第9章

さまざまなクラス機構

第8章ではオブジェクト指向に必要な知識として、
インスタンスとクラスの基本的な使い方について学びました。
この章ではインスタンスとクラスに関する理解をさらに深めた上で、
プログラミングをより便利にしてくれる
「コンストラクタ」と「静的メンバ」について学びましょう。

CONTENTS

9.1 クラス型と参照

9.1.1 仮想世界の真の姿

先輩、「仮想世界」とか「生み出される」とかイメージはわかりましたが、コンピュータの中に「本当に仮想世界の住人がいる」わけではないと思うんです。実際はどうなっているんですか？

もっともだ。クラスに関する数々の便利な機能を学ぶ前に、ここまで学んだ内容が実行されるとき、JVMの中で何が起こっているかタネあかしをしよう。今のうちにこれを理解しておくと、後々の章での学習がグッとラクになるんだ。

ここまでの解説では、インスタンスのことを「操作と属性を持ち、コンピュータ内の仮想世界に生み出される登場人物」という概念的な表現で紹介してきました。しかし、本当に「勇者」や「お化けキノコ」のような存在がコンピュータの中に住んでいて、あれこれ活躍するわけではありません。

本書がこれまで「Java仮想世界」と表現してきたものは、**実際には「コンピュータのメモリ領域」**です。この領域は、Javaのプログラムを実行する際に、JVMが大量にメモリ領域（通常は数百MB～数GB）を使って準備するもので、**ヒープ**（heap）といいます。

そして、私たちがnewを用いてインスタンスを生み出すたびにヒープの一部の領域（通常は数十～数百バイト）が確保され、インスタンスの情報を格納するために利用されます。そのため、多くの属性を持った大きなクラスをインスタンス化すると、消費されるヒープ領域は必要とする容量に従って大きくなります。つまり次の図9-1に示すように、**インスタンスとは「ヒープの中に確保されたメモリ領域」**にすぎないのです。

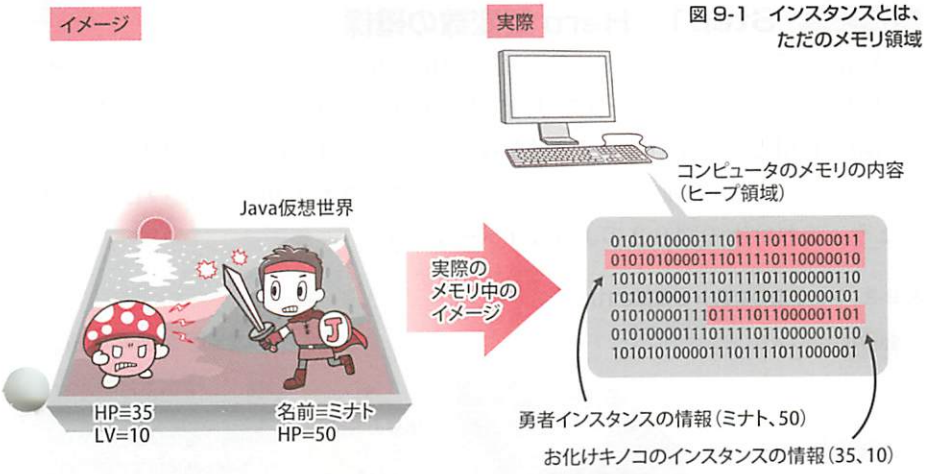
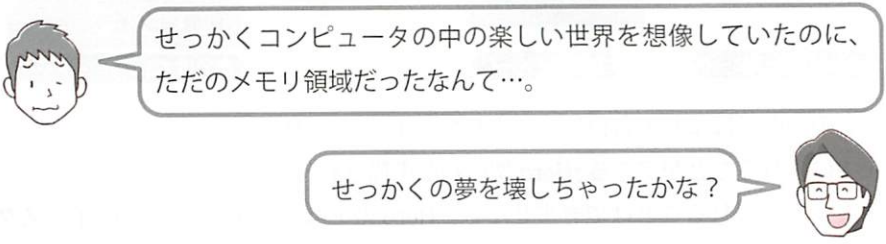


図 9-1 インスタンスとは、ただのメモリ領域



9章

9.1.2 クラス型変数とその内容

インスタンスの正体は「ヒープの一部に確保された単なるメモリ領域」ということがわかりました。ではそのインスタンスが生まれる際に、コンピュータの中で何が起こっているのか、次のコードを例に探っていきましょう。

リスト 9-1 Hero クラスをインスタンス化し利用するコード

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Hero h;
4     h = new Hero();
5     h.hp = 100;
6   }
7 }
```

9.1.3 Step1：Hero 型変数の確保

最初に動くのは、リスト 9-1 の 3 行目「Hero h;」です。この行を実行すると、JVM は「Hero 型の変数 h」をメモリ内に準備します。JVM は広いヒープ領域の中から現在利用していないメモリ領域を探し出して、自動的にそこを確保してくれます。仮に、1928 番地が空いていたので、ここが変数 h 用に確保されたとしましょう。これは次の図 9-2 のようなイメージです。

図 9-2　Hero 型の変数 h が確保された

この段階ではまだ勇者自体は生まれていません。「Hero 型のインスタンスだけを中に入れることができる」Hero 型の箱が準備されるだけです。この箱には数字や文字列を入れることはできませんし、Hero 型でない「お化けキノコ」インスタンスを入れることもできません。

菅原さん、ボクどうしても「Hero 型の箱」っていうのがしっくりこないんです。int 型や String 型だったら、数字や文字列が入るってわかりますが…。何というか、箱に「インスタンス」みたいな複雑なものが入るっていうのが考えにくいんです。

まぁそのうち慣れるよ。それに、この後の Step2 と Step3 の流れを知れば、その悩みもスッキリするはずだよ。

9.1.4 Step2：Hero インスタンスの生成

リスト 9-1 の 4 行目「h = new Hero();」は代入文です。**代入の場合は左辺より先に右辺が評価される**のでしたね。よって、Step2 では、まず「new Hero()」の部分だけを考えます。「new Hero()」が実行されると、JVM は次の図 9-3 のように

ヒープ領域から必要な量のメモリを確保します。今回は仮に「3922 番地から 24 バイト分（3922 ～ 3945 番地）」が確保できたとします。なお、この 3922 番地は Step3 の図 9-4 で出てきますので覚えておいてください。

図 9-3　JVM がヒープ領域からメモリを確保し、Hero インスタンスが生成された

これで仮想世界に勇者という存在が生まれました。しかし、生まれたての Hero インスタンスは属性が設定されていないため、まだ「名前」は空っぽ、「HP」は 0 です。

9.1.5 Step3：参照の代入

Step2 では、リスト 9-1 における 4 行目の「h = new Hero();」の右辺について考えました。次は右辺が実行された後のことについて考えましょう。

右辺の実行が終了した後、4 行目は「h = **右辺の実行結果**」という状態になります。すなわち、右辺の実行結果を変数 h に代入するということが行われます。

このとき、変数 h に代入される右辺の実行結果とは何でしょうか。ここまでの解説では、「Hero 型の箱に、勇者インスタンスが入る」という説明をしてきました。つまり、右辺の実行結果とは勇者インスタンスなのでしょうか。

いいえ、実はそうではありません。右辺の実行結果とは、**new を実行することによって生成されたインスタンスのために確保されたメモリの先頭番地**です。今回の場合、「new Hero()」により、勇者インスタンスが 3922 ～ 3945 番地に生成されているので、**変数 h には 3922 という数字が代入**されます（次ページ図 9-4）。

図 9-4　Hero インスタンスの変数代入

変数 h に入っている 3922 は、ただの数字にすぎません。勇者インスタンスに関する HP や名前などのさまざまな情報は変数 h の中ではなく、別のところにあります。見方を変えれば、「この変数 h には勇者インスタンスの情報の全部は入りきらないから、詳しくは 3922 番地を参照してね」とも捉えることができます。このようなことから、変数 h に入っているアドレス情報を**参照**といいます。

これって…どこかで似たようなものを学習したような気がしますけど？

そうだね、第 4 章の「配列型」と同じしくみだよ。

int[] 型や String[] 型といった「配列型」も、変数に入っているのは「実際の配列内の各データが保存されているメモリ領域の先頭番地」でしたね。Hero 型のような「クラス型」も原理は同じです。このことからクラス型と配列型は総称して「参照型」と呼ばれます。

9.1.6 Step4：フィールドへの値の代入

5 行目の「h.hp = 100;」では、変数 h に格納されている勇者の HP を 100 に設定します。この行を JVM は以下のように解釈して実行します。

①変数 h の内容を調べると、「3922 番地を参照せよ」と書かれている
②メモリ内の 3922 番地にあるインスタンスのメモリ領域にアクセスし、その中の hp フィールド部分を 100 に書き換える

図 9-5　Hero インスタンス hp フィールドへの代入

このように、①まず変数から番地情報を取り出し、②次に実際にその番地にアクセスする、というJVMの動作を参照の解決やアドレス解決といいます(図 9-5)。

インスタンスを生み出したり、フィールドにアクセスしたりするために、JVMはとても複雑なことをしているんですね。

9.1.7 同一インスタンスを指す変数

もし仮想世界に勇者が2人生成され、それぞれh1、h2という変数に格納されていたとしましょう。当然、勇者h1のhpフィールドを10減らしても、勇者h2のhpフィールドの値は減りません。同じクラスから生まれても、異なるインスタンスであれば互いに影響を受けないことをインスタンスの独立性といいます。

さて、以上を踏まえた上で、次のプログラムの実行結果を想像してください。

リスト 9-2　2つの Hero 型変数

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h1;
4      h1 = new Hero();
5      h1.hp = 100;
6      Hero h2;
7      h2 = h1;
8      h2.hp = 200;
9      System.out.println(h1.hp);
```

```
10      }
11  }
```

えーっと、h1 と h2 の 2 つがあって…「インスタンスの独立性」があるから、h1 の hp フィールドには 100、h2 の hp フィールドには 200 が入って…。画面に表示されるのは「100」かな？

いいや違うよ。図に書いて、よく考えてごらん。

このプログラムを正しく理解するためのポイントは 7 行目の「h2 = h1;」です。これは変数 h1 の内容を h2 にコピーする式ですが、ここで 9.1.5 節で説明した**「勇者インスタンス h のために確保してあるメモリの先頭番地」**を思い出してください。ここでコピーされているのは「勇者インスタンスそのもの」ではありません。**「3922」などの番地情報**です（図 9-6）。

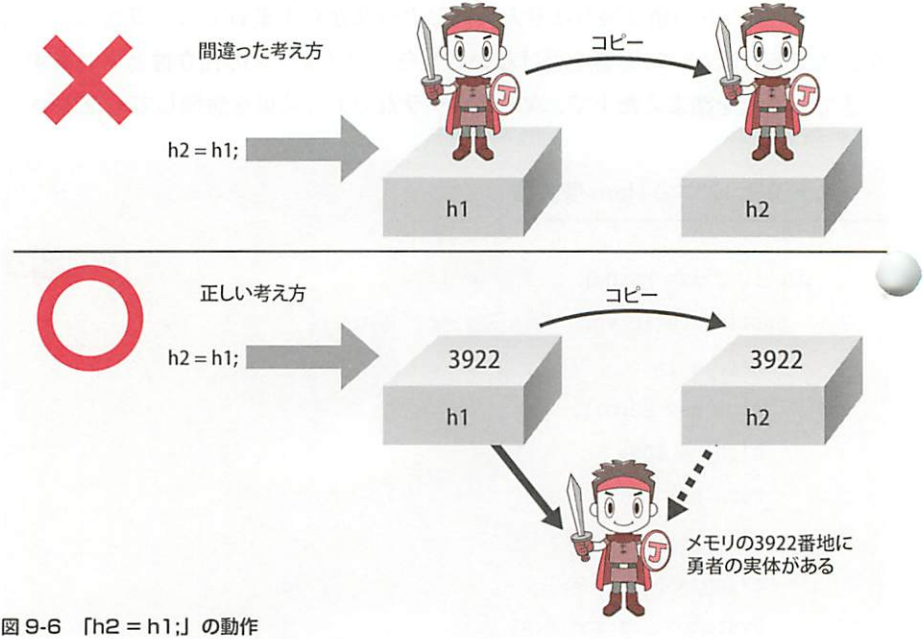


図 9-6　「h2 = h1;」の動作

代入の結果、h1 と h2 の両方に番地情報の 3922 が入ります。ということは、**h1 と h2 はどちらも「まったく同じ 1 人の勇者インスタンス」を指している**のです。そのため h1 の hp フィールドへ代入しても、h2 の hp フィールドへ代入しても、**結局は同じ勇者インスタンスの HP に代入することになる**のです。

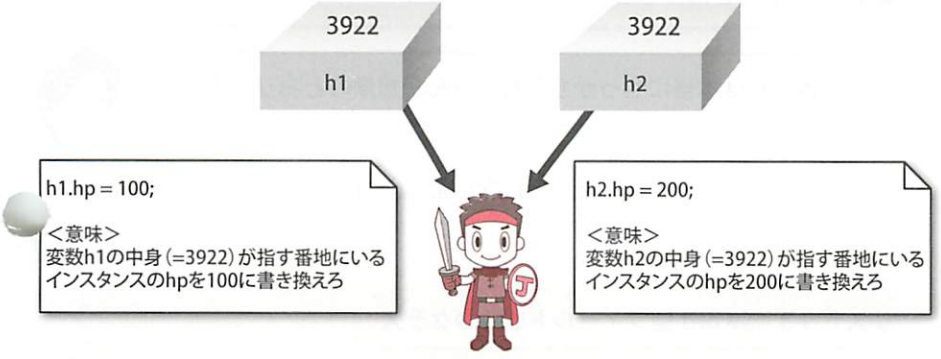


図 9-7　h1.hp と h2.hp は同じもの

リスト 9-2 に示したプログラムの場合、勇者インスタンスの hp フィールドへは h1 経由で 100 が代入されます（5 行目「h1.hp = 100;」）が、その後に同じ hp フィールドには h2 経由で 200 が上書き（8 行目「h2.hp = 200;」）されます。よって 9 行目で表示されるのは「200」なのです（図 9-7）。

そうか、「インスタンスの独立性」というのは勇者インスタンスが 2 人いた場合の話であって、今回のように「勇者が 1 人しかいない」場合には関係ないですね。h1 と h2 があったから、つい「勇者が 2 人いる」と勘違いしました。

わかった！ そもそも、リスト 9-2 の中には、**new が 1 つしかない**のよ。ということは仮に変数がいくつあっても、どんな複雑なプログラムであっても、**「仮想世界には 1 人の勇者しか生まれてない」はず**よね。

すばらしいね、そのとおりだ。いくつかの特殊な例を除いて、基本的にインスタンスを生み出す方法は new しかない。「new Hero()」した回数が勇者の人数だ。

9.1.8 クラス型をフィールドに用いる

先輩、私もう１つ違うことに気づきました。Heroクラスを定義するとHero型の変数を使えるようになるんですよね。ということは、もしフィールドに…。

朝香さんは本当にせっかちだね。うん、想像のとおりだよ。

朝香さんが気づいたのは、たとえば次のようなコードが書けるのではないかということです。

リスト 9-3　Sword 型フィールドを持つクラス

Sword.java

```
1  // まず、Swordクラスを定義しておく
2  public class Sword {
3    String name;
4    int damage;
5  }
```

剣の名前

剣の攻撃力

Hero.java

```
1  // 次にHeroクラスを定義する
2  public class Hero {
3    String name;
4    int hp;
5    Sword sword;
6    void attack() {
7      System.out.println(this.name + "は攻撃した！");
8      System.out.println("敵に5ポイントのダメージをあたえた！");
9    }
10 }
```

勇者が装備している剣の情報

Hero クラスに新しく追加されたフィールド「sword」は、int 型や String 型ではなく Sword 型です。このように、フィールドにクラス型の変数を宣言すること

もできます。なお、今回の例のように「あるクラスが別のクラスをフィールドとして利用している関係」を has-a の関係と言い、次のような図で表すことがあります。

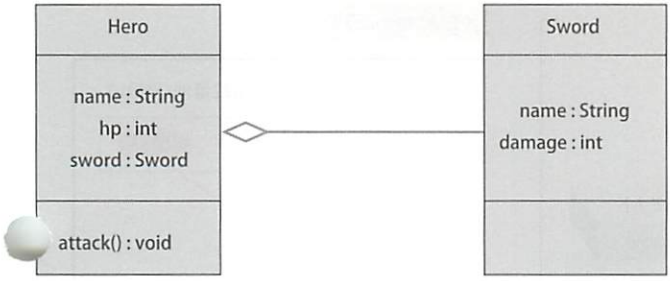


図 9-8　クラス図における has-a の関係（集約関係）

「has-a」と呼ぶ理由は、次のような英文が自然に成立するからです。

Hero has-a Sword（勇者は剣を持っている）

さて、実際にリスト 9-3 を利用する Main クラスは、次のようなものになるでしょう。

リスト 9-4

Main.java

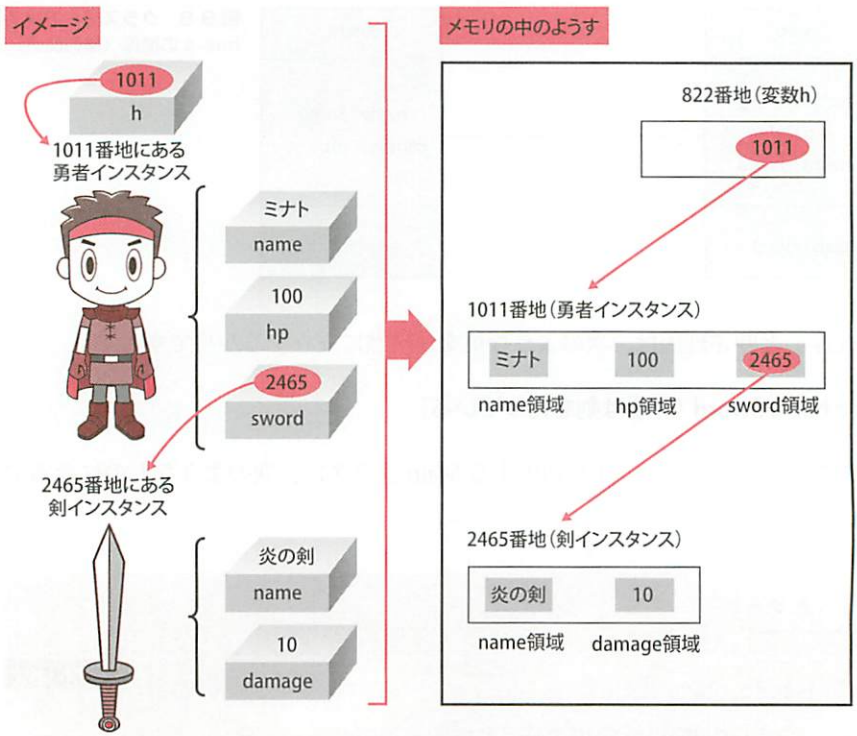
```
1  public class Main {
2    public static void main(String[] args) {
3      Sword s = new Sword();
4      s.name = "炎の剣";
5      s.damage = 10;
6      Hero h = new Hero();
7      h.name = "ミナト";
8      h.hp = 100;
9      h.sword = s;
10     System.out.println("現在の武器は" + h.sword.name);
11   }
12 }
```

sword フィールドに生成済みの剣インスタンス（の番地）を代入

勇者「の」剣「の」名前

10行目は少し複雑ですが、このときのメモリのようすをしっかりイメージしてみましょう。次の図9-9を見てください。

図9-9　has-a関係にあるインスタンスのイメージ



822番地にある変数hには、勇者インスタンスのアドレス情報(1011番地)が入っています。そして勇者インスタンスに含まれるsword領域には、剣インスタンスのアドレス情報(2465番地)が格納されています。すなわち先ほどの「Hero has-a Sword」の関係が成立しており、Heroクラスが Swordクラスをフィールドとして利用していることがわかります。

9.1.9 クラス型をメソッド引数や戻り値に用いる

クラス型はフィールドの型として用いることができるだけではなく、メソッドの引数や戻り値の型として利用することもできます。例として、すでにある勇者クラスに加え、魔法使い(Wizard)のクラスを作ってみましょう。魔法使いは、勇者のHPを回復させる魔法(heal)を使うことができます。

リスト 9-5

Wizard.java

```
1  public class Wizard {
2    String name;
3    int hp;
4    void heal(Hero h) {
5      h.hp += 10;
6      System.out.println(h.name + "のHPを10回復した!");
7    }
8  }
```

引数は Hero 型

勇者の HP に 10 を加える

heal() メソッドが呼び出されると、魔法使いインスタンスは勇者のHPを10回復させます。ただし、仮想世界には勇者が2人以上生み出されている(＝2回以上newされている)可能性もありますから、呼び出されるときに**「どの勇者を回復させるのか」を引数hとして受け取る**必要があります(リスト9-5の4行目)。実際に、このWizardクラスを利用したプログラムは以下のようになります。

リスト 9-6

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h1 = new Hero();
4      h1.name = "ミナト";
5      h1.hp = 100;
6      Hero h2 = new Hero();
```

```
7       h2.name = "アサカ";
8       h2.hp = 100;
9       Wizard w = new Wizard();
10      w.name = "スガワラ";
11      w.hp = 50;
12      w.heal(h1);     // ミナトを回復させる（HP: 100 → 110）
13      w.heal(h2);     // アサカを回復させる（HP: 100 → 110）
14      w.heal(h2);     // アサカを回復させる（HP: 110 → 120）
15    }
16  }
```

9.1.10 String 型の真実

ではこの節の最後に、文字列を利用しているとき JVM の中で何が起こっているか、タネあかしをしよう。

第 1 章から登場していた String 型ですが、実は **int 型や double 型の仲間ではなく、Hero 型と同じ「クラス型」**です。これまで、「String 型変数の中には文字列がそのまま入っている」と解釈していたかもしれませんが、本当の姿は次の図 9-10 ようなものです。

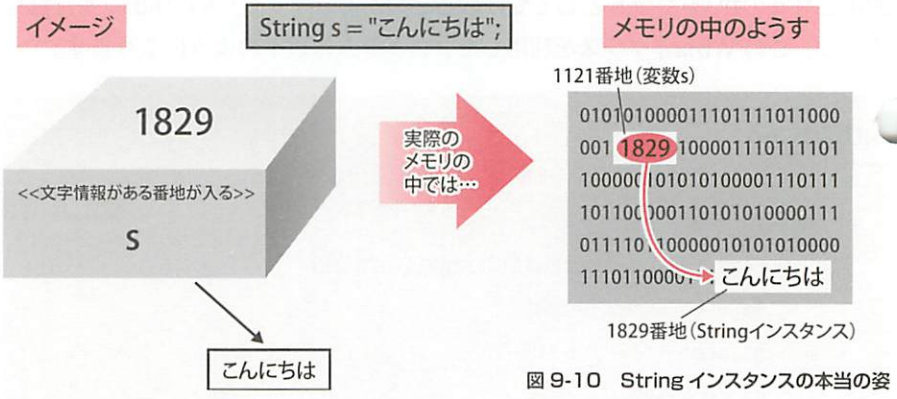


図 9-10　String インスタンスの本当の姿

しかし、疑問も残ります。Hero 型や Sword 型は私たち自身が Hero クラスや Sword クラスを定義したので利用可能になりました。しかし、String クラスを定

義した覚えはないにも関わらず私たちはString型を利用できているのはなぜでしょうか？　この答えはAPIリファレンスが明らかにしてくれます。Mathクラスや Systemクラスのように APIとして標準添付されている膨大な数のクラスの中に Stringクラス（正式名称は java.lang.Stringクラス）が含まれています。

図 9-11　Stringクラスの APIリファレンス

java.lang

クラス String

java.lang.Object
└ java.lang.String

すべての実装されたインタフェース:
Serializable, CharSequence, Comparable<String>

public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence

Stringクラスは文字列を表します。Javaプログラム内の「abc」などのリテラル文字列はすべて、このクラスのインスタンスとして実行されます。

私たちがこれまで String型を「int型と似たようなもの」として扱い続け、本章に至るまで「実は Javaが準備してくれていたクラスを利用していた」と気づかなかった（気づかずに済んだ）理由は、Javaという言語が作られるときに、次のような特別の配慮がなされたためです。

java.lang パッケージに宣言されている：

第6章で紹介したとおり、java.langパッケージに所属するクラスを利用する際は、特例として import文を記述する必要がありません。この特例のおかげで、本来「java.lang.String s;」と宣言する必要があるところを、単に「String s;」と書けば利用できるようになっています。

二重引用符で文字列を囲めばインスタンスを生成・利用できる：

通常、インスタンスを生成するには new演算子を利用する必要があります。しかし文字列はプログラムの中で多用されるため、その都度 newを書いていてはソースコードが「newだらけ」になってしまいます。そこで、「二重引用符で文

字列を囲めば、その文字列情報を持ったStringインスタンスを利用できる」という特例が設けられました。この特例のおかげで、new演算子を使うことなく「String s = " こんにちは ";」というシンプルな記述が可能になっています。

実は、Stringもクラスには違いないのですから、HeroやSwordと同じように「newでインスタンス生成」することもできます（リスト9-7）。ただし、この方法は効率が悪いので、通常は利用しないでください。

リスト9-7

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String s = new String("こんにちは");
4      System.out.println(s);
5    }
6  }
```

画面に「こんにちは」と表示される

ん？　このコードのnewって少し文法がおかしくありませんか？　今までの書き方だと「new String();」と書くのでは？

そうだね。実はStringクラスは**「newするとき、ついでに追加情報を指定できる特別なしくみ」**に対応しているんだ。Stringはインスタンスが生成された直後に、この追加情報（＝「こんにちは」）を自分自身の中に書き込むことができるんだよ。

ひょっとして、その「追加情報付きでnewできるしくみ」はボクたちのHeroクラスやSwordクラスでも使えますか？

うん。とても便利な機能だから、次の節でしっかり学習しよう。

9.2 コンストラクタ

9.2.1 生まれたてのインスタンスの状態

前節をマスターした私たちは、Java の仮想世界にインスタンスを自由に生み出して利用することができるようになりました。しかし、実際にクラスを用いたプログラミングをし始めると、ある「めんどうさ」を感じるようになるはずです。その例として、9.1.9 項に登場したリスト 9-6 を再び次に示します。

リスト 9-6 （抜粋）

```
1   :
2   Hero h1 = new Hero();
3   h1.name = "ミナト";
4   h1.hp = 100;
5   Hero h2 = new Hero();
6   h2.name = "アサカ";
7   h2.hp = 100;
8   Wizard w = new Wizard();
9   w.name = "スガワラ";
10  w.hp = 50;
11  w.heal(h1);
12  w.heal(h2);
13  w.heal(h2);
14  :
```

インスタンス生成して ..
初期値をセット
初期値をセット
ここでもインスタンスを生成して…
また初期値
また初期値
ここでもインスタンスを生成して…
また初期値
また初期値
やっと、ここからメインプログラム

コメントで示したように、new でインスタンスを生成した直後、必ずフィールドの初期値を代入しています。なぜなら new で生み出されたばかりのインスタンスのフィールド（name や hp）には、まだ何も入っていないからです。厳密にいえば、各フィールドには値が「入っていない」わけでなく、次のような初期値が設定されています。

表 9-1 フィールドの初期値

int 型 , short 型 , long 型 等の数値の型	0
char 型（文字）	¥u0000
boolean 型	false
int[] 型などの配列型	null
String 型などのクラス型	null

生まれたばかりの勇者の HP って 0 なんですね。こんな状態で、そのまま冒険に出たら大変だ…。

すぐに死んじゃうね（笑）。

9.2.2 フィールド初期値を自動設定する

こらぁミナトーっ！ やっと見つけた（怒）！

…ど、どうしたの朝香さん。そんなに息切らして。

あなたの作った Hero クラスを使ってプログラムを組んだら、ゲーム開始直後に HP が 0 になってるの！ いきなり死んでるじゃない。こんなバグだらけの Hero クラス、使わせるんじゃないわよ！

勇者の初期 HP は 100 って決まってるんだ。だから朝香さんのほうで new した後に 100 を代入してよ。

そんなの今、初めて聞いたわよ！ っていうか**「勇者として初期 HP=100 と決まってる」**なら自動的にそうなるように Hero クラス側で責任を持ちなさいよ！

まぁまぁ落ち着いて。そんなときのために最適な Java のしくみがあるよ。

湊くんが制作している RPG では、どうやら生まれたばかりの勇者の HP は常に 100 とする決まりのようです。実際に前節のリスト 9-6 では、「ミナト」「アサカ」の 2 人の勇者は共に、new の直後に HP の初期値として 100 を代入していました。

しかし、実際の開発現場において、特にゲームなど大規模な開発を行う場合、1 人ですべてを開発することは、まずありえません。そのためクラスを作るにあたっては、自分以外の開発者が Hero クラスを利用することも考えておかなければなりません。そして、その**開発者が正しく HP に 100 を代入してくれるとは限らないのです**。朝香さんのように、うっかり初期化し忘れるかもしれませんし、Hero クラスを作った人が想定しないような数、たとえば負の数や非常に大きな数で初期化してしまうかもしれません。これではゲームは正しく動作しないでしょう。

「勇者の初期 HP が 100 であること」は、仮想世界における勇者自身に関することですし、Hero クラスの開発者が一番よく知っていることです。逆に、Hero クラスを使う側にとっては、「初期値が何であるか」は知らないのが当然のこととも言えます。ですから「Hero クラスを作る側で責任を持つべきこと」という朝香さんの主張はもっともな話です。

このような場合に備え、Java では**「インスタンスが生まれた直後に自動実行される処理」をプログラミングできる**ようになっています。次ページのリスト 9-8 を見てください。

リスト 9-8

Hero.java

```
1  public class Hero {
2    int hp;
3    String name;
4      :
5    void attack() {
6      :
7    }
8    Hero() {
9      this.hp = 100;     // hpフィールドを100で初期化
10   }
11 }
```

「new された直後に自動的に実行される処理」を書いたメソッド

このクラスには 8 行目から Hero() というメソッドが追加されています。attack() などの通常のメソッドは「誰かから呼ばれないと動かない」ものですが、この Hero() メソッドだけは、「このクラスが new された直後に自動的に実行される」という特別な性質を持っています。このようなメソッドをコンストラクタ（constructor）と呼びます。上記の Hero() メソッドはコンストラクタとして定義されており、new されると自動的に実行されて HP に 100 が代入されます。そのため main メソッド側で HP に初期値を代入する必要はありません。

リスト 9-9

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h = new Hero();
4
5
6      System.out.println(h.hp);
7    }
```

この指示によりインスタンスが生まれ、さらにコンストラクタの働きで HP に 100 が代入される

すでに hp には 100 が代入されているので、画面に「100」と表示される

```
8 }
```

よかった！　これで誰が勇者インスタンスを生成しようと、生まれたばかりの勇者は、必ずHPが100になりますね！

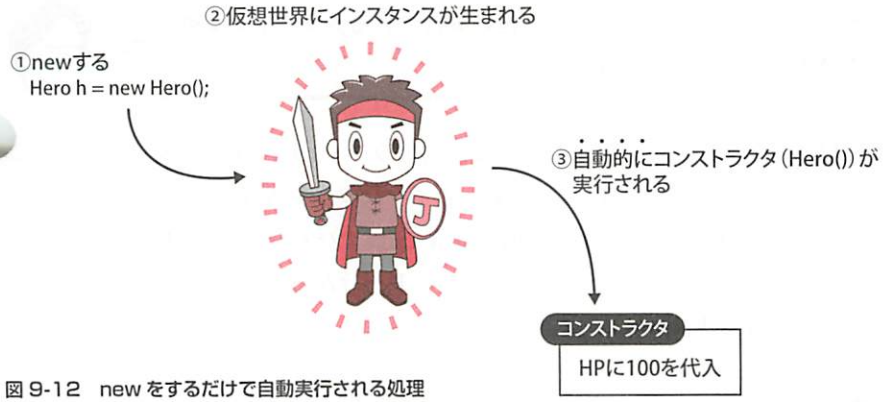


図 9-12　new をするだけで自動実行される処理

ここで意識しておいてほしいのは、**「コンストラクタは、私たちプログラマが直接呼び出すものではない」**という点です。私たちが直接行うことはあくまでも「Hero h = new Hero();」でインスタンスを生成することであって、それによって**間接的に Hero() が実行**されるのです。私たちが main メソッドなどの中から「h.Hero();」のようにコンストラクタを直接呼び出すことはできません。

そういえば現実世界における「赤ちゃん」も、生まれた直後に自分から泣き出しますよね。

そうだね。もし人間にコンストラクタが定義されているとしたら、そこには「泣く ();」って書いてあるだろうね。

9.2.3 コンストラクタの定義方法

先輩、このクラスには attack() など、ほかにもたくさんメソッドがあるのに、なぜ Hero() メソッドだけが自動実行されるんですか？

それは Hero() メソッドだけが「自動実行されるメソッドの条件」を満たしているからだよ。

一見すると、Hero() メソッド（＝コンストラクタ）も、ほかのメソッドと違いがないように見えます。しかし、new でインスタンスを生成したときに自動実行されるのは Hero() だけです。実は Java では、クラスに記述されているメソッドのうち、以下の条件をすべて満たすメソッドだけがコンストラクタと見なされ、自動実行される決まりになっています。

コンストラクタと見なされる条件

①メソッド名がクラス名と完全に等しい
②メソッド宣言に戻り値が記述されていない（void もダメ）

Hero() がコンストラクタとして実行されたのは、「Hero クラス」の中に「Hero()」という完全に同名で定義されており、その戻り値が記述されていないからです。

コンストラクタの基本書式

```
1 public class クラス名 {
2   クラス名() {
3     ⋮  ← ここに自動実行処理を記述する
4   }
5 }
```

フィールド宣言による初期値の定義

フィールドの値を固定の値に初期化するだけでよければ、コンストラクタを用いなくても、フィールド宣言時に代入式を書くことでも実現可能です。しかし、次項で説明するような複雑な初期化を行いたい場合は、コンストラクタを使わなければ実現できません。

9.2.4 コンストラクタに情報を渡す

自動的に勇者の HP が 100 になったのは嬉しいんですが、「名前」は自動的に代入できないんですかねぇ？

new で生み出す勇者全員に「ミナト」って同じ名前が入っちゃうじゃない？ でも勇者の名前って、それぞれ違うわよね？

HP フィールドは「100」という固定の値で初期化すればよいため、単純なコンストラクタで済みました。しかし、勇者の名前は生み出すインスタンスによって異なるはずです。このような場合は次のリスト 9-10 のように、コンストラクタが「毎回異なる追加情報」を引数で受け取れるように宣言することができます。

リスト 9-10　コンストラクタで引数を追加情報として受け取る

Hero.java

```
1  public class Hero {
2    :
3    Hero(String name) {        引数として文字列を 1 つ受け取る
4      this.hp = 100;
5      this.name = name;     // 引数の値でnameフィールドを初期化
6    }
7  }
```

これで、Hero クラスは new するときに名前の初期値も指定できるようになりました。

でも菅原さん、私たちはコンストラクタを「直接呼び出せない」はずですよね？（9.2.2 項参照）。メソッドのように呼び出すときに引数を渡せないとしたら、どうやって引数を渡せばいいんですか？

いい質問だね。代わりに new するときに渡しておくんだよ。

このような Hero クラスを利用する場合は、コンストラクタに渡すべき引数を new する際に指定します。次のリスト 9-11 では、new をした時点で与えられた「ミナト」という引数が、コンストラクタ Hero() が自動実行される際にパラメータとして渡されます。

リスト 9-11　new する際に引数を渡す

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Hero h = new Hero("ミナト");
4
5     System.out.println(h.hp);
6     System.out.println(h.name);
7   }
8 }
```

こう書いておけばコンストラクタには「ミナト」が渡される

100 と表示される

ミナトと表示される

この処理のようすを図で表すと、次の図 9-13 のようになります。

図 9-13　new 時に指定した引数が、コンストラクタ実行時に利用される

9.2.5 2つ以上の同名コンストラクタを定義する

Hero にコンストラクタができて「new Hero(" ミナト ");」のようにできるようになったのは確かに便利です。ただ、簡単な動作テストなど「別に名前はどうでもいい」ときもあって、単に「new Hero();」としたい場合にはどうすればいいんでしょうか？

なるほど。そういうときに、めんどうだから「new Hero();」としたいのに、引数がないためにエラーが出てしまうんだね。

現在の Hero クラスには、「文字列引数を 1 つ受け取るコンストラクタ」が定義されています。そしてコンストラクタが「new された際に必ず自動的に実行されるもの」である以上、「new をする側としては、必ず引数となる文字列を 1 つ与える」必要があります。

つまり、このコンストラクタを作ったことによって、**インスタンスを生成するときには、必ず名前を指定する必要が生じた**わけです。試しに、朝香さんの言うように、引数なしで「new Hero();」を実行するとエラーになります。

この問題は**「引数を受け取らないコンストラクタ」も同時に定義することで解決**できます。次ページのリスト 9-12 を見てください。

リスト 9-12　コンストラクタのオーバーロード

Hero.java

```
1  public class Hero {
2    :
3    Hero(String name) {
4      this.hp = 100;
5      this.name = name;
6    }
7    Hero() {
8      this.hp = 100;
9      this.name = "ダミー";
10   }
11 }
```

以前からあったコンストラクタ①

新しく作ったコンストラクタ②

新ダミーの名前を設定する

先輩。これって、第 5 章で習ったオーバーロード（5.4 節）ですよね？

そのとおりだよ。「同じ名前だが引数が異なるメソッドを複数定義」するオーバーロードは、コンストラクタでも可能なんだ。

ということは、実行時に「どちらが動くか」は Java が空気を読んで判断してくれるんですね。

複数のコンストラクタが定義されていた場合

new するときに渡した引数の型・数・順番に対応するコンストラクタが動作する（複数のコンストラクタが定義されていても、1 つだけしか動作しない）。

コンストラクタのオーバーロードを実際に活用する例が、次のリスト 9-13 です。

リスト 9-13　コンストラクタをオーバーロードしたクラスの利用

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h1 = new Hero("ミナト");
4
5      System.out.println(h1.name);
6      Hero h2 = new Hero();
7
8      System.out.println(h2.name);
9    }
10 }
```

文字列引数があるのでコンストラクタ①が呼び出される

画面に「ミナト」と表示される

引数がないのでコンストラクタ②が呼び出される

画面に「ダミー」と表示される

9.2.6 暗黙のコンストラクタ

ところで、コンストラクタが定義されていない Hero クラスは「new Hero();」で生成できたのに、引数ありコンストラクタを定義しただけで同様のことが不可能になったのはなぜでしょうか。

実は Java では、すべてのクラスはインスタンス化に際して**必ず何らかのコンストラクタを実行すること**になっています。ですから、**本来すべてのクラスは**、「引数のない、何も処理をしないコンストラクタ」でよいので、**最低でも 1 つ以上のコンストラクタ定義を持っていなければなりません**。コンストラクタが 1 つも定義されていないクラスは許されないのです。

えぇ～！ そんなのめんどうですよ。次に開発する予定の「宝の地図」クラス（Map クラス）はインスタンス化の直後に初期化する必要はないんです。

ルールに従うためだけに中身のないコンストラクタを定義しないとならないとしたら不便ね。

リスト 9-14　すべてのクラスは必ずコンストラクタを定義しなければならない？

Map.java

```
1  public class Map {
2    ⋮
3    Map() {
4    }
5  }
```

new 時に自動実行したいことは何もないが、しかたなくダミーのコンストラクタを定義

上記のリスト 9-14 のように、わざわざダミーでコンストラクタを定義するのはめんどうですので、Java では以下のような特例を設けています。

コンストラクタの特例

クラスに 1 つもコンストラクタが定義されていない場合に限って、「引数なし、処理内容なし」のコンストラクタ（デフォルトコンストラクタ）の定義がコンパイル時に自動的に追加される。

これまでサンプルで示してきた Hero クラスは、**この特例によって引数なしコンストラクタがこっそり自動定義されたため、「new Hero();」によるインスタンス化が可能だった**のです。ですが、新たに引数を 1 つ含むコンストラクタを定義した時点でこの特例は適用除外となり、「new Hero();」という記述によるインスタンスの生成はできなくなったのです。

9.2.7 ほかのコンストラクタを呼び出す

菅原さん、コンストラクタを複数作っていて、少し「気持ち悪い」ことがあるんですけど。

「複数のコンストラクタに、同じ処理をいくつも書いてること」だね？

2つ以上のコンストラクタを定義していると、重複する処理を記述することもあるでしょう。たとえば、9.2.5項のリスト9-12をもう一度見てください。コンストラクタ①とコンストラクタ②の内容には重複があります。

2つのコンストラクタは、どちらもHPに100を代入しています。しかし、もし将来「初期HPを200に変更する」ことになったら、コンストラクタ①と②の両方を修正する必要が生じます（この例は極めてシンプルですが、本格的なプログラムにおいて、重複部分はさらに多くなることが一般的です）。

そこで思いつくのが、コンストラクタ②の中で、コンストラクタ①を呼び出す方法です。たとえば、次のリスト9-15のようなコードでHPへの代入を1か所に集中させようと考えるかもしれません。

リスト9-15　コンストラクタの中から別のコンストラクタを呼び出す（エラー）

Hero.java

```
1  public class Hero {
2    ⋮
3    Hero(String name) {
4      this.hp = 100;
5      this.name = name;
6    }
7    Hero() {    // コンストラクタ②
8      this.Hero("ダミー");
9    }
10 }
```

コンストラクタ①

コンストラクタ①を呼び出したいが、この行はエラーになるのでダメ！

しかし残念ながら、このコードはコンパイルエラーになります。なぜなら、Java ではコンストラクタを直接呼び出すことができないからです。ただし、この制限には例外があります。それは**「専用の文法を用いて、コンストラクタの先頭で別のコンストラクタを呼び出す場合に限って」特別に許される**ということです。

別コンストラクタの呼び出しに関するルール

「this. クラス名 (引数);」と記述することはできない。
その代わりに「this(引数);」と記述する。

よって、リスト 9-15 の Hero クラスを正しく記述するには、次のリスト 9-16 のようにします。

リスト 9-16　コンストラクタの中から別のコンストラクタを呼び出す（正常に動作）

```
3   Hero(String name) {
4     this.hp = 100;
5     this.name = name;
6   }
7   Hero() {
8     this("ダミー");
9   }
```

コンストラクタ①

コンストラクタ②

コンストラクタ①を呼び出す。

ちなみに、今回学んだ this() は、今まで利用してきた this と見た目が似ているが、何の関係もない別物と考えたほうがいい。「this. メンバ名」の this は自分自身のインスタンスを表すもの。「this(引数)」の this() は同一クラスの別コンストラクタを呼び出すためのものだ。

9.3 静的メンバ

9.3.1 クラス上に準備されるフィールド

前節までで、この章で最も大切なコンストラクタの解説は終わりだよ。ところで「インスタンスの独立性」について学んだのは覚えているかな？

はい。別々の new で生まれた各インスタンスのフィールドの中身は別、ということですよね。

そうだ。第 9 章の締めくくりに、その独立性の例外を紹介しておこう。

new によって生成される個々のインスタンスは基本的に独立した存在です（9.1.7 項「インスタンスの独立性」参照）。よって、各勇者インスタンスが持つ同名のフィールド hp には、それぞれ別の値を格納することができます（図 9-14）。

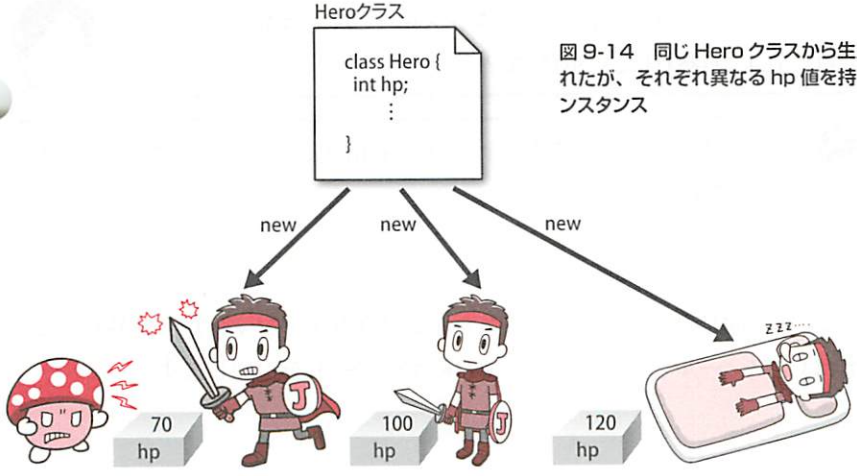


図 9-14　同じ Hero クラスから生成されたが、それぞれ異なる hp 値を持つインスタンス

たとえば勇者３名がチーム（パーティ）を組んで冒険するRPGを作る場合を考えましょう。次のようなクラスから生成されるインスタンスでは、それぞれが名前フィールド（name）やHPフィールド（hp）、そして所持金フィールド（money）を別々に持つことになります。

リスト9-17　同じクラスから作られても、個々のインスタンスは別々のフィールドを持つ

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4    int money;
5    ⋮
6  }
```

しかし、プログラムを開発していると、「各インスタンスで共有したい情報」が出てくることがあります。たとえばRPGなら、チームを組んで行動しているので「チーム全員で１つのお財布」を設定したい場合もあるでしょう。

各Heroインスタンスごとの財布は不要で、すべての勇者で１つの財布を共有すればいいってことですね。

そのとおりだよ。財布は「インスタンスに１つ」ではなく、「何に対して１つ」あればいいのかな？

「すべての勇者で１つ」つまり「Heroというクラスに対して１つ」ですね。

このように同じクラスから生成されたインスタンスでフィールドを共有したい場合には、フィールド宣言の先頭にstaticキーワードを追加します。

リスト 9-18　static キーワードによるフィールドの共有

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4    static int money;
5    ⋮
6  }
```

静的フィールド

static キーワードを指定したフィールドは特に**静的フィールド**（static field）といわれ、下記のような 3 つの特殊な効果をもたらします。

1. フィールド変数の実体がクラスに準備される

通常、フィールドが格納される箱（領域）は個々のインスタンスごとに用意されますが、静的フィールドの箱はインスタンスではなく、クラスに対して 1 つだけ用意されます。イメージで考えるならば、図 9-15 のように「勇者の金型」の上に money の箱が準備されるというイメージになります。

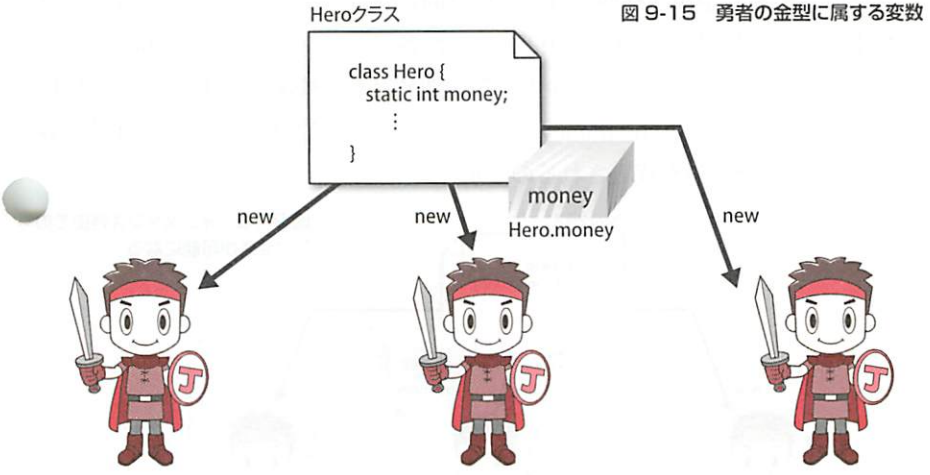


図 9-15　勇者の金型に属する変数

この Hero クラスに準備された箱（静的フィールド「money」）を読み書きするには、「Hero.money」という表記を使います。

静的フィールドへのアクセス方法

クラス名 . 静的フィールド名

リスト 9-19　静的フィールド money へのアクセス

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h1 = new Hero();
4      Hero h2 = new Hero();
5      System.out.println(h1.hp);
6      System.out.println(Hero.money);
7      ⋮
```

インスタンス h1 の箱 hp を表示

クラス Hero の箱 money を表示

2. 全インスタンスに、箱の分身が準備される

共通財産である金額が格納される変数（Hero.money）は、あくまでも金型に作られます。しかし同時に、h1 や h2 といった各インスタンスにも money という名前で「箱の分身」が準備され、金型の箱の別名として機能するようになります。つまり、「h1.money」や「h2.money」という分身の箱に値を代入すれば、本物の箱 Hero.money にその値が代入されるのです。

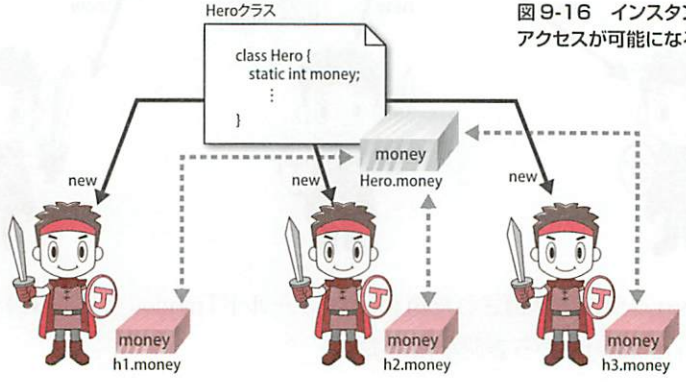


図 9-16　インスタンス経由でのアクセスが可能になる

実質的に「h1.money」「h2.money」「Hero.money」はどれも同一の箱を指すことになるんですね。

静的フィールドへの別名によるアクセス

「インスタンス変数名.静的フィールド名」と書いてもよいが、「クラス名.静的フィールド名」と同じ意味になる。

リスト 9-20　個々のインスタンスから静的フィールド money へアクセスして内容を表示

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h1 = new Hero();
4      Hero h2 = new Hero();
5      Hero.money = 100;
6      System.out.println(Hero.money);
7      System.out.println(h1.money);
8      h1.money = 300;
9      System.out.println(h2.money);
10   }
11 }
```

6 行目：100と表示
7 行目：100と表示
8 行目：h1.moneyに300を代入
9 行目：h2.moneyでも300と表示

このような状態は、「h1 と h2 が money フィールドを共有している」と考えることもできるので、「静的フィールドを用いれば、インスタンス間でフィールドを共有できる」と解説されることもあります。

3. インスタンスを 1 つも生み出さなくても箱が利用可能になる

「Hero.money」は金型の上に作られる箱です。よって、まだ 1 つの実体（インスタンス）も生み出されていない状況であっても利用することができます。

リスト 9-21　インスタンスが生成されていなくても静的フィールドにアクセスできる

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      // 1人も勇者を生み出していない状況で…
4      Hero.money = 100;
5      System.out.println(Hero.money);
6    }
7  }
```

なお、静的フィールドはクラス（金型）にフィールド（箱）が所属するという特徴から、**クラス変数**と言われることもあります。

なんだか static って、ちょっと複雑な効果があるキーワードですね。

慣れれば大したことはないんだけども、もし混乱しそうなら理解は後回しでもいいよ。確かに少しややこしい割には、実際の開発では頻繁に使われるものではないんだ。それよりはコンストラクタの理解のほうが何倍も大事だよ。

public static final コンビネーション

多くの場合、static は final（第 1 章 1.3.5 項）や public（10 章で学習）と一緒に指定され、「変化しない定数を各インスタンスで共有するため」に利用されます。

9.3.2 静的メソッド

先輩、static といえば私たちが今まで使ってきた main メソッドにも付いていますよね？

そうだね。static はメソッドにも付けられるんだよ。

Hero クラスに、「勇者たちの所持金をランダムに設定する」setRandomMoney() メソッドを追加する場合、リスト 9-22 のようなコードを記述します。

リスト 9-22　静的なメソッドの例

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4    static int money;
5    ⋮
6    static void setRandomMoney() {
7      Hero.money = (int) (Math.random() * 1000);
8    }
9  }
```

static を付けたメソッド

static キーワードが付いているメソッドは、**静的メソッド**（static method）または**クラスメソッド**（class method）と呼ばれ、静的フィールドとあわせて**静的メンバ**（static member）と総称されます。静的メソッドを定義すると静的フィールドと同様に以下の 3 つの効果が現れます。

1. メソッド自体がクラスに属するようになる

静的メソッドは、その実体が各インスタンスではなくクラスに属し、「クラス名 . メソッド名 ();」で呼び出せるようになります。

2. インスタンスにメソッドの分身が準備される

静的メソッドは、「インスタンス変数名 . メソッド名 ();」でも呼び出せるようになります。

3. インスタンスを 1 つも生み出すことなく呼び出せる

静的メソッドは、1 つもインスタンスを生み出していない状況であっても、呼び出すことができます。

リスト 9-23　静的メソッドの呼び出し

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero.setRandomMoney();
4      System.out.println(Hero.money);
5      Hero h1 = new Hero();
6      System.out.println(h1.money);
7    }
8  }
```

4 行目：ランダムな金額が表示

6 行目：同じ額を表示

ここまで説明すれば、main メソッドがなぜ static でなければならないのか、想像つくんじゃないかな？ ヒントは「3 番目の効果」だ。

main メソッドが呼び出されるとき、仮想世界にはまだ 1 つもインスタンスが存在していないからですね。main メソッドが属するクラス（Main クラスなど）さえもまだインスタンス化されていない状況で、main は呼び出される必要がありますから。

9.3.3 静的メソッドの制約

実は静的メソッドの利用には重要な制約があるんだ。これを忘れて開発時に悩んでしまう人も少なくない。

静的メソッドの中に記述するコードでは、**staticが付いていないフィールドやメソッドは利用できません**。次のリスト9-24を見てください。

リスト9-24　静的なメソッド中でアクセスできるのは静的メンバだけ

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4    static int money;
5    ⋮
6    static void setRandomMoney() {
7      Hero.money = (int) (Math.random() * 1000);
8      System.out.println(this.name + "たちの所持金を初期化しました");
9    }
10 }
```

エラー

静的メソッドsetRandomMoney()の内部である8行目で、フィールドnameへアクセスしようとしていますが、この処理はエラーとなります。

静的メソッドsetRandomMoney()は、**まだ1つも勇者インスタンスが存在しない状況でも呼び出されることがあるメソッド**です。もし仮想世界に1つも勇者インスタンスがない状況でsetRandomMoney()が動いてしまったら、「自分自身のインスタンス」のメンバである「this.name」をうまく処理できないことは明らかです。ですから静的メソッド内部では、静的メンバしか利用できないことになっているのです。

9.4 第9章のまとめ

この章では、次のようなことを学びました。

クラス型と参照

- クラス型変数の中には、「インスタンスの情報が格納されているメモリ番地」が入っている。
- あるクラス型変数を別変数に代入すると、番地情報だけがコピーされる。
- クラス型は、フィールドやメソッドの引数・戻り値の型としても利用できる。

コンストラクタ

- 「クラス名と同一名称で、戻り値の型が明記されていないメソッド」はコンストラクタとして扱われる。
- コンストラクタは、new によるインスタンス化の直後に自動的に実行される。
- 引数を持つコンストラクタを定義すると、new をする際に引数を指定してコンストラクタを実行させることができる。
- コンストラクタはオーバーロードにより複数定義できる。
- クラスにコンストラクタ定義が1つもない場合に限って、コンパイラが「引数なし・処理内容なし」のデフォルトコンストラクタを自動定義してくれる。
- this () を用いれば、同一クラスの別コンストラクタを呼び出すことができる。

静的メンバ

- static キーワードが付いている静的メンバ（フィールドおよびメソッド）は、
  ① 各インスタンスにではなく、クラスに実体が準備される。
  ②「クラス名 . メンバ名」、「インスタンス変数名 . メンバ名」のどちらでも同じ実体にアクセスすることになる。
  ③ 1つもインスタンスを生み出していなくても利用可能である。
- 静的メソッドは、その内部で静的ではないメソッドやフィールドを利用することができない。

9.5 練習問題

練習 9-1

第 8 章の練習問題で作成した Cleric クラスに関して、以下の 2 つの修正を行ってください。

① 現時点の Cleric クラスの定義では、各インスタンスごとの最大 HP と最大 MP フィールドに情報を保持します。しかし、すべての聖職者の最大 HP は 50、最大 MP は 10 と決まっており、各インスタンスがそれぞれ情報を持つのはメモリのムダです。

そこで、最大 HP・最大 MP のフィールドが各インスタンスごとに保持されないように、フィールド宣言に適切なキーワードを追加してください。

② 以下の方針に従って、コンストラクタを追加してください。

A)このクラスは、new Cleric(" アサカ ", 40, 5) のように、名前・HP・MP を指定してインスタンス化することができます。

B)このクラスは、new Cleric(" アサカ ", 35) のように、名前と HP だけを指定してインスタンス化することもできます。この場合、MP は最大 MP と等しい値で初期化されます。

C)このクラスは、new Cleric(" アサカ ") のように、名前だけを指定してインスタンス化することもできます。この場合、HP と MP は最大 HP と最大 MP で初期化されます。

D)このクラスは、new Cleric() のように、名前を指定しない場合にはインスタンス化することはできないものとします(名前がない Cleric は仮想世界に生み出せない)。

E) コンストラクタは極力重複コードをなくすように記述します。

9.6 練習問題の解答

練習 9-1 の解答

(注)静的フィールド宣言とコンストラクタ宣言部のみを掲載してあります。

```
1  static final int MAX_HP = 50;
2  static final int MAX_MP = 10;
3
4  public Cleric(String name, int hp, int mp) {
5    this.name = name;
6    this.hp = hp;
7    this.mp = mp;
8  }
9  public Cleric(String name, int hp) {
10   this(name, hp, Cleric.MAX_MP);
11 }
12 public Cleric(String name) {
13   this(name, Cleric.MAX_HP);
14 }
```