第 6 章

複数クラスを用いた開発

第 5 章で学習したメソッドを上手に使えば、
ある程度大きなプログラムも 1 人で作ることができます。
しかし大規模なソフトウェアの開発になると、
自分以外の開発者と手分けしてプログラミングする必要があります。
この章では、複数の開発者が分担して部品を作り、それを組み合わせる
Java のしくみを紹介します。

CONTENTS

6.1 ソースファイルを分割する
6.2 複数クラスで構成されるプログラム
6.3 パッケージを利用する
6.4 名前空間
6.5 Java API について学ぶ
6.6 クラスが読み込まれるしくみ
6.7 パッケージに属したクラスの実行方法
6.8 第 6 章のまとめ
6.9 練習問題
6.10 練習問題の解答

6.1 ソースファイルを分割する

6.1.1 1 つのソースファイルによる開発の限界

菅原さん！　はりきって開発をしていたら、クラスの中に 35 個もメソッドができてしまって、ワケがわかんなくなっちゃいました！

そんなときにはクラスを分割すればスッキリするよ。

第 5 章では、長く複雑になってしまった main メソッドを複数のメソッドに分割する方法を学びました。しかし、1 つのソースファイルの中に含まれるメソッドの数が増えると、やはりソースコードの全体を把握することが難しくなり、開発しにくくなっていきます。

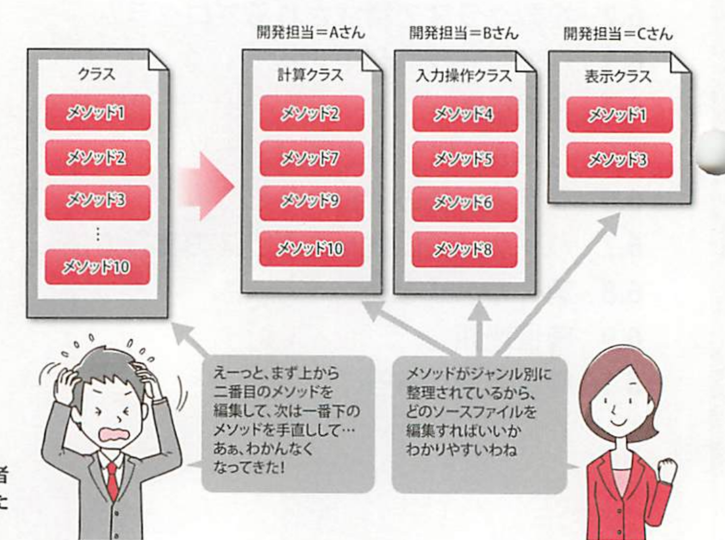


図 6-1　複数の開発者が分担して開発するための便利なしくみ

そこでJavaでは、1つのソースファイルにすべてのメソッドを書くのではなく、複数のソースファイルに分割して記述できるようなしくみが準備されています。

1つのソースファイルにはクラスブロックを最低1つは作成しなければならないルールがありますので、**複数のソースファイルに分けて開発するということは、複数のクラスに分けて開発すること**だと捉えることもできます。

たくさんのメソッドを複数のクラスに分けて記述することには、単に「整理されてわかりやすくなる」だけではなく、「**ファイルごとに開発を分担し、それぞれが並行して開発を進められる（＝分業しやすい）**」というメリットもあります。このように、1つのプログラムを複数の部品に分けることを**部品化**といいます。

6.1.2 計算機プログラムを分割しよう

では、リスト6-1の計算機プログラム（Calc）を、2つのクラスに分割してみましょう。現状の計算機プログラムはmain()、tasu()、hiku()の3つのメソッドから構成されています。

リスト6-1 計算機プログラム

Calc.java

```
1  public class Calc {
2    public static void main(String[] args) {
3      int a = 10; int b = 2;
4      int total = tasu(a, b);
5      int delta = hiku(a, b);
6      System.out.println("足すと" + total + "、引くと" + delta);
7    }
8    public static int tasu(int a, int b) {
9      return (a + b);
10   }
11   public static int hiku(int a, int b) {
12     return (a - b);
13   }
14 }
```

さぁ、この3つのメソッドをのうち、どれを別クラスに切り出そうか。

tasu()とhiku()の2つは「数学的な計算処理をするメソッド」であり、main()は「tasu()やhiku()を呼び出して画面に表示する役割を持つ、全体の流れを司るメソッド」です。よってmain()とそれ以外のメソッドを2つのクラスに分けて整理しましょう。

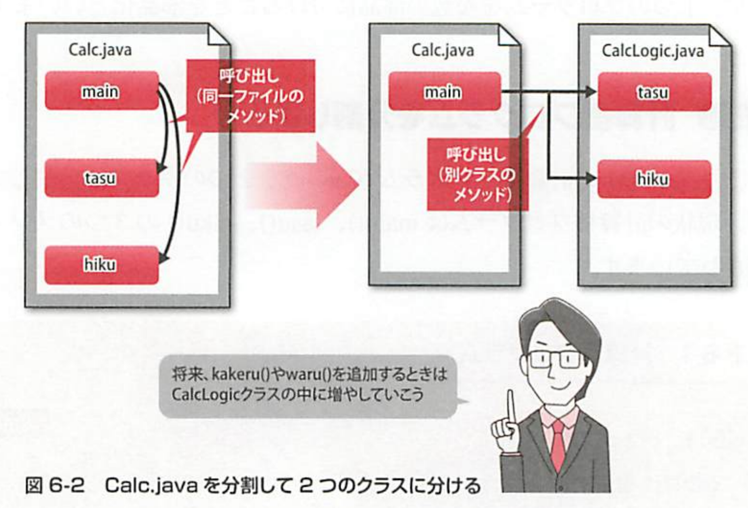


図6-2　Calc.javaを分割して2つのクラスに分ける

Step1　計算処理メソッドを記述するためのソースファイルを作る

まず、tasu()やhiku()といった計算ロジックのメソッドを入れるソースファイルを作ります。新たなファイル名はCalcLogic.javaにします。1.2.1項で「ソースファイル名とクラス名は同じでなければならない」と学びましたね。そのため、CalcLogic.javaの書き始めは「public class CalcLogic」とします。

Step2　tasu()とhiku()を移動する

現在Calc.javaの中にあるtasu()とhiku()を、新たに作ったCalcLogic.javaへ移動します。すると、CalcLogic.javaは次のリスト6-2のようになります。

リスト 6-2　CalcLogic.java に計算処理を追加する

CalcLogic.java

```
1 public class CalcLogic {
2   public static int tasu(int a, int b) {
3     return (a + b);
4   }
5   public static int hiku(int a, int b) {
6     return (a - b);
7   }
8 }
```


Step3　メインメソッド内の呼び出しを修正する

一方で、Calc.java には次のように main() だけが残されているはずです。

リスト 6-3　Calc.java

Calc.java

```
1 public class Calc {
2   public static void main(String[] args) {
3     int a = 10; int b = 2;
4     int total = tasu(a, b);
5     int delta = hiku(a, b);
6     System.out.println("足すと" + total + "、引くと" + delta);
7   }
8 }
```

4 行目と 5 行目で tasu() や hiku() メソッドを呼んでいますが、このままでは「tasu() や hiku() メソッドがないから呼び出せない！」という意味のコンパイルエラーが出てしまいます。**この Calc.java には tasu() や hiku() は存在しない**ので当然です。

今まで、main()の中で単に「tasu(a,b)」と記述すればtasu()を呼び出すことができたのは、**main()とtasu()やhiku()が同じCalcクラスに属していたから**です。しかし今回のソースファイルの分割によって、tasu()やhiku()はCalcLogicクラスに属するようになったため、main()から呼び出す際には「**CalcLogicのtasu()**」や「**CalcLogicのhiku()**」のように、明示的に所属を示す必要があります。これには、main()から以下のように呼び出すことで対応できます。

```
int total = CalcLogic.tasu(a, b);
int delta = CalcLogic.hiku(a, b);
```

ドット(.)は次の章以降でも頻繁に登場するけど、日本語で言う「〜の」という意味だよ。

同じ部署の私たちは先輩を普段「菅原さん」と呼べるけど、別部署の人は「開発部の菅原さん」と言うのと似ていますね。

ここまでで無事、計算機プログラムは2つのクラス(リスト6-2、リスト6-4)に分割することができました。

リスト6-4　Calc.java

Calc.java

```
1  public class Calc {
2    public static void main(String[] args) {
3      int a = 10; int b = 2;
4      int total = CalcLogic.tasu(a, b);
5      int delta = CalcLogic.hiku(a, b);
6      System.out.println("足すと" + total + "、引くと" + delta);
7    }
8  }
```

6.2 複数クラスで構成されるプログラム

6.2.1 複数クラスのコンパイル

前節では計算機プログラムを2つのソースファイルに分割しました。そのため、このプログラムを実行するには、Calc.java と CalcLogic.java のそれぞれをコンパイルする必要があります。javac コマンドでは、以下のように複数のソースファイルを指定することができます。


```
>javac Calc.java CalcLogic.java⏎
```

無事コンパイルが終了すると、それぞれのソースファイルに対応したクラスファイルが生成されます。

```
>dir⏎
2011/06/23 15:52 735 Calc.class
2011/06/23 15:51 234 Calc.java
2011/06/23 15:52 298 CalcLogic.class
2011/06/23 15:51 156 CalcLogic.java
4 個のファイル 1,423 バイト
```

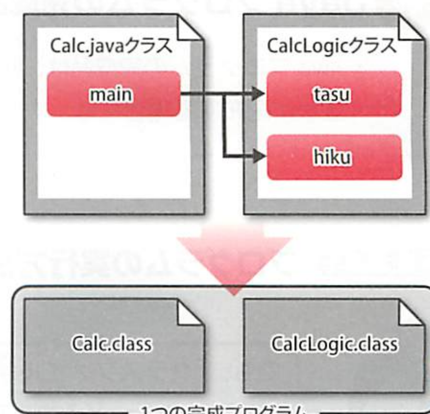


図 6-3　Calc.java の分割により、Calc.java と CalcLogic.java の 2 つで 1 つの完成プログラムを構成

6.2.2 Javaプログラムの完成品

この2つのクラスファイルが計算機プログラムの最終完成品だ。誰かに渡す際には、この2つのクラスファイルが必要なんだよ。

えっ？　2つのファイルを渡すんですか？ なんだか完成品じゃないような気がするんですけど…。

普段用いるパソコンのアプリケーションに慣れ親しんでいると、「Javaプログラムの完成品」のちょっと変わった姿を意外に思うかもしれません。普通のプログラムは、たいていファイルは1つだけだからです。たとえば、Windows7におけるメモ帳プログラムは、C:¥Windows¥System32¥notepad.exeのような単独のファイルであって、これをダブルクリックすれば起動します。

しかし、Javaで開発されたプログラムは「複数のクラスファイルの集まり」であることが多く、ダブルクリックで起動させるのではなく、javaコマンドで起動します。ですからJavaプログラムを誰かに渡す、あるいは納品する場合には、**複数のクラスファイルが入っているフォルダをまるごと「1つの完成品」として渡す**ことがほとんどです。

Javaプログラムの完成品

- Javaプログラムの完成品は、複数のクラスファイルの集合体。
- 誰かに配布する場合には、すべてのクラスファイルを渡す必要がある。

6.2.3 プログラムの実行方法

でも、「クラスファイルがたくさん入ったフォルダをまるごと」受け取ったら、どうやって起動すればいいのかしら？

JARファイル(コラム参照)でまとめた形ではなく、クラスファイルが入ったフォルダをまるごと受け取った場合は、クラス名を指定して実行する必要があります。

```
>java クラス名⏎
```

JVMは起動時に指定されたクラスの中にあるmainメソッドを呼び出してプログラムの実行を開始します。よって、Javaのプログラムを実行する人は「渡された複数のクラスファイルのうち、mainメソッドが含まれているクラスの名前」を指定する必要があります。たとえば、計算機プログラムの場合は「java Calc」と起動すべきであって、「java CalcLogic」では正常に動作しません。

今回の計算機プログラムの場合、私たちは「Calcの中にmain()が入っていて、CalcLogicの中にはない」という事実を知っているので「java Calc」で起動できると判断できました。

しかし、他人が作ったJavaプログラムの場合は、すべてのクラスファイルを受け取っても、「どのクラスの中にmainメソッドがあるか」がわからないと起動できないことに注意しましょう。

複数の完成クラスファイルを渡す場合の注意点

Javaのプログラムを配布・納品するときは、単にすべてのクラスファイルを渡すだけではなく、「どのクラスにmain()が入っているか」も伝える必要がある。

JARファイルとは?

プログラムの完成品が複数のクラスファイルになった場合、メールで送る際などに不便です。そこでJavaでは、「複数のクラスファイルを1つにまとめるファイル形式」としてJAR(Java ARchive)が定められています。JARファイルはZIPファイルととてもよく似たアーカイブファイルであり、JDKに付属するjarコマンドでも作成することができます。

パッケージを利用する

6.3.1 クラスが増えすぎたら…どうしよう？

菅原さ〜ん！

今度はクラスの数が増えすぎて、わかりにくくなっちゃった…だね？

Javaを学習し始めて日も浅い段階では想像がつかないかもしれませんが、大規模なプロジェクトになると、数百個ものクラスを使って1つのプログラムを開発することもあります。しかし、クラスの数も20個を超える規模になると、さすがに管理が大変になってきます。

そこでJavaには、各クラスをパッケージ（package）というグループに所属させて、分類・管理できるようなしくみが準備されています。

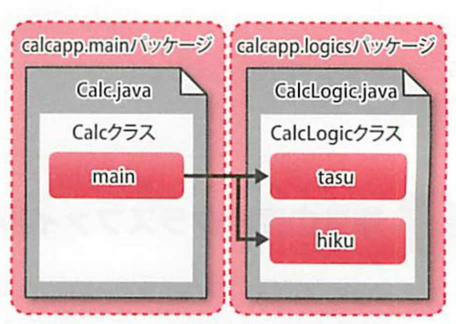


図6-4　2つのパッケージに分割された計算機プログラム。calcapp.mainパッケージとcalcapp.logicsパッケージで構成

main()の行数が増えたら複数メソッドに分割 → メソッド数が増えたら複数クラスに分割 → クラス数が増えたら複数パッケージに分割というわけね。

Javaには部品化のしくみがいくつも準備されているんだね。

それでは、前節でも登場した計算機プログラムを題材にして、パッケージを利用してみましょう。クラスをパッケージに所属させるためには、そのクラスの**ソースコードの先頭に package 文を記述**します。

クラスをパッケージに所属させる

package 所属させたいパッケージ名;

※ package 文はソースコードの先頭に記載する必要がある。

たとえば計算機プログラムの2つのクラスを、図6-4のようにそれぞれのパッケージに所属させるには、次のように記述します。

リスト 6-5　Calc クラスを calcapp.main パッケージに所属させる

Calc.java

```
package calcapp.main;
public class Calc {
  ⋮
```

※ このコードは、後の6.7節で紹介する方法を用いて実行する必要があります。現状ではコンパイルまでできれば構いません。

リスト 6-6　CalcLogic クラスを calcapp.logics パッケージに所属させる

CalcLogic.java

```
package calcapp.logics;
public class CalcLogic {
  ⋮
```

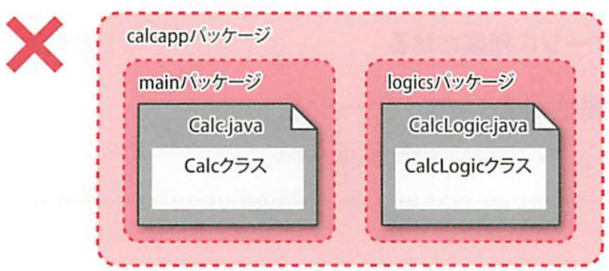
※ このコードは、後の6.7節で紹介する方法を用いて実行する必要があります。現状ではコンパイルまでできれば構いません。

パッケージの名前は、Javaの識別子（1.3.2項）のルールに従っていれば自由に定めることができますが、アルファベットは小文字を使用するのが一般的です。また、「calcapp.main」や「calcapp.logics」のように、ドットで区切ったパッケージ名も多く用いられます。

なお、「calcapp.main」と「calcapp.logics」という2つのパッケージ名を見て、「共通のcalcappパッケージに所属するmainとlogicsという子パッケージで、同じ

グループである」という感覚を抱いてしまうかもしれませんが、両者は相互にまったく関係がない、独立した2つのパッケージです。パッケージの中にパッケージを入れることはできませんし、パッケージに親子関係や階層関係はありません。

正しくないパッケージのイメージ



正しいパッケージのイメージ

図 6-5　パッケージ名の一部が同じであっても、それぞれのパッケージに関連性はない

デフォルトパッケージ

前節まで作成してきたクラスには package 文がなく、どのパッケージにも所属していませんでした。どのパッケージにも所属していないことを「無名パッケージに属している」または「デフォルトパッケージに属している」と表現することもあります。

なお、デフォルトパッケージに属するクラスは後述の import 文でインポートすることはできません。

6.3.2 パッケージを含むクラス名を指定する

ここまでで無事2つのクラスを別パッケージに所属させることができました。しかし、このままコンパイルするとCalc.javaの2つの行に構文エラーが発生してしまいます。

```
int total = CalcLogic.tasu(a, b);
int delta = CalcLogic.hiku(a, b);
```

Calcクラスの中にあるこの2行では、それぞれ「CalcLogic」クラスを利用しようとしています。しかし、この書き方では**「どのパッケージのCalcLogicクラスか」を明示していない**ため、Calcクラスは自分と同じパッケージ（calcapp.mainパッケージ）に所属するCalcLogicクラスを呼び出そうとして失敗してしまうのです。別パッケージに所属しているクラスを利用する場合、次のように**所属パッケージ名を添えたクラス名**を指定する必要があります。

リスト 6-7　別のパッケージにあるクラスを呼び出す

Calc.java

```
1  package calcapp.main;
2  public class Calc {
3    public static void main(String[] args) {
4      int a = 10; int b = 2;
5      int total = calcapp.logics.CalcLogic.tasu(a, b);
6      int delta = calcapp.logics.CalcLogic.hiku(a, b);
7      System.out.println("足すと" + total + "、引くと" + delta);
8    }
9  }
```

※このコードは、後の6.7節で紹介する方法を用いて実行する必要があります。現状ではコンパイルまでできれば構いません。

クラス名の前に所属パッケージ名を付けてあげればいいんですね。

厳密に「calcapp.logicsパッケージ」の「CalcLogicクラス」の「tasu()」と指定するんだよ。

このように、あるクラスから別パッケージのクラスを利用する場合、「パッケージ名を頭に付けた完全なクラス名」を使う必要があります。この完全なクラス名のことを、**完全限定クラス名**（full qualified class name）、または略して**FQCN**といいます。

完全限定クラス名（FQCN）

パッケージ名．クラス名

あまり一般的ではないけど、「同じパッケージに所属する別のクラス」を利用するときに、わざわざFQCNを使っても文法違反にはならないんだ。

同じ部署にいる私が菅原さんのことを「ミヤビリンクの、開発部の、菅原さん」と呼んでも、一応間違いではないのと同じですね。

そうだね。でも実際に社内でそんな呼ばれ方をしたら、「熱でもあるんじゃないのか？」と心配するよ（笑）。

6.4 名前空間

6.4.1 パッケージを使うもう 1 つのメリット

パッケージには「クラスをグループ化して分類・整理することで、プログラムをわかりやすくする」という目的のほかに、もう 1 つ重要な役割があります。 それは、**自分が作るクラスに対して、開発者が自由な名前を付けられるようにすること**です。

え？ 今までも自由にクラス名を付けてきたし、不自由は感じませんでしたけど…。

そうだね。では、200 個ぐらいのクラスを 20 人で分担して開発する場合を考えてみようか。

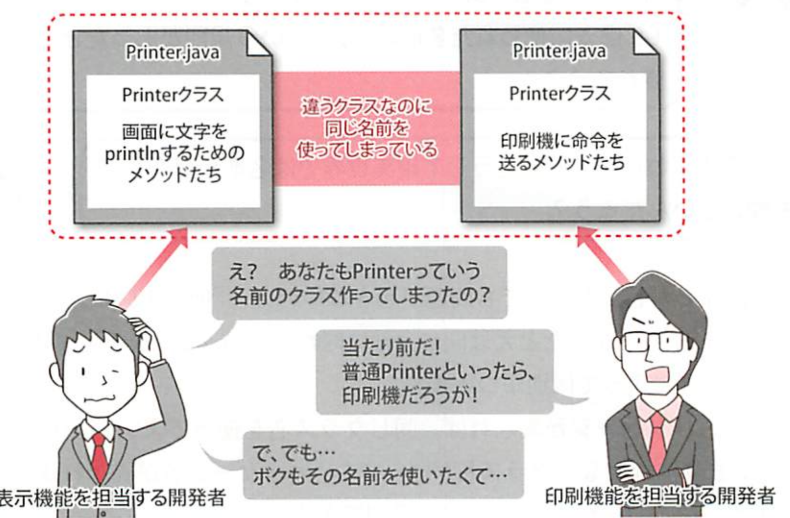


図 6-6　別のクラスに同じ名前を付けてはダメ

大規模な開発になると、複数の開発者が分担して各自が受け持ったクラスを開発します。すると、それぞれの開発者が偶然「同じクラス名を使ってしまう」可能性が出てきます（図 6-6）。

このように、内容が異なる別々のクラスで同じ名前を取り合ってしまうことを**名前の衝突**といいます。

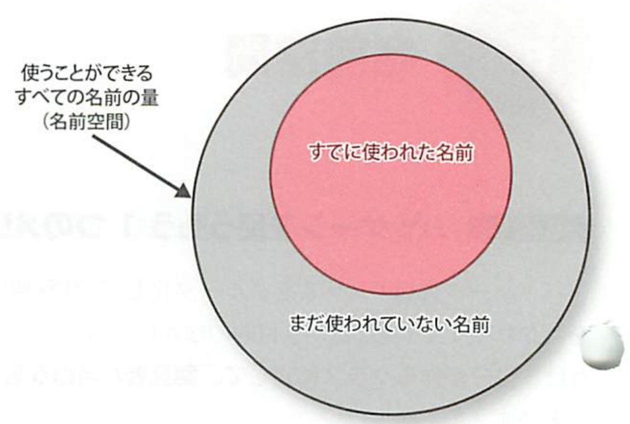


図 6-7　新しいクラスを作るごとに、使えるクラス名が減っていく

異なるクラスで同じクラス名を使うと区別が付かなくなってしまうため、Java ではクラス名の衝突は原則として許されません。使うことができる名前の総量（＝**名前空間**）は限られていて、**新しくクラスを作ると、そのクラス名は使えなくなり、使えるクラス名は減っていく**のです（図 6-7）。

現実世界でも、新しく子どもが産まれて名前を付けるとき、もし「過去に使われた名前はダメ」という規則があったとしたら大変だろう？

そうですよね…。でも、どうして現実世界では名前が衝突しても問題ないんだろう？

現実世界で人名が重複しても問題が起きないのは、「他の手段によって正しく区別できる」からです。たとえば同じ会社に同姓同名の人がいたとしても、「部署」や「役職」などによって区別がつきます。

Java でも**パッケージが異なれば、同じクラス名を使ってよい**というルールになっています。なぜならクラス名が同一でも、パッケージ名が異なれば**完全限定クラス名（FQCN）が異なるので両者を区別できる**からです。

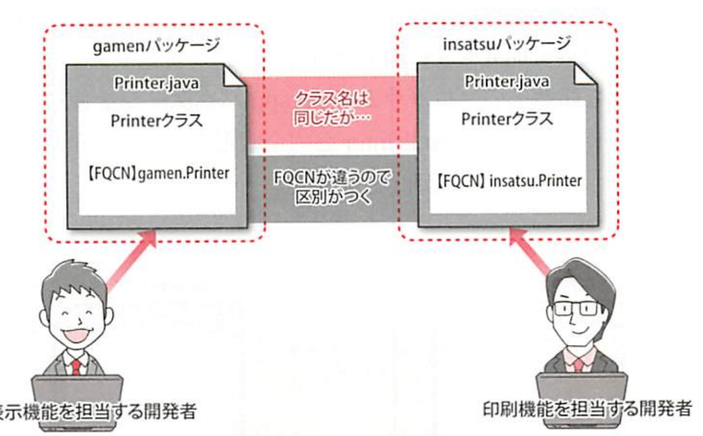


図 6-8　パッケージ名が異なれば完全限定クラス名（FQCN）が異なるので同じクラス名でも区別がつく

つまり**パッケージを使うことによって、それぞれのパッケージ内では自由にクラス名を付けることが可能になる**わけです。

6.4.2 パッケージ名自体の衝突を避ける方法

でも、パッケージ名が衝突しちゃうと困るんじゃないですか？

そうなんだ。だからパッケージ名の付け方については「あるルール」が推奨されているんだよ。

パッケージ名さえ異なればクラス名は重複してもよく、自由にクラス名を付けても構わないことがわかりました。しかし、**パッケージ名が衝突すると、これらの前提はすべて崩れてしまいます。**

自社の開発プロジェクトであれば、誰がどのようなパッケージ名を使うかを事前に決めておくことで衝突を回避できます。しかし、他社のパッケージを利用する際にはどうでしょう？ パッケージ名が衝突しないように事前に摺り合わせするのは困難です。

たとえば、A 社が myapp パッケージを使ってプログラムを開発しているとしましょう。画面表示を担当する Printer クラスは A 社内で開発しましたが、印刷

機能はイギリスの C 社が無料でインターネット上に公開している Printer クラスを利用すれば A 社内で開発せずに済みそうです。

しかし、C 社も偶然そのプログラムで myapp パッケージを使っており、このままでは A 社が作成した myapp.Printer と完全限定クラス名が重なってしまいます。これでは 2 つのクラスを区別できません。

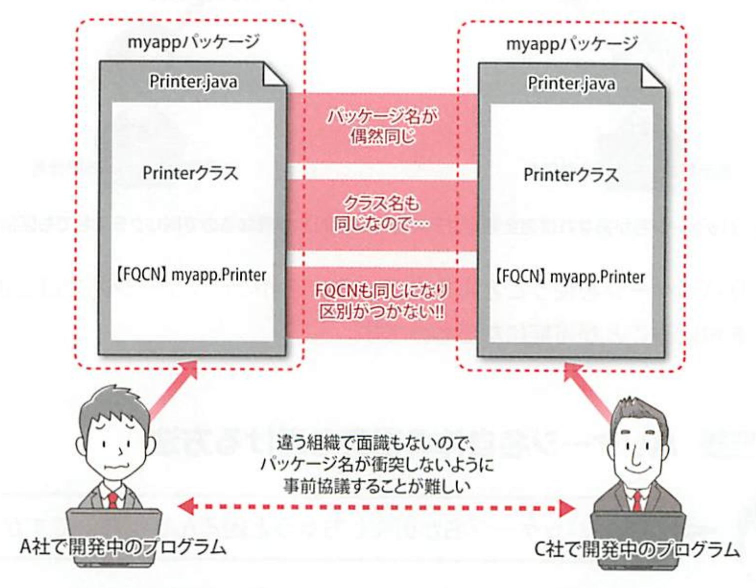


図 6-9　まったく面識がない開発者とはパッケージ名とクラス名の調整ができない!

そこで Java では、次のようなパッケージ名を用いることを推奨しています。

推奨されるパッケージ名

パッケージ名は、「保有するインターネットドメインを前後逆順にしたもの」から始める。

たとえば、foo.example.com というインターネットドメインを取得している企業であれば、com.example.foo で始まるパッケージ名を使うということです。イ

ンターネットドメインは世界に 1 つだけですから、これでパッケージ名が衝突することはありません。com.example.foo より後は、企業や組織内部でパッケージ名が衝突しないよう調整を行えばよいのです。

えっと…会社のホームページが http://miyabilink.jp/ だから…。

jp.miyabilink. から始まるパッケージ名を使えばいいんだよ。

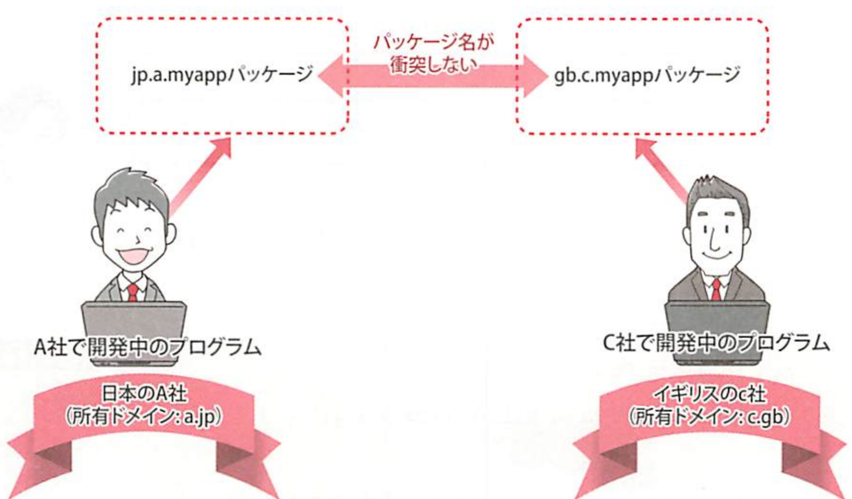


図 6-10　世界に唯一のインターネットドメインをパッケージ名に使うことで、パッケージ名の衝突を心配する必要がなくなる

このパッケージ名のルールのおかげで、自社だけではなく世界中のいろんな人や企業が作ったクラスを自由に組み合わせて利用できるようになるんですね。

そうだよ。Java のプログラムは「世界中の、さまざまな人が作ったクラス」を組み合わせることで効率よく作っていけるんだ。

テストプログラム等では簡単なパッケージ名を付けていても構いませんが、正式なプログラムのクラスには今回紹介した命名規則に従ったパッケージ名を付けましょう。あなたが作って公開するクラスも、世界中の誰かが利用する日が来るかもしれません（本書のサンプルコードでは、解説を簡単にするために、以後も簡単なパッケージ名を使います）。

6.4.3 完全限定クラス名の入力を省略する

名前空間のメリットはわかりましたけど、やっぱりパッケージ名の入力がめんどうですよ…。

めんどうくさがりな君にピッタリの構文があるよ（笑）。

再度、リスト6-7（p.233）のCalc.javaを見て、FQCNを利用している部分を確認してください。

Calc.java

```
1  public class Calc {
2    public static void main(String[] args) {
3      ⋮
4      int total = calcapp.logics.CalcLogic.tasu(a, b);
5      int delta = calcapp.logics.CalcLogic.hiku(a, b);
6      ⋮
7    }
8  }
```

FQCNの利用

「calcapp.logics.CalcLogic」という長い完全限定クラス名（FQCN）を2か所に記述しています。現時点では2か所で済んでいますが、将来プログラムが大きくなったら、この長いFQCNを何度もプログラムコードの随所に入力する必要が出てくるかもしれません。このような場合は、**import文**を使うことによって、FQCN入力のめんどうさを軽減できます。

FQCN 入力の手間を省くための宣言

import パッケージ名 . クラス名 ;

※ import 文はソースコードの先頭に、ただし package 文より後に記述する 。

では、Calc.java の package 文の下に import 文を記述してみましょう。

リスト 6-8　Calc.java に import 文を追加する

Calc.java

```
1  package calcapp.main;
2  import calcapp.logics.CalcLogic;
3  public class Calc {
4    public static void main(String[] args) {
5      :
6      int total = CalcLogic.tasu(a, b);
7      int delta = calcapp.logics.CalcLogic.hiku(a, b);
8      :
9    }
10 }
```

FQCN でなくてもエラーにならない

FQCN を指定してもよい

※ このコードは、後の 6.7 節で紹介する方法を用いて実行する必要があります。現状ではコンパイルまでできれば構いません。

2 行目の import 文に注目してください。この文は、「このソースコードで**単にCalcLogic という表記があったら、それは calcapp.logics.CalcLogic のことだ**と解釈しなさい」という指示です。 頻繁に利用するクラスは import 文を使ってインポートしておくことによって、完全限定クラス名を毎回指定する必要がなくなります。

もし、calcapp.logics パッケージに所属するすべてのクラスをインポートしたい場合には、次のような記述も可能です。

リスト 6-9　calcapp.logics パッケージに属するすべてのクラスをインポート

Calc.java

```
1  package calcapp.main;
```

```
2  import calcapp.logics.*;
3  public class Calc {
4    ⋮
5  }
```

ただし、「import calcapp.*;」という記述ではcalcapp.mainとcalcapp.logicsに所属するすべてのクラスを一度にインポートできないことに注意してください。なぜなら6.3.1項の図6-5にあるように、「calcapp.main」と「calcapp.logics」そして「calcapp」はまったく異なるパッケージであり、親子の関係にないためです。

この指定では、calcappパッケージに所属する全クラスのみがインポートされるのであり、calcapp.mainとcalcapp.logicsに所属するすべてのクラスをインポートしたい場合には以下のように記述する必要があります。

```
import calcapp.main.*;
import calcapp.logics.*;
```

import宣言はあくまでも「めんどうさ軽減機能」

Java以外のプログラミング言語の中には、「include命令」や「require命令」といったものでファイル名を指定することで、他のファイルに記述された機能が利用可能になる言語があります。ときどきJavaのimport文も、include命令やrequire命令のようなものだと勘違いされてしまうことがありますので注意してください。

Javaでは**いっさいの宣言をすることなく、JVMが扱えるすべてのクラスを常時使うことができます**。ただし、その利用に際しては必ずFQCNを利用しなければならず、**import文はあくまでFQCNの記述を省略してめんどうを軽減するため（開発者がラクをするため）の構文にすぎません**。importしたからといって利用できるクラスやメソッドが増えたり、プログラムから利用できる機能が増えたりするようなことはないのです。

6.5 Java APIについて学ぶ

6.5.1 世界中の人々の協力で完成していたHelloWorld

「パッケージの命名規則を守れば、世界中のいろんな人が作ったクラスと自分のクラスを一緒に動かせる（6.4.2 項）」ってことでしたけど…ちょっと想像できません。

そうですよ…ボクが作るプログラムなんて、しょせん社内の数人で作るものばかりで、世界をまたにかけた開発だなんて、そんな大げさな…。

何を言っているんだ。君たちは、この本の冒頭から「世界をまたにかけたプログラム」を作ってきたじゃないか。

この本の冒頭で私たちが初めて開発したプログラムは「HelloWorld」でした。それは画面に文字を表示するだけの、クラスを 1 つしか作らない、とてもシンプルなプログラムでしたね。

しかし、この HelloWorld プログラム、**実は 1 つのクラスだけでできているプログラムではありません**。私たちが作成したクラスは 1 つだけですが、実際には多くのクラスから成り立っています。試しに、java コマンドに特殊なオプションを指定して HelloWorld プログラムを実行してみます。

```
>java -verbose:class HelloWorld⏎
[Opened ...rt.jar]
[Loaded java.lang.Object from ...]
[Loaded java.io.Serializable from ...]
[Loaded java.lang.Comparable from ...]
```

```
[Loaded java.lang.CharSequence from ...]
[Loaded java.lang.String from ...]
 :
 :
[Loaded HelloWorld from ...]
 :
 :
```

実行する環境やJVMのバージョンによって多少の違いはありますが、著者の環境では348行の「Loaded ～.～.～」が表示されました。これらの行に表示される内容は、HelloWorldプログラムが動作するためにJVMに読み込まれたクラスの完全限定クラス名です。

つまりHelloWorldプログラムとは、「**私たちが作った1つのクラスが、ほかの347個のクラスと連携して動く、計348クラスからなるプログラム**」だったわけです。

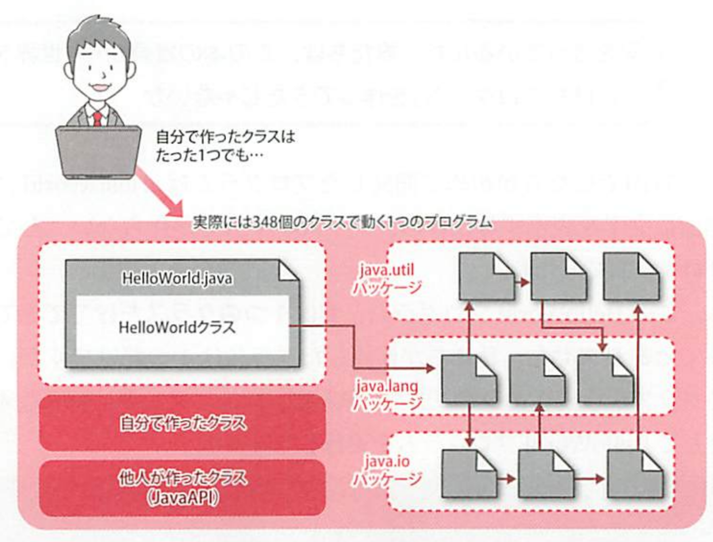


図6-11　自分で作ったJavaのクラスが1つだけだったとしても、数百個のJavaAPIクラスが読み込まれて動作する

私たちが作った1つのクラスを除く347個のクラスは、Javaに初めから標準添付されているクラスであり、それらは**API**（Application Programing Interface）

と総称されます。

Javaでは APIとして、およそ200を越えるパッケージ、3,500を越える多くのクラスが標準提供されていて、私たちプログラム開発者は、それらのクラスをいつでも自由に利用することができます。

たとえば、「5つの要素を持つint型配列」に入っている5つの整数を並び替えるプログラムを開発する場面を想定してみます。並び替えのロジックを自力で開発するのは少し大変ですが、わざわざ自分たちで開発しなくても、APIとして準備されている命令を呼び出せばすぐさま解決できるのです。

リスト6-10　API利用の例

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int[] heights = { 172, 149, 152, 191, 155 };
4     java.util.Arrays.sort(heights);
5     for (int h : heights) {
6       System.out.println(h);
7     }
8   }
9 }
```

Javaが備える並べ替え命令

たった一行で並び替え完了なんて、なんてラクチンなんだ！

しかもラクなだけじゃない。このAPIは数学の専門家が作ったものだから、自分で作るより高速で動くし、バグもないんだよ。

本章まで学んできた今のみなさんであれば、このコードが「java.utilパッケージのArraysクラスにあるsortメソッド」を呼び出していること、そして「java.util.ArraysはJavaが標準で提供するAPIの一部であること」をすぐに理解できるでしょう。

実際、APIに含まれる3,500個を越えるクラスは、それぞれクラスファイル（A

rrays.classなど）の形で、JDKをインストールした際にコンピュータにコピーされています。これらのクラスファイルも、みなさんがHelloWorldクラスを作ったときと同じように、Java言語を作った人たち（その多くが日本国外の技術者）がソースコードを書き、コンパイルして作ったものです。

みなさんは、自分でも気づかないうちに「世界中の人たちが作った347個のクラスと連携して動くクラスを作り、それを動かす」という、世界をまたにかけた開発をしていたのです。何ともスケールの大きい話だと思いませんか？

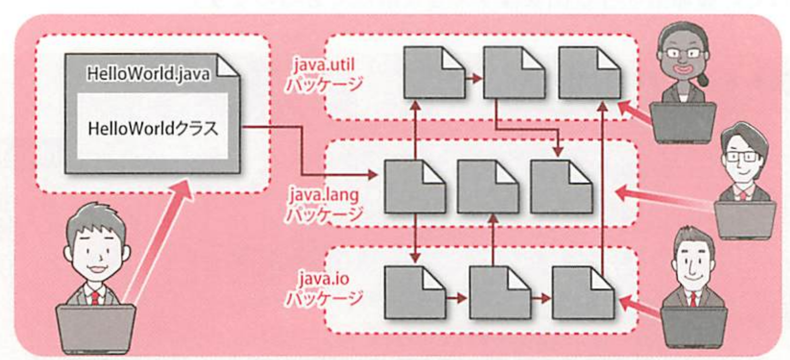


図6-12　Java APIに含まれる3,500個以上のクラスは、世界中の開発者が作り出したものである

6.5.2　APIで提供されるパッケージ

すでに紹介したように、APIには非常にたくさんのパッケージとクラスが含まれていますが、APIのクラスには「java.」または「javax.」で始まるパッケージ名が利用されています。以下は代表的なAPIパッケージです。

表6-1　Java APIに含まれる代表的なパッケージ

java.lang	Javaに欠かせない、重要なクラス群
java.util	プログラミングを便利にするさまざまなクラス群
java.math	数学に関するクラス群
java.net	ネットワーク通信などを行うためのクラス群
java.io	ファイル読み書きなど、データを逐次処理するためのクラス群

特に、java.langパッケージに属するクラスは頻繁に利用するものが多いので、

「特に import 文を記述しなくても自動的にインポートされる」という特別扱いを受けることになっています。java.lang パッケージに属する代表的なクラスとしては、System、Integer、Math、Object、String、Runtime などがあります。

今までずっと画面への表示で使ってきた「System.out.println()」の System は、実は「java.lang.System」クラスだったんだよ。

6.5.3 API リファレンスの読み方

API には、どんなクラスが含まれているんですか？

呼び出すだけで簡単にゲームが作れちゃうようなクラスとかがあったらいいな。

Java が提供してくれている膨大な数の API クラスには、実際にどのようなクラスが含まれていて、どのようなメソッドを持っているか、興味がわいてくるかもしれませんね。

それを調べるためには、API リファレンス（API reference）と呼ばれる API の説明書を読む必要があります。説明書といっても、紙に印刷されたものではありません。Web ページでおなじみの HTML で書かれているファイルで、Web ブラウザを用いて閲覧します。

JDK をダウンロードしたオラクル社のサイトからファイルとしてダウンロードすることも可能ですが、そのままブラウザで閲覧することも可能です。検索サイトで、「Java　API　リファレンス」などのキーワードで検索することで API リファレンスのページに到達できるでしょう。

図 6-13 のように API リファレンスはフレームによって 3 分割されています。

図 6-13　Java の API リファレンスの画面

APIに含まれるあるクラスについて調べたい場合には、画面を次のような手順で操作します。

①左上のフレームで、調べたいクラスが所属するパッケージ名をクリックする。
②左下のフレームで、調べたいクラス名をクリックする。
③右側のフレームに表示されるクラスの説明を読む。

クラスの説明には、概要の説明のほか、そのクラスが持つメソッドやその引数・戻り値の一覧などが詳細に解説されています。

試しにjava.langパッケージのMathクラスを調べてみてください。解説を見ると、randomというメソッドを持っていることや、ほかにも数多くのメソッドを持っていることがわかります。

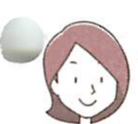
もっとAPIを知りたくなったら

Javaは、以下のように他にもたくさんのAPIを備えています。特によく用いるAPIの利用方法については、本書の続編『スッキリわかるJava入門 実践編 第2版』(インプレス)で紹介しています。

・文字列の比較、照合、編集を行うAPI
・情報をまとめて格納するAPI(コレクション)
・ファイルを読み書きするAPI
・ネットワーク通信を行うAPI
・データベースアクセスを行うAPI
・複数の処理を同時実行するAPI(スレッド)

6.6 クラスが読み込まれるしくみ

6.6.1 必要なときに、必要な分だけ



私たちの HelloWorld を実行したら、裏で 347 もの API クラスが読み込まれて動いていたのには少し感動しました。でも、なぜ 3,500 以上もある API クラスのうち 347 個だけが読み込まれたんでしょうか？

そうだよね。プログラムが動いている途中に「やっぱり別の API クラスも必要になっちゃった」なんてことにはならないのかな？

前節の解説にあるように、1 つの Java プログラムが動くために、とても多くのクラスが JVM に読み込まれて動作します。このように、JVM が必要なクラスファイルを読み込む処理を**クラスローディング**（class loading）といいます。

API として提供されるクラスファイルは 3,500 個を超えますが、JVM は起動

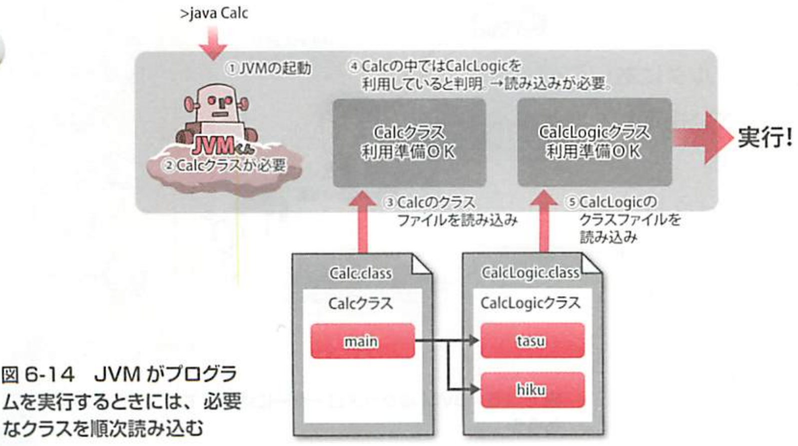


図 6-14　JVM がプログラムを実行するときには、必要なクラスを順次読み込む

直後にそのすべてを読み込むようなことはしません。使わないクラスまでロードしていたらムダにメモリを消費しますし、動作も遅くなってしまうからです。Javaのクラスローディングのしくみは大変に賢く作られていて、一部の例外を除いて「必要になったときに、必要なクラスだけ」を読み込むようになっています。

たとえば、Calcクラスを実行するときの状態を考えてみましょう。最初にJVMはCalc.classを読み込みます。そして、その内容を見て、「Calcクラスの中でCalcLogicクラスを呼び出す必要がある」ことに気づきます。そこでJVMは初めてCalcLogic.classファイルを読みにいくのです。

もしCalcクラスの内部でCalcLogicクラスを使っている箇所をすべて削除した場合には、Calcクラスを実行してもCalcLogicクラスはロードされません。

6.6.2 クラス名だけでクラスファイルを探し出すためのしくみ

JVMの中でクラスファイルを読み込む仕事をしているのは、クラスローダー（class loader）という部分です。たとえば、JVMがクラスローダーに対して、「Calcクラスを利用するから、読み込んで利用可能にしなさい」という指示を出すと、クラスローダーはコンピュータのハードディスクの中にあるCalc.classを読み込みます。

ここで着目してほしいのが、JVMは使いたいクラス名を指定しているだけであって、クラスファイルがハードディスクのどこのフォルダにあるのかをいっさい指定していない点です。

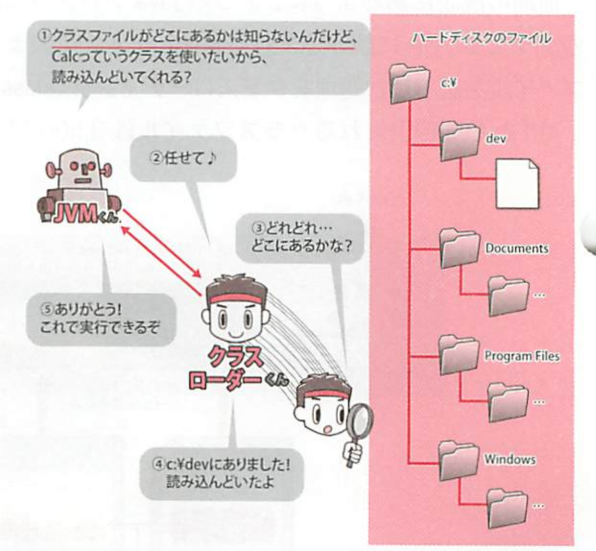


図6-15　JVMはクラスローダーに依頼してクラスを読み込んでもらう

Calc.classという目的のクラスファイルは、c:¥にあるかもしれませんし、c:¥Program Files¥MyCalc¥libにあるかもしれません。しかし、

クラスローダーは膨大な容量を持つハードディスクの中から一瞬で Calc.class ファイルを探し出して読み込んでくれます。

どうして一瞬で見つけられるんですか？　数百 GB もあるハードディスクを検索していたら、数秒…いえ数分はかかってしまいそうです。

確かに。でもクラスローダーはハードディスクの内容をすべて検索したりはせず、賢い方法で探し出すんだ。

クラスローダーは**クラスパス**（classpath）というヒント情報を使うことで、極めて高速に目的のクラスファイルを探し出します。クラスパスとは、「クラスローダーがクラスファイルを探す際に、見にいくべきフォルダの場所」のことで、あらかじめ 1 つ以上の場所を指定しておきます。

たとえばクラスパスとして「c:¥work」が指定してある場合、クラスローダーは c:¥work の中に Calc.class があるか探しにいくだけでよいため、高速に検索することができるのです。

6.6.3 クラスパスの指定方法

今までボクはクラスパスなんて指定してなかったですよ？

そうだね。そのタネあかしをしようか。

JVM が動作するときにクラスファイルを検索する場所であるクラスパスを指定するには、次の 3 つの方法があります。

方法 1：起動時に java コマンドで指定する

java コマンドで JVM を起動する際に、-cp オプションまたは -classpath オプションで指定する方法です。次のように指定します。

```
>java -cp c:¥work Calc⏎
```

方法2：検索場所をOSに登録しておく

javaコマンドを入力するたびに、いちいち-cpオプションを指定するのはめんどうですね。そこで、OSの「環境変数」という設定にクラスパスを登録しておくことができます。javaコマンドは、この環境変数を自動的に読み込んでクラスファイルの検索に利用します。クラスパスは次の方法で登録します。

なお、環境変数の設定方法はOSによって異なりますので、詳細はOSのヘルプファイルや解説書を参照してください。

Windowsの場合

①「コントロールパネル」-「システム」-「システムの詳細設定」-「詳細設定（タブ）」-「環境変数（ボタン）」の順にクリックする。

②ユーザー環境変数としてCLASSPATHを追加し、値を設定する。

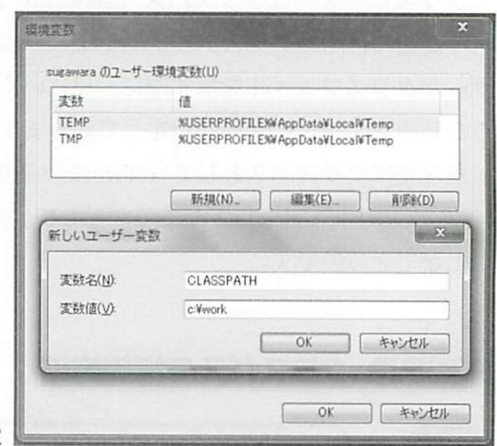


図6-16　環境変数の設定

MacやLinuxの場合

（ユーザーのホームディレクトリ）/.profileというファイルの末尾に以下のような内容で追記する（/var/javadevをクラスパスとする場合）。

```
export CLASSPATH=/var/javadev
```

方法3：特に指定しない

CLASSPATH環境変数に指定がなく、-cpオプションの指定もない場合、デフォルトではjavaコマンドが実行されたフォルダがクラスパスとなります。たとえばc:¥workでjavaコマンドを実行すれば、c:¥workがクラスパスに設定されます。

6.6.4 クラスパスで指定できる対象

クラスパスとして指定することができる対象は、次の3つから選べます。

対象1：フォルダの場所

クラスファイルが置かれているフォルダの場所（絶対パス）を指定するもので、最も一般的です。たとえば「c:¥work」という指定をすると、workフォルダ内のクラスファイルが検索対象となります。

対象2：クラスファイルが入ったJARファイルやZIPファイル

クラスファイルが入っているJARファイルやZIPファイルがあれば、そのファイルの場所（絶対パス）をクラスパスとして指定することができます。クラスローダーは指定されたファイルの中を検索し、もしクラスファイルが見つかれば読み込みます。

たとえば、Calc.classが入ったcalcapp.jarというファイルがc:¥work¥jarsにある場合、「c:¥work¥jars¥calcapp.jar」というクラスパス指定をすることで、Calc.classを読み込むことができるようになります。

対象3：複数のフォルダ、JAR／ZIPファイル、それらの組み合わせ

クラスパスとしては、複数のフォルダやJARファイル、ZIPファイルをデリミタ文字で区切って指定できます。デリミタ文字は、Windowsの場合はセミコロン(;)、LinuxやMacの場合はコロン(:)です。クラスローダーは、クラスファイルを探す際に指定された場所を前から順に探していきます。

Windowsの場合

```
c:¥work;c:¥work¥jars¥calcapp.jar
```

LinuxやMacの場合

```
/var/javadev:/var/javadev/jars/calcapp.jar
```

クラスパスで指定された場所以外にいくらクラスファイルを作っても、JVM はそのクラスを読み込めないんですね。

そのとおり。クラスは作ったのにプログラムがうまく起動できない場合、まずクラスパスを確認しよう。

クラスパスに自動的に加わる rt.jar

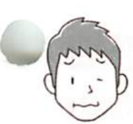
java コマンドでクラスファイルを実行する際には、特に指定しなくても rt.jar（または classes.jar）というファイルがクラスパスに追加されます。このファイルは JDK に含まれているもので、Java をインストールしたフォルダの配下にある lib フォルダの中に含まれています。

試しに、この JAR ファイルを ZIP ファイルの展開ツールで展開してみると、たくさんのフォルダとクラスファイルがあることがわかります。展開されたフォルダには java というフォルダがあり、その中には、lang というフォルダがあり、その中に System.class というクラスファイルがあり……。

そう、これらのファイルこそ膨大な数の API クラス群の実体なのです。rt.jar が自動的にクラスパスに加わるため、私たちは意識することなく System クラスなどの API を利用できていたのですね。

6.7 パッケージに属したクラスの実行方法

6.7.1 実行クラス名の正しい指定



おや？　そういえば、この章の前半で作ったCalcプログラム（p.241のリスト6-8）ですけど、コンパイルはできましたがエラーで実行できないですね。

こんなときは「逃げずに英語のエラーメッセージを読む」だったわね。

この章の前半では、package文を使ってクラスをパッケージに所属させる方法を学びました。CalcクラスとCalcLogicクラスはそれぞれ「calcapp.main」と「calcapp.logics」という別々のパッケージに所属させることができ、リスト6-8はコンパイルも正常に通ります。しかし、完成したCalcクラスをいざ実行しようとするとエラーに直面してしまいます。

```
>java Calc⏎
Exception in thread "main" java.lang.NoClassDefFoundError: Calc
Caused by: java.lang.ClassNotFoundException: Calc
  at java.net.URLClassLoader$1.run(Unknown Source)
  at java.security.AccessController.doPrivileged(Native Method)
  at java.net.URLClassLoader.findClass(Unknown Source)
  at java.lang.ClassLoader.loadClass(Unknown Source)
  at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
  at java.lang.ClassLoader.loadClass(Unknown Source)
```

「NoClassDefFoundError」は直訳すると「クラス定義が見つからない」というエラーです。実はこのプログラム実行方法には2つの問題点があるため、JVMは正しくプログラムを起動できていません。最初の問題点は、そもそも**起動しようとしているクラスの指定が誤っている**こと（図6-17「問題点②」）です。

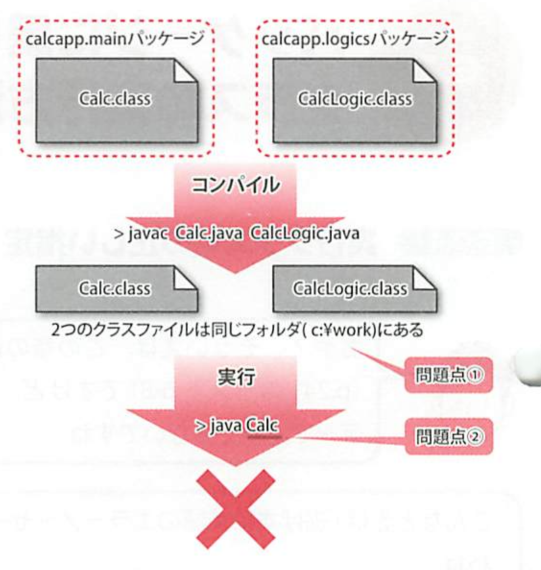


図6-17　コンパイルはうまくいったのに、実行できない!?

初めてjavaコマンドを学習した際に、構文を以下のように紹介しました。

```
>java （ソースファイル名から.javaを取ったもの）
```

これまではこのような理解でも構いませんでしたが、パッケージを利用するようになった今、javaコマンドのより正確な構文を理解する必要があります。

javaコマンドの正確な構文

>java　起動したいクラスの完全限定クラス名（FQCN）

たとえば、計算機プログラムの場合は、次のように起動しなければなりません。

```
>java calcapp.main.Calc⏎
```

この入力により、JVMはクラスパスから「calcapp.main.Calcクラスの中身が格納されているクラスファイル」を自動的に探し出し、そのクラスファイルを読み込んで実行してくれます。

私たちはjavaコマンドを実行する際、「実行したいクラスがハードディスクのどこにあるのか」を指定する必要はありません。「どのクラスを実行したいか」だけを伝えてあげれば、あとはクラスローダーがクラスファイルを自動的に探し出してくれるのです。

「java Calc」ではデフォルトパッケージにあるはずのCalcクラスを実行しようとしてしまうんですね。

6.7.2 クラスファイルの正しい配置

先輩、やっぱり動きません。「Calcクラスが見つからない」っていうエラーが消えなくって…。

どうやらクラスローダーがCalc.classを見つけられていないみたいだね。

FQCNを指定してjavaコマンドを実行しても、まだエラーはなくなりません。実行すると次のようなエラーメッセージが表示されてしまいます。

```
>java calcapp.main.Calc⏎
Exception … java.lang.NoClassDefFoundError: calcapp/main/Calc
        ⋮
Could not find the main class: calcapp.main.Calc.  Program will exit.
```

最後の1行を和訳すると「calcapp.main.Calcというメインクラスが見つからなかった」という意味になります。どうやら、**クラスローダーが目的のクラスファイルを探し出せない**ようです。

前節で解説したとおり、クラスローダーは**クラスパスで指定されたフォルダを対象に**、探しているクラスファイルを調べます。実はこのとき、パッケージに属しているクラスを探す場合には、次のようなルールでクラスファイルを探すことになっています。

クラスローダーの動作原則

あるパッケージx.y.zに属するクラスCを探す場合、クラスローダーは、「クラスパスで指定されたフォルダ¥x¥y¥z¥C.class」というファイルを読み込もうとする。

つまり、パッケージに属したクラスファイルをクラスローダーに読み込んでもらうには、**現在のクラスパスを基準として、パッケージ階層に対応したフォルダ階層を作り、その中に必要なクラスファイルを配置しておく必要がある**のです。

たとえばc:¥workをクラスパスとする場合、コンパイルによって生成されたCalc.classとCalcLogic.classは次のようなフォルダを作成し、その中に配置しておかなければなりません。

```
Calc.class          → c:¥work¥calcapp¥mainフォルダへ
CalcLogic.class     → c:¥work¥calcapp¥logicsフォルダへ
```

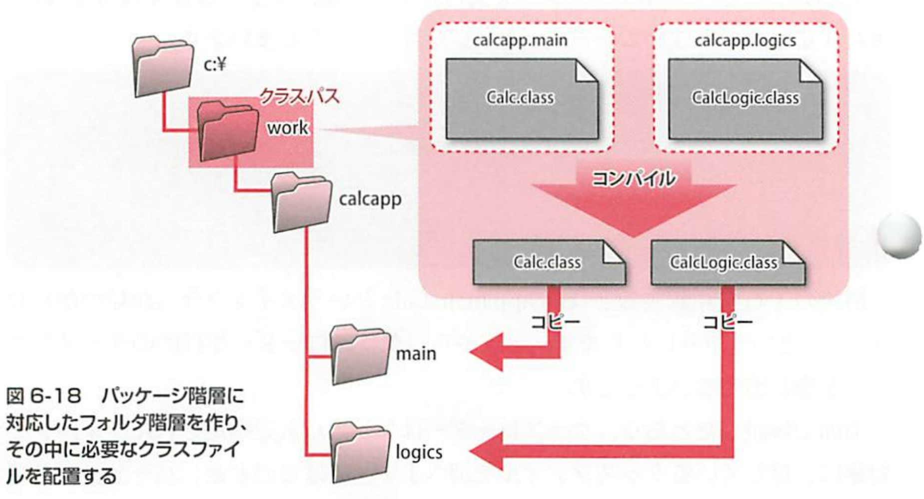


図6-18　パッケージ階層に対応したフォルダ階層を作り、その中に必要なクラスファイルを配置する

クラスファイルを適切なフォルダに置いた上で実行すれば、次のような順序を経て、無事にプログラムは動作するでしょう。

① JVM は起動させるクラス名（calcapp.main.Calc）を受け取る。
② JVM はクラスローダーに対して calcapp.main.Calc を読み込むよう指示する。
③ クラスローダーはクラスパスを確認する。
④ クラスローダーは、クラスパスを基準として「calcapp」→「main」とフォルダを降りていき（すなわち、c:¥work¥calcapp¥main の中）、そこに Calc.class というファイルを発見する。
⑤ クラスローダーは発見した Calc.class を読み込む。
⑥ JVM は読み込み済みの Calc クラスの main メソッドを実行する。

やった！　やっと動きました！

お疲れさま。これで Java の基本は卒業だよ。文法も覚えたし、プログラムをメソッド・クラス・パッケージに部品化する方法もマスターしたね。

API の調べ方ももうわかるから、もうどんな大きなゲームだって作れるはずですよね！　よぉし、スゴいプログラムを作るぞ！

統合開発環境を用いた効率的な開発作業

この章では JDK を用いた開発を通してクラスパスについて学びました。ファイルの配置や起動時の指定など、少し複雑に感じた方もいるかもしれません。実際の開発現場では、より便利で効率的な作業のために「統合開発環境」と呼ばれる開発ツールを利用します。詳細は巻末の付録 B を参照してください。

6.8 第6章のまとめ

この章では、次のようなことを学びました。

クラスの分割

- 複数のクラスで1つのプログラムを構成することができる。
- 別のクラスのメソッドを呼び出す場合は、「クラス名.メソッド名」と指定する。
- Javaプログラムの完成像は複数のクラスファイルの集合体である。
- 実行する際には、mainメソッドが含まれるクラスのFQCNを指定してjavaコマンドを起動する。

パッケージ

- package文を用いて、クラスをパッケージに所属させることができる。
- import文を使うと、コード中のFQCN指定を省略できる。

API

- Javaにあらかじめ添付されている多数のクラス群をAPIという。
- APIは通常「java.」や「javax.」で始まるパッケージ名を用いている。
- java.langパッケージに属するクラスは自動的にインポートされる。
- APIに用意されているクラスは、APIリファレンスで調べることができる。

クラスローダーの動作

- クラスローダーは、読み込み対象クラスのFQCNに基づき、クラスパスを基準としてパッケージ階層に従ったフォルダ構成内を探し、読み込む。
- コンパイルして生成したクラスファイルは、実行時にクラスローダーが見つけられるように、適切なフォルダに配置しなければならない。

練習問題

練習 6-1

次のソースコードを３つのクラスに分割することを考えます。

Main.java

```
1  public class Main {
2    public static void main(String[] args) throws Exception {
4      doWarusa();
5      doTogame();
7      callDeae();
8      showMondokoro();
9    }
10   public static void doWarusa() {
11     System.out.println("きなこでござる。食えませんがの。");
12   }
13   public static void doTogame() {
14     System.out.println("この老いぼれの目はごまかせませんぞ。");
15   }
16   public static void callDeae(){
17     System.out.println
         ("えぇい、こしゃくな。くせ者だ！であえい！");
18   }
19   public static void showMondokoro() throws Exception {
20     System.out.println("飛車さん、角さん。もういいでしょう。");
21     System.out.println("この紋所が目にはいらぬか！");
22     doTogame();   // もう一度、とがめる
23   }
24 }
```

次のルールに従い、このクラスを３つのクラスに分割してください。

① comment パッケージに属する Zenhan クラスを作成し、前半に実行される２つのメソッドをそこに移動する。
② comment パッケージに属する Kouhan クラスを作成し、後半に実行される２つのメソッドをそこに移動する。
③ デフォルトパッケージに属する Main クラスには main メソッドだけを残す。そして、このクラスの先頭では Zenhan クラスだけをインポートする。

なお、２つのメソッド宣言についている「throws Exception」の意味は、現時点では理解しなくて構いません（第 15 章で解説します）。

練習 6-2

練習 6-1 で分割した各ソースファイルをコンパイルし、完成した３つのクラスファイルを適切なフォルダにコピーしてください。その上で、java コマンドを実行し、プログラムを正常に動作させてください。

練習 6-3

showMondokoro() メソッドを修正し、「この紋所が目にはいらぬか！」の後に３秒間の「待ち時間」を入れます。API リファレンスで java.lang.Thread クラスを調べ、プログラムを一時的に止めるメソッドを呼び出すよう修正してください。

練習 6-4

Windows の環境変数 CLASSPATH として、「c:¥work¥ex64」が設定されているとします。このとき、現在のフォルダ（カレントディレクトリ）によらず「java Main」というコマンドで練習 6-3 のプログラムが動作するには、Main.class、Zenhan.class、Kouhan.class を、どのフォルダに配置すればよいか答えてください。

練習 6-5

「java Main」というコマンドを実行すると、練習 6-3 のプログラムが動作する Windows コンピュータがあります。また、このコンピュータの Zenhan.class は、「c:¥javaapp¥koumon¥comment」というフォルダの中に存在しています。このとき環境変数 CLASSPATH として設定されている内容を答えてください。

練習 6-1 の解答

Main.java

```
1  import comment.Zenhan;
2  public class Main {
3    public static void main(String[] args) throws Exception {
4      Zenhan.doWarusa();
5      Zenhan.doTogame();
6      comment.Kouhan.callDeae();
7      comment.Kouhan.showMondokoro();
8    }
9  }
```

前半（4～5行目）　後半（6～7行目）

Zenhan.java

```
1  package comment;
2  public class Zenhan {
3    public static void doWarusa() {
4      System.out.println("きなこでござる。食えませんがの。");
5    }
6    public static void doTogame() {
7      System.out.println("この老いぼれの目はごまかせませんぞ。");
8    }
9  }
```

Kouhan.java

```
1  package comment;
2  public class Kouhan {
3    public static void callDeae() {
4      System.out.println
           ("えぇい、こしゃくな。くせ者だ！であえい！");
```

```
5   }
6   public static void showMondokoro() throws Exception {
7     System.out.println("飛車さん、角さん。もういいでしょう。");
8     System.out.println("この紋所が目にはいらぬか！");
9     Zenhan.doTogame();    // もう一度、とがめる
10  }
11 }
```

練習 6-2 の解答

ここでは一般的と考えられる方法を示します。他に、環境変数を設定するなどの方法があります。

1. コンピュータに適当なフォルダ(たとえば、c:¥japp とする)を作成する。
2. C:¥japp フォルダの中に、Main.class をコピーする。
3. C:¥japp の中に comment というフォルダを作成する。
4. C:¥japp¥comment の中に、Zenhan.class と Kouhan.class をコピーする。
5. C:¥japp を現在のフォルダ(カレントディレクトリ)とする。
6. 「java Main」として java コマンドを実行する。

練習 6-3 の解答

showMondokoro( ) メソッドのみを抜粋してあります。

```
1 public static void showMondokoro() throws Exception {
2   System.out.println("飛車さん、角さん。もういいでしょう。");
3   System.out.println("この紋所が目にはいらぬか！");
4   Thread.sleep(3000);     // この行を追加
5   Zenhan.doTogame();      // もう一度、とがめる
6 }
```

練習 6-4 の解答

Main.class　→　c:¥work¥ex64 フォルダ
Zenhan.class と Kouhan.class　→　c:¥work¥ex64¥comment フォルダ

練習 6-5 の解答

c:¥javaapp¥koumon

似ているようで異なる java と javac の引数

付録 A.3.4 項で「javac でコンパイルするときにはソースファイル名に拡張子 ".java" を付けるが、java コマンドでクラスファイルを実行するときには拡張子は不要」と強調しました。なぜ、このような違いがあるのでしょうか？

実は、java コマンドと javac コマンドの引数には次のような意味の違いあるのです。

- javac コマンドは「どのソースファイルをコンパイルするか」をファイル名で指定して実行するもの
- java コマンドは「どのクラスの main メソッドを起動するか」をクラス名（FQCN）で指定して実行するもの

両者は、まったく別のものを指定するコマンドだったのですね。