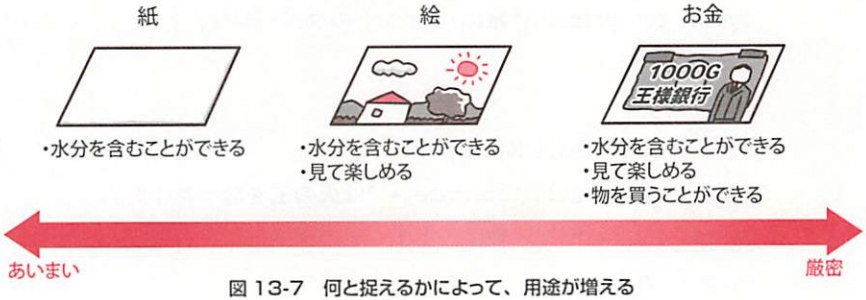


図 13-7　何と捉えるかによって、用途が増える

このたとえ話が示唆しているのは、「まったく同一である1つの存在に対して、複数の異なる捉え方ができる」ということと、「何かを利用する人は、それを何と捉えているかによって、利用方法が変わる」ということです。あいまいで抽象的なほど用途は限定され、具体的に捉えるほど用途が増えていきます。

13.3.2 呼び出せるメソッドの変化

この「捉え方が変わると、利用方法が変わる」という現実世界の現象は、実はJava仮想世界でもちゃんと再現されています。p.437の図11-14におけるCharacterとWizardを使って、このことを解説していきましょう。まず、次の2つのクラスのソースコードを見てください。

リスト13-2

Character.java

```
public abstract class Character {
  String name;
  int hp;
  public abstract void attack(Matango m);
  public void run() { … }
}
```

Wizard.java

```
public class Wizard extends Character {
  int mp;
  public void attack(Matango m) {
    System.out.println(this.name + "の攻撃！");
    System.out.println("敵に3ポイントのダメージ");
    m.hp -= 3;
  }
  public void fireball(Matango m) {
    System.out.println(this.name + "は火の玉を放った！");
    System.out.println("敵に20ポイントのダメージ");
    m.hp -= 20;
    this.mp -= 5;
  }
}
```

Wizard は魔法使いとして attack() や fireball() のメソッドを持っていますので、インスタンス化すれば attack させたり fireball を使わせたりできます。

リスト 13-3

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Wizard w = new Wizard();
4      Matango m = new Matango();
5      w.name = "アサカ";
6      w.attack(m);
7      w.fireball(m);
8    }
9  }
```

さて、13.2 節で学んだように Wizard は Character の一種なので、Character 型変数に代入することが可能です。しかし、いざ Character 型に代入して fireball を呼び出そうとするとエラーが起きます。

リスト 13-4

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Wizard w = new Wizard();
4      Character c = w;
5      Matango m = new Matango();
6      c.name = "アサカ";
7      c.attack(m);
8      c.fireball(m);
9    }
10 }
```

Character 型の箱に代入

この行でエラーが発生する

え？ いくら Character 型の箱に入っているとはいえ、
中身は正真正銘の魔法使いさんのはずなのに…。

魔法使いであれば fireball() を呼び出せるはずなのに、
どうしてエラーになるんだろう？

それはね、「本当は魔法使いなんだけど、呼び出す側が
魔法使いと思ってない」から呼び出せないんだよ。

Character 型の変数 c に格納されているとはいえ、箱の中身のインスタンスは正真正銘の Wizard インスタンスです。そして Wizard ならば fireball が使えるはずです。それにも関わらず、なぜコンパイルエラーになるのでしょうか？

この章の「ザックリ捉える文法（13.2.1 項）」で説明したとおり、Character 型の変数に代入するということは、中身のインスタンスを「（Hero だか Wizard だかわからないけど）なにかのキャラクター」程度にザックリ捉えるということにほかなりません。よって、**箱の中身が Wizard であることを忘れてしまいます。**

あいまいな型の箱へのインスタンスの代入

インスタンスをあいまいに捉えることとなり、「厳密には何型のインスタンスだったか」がわからなくなってしまう。

リスト 13-4 の 3 行目では確かに魔法使いを生み出しています。しかし 4 行目の代入を行った瞬間、私たちは箱 c の中身が「Hero なのか Wizard なのか、はたまた別の職業のクラスなのか」がわからなくなってしまいます。**確実に言えることは、「この箱に入っているのは、キャラクターの一種であること」だけ**です。

そう考えると、attack() が呼び出せて fireball() が呼び出せなかった理由にも説明がつきます。

■ attack() が呼び出せた理由

箱の中身が Hero でも Wizard でも、Character の一種である限り attack() メソッドは継承して持っているはずだから（どんなキャラクターでも最低限、攻撃はできるはずだから）。

■ fireball() が呼び出せなかった理由

箱の中身が Hero の場合など、fireball() メソッドを持っている職業とは限らないから（キャラクターであれば必ず火の玉を放てるとは限らないから）。

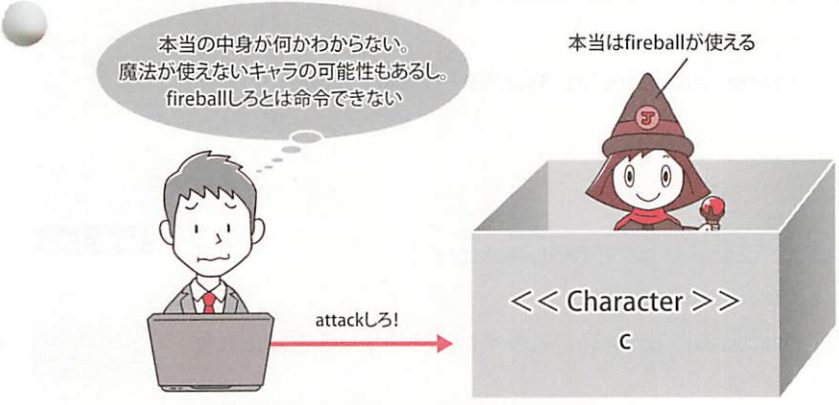


図 13-8　本当は Wizard だが、箱の外部からはわからない

箱の中の魔法使いさんが火の玉を放てなくなっちゃったわけではないんですね。

彼自身は「火の玉を放って」とお願いされれば、いつでも放つことができるんだ。ただ、私たちが彼を「魔法使い」と認識しないと「お願いできない」（しようと思わない）だけなんだよ。

私たちがこのインスタンスを「Character」と捉えている限り（＝ Character 型の変数に入っている限り）、私たちはこのインスタンスに「少なくとも Character ならできる最低限のこと」しか命令できません。箱の中身のインスタンスがどんなに多くのメソッドを持っていたとしても、Character として持つメソッドだけしか外部からは**呼び出すことができない**のです。

13.3.3 メソッドを呼び出せた場合に動く処理

前項では「どの箱に入れるかによって、呼べるメソッドが変わる」ことを学習しました。では次に、「もしメソッドが呼べたとしたら、その動きはどうなるか」についてモンスター関連クラスを題材に実験してみましょう。

リスト 13-5

Monster.java

```
1  public abstract class Monster {
2    public void run() {
3      System.out.println("モンスターは逃げ出した。");
4    }
5  }
```

Slime.java

```
1  public class Slime extends Monster {
2    public void run() {
3      System.out.println("スライムはササッと逃げ出した。");
4    }
5  }
```

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Slime s = new Slime(); Monster m = new Slime();
4      s.run(); m.run();
5    }
6  }
```

さぁ問題だ。実行結果として、画面には何と表示されるかな？

Slime 型の run() と Monster 型の run() を呼び出しているってことは…。

まず「スライムはサササっと逃げ出した。」、次に「モンスターは逃げ出した。」かな。

では、実行して正解を見てみよう。

実行結果

```
スライムはサササっと逃げ出した。   ← s.run() の結果
スライムはサササっと逃げ出した。   ← m.run() の結果
```

実行結果の２行目に注目してください。Monster 型の変数 m の run() メソッドを呼び出しているのに「モンスターは逃げ出した。」という表示になりません。

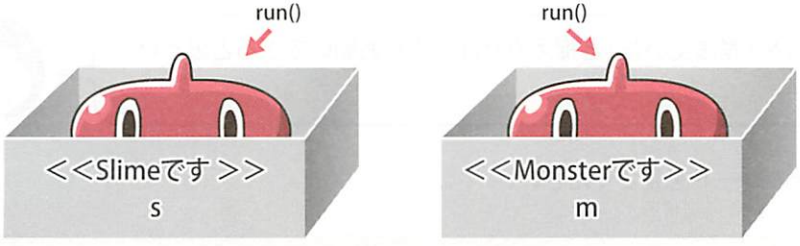


図 13-9　逃げ出すのは「実体」

変数 s と変数 m とは、箱の表面に書かれた「○○が入っています」という表記に違いがあるものの、両方とも中身はあくまでスライムです。ですから「逃げろ！」という命令が届きさえすれば、当然スライムが逃げます。つまり、Slime の run() が動作するのです。

このように、実際に動くメソッドの中身はインスタンスの型（中身の型）によって決まります。それがどんな型の箱に入っているかは関係ありません。

最後に「箱の型」と「中身の型」について、もう一度まとめておきましょう。

「箱の型」と「中身の型」

[箱の型]　どのメソッドを「呼べるか」を決定する。

[中身の型] メソッドが呼ばれたら、「どう動くか」を決定する。

13.4 捉え方を変更する方法

13.4.1 捉え方を途中で変える

図 13-8（13.3.2 項）を見て思ったんだけど、Character 型の箱に入れられた魔法使いに対しては、もう 2 度と fireball() を呼び出せなくなっちゃうのかなぁ？

彼を「魔法使いだ」と捉え直せば、またお願いできると思うけど…。

次のコードを見てください。

```
Character c = new Wizard();
```

このとき、変数 c に対して fireball() メソッドを呼べなくなることはすでに学びました。しかし、「中身が本当は Wizard だとわかっているし、どうしてもこのインスタンスに fireball を使わせたい」という場合がまれにあります。

fireball を使えるようにするためには、変数 c の中身を「Wizard であると捉え直す」必要があります。そのためにはインスタンスを Wizard 型変数に代入すればよいと想像がつくでしょう。しかし、次のコードはエラーになります。

```
1  Character c = new Wizard ();
2  Wizard w = c;
```

惜しい！ エラーになる

なぜコンパイラは、この代入を許さないのでしょうか？ 私たち開発者はコード 1 行目の経緯を知っています。そのため、「もともと Wizard インスタンスとして生み出して、それを Character と見なしたものの、再び元の Wizard と見直

して何が悪い！」と感じます。

しかしコンパイラは基本的にプログラムを１行ずつ解釈・翻訳しようとしますので、今回のエラーも２行目だけを見て出しているのです。

２行目だけを見ると論理上は Character 型の変数 c に入っている中身は、Hero や Thief の可能性もあります。万が一、変数 c の中身が Hero なのに、それを w に代入してしまったら、「Wizard 型の変数に Hero インスタンスが入っている」という嘘の構図になってしまいます。

ですからコンパイラは、このような失敗する可能性のある代入について、「中に入っているものが常に Wizard とは限らないから代入できない」と拒否します。

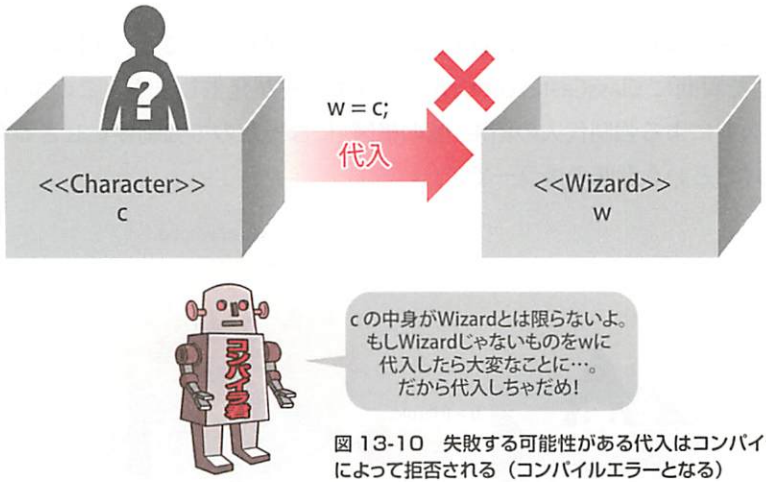


図 13-10　失敗する可能性がある代入はコンパイラによって拒否される（コンパイルエラーとなる）

それでも変数 c の中身を強制的に「Wizard として捉え直したい」場合には、次のように書きます。

```
1  Character c = new Wizard();
2  Wizard w = (Wizard) c;
```

いいから黙って Wizard と見なせ！

第２章で登場した「キャスト演算子」の再登場です。この演算子は強制的な型変換をコンパイラに対して明示的に指示する、とても強力な演算子です。こうすればコンパイラは文句を言わず、コンパイルを通してくれます。特に、今回のような「あいまいな型に入っている中身を厳密な型に代入する」キャストはダウンキャスト（down cast）といわれ、失敗の危険が伴います。

13.4.2 キャストの失敗

ダウンキャストは「代入が失敗する可能性」を懸念してエラーを出すコンパイラに対し、「代入をしても矛盾のある状態にはならないから、黙って代入しろ」と頭ごなしに型変換を実行させる命令です。前節のコードの例ではうまく動きますが、次のようなケースではどうでしょうか？

```
1  Character c = new Wizard();
2  Hero h = (Hero) c;
```

いいから黙って Hero と見なせ！

このソースのコンパイルは成功します。しかし動作させると、実際に代入しようとした瞬間に ClassCastException というエラーが発生します。このエラーは、「キャストによる強制代入の結果『嘘の構図』になったので強制停止せざるを得なくなった」という意味のエラーです。

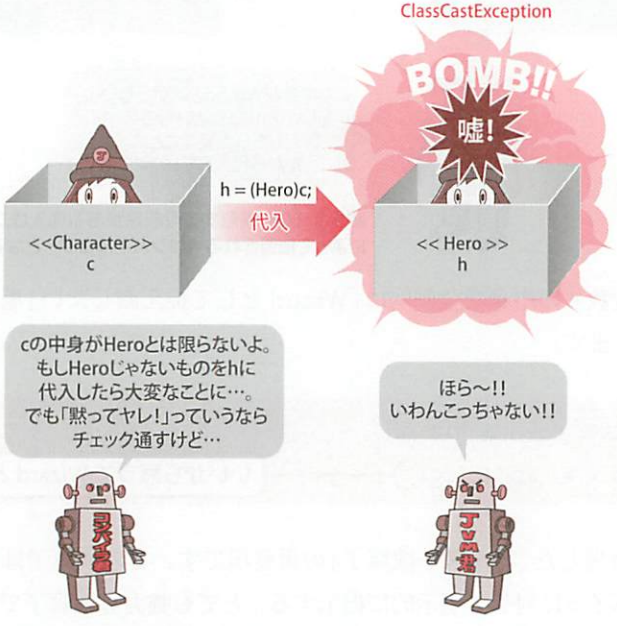


図 13-11　誤ったキャストにより ClassCastException が発生する

13.4.3 インスタンスを代入可能かチェックする

ダウンキャストによる ClassCastException を確実に回避するには、キャストする前に「キャストしても大丈夫かどうか」をチェックします。Java にはそのための instanceof 演算子が用意されています。

安全にキャストできるかを判定する

変数 instanceof 型名

※変数を型名の箱に代入可能ならば true が返る。

instanceof 演算子は、「指定の型に代入しても絵として嘘にならないか」を判定してくれます。この演算子を利用して、たとえば次のようなコードを記述することもできます。

```
1 if (c instanceof SuperHero) {
2
3   SuperHero h = (SuperHero) c;
4   h.fly();
5 }
```

もしcの中身がSuperHeroと見なして大丈夫なら…

「SuperHero」と見なせ！

13.5 多態性のメリット

13.5.1 ザックリ捉えることによるメリット

ザックリ捉えると、呼び出せるメソッドが減ってしまうんですよね？それって不便じゃないですか？

呼び出せるメソッドが減っても、ザックリ捉えることでそれに勝るメリットがあるんだ。

13.2 節では、「多態性を用いて、インスタンスをザックリ捉える方法」を学びました。そして 13.3 節では、それによって「呼び出せるメソッドが減る」ことも学びました。

これだけを見ると、「ザックリ捉えることは、利用できるメソッドが減るだけで、まったくメリットがない」ように感じられますが必ずしもそうではありません。この節では、多態性を用いてザックリ捉えることによるメリットを具体的に Java のコードで紹介していきます。

13.5.2 同一視して配列を利用する

5 人のキャラクター(Hero が 2 人、Thief が 1 人、Wizard が 2 人)が旅をするゲームを考えてみましょう。この 5 人は 1 つのパーティ(冒険のためのチーム)です。彼らが宿屋に泊まり、全員の HP を 50 ずつ回復するプログラムを書く場合、次のようなものになるでしょう。

リスト 13-6

Main.java

```
1  public class Main {
```

```
2    public static void main(String[] args) {
3      Hero h1 = new Hero();
4      Hero h2 = new Hero();
5      Thief t1 = new Thief();
6      Wizard w1 = new Wizard();
7      Wizard w2 = new Wizard();
8      // 冒険開始！
9      // まず宿屋に泊まる
10     h1.setHp(h1.getHp() + 50);
11     h2.setHp(h2.getHp() + 50);
12     t1.setHp(t1.getHp() + 50);
13     w1.setHp(w1.getHp() + 50);
14     w2.setHp(w2.getHp() + 50);
15   }
16 }
```

※このコードの前提
- Hero や Wizard、Thief は、抽象クラス Character を継承している。
- Character は name と hp フィールドおよびその getter／setter、attack() と run() メソッドを持つ。

このプログラムの宿泊処理（10 行目から 14 行目）には 2 つの課題があります。

【コードに重複が多い】

「～ .setHp(～ .getHp() + 50)」という同じ処理が何度も登場するため、記述がめんどうです。変数名の取り違えも発生するかもしれません。

【将来的に多くの修正が必要】

パーティの人数が増えた場合、宿泊処理に行を追加しなければなりません。また、インスタンス変数名が変更になった場合も、コードに修正が必要です。

しかし、多態性と配列を上手に組み合わせれば、この問題は解決します。次のリスト 13-7 を見てください。

リスト 13-7

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Character[] c = new Character[5];
4      c[0] = new Hero();
5      c[1] = new Hero();
6      c[2] = new Thief();
7      c[3] = new Wizard();
8      c[4] = new Wizard();
9      // 宿屋に泊まる
10     for (Character ch : c) {
11       ch.setHp(ch.getHp() + 50);
12     }
13   }
14 }
```

1 名ずつ順に取り出し

HP を 50 回復する

ポイントは 3 行目で Character 配列を使っている点です。従来のように 5 人のインスタンスを厳密に Hero や Thief として扱おうとする限り、それらを一括しては扱えません。しかし、それぞれを Character だとザックリ見なせば「どれもキャラクター」ですので、5 つのインスタンスを Character 配列にまとめ、ループを回して一括で処理することも可能になります。

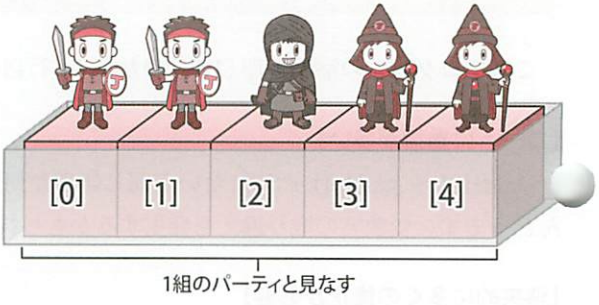


図 13-12　さまざまな職業のキャラクターたちも、すべて Character 配列に格納して一括で処理できる

13.5.3 同一視してザックリとした引数を受け取る

多態性の活用法、もう１つ思いつきました！ ずっと気になっていたことが、これで一気に解決ですよ！

湊くんがずっと気にしていたのは、「勇者や魔法使いの attack() メソッドが、必ず次のような宣言になっていた」ということです（以下は Matango の例です）。

```
public void attack(Matango m) {
  ⋮
}
```

「お化けキノコしか攻撃できないゲーム」なんてありえないから、今まで Hero は次のようにしていたんですよ。

リスト 13-8

Hero.java

```
1  public class Hero extends Character {
2    public void attack(Matango m) {        ← お化けキノコ攻撃用
3      System.out.println(this.name + "の攻撃！");
4      System.out.println("敵に10ポイントのダメージをあたえた！");
5      m.hp -= 10;
6    }
7    public void attack(Goblin g) {         ← ゴブリン攻撃用
8      System.out.println(this.name + "の攻撃！");
9      System.out.println("敵に10ポイントのダメージをあたえた！");
10     g.hp -= 10;
11   }
12   // 以下スライム用など続く
13 }
```

この方法はうまくいきますが、コードの重複が多くメンテナンスが大変です。また、将来新たなモンスターが増えるたびに attack() メソッドも増やさなければなりません。そこで、attack() メソッドを次のように修正しましょう。

リスト 13-9

Hero.java

```
1  public class Hero extends Character {
2    public void attack(Monster m) {
3      System.out.println(this.name + "の攻撃！");
4      System.out.println("敵に10ポイントのダメージをあたえた！");
5      m.hp -= 10;
6    }
7  }
```

モンスター攻撃用

2 行目の attack() メソッドの引数に注目してください。「攻撃する相手は、ザックリみれば何らかのモンスターであれば何でもいい」という表明です。このような attack() メソッドであれば、Monster クラスを継承している Slime や Goblin、そして将来登場するモンスターたちも攻撃することができます。

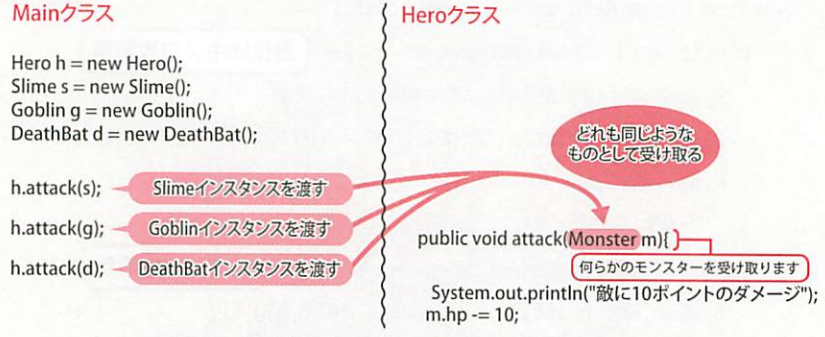


図 13-13　異なるインスタンスを引数で同一視して受け取る

13.5.4 ザックリ利用しても、ちゃんと動く

なるほどなぁ。厳密には違う物を同一視することで、同じ配列に入れたり、同じ引数として処理したりできちゃうんだ。

そうだよ。でも「多態性の本当のすごさ」は、ただ単に処理をまとめられるという単純なものだけじゃないんだ。

多態性の真価は、これまで学んだ次の２つのことを組み合わせたときに現れます。

1. ザックリ捉えてまとめて扱う

   13.5.2 項（配列でまとめて扱う）や 13.5.3 項（引数でまとめて扱う）で紹介したように、厳密には異なるインスタンスをまとめて扱うことができます。

2. メソッドの動作は中身の型に従う

   13.3 節の最後に学んだように「インスタンスは何型の箱に入っていようと、自身の型のメソッドが動作する」という原則があります。

リスト 13-10

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Monster[] monsters = new Monster[3];
4      monsters[0] = new Slime();
5      monsters[1] = new Goblin();
6      monsters[2] = new DeathBat();
7      for (Monster m : monsters) {
8        m.run();
9      }
10   }
11 }
```

同じ指示を繰り返す

実行結果

```
スライムは、体をうねらせて逃げ出した。
ゴブリンは、腕をふって逃げ出した。
地獄コウモリは、羽ばたいて逃げ出した。
```

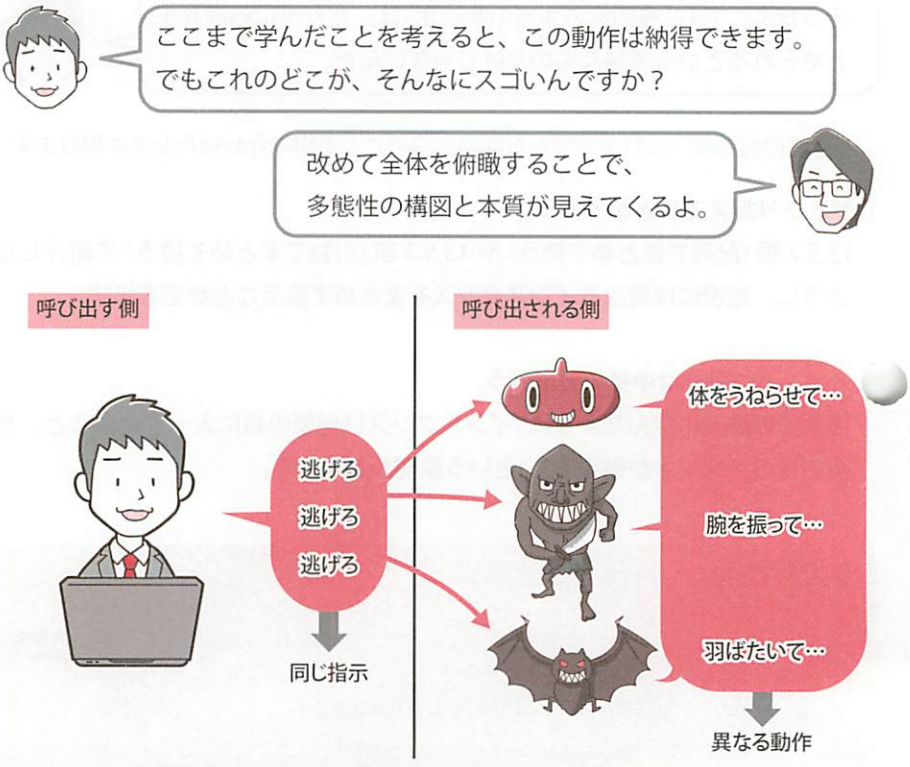


図 13-14　指示する側はいいかげん。動く側は自分のやり方で動く

コードの７～９行目で、指示を出す側（メソッドを呼び出す側）は、それぞれのモンスターに対して同じように「とにかく逃げろ」と、いいかげんな指示を繰り返しているだけです。

一方、モンスターたちは「逃げろ」と言われたら、きちんと**独自の方法で**逃げます。「どう逃げるか」については自分で理解していて、その方法（自分のクラスに定義された run() メソッドの内容）を使って逃げるのです。

このように、**呼び出し側は相手を同一視し、同じように呼び出す**のに、**呼び出される側は、きちんと自分に決められた動きをする**（同じ呼び出し方なのに、多数の異なる状態を生み出すことがある）という特性から「多態性」という名前が付けられています。

13.6 第13章のまとめ

この章では、次のようなことを学びました。

インスタンスをあいまいに捉える

継承により is-a の関係が成立しているなら、インスタンスを親クラス型の変数に代入することができる。

- 親クラス型の変数に代入することは、あいまいに捉えること。

「箱の型」と「中身の型」の役割

- どのメンバを利用できるかは、箱の型（対象をどう捉えているか）で決まる。
- メンバがどう動くかは、中身の型（対象が何であるか）で決まる。

捉え方の変更

- キャスト演算子を用いれば、厳密な型への強制代入ができる。
- 不正な代入が行われた場合、ClassCastException が発生する。

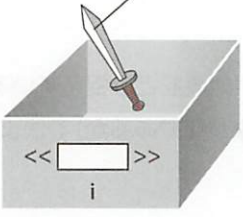
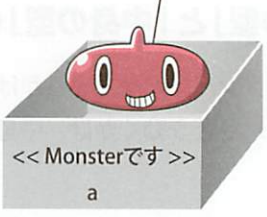
多態性

- 厳密には異なる複数のインスタンスを同一視して、親クラス型の配列にまとめて格納できる。
- 同様に、親クラス型の引数や戻り値を利用して、厳密には異なる対象をまとめて処理できる。
- 同一視して取り扱っても、個々のインスタンスは各クラスにおける定義に従い、異なる動作を行う。

13.7 練習問題

練習 13-1

次の図中の四角に入る適切なクラス名を考えてください。

	(1)	(2)
コード	Item i = new Sword();	□ a = new □ ();
イメージ図	□ インスタンス  	□ インスタンス 
解説文	□ を生成したが、 ザックリと □ と と見なす。	Slimeを生成したが、 ザックリと □ と見なす。

練習 13-2

次のようにクラスが宣言されています。

A.java

```
1  public final class A extends Y {
2    public void a() { System.out.print("Aa"); }
3    public void b() { System.out.print("Ab"); }
4    public void c() { System.out.print("Ac"); }
5  }
```

B.java

```
1  public class B extends Y {
2    public void a() { System.out.print("Ba"); }
3    public void b() { System.out.print("Bb"); }
4    public void c() { System.out.print("Bc"); }
5  }
```

X.java

```
1  public interface X { void a(); }
```

Y.java

```
1  public abstract class Y implements X {
2    public abstract void a();
3    public abstract void b();
4  }
```

このとき、次の問いに答えてください。

①「X obj = new A();」として A インスタンスを生成した後、変数 obj に対して呼ぶことができるメソッドを、a()、b()、c() の中からすべて挙げてください。

②「Y y1 = new A(); Y y2 = new B();」として A と B のインスタンスを生成した後、「y1.a(); y2.a();」を実行した場合に画面に表示される内容を答えてください。

練習 13-3

練習 13-2 で用いた A クラスや B クラスのインスタンスをそれぞれ 1 つずつ生み出し、要素数 2 からなる単一の配列に格納するとします。格納した後には配列の中身をループで順に取り出し、それぞれのインスタンスの b() メソッドを呼ぶ必要があります。以上の前提に基づき、次の問いに答えてください。

①配列変数の型としては何を用いるべきか答えてください。
②問題文に記述された内容のプログラムを作成してください。

13.8 練習問題の解答

練習 13-1 の解答

（1）（左上から順に）Sword、Item、Sword、Item

（2）（左上から順に）Monster、Slime、Slime、Monster

練習 13-2 の解答

① a() メソッド

② AaBa

練習 13-3 の解答

① Y[] 型

②以下のリストを参照。

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Y[] array = new Y[2];
4      array[0] = new A();
5      array[1] = new B();
6      for (Y y : array) {
7        y.b();
8      }
9    }
10 }
```