

Accountなどのクラスを開発する私たち自身がhashCode()を正しくオーバーライドする必要があります。hashCodeの記述方法(ハッシュ値の計算アルゴリズム)にはさまざまなものが考えられますが、リスト4-5はその中の一例です。

リスト4-5 HeroクラスでhashCode()をオーバーライドする

```

1 class Hero {
2     String name;
3     int hp;
4     public int hashCode() {
5         int result = 37; ①適当な初期値を決める
6         result = result * 31 + name.hashCode(); ②各フィールドの影響を加える
7         result = result * 31 + hp; ③結果を返す
8         return result;
9     }
10 }

```

ここでもう一度、図4-4を見てください。HashSetは各要素のhashCode()を呼び出しハッシュ値の比較を行います。ハッシュ値の比較は単なる整数同士の比較なのでequals()よりはるかに高速に行うことができます。そしてハッシュ値が一致した場合に限ってequals()を用いて厳密に等価判定を行います。リスト4-4でうまく動かなかったのは、HeroクラスのhashCode()がオーバーライドされていなかったため、図4-4の⑧で正しくないハッシュコードが返され、equals()を呼ぶまでもなく削除候補から外されてしまったためなのです。

ハッシュ計算に登場する整数たち

リスト4-5のハッシュ計算アルゴリズムに「31」や「37」といった謎の整数が登場しました。ハッシュ値の初期値とした「37」については、0以外であれば何でも構いません。一方、各要素の影響を加えるために用いた乗数については、奇数かつ素数である31がよく用いられます。

4.5 インスタンスの順序づけ

4.5.1 インスタンスの並び替え

第3章のコラムで紹介したCollectionクラスのsort()メソッドはその名が示すとおり、呼び出すだけで中身の要素を順番に並び替えてくれる便利な静的メソッドです。たとえば、ArrayListに格納した複数の口座インスタンスを並び替える場合、リスト4-6のような使い方をします。

リスト4-6 口座インスタンスの並び替え

```

1 import java.util.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         List<Account> list = new ArrayList<Account>();
6         :
7         Collections.sort(list); ①これだけで要素が並び替えられる
8     }
9 }

```

しかしsort()には1つ重要な制約があり、それを意識せずに上記のようなコードを書くとうのように文法エラーが発生します。

制約の不一致: 型 Collections の総称メソッド sort(List<T>) は引数 (List<Account>) に適用できません。

推測される型 Account は、制約付きパラメーター <T extends Comparable<? super T>> の代替として有効ではありません



うわっ、なんだこの意味不明なエラーメッセージ…。なんだよ、単に並び替えるって指示しただけじゃないか。

「単に」って言うけど、JVMの立場になってごらん。



一口に並び替えるといっても、口座の並び替え順序としては「残高の多い順」「名義人の名前順」「口座番号順」などいろいろな方法が考えられます。単に並び替えると指示されてもJVMは困ってしまうのです。

ただし、もし「口座を並び替える」といったら、口座番号順に並べるのが普通でしよ」とあらかじめ宣言しておけば話は別です。あるクラスについて一般的に想定される並べ順のことを自然順序づけ (natural ordering) といいます。自然順序が定められているクラスであれば、単なる「list.sort();」という指示でもエラーは出ず、並び替えることができます。

4.5.2 Comparable インタフェースの実装

私たちがいるクラスを開発する際、そのクラスの自然順序を宣言するためには java.lang.Comparable インタフェースを実装します。このインタフェースを実装すると、次のリストのように compareTo() メソッドのオーバーライドが強制されることになります。このメソッドを使って私たちは自然順序づけの方法を宣言することができるのです。

リスト 4-7

```

1 public class Account implements Comparable<Account> {
2     int number; // 口座番号
3     :
4     public int compareTo(Account obj) {
5         if (this.number < obj.number) {
6             return -1;

```

Account.java

<~> で自身を指定

```

7     }
8     if (this.number > obj.number) {
9         return 1;
10    }
11    return 0;
12 }
13 }

```

compareTo() メソッドは、引数で渡されてきたインスタンス obj と自分自身とを比較し、その大小関係を判定するという責務を負っています。具体的には次のような戻り値を返すようにしなければなりません。

- ・自分自身のほうが obj よりも小さい場合 …… 負の数
- ・自分自身と obj とが等しい場合 …… 0
- ・自分自身のほうが obj よりも大きい場合 …… 正の数

Account クラスが「compareTo()」を持ち、自然順序づけが定義されているクラスならば、Collections クラスの sort() は、格納しているそれぞれのインスタンスの compareTo() を呼び出し大小関係を比較しながら並び替えを実行してくれます (図 4-6)。

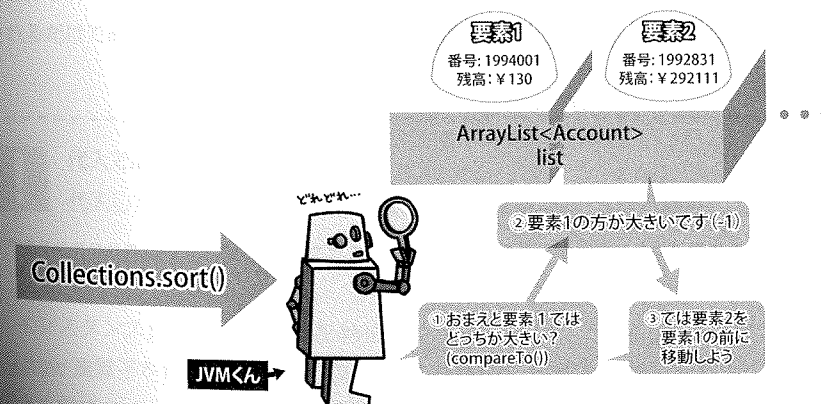


図 4-6 compareTo() を呼び出しながら並び替えを行う sort() メソッド

なお、第 3 章で少しだけ紹介した TreeSet は、内部で常に並び替えを行いながら要素を格納します。これは基本的に Comparable を実装したクラスを要素として格納することを前提としたコレクションクラスですので、Comparable が適切に実装されているか注意が必要です。

! Comparable を実装して便利に
開発するクラスに自然順序付けを考慮ができるならば、Comparable を実装しておくことで、並び替えなどが便利に行えるようになる。



今回は説明を割愛するが、もし機会があれば API リファレンスで Comparator クラスについて調べてみるといいだろう。

4.6 インスタンスの複製

4.6.1 コピーと参照




さて、では次に Java に備わっている「インスタンスをコピーする機能」について解説しよう。

え？ コピーなんて、代入すればできちゃうじゃないですか。



プログラムを開発していると、インスタンスをコピーする必要性が生じることもしばしばです。湊くんのように代入演算子を使う方法を思いつくかもしれません。

```
Hero h1 = new Hero("ミナト");
Hero h2 = h1;  コピー (したつもり)
```

しかし、この方法ではインスタンスのコピーは作られないという落とし穴があります。h1 の中にいた勇者インスタンスを指すアドレス情報(参照)がコピーされるだけであって、インスタンスの実体は 1 つのままです(次ページの図 4-7)。

! 代入ではコピーされない

代入しても参照がコピーされるだけで、実体はコピーされない。

インスタンス自体を複製するためには、

- ① new 演算子を用いて別のインスタンスを作成し変数 h2 に入れる。
 - ② h1 のすべてのフィールド内容を h2 にコピーする。
- という手順を踏まなければなりません。

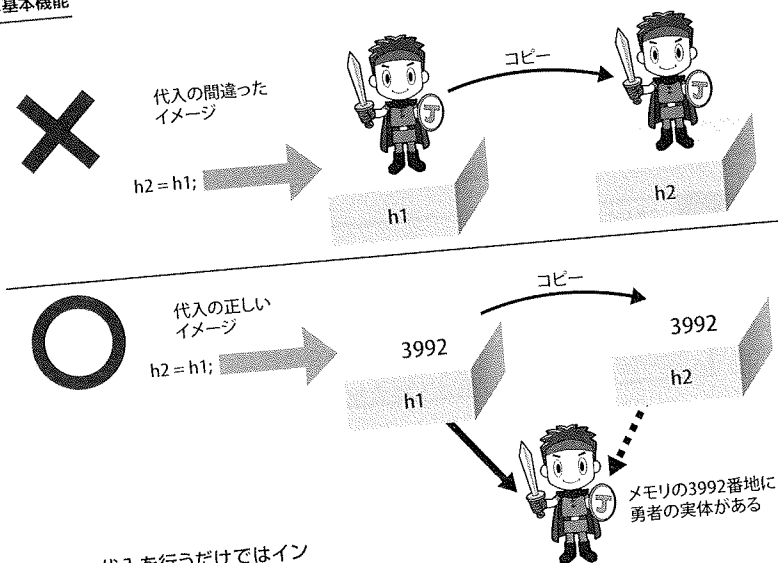


図 4-7 代入を行うだけではインスタンス自体はコピーされない

4.6.2 clone() メソッド

インスタンス自体を複製するために「① new と②全フィールドのコピー」をいちいち行わなければならないのは大変です。そこで、すべてのクラスは「自身の複製インスタンスを作って返す」という責務を持った clone() メソッドを Object クラスから継承しています。次のような簡単な記述で、「① new と②全フィールドコピー」済みの新たなインスタンスを簡単に得ることができます。

```
Hero h1 = new Hero("ミナト");
Hero h2 = h1.clone();
```

clone() を呼んで一発コピー



ははぁん。さては先輩、この機能も「実はオーバーライドしないと使えない物にならない」とか制約があるんですね？

2人が作るクラスで clone() による複製をサポートするためには、次ページのリスト 4-8 のように 2つの作業が必要なんだ。

リスト 4-8 clone() による複製をサポートした Hero クラス

```
1 public class Hero implements Cloneable {
2     String name; // 名前
3     int hp;      // HP
4     Sword sword; // 装備している剣
5     :
6     public Hero clone() {
7         Hero result = new Hero();
8         result.name = this.name;
9         result.hp = this.hp;
10        result.sword = this.sword;
11        return result;
12    }
13 }
```

Hero.java

① Cloneable インタフェースを実装する

まず、clone() による複製をサポートしていることを外部に対して表明するために java.lang.Cloneable インタフェースを実装する必要があります。

② clone メソッドを public でオーバーライドする

さらに clone() メソッドをオーバーライドしなければなりません。通常「新たなインスタンスを new で生成し、自身の全フィールドをコピーして return で返す」という処理内容になります（なお clone() をサポートした親クラスがある場合は、new ではなく親の super.clone() を呼んでインスタンスを生成します。また、Java5 以降では戻り値の型は自身のクラス名にします）。

このとき注意が必要なのが、アクセス修飾子の指定です。Object クラスで定義してある clone() メソッドは protected で宣言されているために外部から呼び出せません。オーバーライドの際には public でオーバーライドして外部から呼び出せるようにします。



API リファレンスで Cloneable を調べて驚きました！ このインタフェース、1 つもメソッドを持ってないんですね。

ほんとだ。てっきり clone() が定義されているのかと思いました。



compareTo() を定義している Comparable とは異なり、Cloneable は clone() を定義していません。Cloneable は「clone() を実装することによって複製に対応していることを表明するため」だけに存在しています。このような目的で利用する特殊なインタフェースを特にマーカーインタフェース (marker interface) とい、ほかには第 10 章で学ぶ java.io.Serializable が有名です。

オーバーライドによるアクセス修飾の拡大

protected を public で上書きする clone() のオーバーライドに限らず、Java ではオーバーライド時にアクセス修飾を変更することが許されています。ただし、以下のような制約があるため十分に注意してください。

子クラスにおけるオーバーライド時のアクセス修飾は、親クラスにおけるそれと同じか、より緩いアクセス修飾に限定される。

4.6.3

複製の失敗



よし！これで簡単に勇者を複製できるようになったぞ！

本当にそうかな？ 試しに、持ってる剣の名前を変えてごらん。



リスト 4-8 のように clone() による複製をサポートした Hero クラスですが、使い方によっては想定外の動きをすることがあります。次のリスト 4-9 を実行してみましょう。

リスト 4-9 複製した勇者の剣の名前を変更してみる

Main.java

```

1 public class Main {
2     public static void main(String[] args) {
3         Hero h1 = new Hero("ミナト");
4         Sword s = new Sword("はがねの剣");
5         h1.setSword(s);
6         System.out.println("装備:" + h1.getSword().getName());
7         System.out.println("clone() で複製します");
8         Hero h2 = h1.clone(); ここで複製
9         System.out.println("コピー元の勇者の剣の名前を変えます");
10        h1.getSword().setName("ひのきの棒");
11        System.out.println
12            ("コピー元とコピー先の勇者の装備を表示します");
13        System.out.print("コピー元:" + h1.getSword().getName() +
14            " / コピー先:" + h2.getSword().getName());
15    }
16 }
```

実行結果

装備: はがねの剣

clone() で複製します

コピー元の勇者の剣の名前を変えます

コピー元とコピー先の勇者の装備を表示します

コピー元: ひのきの棒 / コピー先: ひのきの棒 コピー先の装備まで変わってしまった

4.6.4

深いコピーと浅いコピー



ちゃんとコピーして h1 と h2 は別物になったはずなのに、どうして h1 の武器名変更が h2 にまで影響しちゃうんだろう。

一口に「同じ」といっても等値と等価があったように、「コピー」にも 2 種類のコピー方式があります。

リスト 4-8 の clone() メソッドの中身は浅いコピー (shallow copy) と呼ばれる方法で記述されています。浅いコピーでは各フィールドのコピーに代入演算子を使うため、たとえば 10 行目では sword の参照がコピーされるのみで剣のインスタンス自体は複製されません。これこそ「片方の勇者の剣の名前を変えると他方の勇者の剣の名前も変わる」という副作用の原因です (図 4-8)。

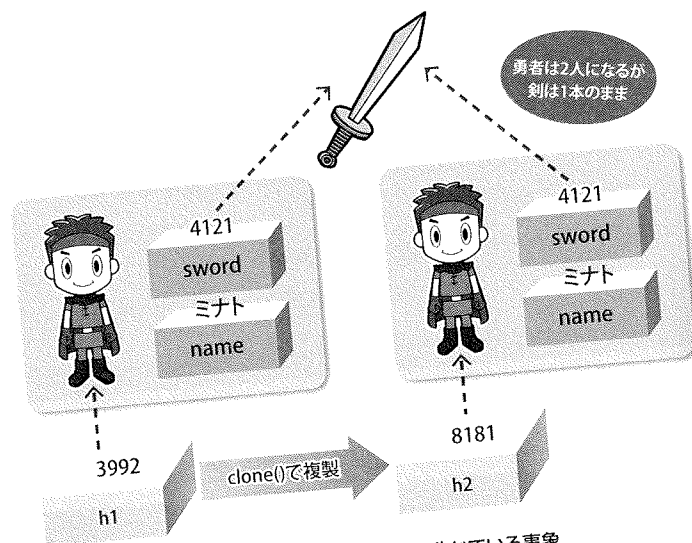


図 4-8 浅いコピーによって生じている事象

一方、各フィールドについても clone() などを使い別インスタンスとしてコピーする方法を深いコピー (deep copy) と呼びます。リスト 4-8 の 10 行目を「result.sword = this.sword.clone();」のように深いコピーにすれば、勇者と剣の両方が正しく複製されます。

4.7 この章のまとめ

Object の 3 大基本操作

- ・ toString() をオーバーライドして文字列表現を定義できる。
- ・ equals() をオーバーライドして意味的に正しく等価を判定できるようにする必要がある。
- ・ hashCode() を正しくオーバーライドしなければ、ハッシュ値を用いるクラス (HashMap など) で誤作動の原因となる。

順序付けと複製

- ・ Comparable を実装することで自然順序付けを定義でき、容易に並び替えできるようになる。
- ・ Cloneable を実装し clone() をオーバーライドすることにより、インスタンスの複製を簡単に作れるようになる。
- ・ インスタンスの複製には深いコピー (deep copy) と浅いコピー (shallow copy) がある。