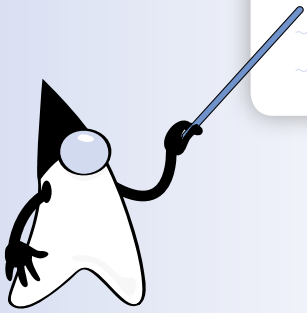
第８章

クラスの基本

オブジェクト指向プログラミングを支える最も根幹的で基礎的な技術を提供するのが、クラスの考え方です。本章では、銀行口座を扱うプログラムと自動車を扱うプログラムを通じて、クラスの基本を学習します。

- □クラス
- □クラス型変数
- □メンバアクセス演算子
- □インスタンスとオブジェクト
- □フィールドとインスタンス変数
- □コンストラクタとインスタンスメソッド
- □this
- □データ隠蔽とカプセル化



8-1 クラスとは

前章では、一連の処理をまとめたプログラムの部品であるメソッドについて学習しました。メソッドと、その処理対象となるデータを組み合わせた構造を表すのが、クラスです。メソッドより一回り大きな単位の部品であるクラスは、オブジェクト指向プログラミングを支える最も根幹的で基礎的な技術です。本節では、クラスの基本を学習します。

データの扱い

List 8-1 のプログラムを見てください。足立君と仲田君の銀行口座のデータを表す変数に値を設定して表示する、という単純なプログラムです。

List 8-1 Chap08/Accounts.java

```
// 二人分の銀行口座データを扱うプログラム

class Accounts {

   public static void main(String[] args) {
      String adachiAccountName    = "足立幸一";     // 足立君の口座名義
      String adachiAccountNo      = "123456";       //   〃   の口座番号
      long   adachiAccountBalance = 1000;           //   〃   の預金残高

      String nakataAccountName    = "仲田真二";     // 仲田君の口座名義
      String nakataAccountNo      = "654321";       //   〃   の口座番号
      long   nakataAccountBalance = 200;            //   〃   の預金残高

      adachiAccountBalance -= 200;                  // 足立君が200円おろす
      nakataAccountBalance += 100;                  // 仲田君が100円預ける

      System.out.println("■足立君の口座");
      System.out.println("  口座名義：" + adachiAccountName);
      System.out.println("  口座番号：" + adachiAccountNo);
      System.out.println("  預金残高：" + adachiAccountBalance);

      System.out.println("■仲田君の口座");
      System.out.println("  口座名義：" + nakataAccountName);
      System.out.println("  口座番号：" + nakataAccountNo);
      System.out.println("  預金残高：" + nakataAccountBalance);
   }
}
```

実行結果

```
■足立君の口座
  口座名義：足立幸一
  口座番号：123456
  預金残高：800
■仲田君の口座
  口座名義：仲田真二
  口座番号：654321
  預金残高：300
```

２人分の銀行口座データを６個の変数で表しています。たとえば、*adachiAccountName* は**口座名義**、*adachiAccountNo* は**口座番号**、*adachiAccountBalance* は**預金残高**です。

『名前が *adachi* で始まる変数は、**足立君の銀行口座に関するものである。**』

ということは、変数名やコメントから推測できます。

とはいうものの、足立君の口座名義を *nakataAccountNo* で表したり、仲田君の口座番号を *adachiAccountName* で表したりすることも不可能ではありません。

問題は、変数間の関係を変数名から**推測**はできるものの**確定ができないこと**です。バラバラに宣言された口座名義・口座番号・預金残高の変数が、一つの銀行口座に関するものであるという関係は、プログラム上で表現されていません。

クラス

私たちがプログラムを作る際は、**現実世界のオブジェクト（物）や概念を、プログラムの世界のオブジェクト（変数）に投影します。**

本プログラムでは、**Fig.8-1** a に示すように、一つの《口座》に関する口座名義・口座番号・預金残高のデータが、個別の変数へと投影されています。

▶ この図は一般化して表したものです。足立君の口座・仲田君の口座に対して、三つのデータが別々の変数として投影されます。

口座の一側面ではなく複数の側面に着目して、図 b に示すように、口座名義と口座番号と預金残高を、ひとまとめにしたオブジェクトとして投影したらよさそうです。このような投影を行うのが**クラス**（*class*）の考え方の基本です。

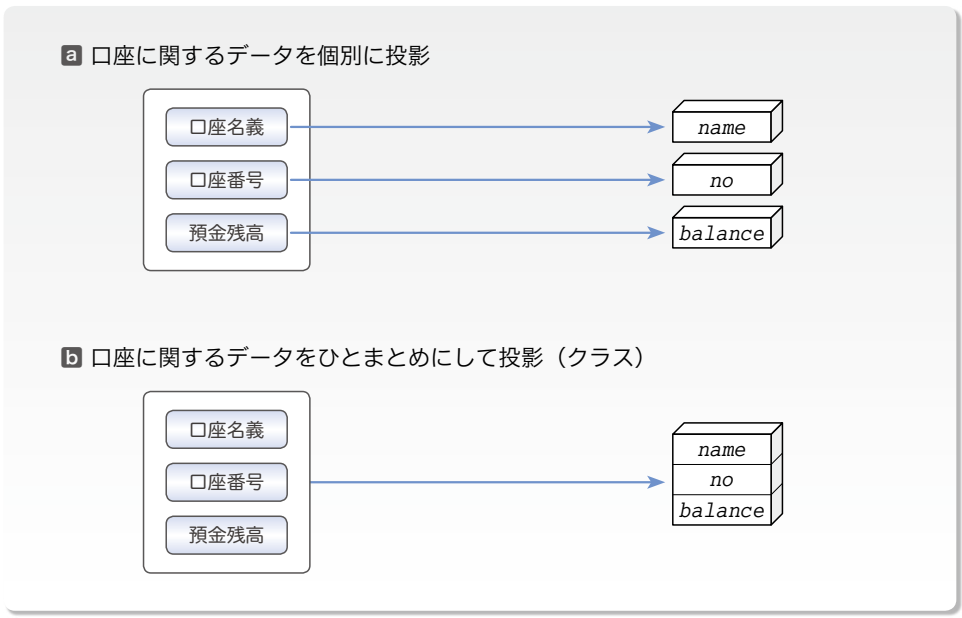


Fig.8-1　オブジェクトの投影とクラス

プログラムで扱う問題の種類や範囲によっても異なりますが、現実の世界からプログラムの世界への投影は、

・まとめるべきものは、まとめる。

・本来まとまっているものは、そのままにする。

といった方針にのっとると、より自然で素直なものとなります。

クラス

前ページの方針に基づいて書き直したプログラムを **List 8-2** に示します。

List 8-2 Chap08/AccountTester.java

```
// 銀行口座クラス【第１版】とそれをテストするクラス

// 銀行口座クラス【第１版】          クラス宣言
class Account {
    String name;        // 口座名義
    String no;          // 口座番号
    long balance;       // 預金残高
}

// 銀行口座クラスをテストするクラス
class AccountTester {

    public static void main(String[] args) {
        Account adachi = new Account();  // 足立君の口座
        Account nakata = new Account();  // 仲田君の口座

        adachi.name    = "足立幸一";      // 足立君の口座名義
        adachi.no      = "123456";       //   〃   の口座番号
        adachi.balance = 1000;           //   〃   の預金残高

        nakata.name    = "仲田真二";      // 仲田君の口座名義
        nakata.no      = "654321";       //   〃   の口座番号
        nakata.balance = 200;            //   〃   の預金残高

        adachi.balance -= 200;           // 足立君が200円おろす
        nakata.balance += 100;           // 仲田君が100円預ける

        System.out.println("■足立君の口座");
        System.out.println("  口座名義：" + adachi.name);
        System.out.println("  口座番号：" + adachi.no);
        System.out.println("  預金残高：" + adachi.balance);

        System.out.println("■仲田君の口座");
        System.out.println("  口座名義：" + nakata.name);
        System.out.println("  口座番号：" + nakata.no);
        System.out.println("  預金残高：" + nakata.balance);
    }
}
```

1 2 3

実行結果

```
■足立君の口座
  口座名義：足立幸一
  口座番号：123456
  預金残高：800
■仲田君の口座
  口座名義：仲田真二
  口座番号：654321
  預金残高：300
```

本プログラムは、**二つのクラスから構成される**という点で、これまでのプログラムとは大きく異なります。それぞれのクラスの概略は以下の通りです。

Account	：	銀行口座クラス
AccountTester	：	クラス *Account* をテストするクラス

これまでのプログラムのクラスは、**main** メソッドを中心とする構造でした。本プログラムでは、それに相当するのがクラス *AccountTester* です。プログラムの実行時には、このクラス中の **main** メソッドが実行されます。

ソースプログラムのファイル名も、このクラス名に拡張子 .java を付けたものとなっています（AccountTester.java であって、Account.java ではありません）。

プログラムのコンパイルは、以下のように行います（カレントディレクトリは、第8章用のディレクトリである Chap08 であるとします）。

```
▶ javac AccountTester.java↵
```

クラスファイルは、クラスごとに作られることになっていますので、二つのクラスファイル Account.class と AccountTester.class が生成されます（**Fig.8-2**）。java コマンドで実行するのは、クラス *AccountTester* です。

```
▶ java AccountTester↵
```

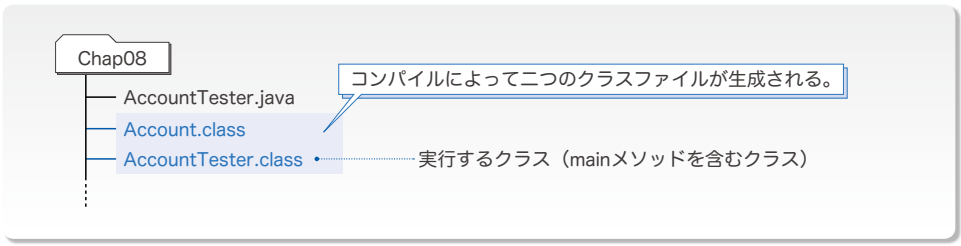


Fig.8-2　ソースプログラムから作られるクラスファイル

クラス宣言

1 に着目しましょう。これは、クラス *Account* が“口座名義・口座番号・預金残高をセットにしたもの”であることを表すための宣言です。先頭の“**class** *Account* {”が宣言の開始であり、その宣言は“}”まで続きます。このような宣言は、**クラス宣言**（*class declaration*）と呼ばれるのでしたね（p.12）

{ } の中に置かれているのは、クラスを構成するデータを表す**フィールド**（*field*）の宣言です。クラス *Account* は、三つのフィールドから構成されています（**Fig.8-3**）。

- 口座名義を表す **String** 型の *name*
- 口座番号を表す **String** 型の *no*
- 預金残高を表す **long** 型の *balance*

▶ “フィールド”という用語は、前章で学習しました（p.236）。

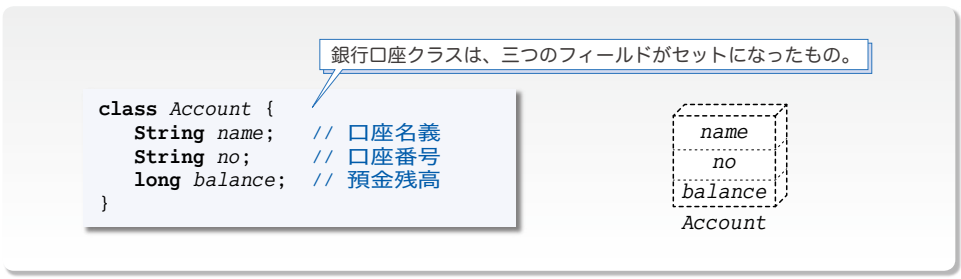


Fig.8-3　クラスとフィールド

クラスとオブジェクト

クラス宣言は、《型》を宣言するものであって、実体（変数）を宣言するものではありません。クラス *Account* 型の変数は、以下のように宣言します。

```
Account adachi;              // 足立君の口座（クラス型変数）
Account nakata;              // 仲田君の口座（クラス型変数）
```

ただし、この宣言で作られる *adachi* や *nakata* は、銀行口座クラスの実体ではなく、それを**参照するための**クラス型変数（*class type variable*）です。

▶ 配列の本体を参照する変数が、配列変数でした（第 6 章）。それと同じです。

クラスは、タコ焼きを焼くための「カタ」のようなものであって、本物のタコ焼きである実体は、別に生成する必要があります。配列の場合と同様に、クラスの本体である「実体」の生成は、**new** 演算子を利用して以下のように行います。

```
new Account()
```

この式は、“**new** クラス名 **()**”という形式ですね。

クラス型変数 *adachi* と *nakata* の宣言と、本体である「実体」の生成の様子を示したのが、**Fig.8-4** です。

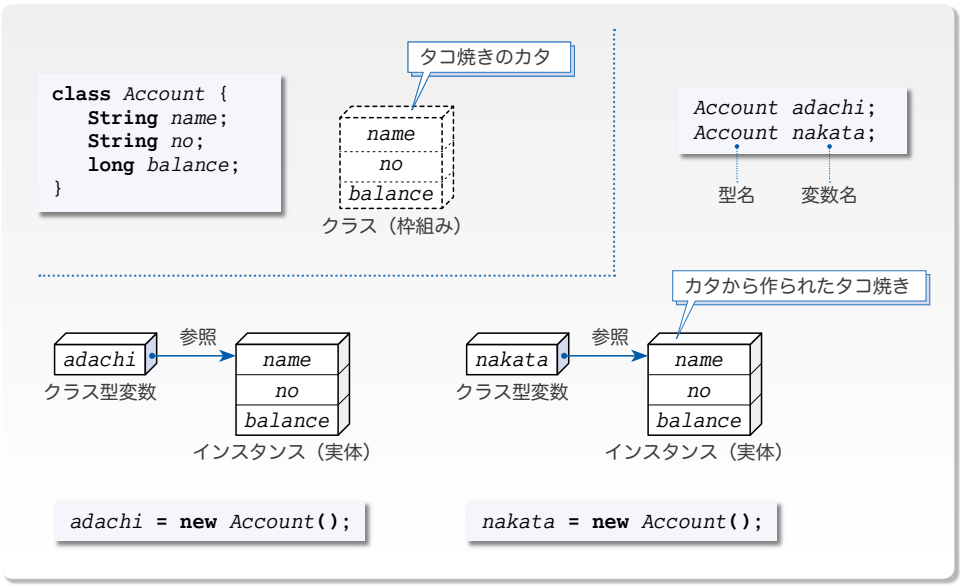


Fig.8-4　クラスとインスタンス

new 演算子によって生成されたクラス型の「実体」を**インスタンス**（*instance*）と呼び、インスタンスを生成することを**インスタンス化**と呼びます。これらの用語は、必ず覚えましょう。

重要 クラス型の実体のことをインスタンスと呼び、インスタンスを生成することをインスタンス化と呼ぶ。

クラス型変数とインスタンスは、関係付けが必要です。そのために行うのが、以下に示す代入です。

```
adachi = new Account();
nakata = new Account();
```

これで、*adachi* や *nakata* は生成されたインスタンスを参照することになります。

▶ **new** 演算子による生成式を評価すると、**インスタンスへの参照**が得られます。生成されたインスタンスへの参照が、変数 *adachi* と *nakata* に代入されます。

*

配列型と同様に、クラス型は**参照型**の一種です（p.142）。

そのため、変数とは別に本体（実体）を生成し、それらを関連付ける手順は、**配列とほぼ同じです。Fig.8-5** に示すように、クラス型変数（配列変数）の宣言の際に、本体を生成する式を初期化子として与えると、プログラムは簡潔になります。本プログラムの2の箇所も、このようになっています。

a クラス

```
型      クラス型変数 = 生成式;
Account adachi = new Account();     // Account型のadachi
Account nakata = new Account();     // Account型のnakata
```

b 配列

```
型         配列変数  = 生成式;
int[]      a     = new int[10];     // int型の配列a
```

Fig.8-5 配列とクラスの生成

配列の本体を**オブジェクト**（*object*）と呼ぶことを第 6 章で学習しました。クラスのインスタンスと配列の本体は、いずれも **new** によって動的に生成されます。オブジェクトとは、プログラムの実行時に動的に生成される実体の総称です。

重要 クラスのインスタンスと配列の本体をオブジェクトと総称する。

▶ クラス・インスタンス・オブジェクトという言葉の意味を以下に示します。
class …『組』、『部類』、『項目』。
instance …『実例』、『事実』。
object …『物体』、『対象』、『目的』。

インスタンス変数とフィールドアクセス

クラス *Account* 型のインスタンスは、口座名・口座番号・預金残高が“セット”になった変数です。その中の特定のフィールドをアクセスするために利用するのが、**Table 8-1** に示す**メンバアクセス演算子**（*member access operator*）です。この演算子の通称は**ドット演算子**ですから、その名前も覚えておきましょう。

Table 8-1　メンバアクセス演算子（ドット演算子）

x.y	*x* が参照するインスタンス内のメンバ（要素）*y* をアクセスする。

▶ フィールドを特定することから**フィールドアクセス演算子**とも呼ばれます。ただし、この演算子はメソッドの特定でも利用されます（p.283）。なお、ドット（dot）は『点』という意味です。

たとえば、足立君の口座の各フィールドをアクセスする式は次のようになります。

```
adachi.name       // 足立君の口座名義
adachi.no         //   〃   の口座番号
adachi.balance    //   〃   の預金残高
```

仲田君の口座のフィールドも同様にアクセスできます（**Fig.8-6**）。

▶ メンバアクセス演算子は、日本語の“の”に相当します。たとえば、*adachi.name* は《足立君の口座名義》であり、*nakata.balance* は、《仲田君の預金残高》です。

*

インスタンス内のフィールドは、インスタンスごとに作られる変数ですから、**インスタンス変数**（*instance variable*）と呼ばれます。*adachi.name* も *nakata.balance* もインスタンス変数です。

重要　インスタンス内の個々のフィールドであるインスタンス変数は、メンバアクセス演算子．を利用した“クラス型変数名．フィールド名”によってアクセスできる。

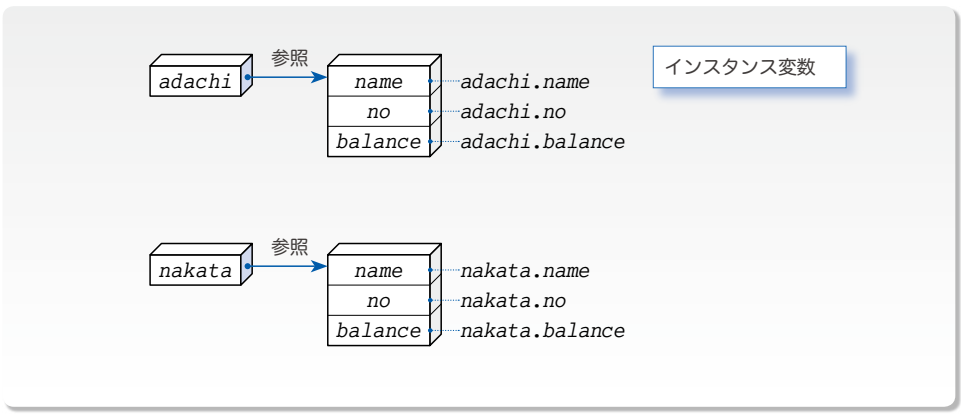


Fig.8-6　フィールド（インスタンス変数）のアクセス

フィールドの初期化

プログラムから、各インスタンス変数に値を設定する3の部分を削除してみましょう。プログラムを実行すると、右に示す実行結果が得られます。

実行結果

```
■足立君の口座
  口座名義：null
  口座番号：null
  預金残高：0
■仲田君の口座
  口座名義：null
  口座番号：null
  預金残高：0
```

この結果から、**String**型の口座名義と口座番号のインスタンス変数は空参照 **null** で初期化されて、**long** 型の預金残高のインスタンス変数は 0 で初期化されていることが分かります。

配列内の個々の構成要素は、**“既定値”**である 0 で初期化される（p.183）のでしたね。それと同じです。クラス内の個々のインスタンス変数も、既定値で初期化されるのです。

重要 クラスインスタンス内のフィールドであるインスタンス変数は、既定値で初期化される（配列の構成要素と同様である）。

▶ クラス型である **String** 型は、一種の参照型です。参照型の既定値は、空参照 **null** です（p.183）から、口座名義と口座番号を出力すると『null』と表示されます（p.206）。

問題点

クラスの導入によって、口座のデータを表す変数間の関係がプログラム中に明確に埋め込まれました。しかし、まだ問題が残っています。

① 確実な初期化に対する無保証

口座インスタンスの各フィールドは、明示的に**初期化されていません**。インスタンスを作った後で値を**代入**しているだけです。

値を設定するかどうかがプログラマに委ねられている状態となっていますので、初期化を忘れた場合は、思いもよらぬ結果が生じる危険性があります。実際、上に示した例では、口座名義と口座番号が **null** になっています。これは、まずいですね。

初期化すべきフィールドは、初期化を強制すべきです。

② データの保護に対する無保証

足立君の預金残高である *adachi.balance* の値は、プログラム（他のクラス）から自由に読書きできます。このことを現実の世界に置きかえると、足立君でなくても（通帳や印鑑がなくても）、足立君の口座から、勝手にお金をおろせることになります。

一般に、口座番号を公開することはあっても、預金残高を操作できるような状態で公開するといったことは、現実の世界ではあり得ません。

上記の問題点を解決するようにクラスを定義・利用することができます。そのように改良しましょう。

銀行口座クラス 第2版

改良したプログラムを **List 8-3** に示します。クラス *Account* が複雑になった一方で、それを利用するクラス *AccountTester* は簡潔になっています。

▶ 本書では、複数の段階にわたってクラスを改良することがあります。その際は、順番に“第＊版”と銘打つことにします。

＊

Fig.8-7 に示すように、このソースプログラムは、ディレクトリ account2 に格納することにします。というのも、第1版と第2版を同じディレクトリに入れるわけにはいかないからです。同一名のクラスをもつソースファイルを、同一ディレクトリ中に入れるのは混乱のもとです。

▶ 第2版のテストクラスの名前を *AccountTester2* とし、ファイル名を AccountTester2.java としたら、問題が解決できるような気がします。

しかし、銀行口座クラスの名前が *Account* のままだと、コンパイルの結果作られるクラスファイルの名前 Account.class が第1版のものと衝突します。異なるソースファイルから、同一名のクラスファイルが作られるのは、まずいですね。**同一ディレクトリ上のソースファイル中に、同一名のクラスを存在させないようにする必要があります。**

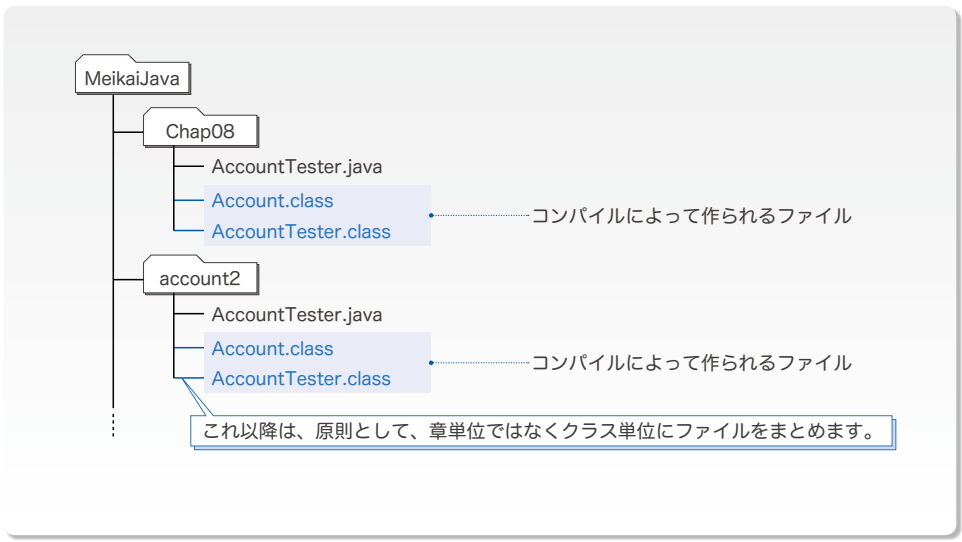


Fig.8-7 ディレクトリの構成

なお、次章以降も同様にします。一般に、クラス *Abcd* の“第?”版は、abcd? という名前のディレクトリに格納することにします。

▶ ディレクトリの先頭文字は小文字とします。その理由は、第11章で学習します。なお、テスト的な小規模のプログラムは、これまで通り Chap** ディレクトリに格納します。

List 8-3 account2/AccountTester.java

```
// 銀行口座クラス【第２版】とそれをテストするクラス

// 銀行口座クラス【第２版】
class Account {
   private String name;        // 口座名義
   private String no;          // 口座番号
   private long balance;       // 預金残高

   //--- コンストラクタ ---//
   Account(String n, String num, long z) {
      name = n;                // 口座名義
      no = num;                // 口座番号
      balance = z;             // 預金残高
   }

   //--- 口座名義を調べる ---//
   String getName() {
      return name;
   }

   //--- 口座番号を調べる ---//
   String getNo() {
      return no;
   }

   //--- 預金残高を調べる ---//
   long getBalance() {
      return balance;
   }

   //--- k円預ける ---//
   void deposit(long k) {
      balance += k;
   }

   //--- k円おろす ---//
   void withdraw(long k) {
      balance -= k;
   }
}

// 銀行口座クラス【第２版】をテストするクラス
class AccountTester {

   public static void main(String[] args) {
      // 足立君の口座
      Account adachi = new Account("足立幸一", "123456", 1000);
      // 仲田君の口座
      Account nakata = new Account("仲田真二", "654321",  200);

      adachi.withdraw(200);              // 足立君が200円おろす
      nakata.deposit(100);               // 仲田君が100円預ける

      System.out.println("■足立君の口座");
      System.out.println("  口座名義：" + adachi.getName());
      System.out.println("  口座番号：" + adachi.getNo());
      System.out.println("  預金残高：" + adachi.getBalance());

      System.out.println("■仲田君の口座");
      System.out.println("  口座名義：" + nakata.getName());
      System.out.println("  口座番号：" + nakata.getNo());
      System.out.println("  預金残高：" + nakata.getBalance());
   }
}
```

実行結果

```
■足立君の口座
  口座名義：足立幸一
  口座番号：123456
  預金残高：800
■仲田君の口座
  口座名義：仲田真二
  口座番号：654321
  預金残高：300
```

データ隠蔽

新しいクラス *Account* の構造を示したのが、**Fig.8-8** です。クラス宣言の内部が、大きく三つの部分で構成されていることが分かります。

▶ 本クラスでは、フィールド・コンストラクタ・メソッドの順に並べています。それぞれは、まとまっている必要もありませんし、順序も任意です。

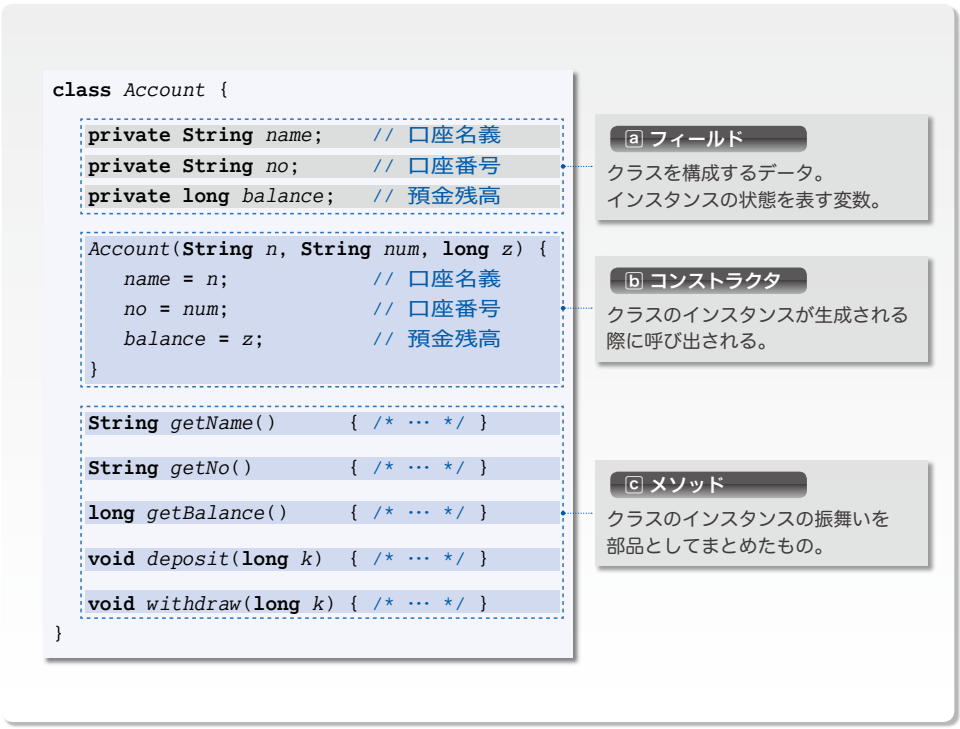


Fig.8-8 クラスAccountの構造

a フィールド

クラスに三つのフィールド *name*, *no*, *balance* が存在する点は第 1 版と同じです。ただし、フィールドの宣言には、キーワード **private** が付いています。

private の付いたフィールドのアクセス性は、**非公開アクセス**（*private access*）となります。非公開アクセスのフィールドは、**クラスの外部に対して存在を隠します。**

> **重要** **private** 宣言されたフィールドは、クラスの外部に対して非公開となる。

そのため、クラス *Account* にとって外部である、クラス *AccountTester* の **main** メソッドでは、非公開のフィールド *name*, *no*, *balance* にアクセスすることはできません。

もしも **main** メソッドに以下のようなプログラムがあれば、コンパイル時にエラーとなります。

```
adachi.name = "柴田望洋";  // エラー：足立君の口座名義を書きかえる
adachi.no = "999999";      // エラー：足立君の口座番号を書きかえる
System.out.println(adachi.balance);    // エラー：足立君の預金残高を表示
```

クラスの外部から『お願いですから、このデータを特別に見せてください。』と頼むことはできません。**情報を公開するかどうかを決定するのはクラス側です。**

データを外部から隠して不正なアクセスから守ることを**データ隠蔽**（*data hiding*）といいます。みなさんは、銀行口座のキャッシュカードの暗証番号を秘密にしていますよね。それと同じです。フィールドを非公開としてデータ隠蔽を行えば、**データの保護性・隠蔽性だけでなく、プログラムの保守性も向上する**ことが期待できます。

すべてのフィールドは、原則として非公開にします。

重要 データ隠蔽を実現して、プログラムの品質を向上させるために、クラス内のフィールドは、原則として非公開にすべきである。

▶ この後で学習するように、フィールドの値は、コンストラクタとメソッドを通じて間接的に読んだり書いたりできます。したがって、フィールドを非公開することによって不都合が生じることは基本的にありません。

なお、**private** が指定されていないフィールドは**デフォルトアクセス**（*default access*）となります。デフォルトアクセスは《公開》のことである、と理解しておきましょう。

▶ もう少し厳密に説明すると、パッケージ内で《公開》となり、パッケージ外部で《非公開》となります。このことから、デフォルトアクセスは、**パッケージアクセス**（*package access*）とも呼ばれます。詳細は第 11 章で学習します。

b コンストラクタ

b の部分は**コンストラクタ**（*constructor*）と呼ばれます。形だけ見ると、メソッドと似ています。ただし、以下の点がメソッドとは異なります。

・クラスと同じ名前である。
・返却型がない。

詳細は次ページで学習します。

c メソッド

c はメソッドです。メソッドについては、前章で学習しましたね。ただし、前章で学習したメソッドとは異なり、**static** を付けずに宣言されています。

コンストラクタを学習した後に、p.282 で学習します。

*

なお、フィールドやメソッドのことを**メンバ**（*member*）と総称します。

▶ member は、『会員』『構成員』『一部』といった意味の語句です。後の章で学習しますが、文法の定義上、コンストラクタはメンバには含まれません。

コンストラクタ

メソッドとよく似た形をした**コンストラクタ**（*constructor*）の役割は、クラスのインスタンスを初期化することです。

コンストラクタが呼び出されるのは、インスタンスの生成時です。すなわち、プログラムの流れが以下の宣言文を通過して、網かけ部の式が評価される際に、コンストラクタが呼び出されて実行されます。

```
Account adachi = new Account("足立幸一", "123456", 1000); ➊
Account nakata = new Account("仲田真二", "654321",  200); ➋
```


Fig.8-9 に示すように、呼び出されたコンストラクタは、仮引数 *n*，*num*，*z* に受け取った値をフィールド *name*，*no*，*balance* に代入します。

代入先は、*adachi.name* や *nakata.name* ではなく、**単なる** *name* です。➊で呼び出されたコンストラクタでの *name* は *adachi.name* を表し、➋で呼び出されたコンストラクタでの *name* は *nakata.name* を表します。

このように、フィールド名だけで表せるのは、コンストラクタが、**自分自身のインスタンスが何者であるのかを知っているからです**。図に示すように、個々のインスタンスに対して、専用のコンストラクタが存在するのです。

▶ 個々のインスタンスにコンストラクタを用意することは、現実には不可能です。『個々のインスタンスに対して、専用のコンストラクタが存在する』というのは、概念上のものであって、物理的にそうなっているわけではありません。コンパイルによって生成されるコンストラクタ用の内部的なコードは、実際は 1 個だけです。

さて、➊と➋の宣言を以下のように書きかえてみましょう。コンパイルすると、エラーになります。

```
Account adachi = new Account();              // エラー：引数がない
Account nakata = new Account("仲田真二");    // エラー：引数が不足
```

このことから、コンストラクタが**不完全あるいは不正な初期化を防止する**ことが分かりますね。コンストラクタの役目は、**インスタンスを適切に初期化する**ことです。

> **重要** クラス型を宣言するときは、必ずコンストラクタを用意して、インスタンスを確実かつ適切に初期化する手段を提供せよ。

▶ construct は『構築する』という意味です。そのため、コンストラクタは**構築子**と呼ばれることもあります。

なお、コンストラクタはメソッドとは異なり、**値を返却できません**。誤って返却型を指定しないようにしましょう。

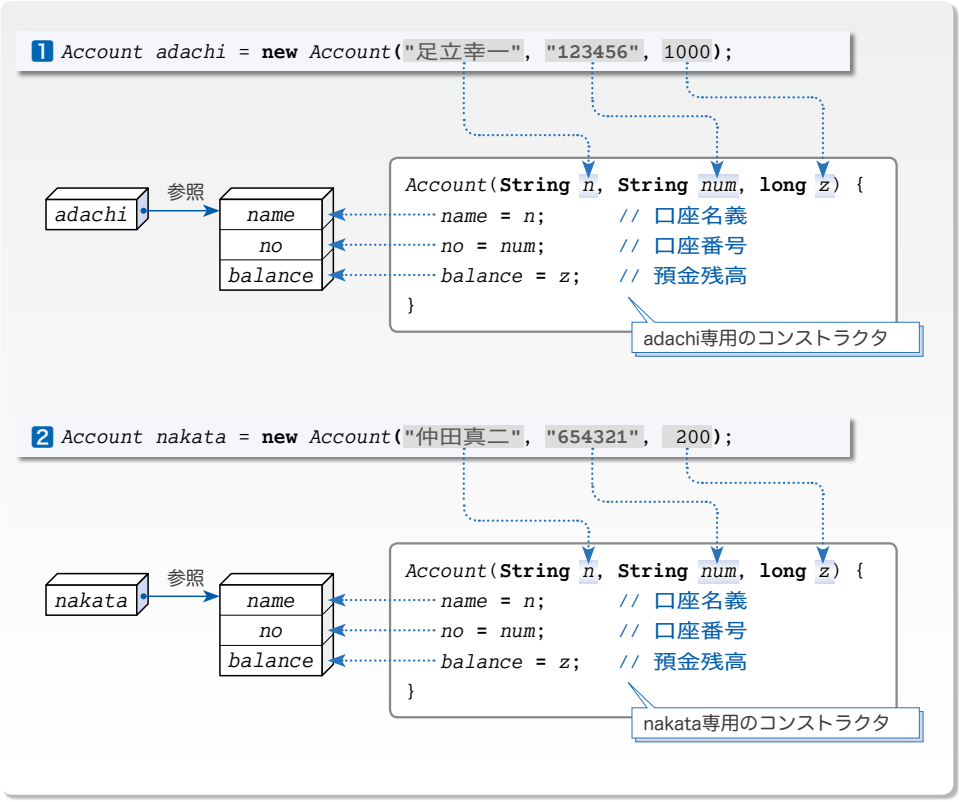


Fig.8-9　クラスのインスタンスとコンストラクタ

第 1 版のクラス *Account* ではコンストラクタを定義していませんでした。コンストラクタを定義していないのに、どうしてインスタンスを生成できたのでしょうか。

実は、コンストラクタを定義しないクラスには、引数を受け取らず、その本体が空である**デフォルトコンストラクタ**（*default constructor*）が自動的に作られるのです。

重要　クラスにコンストラクタを定義しなければ、本体が空のデフォルトコンストラクタが自動的に定義される。

すなわち、第 1 版のクラス *Account* では、以下のコンストラクタがコンパイラによって作られていたのです。

```
Account() { }
```

▶ 第 1 版のクラス *AccountTester* で、以下のように ( ) の中を空にしてインスタンスを生成したのは、引数を受け取らないデフォルトコンストラクタを呼び出すためだったことが分かりますね。

```
Account adachi = new Account();      // 引数を受け取らないコンストラクタを呼び出す
```

なお、デフォルトコンストラクタの内部は、本当は空ではありません。第 12 章で学習します。

メソッド

フィールドとコンストラクタを学習しました。最後に残ったのがメソッドです。クラス *Account* のメソッドを **Fig.8-10** に再掲します。これらの概要は以下のとおりです。

■ getName

口座名義を調べるためのメソッドです。フィールド *name* の値を **String** 型で返します。

■ getNo

口座番号を調べるためのメソッドです。フィールド *no* の値を **String** 型で返します。

■ getBalance

預金残高を調べるためのメソッドです。フィールド *balance* の値を **long** 型で返します。

■ deposit

お金を預けるためのメソッドです。預金残高が *k* 円だけ増えることになります。

■ withdraw

お金をおろすためのメソッドです。預金残高が *k* 円だけ減ることになります。

```
// 口座名義を調べる
String getName() {
    return name;
}

// 口座番号を調べる
String getNo() {
    return no;
}

// 預金残高を調べる
long getBalance() {
    return balance;
}

// k円預ける
void deposit(long k) {
    balance += k;
}

// k円おろす
void withdraw(long k) {
    balance -= k;
}
```

Fig.8-10
クラスAccountのメソッド

前章のメソッドとは異なり、クラス *Account* の全メソッドが **static** を付けずに宣言されています。**static** を付けずに宣言されたメソッドは、そのクラスの個々のインスタンスごとに作られます。

すなわち、*adachi* も *nakata* も、自分専用のメソッド *getName*, *getNo*, *getBalance*, … をもつのです。

▶ 個々のインスタンスごとにメソッドが作られるというのは、あくまでも概念上のものです。コンストラクタと同様に、コンパイルされたクラスファイル内に作られるコードは 1 個です。

static の付かないメソッドは、**インスタンスメソッド**（*instance method*）と呼ばれます。個々のインスタンスに所属するからです。

重要 **static** を付けずに宣言されたインスタンスメソッドは、概念的には、個々のインスタンスごとに作られて、そのインスタンスに所属する。

インスタンスメソッドの中では、*adachi.name* や *nakata.name* ではなく、単なる *name* によって、自分の所属するインスタンスの口座名義フィールドにアクセスします（コンストラクタと同じです）。

また、メソッドはクラス *Account* にとって《内輪》の存在ですから、非公開フィールドにもアクセスできます（この点もコンストラクタと同じです）。

▶ インスタンスメソッドと区別するために、**static** 付きで宣言されたメソッドを**クラスメソッド**（*class method*）と呼びます。前章で学習したメソッドは、クラスメソッドだったわけです。クラスメソッドとインスタンスメソッドの相違点などは、第 10 章で学習します。

インスタンスメソッドを呼び出す例を以下に示します。

```
adachi.getBalance()          // 足立君の預金残高を調べる
adachi.withdraw(200)         // 足立君の口座から200円おろす
nakata.deposit(100)          // 仲田君の口座に100円預ける
```

フィールドをアクセスする場合と同様に、メンバアクセス演算子 . を利用します。

Fig.8-11 に示すのは、足立君の預金残高を調べて表示する様子です。呼び出されたメソッド *getBalance* は、フィールド *balance* の値をそのまま返却します。

クラスの外部から直接アクセスできない口座番号や預金残高などのデータも、**メソッドを通じて間接的にアクセスできる**わけです。

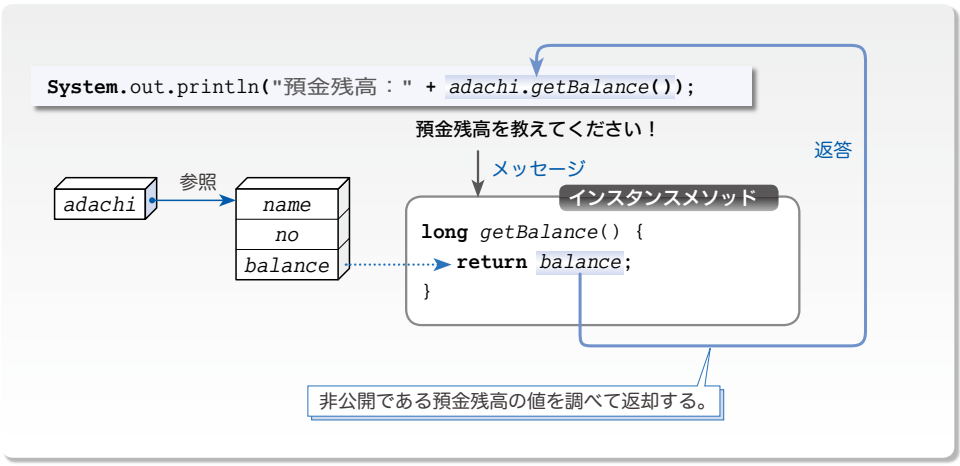


Fig.8-11　インスタンスメソッドの呼出しとメッセージ

なお、コンストラクタはメソッドではありませんから、生成済みのインスタンスに対してメソッドと同様な方法でコンストラクタを呼び出すことは**できません**。

```
adachi.Account("足立幸一", "123456", 5000)      // エラー
```

メソッドとメッセージ

オブジェクト指向プログラミング（*object oriented programming*）の世界では、インスタンスメソッドを呼び出すことを、次のように表現します。

> **オブジェクトに“メッセージを送る”。**

前ページの図に示すように、*adachi.getBalance()* は、オブジェクト（インスタンス）*adachi* に『預金残高を教えてください！』というメッセージを送っているわけです。

その結果、*adachi* は『預金残高を返却してあげればいいのだな。』と**能動的に意志決定を行って**、『預金残高は、○○円ですよ。』と返答処理を行います。

クラスとオブジェクト

一般に、メソッドは、フィールドの値をもとに処理を行って、必要に応じてフィールドの値を更新します。メソッドとフィールドは、密接に連係しているわけです。

フィールドを非公開として外部から保護した上で、メソッドとフィールドとをうまく連係させることを**カプセル化**（*encapsulation*）といいます。

▶ 成分を詰めて、それが有効に働くようにカプセル薬を作ること、と考えればいいでしょう。

本書では、カプセル化されたクラスのイメージを **Fig.8-12** のように表します。

クラスは「回路」の《設計図》に相当します（図ⓐ）。そして、その設計図に基づいて作られた実体としての《回路》が、クラスのインスタンスであるオブジェクトです（図ⓑ）。

そのインスタンスを参照するクラス型変数は、回路を操るための《リモコン》です。

▶ *adachi* リモコン（クラス型変数）は *adachi* の回路（インスタンス）を操作し、*nakata* リモコンは *nakata* の回路を操作します。リモコン（クラス型変数）がもっているのは、メソッドではなくて、メソッドを呼び出すためのボタンです。

インスタンスの回路を起動するとともに、受け取った口座名義・口座番号・預金残高を各フィールドにセットするのがコンストラクタです。コンストラクタは、《電源ボタン》によって呼び出されるチップ（小型の回路）と考えられますね。

そして、フィールド（インスタンス変数）の値は、その**回路（インスタンス）の現在の状態**を表すことになります。そのため、フィールドは、**ステート**（*state*）とも呼ばれます。

▶ state は『状態』という意味です。

Column 8-1　クラスと同一名のメソッド

文法上、クラス名と同一名のメソッドを定義できることになっています。右に示すように、返却型が指定されていなければコンストラクタとみなされ、返却型が指定されていればメソッドとみなされます。

```
class Abc {
     Abc() { /*コンストラクタ*/ }
  int Abc() { /*メソッド*/ }
}
```

ただし、コンストラクタと見分けが付きにくくなるため、このようなメソッドを定義することは推奨されません。

一方、メソッドは回路の**振舞い**（*behavior*）を表します。各メソッドは、回路の現在のステート（状態）を調べたり、変更するためのチップです。

そして、各チップ（メソッド）を間接的に操るのが、リモコン上のボタンです。

▶ リモコンを操る側では、**private** な預金残高の値（状態）を直接見ることはできません。その代わり、*getBalance* ボタンを押すことによって調べられるようになっています。

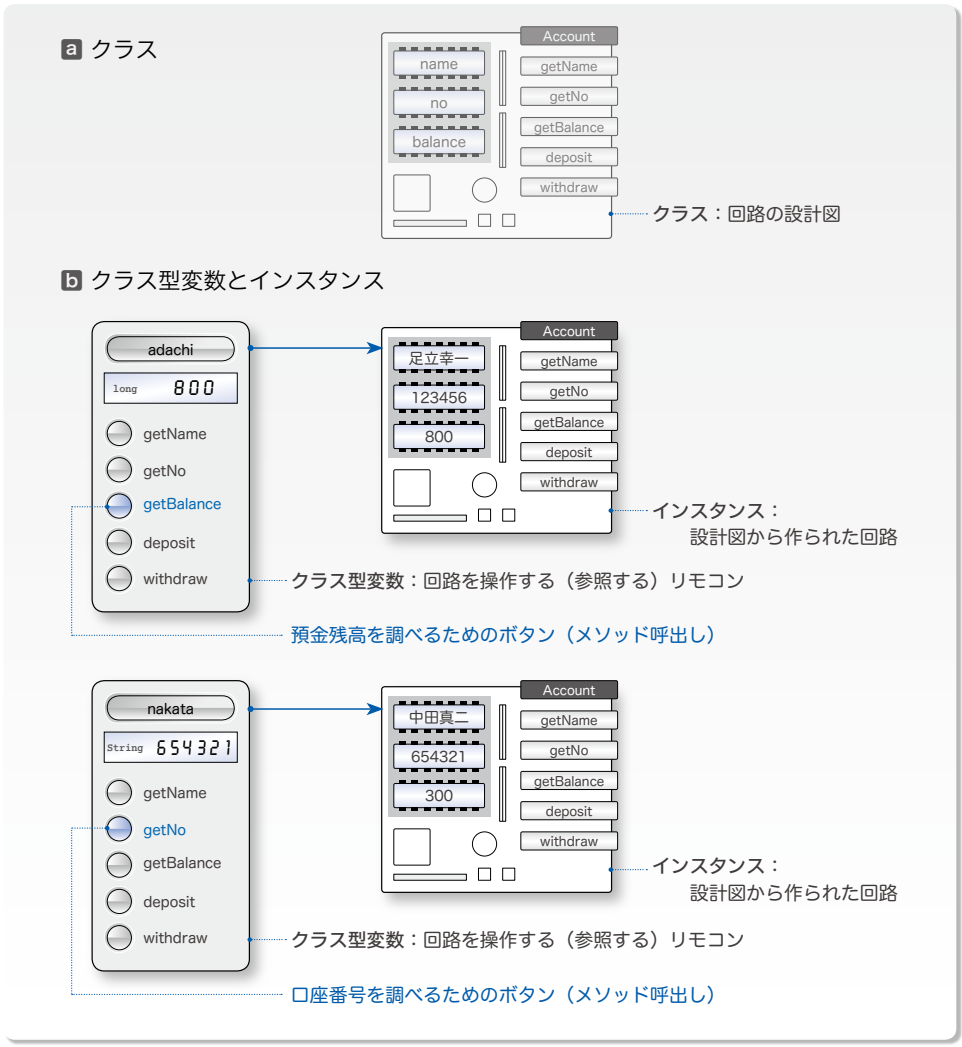


Fig.8-12 クラスとインスタンスとクラス型変数

前章までのプログラムは、実質的には**メソッドの集合**であり、クラスは、メソッドを包むだけの存在でした。本来の Java プログラムは、**クラスの集合**です。クラスを優れたものとすれば、Java がもつ強大なパワーを発揮できます。

▶ 文法上は、Java のプログラムは、クラスの集合ではなく、パッケージの集合ということになっています。パッケージについては第 11 章で学習します。