クラス（教科書第 8 章、第 9 章 p.231～p.291）

前回は処理をまとめる方法として、メソッドについて学習した。今回はメソッドとその処理の対象となるデータをまとめるためのクラスについて学習する。このクラスはオブジェクト指向プログラミングを実現するための最も重要で基本的な技術であり、メソッドより一回り大きなプログラムの部品を構成する。

今回はクラスにおけるデータの扱いとクラスの作成方法、使用方法について説明していく。

データの扱い

以下のプログラムは 2 名の銀行口座のデータを変数で定義し、初期化した値を出力している簡単なプログラムである（教科書 p.232、List8-1）。

プログラム例 1（ソースファイル名：Accounts.java）

```
public 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";   // 足立君の口座番号
                long   adachiAccountBalance = 1000;       // 足立君の預金残高

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

                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
```

このプログラムでは口座名義、口座番号、預金残高の３つのデータについて 2 名分の変数を定義して利用している。例えば、「足立」用ならば、adachiAccountName、adachiAccountNo、adachiAccountBalance というように変数名に adachi を付けて宣言して区別している。しかし、これらのデータはひとまとまりとして扱った方がすっきりするし、便利である。Java ではこういっ

た場合クラスを用いて、以下のように書くことが出来る（教科書 p.234、List8-2)。

プログラム例２（ソースファイル名：account/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】
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
public 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";      // 足立君の口座番号
                adachi.balance = 1000;          // 足立君の預金残高

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

                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);
        }
}
```

実行結果

同じ

このプログラムでは今までと違い、クラスが二つ存在している。それぞれのクラスは以下の通りである。

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

ソースファイルはフォルダ「account」を作成し、ファイル名を「AccountTester.java」として

保存すること。これは main メソッドが含まれるクラス名をファイル名.java として利用してきた今までの使い方と変わらない。プログラムをコンパイルすると、クラス毎にクラスが作成されるため、以下のようなファイル構成になる。

```
account
   |-AccountTester.java
   |-Account.class
   |-AccountTester.class
```

クラス宣言

プログラム例２において、銀行口座クラスを宣言した部分を抜き出すと以下になる。

```
// 銀行口座クラス
class Account
{
        String name;      // 口座名義
        String no;        // 口座番号
        long   balance;   // 預金残高
}
```

クラスを構成するデータは「**フィールド**」と呼ばれ、銀行口座クラスは３つフィールドから構成されている。クラス宣言は「型」を宣言するため、「変数」を用意するためには以下のようにして行う。

```
Account adachi = new Account(); // 足立君の口座
Account nakata = new Account(); // 仲田君の口座
```

Account 型によって宣言された adachi や nakata は銀行口座クラスを参照するクラス型変数であるため、実体そのものは new で生成して代入する。この使い方は配列の生成、乱数の生成（Random）やキーボードからの入力（Scanner）の時に利用した「new クラス名()」という形式と同様である。

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**」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。

また、配列の本体やクラスのインスタンスはプログラム上、new によって実体が動的に生成される。プログラム実行時に動的に生成される実体を総称して「**オブジェクト**」と呼ぶ。

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３つのフィールドをもっている。これらの個別のデータにアクセスするには「メンバアクセス演算子」(.) を用いる。これは「フィールドアクセス演算子」や「ドット演算子」とも呼ばれる。プログラム内での使用方法は以下のようになる。

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

フィールドはインスタンス内の変数であるので「**インスタンス変数**」とも呼ばれる。

フィールドの初期化

配列を学んだ時にも述べたが、配列の構成要素は生成時に「**既定値**」によって初期化される。同様にして、フィールドも既定値によって初期化が行われている。プログラム例２において、値の代入部分を削除した場合には以下のような出力結果となる。

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
```

銀行口座クラス第２版

銀行口座クラス第１版ではクラスを導入し、変数をまとめた。しかし以下の問題点が残っている。

・データの保護に対する無保証

各銀行口座クラスのデータは他のプログラム（クラス）から自由に変更することが可能となっている。これらのデータに対する変更はプログラム上制限されているほうが良い。これについては「**データ隠蔽**」という仕組みが Java には用意されている。

・初期化に対する無保証

フィールドは既定値で暗示的に初期化される。その後、値の設定を行うかどうかをプログラマに委ねている。値を設定し忘れたプログラムは予期しない動作を起こす可能性があるため、とても危険である。初期化する必要があるフィールドは強制的に行う状態にすべきである。この問題を解決するために Java ではこの場合「**コンストラクタ**」というものが用意されている。

これらの仕組みを反映させた銀行口座クラス【第２版】のプログラムを以下に示す(教科書 p.241、List8-3)。

プログラム例３（ソースファイル名：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
{
        // フィールド
        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
        }

        // メソッド
        // 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;
        }
}

// 銀行口座クラスをテストするクラス
public class AccountTester
{
        public static void main(String[] args)
        {
                Account adachi = new Account("足立幸一", "123456", 1000);
                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

同じ

このプログラムでは Account クラスがフィールドとコンストラクタとメソッドの３つによって構成されている。また、プログラム例２と比べると、フィールドの宣言には以下のように private

という記述が追加されている。

```
private String name;    // 口座名義
private String no;      // 口座番号
private long   balance; // 預金残高
```

この記述を追加したフィールドは「**非公開アクセス**」となり、クラスの外部に対して存在を隠す（データ隠蔽）。これはどういうことかというと、例えば以下のような記述が main メソッド側にあると、コンパイルエラーとなる。

```
// main メソッド内にあるとコンパイルエラーになる処理
adachi.name = "仲田真二";        // 口座名義の変更
adachi.no = "999999";            // 口座番号の変更
System.out.println(adachi.balance); // 預金残高の表示
```

つまり、main メソッド内では adachi.name などのフィールドは直接利用することができず、これらのフィールドへのアクセスは adachi.getName()などのように、メソッドを利用して行うことになる。アクセス手段（値を利用するだけなのか、フィールドへの値の代入を許可するのか）は用意するメソッドによって確保する。

フィールドを非公開にしてデータ隠蔽を行うとデータの保護性やプログラムの保守性が向上することが期待され、基本的には全てフィールドは private で宣言することが望ましい。private を省略してしまうと、フィールドは「**デフォルトアクセス**」となる。これはフィールドが公開されている状態である。

注：デフォルトアクセスは、正確にはパッケージ内で公開となり、パッケージ外部では非公開となることから、パッケージアクセスとも呼ばれる。パッケージについては教科書第 11 章に示されている。パッケージとは簡単に言うと、関連するクラスをまとめたものである。

プログラム例３からコンストラクタの部分を抜き出すと以下のようになっている。

```
// 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
}
```

コンストラクタはメソッドに似ているが、その特徴はクラス名と同じであり、返却値を持つことができない。コンストラクタは以下の下線部のインスタンス生成時に呼び出される。

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

コンストラクタは以下のように引数が一致しない場合にはエラーとなる。

```
// コンパイルエラーになる処理
new Account();
new Account("足立幸一")
```

この仕組みによって、インスタンスの初期化をクラスの利用者に強制することで適切な初期化を

実現している。
　次に Account クラスに宣言したメソッドについて述べていく。メソッドは以下のような形で宣言する。

```
// 口座名義を調べる
String getName()
{
        return name;
}
 :
 :
// k 円預ける
void deposit(long k)
{
        balance += k;
}
 :
```

　private で宣言したフィールドへアクセスするメソッドには static を付けない。static を付けるとコンパイルエラーとなる。このプログラムの場合、それぞれのクラス用のインスタンスが生成される。つまり、adachi、nakata それぞれのフィールドにアクセスするためのメソッド（getName()の場合は adachi.getName()と nakata.getName()が用意される）を持つことになる。static を付けないメソッドは個々のインスタンスに所属するため、「**インスタンスメソッド**」と呼ばれる。
　メソッドはクラス Account 内で宣言しているため、非公開フィールドへのアクセスが許可される。クラスの外部（main メソッドなど）から直接アクセスできない private で宣言した非公開フィールドはクラスメソッドを利用して間接的にアクセスすることができる。これはコンストラクタも同様である。
　このように非公開フィールドによって外部からデータを保護した上で、メソッドとフィールドを連携させて利用することを「**カプセル化**」という。この場合フィールドはインスタンスの状態を表すことになるので、「**ステート**」とも呼ばれる。
　オブジェクト指向プログラミングではインスタンスメソッドを呼び出すことを「オブジェクトにメッセージを送る」と表現する。プログラム例３の場合、例えば adachi.getName()を呼び出すことによって、オブジェクト（インスタンス）である adachi に口座名を教えて欲しいとメッセージを出し、結果オブジェクト adachi は返答処理として、口座名を返却するという流れになる。

演習と課題 6

演習１　プログラム例 1～3 を作成して、出力を確認しなさい。

演習２　教科書 p.250～の自動車クラスについて教科書を読み、List8-4～6 を作成しなさい。この場合、それぞれのファイルは別のファイルに記述する必要がある。同じフォルダに 3 つのファイルを作成し、List8-4 については最初にコンパイルして.class ファイルを生成しておく必要がある。ファイルについては作成の他、著者のダウンロードサイトから得たファイルを使用してもよい。

　List8-4　自動車クラス【第１版】
　List8-5　自動車クラス【第１版】の利用例（その１）
　List8-6　自動車クラス【第１版】の利用例（その２）

演習３　名前・身長・体重などメンバとして持つ≪人間クラス≫を作成しなさい（フィールドやメ

ソッドなどは自分で自由に設計すること）

kadai6_1　演習 2 で作成した自動車クラス Car に自由にフィールドやメソッドを追加しなさい。

演習 4　教科書 p.261～の日付クラスについて教科書を読み、List9-10 を作成しなさい。

List9-10　日付クラス【第 3 版】

さらに、日付クラス【第 3 版】を利用するプログラムを作成しなさい。すべてのコンストラクタを動作させること。

演習 5　教科書 p.324～のクラス型のフィールドについて読み、List9-14 を作成しなさい。動作には List9-11 も必要となる。

List9-11　自動車クラス【第 2 版】
List9-14　自動車クラス【第 2 版】の利用例（その 3）

kadai6_2　日付クラス【第 3 版】を利用しつつ、銀行口座クラス【第 2 版】（テキストのプログラム例 3）に口座開設日のフィールドと toString メソッドを追加しなさい。また、コンストラクタを変更したり、口座開設日のゲッタ（口座開設日フィールドが参照する日付インスタンスのコピーを返す）などのメソッドを追加したりすること。また、toString メソッドについては教科書 pp.278-279 を読むこと。

kadai6_3　演習 3 で作成した≪人間クラス≫に、誕生日のフィールドと toString メソッドを追加しなさい。また、コンストラクタを変更したり、誕生日のゲッタなどのメソッドを追加したりすること。この課題でも日付クラス【第 3 版】を利用すること。

kadai6_4　ここまで習ったことを十分に活用し、クラスおよびそのクラスを利用する main メソッドを含むクラスを持つオリジナルのプログラムを作成する。