第 10 章

カプセル化

第 8 章や第 9 章で学んだ文法を用いれば、
クラスやインスタンスを利用して現実世界を模倣した
オブジェクト指向のプログラムを自由に開発できます。
しかし、間違えて属性を書き換えてしまったり、
誤った操作を呼び出してしまうなどのヒューマンエラーを
完全になくすことはできません。
そのため、Java にはミスを未然に防ぐ「カプセル化」のしくみがあります。
この章では、その便利なしくみについて学んでいきましょう。

CONTENTS

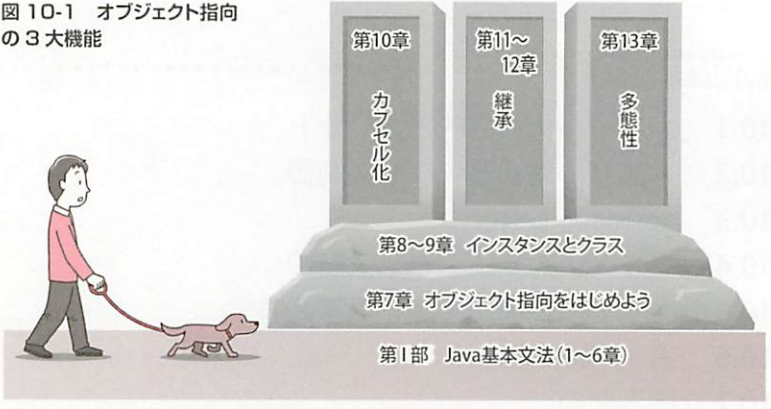
10.1 カプセル化の目的とメリット

10.1.1 オブジェクト指向の３大機能

第８章と第９章で学んだ「クラスの宣言方法とインスタンスの利用法」をマスターしていれば、オブジェクト指向の考え方に沿ったプログラムを作ることができます。

ですが、もし「オブジェクト指向プログラミングを、さらにラクに、さらに楽しくするしくみ」があるとしたら、もっと魅力的だと思いませんか？ たとえば今までの半分の労力で、もっと楽しく、しかもエラーをできるだけ防げるプログラムが書けたとしたらどうでしょう。実は Java には、そのような嬉しいしくみが３つも備わっています（図 10-1）。

図 10-1　オブジェクト指向の３大機能



この図に示した「カプセル化」「継承」「多態性」は、オブジェクト指向の３大機能と呼ばれています。この章では、まず最も簡単な「カプセル化」から学びましょう。

10.1.2 カプセル化とは？

Java に備わっている「カプセル化」とは、フィールドへの読み書きやメソッド

の呼び出しを制限する機能です。たとえば、「このメソッドは、A クラスからは呼び出せるけど、B クラスからは呼び出せない」「このフィールドの内容は、誰でも読めるけど、書き換えは禁止」といったことを実現できます。

「制限して不便にしてしまう機能」なんて意味あるんですか？ 誰でも自由にメンバを利用できるほうが便利だと思うけど…。

いや、そうとも限らないんだよ。

「大切なモノに対するアクセスは不自由であるほうがよい」ことを、私たちは現実世界でもよく知っています。たとえば、あなたの大切な銀行口座に対して、誰もが出し入れ可能だとしたら、どうでしょうか。あなたが気づかないうちにあなたのお金を誰かが引き出してしまうかもしれませんね。

ほかにも「登録された人しか立ち入れないように、塀に囲まれ、門には守衛がいる軍事施設」など、私たちの周りでは「制限」が行われている例が多くあります（図 10-2）。確かに不便ではありますが、この制限があるからこそ、次のようなメリットを享受できるのです。

- 悪意のある人が軍事施設に入り、ミサイルを発射してしまうことを防げる
- 子どもがうっかり軍事施設に入り、ミサイルを発射してしまうことを防げる
- 万一、何者かがミサイルを発射した場合、登録された人に犯人を絞り込める

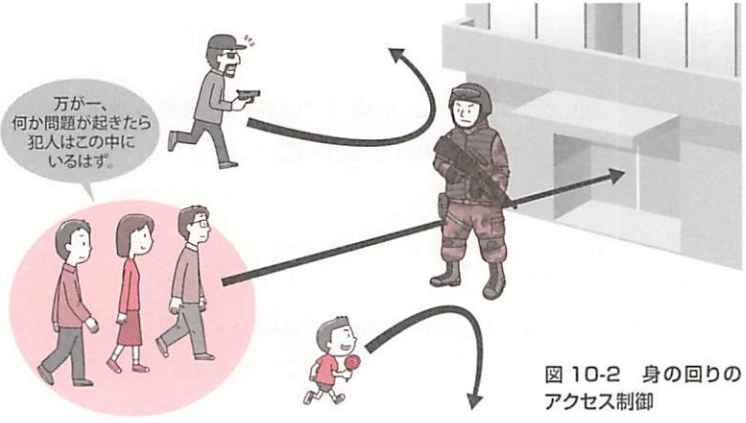


図 10-2　身の回りのアクセス制御

この例のように、情報へのアクセスや動作の実施について、「誰に何を許すか」を定めて制限することを、アクセス制御（access control）といいます。Java におけるカプセル化とは、大切な情報（フィールド）や操作（メソッド）についてアクセス制御をかけることにより、悪意や間違いによるメンバの利用を防止し、想定しない利用が発生したならば、その原因箇所を特定しやすくするためのしくみなのです。

10.1.3 アクセス制御されない怖さ

カプセル化によるアクセス制御の方法を学ぶ前に、「アクセス制御されないプログラムがいかに怖いか」を、RPG の開発プロジェクトを例に考えてみましょう。次のリスト 10-1 は、第 9 章までに作成した Hero クラスを改良したものです。

リスト 10-1　アクセス制御されていないプログラムの例

Hero.java

```
1  public class Hero {
2    int hp;
3    String name;
4    Sword sword;
5    static int money;
6    void bye() {
7      System.out.println("勇者は別れを告げた");
8    }
9    void die() {
10     System.out.println(this.name + "は死んでしまった！");
11     System.out.println("GAME OVERです。");
12   }
13   void sleep() {
14     this.hp = 100;
15     System.out.println(this.name + "は眠って回復した！");
16   }
17   void attack(Matango m) {
```

```
18      System.out.println(this.name + "の攻撃！");
19      ⋮
20      System.out.println("お化けキノコ" + m.suffix
            + "から2ポイントの反撃を受けた");
21      this.hp -= 2;
22      if (this.hp <= 0 ) {
23        this.die();
24      }
25    }
26    ⋮
27  }
```

反撃を受けると HP が 2 減る

この勇者はインスタンス化されると HP が 100 に設定されます。そして敵との戦いで HP が減少し、0 以下になったら死亡しゲームオーバーとなります。しかし、あなたは Hero クラスを使ったゲームのテスト中に、「一度もモンスターと戦っていないのに勇者の HP がマイナス 100 になっている」ことに気づきます（図 10-3）。

図 10-3　戦っていないのに HP がマイナスの勇者（死んでいる！）

あなたは数万行もあるゲームプログラムのいったいどこに不具合の原因があるのかを夜中まで調査し、原因をつきとめました。それは新入社員の A さんが開発した次のリスト 10-2 の「宿屋クラス」でした。

リスト 10-2　「宿屋クラス」の不具合

Inn.java

```
1  public class Inn {
2    void checkIn(Hero h) {
3      h.hp = -100;
4    }
5  }
```

ここが不具合の原因！

タイプミスして、100 の前にマイナス記号を付けちゃったのね。

そっかぁ。コンパイルエラーにはならないし、代入もできちゃうから気づかないよなぁ。

その翌日、今度は「冒険中にお城で会話をすると、なぜか勇者が理由もなく急死してゲームオーバーになる」という問題が見つかります。原因を調査したところ、またもや A さんが作ったリスト 10-3 の「王様クラス」に問題がありました。

リスト 10-3　「王様クラス」の問題点

King.java

```
1  public class King {
2    void talk(Hero h) {
3      System.out.println
           ("王様：ようこそ我が国へ、勇者" + h.name + "よ。");
4      System.out.println("王様：長旅疲れたであろう。");
5      System.out.println
           ("王様：まずは城下町を見てくるとよい。ではまた会おう。");
6      h.die();
7      ：
8    }
9  }
```

ここが不具合の原因。勇者が死ぬ！

先輩から「bye() を呼べ」と指示されたのを「die()」に聞き間違えたのかな？

うーん、英単語の意味を考えればわかりそうなものだけど…でも人間だから、間違えることもあるよね…。

今回の一連の不具合は A さんの不注意やスキル不足がきっかけで起こりました。しかし、見方を変えれば、簡単に「HP をマイナス 100 に設定できてしまうこと」や「王様が会話中に勇者を殺せてしまうこと」にも問題があります。

プログラムに以下のようなアクセス制御が盛り込まれていれば、このようなバグは事前に見つかったはずです。

Hero クラス以外からは hp フィールドに値を設定できない
die() メソッドを呼べるのは Hero クラスだけ

カプセル化はこのようなアクセス制御を実現し、想定外に発生する不具合を未然に防ぐためのしくみです。ぜひ次節以降のカプセル化の文法をしっかりマスターして、不具合が発生しにくいクラスを開発できるようになりましょう。

人間は必ずミスをする。だから不具合の原因を決して「人」に求めてはならない。原因は「ミスを未然に防ぐしくみがないこと」に求めるべきなんだ。

10.2 メンバに対するアクセス制御

10.2.1 4つのアクセス制御レベル

Javaでは、それぞれのメンバ(フィールドおよびメソッド)に対してアクセス制御の設定を行うことができます。ですが、それぞれのメンバに「Aクラス、Dクラス、Rクラスからの利用は許す」「Bクラス、Zクラスからの利用は許す」のように細かく指定すると、とても手間がかかってしまいます。そこで、表10-1のようにザックリと、4段階からアクセス制御の方法を選ぶようなしくみになっています。

表10-1　Javaにおけるアクセス制御の範囲と指定方法（メンバ編）

制限の範囲	名前	プログラム中の指定方法	アクセスを許可する範囲
制限が厳しい	private	private	自分自身のクラスのみ
↕	package private	(何も書かない)	自分と同じパッケージに属するクラス
↕	protected	protected	自分と同じパッケージに属するか、自分を継承した子クラス
制限が緩い	public	public	すべてのクラス

privateやpublicなどはアクセス修飾子(access modifier)と呼ばれ、フィールドやメソッドを宣言する際、先頭に記述することでアクセス制御が可能になります。

フィールドのアクセス制御

```
アクセス修飾子 フィールド宣言;
```

メソッドのアクセス制御

```
アクセス修飾子 メソッド宣言{…}
```

現時点で表 10-1 の 4 つすべてを覚える必要はないよ。まずは public と private だけ覚えておけば十分だ。特に protected に関しては後の「継承」に関する章で学ぶから、それまで完全に忘れていても構わないよ。

10.2.2 private を利用する

それでは private によるアクセス制御を体験してみましょう。10.1.3 項の「アクセス制御されない怖さ」では、Hero クラスの hp フィールドにマイナス 100 が設定されてしまいました。本来、HP フィールドは attack() したときに 2 ずつ減り、sleep() したときに回復すればよいので、他のクラスから変更できる必要はありません。よって HP は private にしておきましょう（リスト 10-4）。

リスト 10-4　HP を private にしたサンプルコード

Hero.java

```
1  public class Hero {
2    private int hp;
3    String name;
4    Sword sword;
5    static int money;
6    ⋮
7    void sleep() {
8      this.hp = 100;
9      System.out.println(this.name + "は、眠って回復した！");
10   }
11   ⋮
12 }
```

hp フィールドに private を指定したため、宿屋クラスの checkIn() メソッドでは「hp フィールドへはアクセスできない」というコンパイルエラーが発生するようになります。

しかし、勇者のhpフィールドがいっさい変更できなくなるわけではない点に注意してください。privateなフィールドであっても、同じクラスのメソッドからであれば、リスト10-4のsleep()メソッドのように「this」を用いて読み書きすることができます。宿屋クラスのcheckIn()メソッドの中では、勇者のHPフィールドに直接100を代入できない代わりに、sleep()メソッドを呼び出すように修正すればよいのです。

privateアクセス修飾

privateであっても、自分のクラスからthis.~での読み書きは可能。

また、die()メソッドについても、王様など、ほかのクラスからみだりに呼び出されることがないようにprivateにします（リスト10-5）。

リスト10-5　die()メソッドをprivateとして指定する

Hero.java

```
1  public class Hero {
2    ⋮
3    private void die() {
4      System.out.println(this.name + "は死んでしまった！");
5      System.out.println("GAME OVERです。");
6    }
7   ⋮
8  }
```

これでdie()メソッドは外部のクラスからは呼び出せなくなりますが、同じクラス内にあるattack()メソッドからの呼び出し（リスト10-1の23行目）は問題ありません。

10.2.3 public や package private を利用する

勇者は戦うのが仕事ですから、いろいろなクラスから attack() メソッドが呼び出される可能性があります。よって attack() は、どのようなクラスからでも呼び出せるように public 指定を付けておきましょう（リスト 10-6）。

リスト 10-6　attack() メソッドは public として指定する

Hero.java

```
1  public class Hero {
2    ⋮
3    void sleep() {
4    ⋮
5    }
6    public void attack(Matango m) {
7    ⋮
8    }
9    ⋮
10 }
```

ちなみに、sleep() メソッドには public を付けないままにしています。この場合、package private を指定したと見なされ、同じパッケージに属するクラスからの呼び出しのみ可能になります。仮に図 10-4 のように Hero クラスが rpg.characters パッケージに属しているとすれば、他のパッケージに属する Slime クラスなどからは利用できなくなります。

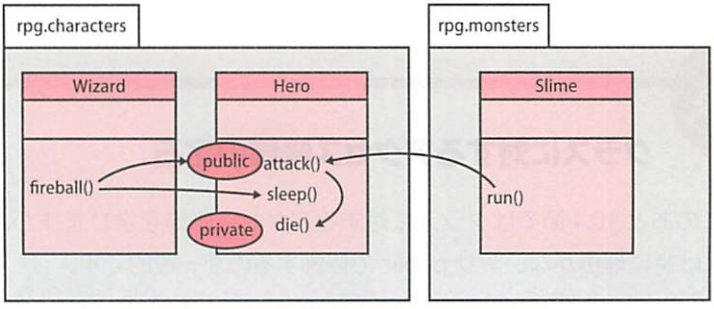


図 10-4　public、package private、private のアクセス制御

10.2.4 アクセス修飾の定石

private、public、package private についてはよくわかりました。
でも、どれを選べばよいのか迷っちゃいます…。

大丈夫、お決まりのパターンがあるんだよ。

どのメンバに、どのアクセス修飾子を指定すべきか、Java の文法では定められていません。アクセス修飾子は自由に指定できるので、「メンバの使われ方をプログラマがよく考慮した上で、最適なものを選ぶべき」というのが教科書的な答えです。しかし、「ほとんどのケースでは、このように指定しておけば大丈夫」あるいは「このような指定の仕方が基本」というパターン（定石）があります。

メンバに関するアクセス修飾の定石

- フィールドはすべて private
- メソッドはすべて public

とりあえずは、このパターンに沿ってアクセス修飾を行い、その後、Hero クラスの die() メソッドのように、クラス内部だけで利用するメソッドのみを private に指定し直すような「微調整」をすればよいのです。

クラスに対するアクセス修飾の定石

このあと 10.4 節ではクラスに対するアクセス修飾を学びますが、クラスは特に理由がない限り public で修飾するのが一般的です。

10.3 getter と setter

10.3.1 メソッドを経由したフィールド操作

前節の「定石」を見ていて思ったんですが、フィールドをすべて private にすると、外部からいっさい読み書きできなくなっちゃいませんか？

いや、そんなことはないよ。メソッドを経由すればフィールドにアクセスできるんだ。

ここでもう一度、リスト 10-4 をよく見てください。hp フィールドは private 指定され、ほかのクラスからはアクセスできなくなっています。しかし、外部のクラスから hp フィールドの値を変更できないかというと、そんなことはありません。

外部のクラスからであっても、attack() メソッドを呼べば HP を 2 減らすことができ、sleep() メソッドを呼べば HP を回復できます（図 10-5）。

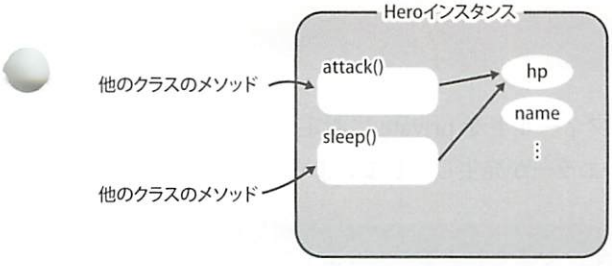


図 10-5　メソッドを経由しなければフィールドにはアクセスできない

hp フィールドの値を変化させるには、必ず attack() か sleep() を経由しなければならない点に注目してください。勇者の HP を増減するためには、このどちらかのメソッドを経由するほかありません。

つまり、他のクラス（宿屋クラスや王様クラス）の開発者がバグを含んだコード

を書いたとしても、勇者のHPをマイナス100に設定することは不可能です。

もし万が一、HPに異常な値が設定される不具合に直面しても、そのときはattack()かsleep()のどちらかのバグだということが簡単に予想できますから、不具合の修正もスムーズにできるでしょう。

基本的にフィールドはメソッド経由でアクセスするものなんですね。

10.3.2 単純にフィールド値を取り出すだけのメソッド

Heroクラスには名前を格納したnameというフィールドがあります。勇者の名前は、さまざまな場面で多くのクラスから利用されます。たとえば次のリスト10-7のように王様クラスの中でも利用されています。

リスト10-7　王様クラスで利用されるnameフィールド

King.java

```
1  public class King {
2    void talk(Hero h) {
3      System.out.println
           ("ようこそ我が国へ、勇者" + h.name + "よ。");
4      ⋮
5    }
6  }
```

しかし、Heroクラスの全フィールドをprivateに設定すると、このKingクラスでは次のようなコンパイルエラーが発生してしまいます。

```
name は Hero で private アクセスされます。
```

Heroクラスのnameフィールドはprivateであるため、Kingクラスからはその存在が「見えない」のです。このままでは王様が勇者の名前を得ることができず、名前を呼ぶことができません。

そこで、Hero クラスにリスト 10-8 のような getName メソッドを追加して、王様が勇者の名前を知ることができるようにしましょう。

リスト 10-8　Hero クラスに getName メソッドを追加

Hero.java

```
1  public class Hero {
2    private String name;
3    :
4    public String getName() {
5      return this.name;
6    }
7  }
```

そして King クラスでは、name フィールドにアクセスしている部分を、getName() を呼び出すように修正すれば完成です（リスト 10-9）。

リスト 10-9　King クラスの talk() メソッド内を以下のように修正

King.java

```
1  public class King {
2    void talk(Hero h) {
3      System.out.println
           ("王様：ようこそ我が国へ、勇者" + h.getName() + "よ。");
4      :
5    }
6  }
```

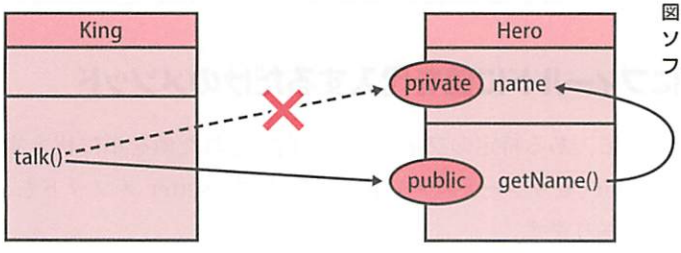


図 10-6　getName() メソッドを経由して name フィールドにアクセスする

このgetName()メソッドは、attack()メソッドなどとは異なり、単にnameフィールドの中身を呼び出し元に返すだけの単純なメソッドです。このようなメソッドを総称してgetter(ゲッター)メソッドといいます。

10.3.3 getterメソッドの書き方

ある特定のフィールド値を単に取り出すだけのメソッドは、すべてgetterメソッドと言えます。このgetterメソッドの書き方にも「定石」があります。

getterメソッドの定石

```
public　値を取り出すフィールドの型　get フィールド名 ()　{
    return this. フィールド名 ;
}
```

メソッド名の最初の3文字を「get」にし、それに続けて「フィールド名の先頭を大文字にしたもの」にします。 たとえば、フィールド名がnameならgetName()となります(例外として戻り値がboolean型の場合のみisXxxx()というメソッド名にすることがあります)。このメソッド名の付け方はJava開発者の間で常識になっている風習みたいなものだと思ってください。

たとえば、開発現場で「nameのgetter」などの言葉が飛び交うことがある。これはnameフィールドに対するgetterメソッド、つまりgetName()メソッドのことを指しているんだ。

10.3.4 単純にフィールドに値を代入するだけのメソッド

getterメソッドとは逆に、ある特定のフィールドに指定された値を単に代入するだけのメソッドをsetter(セッター)メソッドといいます。setterメソッドも、その記述方法には定石があります。

setter メソッドの定石

```
public　void　set フィールド名 ( フィールドの型　任意の変数名 ) {
  this. フィールド = 任意の変数名 ;
}
```

たとえば、Hero クラスについて、name フィールドに対応する setter メソッドを追加するとリスト 10-10 のようになります。

リスト 10-10　setter メソッドの例

Hero.java

```
1  public class Hero {
2    ⋮
3    public void setName(String name) {
4      this.name = name;
5    }
6  }
```

カプセル化とは関係ないことだが、このコードの代入式では左辺に this. を忘れると大事故につながることを再認識しておこう。なぜかわからない人は 8.2.5 項を読み返してほしい。

10.3.5 getter／setter の存在価値

ちょっと待ってください！ せっかく name フィールドを private にして外部のアクセスから守ったのに、また getter／setter を用意してアクセスを外部に開放したら private の意味がないんじゃありませんか？

いや、そんなことはないよ。getter／setter には重要な存在価値があるんだ。

Hero クラスの name フィールドに関係する部分だけを取り出して、カプセル化の前（リスト 10-11）と、カプセル化の後（リスト 10-12）のコードを見比べてみましょう。

リスト 10-11　カプセル化を行う前

Hero.java

```
1  public class Hero {
2    ⋮
3    String name;
4    ⋮
5  }
```

リスト 10-12　カプセル化を行った後

Hero.java

```
1  public class Hero {
2    private String name;
3    public String getName() {
4      return this.name;
5    }
6    public void setName(String name) {
7      this.name = name;
8    }
9  }
```

どちらも name フィールドの読み書きができることに違いはありません。むしろコードの行数が増えるため、getter や setter を利用することに意味が感じられないかもしれません。しかし、getter と setter には次のようなメリットがあります。

メリット 1：Read Only、Write Only のフィールドを実現できる

リスト 10-12 の setName() メソッドを削除すれば、name フィールドを「外部から読めるが書き換えられない（Read Only）」フィールドにできます。実際のプログラミングでも「外部から自由に読めるようにしたいが、変更されては困る」というフィールドが必要になることがありますが、その際に多用されるテクニックです。

また、あまり使われませんが、setter メソッドだけを準備して「外部から自由に書き換えできるが、読めない（Write Only）」フィールドも作成可能です。

メリット 2：フィールドの名前など、クラスの内部設計を自由に変更できる

たとえば将来、何らかの理由で name というフィールド名を newName に変更したくなったとしましょう。もし getter／setter を準備せず、他のクラスから直接、name フィールドを読み書きしていた場合、他のクラスのすべての開発者に「アクセスするフィールド名を変更してもらうお願い」をして回らなければなりませ

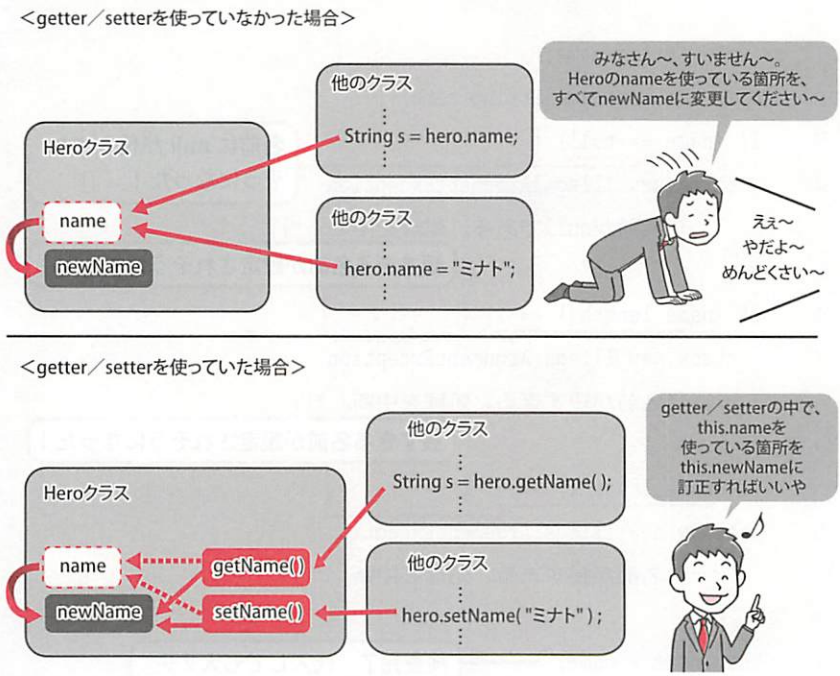


図 10-7　外部から隠せば、内部仕様の変更が柔軟に行える

ん（図 10-7 の上側）。

一方、name フィールドを隠し、外部からは getter／setter 経由で読み書きさせるのならフィールド名の変更は自由です。なぜなら getter や setter の内部でフィールドを使っている箇所だけを修正すればよく、getName() や setName() を呼び出している他の開発者には影響がないからです（図 10-7 の下側）。

メリット 3：フィールドへのアクセスを検査できる

クラス外部から name フィールドの値を書き換えたい場合、setter を使う必要が生じます。つまり、「setter を実行せずに、name フィールドを書き換えることは不可能」です。

このことを利用して、setter で「設定されようとしている値が妥当かを検査する」ことも Java プログラミングの定石です。たとえばリスト 10-13 のように setName() を改良してみましょう。

リスト 10-13　setter メソッドの中で値の妥当性をチェックする

```
1  private String name;
2  public void setName(String name) {
3    if (name == null) {
4      throw new IllegalArgumentException
           ("名前がnullである。処理を中断。");
5    }
6    if (name.length() <= 1) {
7      throw new IllegalArgumentException
           ("名前が短すぎる。処理を中断。");
8    }
9    if (name.length() >= 8) {
10     throw new IllegalArgumentException
           ("名前が長すぎる。処理を中断。");
11   }
12   this.name = name;
13 }
```

名前に null が代入されそうになった！

短すぎる名前が設定されそうになった！

長すぎる名前が設定されそうになった！

検査完了。代入しても大丈夫。

「throw new IllegalArgumentException」は、今の段階では「エラーを出してプログラムが強制停止する命令」と理解しておいてほしい。

このsetName()メソッドは、nameフィールドの値を変更しようとするたびに検査を行います。もし、リスト10-14のような問題のあるプログラムを実行すると、プログラムはきちんと停止するため、開発者はバグに気づくことができます。

リスト10-14　setNameが正しく機能するかを確認する

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h = new Hero();
4      h.setName("");
5    }
6  }
```

長さ0文字の名前をセットしようとする

実行結果

```
Exception in thread "main" java.lang.IllegalArgumentException:
名前が短すぎる。処理を中断。
  at Hero.setName(Hero.java:7)
  at Main.main(Main.java:4)
```

ぜひ、「どのように誤っても、どのような悪意を持ったアクセスでも、外部から絶対に不正な値を設定できない」安全・安心なクラスのプログラミングを目指して、検査を徹底させた強固なsetterを書くように心がけてください。

10.4 クラスに対するアクセス制御

10.4.1 2つのアクセス制御レベル

メンバへのアクセス制御と同じく、あるクラス全体に対してアクセス制御を設定することができます。クラスのアクセス制御レベルは、表 10-2 のように 2 種類しかありません。

表 10-2　クラスへのアクセス制御の指定方法と範囲

名前	Java での記述	許可する範囲	制限
package private	（何も書かない）	自分と同じパッケージに属するクラス	厳しい
public	public	すべてのクラス	緩い

これまで、クラス宣言の前には public を付けると丸暗記していましたが、実はクラス宣言の先頭に public という記述がない場合、そのクラスは同一パッケージに属するクラスからのアクセスのみ許可されます。

他のパッケージに属するクラスからのアクセスが禁止されるわけですが、イメージとしては「他のパッケージに属するクラスから、そのクラスの存在自体が見えなくなる」と捉えたほうがよいでしょう。

そのため、たとえ public 指定されたメソッドであっても、属するクラスが

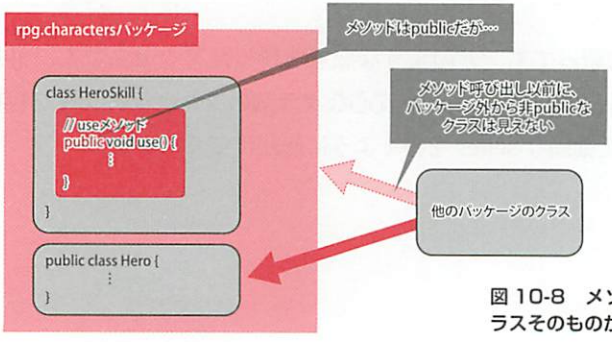


図 10-8　メソッドを呼び出す以前にクラスそのものが見えない

package private ならば、別パッケージのクラスからはそのメソッドを呼び出せなくなります（図 10-8）。

10.4.2 非 public クラスとソースファイル

別パッケージのクラスから見えなくなってしまう package private クラスですが、その代わりに public クラスでは許可されない次の 2 つが許されています。

非 public クラスの特徴

①クラスの名前はソースファイル名と異なってもよい。
②1 つのソースファイルに複数宣言してもよい。

これまでは「1 つのファイルに 1 つのクラス」「ファイル名＝クラス名」が原則だと紹介してきましたが、より正確には、「1 つのファイルに 1 つの public クラス」「ファイル名＝ public クラス名」というルールです（図 10-9）。

public が付いていないクラスは、どのソースファイルにいくつ宣言されても構いません。なお、ソースファイルに public クラスが 1 つも含まれない場合、ソースファイル名は自由に決めることができます。

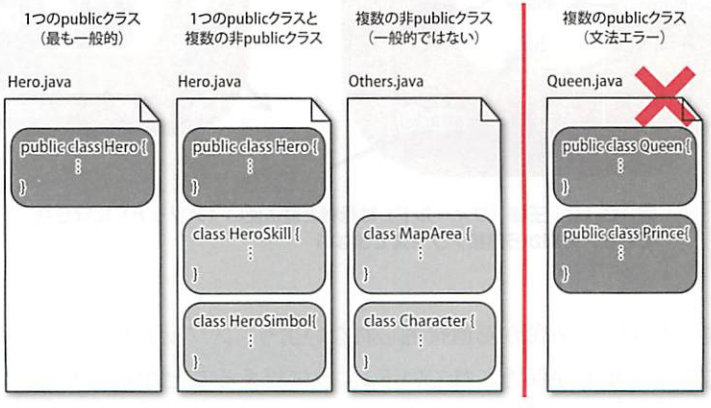


図 10-9　1 つのファイルに複数のクラスを宣言するバリエーション

10.5 カプセル化を支えている考え方

10.5.1 メソッドでフィールドを保護する

この章では public や private を用いて、クラスやメンバに対するアクセス制御の方法を学びました。

特に重要なメンバのアクセス制御では、特別な理由がない限り、「フィールドは private として外部から隠し、必ず setter／getter メソッド経由でアクセスする」という定石についても理解できたと思います。フィールドはメソッドによって守られており、外部から直接アクセスできないようにするのです（図 10-10）。

身分の高い王様は護衛に守られていて、直接は謁見できないのと似ていますね。

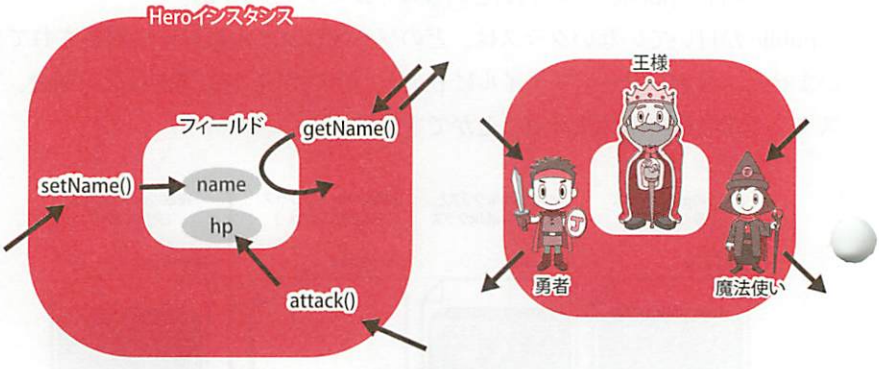


図 10-10　王様（フィールド）は勇者と魔法使い（メソッド）に守られており、外部から直接アクセスできない

この図のように、「外部から直接触られないよう、メソッドという殻（カプセル）によってフィールドが保護されている」ように見えることから、カプセル化という名前が付いています。

ところで、なぜカプセル化ではメソッドではなくフィールドを保護しようとするのでしょうか？ それは、メソッドよりフィールドのほうが異常な状態（不具合）になりやすいからです。

メソッドの処理内容は、プログラミング段階で決定し、一度コンパイルされればプログラム実行中に変化することはありません。一方、フィールドの値は、プログラムが動作する間に刻々と変化していきます。そのため、動作中に異常な値になる危険性もありえます。結果的に、不具合の多くは「フィールドに予期しない値が入る」という形で発現します。

したがって、「プログラムの不具合を減らすためには、メソッドよりもフィールドを保護することが重要」なのです。プログラムの不具合を防止するために、どんどんカプセル化を活用していきましょう。

適切にカプセル化されていれば、インスタンスは大切なフィールドを直接外部にさらすことなく、互いに公開したsetter／getterやその他のメソッドを呼び合うことで、安全に連携できるのです。

10.5.2 カプセル化の本質

よし、これでカプセル化もマスターだ！ でもやっぱり、「オブジェクト指向といえば継承！」なんですよね？

確かに次章で学ぶ継承のほうが有名だけど、カプセル化こそがオブジェクト指向の本質を支えているんだ。

第7章で学んだオブジェクト指向の本質を思い出してください。システムやプログラムというのは、突き詰めれば「現実世界における何かの相互作用」を自動化するためのものでした。そして「現実世界の登場人物たちの動きを、そっくり仮想世界に再現する」ことがオブジェクト指向の基本的な考え方です。

では、このオブジェクト指向の世界において、「バグ」「不具合」とはいったい何でしょうか？ それは、すなわち 次の状態にほかなりません。

不具合とは

「そもそもバグとは、現実世界と仮想世界が食い違ってしまうこと」

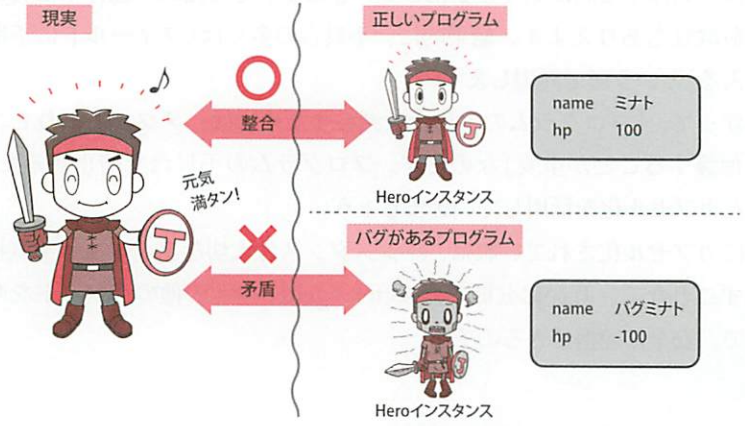


図 10-11 「不具合」とは、現実と矛盾した状態の仮想世界のこと

「実際の勇者は元気なのに、ゲームではなぜか HP が -100 になっている（図 10-11）」「実際の在庫数は 800 なのに、システム内ではなぜか 80 になってしまう」からバグなのです。

しかし、この章で学んだカプセル化を使えば、どのように利用されても、フィールドに不正な値が入ってしまうことのない、「現実の登場人物と矛盾しないクラス」を作ることができます。そして、その「現実の登場人物と矛盾しないクラス」を集めてプログラムを作れば、「現実世界と矛盾しないプログラム」になるという考えがカプセル化の本当の狙いなのです。

以降の章で学ぶ「継承」や「多態性」に比べれば、この章で学んだカプセル化は比較的簡単で、文法や効果にも華やかさはありません。しかし「現実世界を忠実にまねる」というオブジェクト指向の本質と直結している、最も重要な位置づけにある機能の 1 つだといえるでしょう。

10.6 第10章のまとめ

この章では、次のようなことを学びました。

カプセル化の概要

- カプセル化を用いるとメンバやクラスについてアクセス制御が可能になる。
- 特に、フィールドに「現実世界ではありえない値」が入らないように制御する。

メンバに対するアクセス修飾

- private 指定されたメンバは、同一クラス内からしかアクセスできない。
- package private 指定されたメンバは、同一パッケージ内のクラスからしかアクセスできない。なお、メンバ宣言に特定のアクセス修飾子を付けなければ package private になる。
- public 指定されたメンバは、すべてのクラスからアクセスできる。

クラスに対するアクセス修飾

- package private 指定（修飾子なし）で宣言されたクラスは、同一パッケージ内のクラスからしかアクセスできない。
- public 指定されたクラスは、すべてのクラスからアクセスできる。

カプセル化の定石

- クラスは public、メソッドは public、フィールドは private で修飾する。
- フィールドにアクセスするためのメソッドとして getter や setter を準備する。
- setter 内部では引数の妥当性検査を行う。

10.7 練習問題

練習 10-1

次の２つのクラス「Wizard（魔法使い）」「Wand（杖）」のすべてのフィールドとメソッドについて、カプセル化の定石に従ってアクセス修飾子を追加してください（Wizard クラスにコンパイルエラーが発生しますが、それは構いません）。

Wand.java

```
1  public class Wand {
2    String name;    // 杖の名前
3    double power;   // 杖の魔力
4  }
```

Wizard.java

```
1  public class Wizard {
2    int hp;
3    int mp;
4    String name;
5    Wand wand;
6    void heal(Hero h) {
7      int basePoint = 10;                  // 基本回復ポイント
8      int recovPoint = (int) (basePoint * this.wand.power);
9                                           // 杖による増幅
10     h.setHp(h.getHp() + recovPoint);     // 勇者のHPを回復させる
11     System.out.println
           (h.getName() + "のHPを" + recovPoint + "回復した!");
12   }
13 }
```

練習 10-2

問題 10-1 で作成した Wand クラスと Wizard クラスのすべてのフィールドについて、定石に従って getter メソッドと setter メソッドを作成してください。また、Wizard クラスの heal メソッドで発生しているコンパイルエラーを解決してください。ただし、setter メソッドに関しては引数の妥当性検証は不要です。

練習 10-3

問題 10-2 で作成した Wand クラスと Wizard クラスの各 setter メソッドについて、以下 4 種類のルールに従って引数の妥当性検証を追加してください。不正な値がセットされそうになった場合には、「throw new IllegalArgumentException(" エラーメッセージ ");」を記述してプログラムを中断させてください。

①魔法使いや杖の名前は null であってはならず、必ず 3 文字以上である。
②杖の魔力による増幅率は、0.5 以上 100.0 以下である。
③魔法使いの杖は null であってはならない。
④魔法使いの HP と MP は 0 以上である。ただし HP については負の値が設定されそうになると代わりに 0 が設定される。

10.8 練習問題の解答

問題10-1の解答

Wand.java

```
1  public class Wand {
2    private String name;      // 杖の名前
3    private double power;     // 杖の魔力
4  }
```

Wizard.java

```
1  public class Wizard {
2    private int hp;
3    private int mp;
4    private String name;
5    private Wand wand;
6    public void heal(Hero h) {
7      int basePoint = 10;                  // 基本回復ポイント
8      int recovPoint = (int) (basePoint * this.wand.power);
9                                           // 杖による増幅
10     h.setHp(h.getHp() + recovPoint);     // 勇者のHPを回復
11     System.out.println
           (h.getName() + "のHPを" + recovPoint + "回復した!");
12   }
13 }
```

問題10-2の解答

Wand.java

```
1  public class Wand {
2    private String name;      // 杖の名前
3    private double power;     // 杖の魔力
```

```
4    public String getName() { return this.name; }
5    public void setName(String name) { this.name = name; }
6    public double getPower() { return this.power; }
7    public void setPower(double power) { this.power = power; }
8  }
```

Wizard.java

```
1  public class Wizard {
2    private int hp;
3    private int mp;
4    private String name;
5    private Wand wand;
6    public void heal(Hero h) {
7      int basePoint = 10;                    // 基本回復ポイント
8      int recovPoint =                       // 杖による増幅
           (int) (basePoint * this.getWand().getPower());
9      h.setHp(h.getHp() + recovPoint);       // 勇者のHPを回復
10     System.out.println
           (h.getName() + "のHPを" + recovPoint + "回復した!");
11   }
12   public int getHp() { return this.hp; }
13   public void setHp(int hp) { this.hp = hp; }
14   public int getMp() { return this.mp; }
15   public void setMp(int mp) { this.mp = mp; }
16   public String getName() { return this.name; }
17   public void setName(String name) {this.name = name; }
18   public Wand getWand() { return this.wand; }
19   public void setWand(Wand wand) { this.wand = wand; }
20 }
```

問題10-3の解答

Wand.java

```
1  public class Wand {
2    private String name;      // 杖の名前
3    private double power;     // 杖の魔力
4    public String getName() { return this.name; }
5    public void setName(String name) {
6      if (name == null || name.length() < 3 ) {
7        throw new IllegalArgumentException
             ("杖に設定されようとしている名前が異常です");
8      }
9      this.name = name;
10   }
11   public double getPower() { return this.power; }
12   public void setPower(double power) {
13     if (power < 0.5 || power > 100.0) {
14       throw new IllegalArgumentException
             ("杖に設定されようとしている魔力が異常です");
15     }
16     this.power = power;
17   }
18 }
```

Wizard.java

```
1  public class Wizard {
2    private int hp;       private int mp;
3    private String name;  private Wand wand;
4    public void heal(Hero h) {
5      int basePoint = 10;                  // 基本回復ポイント
6      int recovPoint =                     // 杖による増幅
         (int) (basePoint * this.getWand().getPower());
7      h.setHp(h.getHp() + recovPoint);     // 勇者のHPを回復
8      System.out.println
```

```
            (h.getName() + "のHPを" + recovPoint + "回復した!");
 9    }
10    public int getHp() { return this.hp; }
11    public void setHp(int hp) {
12      if (hp < 0) { this.hp = 0; } else { this.hp = hp; }
13    }
14    public int getMp() { return this.mp; }
15    public void setMp(int mp) {
16      if (mp < 0) {
17        throw new IllegalArgumentException
              ("設定されようとしているMPが異常です");
18      }
19      this.mp = mp;
20    }
21    public String getName() { return this.name; }
22    public void setName(String name) {
23      if (name == null || name.length() < 3) {
24        throw new IllegalArgumentException
              ( "魔法使いに設定されようとしている名前が異常です" );
25      }
26      this.name = name;
27    }
28    public Wand getWand() { return this.wand; }
29    public void setWand(Wand wand) {
30      if (wand == null) {
31        throw new IllegalArgumentException
              ( "設定されようとしている杖がnullです" );
32      }
33      this.wand = wand;
34    }
35  }
```