第11章

継承

たくさんのクラスを作るうちに、
「以前作ったクラスにとてもよく似ているが、一部だけ違うクラス」
を作る必要に迫られ、めんどうに感じることも増えてきます。
この章では、このような課題を解決してくれるオブジェクト指向の
花形機能、「継承」の基本を学びましょう。

CONTENTS

11.1 継承の基礎

11.1.1 似かよったクラスの開発

Java で大きなプログラムを書き始めると、**以前作ったクラスと似かよったクラスを作る**必要に迫られることがあります。

「ほとんど同じだけどフィールドが 2 つほど多い」あるいは「ほとんど同じだけどメソッドが 1 つ多い」という具合です。そのようなクラスは、どうすれば効率よく作れるでしょうか。

例として勇者クラス（Hero クラス）を取り上げて考えてみましょう。この章では理解しやすさのために、リスト 11-1 のような単純な Hero クラスから始めます。

リスト 11-1　「戦う」と「逃げる」しかできない Hero クラス

Hero.java

```
1  public class Hero {
2    private String name = "ミナト";
3    private int hp = 100;
4    // 戦う
5    public void attack(Matango m) {
6      System.out.println(this.name + "の攻撃！");
7      m.hp -= 5;
8      System.out.println("5ポイントのダメージをあたえた！");
9    }
10   // 逃げる
11   public void run() {
12     System.out.println(this.name + "は逃げ出した！");
13   }
14 }
```

このHeroは冒険するにつれ進化してゆき、以下のような能力を持ったSuperHeroという職業になれるとしましょう。

スーパーヒーローはfly()で空を飛ぶことができ、land()で着地できる。
ヒーローができるすべての動作は、スーパーヒーローもできる。

では、リスト11-1のHeroクラスを元に、SuperHeroクラスを開発してみましょう。

簡単よ。Heroのコードをコピー＆ペーストし、クラス名をSuperHeroに変更して、それにfly()とland()のメソッドを足せば、あっという間にできあがり！名付けて「コピペ解決法」よ！

朝香さんはHeroクラスを元に、次のようなリスト11-2のコードを書きました。うまく動作するでしょうか？

リスト11-2　朝香さんが作成したSuperHeroクラス

SuperHero.java

```
1  public class SuperHero {
2    private String name = "ミナト";
3    private int hp = 100;
4    private boolean flying;
5    // 戦う
6    public void attack(Matango m) {
7      System.out.println(this.name + "の攻撃！");
8      m.hp -= 5;
9      System.out.println("5ポイントのダメージをあたえた！");
10   }
11   // 逃げる
12   public void run() {
13     System.out.println(this.getName() + "は逃げ出した！");
14   }
```

クラス名を書き換えた

flying フィールドを追加

```
15    // 飛ぶ
16    public void fly() {
17      this.flying = true;
18      System.out.println("飛び上がった！");
19    }
20    // 着地する
21    public void land() {
22      this.flying = false;
23      System.out.println("着地した！");
24    }
25  }
```

fly() を追加

land() を追加

11.1.2 「コピペ解決法」の問題点

さすが朝香くん。あっという間に SuperHero を作りあげたね。

はいっ。大学のレポート作成の課題では、ネットで調べてコピペ使いまくりましたから！

レポートの話は聞かなかったことにするけど、この方法だと後で困るかもしれないよ。

朝香さんのように元となるコードをコピー＆ペーストして、それに新しい機能を追加すれば、簡単に元のクラスを発展させることができます。解決方法としてはとてもシンプルですし、コードとしても問題なく動作するでしょう。

しかし、この方法によって作成された SuperHero クラスには、次のような 2 つの問題があります。

■追加・修正に手間がかかる

Hero クラスに新しいメソッドを加えたときや、Hero クラス内のメソッドを変更した場合、その変更を SuperHero クラスにも行う必要があります。なぜ

ならスーパーヒーローとは、「たくさんいるヒーローの中でも特に優れたひとにぎりの者」だからです。SuperHero は Hero ができることは当然すべてできなければなりません。

■把握や管理が難しくなる

SuperHero クラスは Hero クラスを元にしているので、この2つのクラスでソースコードの大半が重複することになります。これによりプログラム全体の見通しが悪くなり、メンテナンスがしづらくなります。

もしかしたら今後、Hero を元にした別のクラス（たとえば SuperHero や HyperHero、LegendHero、MagicalHero など）を作る必要が出てくるかもしれません。すると Hero クラスに何か変更があるたびに、すべての～ Hero クラスに対して Hero クラスと同じ修正を行う必要が生じます。これは、とてもめんどうですね。

11.1.3 継承による解決

このゲームは後で職業を増やしていきたいんです。ですから新しい職業クラスを作るたびにコピペしていると、後から問題が出て困りそうです。

そうだね。そもそも同じコードが何か所にも分散して書かれていることが諸悪の根源だ。

「コピペ解決法」を用いて類似したクラスを作成していくと、将来、元となったクラスが変更された際に、すべての類似クラスも修正しなければなりません。

しかし Java には、このようなことを懸念することなく類似したクラスを作ることを可能にする機能「継承」があります。これを使えば、SuperHero クラスは次ページのリスト 11-3 のようにスッキリと記述することができます。

リスト 11-3　Hero クラスを継承して SuperHero を作成する

SuperHero.java

```
1  public class SuperHero extends Hero {
2    private boolean flying;
3    public void fly() {
4      this.flying = true;
5      System.out.println("飛び上がった！");
6    }
7    public void land() {
8      this.flying = false;
9      System.out.println("着地した！");
10   }
11 }
```

追加した flying

「基本的には Hero と同じ」と宣言

追加した fly()

追加した land()

ポイントは、1 行目の extends です。この修飾子を用いた「class SuperHero extends Hero」という宣言は、「基本的に Hero クラスをベースにして SuperHero クラスを定義するので、Hero と同じメンバの定義は省略します（違いだけを記述します）」という意味になります。

Hero クラスを継承しているから「新しく増えたメンバだけ」を SuperHero クラスに書けばいいわけですね。

継承を用いたクラスの定義

```
class クラス名 extends 元となるクラス名 {
 親クラスとの「差分」メンバ
}
```

この SuperHero クラスがインスタンス化されるときに、JVM は「省略されているけれども、SuperHero クラスは Hero クラスに含まれている run()、

attack()、hp、name も持っている」と判断してくれます。

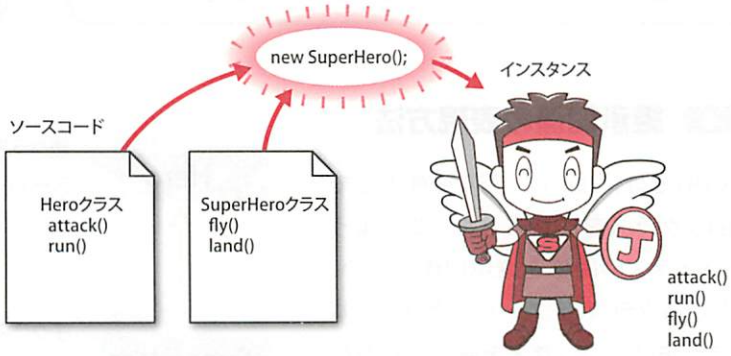


図 11-1　2 つのクラス定義に基づき、1 つのインスタンスが生成される

よって、SuperHero クラスのソースコードには run() メソッドがありませんが、インスタンス化されれば run() メソッドを呼び出せます（リスト 11-4）。

リスト 11-4

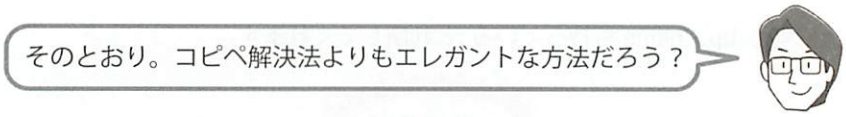
Main.java

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

このように、extends 修飾子を用いることによって、元となるクラスの「差分」だけを記述して新たなクラスを宣言することができます。

新たに定義するクラス (SuperHero) に着目すると、まるで「元となるクラス (Hero) から、メンバが自動的に引き継がれているように見える」ことから「継承」という名前が付いています。

もし Hero クラスに将来メソッドやフィールドの宣言が追加されれば、SuperHero でも自動的に使えるようになるのね。

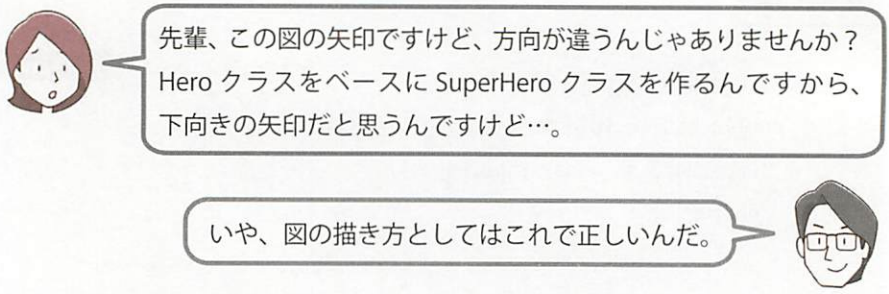


11.1.4 継承関係の表現方法

図 11-2　図における継承関係の記述方法

今回の例では、Hero クラスを継承して SuperHero クラスを作りました。このような２つのクラスの関係を**継承関係**といい、その元となるクラスを「スーパークラス」「基底クラス」「親クラス」などと呼び、新たに定義されるクラスを「サブクラス」「派生クラス」「子クラス」などと呼びます。

なお、継承関係を図で表現する場合は、図 11-2 のような矢印で記述するルールになっています。



慣れるまではこの矢印の方向に違和感を覚えるかもしれません。この矢印を直感とは逆に描くのには理由がありますが、それは本章の最後に説明します。今のところは、「クラス図では、継承の矢印は直感と逆で、子クラスから親クラスに向かって引く」とだけ覚えておいてください。

11.1.5 継承のバリエーション

継承のバリエーションは、２つのクラスの関係にはとどまりません。１つのクラスをベースとして、複数の子クラスを定義することもできますし、孫クラスや曾孫クラスを定義することもできます（図 11-3）。

ただし、Java では許されていない継承の構図が 1 つだけあります。

図 11-4 のように、複数のクラスを親として 1 つの子クラスを定義することを**多重継承**といいますが、Java ではこれを許可していません。

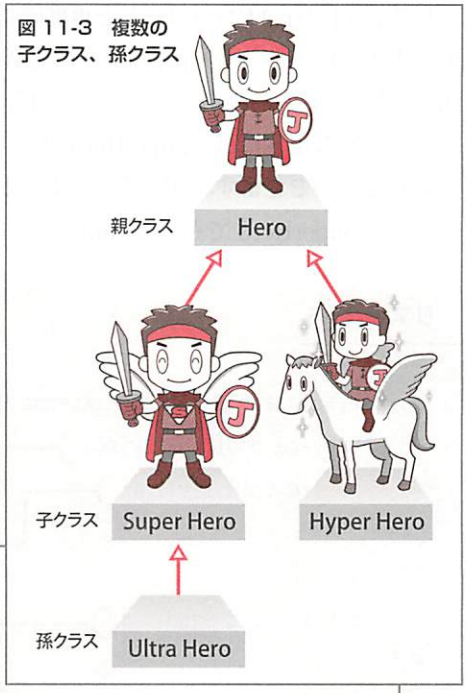


図 11-3　複数の子クラス、孫クラス

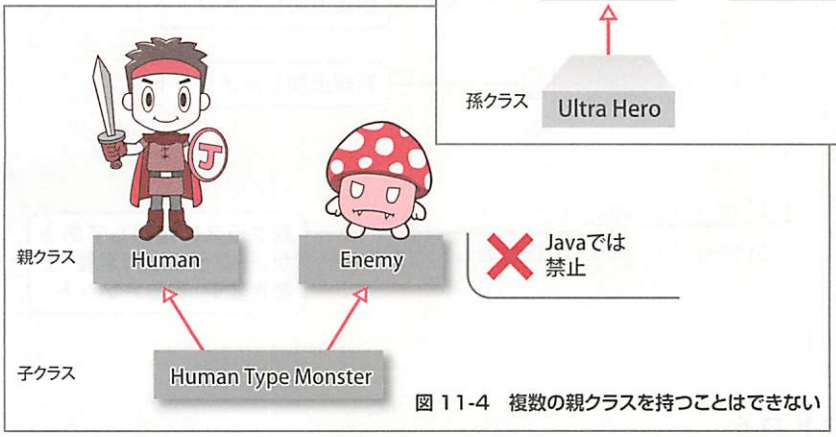


図 11-4　複数の親クラスを持つことはできない

11.1.6 オーバーライド

SuperHero の run() を呼んだときに表示される「逃げ出した」という表示を「撤退した」に変えたいけど、どうしよう…。

何やってるのよ、SuperHero クラスの run() メソッドを書き換えればいいじゃない！ …って、SuperHero クラスには run() メソッドの宣言がないんだっけ！

リスト 11-3(p.412)のコードを再度確認してください。SuperHero クラスには、fly() と land() の 2 つのメソッドしか定義されていません。SuperHero クラスの run() メソッドの動きだけを変えたい場合、どうすればよいでしょうか？

このような場合には、SuperHero クラスのコードに新しい run() メソッドを記述することができます。親クラスである Hero にも run() メソッドはありますが、子クラス SuperHero でも再度 run() メソッドを定義するのです。

リスト 11-5

SuperHero.java

```
1  public class SuperHero extends Hero {
2    private boolean flying;
3    public void fly() {
4      :
5    }
6    public void land() {
7      :
8    }
9    public void run() {
10     System.out.println("撤退した");
11   }
12 }
```

新規追加したフィールド

新規追加したメソッド

新規追加したメソッド

親クラスに定義してあるが、子クラスで再定義（上書き変更）するメソッド

リスト 11-6

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h = new Hero();
4      h.run();
5      SuperHero sh = new SuperHero();
6      sh.run();
7    }
8  }
```

実行結果
ミナトは逃げ出した
撤退した

リスト 11-5 の 1 行目で SuperHero クラスを定義する際、「基本的に Hero クラスをベースにする」と宣言したものの、9 行目で改めて run() メソッドを違う内容で定義し直しています（内容を上書きしている）。このように、親クラスを継承して子クラスを宣言する際に、親クラスのメンバを子クラス側で上書きすることを、**オーバーライド**（override）といいます。

以前に学んだオーバーロード（5.4 節）と**名前が似ているが、まったく異なるもの**なので混同しないでほしい。

継承を用いて子クラスに宣言されたメンバ

①親クラスに同じメンバがなければ、そのメンバは「追加」になる。
②親クラスに同じメンバがあれば、そのメンバは「上書き変更」される。

11.1.7 継承やオーバーライドの禁止

文字数の長さ制限がある LimitString クラスを作りたいんですけど、継承がうまくできなくて。

なるほど、String クラスの継承に失敗しているようだね。

朝香さんが作成しようとしているクラスは次のとおりです。

LimitString.java

```
public class LimitString extends String {
  ⋮
}
```

String クラスを継承

しかし、このソースコードをコンパイルしようとするとエラーが発生します。なぜなら、実は継承しようとしている String クラス（java.lang.String）は、「このクラスを継承して他のクラスを作ってはいけません（継承禁止）」と特別に指定されているクラスだからです。

Java の API リファレンスを見ると、String クラスは次のように宣言されていることがわかります。

```
public final class String extends Object…
```

Java では、**宣言時に final が付けられているクラスは継承できない**ことになっています。

もちろん、私たちがクラスを作成する際にも final を付ければ「継承禁止」にできます。たとえば、Main クラスの継承を禁止にするには、クラス宣言に final を追加します。

Main.java

```
1  public final class Main {
2    public static void main(String[] args) {
3      // メインメソッド
4    }
5  }
```

そもそもなぜ String クラスは継承禁止クラスとして宣言されているんですか？ 継承できたほうが絶対便利なのに。

String クラスを作った人は「String クラスのおかしな類似品」が世の中に出回る危険性を考えたのだろうね。

不具合なく完璧に動作するクラスがあっても、技術力がない人がそのクラスを継承し、オーバーライドによってメソッドの内容をメチャクチャに上書きしてしまったら、「異常な動作をする子クラス」ができてしまいます。

この「異常な子クラス」は、親クラスと似ているようでその内容はまったく異なる困った類似品であり、バグの原因となる危険性があります。特に String クラスはプログラム内で多用される大切なクラスなので、「正しく動作しない String の類似品」が出回ると致命的な不具合の原因になりかねません。

このような理由から String クラスには final が付けられていて、すべてのメソッドはオーバーライドできないようになっています。

確かに、String はどんなプログラムでもほぼ確実に使うものだし、バグがあったら大変なことになっちゃいます。

もし、クラスの継承は許可するものの、一部のメソッドについてのみオーバーライドを禁止したい場合は、そのメソッドの宣言に final を付けてください。**宣言に final が付けられたメソッドは、子クラスでオーバーライドができない**ことになっています。

リスト 11-7

Hero.java

```
1  public class Hero {
2    :
3    public final void slip() {
4      this.hp -= 5;
5      System.out.println(this.getName() + "は転んだ！");
6      System.out.println("5のダメージ");
7    }
8    public void run() {
9      System.out.println(this.getName() + "は逃げ出した！");
10   }
11   :
12 }
```

final が付いている slip() メソッドは子クラスでオーバーライド禁止

run メソッドは子クラスでオーバーライド可能

継承やオーバーライドの禁止

- クラス宣言に final を付けると、継承禁止
- メソッド宣言に final を付けると、オーバーライド禁止

実は継承の基本的な使い方や知識は、ここまでに紹介した内容がすべてだよ。

なんだぁ…花形機能というから、どんなに難しいかビクビクしていました。

とりあえずここまでの知識で大丈夫だ。どんどん継承を使ってほしい。すぐにいくつかの不自由に直面し、より深く継承を理解する必要が出てくるはずだからね。

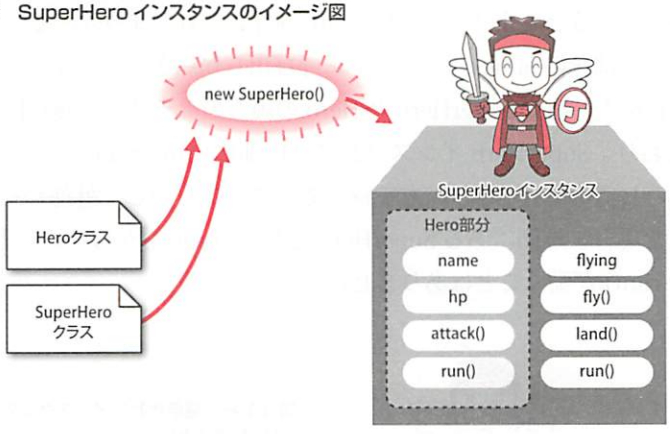
11.2 インスタンスの姿

11.2.1 インスタンスの多重構造

より踏み込んで継承を使いこなすには、継承を用いて定義されたSuperHeroのようなクラスから生まれたインスタンスが、実際にどのような姿をしていて、どのようにふるまうかを理解しておくことがとても重要です。継承を用いているインスタンスの姿を正しくイメージできれば、次節以降はもちろん、第12章の「高度な継承」や第13章の「多態性」もスムーズに理解できるでしょう。

では、SuperHeroのインスタンスの姿をイメージ図で見てみましょう。

図 11-5　SuperHero インスタンスのイメージ図



このインスタンスは、外から見れば1つのSuperHeroインスタンスなのですが、内部にHeroクラスから生まれたHeroインスタンスを含んでおり、全体として**二重構造になっている**ことに着目してください。

スーパーヒーローさんは、胸の中に「普通のヒーローとしての自分」を秘めているのね。

これ以降では、外側の部分を「子インスタンス部分」、内側の部分を「親インスタンス部分」と呼び、このイメージ図を通してさまざまな呼び出しや動作原理を考えていきましょう。

ちなみに、「親・子・孫」と3つのクラスが継承関係にある場合、孫クラスのインスタンスは三重構造になるよ。

11.2.2 メソッドの呼び出し

インスタンスの外部からメソッドの実行依頼が届く（呼び出しがある）と、多重構造のインスタンスは、**極力、外側にある子インスタンス部分のメソッドで対応**しようとします。

たとえば、fly() が呼び出されれば SuperHero クラスで定義された fly() メソッドが動きます。一方、attack() への呼び出しは、まずは外側の子インスタンス部分で対応しようとしますが、外側に attack() は存在しません。そこで内側の親インスタンス部分の attack() メソッドに呼び出しが届き、それが動作します。

run() メソッドは、SuperHero と Hero の両方のクラスで定義（オーバーライド）されており、SuperHero インスタンスは内部に**「SuperHero としての逃げ方」と「Hero としての逃げ方」の両方を持っています**。しかし、外部から run() を呼び出された場合、外側にある SuperHero としての run() が優先的に動作するため、内側の run() が動くことはありません。

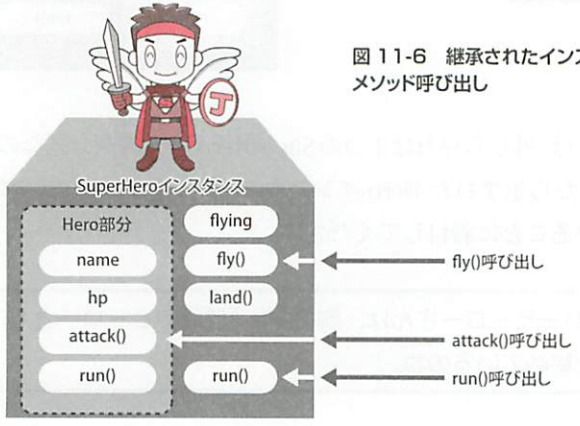


図 11-6　継承されたインスタンスのメソッド呼び出し

子クラスで run() を定義しても、親クラスの run() メソッドは上書きされてなくなるわけではないんですね。

親クラスの run() も子クラスの run() も両方ともインスタンスの中にあるんだ。ただ、親クラスの run() には呼び出しが届かないから「上書きされたように見える」だけなんだよ。

11.2.3 親インスタンス部へのアクセス

先輩、親インスタンス部の run() は、どうせ外部から呼び出せないんですし、存在価値がないように思いますけど？

いや、親インスタンスのメソッドが役立つこともあるんだよ。

頻度として多くはありませんが、内側の親インスタンス部に属するメソッドが大活躍することもあります。たとえば、次のような例を考えてみましょう。

SuperHero の追加仕様

SuperHero は、空を飛んでいる状態で attack() すると、Hero では 1 回だった攻撃を 2 回連続で繰り出すことができる。

この条件に従うために、次のようなオーバーライドを思いつくかもしれません。

リスト 11-8

SuperHero.java

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

1 回目の攻撃

```
6       if (this.flying) {
7         System.out.println(this.name + "の攻撃！");
8         m.hp -= 5;
9         System.out.println("5ポイントのダメージをあたえた！");
10      }
11    }
12    ⋮
13  }
```

2回目の攻撃

しかし、この方法では、将来 Hero クラスの attack() メソッドの処理内容が変わった場合に困った事態に陥ります。

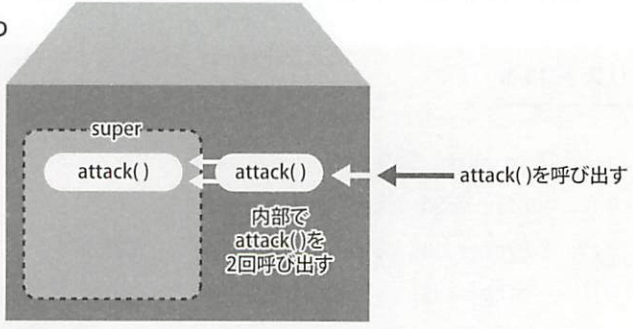
たとえば、Hero クラスの attack() メソッドが修正され、1 回の攻撃で敵に与えるダメージが 10 に修正されたとしましょう。SuperHero インスタンスを生み出し、fly() を呼び出した後で attack() を呼び出したらどうなるでしょうか？

飛んでいるスーパーヒーローは、Hero の攻撃を 2 回するわけだから、「10 ポイントダメージの攻撃を 2 回」になるべきですよね？

残念ながら「5 ポイントダメージの攻撃が 2 回」のままなんだ。SuperHero クラスでオーバーライドしちゃってるからね。

このような場合には、次のような呼び出しを実現できれば目的を果たせます。

図 11-7　親インスタンスのメソッドを呼び出す



Java のコードでは、次のように記述することで実現できます。

リスト 11-9

SuperHero.java

```
1  public class SuperHero extends Hero {
2    ⋮
3    public void attack(Matango m) {
4      super.attack(m);          ← 親インスタンス部の attack() を呼び出し
5      if (this.flying) {
6        super.attack(m);        ← 親インスタンス部の attack() を呼び出し
7      }
8    }
9    ⋮
10 }
```

super とは、「親インスタンス部」を表す予約語です。これを利用すれば、親インスタンス部のメソッドやフィールドに子インスタンス部からアクセスすることができます。

親インスタンス部のフィールドを利用する

super. フィールド名

親インスタンス部のメソッドを呼び出す

super. メソッド名 (引数)

super を付けずに単に「attack()」と呼び出してはダメなんですか？

ダメだ。super を付けないということは「this.attack()」と同じ意味になるね。それでは図 11-8 のように、attack() を呼び出し続ける無限ループになってしまうよ。

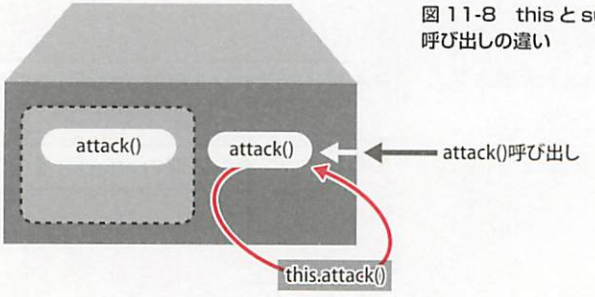


図 11-8　this と super による呼び出しの違い

「祖父母」インスタンス部へのアクセスは不可能!

A クラス（祖父母）、B クラス（親）、C クラス（自分）という 3 つのクラスが継承関係にあるとき、そこから生成されたインスタンスも 3 重構造になりますが、その際に一番外側に相当する C インスタンスについて考えましょう。

C クラスのメソッドは、一番外側のインスタンス部（自分自身）へは this、そして親インスタンス部へは super でアクセスできます。しかし、残念ながら「祖父母」にあたるインスタンス部へアクセスする手段は準備されていません。つまり、C クラスのメソッドが A クラスのインスタンス部に直接アクセスすることはできないのです。

11.3 継承とコンストラクタ

11.3.1 継承を利用したクラスの作られ方

前節では、インスタンス化されたSuperHeroがどのような姿なのかを紹介しました。その姿は、Heroインスタンスを内部に持つ多重構造になっていることを理解できたと思います。この多重構造は、クラスがnewされた際に以下のような段階を経て構築されます。

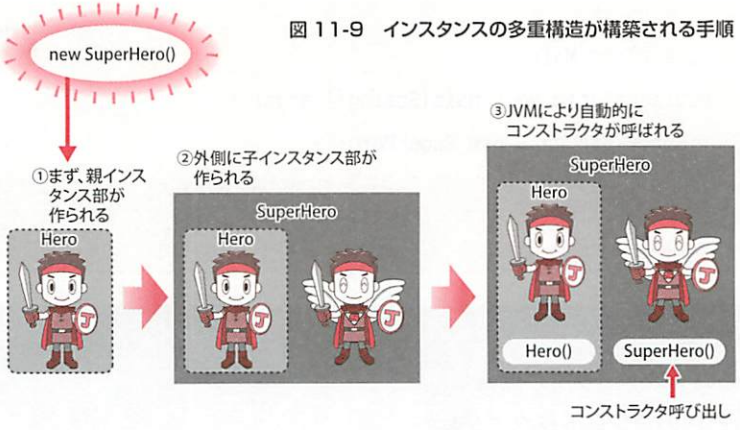


図 11-9　インスタンスの多重構造が構築される手順

図11-9の③にあるように、SuperHeroインスタンスが完成すると、JVMは自動的にSuperHero()コンストラクタを呼び出します。ここで次のようなコードを書いてみると、興味深い内部動作を確認できます。

リスト 11-10

Hero.java

```
1  public class Hero {
2    ⋮
3    public Hero() {
```

```
4       System.out.println("Heroのコンストラクタが動作");
5     }
6     ⋮
7   }
```

SuperHero.java

```
1   public class SuperHero extends Hero {
2     ⋮
3     public SuperHero() {
4       System.out.println("SuperHeroのコンストラクタが動作");
5     }
6     ⋮
7   }
```

Main.java

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

実行結果

```
Heroのコンストラクタが動作
SuperHeroのコンストラクタが動作
```

あれ？ SuperHero インスタンスが完成したら JVM は SuperHero のコンストラクタを呼び出すはずよね？

その前に、どうして Hero のコンストラクタも動いているんだろう？

SuperHero を new することで SuperHero() コンストラクタが動作するのは理解できますが、なぜか内側インスタンスの Hero() コンストラクタも勝手に動作しています。

実は Java では、「**すべてのコンストラクタは、その先頭で必ず内部インスタンス部（＝親クラス）のコンストラクタを呼び出さなければならない**」というルールになっています。

同じクラスの別コンストラクタを呼び出すための「this()」（9.2.7 項）に似た「super()」という記述で親クラスのコンストラクタを呼び出すことができます。

親クラスのコンストラクタの呼び出し

super(引数);

※ただし、コンストラクタの最初の行にしか記述できない。

よって、本来 SuperHero コンストラクタは、以下のような書き方をしなければなりません。

```
public SuperHero() {
  super();
  System.out.println("SuperHeroが生成されました");
}
```

もしプログラマがコンストラクタの一行目に super() を書いていない場合、コンパイラによって「**super();**」という行が自動的に挿入されます。リスト 11-10 のプログラムでは、この「暗黙の super()」が、自動的に Hero() コンストラクタを呼び出していたというわけです。

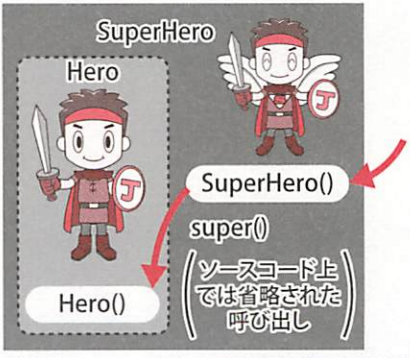


図 11-10　super() によるコンストラクタの呼び出し

つまり、コンストラクタは内側のインスタンス部分のものから順に呼ばれていくということですね。

そのとおり。ちなみに親インスタンス部のコンストラクタを呼び出す super() は、前節で学習した「super. メンバ名」とはまったく関係がないので混同しないよう注意してほしい。

11.3.2 親インスタンス部が作れない状況

このように、インスタンスが構築・初期化される手順を理解すると、ある条件で困ったことが発生します。次のリスト 11-11 を題材に考えてみます。

リスト 11-11

Item.java

```
1  public class Item {
2    private String name;
3    private int price;
4    public Item(String name) {
5      this.name = name;
6      this.price = 0;
7    }
8    public Item(String name, int price) {
9      this.name = name;
10     this.price = price;
11   }
12 }
```

引数 1 つのコンストラクタ

引数 2 つのコンストラクタ

Weapon.java

```
1  public class Weapon extends Item { … }
```

Item を継承し Weapon を定義

Main.java

```
1  public class Main {
```

```
2    public static void main(String[] args) {
3      Weapon w = new Weapon();
4    }
5  }
```

このコードではエラーが発生しますが、その理由を1ステップずつ整理しながら解説していきましょう。

Main.javaの3行目のnew Weapon()により、JVMはWeaponインスタンスを生成しようとします。Weaponクラスは Itemクラスを継承していますので、このインスタンスは、内部にItemインスタンスを含む多重構造になっているはずです。

二重構造のインスタンスを作り終えると、JVMは自動的にWeapon()コンストラクタを呼び出そうとします。しかし、Weaponクラスにはコンストラクタが定義されていないため、暗黙的に次のような「デフォルトコンストラクタ(9.2.6項)」が定義され動作します。

```
public Weapon() {
}
```

しかし、ここで前項で学んだことを思い出してください。すべてのコンストラクタの先頭行には実は「super();」が隠れていますので、実際には次のようになります。

```
public Weapon() {
  super();
}
```

このように自動生成されたWeaponクラスのコンストラクタは、親クラスItemのコンストラクタを**引数なしで**呼ぼうとします。ここで、呼び出される側のItemクラスのコンストラクタの宣言を見てみましょう(リスト11-11)。引数1つのもの(Item.javaの4行目)と2つのもの(同8行目)、あわせて2つのコンストラクタが宣言されていますが、**引数が0個のコンストラクタは存在しません。**

つまり、Item クラスのコンストラクタ呼び出しには、必ず引数が 1 つか 2 つ必要であり、Weapon クラスのコンストラクタからであっても「super();」のように引数がない呼び出しはできないのです。

11.3.3 内部インスタンスのコンストラクタ引数を指定する

このように、内部インスタンスの初期化を行うコンストラクタ（Item() コンストラクタ）に引数を与える必要がある場合は、super() の呼び出し時に明示的に引数を渡します。

Weapon.java

```
1  public class Weapon extends Item {
2    public Weapon() {
3      super("ななしの剣");
4    }
5  }
```

引数 1 つの親クラスコンストラクタを呼び出す

これで、Weapon クラスのインスタンス化によって内部で Item インスタンスが作られる際、リスト 11-11 の Item.java の 4 行目のコンストラクタが動作し、常に「ななしの剣」という名前になります。「super(" ななしの剣 ",300);」と記述すれば、常に 2 つの引数を持つ 8 行目の Item コンストラクタが動作するでしょう。

つまり、super() に与える引数の数と型によって、「親インスタンス部が初期化されるときに利用されるコンストラクタ」を明示的に指定できるのです。

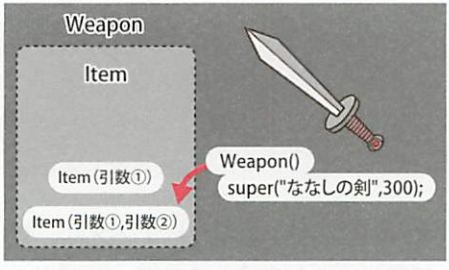


図 11-11　引数によって指定した親コンストラクタの呼び出し

11.4 正しい継承、間違った継承

11.4.1 is-a の原則

継承に「正しい」とか「間違ってる」なんてあるんですか？　ボクのプログラムで誤った継承の使い方をしていないか心配になってきました…。

大丈夫。簡単なチェック方法があるよ。

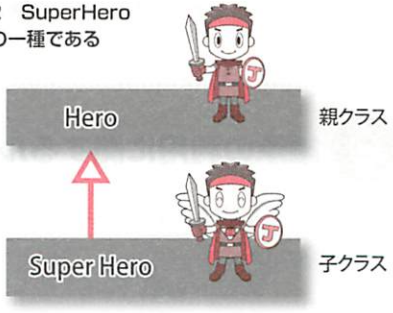
正しい継承とは、「is-a の原則」といわれるルールに則っている継承のことです。そして is-a の原則とは、子クラスと親クラスの間に次のような概念的な関係が成立しているべきであるとする原則です。

is-a の関係

子クラス is-a 親クラス（子クラスは、親クラスの一種である）

A is-a B は日本語で「A は B の一種である」という意味であり、この文章の意味が自然であれば正しい継承です。スーパーヒーローは「特殊能力を持った特別なヒーロー」ですが、あくまでヒーローの一種であることには間違いありません。

図 11-12　SuperHero は Hero の一種である



もし、「(子クラス) is-a (親クラス)」「(子クラス) は (親クラス) の一種である」という文章を作って不自然さを感じたら、継承の誤りを疑いましょう。

11.4.2 間違った継承の例

現実世界の登場人物同士に**概念として is-a の関係がないにもかかわらず、継承を使ってしまうのが「間違った継承」**です。例を挙げてみましょう。

ここに「名前」と「値段」のフィールドを持つ Item クラスがあります。このクラスは、勇者たちが冒険のために持ち歩く「薬草」や「ポーション」などのアイテム(小道具)を表すクラスです。そして今、私たちは新たに House クラスを作ろうとしています。House クラスには、所有者や床面積、間取りや住所などのほか、「家の名前」「家の値段」のフィールドも必要です。

Item クラスを継承して House クラスを作ればいいと思いますけど？

いや、その発想こそが「間違った継承」の原因だよ。

Item クラスを継承して House クラスを作ることは原理上、可能です。実際、名前と値段のフィールドも継承され、問題なく動作します。ですが、「House is-a Item (家はアイテムの一種である)」という文章には違和感を覚えませんか？勇者は冒険のために家を持ち歩くことはありません。

このように、「フィールドやメソッドが流用できるから」という安易な理由で継承をしてはいけません。「動くか動かないか、便利か便利でないか」ではなく、**is-a であるかどうかに基づいて**、継承は利用すべきです。

継承の利用に関するルール

is-a の原則が成立しないならば、ラクができるとしても継承を使ってはならない。

11.4.3 間違った継承をすべきでない理由

便利だからいいじゃないですか。なんでダメなんですか？

is-a の関係ではない継承を使ってはならない理由は 2 つあります。

- 将来、クラスを拡張していった場合に現実世界との矛盾が生じるから。
- オブジェクト指向の 3 大機能の最後の 1 つ「多態性」を利用できなくなるから。

多態性については第 13 章で解説するとして、ここでは「現実と矛盾が生じていくこと」について、House クラスと Item クラスの例を用いて解説しましょう。

確かに House クラスを作った時点では、Item クラスを継承していても問題がないように思えます。しかし、これは単に「たまたま現時点では実害がないだけ」であって、より忠実に現実世界の家やアイテムをまねようとクラスを改良していくと次々と矛盾が生じます。

たとえば、アイテムは敵に投げつけてダメージを与えることができるとしましょう。そこで、Item クラスに「敵に投げつけたときにあたえるダメージを返すメソッド」、getDamage() を追加します。

Item.java

```
public class Item {
  ⋮
  public int getDamage() {
    return 10;
  }
  ⋮
}
```

このメソッドは継承され House クラスでも利用可能になりますが、現実に沿って考えると家を投げることなどできるわけがありませんし、そのダメージを算出する、getDamage() メソッドが House クラスに対して呼べること自体が極めて不自然です。

図 11-13　間違った継承を用いると、現実とは矛盾したメンバが現れる

「投げつけたときのダメージがある家」という House クラスは、すでに**現実世界の家と乖離**してしまっており、**オブジェクト指向の原則から外れています**。

なるほど…でも、「House クラスには getDamage() があるけど、無視して使わない」ことにすればいいんじゃない？

だめよ。「存在するけど実は使っちゃダメなメンバ」がいくつもあるクラスなんて、怖くて使えないわ。

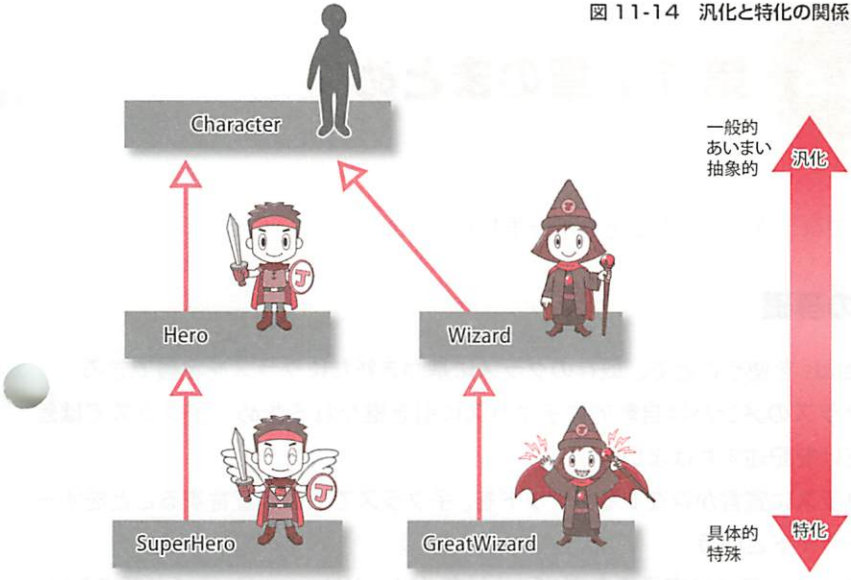
11.4.4 汎化・特化の関係

正しい継承が「is-a」の関係で結ばれるということは、子クラスになるほど「特殊で具体的なもの」に具体化（**特化**）していき、親クラスになるほど「一般的で、抽象的・あいまいなもの」に一般化（**汎化**）していくことになります。

特化すればするほど、より詳細にフィールドやメソッドを定めることができ、メンバは増えていきます。逆に汎化すればするほど、フィールドやメソッドを多く定めることは難しくなってきます。

たとえば**キャラクターであれば、どんなものでも必ず名前と HP は持っている**でしょうから、Character クラスには name と hp フィールドぐらいは定義できます。

図 11-14　汎化と特化の関係



より具体的な魔法使いになると、最低でも MP を持っていて火の玉ぐらいは放てるはずであり、クラス定義には mp フィールドや fireball() メソッドが加わるでしょう。 さらに具体的な「ひとにぎりの大魔法使い（GreatWizard）」は雷を落とす lightning() メソッドなど、Wizard が持っていないメソッドも持つでしょう。

ちなみに、クラス図において継承関係を表す矢印は「クラスが汎化していく方向」を表すための矢印なんだ。

なるほど。だからクラス図の矢印は、継承の方向（特化の方向）とは逆向きに描かれていたのね。

これまでは、継承のことを「コードの重複記述を減らすための道具」と捉えて学習してきたと思います。しかし継承は、**「ある 2 つのクラスに特化・汎化の関係があることを示す」ための道具**でもあるのです。

11.5 第11章のまとめ

この章では、次のようなことを学びました。

継承の基礎

- extends を使うことで、既存のクラスに基づき新たにクラスを定義できる。
- 親クラスのメンバは自動的に子クラスに引き継がれるため、子クラスでは差分だけを記述すればよい。
- 親クラスに宣言が存在するメソッドを、子クラスで上書き宣言することをオーバーライドという。
- final 付きクラスは継承できず、final 付きメソッドはオーバーライドできない。
- 正しい継承とは「子クラス is-a 親クラス」の文章に不自然がない継承である。
- 継承には、「抽象的・具体的」の関係にあることを定義する役割もある。

インスタンスの姿

- インスタンスは内部に親クラスのインスタンスを持つ多重構造をとる。
- より外側のインスタンス部に属するメソッドが優先的に動作する。
- 外側のインスタンス部に属するメソッドは、super を用いて内側インスタンス部のメンバにアクセスできる。

コンストラクタの動作

- 多重構造のインスタンスが生成されると、JVM は自動的に一番外側のコンストラクタを呼ぶ。
- すべてのコンストラクタは、先頭で「親インスタンス部のコンストラクタ」を呼び出す必要がある。
- コンストラクタの先頭に super() がなければ、暗黙的に「super();」が追加される。

11.6 練習問題

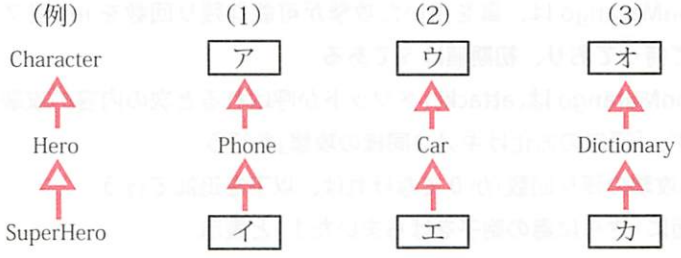
練習 11-1

次の中から「誤った継承」であるものをすべて選んでください。

①スーパークラス:Person　　サブクラス:Student
②スーパークラス:Car　　サブクラス:Engine
③スーパークラス:Father　　サブクラス:Child
④スーパークラス:Food　　サブクラス:Susi
⑤スーパークラス:SuperMan　　サブクラス:Man

練習 11-2

次のクラスに対する「親クラス」と「子クラス」を 1 つずつ考案して自由に挙げてください。



練習 11-3

次のようなクラス Matango があります。

Matango.java

```
1  public class Matango {
2    int hp = 50;
3    private char suffix;
4    public Matango(char suffix) {
5      this.suffix = suffix;
```

```
6    }
7    public void attack(Hero h) {
8      System.out.println("キノコ" + this.suffix + "の攻撃");
9      System.out.println("10のダメージ");
10     h.setHp(h.getHp() - 10);
11   }
12 }
```

このクラスを利用し、次の仕様に則ったPoisonMatangoクラスを作成してください。

ア．お化け毒キノコ（PoisonMatango）は、お化けキノコ（Matango）の中でも特に「毒攻撃」ができるもの。

イ．PoisonMatangoは以下のコードでインスタンス化できるクラスとする。

```
PoisonMatango pm = new PoisonMatango('A');
```

ウ．PoisonMatangoは、毒を用いた攻撃が可能な残り回数をint型フィールドとして持っており、初期値は5である。

エ．PoisonMatangoは、attack()メソッドが呼ばれると次の内容の攻撃をする。

①まず、「通常のお化けキノコ同様の攻撃」を行う。

②「毒攻撃の残り回数」が0でなければ、以下を追加で行う。

③画面に「さらに毒の胞子をばらまいた！」と表示。

④勇者のHPの1/5に相当するポイントを勇者のHPから引き、そのポイントを示すよう「～ポイントのダメージ」と表示する。

⑤「毒攻撃の残り回数」を1減らす。

11.7 練習問題の解答

問題 11-1 の解答

誤っているものは②、③、⑤です。

②・・・エンジンは車の「一部」であり、両者は has-a の関係 (p.341) にあります。

③・・・継承では親クラスや子クラスという用語を用いますが、概念としての親や子とは関係ありません。「子どもは父親の一種」ではありません。

⑤・・・スーパーマンは人間の一種ですので、スーパーという用語が付いていても、サブクラス(子クラス)です。

問題 11-2 の解答

以下は解答例です。このほかにも多数考えられます。

(ア) Device (装置)、Tool (道具)など。
(イ) MobilePhone (携帯電話)、SmartPhone (スマートフォン)など。
(ウ) Vehicle (乗り物)、Property (資産)など。
(エ) SportsCar (スポーツカー)、HybridCar (ハイブリッドカー)など。
(オ) Book (書物)、InformationSource (情報源)など。
(カ) EJDictionary (英和辞典)、Encycropedia (百科事典)など。

問題 11-3 の解答

以下は解答例です。

PoisonMatango.java

```
1  public class PoisonMatango extends Matango {
2    private int poisonCount = 5;
3    public PoisonMatango(char suffix) {
4      super(suffix);
5    }
6    public void attack(Hero h) {
```

```
7       super.attack(h);
8       if (this.poisonCount > 0) {
9         System.out.println("さらに毒の胞子をばらまいた");
10        int dmg = h.getHp() / 5;
11        h.setHp(h.getHp() - dmg);
12        System.out.println(dmg + "ポイントのダメージをあたえた！");
13        this.poisonCount--;
14      }
15    }
16  }
```