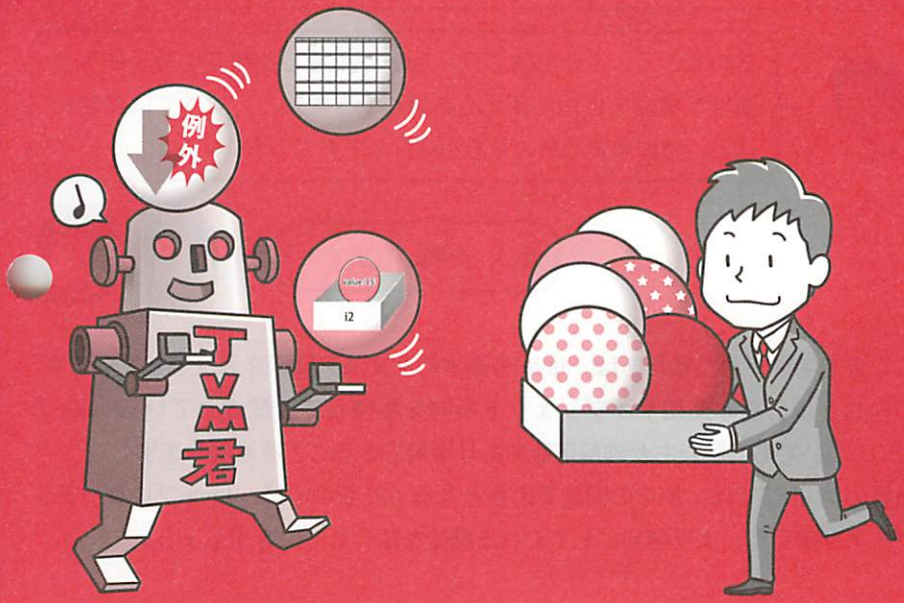
第Ⅲ部

もっと便利にAPI活用術

第14章　Javaを支える標準クラス
第15章　例外
第16章　まだまだ広がるJavaの世界



もっと楽しく、もっと便利に

読める…！　読めるぞっ…！！

どっ、どうしたの。急にPCの画面を指でなぞったりし始めちゃって…。

なるほど、第I部（p.247）で少しだけ紹介したJavaのAPIリファレンスだね。

はい。あのときもこれを読もうとしたんですが、staticとかextendsとか出てきて全然意味がわかんなかったんです。でも今は、APIを見ていたら意味や使い方が、ある程度ならば推測できるんです！

おめでとう。ふたりはもう、Javaが用意した数万に及ぶ命令や高度な機能を自由に使える準備ができたんだ。すべては無理だけど、特に重要なものたちを紹介しておこうか。

はい、お願いします！

ここまで学び終えた私たちは、Javaの基本的な文法をほぼマスターしています。このことは、基本文法やオブジェクト指向を駆使して作成されているJavaのAPIについても、私たちが理解可能であり活用できることを意味します。
最後の第III部では、多くの開発で必要となるAPIを中心に、Javaプログラミングをより楽しく、より便利にしてくれる機能を紹介していきましょう。

第14章

Javaを支える標準クラス

APIに含まれるクラスや、そのメンバに関する解説は
APIリファレンスに掲載されていますが、
初めてJavaプログラムを学ぶ人にとって、
その解説は理解しやすいものではありません。
そこで本章以降では、Javaが持つ多くのAPIの利用方法を
やさしく紹介していきます。

CONTENTS

14.1 日付を扱う
14.2 すべてのクラスの祖先
14.3 基本データ型をオブジェクトとして扱う
14.4 第14章のまとめ
14.5 練習問題
14.6 練習問題の解答

14.1 日付を扱う

14.1.1 日時情報を扱う 2 つの基本形式

ここからは特に大事な API のクラスたちを紹介していくよ。まずは日付の情報を扱うための Date クラスから紹介しよう。

プログラムを開発していると、日付情報を扱う局面に多く遭遇します。たとえば ATM のプログラムであれば、「取引を行った時刻が、何年何月何日の何時何分何秒であるか」という情報を変数に入れて取り扱う必要があるでしょう。

Java でも日時情報を扱うことができますが、やや複雑な取り扱いが必要です。

そういえば RPG の開発で日付を使いたかったのですが、API リファレンスが難しくて理解できず、結局は諦めてしまいました。

Java では「日付情報」を表すために使う型（クラス）が 1 つではないということが重要なポイントだよ。

Java では日時の情報を表すための形式が 4 つあり、それぞれ用途に合わせて使い分ける必要があります。まずは基本となる 2 つの形式を学びましょう。

形式 1：long 型の数値

基準日時である 1970 年 1 月 1 日 0 時 0 分 0 秒（これを「エポック」といいます）から経過したミリ秒（1/1000 秒）数で日時情報を表現する方法です。たとえば、1316622225935 という long 値は「2011 年 9 月 22 日 1 時 23 分 45 秒」を意味します。

この long 型による形式はシンプルであるため、コンピュータにとってとても扱いやすく、JVM 内部のさまざまな部分で利用されています。たとえば、System.currentTimeMillis() メソッドを呼べば、現在日時を long 型で得られるため、リスト 14-1 のような「処理時間の計測」を簡単に行うことができます。

リスト 14-1

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     long start = System.currentTimeMillis();
4     // ここで何らかの時間がかかる処理
5     long end = System.currentTimeMillis();
6     System.out.println("処理にかかった時間は…"
          + (end - start) + "ミリ秒でした" );
7   }
8 }
```

しかし人間は、この long 値から「年・月・日・時・分・秒」を読み取ることができません。また、long 型は日付情報以外の数値の格納にも利用される型なので、必ずしも変数の中身が日時情報だと断定できないという問題もあります。

形式 2：Date 型のインスタンス

long 型の課題を克服するために広く用いられているのが java.util.Date クラスです。このクラスは、内部で long 値を保持しているだけなのですが、「Date 型の変数であれば、中身は日付情報である」と一目でわかるため、**Java で日時の情報を扱う場合に最も利用される形式**となっています。Date インスタンスを生成して利用するには、以下の構文を用います。

現在日時を持つ Date インスタンスの生成

```
Date d = new Date(); // 現在日時を持つインスタンス生成
```

指定時点の日時を持つ Date インスタンスの生成

```
Date d = new Date( long 値 ); // long 値の日時を持つインスタンス生成
```

Date インスタンスの内部に格納されている long 値を取り出したい場合は getTime() メソッドを、long 値をセットしたい場合は setTime() メソッドを用います。これら Date クラスの構文を用いて現在日時を表示するプログラムがリスト 14-2 です。

リスト 14-2

Main.java

```
1  import java.util.Date;
2  public class Main {
3    public static void main(String[] args) {
4      Date now = new Date();
5      System.out.println(now);
6      System.out.println(now.getTime());
7      Date past = new Date(1316622225935L);
8      System.out.println(past);
9    }
10 }
```

import しておくと便利

現在の日時を取得

実行結果

```
Fri Aug 12 16:05:55 GMT+09:00 2011
1313132755277
Thu Sep 22 01:23:45 GMT+09:00 2011
```

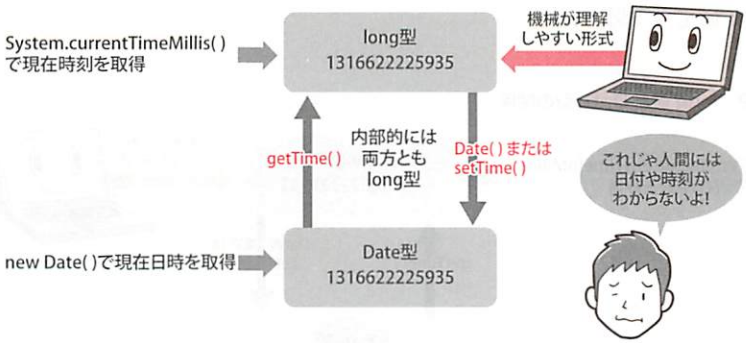
（※実行の日時により以上の日付と数値は変わります）

ここで、これまでに学習した 2 つの形式について図 14-1 に整理します。

java.sql.Date と混同しない

java.util.Date と似たクラスとして、java.sql.Date というクラスがあります。これはデータベースを利用したプログラムで利用するもので、通常の日時情報の格納には使いません。

図 14-1　long 値と Date 型の関係



14.1.2 人間が扱いやすい 2 つの形式

long 値も Date クラスも、結局はエポックからの経過ミリ秒数を扱っていることに違いはなく、人間が日付情報を扱うには不便です。そこで、人間にとって使いやすい 2 つの日時形式を紹介します。

形式 3：人間が読みやすい String 型のインスタンス

人間が読みやすいのは「2011 年 9 月 22 日 1 時 23 分 45 秒」のような文字列としての形式です。画面に時刻を表示する場合、この形式に変換する必要があります。

ただし、一口に文字列といっても、さまざまな形式が考えられる点には考慮が必要です。たとえば「2011/9/22 1:23:45」という形式で表示したいこともあるでしょうし、あるいは「11-09-22 01:23:45AM」と表現したいこともあるでしょう。

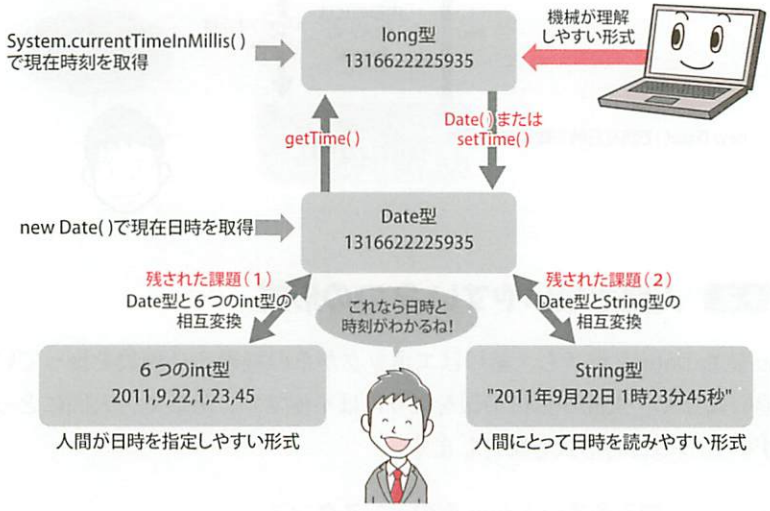
形式 4：人間が指定しやすい「6 つの int」形式

前述の System.currentTimeMiilis() メソッドを使うか、Date クラスを new すれば「現在の時刻」は簡単に得られます。しかし、ある特定の日時情報（たとえば 2011 年 9 月 22 日 1 時 23 分 45 秒）を人間がキーボードやマウスなどで入力する場合には、「年・月・日・時・分・秒」をそれぞれ整数（int 値）として指定することが一般的です。

ここまでで、私たちは日時を表す 4 つの形式を学びました。機械が扱いやす

い形式としての long 型と Date 型、人間が扱いやすい形式としての文字列型と 6 つの int 値です（図 14-2）。

図 14-2　long 値と Date 型の関係



これら 4 つの形式を自由に利用できるようになるために、「(1) Date 型と 6 つの int 値の相互変換」と「(2) Date 型と文字列の相互変換」の 2 つを学びましょう。

14.1.3 Calendar クラスの利用

「課題 (1) Date 型と 6 つの int 値の相互変換」の解決のためには、java.util.Calendar クラスが準備されており、構文は次のとおりです。また、そのサンプルをリスト 14-3 に示します。

「6 つの int 値」から Date インスタンスを生成する

```
Calendar c = Calendar.getInstance();
c.set(年,月,日,時,分,秒); または c.set(Calendar.～,値);
Date d = c.getTime();
```

※～には YEAR、MONTH、DAY_OF_MONTH、HOUR、MINUTE、DAY などを指定する。

Date インスタンスから「6 つの int 値」を生成する

```
Calendar c = Calendar.getInstance();
c.setTime(d);   // d は Date 型変数
int year = c.get(Calendar.YEAR);
int month = c.get(Calendar.MONTH);
int day = c.get(Calendar.DAY_OF_MONTH);
int hour = c.get(Calendar.HOUR);
int minute = c.get(Calendar.MINUTE);
int second = c.get(Calendar.SECOND);
```

リスト 14-3

Main.java

```
1  import java.util.Calendar;
2  import java.util.Date;
3  public class Main {
4    public static void main(String[] args) {
5      // 現在の年を表示する
6      Date now = new Date();
7      Calendar c = Calendar.getInstance();
8      c.setTime(now);
9      int y = c.get(Calendar.YEAR);
10     System.out.println("今年は" + y + "年です");
11     // 指定した日のDate型の値を得る
12     c.set(2010,8,22,1,23,45);
13     c.set(Calendar.YEAR, 2011);
14     Date past = c.getTime();
15   }
16 }
```

年だけを 2011 に変更

「月」の値にご用心

Calendar を用いて「月」の情報を取得・設定する場合には、1 ～ 12 ではなく 0 ～ 11 で指定することになっているため注意が必要です。たとえば、2 月を設定したい場合には、「c.set (Calendar.MONTH, 1)」とします。

14.1.4 SimpleDateFormat クラスの利用

最後に残された「課題 (2) Date 型と文字列型の相互変換」の解決には、簡単な方法と高度な方法があります。「Sun Aug 07 16:38:59 JST 2011」のような文字列（多少、読みづらいですが）でよければ、Date インスタンスの toString() メソッドを呼び出すだけで取得できます。

もし、独自に書式を指定して「2011 年 8 月 7 日 16 時」のような文字列を生成したい場合は、java.text.SimpleDateFormat クラスを用いる必要があります。

Date から String を生成する

```
SimpleDateFormat f = new SimpleDateFormat ( 書式文字列 );
String s = f.format (d);    // d は Date 型変数
```

String から Date を生成する

```
SimpleDateFormat f = new SimpleDateFormat ( 書式文字列 );
Date d = f.parse ( 文字列 );
```

この「書式文字列」には何を指定すればいいんですか？

"yyyy/MM/dd" や "yyyy 年 MM 月 dd 日 " のように、日付の書式を指定するんだよ。

書式文字列に使うことができる主な記号は以下のとおりです。

表 14-1　書式文字列として利用可能な文字（一部）。

文字	意味
y	年
M	月
d	日
E	曜日
a/p	午前／午後
H	時（0 ～ 23）
K	時（0 ～ 11）
m	分
s	秒

次のリスト 14-4 は、SimpleDateFormat を用いて指定した形式で日付情報を表示するコード例です。

リスト 14-4

Main.java

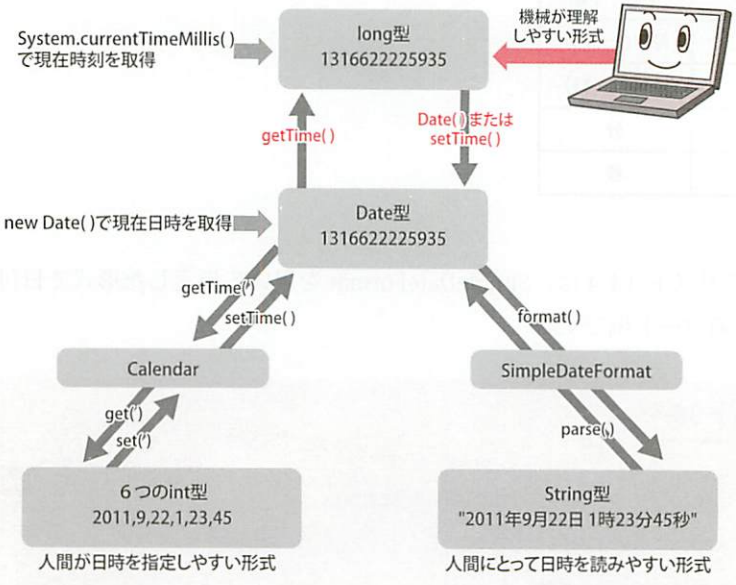
```
1  import java.text.SimpleDateFormat;
2  import java.util.Date;
3  public class Main {
4    public static void main(String[] args) throws Exception {
5      // 本日の日時を表示する
6      Date now = new Date();
7      SimpleDateFormat f =
```

throws については 15 章で解説します

```
            new SimpleDateFormat("yyyy/MM/dd HH:mm:ss" );
 8      String s = f.format(now);
 9      System.out.println(s);
10      // 指定日時の文字列を解析しDate型として得る
11      Date d = f.parse("2011/09/22 01:23:45");
12    }
13  }
```

図 14-3 は 4 つの形式をまとめたものだ。日時情報は Date 型で扱う方法を基本とし、必要に応じてほかの形式に変換して使ってほしい。

図 14-3　long 値と Date 型の関係



ここまで紹介してきたように、Java における日付の取り扱いは少し複雑です。また、あいまいな日時情報や時差情報をうまく扱うこともできません。

Java8 以降では、上記のような課題を克服した新しい日付 API が追加されています。

14.2 すべてのクラスの祖先

14.2.1 暗黙の継承

14.1.4 節では Date インスタンスには toString() というメソッドがあることに触れたね。実は、この toString() というメソッドは、すべてのクラスで利用できるんだ。

すべてって、Hero や Matango にもですか？ でもボクたち、Hero クラスには、そんなメソッドを宣言していませんよ？

湊くんが不思議に思うのもしかたありません。しかし、次のリスト 14-5 は何の問題もなくコンパイルでき、実行が可能です。

リスト 14-5

Empty.java

```
1  public class Empty {}
```

Main.java

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

メソッドもフィールドもいっさい定義していないクラスの toString() を呼び出せるのは、Java には次のような約束が備わっているからです。

暗黙の継承

あるクラスを定義するとき、extends で親クラスを指定しなければ、java.lang.Object を親クラスとして継承したと見なされる。

つまり、先ほどの Empty クラスは、以下のようなクラス定義と実質的に同じものなのです。

Empty.java

```
public class Empty extends Object {}
```

extends を指定しなくても必ず Object を継承するということは、「Java では親なしのクラスを定義できない」ということにほかなりません。これまで紹介してきたさまざまなクラスも、その親クラスを順に辿っていくと、最終的には java.lang.Object クラスに到達します。

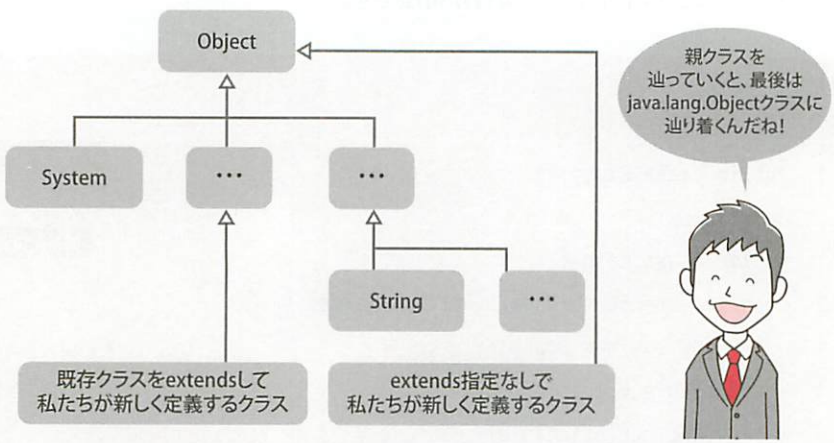


図 14-4　java.lang.Object はすべてのクラスの祖先

API リファレンスを調べると、全クラスの祖先である Object クラスには、次のようなメソッドが定義してあることがわかります。

- equals()　あるインスタンスと自分自身とが同じかを調べる
- toString()　自分自身の内容の文字列表現を返す

Empty クラスで toString() を利用できたのは、「暗黙の親クラスである Object クラスから継承していたから」だったのです。

14.2.2 Object クラスの存在価値

Java では、「すべてのクラスは絶対に java.lang.Object の子孫」なんですね。

でも、そもそも Java を作った人は、なぜ「すべての先祖 Object クラス」なんていうものを作ったんだろう。

いい疑問だね。それでは Object クラスが必要な理由や、その存在価値を紹介しよう。

なぜ Java には、java.lang.Object のような「全クラスの祖先」にあたるクラスが準備してあるのでしょうか。その理由としては次の 2 つが考えられます。

理由 1：多態性を利用できるようになるから

すべてのクラスが Object を先祖に持つのですから、「すべてのクラス is-a Object」ということができます。あらゆるクラスは「ザックリみれば、どれも Object」として同一視できるのです（第 13 章を参照）。

これは、次のリスト 14-6 のように、「Object 型の変数には、どんなインスタンスも代入できる」ことを意味しています。

リスト 14-6

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
```

```
3       Object o1 = new Empty();
4       Object o2 = new Hero( );
5       Object o3 = "こんにちは";
6     }
7   }
```

また次のリスト 14-7 のように、引数として Object 型を用いることで、「何型でもいいからインスタンスを渡せる」メソッドを作ることもできます。

リスト 14-7

Main.java

```
1  public class Main {
2    public void printAnything(Object o) {
3      // 何型でもいいから、引数を1つ受け取り画面に表示
4      System.out.println(o.toString());
5    }
6  }
```

System.out.println() の中身

ちなみに、リスト 14-7 の printAnything() メソッドとほぼ同じ内容を持つのが、私たちがいつも利用している System.out.println() メソッドです。API リファレンスで System.out.println() を調べると、引数として Object 型を受け取ることができるようになっていることがわかります。

System.out.println() は渡されたインスタンスの内容を画面に表示する役割を持っていますが、その実現のために、渡されたオブジェクトの toString() メソッドを呼んで文字列表現を得て、それを画面に出力しています。

このように Object 型の変数は、あらゆる参照型のインスタンスを格納できます。格納できないのは、基本データ型（int や long など）の情報だけです。

理由 2：すべてのクラスが最低限備えるべきメソッドを定義できるから

「少なくとも Java のクラスであれば、最低限備えておいたほうがよい機能」というものがあります。たとえば、インスタンス同士の内容が同じものかをチェックしたり、インスタンスの内容がどのようなものかを文字情報として表示させたりしたいことは頻繁にあります。

Object クラスで equals() や toString() などが定められているおかげで、私たちはクラスの種類を気にすることなく、常に同じ方法で内容を比較したり表示したりできるのです。

14.2.3 デフォルトの文字列表現

さて突然ですが、次のコードを見て表示される結果を想像してみてください。

リスト 14-8

Hero.java

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

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h = new Hero();
4      h.name = "ミナト";
5      h.hp = 100;
6      System.out.println(h.toString());
7    }
8  }
```

引数は単に h でもよい

toString() メソッドは、オブジェクトの中身の情報を文字列にしてくれるのよね。「名前＝ミナト、HP=100」みたいな表示が出るのかな。

ボクは「名前：ミナト／ HP：100」だと思うな。

2 人の予想とは大きく異なり、実行結果は次のようなものになります。

実行結果

```
Hero@3487a5cc
```

※@以降は実行環境ごとに異なります。

あれ？ なんでこんな変な出力になっちゃうんだろう。

試しに Date クラスを new して toString() を呼んでみましたけど、こんなおかしな表示ではなく、ちゃんと日時が表示されました。どうして Hero クラスだけ？

リスト 14-8 でわかるように、Hero クラスには toString() メソッドが宣言されていません。ということは、Main.java の 6 行目で呼び出されて動作しているのは、Object クラスに宣言され、Hero クラスに継承されてきた toString() メソッドです。

実際、Object クラスに定義されている toString() メソッドは、「型名 @ 英数字」という形式で情報を表示するという極めてシンプルな処理内容になっています。

14.2.4 文字列表現を定義する

「println(h.toString())」だけで「名前：ミナト／ HP：100」みたいな表示をしてほしいです。

そのためには「toString() が呼ばれたら、どのフィールドの内容を、どう修飾して文字列表現にするか」を湊くんが指示してあげる必要があるね。

Hero クラスの toString() メソッドが呼ばれた際、湊くんが期待したような文字列を得るには、次のように Hero クラスで toString() メソッドをオーバーライドする必要があります。

リスト 14-9

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4      :
5    public String toString() {
6      return "名前：" + this.name + "／HP：" + this.hp;
7    }
8  }
```

オーバーライドする

Main.java

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

この 1 行で内容を表示

```
8  }
```

Date クラスの toString() を呼び出したときに変な表示にならなかったのは、きっと Date クラスでオーバーライドされていたからなのね。

このように toString() メソッドをオーバーライドしておくことで、インスタンスの内容を画面に出力することが簡単にできるようになります。いくつもの情報を内部に持つようなクラスを開発したら、ぜひ toString() メソッドをオーバーライドすることを検討してください。

14.2.5 等値と等価の違い

Object クラスで定義されているメソッドの中でも、toString() と並んで有名なのが equals() メソッドです。equals() メソッドは、2 つのインスタンスが「同じ内容であるか」を判定するため、次のように用いられます。

リスト 14-10

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 = "ミナト";
8      h2.hp = 100;
9      if (h1.equals(h2) == true) {
10       System.out.println("同じ内容です");
11     } else {
12       System.out.println("違う内容です");
```

1 人目の勇者

2 人目の勇者

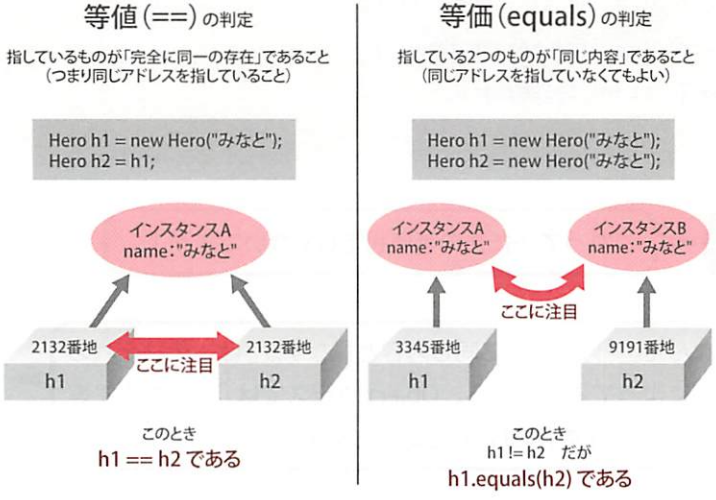
```
13        }
14    }
15  }
```

同じかどうかの判断って「if (h1 == h2)」ではダメなんですか？

そういえば第 3 章で「文字列の比較は equals() を使う」と丸暗記しましたけど、"equals" と "==" って何が違うんでしょう？

湊くんがいうように、同じかどうかの判定には、「if (h1==h2)」という書き方もあります。しかし、**== を使った判定と equals を使った判定では、微妙に意味が異なる**ことに注意が必要です。前者のような比較は**等値**(equality)であるか、後者は**等価**(equivalence)であるかを判定するためのものです。

図 14-5　等値と等価の違い



だから String 型は equals() で判定が必要だったのね。

14.2.6 等価判定方法の指定

equals() は、すべてのクラスで使えるので、等価か調べたいときには、とにかく equals() を呼び出せばいいんですね！

いや、equals() についても湊くんが「何をもって同じものと見なすか」を指定してあげないと正しく動かないんだ。

何の準備もなく自分で作ったクラスの equals() メソッドを呼び出しても、うまく動きません。たとえば先ほどのリスト 14-10 では内容が同じはずの 2 つの Hero インスタンスを比較していますが、実行すると画面には「違う内容です」と表示されてしまいます。

実は Object クラスから継承される equals() メソッドの処理内容は、おおむね以下のようなものになっています。

```
public boolean equals(Object o) {
  if (this == o) { return true; }
  else { return false; }
  }
```

これって…ただの等値判定じゃないですか！

そうなんだ。「何をもって同じ内容と見なすか」は、それぞれのクラスによって異なるから、Object クラスでは「とりあえず等値なら true を返す」作りにしてあるんだよ。

そもそも等価の判定は、機械的には行うことができないものです。なぜなら**「何をもって、意味的に同じものと見なすか」は、クラスによって異なり、一律には決められない**からです。たとえば、名前が「ミナト」の勇者と「ミナト」の勇者を同じものと見なすかどうかは、作るゲームによって異なるでしょう。

そこでクラスの開発者は、そのクラスのインスタンスについて、「何をもって、意味的に同じと見なすか」を equals() メソッドのオーバーライドという形で指定しなければなりません。

仮に、Hero クラスは「名前が同じであれば同じ内容のインスタンスと見なす」と定義するならば、次のようなコードになるでしょう。

リスト 14-11

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4     :
5    public boolean equals(Object o) {
6      if (this == o) { return true; }
7      if (o instanceof Hero) {
8        Hero h = (Hero) o;
9        if (this.name.equals(h.name)) {
10         return true;
11       }
12     }
13     return false;
14   }
15 }
```

等値なら間違いなく等価

名前が等しければ等価

Hero クラスに上記のような修正を行った上でリスト 14-10 の Main クラスを実行すると、「同じ内容です」という表示結果を得られます。

toString() と equals() のオーバーライド

新しくクラスを開発したら、toString() と equals() をオーバーライドする必要性がないかを検討する。

14.3 基本データ型をオブジェクトとして扱う

14.3.1 ラッパークラスとは

先輩。API リファレンスを見ていたら、java.lang パッケージの中に、Long とか Boolean とか、おもしろい名前のクラスを見つけました。

それらは「ラッパークラス」と呼ばれるクラスだよ。

これまでに学んだように、Java で利用できる型は、大きく「基本データ型」と「参照型」の 2 種類に分けられます。特に int 型や boolean 型といった基本データ型については、この本の冒頭から数多く使ってきました。

ところで、Java の API では、それぞれの基本データ型に対応したクラスを準備しており、それらはラッパークラス（wrapper class）と総称されています。

表 14-2　基本データ型とラッパークラスの対応

基本データ型	ラッパークラス
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

これらラッパークラスには 2 つの役割があります。

第 1 の役割：便利なメソッドの提供

最初の役割は、「ある基本データ型に関する、便利なメソッドを提供すること」です。 たとえば、Integer クラスは以下のような int にまつわる多くの便利なメソッドを持っています。

表 14-3　Integer クラスが持つ便利なメソッド（一部）

メソッド名	ふるまい
parseInt()	数字の文字列を int 型に変換する
toHexString()	整数を 16 進数表現に変換する

これらの便利メソッドは、いずれも静的メソッド（static）で宣言されているため、ラッパークラスをインスタンス化することなく、手軽に利用できます。

第 2 の役割：インスタンスとして扱えるようにする

第 2 の役割は、基本データ型の情報をインスタンスとして扱えるようにすることです。次のコードを見てください。

```
int i1 = 15;
Integer i2 = Integer.valueOf(i1);
System.out.println(i2);
```

int → Integer 変換

このコードの 2 行目では、Integer クラスの valueOf メソッドを呼び出し、**基本データ型 int の変数 i1 の内容を、ラッパークラス型 Integer の変数 i2 に変換**しています。

この変数 i2 に格納されている Integer インスタンスは、内部で「15」という値を保持するだけの極めてシンプルなインスタンスです。HP や名前など、多数の情報を保持していた Hero インスタンスなどとは大違いですね。

図 14-6　基本データ型とラッパークラス

図 14-6 のどちらの変数も、結局「15」という情報を持っています。箱の中に入っているものが数値そのものか、インスタンスに包まれた数値なのかの違いでしかありません。

しかも、基本データ型のほうが、よりメモリの消費量が少なく、高速に読み書きできることを考えると、Integer 型のようなラッパークラスを使う必要性を感じられないかもしれません。

しかし、Java には**「基本データ型を利用できない API」がいくつか存在します。**そのような API のメソッドを私たちが利用する場合、「15」という数字情報を渡したいときには、int 型の変数ではなく Integer 型を利用する必要があるのです。

14.3.2 基本データ型とラッパークラスの相互変換

第 2 章で学んだように、基本的に異なる型の変数に値は代入できません。よって「15」という数字を int 型の変数に代入することはできても、Integer 型の変数にそのまま代入することはできません。

つまり「int 型の変数に入っている値を Integer 型の変数に入れたい場合」や、逆に「Integer 型の変数に入っている値を int 型の変数に入れたい場合」は、相互に型を変換する必要があります。

ラッパークラスへの変換には valueOf を、基本データ型への変換には intValue() や longValue() などの「～ Value()」メソッドを利用します。

リスト 14-12

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int i1 = 15;
4      Integer i2 = Integer.valueOf(i1);
5      int i3 = i2.intValue();
6    }
7  }
```

int → Integer 変換

Integer → int 変換

「valueOf() や intValue() を呼ぶのは理解できたけど、ちょっとめんどうですよぉ〜」とか、湊は思っているんじゃない？

そ、そんなことは断じて…ある…かも。

湊くんの感覚は正しいよ。だから Java には相互変換を自動的に行うしくみが備わっているんだ。

古いバージョンの Java では、「基本データ型とラッパークラスとの間の相互型変換」を valueOf() と〜value() メソッドを用いて明示的に行う必要がありました。しかし Java5 以降では、暗黙的に相互変換を行う AutoBoxing と AutoUnboxing というしくみが導入されました。

AutoBoxing と AutoUnboxing

ラッパークラス型と基本データ型との間で代入を行う式を記述すると、自動的に valueOf() や〜 value() による型変換が行われる。

このしくみのおかげで、次のリスト 14-13 を記述しても文法エラーが発生することなくコンパイルできます。ぜひ、リスト 14-12 と見比べてください。

リスト 14-13

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int i1 = 15;
4      Integer i2 = i1;
5      int i3 = i2;
6    }
7  }
```

int → Integer 自動変換

Integer → int 自動変換

どうせ似たものだからと「空気を読んで」変換してくれるのね。

これでラッパークラスも使えるようになったから、いつ「基本データ型では使えない一部の API」を使う日がきても大丈夫だ！

そうだね。きっとその日も遠いことではないよ。

14.4 第 14 章のまとめ

この章では、次のようなことを学びました。

日付の扱い

- Java における日付情報は基本的に java.util.Date 型で扱う。
- その他、必要に応じて long 値、6 つの int、String 型に変換して用いる。
- 「年月日時分秒」の 6 つの int 値から Date インスタンスを得るためには Calendar クラスを使う。
- Date インスタンスの内容を任意の書式で文字列に整形したい場合は、SimpleDateFormat クラスを使う。

Object クラス

- Java において、すべてのクラスは Object クラスの子孫である。
- すべてのインスタンスは Object 型変数に格納可能である。
- すべてのクラスは Object から toString() や equals() を継承している。
- 自分で作成したクラスにおいては、文字列表現や等価判定方法を指定するため、toString() や equals() をオーバーライドする。

ラッパークラス

- 基本データ型に対応したラッパークラスが java.lang パッケージに存在する。
- 基本データ型とラッパークラスのデータは、valueOf() や ～ value() メソッドで明示的に変換できる。
- 両者は AutoBoxing ／ AutoUnboxing 機構により暗黙的にも変換される。

14.5 練習問題

練習 14-1

main() メソッドのみを持つクラス Main を定義し、以下の手順を参考にして「現在の 100 日後の日付」を「西暦 2011 年 09 月 24 日」という形式で表示するプログラムを作成してください。

① 現在の日時を Date 型で取得します。
② 取得した日時情報を Calendar にセットします。
③ Calendar から「日」の数値を取得します。
④ 取得した値に 100 を足した値を Calendar の「日」にセットします。
⑤ Calendar の日付情報を Date 型に変換します。
⑥ SimpleDateFormat を用いて、Date インスタンスの内容を表示します。

練習 14-2

口座番号を表す String 型フィールド accountNumber と、残高を表す int 型フィールド balance を持つ銀行口座クラスを作ってください。さらに、このクラスにメソッド宣言を追加し、次の①と②の条件を満たすように修正してください。

①口座番号 4649、残高 1592 円の Account インスタンスを変数 a に生成し、「System.out.println (a);」を実行すると、画面に「¥1592（口座番号＝ 4649）」と表示されること。
② 口座番号が等しければ等価と判断されること。ただし、「 4649」など、口座番号の先頭に半角スペースが付けられたものは、それを無視して比較すること（「 4649」口座と「4649」口座は同じものと捉える）。
（ヒント：java.lang.String クラスの trim () メソッドを利用します。）

14.6 練習問題の解答

練習 14-1 の解答

Main.java

```
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
public class Main {
  public static void main(String[] args) {
    // ①現在の日時をDate型で取得
    Date now = new Date();
    Calendar c = Calendar.getInstance();
    // ②取得した日時情報をCalendarにセット
    c.setTime(now);
    // ③Calendarから「日」の情報を取得
    int day = c.get(Calendar.DAY_OF_MONTH);
    // ④取得した値に100を足してCalendarの「日」にセット
    day += 100;
    c.set(Calendar.DAY_OF_MONTH, day);
    // ⑤Calendarの日付情報をDate型に変換
    Date future = c.getTime();
    // ⑥指定された形式で表示
    SimpleDateFormat f =
        new SimpleDateFormat("西暦yyyy年MM月dd日");
    System.out.println(f.format(future));
  }
}
```

練習 14-2 の解答

Account.java

```
1  public class Account {
2    String accountNumber;     // 口座番号
3    int balance;              // 残額
4    /* ①文字列表現のメソッド */
5    public String toString() {
6      return "¥¥" + this.balance +
           " (口座番号：" + this.accountNumber + ") ";
7    }
8    /* ②等価判定のメソッド */
9    public boolean equals(Object o) {
10     if (this == o) {
11       return true;
12     }
13     if (o instanceof Account) {
14       Account a = (Account) o;
15       String an1 = this.accountNumber.trim();
16       String an2 = a.accountNumber.trim();
17       if (an1.equals(an2)) {
18         return true;
19       }
20     }
21     return false;
22   }
23 }
```