第12章

高度な継承

私たちは現実世界において、
無意識に多くのものを抽象的に捉え、利用しています。
オブジェクト指向の目的が「現実世界の再現」である以上、
Javaでも「抽象的な登場人物」を上手に扱える必要があります。
この章では、Java仮想世界においても、抽象的であいまいなクラスを
正しく・安全に・便利に利用するために準備されている
クラスの定義方法を紹介します。

CONTENTS

12.1 未来に備えるための継承
12.2 高度な継承に関する2つの不都合
12.3 抽象クラス
12.4 インタフェース
12.5 第12章のまとめ
12.6 練習問題
12.7 練習問題の解答

未来に備えるための継承

12.1.1 高度な継承を学ぶにあたって

第 11 章ではオブジェクト指向の花形「継承」について学びました。そのメリットを十分に実感できたのではないでしょうか。また、章の最後では、正しい継承が「抽象的なクラスと具体的なクラスの関係を結ぶ」ことも学びました。図 11-14（p.437）のような継承ツリーを親クラス、その親クラス、さらにその親クラス…と辿っていくほど、クラスはあいまいで抽象的なものになるのでしたね。

この第 12 章では、主にこのツリーの上側に登場する**「あいまいなクラスたちの定義方法」**について学びます。

これまでに学んできた通常の方法でこれらのクラスを定義しても、プログラムは動作します。しかし、「あいまいなクラスたち」専用のクラス定義方法をマスターし、高度な継承を実現することで、より安全で便利にクラスを利用できるようになるのです。

具体的には「抽象クラス」や「インタフェース」というものを学んでいくよ。

なんだか名前からして難しそうですね…。

確かに難しそうなイメージがあるけど、コツを 1 つ知っておくだけでグンと楽になるから安心してほしい。

残念なことに、「普通の継承はすぐ理解できたのに、高度な継承でつまずく」という人も珍しくありません。実は「高度な継承を学習するには、**ある意識を今までと切り替える必要がある**」というコツがあるのです。

そこで章の始めに、まずこの「意識の切り替え」について紹介していきます。

抽象クラスやインタフェースの理解に自信がないという人、そして過去の学習で挫折したという人も、このコツをおさえた上でぜひ再チャレンジしてほしい。きっとマスターできるはずだ。

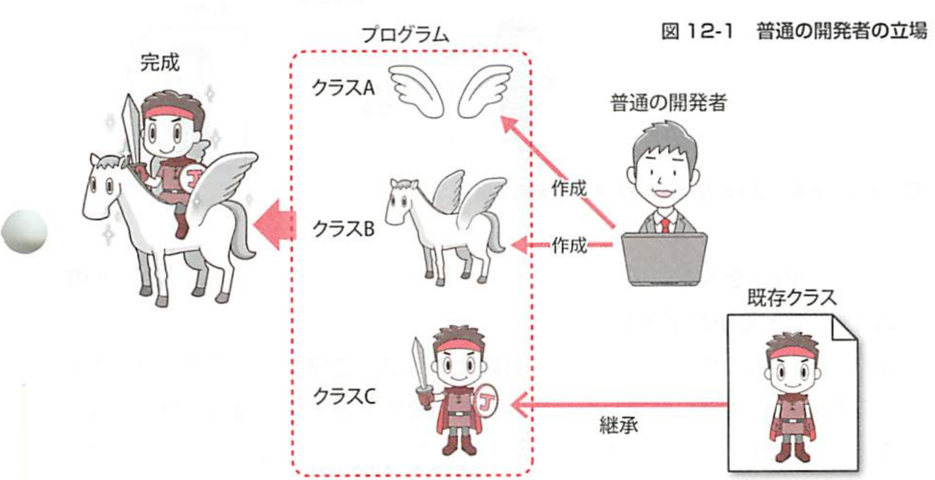
12.1.2 新しい「立場」で考える

この章で学ぶ高度な継承は、文法的に難しいものではありません。にもかかわらず、なぜつまずく人が多いかというと、**高度な継承を使うときの「立場」が、今までの「立場」とはまったく違う**ということを意識せず学習を始めてしまうからなのです。

今までみなさんは、自分がどのような「立場」でJavaを使うことをイメージしてきましたか？ その多くは、作る必要がある（または作りたい）プログラムは明確に決まっていて、**そのプログラムのためだけに必要なクラスを作って目的のプログラムを完成させる「立場」**ではないでしょうか。

そして、もし開発すべきクラスと類似した既存のクラスがあれば、**継承を利用して子クラスを作る**ことにより、ゼロから開発することなく、いくつかのメンバを追加するだけで効率よくクラスを開発できるのでしたね。

図 12-1　普通の開発者の立場



ここで「既存クラス」に注目して、少し想像を膨らませてみましょう。きっとこの**クラスを事前に開発しておいてくれた開発者**がどこかにいるはずです。

その作者は、自分の作ったクラスがどんなプログラムに利用されるか想像もつかない過去の段階で、「いつか誰かが、このクラスを継承して開発したら便利だろう」と未来に思いを馳せ、継承の材料となる既存クラスを作ってくれたのです。とてもありがたいことですね。

ここで次の図 12-2 を見ると、異なる「立場」で活躍する 2 種類の開発者がいることがわかります。

立場 1：現在、目の前のプログラム開発に必要なクラスを作る開発者（既存クラスを継承し子クラスを作る）。

立場 2：未来に備え、別の開発者が将来利用するであろうクラスを準備しておく開発者（親クラスとなるクラスを作っておく）。

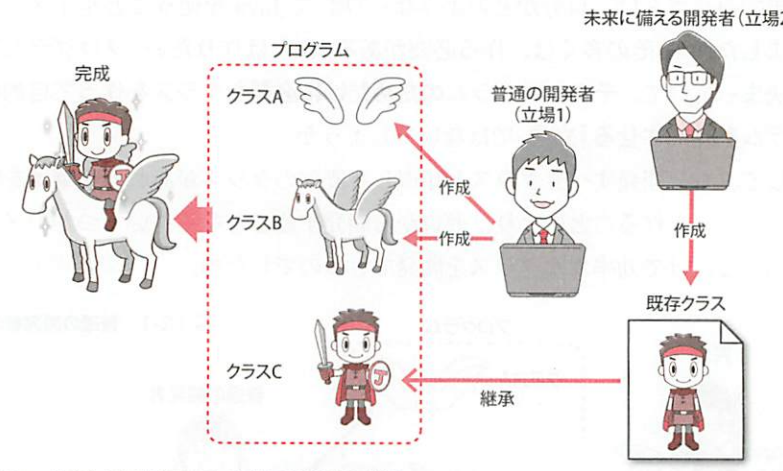


図 12-2　未来に備え既存クラスを作る開発者の立場

この章の内容をスムーズに理解するためには、この 2 つの立場の存在を明確に意識・区別できるかがポイントです。

前章までに私たちが学習してきた知識はすべて、立場 1 としてプログラム開発をするために必要なものでした。一方、この章で学習する知識は、みなさんが立場 2 としてプログラムを作るときに必要となるものです。なぜなら、この章で学ぶ「抽象クラス」「インタフェース」とは、立場 2 の人たちが、「立場 1 の人たちに安全・便利に使ってもらえる親クラスを作る」ための道具だからです。ぜひ、立場 2の開発者になってクラスを作ることを想像しながら本章を読み進めてください。

なるほど。でも、ボクのような新入社員が入社直後に、いきなり「未来に備える大事な立場」を任されることはなさそうですね。

抽象クラスやインタフェースを自ら作ることはなくても、継承元として利用することはあるから、マスターしておく必要はあるよ。

12.1.3 「未来に備える開発者」の立場の具体例

たとえばゲーム開発の例を考えてみましょう。プロジェクトではAさん、Bさん、Cさん、そしてあなたの4人で開発を進めていますが、ある日「開発効率が悪い」ということが問題になりました。

そこで「プロジェクト全体の開発効率を改善する責任者」に任命されたあなたが調査したところ、「開発者がそれぞれ、HeroやWizardなどの似たクラスをイチから作っている」ということに気づきます（図12-3）。

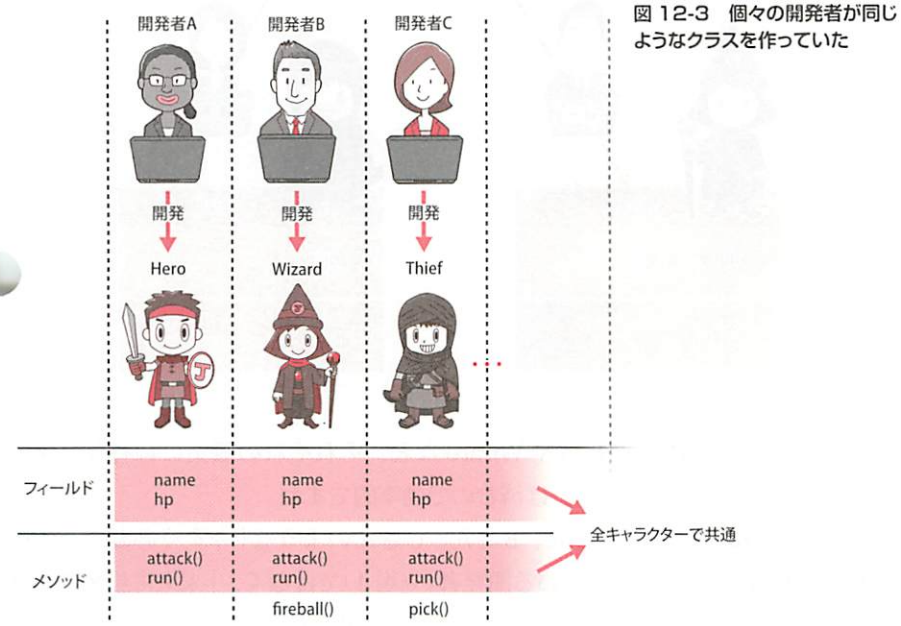
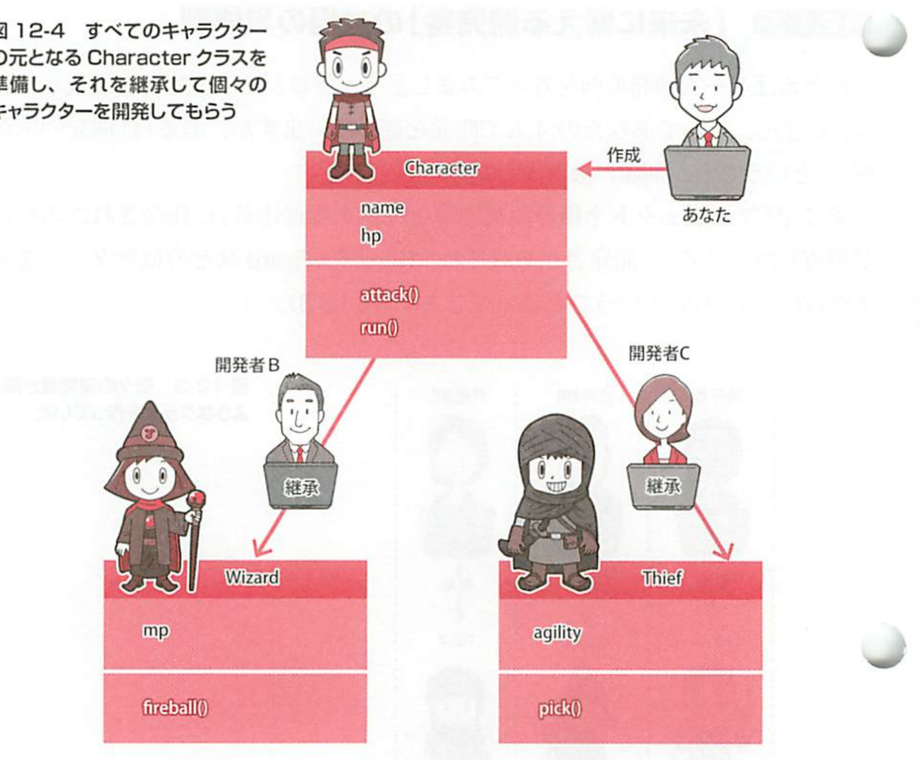


図12-3 個々の開発者が同じようなクラスを作っていた

各キャラクターのクラスでは、name や hp などのフィールドと、attack()、run() などのメソッドは共通ですので、それぞれ別に開発するのはムダです。また、今後のバージョンアップにより「商人」や「占い師」など、さまざまなキャラクターが増える予定ですが、それらも一から開発していては効率が悪そうです。

そこで、あなたは各クラスに共通するフィールドやメソッドを持つ Character クラスを準備し、各開発者に対しては「みなさんは私の作った Character クラスを継承して、独自のフィールドやメソッドを付け足すだけで大丈夫ですよ」とアナウンスします（図 12-4）。

図 12-4　すべてのキャラクターの元となる Character クラスを準備し、それを継承して個々のキャラクターを開発してもらう



これなら各開発者は、Hero や Wizard など、それぞれの職業に特有なフィールドやメソッドだけを開発すれば済むので効率的ですね。

このときのあなたは、A さん、B さん、C さんのような「今すぐ必要な、実際に利用されるクラス」を作っている開発者（立場 1）ではなく、「未来に備えて、継承元となるクラス（継承の材料）」を作るという立場（立場 2）にあります。

開発プロジェクトの最前線で今すぐ必要な Hero、Wizard クラスの開発をがんばる A さん、B さん、C さんに対して、あなたは後方から開発支援の道具（＝共通部分まで事前に作っておいた Character クラス）を供給して援護していると捉えてもいいでしょう。

あなたの作る、たった 1 つのクラスの優劣が、それを利用する複数の開発者たちの開発効率に影響するわけです。立場 2 のあなたが意識すべきことは、「立場 1 の開発者が効率よく安心して利用できる継承の材料をいかに作るか」ということなのです。

「未来に備える開発者」の役割

ほかの開発者が効率よく安心して利用できる継承の材料を作ること。

未来のために継承の材料を作っておくという立場では、ほかの開発者や未来の開発者に少しでもラクをしてもらいたい、という思いやりが大事なんですね。

高度な継承に関する2つの不都合

12.2.1　2つの不都合、3つの心配

効率アップのためにCharacterクラスを作れば一件落着ですね。

いやいや、ところが落着しないんだ。湊くんが本当に、Aさんや B さんがラクになれるよう心を砕いて Character クラスを作ろうとすると、いくつか「不安」が出てくるはずなんだ。

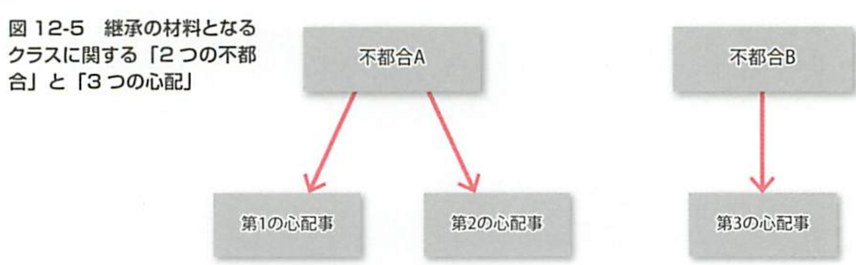
えっ！ どうしてですか？

立場 2 として何を意識すべきかを真剣に考えないなら、単に Character クラスを作って終わりにすることもできます。

しかし、「他の開発者がラクできるように心を砕くこと」を強く意識して Character のようなクラスを開発していると 2 つの「不都合」に直面します。その不都合を原因とする、さまざまな「心配」も出てくることでしょう。「抽象クラス」や「インタフェース」は、この不都合や不安を解決してくれる道具です。

それでは、図 12-5 のような関係にある、それぞれの不都合と不安を 1 つずつ見ていきましょう。

図 12-5　継承の材料となるクラスに関する「2 つの不都合」と「3 つの心配」



12.2.2 最初の不都合

まずは最初の不都合Aを体験するため、実際にCharacterクラスを作成してみましょう。

リスト 12-1

Character.java

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

ここを記述しようとして手が止まる

Characterクラスを実際に書こうとすると、attack()メソッドの内容に差しかかったところで手が止まってしまうはずです。

そうなんです…。「attack()メソッドを書かなきゃいけない」んだけど、「ダメージを何ポイントと書いたらいいのかわからない」というか…。

なぜ「手が止まってしまったか」を、より深く考えてみましょう。このCharacterクラスは将来、さまざまな開発者によって継承され、HeroやWizardやDancerなどを開発する際の材料として利用されます。

12章

しかし、未来に完成するであろうHeroやWizard、そしてDancerは、それぞれお化けキノコを攻撃したときに与えるダメージが違うはずです。腕っ節の強いHeroであれば与えるダメージは10ポイント、ひ弱なWizardなら5ポイント、さらに今は存在しませんが、未来に追加されるかもしれない強力なキャラクターでは100ポイントなどもありえます。

つまり、Characterクラスを作っている時点では、まだ「attack()メソッドの内容を確定できない」ため、書きようがないのです。

不都合A

継承の材料となるクラスを作る時点では、その処理内容をまだ確定できない「詳細未定メソッド」が存在する。

12.2.3 不都合Aに対する間違った解決方法

それではattack()メソッドのような内容を確定できないメソッドを、どのように記述すればよいのでしょうか？ まず考えつくのが、Characterクラスに、そもそもattack()メソッドを記述しないようにするという方法です（図12-6）。HeroやWizardなどの新しいキャラクタークラスを作成する際には、それぞれ継承先のクラスでattack()メソッドを追加してもらいます。

しかし、この方法では他の開発者が将来新しいキャラクタークラスを作成する際、継承先のクラスにattack()メソッドを追加し忘れると「攻撃できないキャラクター」ができてしまいます。

そもそも「現実世界（ゲームの世界）のすべてのキャラクターはHP属性を持ち、攻撃ができる」という前提でCharacterクラスを作り始めたのに、「attack()を持たないキャラクター」ができてしまっては困ります。

うーん。でも、そんなことは「必ずattack()を作るように各開発者自身が気をつければいい」んじゃないですか？

Character
name
hp
run()
作成
自分
自分は
attack()は
作らない
継承
Wizard
mp
attack()
fireball()
未来の開発者
それぞれの
職業クラスで
attack()を作る

図 12-6　それぞれ継承先のクラスで attack() メソッドを追加してもらう

> 第 10 章でも説明したとおり、人間は必ずミスをする。その責任を人に求めるのではなく「ミスを防ぐしくみ」を考えるべきなんだ。

そもそもオブジェクト指向とは、「現実世界（今回の場合はゲームキャラクターたちが住む世界）を正確に写し取る」ことでした。そして、**現実世界と Java コードの世界に矛盾が生じる余地があるから不具合が生じる**のです。

「キャラクターであれば少なくとも攻撃ができるはず」という前提を考えると、「攻撃できないキャラクターが作れてしまうこと」は万が一にもあってはなりません。**Character クラスは必ず attack() メソッドを持っているべき**なのです。

現実世界に対応したメソッド定義の必要性

「現実世界の登場人物が持つ操作」なのであれば、クラスのメソッドは存在しているべきである（仮に、メソッドの処理内容は確定困難であったとしても）。

12.2.4 不都合Aに対する対応策と2つの心配

不都合Aの対応策として、「Characterクラスのattack()メソッドは内容を確定できないので、とりあえず空にしておこう」と思いつくかもしれません。他の開発者がattack()メソッドを継承して、それぞれの職業クラスを作成する際に、その職業に最適なattack()メソッドでオーバーライドしてもらう、という考え方です。

リスト12-2　attack()メソッドの中身を空にしておく

Character.java

```
1  public class Character {
2    String name;
3    int hp;
4    public void run() {
5      System.out.println(this.name + "は逃げ出した");
6    }
7    public void attack(Matango m) {
8    }
9  }
```

メソッドの中身を空にしておく（7～8行目）

リスト12-3　未来の開発者が開発するコード

Hero.java

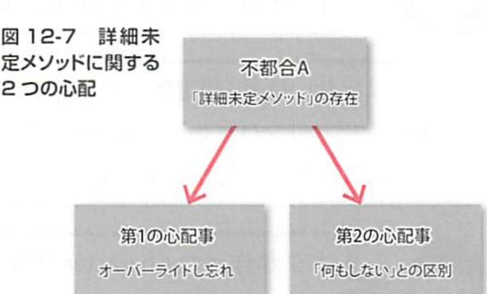
```
1  public class Hero extends Character {
2    public void attack(Matango m) {
```

Characterのattackメソッドをオーバーライドする（2行目）

```
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 }
```

なかなか悪くない方法です。しかし、あなたの「Characterクラスを利用してくれる未来の開発者たち」のことを思いやる気持ちが強いほど、次の２つの心配が頭をよぎるでしょう。

図 12-7　詳細未定メソッドに関する２つの心配



12.2.5 第１の心配事：オーバーライドし忘れ

未来の開発者が Hero や Wizard など具体的な職業のクラスを作る際に attack() のオーバーライドを忘れてしまうと、重大な不具合に直結します。たとえば Hero クラスを作る際、attack() メソッドをオーバーライドし忘れたとします。

リスト 12-4

Hero.java
```
1 public class Hero extends Character {
2 }
```
attack() をオーバーライドすべきなのにしていない

Main.java
```
1 public class Main {
2   public static void main(String[] args) {
3     Hero h = new Hero();
4     Matango m = new Matango();
5     h.attack(m);
6   }
7 }
```
メソッドは呼び出せるが…

12章

Heroクラスは親クラスであるCharacterから**内容が空のattack()メソッド**を受け継いでいます。そのため、main()メソッドなどから「attackメソッドを**呼び出せるが、何も起きない**」という不具合を抱えたクラスになってしまいます。

main()メソッドでは「当然、何か攻撃してくれるんだろう」と思ってattack()を呼び出すのに、何もしてくれないなんて…。

エラーは出ないけど「想定外の変な動きをする」というタチが悪い不具合だ。コンパイル時や実行時にエラーが起きてくれたほうが、誤りに気づけるからまだマシなんだけどね。

ボクが作ったCharacterクラスを利用する開発者の人たちが、こんな不具合に苦しんでほしくないなぁ…。

解決策の1つとして、Characterクラスのattack()メソッドを作成する際、次のようにコメントを残しておくという方法があります。

```
1  // 未来の開発者さまへ
2  // 私はCharacterクラス開発者のミナトです。
3  // このクラスを開発している時点では、将来このクラスを継承して
4  // 作られるそれぞれの職業クラスが何ポイントのダメージを与えるか
5  // を確定できないため、以下のメソッドは中身を空にしてあります。
6  // Characterクラスを継承して様々な職業クラスを作る際には、
7  // attack()の中身を必ずオーバーライドして使ってください。
8  public void attack(Slime s) {
9  }
```

しかし、Characterクラスを継承する未来の開発者が、このコメントを見逃したり無視したりする可能性は残ります。仮に無視しなかったとしても、次のようなミスを犯してしまう可能性はあります。

リスト 12-5

Hero.java

```
1  public class Hero extends Character {
2    public void atack(Matango 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    }
6  }
```

attack の t が 1 文字足りずオーバーライドになっていない！

そっか。このクラスは継承してきた空の attack() と自分自身で定義した atack() の 2 つのメソッドを持ってしまうんですね。

そうだ。Hero クラスの作者は、「自分がちゃんと attack() をオーバーライドして空の処理を上書きした」と思い込んでしまっているだろう。

何も知らずに、main() メソッドとかから Hero の attack() を呼び出しちゃったら…と思うとゾッとしちゃいます。

12.2.6 第 2 の心配事:「本当に何もしない」と区別がつかない

Character クラスの attack() メソッドを、もう一度よく見てください。次のようになっているはずです。

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

そもそも、この書き方は「呼ばれても何もしない」メソッドを作りたい場合に行うものです。しかし今回の場合、attack() は**「何もしない」のではなく、「何をするかが未定で記述できない」**のです。

未来の開発者がこのメソッドを見たときに、「何もしないのが正しい」のか、それとも「何をするか未定」なのか、**区別がつかないおそれがある**のです。

そういえば、「詳細未定」欄だらけの菅原さんの年間営業計画ですが…もし空白のままだったら「未定」か「何もやらない（決定）」かが区別つきませんね…。ボクも見習わせていただきます（笑）。

そんなところはマネしなくていいから！

12.2.7 第3の心配事：意図せずnewして利用されてしまう

ここまでCharacterクラスに関する2つの心配事について考えました。それらはいずれも「詳細未定メソッド」に関係した心配事でしたが、まったく別の観点からの心配事がもう1つあります。それは、**「未来の開発者が間違ってCharacterクラスをnewして利用してしまうかもしれない」という心配**です。

たとえば、プロジェクトに新しく入ったJava初心者のDさんに、あなたが「便利なクラスだからどうぞ使ってください」と言ってCharacterクラスを渡したら、Dさんは次のようなコードを書いてしまうかもしれません。

リスト 12-6

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Character c = new Character();
4     Matango m = new Matango('A');
5     c.attack(m);
6   }
7 }
```

オーバーライドされていないので何も動かない！

HeroやWizardではなくCharacterをnewしてしまった！

いやいやいや！ そもそも使い方が完全に間違っていますよ！ この Character クラスは継承の材料として使われるべきものであって、Hero や Wizard みたいに new して使うためのものじゃないんです！！

そのとおり。ただ、Character クラスが作られた経緯をよく知らない人や Java 初心者は、間違って new しようとするかもしれないね。

実は、この Character クラスから実体であるインスタンスが生み出されたり、そのインスタンスが仮想世界の中で活動してしまったりすることはかなりの異常事態です。なぜなら、詳細未定な attack() メソッドを含む Character クラスは「詳細未定につき、作りこんでない部分が残っている未完成な設計図」のようなものだからです。

Character クラスの例に限らず、「未来の開発者のために準備しておくクラス」は、多かれ少なかれ未完成な部分が残っているものです。そのような設計図に基づいて、実体である製品（たとえば車）を生産・利用したら大変な事故につながることは容易に想像できるでしょう。

そもそも「一部でも未完成部分が残っている設計図から、実体を生み出してはならない」のです。

そもそも new されるべきではないクラス

Character のように、「詳細未定」な部分が残っているクラスはインスタンス化されてはならない。

12.2.8　第3の心配事の原因

それではなぜ、第3の心配事（間違って new されてしまう心配）が出てきてしまったのでしょうか？　今まであまり意識することはありませんでしたが、そもそもクラスには2つの「利用の仕方」があります。

① new による利用　　：インスタンスを生み出すために、そのクラスを利用する。
② extends による利用：別のクラスを開発する際、ゼロから作ると効率が悪いので、あるクラスを継承元として利用する。

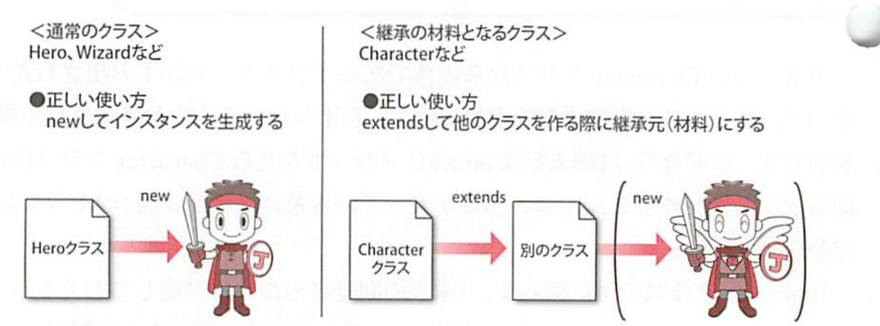


図 12-8　クラスの2つの利用方法

そして Hero や Wizard は new するためのクラスとして、また Character は extends するためのクラスとしてそれぞれ開発されています（図 12-8）。

しかし残念ながら、Character クラスの作者が「extends して利用してほしい」と願っても、未来の開発者は「new による利用」と「extends による利用」のどちらも選べてしまいます。ですから Character のような未完成なクラスが誤って new されてしまうという事態が起きるのです。

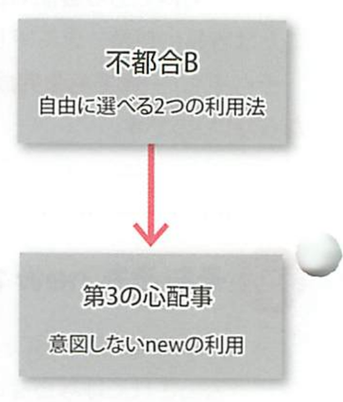


図 12-9　第3の心配

「クラスには自由に選べる2つの利用法がある」という利点が、皮肉にも「意図せず new される」という心配の原因になっています。

このような過ちが起こることがないように、ソースファイルの先頭にコメント

を書くというアイデアも考えられます。

Character.java

```
1  /*
2   * 未来の開発者さまへ
3   * 私はCharacterクラス開発者、ミナトです。
4   *
5   * このクラスは、普通のクラスのようにnewして使うためのものでは
6   * ありません。 HeroやWizardなどの職業クラスを皆様が作る際に
7   * 少しでもラクができるように、全職業クラスに共通するフィールド
8   * やメソッドをそなえた「継承の材料」です。
9   *
10  * このクラスを継承して、必要なフィールドやメソッドを追加する
11  * ことで、それぞれの職業クラスを完成させてください。
12  * 逆に言えば、このCharacterクラスは、それ自体では未完成の
13  * クラスです。たとえばattack()メソッドは中身が確定できないので
14  * 空にしてあります。
15  * よって、このクラスをnewして実際に利用（冒険させたり
16  * 戦闘させたり）しないでください。不具合の原因になります。
17  */
18 public class Character {
19 }
```


それでも、このようなコメントも読み落とされたり無視されたりする可能性があります。どうすればよいのでしょうか？

ボクの作ったCharacter クラスが、未来の開発者によって正しくない使われ方をされて不具合の原因にならないか不安ですよ…。

しかし、湊くんの心配は無用です。なぜなら Java には、これまでに見てきた3つの心配事を解決するしくみがあるからです。次の節で、それを学んでいきましょう。

12.3 抽象クラス

12.3.1 安全な「継承の材料」を実現するために

前節では、継承の材料となるクラス（Character クラスなど）を作ろうとすると問題になる 2 つの不都合と 3 つの心配事について考えてみました。

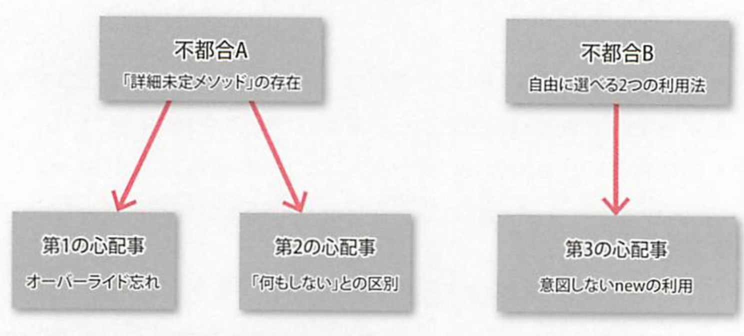


図 12-10　2 つの不都合と 3 つの心配事

そして Java には、この 3 つの心配事を解決するしくみが準備されています。この節では、まず第 2 の心配事と第 3 の心配事、そして第 1 の心配事の順に解決の方法を見ていきましょう。

12.3.2 詳細未定メソッド専用の書き方

まずは第 2 の心配事から解決していきましょう。

第 2 の心配事

空のメソッドを作っておくと、「現時点で処理内容を確定できないメソッド」なのか「何もしないメソッド」なのか、区別がつかない。

実は Java には、「詳細未定メソッド」を記述する専用の構文が準備されています。

詳細未定メソッド（抽象メソッド）の宣言

アクセス修飾子 abstract 戻り値 メソッド名（引数リスト）;

たとえばCharacterクラスのattack()メソッドは次のように書きます。

リスト 12-7

Character.java

```
1  public class Character {
2    ⋮
3    public abstract void attack(Matango m);
4    ⋮
5  }
```

abstract（アブストラクト）とは「抽象的・あいまい」という意味の英単語です。これをメソッドの宣言に付けることで、「attack()というメソッドは少なくとも宣言すべきなのですが、具体的にどう動くか、内容がどうなるかまでは現時点では確定できないので、メソッド内部の処理はここでは記載しない」という表明になります。

メソッドの処理内容は未定で記載できないわけですから、メソッド宣言の後ろにはブロック記号の{}さえ付けず、その代わりにセミコロンを書きます。abstract修飾子が付けられたメソッドは、抽象メソッド（abstract method）と呼ばれます。

第2の心配事は解決！

空メソッドは「何もしないメソッド」、抽象メソッドは「何をするか現時点で確定できないメソッド」と区別できる。

12章

12.3.3 未完成のため new してはいけないクラスの宣言

次に第 3 の心配事を解決しましょう。

第 3 の心配事

未完成部分を含む継承専用のクラスを誤って new される可能性がある。

Java では、「未完成部分（＝抽象メソッド）を 1 つでも含むクラス」は、次の構文に従って宣言しなければならないことになっています。

抽象メソッドを含むクラスの宣言

アクセス修飾子 abstract class クラス名 {
　⋮
}

たとえば、抽象メソッド attack() を持つ Character クラスを宣言するには、次のリストのように書きます。

リスト 12-8

Character.java

```
1 public abstract class Character {
2   String name;
3   int hp;
4   public void run() {
5     System.out.println(this.name + "は逃げ出した。");
6   }
7   public abstract void attack(Matango m);
8 }
```

抽象クラスとして Character を宣言

Java のルールで、抽象メソッドを含むクラスは必ず abstract 付きのクラスにしなければならない。もし 1 行目の abstract を忘れてしまったらコンパイルエラーになるよ。

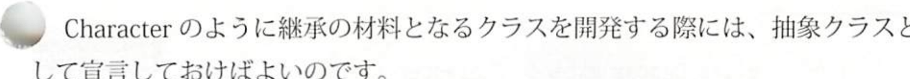
このように、abstract が付いたクラスは特に抽象クラスと呼ばれます。Character クラスを普通のクラスではなく抽象クラスとして宣言すると、次のような特殊な制約がかかります。

抽象クラスの制約

抽象クラスは、new によるインスタンス化が禁止される。

たとえば、抽象クラスとして宣言された Character をインスタンス化しようとする次のコードはコンパイルに失敗します。

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



```
エラー： Character は abstract です。インスタンスを生成することはできません。
```


Character のように継承の材料となるクラスを開発する際には、抽象クラスとして宣言しておけばよいのです。

第 3 の心配事も解決！

継承専用のクラスは抽象クラスとして宣言すれば、間違って new されることはない。

抽象メソッド（＝未完成部分）が1つでもあるクラスは、抽象クラスにしなければコンパイルが通らないことも併せて考えてみよう。

「詳細未定」の抽象メソッドがある→そのクラスは必ず抽象クラスになる→インスタンス化できない…。

「一部でも未定な部分がある設計図」から実体が生まれてしまうようなこと（12.2.7項）が絶対になくなるんですね！

12.3.4 オーバーライドの強制

最後は第1の心配事の解決です。

第1の心配事

未来の開発者が、詳細未定メソッドをオーバーライドし忘れる可能性がある。

この心配事は、実はすでに解決しています。「Character を継承して Dancer を作る新人開発者 D さんの立場」になって考えてみましょう。

Dancer は Hero や Wizard と同じように、new して実体を生み出し冒険させるためのクラスです。

リスト 12-9

Dancer.java

```
1  public class Dancer extends Character {
2
3    public void dance() {
4      System.out.println(this.name + "は情熱的に踊った");
5    }
6  }
```

Character は抽象クラス

attack() をオーバーライドし忘れている

Dさんは、Dancer特有の能力である「踊る(dance)」というメソッドの作成に気を取られてattack()のオーバーライドを忘れてしまいました。

しかし、このソースをコンパイルしようとすると「未完成状態のクラスであるDancerは、abstractを付けて抽象クラスにしなければならない」という意味のエラーメッセージが表示され、コンパイルは失敗します。

Dancerクラスの定義には1つも抽象メソッドがないから、抽象クラスにしなくてもいいハズなのに…。

いや、Dancerには隠れた「抽象メソッド」が潜んでいるんだよ。

ここで継承の基本を再確認しましょう。DancerクラスはCharacterクラスを親クラスとしますので、Characterクラスが持つすべてのメンバを継承しています。そして、親クラスから継承したメンバの中には、抽象メソッドattack()も含まれています。つまり、Dancerクラス自体のソースコードに抽象メソッドはなくても、親クラスから抽象メソッドを継承して持っているのです。抽象メソッドが1つでもある以上、Dancerクラスも抽象クラスにしなければコンパイルエラーが出て当然ですね。

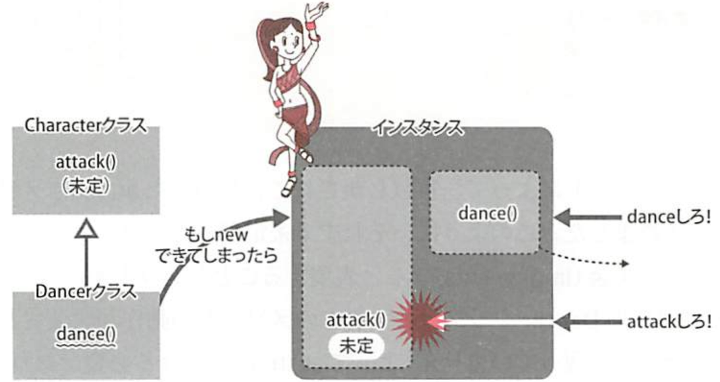


図12-11 Dancerクラス内に抽象メソッドattack()が含まれている

このエラーに対処する方法は2つあります。

① Dancer クラスの宣言に abstract を付けて抽象クラスにする。
② Dancer クラス内部の「未完成部分」をすべてなくす。

①の方法で解決を図ればコンパイルエラーを消すことはできます。しかし、抽象クラスとなってしまった Dancer は new できませんので、Hero や Wizard のようにインスタンスを生み出すことができません。

D さんが「Dancer を Hero や Wizard のようにインスタンス化して冒険に出せるクラスとして開発したい」ならば、残された選択肢は②だけになります。すなわち Dancer クラスの中で attack() をオーバーライドし、未完成メソッドをなくしてあげればよいのです（リスト 12-10）。

リスト 12-10

Dancer.java

```
1  public class Dancer extends Character {
2    public void dance() {
3      System.out.println(this.name + "は情熱的に踊った");
4    }
5    public void attack(Matango m) {
6      System.out.println(this.name + "の攻撃");
7      System.out.println("敵に3ポイントのダメージ");
8      m.hp -= 3;
9    }
10 }
```

親から継承した「詳細未定の attack()」を上書きする

このオーバーライドによって、宣言しかされていなかった attack() メソッドの動作が決定されました。このように、それまで未定だったメソッドの内容を確定させることを、**実装**（implements）すると表現することもあります。

リスト 12-10 の Dancer クラスは、すべてのメソッドの動作が実装されており「詳細未定」な部分は残っていません。よって abstract を付ける必要はありません。Dancer は **new して使える通常のクラス**になりました。

今回の Dancer の例を振り返ってみると、「抽象クラスは new できない」というルールが Java に備わっている以上、「あるメソッドを抽象メソッドとして宣言し

ておくことで、未来の開発者にオーバーライドを強制できる（オーバーライドしないと new して使えないから）」という効果があるとも言えます。

第 1 の心配事も解決！

詳細未定なメソッドを抽象メソッドとして宣言すれば、未来の開発者にオーバーライドを強制できる。

以上で 3 つの心配事がすべて解決しました。抽象クラスと抽象メソッドを用いることで、未来の開発者が安全かつ便利に利用できる「継承の材料」となるクラスを開発できるのです。

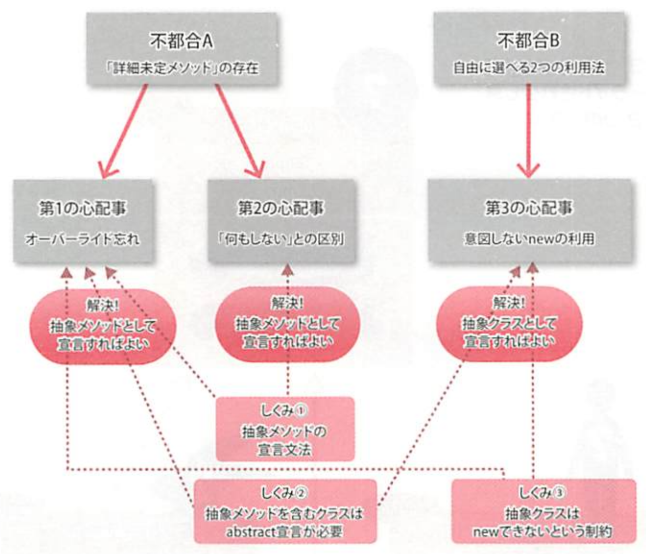


図 12-12　3 つの心配事の解決

12.3.5 多階層の抽象継承構造

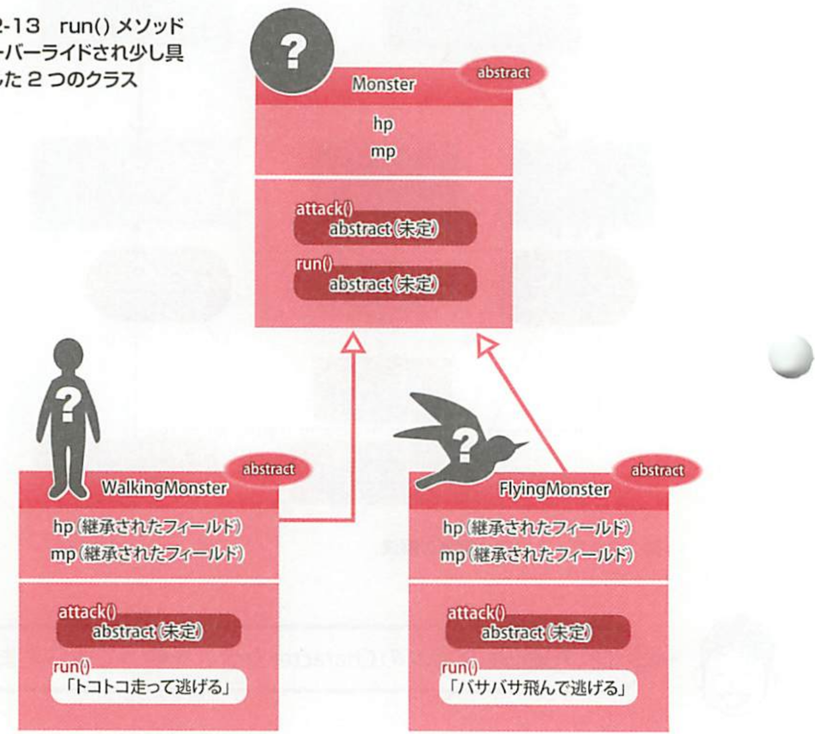
ここまでの解説で、「抽象メソッドには、未来の開発者が継承する際にオーバーライドを強制する効果があること」が理解できたと思います。

ちなみに、抽象クラスを継承した子クラスで「すべての抽象メソッドをオーバーライドしてメソッドの内容を実装する」必要があるわけではありません。

そのクラスでは確定できない抽象メソッドについては、必要に応じて、その孫クラス、あるいは曾孫クラスでオーバーライドして内容を確定させてもよいのです。その代わり、**すべての抽象メソッドの処理内容が確定しなければ abstract を外すことは許されず**、つまり **new して利用することはできません**。

たとえば、さまざまなモンスターたちの親クラスとして Monster クラスというものを考えましょう。Monster クラスは attack() と run() のメソッドを持っていますが、モンスターによって、どう攻撃するか、どう逃げるかについては、現時点ではわかりません。よって、両方とも abstract が付いた抽象メソッドです。

図 12-13　run() メソッドがオーバーライドされ少し具体化した 2 つのクラス



次に、もう少し具体的なMonsterを定義するとします。「WalkingMonster」は「トコトコ走って」、「FlyingMonster」は「バサバサ飛んで」逃げていくと決まれば、run だけはオーバーライドして内容を実装できます（図 12-13）。

しかし、WalkingMonster と FlyingMonster にはまだ抽象メソッドが残っています。attack() の詳細が未定なので、これら２つのクラスは共に抽象クラスとしなければならず、クラス宣言から abstract 宣言は外せません（つまり new できない）。

もし、WalkingMonster の子として Goblin クラスを定義し、このクラスでは attack() をオーバーライドすると、ここでやっと抽象メソッドがなくなります。Goblin クラスは abstract 宣言を付ける必要はなくなり、通常のクラスとして new して利用することが可能になります。同様に FlyingMonster の子として

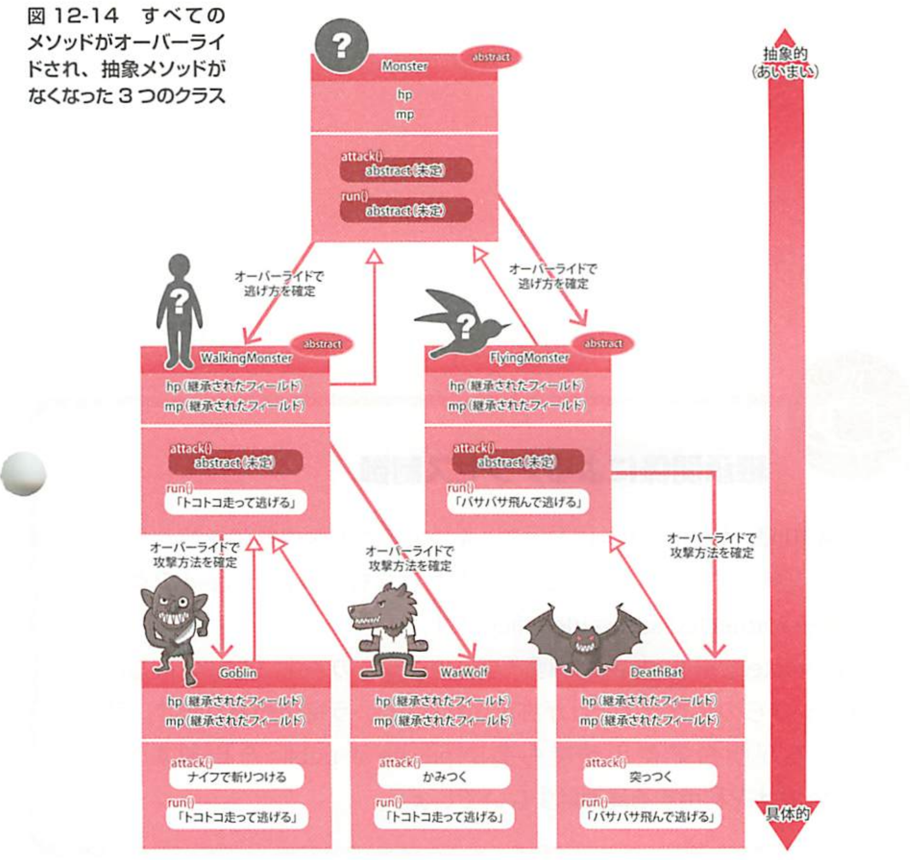


図 12-14　すべてのメソッドがオーバーライドされ、抽象メソッドがなくなった３つのクラス

DeathBatを定義し、このクラスでattack()をオーバーライドすると抽象メソッドはなくなり、これもnewして利用できるようになります(図12-14)。

この継承図を眺めると、継承が繰り返されるたびに具体化していくことがわかります。Monsterというクラスは大変にあいまいで「HPとMPがある」程度しか決められていません。

もう少し具体化したWalkingMonsterやFlyingMonsterでは、逃げる処理の内容が確定します。さらに具体化したGoblinやDeathBatは、攻撃の方法も確定し、あいまいさがまったくありません。

第11章の最後でも述べたように、継承を正しく用いたJavaのクラスは、継承階層を降りていくほどに具体的になり、メソッドの処理内容が実装されていきます。

継承関係によるアクセス制御

第10章「カプセル化」において、4つのアクセス修飾子を紹介しましたが、その中で簡単にしか触れていなかったものが**protectedアクセス修飾子**(protected access identifier)です。

protectedが付いたメンバは、「**自分のクラスの子孫、または、同じパッケージからのアクセスだけが許可される**」という特徴があります。使いどころが明白で使用頻度も高いprivateやpublicと比較すると、protectedを利用する局面は少ないでしょう。

12.4 インタフェース

12.4.1 抽象階層を上に辿ると…

前節の最後では、「継承階層を下に辿っていくとどうなるか」を見ていきました。継承階層が下がっていくたびにクラスは具体化してゆき、最終的にはメソッドの処理内容が実装されていくのでしたね。では、今度は逆に、階層を下から上に昇ってみましょう。

以下の条件に沿って Monster クラスの親クラスを作っていきます。

① Monster と Character の共通の親として戦闘に参加する動物（BattleCreature）を定義します。戦闘に参加する動物の中には、専守防衛的な動物もいるかもしれませんので attack() は定義できません。

② BattleCreature の親として動物（Creature）を定義します。これは村人やお姫様のように戦闘に参加しない生き物も含んでいるため、HP フィールドはあるとは限りませんが、どのような動物であっても脅威から逃げるための run() は持っています。

Goblin であれば HP、MP、名前、攻撃力などのフィールドと、attack() や run()、useItem() などの内容が確定した具体的なメソッドを備えているでしょう。しかし Creature のようにあいまいになると、攻撃力や useItem() はおろか、名前さえも持っているとは限りません。**もはや「逃げる run() メソッドぐらいは最低でも持っている」ということしか決められない**のです。

この例に限らず、正しく継承が用いられている継承ツリーを上へ辿ると、次のような現象が順に起こります（次ページの図 12-15）。

①抽象メソッドが増える

「内容は確定できないが、一応存在する」という抽象メソッドが現れ始めます。

②抽象メソッドやフィールドが減っていく

クラスに定義してある抽象メソッドやフィールドが減っていきます。

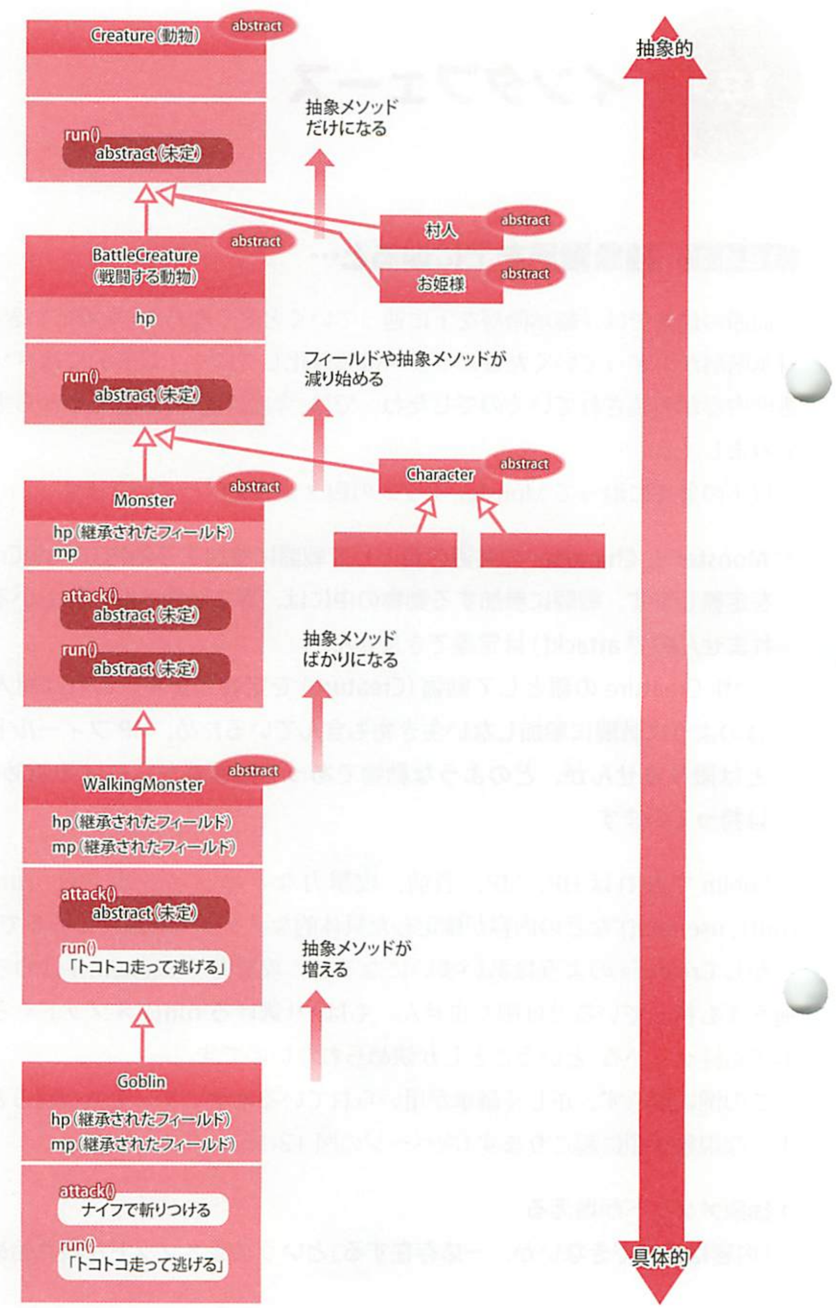


図 12-15　抽象継承階層を上に辿っていくと…

継承階層を上に辿るということは、「どんどんあいまいなものになっていく」ということです。クラスがあいまいになるにつれ、「どのような内部情報を持っているか（フィールド）」「どのような動きをするか（メソッド）」は、あやふやになり、決めることができなくなっていくのです。

12.4.2 抽象クラスの特別扱い

Creature クラスぐらいになると、ものすごくあいまいで「抽象クラスの中の抽象クラス」みたいな感じがしてきますね。

そうだね。そして、そんな「抽象クラスの中の抽象クラス」だけを特別扱いする文法が Java にはあるんだよ。

ここまで見てきたように、継承階層を上に辿ると、上流のクラスはすべて抽象クラスになります。そして Java では、次の条件を満たす「特に抽象度が高い抽象クラス」を、**インタフェース**（interface）として特別に扱うことができます。

インタフェースとして特別扱いできる 2 つの条件

①すべてのメソッドは抽象メソッドである。

②基本的にフィールドを 1 つも持たない。

たとえば次に示す抽象クラス Creature のコードを見てください。

リスト 12-11

Creature.java

```
1  public abstract class Creature {
2    public abstract void run();
3  }
```

このクラスには抽象メソッドしかなく、フィールドもありません。このまま抽象クラスとしておいてもよいのですが、以下のような構文を用いてインタフェースとして定義することも可能です。

インタフェースの宣言

```
アクセス修飾子 interface インタフェース名 {
    ⋮
}
```

では、インタフェースとして宣言したCreatureを見てみましょう。

リスト 12-12

Creature.java

```java
public interface Creature {
  public abstract void run();
}
```

なお、「インタフェースに宣言されたメソッドは、自動的にpublicかつabstractになる」というルールがあるので、通常は次のように書きます。

リスト 12-13

Creature.java

```java
public interface Creature {
  void run();
}
```

public abstractを省略しても大丈夫

初めてインタフェースという用語を聞いたときは、クラスとはまったく関係ない新しい何かだと思っていました。

でも、実は「クラスの仲間」で、「抽象クラスの親戚みたいなもの」なんですね。

そうだよ。初めのうちは難しく考えすぎないで、「あまりにあいまいすぎて特別扱いされた抽象クラス」と理解しておけばいいよ。

インタフェースにおける定数宣言

先ほどの「インタフェースとして特別扱いできる2つの条件」によれば、インタフェースは基本的にフィールドを持ちません。しかし、「public static final が付いたフィールド（定数）」だけは宣言が許されます。

さらにそのようなフィールドを宣言する場合は、「public static final」を省略してもよいことになっています。つまり、インタフェース内でフィールドを宣言すると自動的に public static final が補われ、定数を宣言したことになるのです。

次のコードは、円周率 PI をインタフェースに定義した例です。

Circle.java

```
1  public interface Circle {
2    double PI = 3.141592;
3  }
```

自動的に public static final が補われる

12.4.3 インタフェースの名前の由来

特にあいまいな抽象クラスを特別扱いするのはわかりましたけど、なぜ「インタフェース」という新しい名前なんですか？「スーパー抽象クラス」とかでいいんじゃないのかな？

なぜ２つの条件を満たした抽象クラスに「インタフェース」という、まったく新しい別の名前が付いているのでしょうか。その理由を探るため、次のインタフェースを見て意味を考えてみましょう。

リスト 12-14

CleaningService.java

```
public interface CleaningService {
  Shirt washShirt(Shirt s);
  Towl washTowl(Towl t);
  Coat washCoat(Coat c);
}
```

この CleaningService はシャツとタオル、そしてコートを渡せば、それを洗って返してくれます。しかし布団やマフラーは扱っていないようですね。また、すべてのメソッドは抽象メソッドであり、処理内容が記述されていません。つまり、**「どのようにして洗うか」というクリーニング店の内部で行われる作業については明かされていない**わけです。この CleaningService インタフェースは、まるでクリーニング店の店頭メニューのようですね。

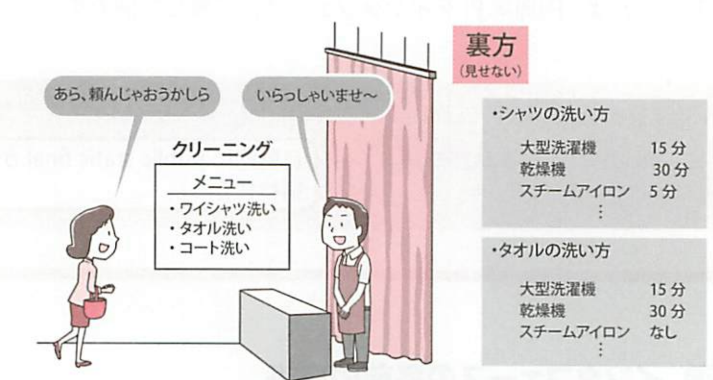


図 12-16　クリーニング屋のメニューには、どのように洗うかまでは書いていない

店頭メニューは、クリーニング店が「こういう仕事を受け付けますよ」と表明するためのものです。そしてお客さんはメニューを見て、「この仕事をお願いします」と依頼をします。

つまり、メニューは店主とお客さんとの接点(英語で「interface」といいます)の役割を果たしているのです。

12.4.4 インタフェースの実装

CleaningServiceインタフェースが店頭メニューだとすれば、それを継承して次のように記述したKyotoCleaningShopクラスこそが「クリーニング店そのもの」といえるでしょう。

リスト 12-15

KyotoCleaningShop.java

```
1  public class KyotoCleaningShop implements
   CleaningService {
2    private String ownerName;
3    private String address;
4    private String phone;
5    /* シャツを洗う */
6    public Shirt washShirt(Shirt s) {
7      // 大型洗濯機15分
8      // 業務用乾燥機30分
9      // スチームアイロン5分
10     return s;
11   }
12   /* タオルを洗う */
13   public Towl washTowl(Towl t) {
14     ︙
15   }
16   /* コートを洗う */
17   public Coat washCoat(Coat c) {
18     ︙
19   }
20 }
```

店主の名前

住所

電話番号

インタフェースを継承しクラスを宣言する場合はimplements

12章

KyotoCleaningShop クラスの1行目にあるように、インタフェースを継承して子クラスを定義する場合は extends ではなく **implements** を使います。これを「CleaningService インタフェースを**実装して** KyotoCleaningShop を作る」などと表現します。

インタフェースの実装

```
アクセス修飾子 class クラス名 implements インタフェース名 {
    ⋮
}
```

なお、インタフェースという名前であっても、「しょせんは抽象クラスみたいなもの」ということを思い出してください。インタフェースで定義された washShirt()、washTowl()、washCoat() は、すべて抽象メソッドですので、子クラスである KyotoCleaningShop で、それぞれオーバーライドしなければなりません。

なお、図 12-17 で示したように、クラス図ではインタフェースの実装を点線の矢印記号で表します。

図 12-17　インタフェースの実装

実装する（implements）という用語が使われるのは、**親インタフェースで未定だった各メソッドの内容をオーバーライドして実装し確定させるから**だよ。

ところで、全国チェーンのクリーニング店では、どの店でも同じ店頭メニューを使っていることがあります。おそらく本社で作ったメニューを、すべての店で掲示しているのでしょう。

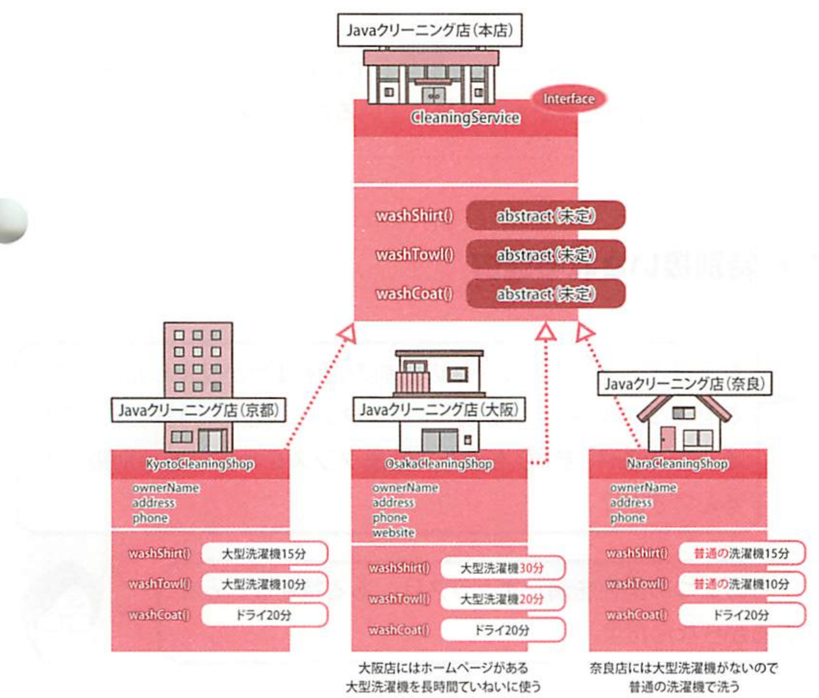


図 12-18　1 つのインタフェースを実装する複数のクラス

チェーン店とはいえ、京都店・大阪店・奈良店は別の店ですので、それぞれの店が持つ設備や洗濯の手順もさまざまでしょう。つまり、共通の CleaningService を実装していたとしても、個々の～ CleaningShop クラスのフィールドやメソッドの詳細は異なっても構いません。

しかし、このクリーニングチェーンの加盟店は、共通の店頭メニューを出している以上、どの店も「シャツ・タオル・コートの洗濯」はできる必要があります。個々の～ CleaningShop クラスは、CleaningService インタフェースで定められた抽象メソッドをオーバーライドして処理を実装している必要があるのです。

このように考えると、あるインタフェースに複数のメソッドを定義しておくことは、次のような 2 つの効果を生み出すと考えることができます。

インタフェースの効果

①同じインタフェースをimplementsする複数の子クラスたちに、共通のメソッド群を実装するよう強制できる。
②あるクラスがインタフェースを実装していれば、少なくともそのインタフェースが定めたメソッドは持っていることが保証される。

12.4.5 特別扱いされる理由

なるほど、インタフェースの名前の由来はわかりました。でも、なぜ2つの条件（12.4.2項「インタフェースとして特別扱いできる2つの条件」参照）を満たす抽象クラスをわざわざ特別扱いするんですか？

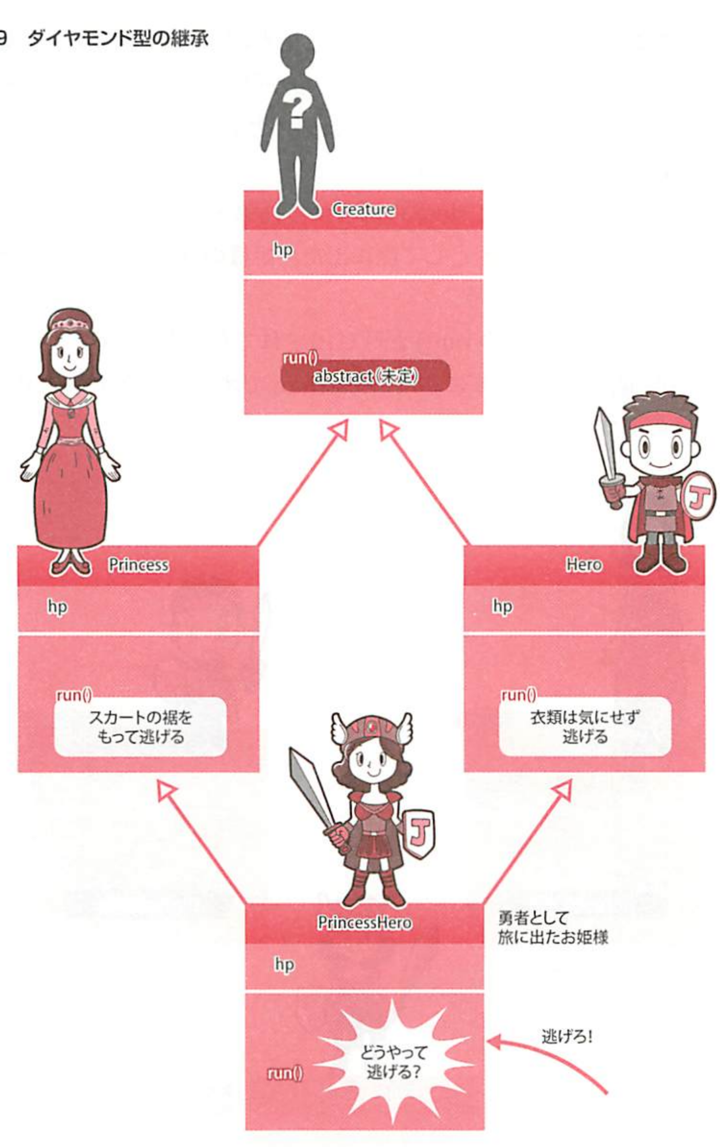
それは、この2つの条件を満たすクラスは、ある特別なことを実現できるからなんだよ。

クリーニング店の例で見たように、インタフェースは「このようなメソッド群を持ち、このような引数を与えれば、このような結果を返す」という表面的な確約をするだけで、その内部実装（メソッドの処理動作）をいっさい定めていません。インタフェースが特別扱いされるのは、この「内部実装をいっさい定義しない」という性質があるからです。この性質のおかげで、インタフェースでは特別に多重継承が許されています。

多重継承は第11章（11.1.5項）で少し触れたように、あるクラスを作成する際に2つの親クラスを使うことができる、とても便利な機能です。しかし、多重継承は誤用されやすく危険なので、Javaでは基本的に「クラスの多重継承」は禁止されました。

なぜ多重継承が危険で禁止されたのかを次の例で考えてみましょう。

図 12-19　ダイヤモンド型の継承



PrincessHeroは、PrincessとHeroの両方からメソッドを継承しています。このとき、PrincessHeroのrun()メソッドが呼び出されたら、このキャラクターはどのように逃げるのでしょうか？　「スカートの裾を持って」逃げるのでしょうか？　それとも「衣類を気にせず」逃げるのでしょうか？

このように、多重継承を用いると、「両方の親から同じ名前でありながら異な

る内容の2つのメソッドを継承してしまう」ことが起こりえるため、「お姫様としての逃げ方」と「勇者としての逃げ方」のどちらが動くべきなのか混乱を招いてしまいます。しかし、これがHeroインタフェースとPrincessインタフェースからの多重継承ならば、どうでしょうか？

PrincessHeroはPrincessとHeroの両方から抽象メソッドであるrun()を継承しますので、これを必ず「勇者として旅に出たお姫様の独自の逃げ方」でオーバーライドすることになるでしょう。

この場合、PrincessHeroのrun()が呼び出されても、先ほどの「クラスの多重継承」のような混乱は起こらずに、PrincessHeroでオーバーライドし定義されたrun()が動くのです。

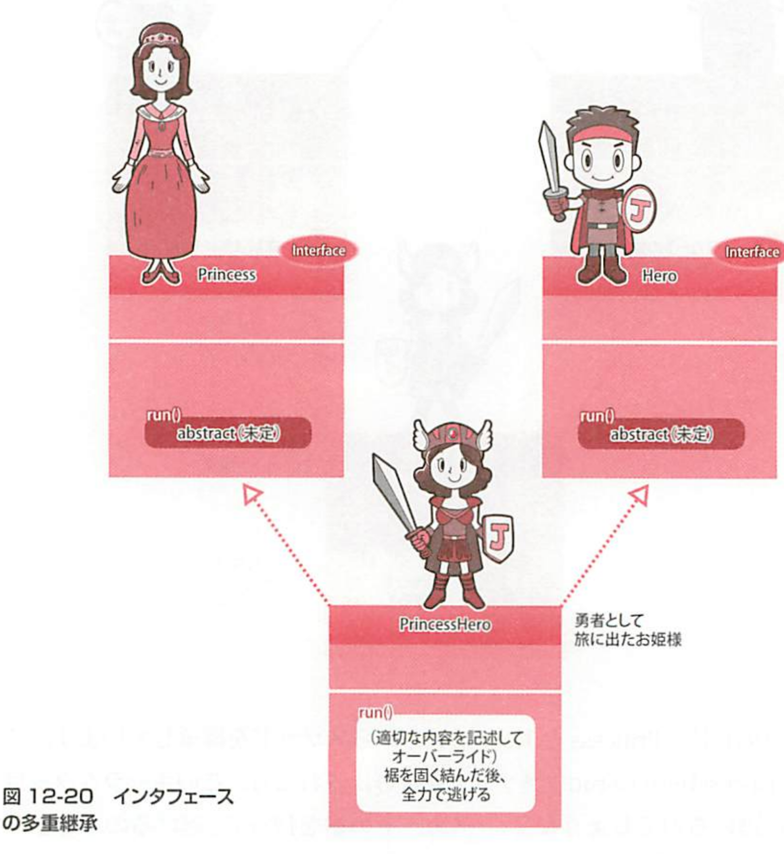


図12-20　インタフェースの多重継承

そもそも多重継承が問題なのは、「両方の親クラスから同じ名前でありながら**異なる内容**のメソッドを継承して衝突してしまう」からです。しかし、両方の親がインタフェースの場合、**どちらもメソッドの内容をいっさい定めていません**から、「親から継承した2つの処理内容が衝突する」ことは起こりえないのです。

クラスにはないインタフェースの特権

異なる実装が衝突する問題が発生しないため、複数の親インタフェースによる多重継承が認められている。

インタフェースによる多重継承は、次のような構文で行います。

インタフェースによる多重継承

```
アクセス修飾子 class クラス名
      implements 親インタフェース名 1,
親インタフェース名 2,… {
      ⋮
}
```

次の例のように3つ以上のインタフェースを実装することも可能です。

PrincessHero.java

```
1  public class PrincessHero
       implements Hero, Princess, Character {
2    ⋮
3  }
```

これら3つはすべてインタフェースとします

12.4.6 インタフェースの継承

ところで、あるインタフェースを定義する場合、ゼロから開発せずに既存のインタフェースを拡張することもできます。

リスト 12-16

Human.java

```
1  public interface Human extends Creature {
2    void talk();
3    void watch();
4    void hear();
5    // さらに、親インタフェースからrun()を継承
6  }
```

あれ？ インタフェースを継承するときは extends ではなく implements を使うはずじゃ…。

今回の場合、Human は Creature の run() メソッドをオーバーライドして**処理の内容を確定しているわけではありません**ので、implements ではなく extends を使います。混乱しやすい部分かもしれません。

implements と extends の使い分けは次ページの表 12-1 のとおりです。

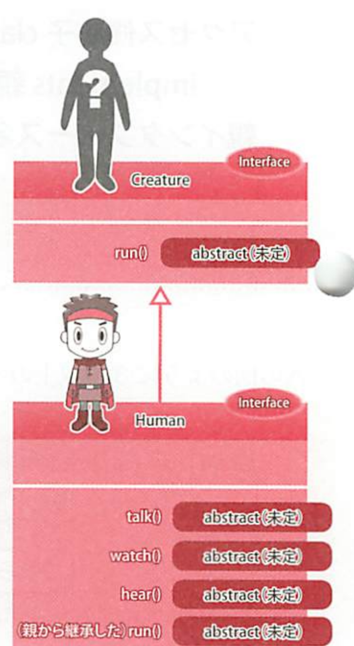


図 12-21　インタフェースの拡張

表 12-1 implements と extends の使い分け

継承元	継承先	使用するキーワード	継承元の数
クラス	クラス	extends	1つ
インタフェース	クラス	implements	1つ以上
インタフェース	インタフェース	extends	1つ以上

「同種（クラス同士、インタフェース同士）の継承の場合は extends を使う」「異種なら implements」と覚えておこう。

12.4.7 extends と implements を一緒に使う

クラス定義の際に extends と implements の両方を利用することもあります。

extends と implements の両方を使ったクラス定義

```
アクセス修飾子 class クラス名 extends 親クラス
        implements 親インタフェース 1, 親インタフェース 2,
… {
    ⋮
}
```

たとえば次のような使い方をします（リスト 12-17 および図 12-22）。

リスト 12-17

Fool.java

```
1  public class Fool extends Character implements Human {
2    // Characterからhpやget Name()などのメンバを継承している
3    // Characterから継承した抽象メソッドattack()を実装
4    public void attack(Matango m) {
```

```
5        System.out.println
              (this.getName() + "は、戦わず遊んでいる。");
6      }
7      // さらにHumanから継承した4つの抽象メソッドを実装
8      public void talk() { … }
9      public void watch() { … }
10     public void hear() { … }
11     public void run() { … }
12   }
```

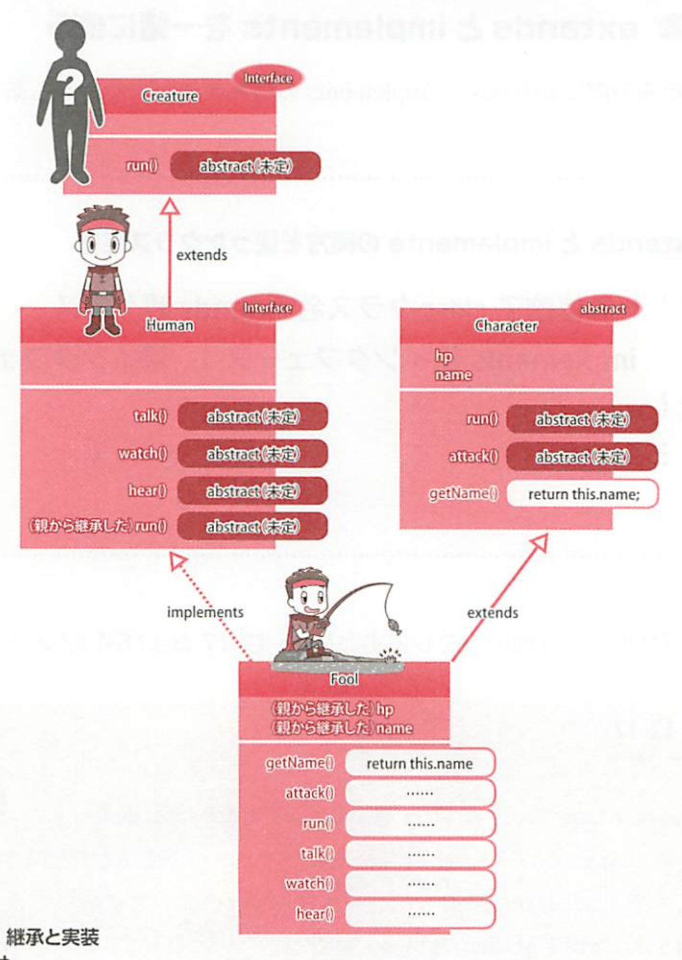


図 12-22　継承と実装の組み合わせ

インタフェースメソッドのデフォルト実装

12.4.1 項で紹介したように、インタフェースが持つことのできるメソッドは処理内容を持たない抽象メソッドに限られます。しかし Java8 以降では、default キーワードを用いることで、処理のデフォルト実装を添えた抽象メソッドを定義できるようになりました。

デフォルト実装付き抽象メソッドの宣言

```
default 戻り値 メソッド名 ( 引数 ) {
  処理のデフォルト実装
}
```

このようにして定義された抽象メソッドは、もし継承先でオーバーライドされなかった場合、自動的に、デフォルト実装として定めた処理内容でオーバーライドされたものと見なされます。

上手に使うとオーバーライドの手間を省くことができる便利な機能ですが、多重継承によるデフォルト実装の衝突を招くこともある点には注意が必要です。

12.5 第12章のまとめ

この章では、次のようなことを学びました。

継承の材料を作る開発者の立場と役割

- 「他の人が継承の材料として使うであろう親クラスを作る立場」の開発者も存在する。
- 「未来の開発者が効率よく安心して利用できる継承の材料を作ること」がその使命。
- その使命を達成するために、Java では抽象クラスやインタフェースという道具を提供している。

抽象クラス

- 中身を決定できない「詳細未定メソッド」には abstract を付けて抽象メソッドとする。
- 抽象メソッドを 1 つでも含むクラスは、abstract を付けた抽象クラスにしなければならない。
- 抽象クラスはインスタンス化することが禁止されている。
- 抽象クラスと抽象メソッドを活用した「継承の材料」としての親クラスを開発すれば、予期しないインスタンス化やオーバーライド忘れの心配がない。

インタフェース

- 抽象クラスのうち、基本的に抽象メソッドしか持たないものを「インタフェース」として特別扱いできる。
- インタフェースに宣言されたメソッドは自動的に public abstract となり、フィールドは public static final になる。
- 複数のインタフェースを親とする多重継承が許されている。
- インタフェースを親に持つ子クラスの定義には implements を用いる。

12.6 練習問題

問題 12-1

ある会社では、会社の資産として保有するものを管理するプログラムを作ろうとしています。現時点では、「コンピュータ」「本」を表す、次のような２つのクラスがあります。

Book.java

```
1  public class Book {
2    private String name;
3    private int price;
4    private String color;
5    private String isbn;
6    // コンストラクタ
7    public Book
         (String name, int price, String color, String isbn) {
8      this.name = name;
9      this.price = price;
10     this.color = color;
11     this.isbn = isbn;
12   }
13   // getterメソッド
14   public String getName() { return this.name; }
15   public int getPrice() { return this.price; }
16   public String getColor() { return this.color; }
17   public String getIsbn() { return this.isbn; }
18 }
```

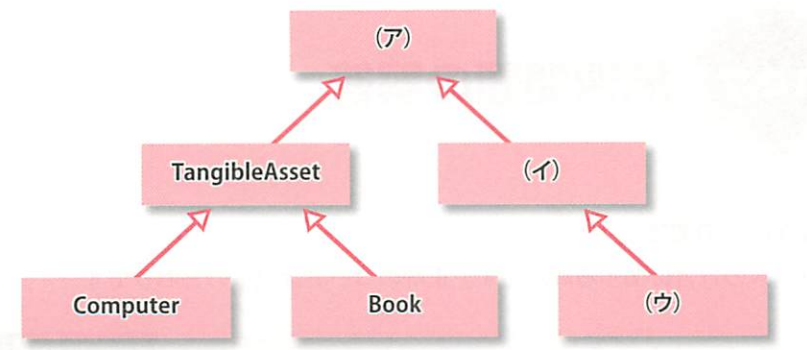
Computer.java

```
1  public class Computer {
2    private String name;
3    private int price;
4    private String color;
5    private String makerName;
6    // コンストラクタ
7    public Computer
         (String name, int price, String color, String makerName) {
8      this.name = name;
9      this.price = price;
10     this.color = color;
11     this.makerName = makerName;
12   }
13   // getterメソッド
14   public String getName() { return this.name; }
15   public int getPrice() { return this.price; }
16   public String getColor() { return this.color; }
17   public String getMakerName() { return this.makerName; }
18 }
```

今後、コンピュータと本以外にも、さまざまな形ある資産を管理していきたい場合に有用な「有形資産（TangibleAsset）」という名前の抽象クラス（継承の材料）を作成してください。また、Computer や Book は、その親クラスを用いた形に修正してください。

問題 12-2

問題 12-1 の会社では、形のない無形資産（IntangibleAsset）も管理しようと考えています。無形資産には、たとえば特許権（Patent）などがあります。無形資産も有形資産も資産（Asset）の一種です。この前提に従い、次の継承図の（ア）～（ウ）にあてはまるクラス名を考えてください。



また、(ア)に入る抽象クラスを開発し、このクラスを継承するように TangibleAsset を修正してください。

問題 12-3

資産かどうかとは関わりなく、形がある物(Thing)であれば、「重さ」を得ることができるはずです。そこで、double 型で重さを取得する getter メソッド getWeight() と setter メソッド setWeight() を持つインタフェース Thing を定義してください。

問題 12-4

有形資産(TangibleAsset)は、資産 (Asset) の一種でもありますし、形ある物(Thing)の一種でもあります。この定義に沿うように TangibleAsset のソースコードを修正してください。この際、TangibleAsset にフィールドやメソッドの追加が必要であれば、適宜追加してください。

12.7 練習問題の解答

問題12-1の解答

クラスの宣言に関する問題です。正解のコードは以下のとおりです。

TangibleAsset.java

```
1  public abstract class TangibleAsset {
2    private String name;
3    private int price;
4    private String color;
5    public TangibleAsset(String name, int price, String color) {
6      this.name = name;
7      this.price = price;
8      this.color = color;
9    }
10   public String getName() { return this.name; }
11   public int getPrice() { return this.price; }
12   public String getColor() { return this.color; }
13 }
```

Book.java

```
1  public class Book extends TangibleAsset {
2    private String isbn;
3    public Book
         (String name, int price, String color, String isbn) {
4      super(name, price, color);
5      this.isbn = isbn;
6    }
7    public String getIsbn() { return this.isbn; }
8  }
```

Computer.java

```
1  public class Computer extends TangibleAsset {
2    private String makerName;
3    public Computer
         (String name, int price, String color, String makerName ) {
4      super(name, price, color);
5      this.makerName = makerName;
6    }
7    public String getMakerName() { return this.makerName; }
8  }
```

練習 12-2の解答

（ア）Asset　（イ）IntangibleAsset　（ウ）Patent

Asset.java

```
1  public abstract class Asset {
2    private String name;
3    private int price;
4    public Asset(String name, int price) {
5      this.name = name;
6      this.price = price;
7    }
8    public String getName() { return this.name; }
9    public int getPrice() { return this.price; }
10 }
```

TangibleAsset.java

```
1  public abstract class TangibleAsset extends Asset {
2    private String color;
3    public TangibleAsset(String name, int price, String color) {
4      super(name, price);
5      this.color = color;
6    }
7    public String getColor() { return this.color; }
```

```
8 }
```

練習12-3の解答

メソッドの宣言に関する問題です。以下は解答例のコードです（おおむね合っていれば正解とします）。

Thing.java

```
1  public interface Thing {
2    double getWeight();
3    void setWeight(double weight);
4  }
```

練習12-4の解答

以下は解答例のコードです（おおむね合っていれば正解とします）。

TangibleAsset.java

```
1  public abstract class TangibleAsset extends Asset
       implements Thing {
2    private String color;
3    private double weight;
4    public TangibleAsset(String name, int price, String color) {
5      super(name, price);
6      this.color = color;
7    }
8    public String getColor() { return this.color; }
9    public double getWeight() { return this.weight; }
10   public void setWeight(double weight){ this.weight = weight; }
11 }
```