7 JAVA の本質に触れる ～クラス～

7.1 クラスとは？

7.1.1 本章の説明

さて、ここからご紹介するのは、**クラス**と呼ばれる、ある意味 JAVA の本質とも言える機能です。ちょっと勉強熱心な方なら、**オブジェクト指向**という言葉を置き気になったことがあるかもしれません。これは、プログラムをする上での考え方の一つで、JAVA、C++、C#、Ruby といったプログラム言語がオブジェクト指向プログラミングができるようになっています。モチロン、オブジェクト指向プログラミングをしなくたってかまいませんが。

そのオブジェクト指向型プログラミングで基本となるのが、このクラスです。本章では、オブジェクト指向の真髄とも言うべきクラスの作り方や、使い方について学んで生きたいと思います。

なお、クラスについては一章で語りつくすのはかなり難しいので、2 章にわたってクラスについて語りまくります。

本章と，次章ではクラスについて説明する上で，携帯電話のパケット通信記録クラスを作成して行きたいと思います．携帯電話でパケット通信を行うと，それを記録してその量に応じてパケット通信料を計算してくれるクラスです．

7.1.2 クラスとは

これまで，変数として整数型や実数型などを使ってきました．しかしながら，世の中数字だけで表せるものばかりではありません．

ここで，携帯電話会社に就職したあなたが，Java で携帯電話用のアドレス帳を作る仕事を任されたとしましょう．このとき，アドレス帳は数字だけで作れるでしょうか？ちょっと難しいですよね．

アドレス帳には，一人につき

- 名前
- 電話番号
- 住所

などの情報を付加しておかなければいけません．もちろん，文字列変数 String も知っていますからこれまでに勉強したものだけでもアドレス帳を作ることは不可能ではありません．こんな感じで作ることも可能でしょう．

```
String name;
String address;
String phoneNumber;
```

しかしながら，これでは一人分のデータしか扱えません．たくさんの人のデータを扱う

には，配列でも使わないと駄目でしょう．

というわけで，10 人分のデータを保存できるアドレス帳を作るには，

```
String[] name = new String[10];
String[] address = new String[10];
String[] phoneNumber = new String[10];
```

とすれば良いかも知れません．しかし，これではデータが名前なら名前ごとに，住所なら住所ごとにまとめられてしまっています．普通我々はアドレス帳をこのようには作りません．たいていの場合，個人一人一人についてデータをまとめる物です．

となると，こんな形でデータを作る方法が考えられます．

```
String[] personalData = new String[3];
String name = personalData[0];
```

これで，personalData という配列に，name から emailAddress までの文字列データを全部一つにまとめることが出来ます．

ただし，これだと一人分しかデータを扱うことが出来なくなってしまいます．一人分しかデータがないアドレス帳なんて寂しすぎます．どれだけ友達がいないんだ．

というわけで，以前配列で勉強した多重配列も使わないと駄目層です．じゃあ，こうすれば問題解決．

```
String[][] personalDataArray = new String[10][3];
```

うん，これなら 10 人分の文字列データを 4 つ確保できますので，名前から住所までの 3 つのデータをそれぞれの配列の要素としたデータを 10 人分確保できました．一人目のデータを使いたい場合は，

```
String[] personalData = personalDataArray[0];
String name = personalData[0];
```

とすればいいでしょう．

ところで，ここでこのアドレス帳を使って，何回電話したかを記録しておきたいとしましょう．これで，電話した人上位から順番にアドレス帳を表示したりすることが出来るようになります．

さて，そんなときに，どうやって会った回数を記録しましょうか．簡単に考えると，アドレス帳の個人データを一つ増やせばいいような気がします．

```
String[][] personalDataArray = new String[10][4];
```

これで，5 つめに会った回数を記録･･･と行きたいところですが，この personalDataArray は文字列しか保存できません．ということは，電話した回数という整数型は保存できないのです．もちろん，

```
personalData[3] = "5";
```

などとして，5 回電話したことを記録することは出来ますが，さらにもう一回あったときに 5 に 1 を足して 6 にするということができません．

```
personalData[3]++; //できない
```

ということは，この形式では我々は合った回数を管理することが出来ないわけです．しょうがないので，

```
int[] callCount = new int[10];
```

結局こんな配列を作らなければいけません．これでは結局最初の方法と同じになってしまい，結局あった回数だけ別で管理する必要が出てきてしまいました．こんなアドレス帳は中途半端すぎです．

さらに，アドレス帳のどこにどのデータが入っていたか分からなくなる可能性というのもありますよね．最初のデータに名前が．次のデータに住所が・・・と一応決まってはいますが，どのデータが入っているのかをいちいち思い出さなければ使えません．通常の手帳ならば，「住所」などの表題が書いてありますが，配列の場合どこにどのデータという名前をつけることが出来ないのです．最初の方法ならば name 配列には名前が，address 配列には住所が入っているな，と一目で分かるのに．

このように，配列だけを使ってデータ管理をしようとすると色々無理が出てきてしまいます．
これは困った．

そんな困った皆さんにクラスがお勧めになります．（やっと話をクラスに持っていけた・・・）
クラスとは、現実社会にある「モノ」をプログラム上で表現するためのものです。この「モノ」のことを「オブジェクト」と呼びます．たとえば，アドレス帳の個人データはオブジェクトですし，アドレス帳そのものもオブジェクトということができます．これらの「モノ」は，これまで学んできた変数などで表現することはできませんよね．
このような，これまでにない独自の「型」を Java ではクラスと呼び，自作することができるのです．

例えば，アドレス帳の個人データであれば，

```
PersonalData personalData;
```

という変数を作成することができるのです．
このように，Java にもともとある型=プリミティブ型以外の型を自作したとき，その型のことを**クラス**と呼びます．
この，PersonalData 型がどのような型なのかを記述するのが，クラスの記述となります．本章では，クラスの記述方法とその利用方法について解説して行きたいと思います．

7.2 クラスの作成とインスタンス

7.2.1 最初のクラス

今回作成するクラスはアドレス帳に載せる一人分のデータです.

まずは，暫定完成形をご覧ください.

```
public class PersonalData {
	String name;
	String address;
	String phoneNumber;
	int callCount;

	void call(){
		System.out.println(name+"に電話しました");
		callCount++;
	}
}
```

ずいぶん簡単なものですが，これで完成です.

このファイルを「PersonalData.java」という名前で保存します．というか，この辺はこれまで作ってきたプログラムの main 行がないものと考えれば難しいことはありません.

詳しい説明はこれからしますが，とりあえずはこういうものを作成してみてください.

ちなみに，これはクラスの中でも非常に機能を限定して作ったものに過ぎません．これからどんどん肉付けしていきますので，お楽しみに.

7.2.2 インスタンスの作成

クラスというのは変数型の一種のように扱われます。たとえば、整数型の変数は

```
int i;
i=0;
```

というように変数を宣言、その後値を代入して始めて利用可能となります。

クラスもこれとまったく同様に、宣言して代入することで使うことが可能となります。作成したクラスの変数のことを**インスタンス**と呼びます。

いうなれば、クラスというのは何かの設計図であり、その設計図を元に実際にものを作ったとき、そのもののことがインスタンスなのです。ここでは，アドレス帳の個人データはこういう形式で作成するよ，と決めたのが先ほどのクラスの記述です.

で，設計図を，つまり個人データの形式を基に実際に個人データを書き込んだものがあります．その個人データそのものが**インスタンス**なのです.

設計図、すなわちクラスの書き方については後で説明するとして、すでにクラスについては書いてあるものとして先ほどの PersonalData クラスの変数を作成してみます。

```
	static public void main(String[] args){
		PersonalData pd;
		pd = new PersonalData();
	}
```

```
```

なんだかあたり一面 PersonalData だらけでわけが分からなくなってきましたが，基本は整数型などのプリミティブ型変数を作るときと同じです．

まずは、

```
PersonalData pd;
```

で pd という変数が，PersonalData 型だよ，ということを宣言しています．そして，問題は次の部分です．

```
pd = new PersonalData();
```

これは，**新たに PersonalData を作成して，pd に代入する**という意味です．

整数型の場合は，値を代入するときに，直接

```
int a;
a = 10;
```

などのように直接数字を書き込めましたが，PersonalData 型だとそうはいきませんよね．だって，PersonalData をどうやって数値や文字で表現すればよいのでしょう？無理ですよね．

というわけで，新しい PersonalData を作る作業が必要になるのです．

それが、new という宣言です。これは、新しい PersonalData を作成するよ、ということを Java に知らせます。これによって、Java は PersonalData クラスを設計図として、まったく新しい PersonalData を作成して、それを pd へ代入する、つまり pd がその PersonalData を指すことになります。

new によって作成された PersonlData そのものをオブジェクト、またはインスタンスと呼びます。そして、それをさす pd をオブジェクト変数と呼びます。

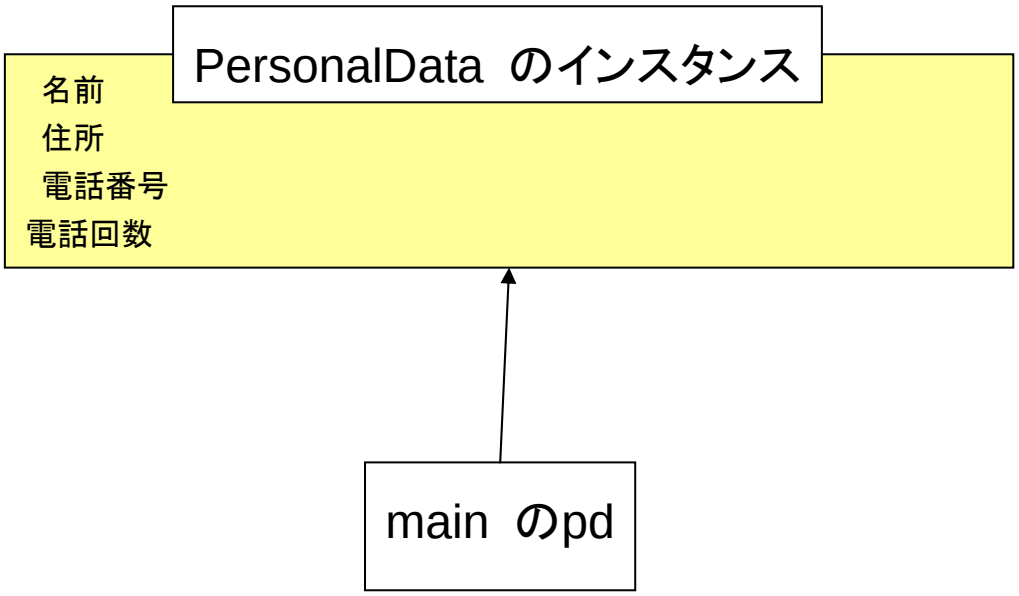


図　インスタンスの作成

ポイントは、オブジェクト変数はオブジェクトを指しているだけで、オブジェクトそのものではないという点です。これを常に意識しておいてください。

さて、設計図から個人データを実際に作るためのキーワードが **new** です。

```
new クラス名(引数);
```

と書くことで、新しいオブジェクトが作成されます。

ちなみに、クラス名の後ろに()を書いて、まるでメソッドを呼び出すときのようですよね。実際そのとおりで、new で新しいオブジェクトを作る場合は、**コンストラクタ**という初期化専用のメソッドを呼び出しているのです。詳しくはのちほどコンストラクタで説明しますが、インスタンスを作成するときは「どんな中身を持っているのか」を指定して作成することが多くなります。たとえば、通常の場合、アドレス帳に個人データを入力するのであれば名前を基準にデータを作ることになるでしょう。

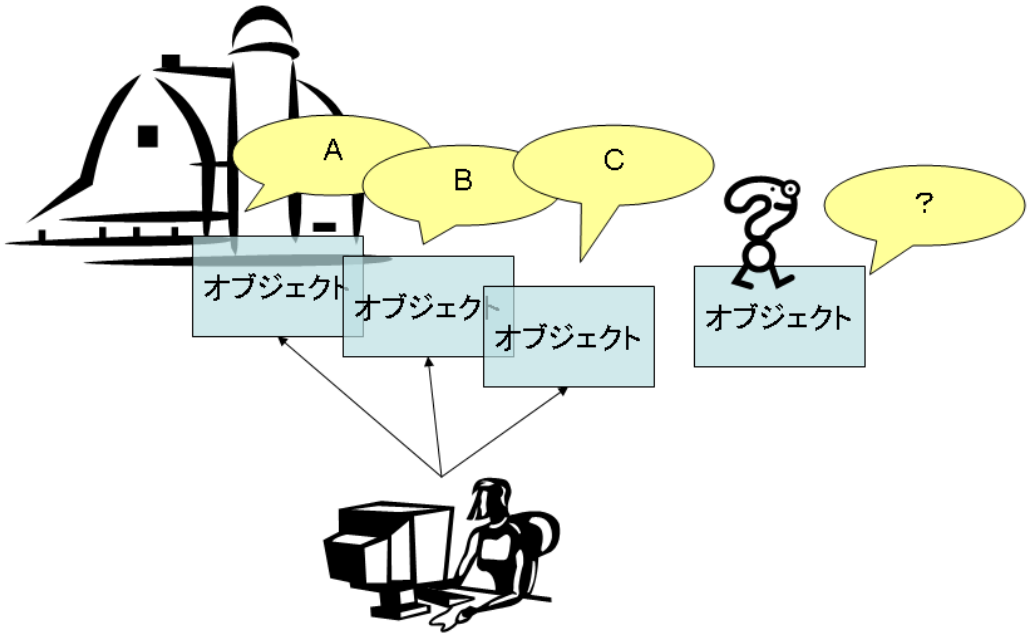
まあ、それは後で説明するとして、

```
new PersonalData();
```

と書くことでクラス設計図に基づいて PersonalaData インスタンスを作成します。

なお、インスタンスを作成したら、すぐに変数に代入しないと二度とそのインスタンスを利用することができなくなってしまいます。インスタンスができても、そのインスタンスを管理している人がいなければ、紛失してしまいます

ちゃんと名前をつけて管理してあげなければ、どのオブジェクトがいつ誰が何の目的で作ったものだかわからなくなってしまいますからね。



図：放っておくと紛失

というわけで、新しく作られた PersonalData オブジェクトを pd という名前で保存しておくのです。

```
pd = new PersonalData()
```

7.2.3 インスタンスの利用

さて、インスタンスは作っただけでは意味がないので利用してみましょうか。アドレス帳の個人データクラスでは、名前や住所といった変数を持っています。これらの変数のことを、**フィールド**と呼びます。なんで変数じゃ駄目なんだ？と思うかもしれませんが、その理由はおいおい分かってきます。あるクラスが特有に持っている変数のことをフィールドと呼ぶと覚えておいてください。

さて、名前や住所といったフィールドを使うには、以下のようにします。

```
PersonalData pd = new PersonalData();

pd.name = "マリオ";
System.out.println("名前は"+pd.name);
```

あんまり難しいことはありません。フィールドを利用する場合、

```
インスタンス名.フィールド名
```

と書けば利用可能です。

ポイントは、「.」をつけるという点だけです。

さ、さてこれによって、「pd」という個人データの name フィールドを利用することができるようになりました。どうように、住所や電話番号も値を変更したり、利用したりすることができます。

また、クラスにはそのクラス特有のメソッドを作成することができます。個人データクラスの場合は、

```
        void call(){
                System.out.println(name+"に電話しました");
                callCount++;
        }
```

この部分がメソッドになります。前章で学んだメソッドとはちょっと書き方が違う部分もありますが、基本的には一緒です。ちなみに、このメソッドは、電話することを意味するメソッドですよ。

さて、このメソッドを呼びたい場合はどうすればいいのかというと、先ほどと同じように、利用するインスタンス名を使って、

```
                pd.call();
```

とします。これによって、pd のメソッドが呼び出されるのです。pd のメソッドが呼び出されるというのはちょっと意味が分かりづらいかもしれませんが今は気にしないでください。

とにかく、あるクラス特有のメソッドを呼び出す場合は、

```
インスタンス名.メソッド名();
```

とします。もちろん、メソッドに引数を与える場合は()内に引数を書けば Ok です。

では、最後にこれら二つの操作をまとめて、PersonalData を使ったプログラムを書いてみると・・

```
                PersonalData pd = new PersonalData();

                pd.name = "マリオ";
                System.out.println("名前は"+pd.name);

                pd.call();
                System.out.println(pd.name+"への電話回数は"+pd.callCount+"回");
```

こんな感じになり、その実行結果は、

```
名前はマリオ
マリオ太郎に電話しました
マリオへの電話回数は１回
```

となります。これによって、マリオ君には過去一回だけ電話をしたことがあることが分かります。

7.2.4　　複数のインスタンス

ところで、インスタンスのメソッドを使う場合は、必ずインスタンス名を書いてからメソッドを書かなければいけません。なぜこんな面倒な書き方をしなければいけないのでし

ょうか？

じつは、メソッドの動作結果というのはインスタンスによって異なるからなのです。そのため、「どのインスタンスのメソッドを使うか」、ということを明示的に示しておかないと、プログラムはどのメソッドを使えばいいのかわからなくなってしまうんですね。

ここで例を挙げてみましょう。

さて、先ほどマリオ君に一回電話をしてそれを確認しました。今回、それとは別の人のデータアドレス帳に入力して、3 回くらい電話をかけてみたいと思います。どうなるでしょうか？

```
PersonalData pd = new PersonalData();
pd.name = "マリオ";
pd.call();

PersonalData pd2 = new PersonalData();
pd2.name="ピーチ";
pd2.call();
pd2.call();
pd2.call();

System.out.println(pd.name+"への電話回数は"+pd.callCount+"回");
System.out.println(pd2.name+"への電話回数は"+pd2.callCount+"回");
```

このプログラムでは、マリオ君だけではなく、ピーチさん（なんて読むのか考えてはいけません）のデータを入れておくために、pd2 という PersonlData 型のインスタンスを作成して、pd2 に対して操作を行っています。

ポイントは、pd2 に対して 3 回 call メソッドを読んでいる、つまりピーチさんにだけ 3 回もメールを書いている点に気をつけてください。

さあ、この結果はどうなったでしょうか？

```
マリオに電話しました
ピーチに電話しました
ピーチに電話しました
ピーチに電話しました
マリオへの電話回数は 1 回
ピーチへの電話回数は 3 回
```

ほら、ちゃんとマリオ君とピーチさんへの電話回数を別々にカウントしていることが分かりますよね。

このように、インスタンスは同じ設計図から複数作成することができ、それぞれを独立して扱うことができます。

さて、これでなぜインスタンスのメソッドを使う場合に、

```
インスタンス名.メソッド(引数)
```

という形式にしなければいけなかったか、わかりましたね。今後は何度ピーチさんに電

話をしてもマリオ君にそれがばれる心配はありません。ストーカーし放題。（注：ストーカー行為は犯罪です）

このように、インスタンスごとにメソッドの実行結果は変わってくるのですから、「どの」インスタンスのメソッドを使うのかが重要な意味を持ってくるからなのです。

というわけで、メソッドの使い方、ぜひ覚えておいてください。

7.3 クラスの基本構文

7.3.1 クラスの文法

クラスをどう使うかはわかったかと思いますが、次はクラスを作るほうについての説明です。

実は、Java でプログラミングするということは、「**クラスを作っていくこと**」に他なりません。一見大げさな言いように見えますが、実は本当にこの言葉通りなのです。Java におけるプログラミングはすべてクラスを作ることに集約されています。これまでに作ってきたプログラムも実はクラスを利用しているのです。まあ、察しのよい方はすでに分かってきていると思いますが。これについては、次章で詳しく説明したいと思います。

本章では、クラスをどうつくるのか、その構文について説明します

クラスとは、設計図だというお話をしましたが、設計図にも書き方が決められています。決まりを守っていない書き方ではコンピュータはそれがクラスであることも、どういう設計図であるのかも理解できません。そこで、ちゃんと設計図であるクラスの文法を学んでいきましょう。

まずは一般的なクラスの文法を示します。

```
class クラス名{
        //フィールド
        フィールド型　フィールド名;
        フィールド型　フィールド名;
        ・・・


        //メソッド
        public 返値型　メソッド名(引数){
                処理;
                return 返り値;
        }

        //コンストラクタ
        public クラス名(引数){
                初期化処理;
        }

}
```

クラスはクラス宣言と、

- フィールド
- メソッド
- コンストラクタ

から成り立ちます。それぞれ、「データ」「操作」「初期化」に対応します。

先ほど例にあげた個人データクラスであれば、

フィールド：「名前」「住所」「電話番号」「Email アドレス」「電話回数」

メソッド：電話をかける

コンストラクタ：データ初期化

となります。

では、それぞれ詳しく見ていきましょう。

7.3.2 クラス宣言

まず、新しいクラスを作成しようと思ったときの宣言のやり方です。

```
class クラス名{
```

まずこの行は、クラス名を宣言しています。簡単に言うと「これから説明するクラスは*クラス名*という名前です」と宣言しているわけです。ちなみに、クラス名には

- 同一パッケージには同じ名前のクラスは作れない
- クラスを宣言するファイルは基本的に「クラス名＋.java」という名前でなければいけない
- 基本的に一クラス一ファイル

という制限があります。難しいことはあとで説明しますが、クラスについての記述は、

***クラス名*.java**

というファイルに書くものと思っていてください。先ほど例に挙げた PersonalData クラスであれば、PersonalData.java というファイルを作成して、その中に先ほどのソースコードを書くことになります。

7.3.3 フィールド

クラスとは、データとその操作を扱うものですが、フィールドはそのうちのデータをあらわすものです。簡単にいってしまえば、フィールドとはこれまでに扱った変数のことです。クラスで利用する変数のことをフィールドと呼ぶのです。正確に言うと「インスタンス変数のことをフィールドという」のですが、まあ、難しい話になるので後々説明していきたいと思います。

これまでに扱ってきた変数はひとつのメソッドの中でしか有効ではありませんでした。しかしながら、フィールドとはひとつのクラスのインスタンス全体で有効な変数なのです。つまり、同じ変数を異なるメソッド間で共有して使うことができるわけでです。というこ

とは、いちいち変数を引数として渡したり、返り値として受け取らなくても、変数を変更したりできるわけです。これはかなり便利な機能です。ひとことでまとめれば、**フィールドはクラス全体で使える変数である**ということになりますね。

個人データクラスの場合は、

```
String name;
String address;
String phoneNumber;
int callCount;
```

整数型の「名前」「住所」「電話番号」「Email アドレス」をあらわす文字列型のフィールドと、「電話回数」を表す整数型のフィールドがクラスのどこでも使うことができます。

なお、これらの変数を具体的に使うのはメソッドの中になります。では、その説明に移りましょう。

7.3.4 メソッド

クラスには第 6 章で勉強したメソッドを付け加えることができます。これはデータの操作を行うためにおもに使われます。

先ほどインスタンスのところで説明しましたが、メソッドはクラス外からインスタンスを通して利用することができます。PersonalData クラスでは call メソッドを通して電話をかけることを表現していました。

```
void call(){
        System.out.println(name+"に電話しました");
        callCount++;
}
```

実際に電話はかけられないので、やっていることは電話をかけたことを表示して、電話回数を一回増やしているだけですが・・・

格クラスのメソッド内では、クラスにあるフィールドを自由に使うことができます。たとえば、ここでは name と callCount を利用していますが、address、phoneNumber や emailAddress も利用することが可能です。しかも、通常使う場合は

```
変数名.フィールド名
```

という書き方をしていましたが、メソッド内では変数名を省略することが可能です。これは、フィールドは各クラスのインスタンスに結びついて作られているため、変数名をつけなくても、どのインスタンスのフィールドだか分かるためです。

というより、インスタンス自身は自分につけられている変数名は分からないので、変数名はつけようがないんですけどね。

なお、このメソッドを利用する場合はすでに学んだとおり、

```
pd.call();
```

のようにします。

ただし、同じインスタンスのメソッド内からほかのメソッドを呼び出す場合には、フィールドを利用するときと同様に変数名をつける必要はありません。

最後に、メソッドの文法は、以下のとおりです。

```
public 返値型　メソッド名(引数){
        処理;
        return 返り値;
}
```

メソッドを作成する場合は、返値型、メソッド名、引数、処理、および返値を記述しておきます。基本的に前章で勉強したメソッドの書き方と同じですね。

しかし、一点だけ違う点があります。それが、static です。以前学んだメソッドでは、

```
static int addArray(int[] array){
```

のような書き方をしたことを覚えていますか？

今回紹介したメソッドに比べると、static と書いてある点が異なっていると思います。これについては、後で詳しい説明をしますが、**インスタンスでメソッドを作る場合、static を書く必要はない**、というルールがあります。static にはかなり特殊な意味があるので、それについては後でちゃんと説明しますので、今回は「クラスに作成するメソッドには static はつけない」と覚えておいてください。

7.3.5　メソッド命名の注意点

ところで、クラスのメソッドの名前は、基本的に変数と同じようにどんな名前でも付けることができます。これは前章で説明したとおりです。じつは、これ以外にもメソッド名には制限があります。それは、「**クラス名と同じ名前のメソッドは作ることができない**」という制限です。

たとえば、この PersonalData クラスに置いては、PersonalData という名前のメソッドを作ることはできないのです。クラス名と同じ名前のメソッドは、自動的にコンストラクタであると判断されしまうんですね。ん？コンストラクタってなんだ？と思われたかもしれませんが、それはすぐにご説明しますからちょっとだけ待ってください。簡単に言えば、初期化の時に使われる非常に特殊なメソッドです。

で、もしもどうしても PersonalData という名前をメソッドに付けたい場合は、personalData と最初の文字を小文字にしたりちょっと変化をつけましょう。

ちなみに、メソッドの名前は最初小文字にするのが一般的です。今後自分で新しいメソッドを作りたくなったら、最初の字を小文字にするようにしてください。ぜひ。

7.3.6　コンストラクタ

7.3.6.1　コンストラクタの基本

クラスメイト同じ名前のメソッドはコンストラクタになるからつけられません、という話をしました。

じゃあ、コンストラクタとは何かといえば、初期化のためのメソッドです。

ようするに、コンストラクタはインスタンスを作ったときに最初に呼び出されるメソッ

ドなのです。

そこで、コンストラクタの効果を確認するために、PersonalData クラスに以下のようなものを追加してみましょう。

```
PersonalData(String n){
        name = n;
}
```

これは、渡された引数 n を name フィールドに代入するコンストラクタです。つまり、引き数に渡された文字列が名前となるわけです。

これを使うプログラムを以下のように作りましょう。

```
public static void main(String[] args) {
        PersonalData pd = new PersonalData("マリオ");
        System.out.println("作った個人データの名前は、"+pd.name);
}
```

これまで、new のあとにはクラス名+()しか書いていませんでしたが、今回初めて()内に引数が渡されることになりました。

これを見ると、なんとなくなぜコンストラクタが呼ばれるのか理解できませんか？確かに、PersonalData というメソッドを"マリオ"という引数を使って呼び出していそうですよね。

さて、この実行結果は・・・

```
作った個人データの名前は、マリオ
```

どうでしょうか、ちゃんとコンストラクタが呼ばれて pd の name が変更されていることが分かりますよね。

このように、new によって新しいインスタンスが作られると、それに応じて自動的にコンストラクタが呼ばれるルールとなっています。

では、最後にコンストラクタの書き方についてです。コンストラクタを書く場合のルールは以下の二点です。

- メソッドの名前に対応する個所が、クラス名そのものである
- コンストラクタには返り値はない

コンストラクタはクラス名を名前にして返り値のないメソッドとして定義すると覚えてください。

特に、返り値がない点には注意してください。間違えて返り値を書いてしまうとコンパイルに失敗してしまいますからね。

では、以上コンストラクタの書式をまとめておきましょう。

```
クラス名(引数){
        //処理内容
}
```

基本的には普通のメソッドと書き方は一緒ですので、あんまり混乱はないと思います。

7.3.6.2　　複数コンストラクタ

コンストラクタは何種類か設定することができます。メソッドでオーバーロードという「同じ名前のメソッドを複数作ることができる」というルールがありましたが、それと同じことがコンストラクタでもできるのです。

先ほど作ったコンストラクタは、最初に名前だけを設定していましたが、今回は名前のほかに住所と電話番号と e-mail アドレスを設定するコンストラクタを作りましょう。

```
        public PersonalData(String n, String a, String p){
                name = n;
                address = a;
                phoneNumber = p;
                callCount = 0;
        }
```

これで、名前から電話番号までフィールドをすべて任意の値に初期化することができるようになりました。ちなみに、最後の行では callCount を 0 にしています。このように、引数と無関係なフィールドを初期化することもできますよ。

なお、このコンストラクタを呼ぶ場合は以下のとおりにします。

```
                PersonalData pd2 = new PersonalData("ピーチ", "キノコ王国", "不明");
                System.out.println("作った個人データの名前は、"+pd2.name);
                System.out.println("住所は、"+pd2.address);
                System.out.println("電話番号は、"+pd2.phoneNumber);
```

これで、住所と電話番号とアドレス付の PersonalData ができました。

これを実行すると、

```
作った個人データの名前は、ピーチ
住所は、キノコ王国
電話番号は、不明
```

という、やっぱりあんまり役に立ちそうもないデータが完成しました。

ま、いずれにせよ、一つのクラスに付きコンストラクタはいくつか作成することができます。ただし、まったく同じ引数のコンストラクタは作ることができません。何しろ、コンストラクタの名前はクラス名と同じでなければいけないという制限がありますから、同じ引数のコンストラクタを作成したら、どちらのコンストラクタを呼べばよいか分からなくなってしまいますからね。

ですので、アドレスだけわかった、という場合は、

```
        public PersonalData(String n, String a){
                name = n;
                address = a;
                callCount = 0;
        }
```

というようなコンストラクタを作ればよいかもしれません。ですが、同時に電話番号だけ分かったバージョン、

```
        public PersonalData(String n, String p){
                name = n;
```

```
                phoneNumber = p;
                callCount = 0;
        }
```

は作れません。

```
PersonalData pd2 = new PersonalData("ピーチ", "キノコ王国");
```

と書いたら、電話番号と住所のどちらが「キノコ王国」なのか、Java には区別することができませんからね。

コンストラクタを色々作れば、その分インスタンスも色々な作り方ができるので、目的に合ったコンストラクタを作っておきましょう。

7.3.6.3 コンストラクタの意義

ところで、コンストラクタは、初期化メソッドですので初期化に必要な処理をすべて書いておくことが望ましいでしょう。

とはいえ、

「後で書けばいいんじゃないの？」

と思う方もいるかもしれません。しかしながら、たとえば、名前が書いていないアドレス帳のデータがあったらどうでしょうか？誰のだかわからない住所や電話番号があっても使えませんよね。住所が書いてない個人データはあってもいいですが、名前だけは絶対に書いておいてほしいものです。

そこで、インスタンスを作った時点で少なくとも名前だけは書き忘れないように、名前を引数としたコンストラクタを作っておけば安心です。

7.3.6.4 コンストラクタまとめ

というわけで、コンストラクタの構文は以下のようになります。

```
        //コンストラクタ
        public クラス名(引数){
                初期化処理;
        }
```

また、コンストラクタの決まり事は以下のとおりです。

1. コンストラクタ名はクラス名と同じ。
2. 返り値は設定できない。
3. new で新しいインスタンスを作ったときに自動的に呼ばれる。
4. コンストラクタでフィールドの値を初期化する。
5. 引数を変えることで複数のコンストラクタを作成可能。

7.4 間違えやすいクラス

7.4.1　間違えやすいデフォルトコンストラクタ

さて、先ほどから PersonalData は何回か作られていますが、これまではコンストラクタを作っていませんでしたよね。

そんなとき内部的にはどう動いていたのでしょうか？

実は、すべてのメソッドには**暗黙的なデフォルトコンストラクタ**というものが用意されています。

簡単に言えば、PersonalData クラスの場合、

```
public PersonalData(){
        //何もしない・・・
}
```

というコンストラクタが作られていたものとして扱われるのです。

したがって、コンストラクタを書かなければ、何も初期化しないコンストラクタが自動的に作られるのです。

したがって、何も引数をおかずに、new をしようとすると、

```
PersonalData pd = new PersonalData();
```

自動的にデフォルトコンストラクタが呼ばれるのです。

いちいち自分でコンストラクタを作らなくて良いので、便利なのですがちょっと問題があります。実は、このデフォルトコンストラクタは、**ほかにコンストラクタを作ったら使えなくなってしまいます**。

つまり、名前を指定して個人データを作成するコンストラクタ、

```
public PersonalData(String n){
        name = n;
        callCount = 0;
}
```

を作った時点で、デフォルトコンストラクタが使えなくなってしまうのです。

これは、「どうしても最初に決めておきたいフィールドがある場合」、たとえば、個人データで言えば名前は絶対に必要ですが、こういったフィールドはコンストラクタで必ず指定しておくようにしないと困ります。

なのに、デフォルトコンストラクタが使えてしまうと、名前を決めないで PersonalData のインスタンスが作れてしまいます。これではあとでプログラムを書いていて困る可能性があります。

そんなわけで、デフォルトコンストラクタは、

- コンストラクタを作っていない場合に、引数なしで勝手に作られるコンストラクタ
- ほかのコンストラクタがあれば、呼び出せない

というものだと覚えておいてください。

ちなみに、引数なしのコンストラクタが必要な場合は、単に、

```
PersonalData(){
        //処理内容
}
```

というコンストラクタを作れば Ok です。

7.4.2　コンストラクタでコンストラクタを呼ぶ

コンストラクタは複数作成することができるというお話がありましたが、複数のコンストラクタを併用したい場合というのがあります。

たとえば、PersonalData のコンストラクタを 2 種類作成しました。

```
public PersonalData(String n){
        name = n;
        callCount = 0;
}


public PersonalData(String n, String a, String p){
        name = n;
        address = a;
        phoneNumber = p;
        callCount = 0;
}
```

しかしこれを見ると、名前を記録する部分と callCount を 0 にする部分はまったく一緒です。

じゃあ、面倒だから、これを一緒にしたいですよね。そのような場合、こうすることが可能です。

```
public PersonalData(String n, String a, String p){
        this(n);
        address = a;
        phoneNumber = p;
}
```

ポイントは、this(n)です。

これは、別のコンストラクタを呼ぶ方法なのです。

通常コンストラクタが呼ばれるのは、

```
new PersonalData(name);
```

のように new を使って新しいインスタンスを作った場合だけですが、コンストラクタ内でだけ、特別に、

```
        this(n);
```

という書き方をして、別のコンストラクタを呼び出すことができるのです。ちなみに、

```
PersonalData(n);
```

という書き方はできません。

また、もうひとつ細かいルールがあります。それは、**別のコンストラクタはコンストラ**

クタの最初でしか呼べないというものです。

ちょっと分かりづらいですが、

```
public PersonalData(String n, String a, String p){
        address = a;
        phoneNumber = p;
        this(n);
}
```

という書き方はできないということです。

なぜこんなルールがあるのか不思議ですが、とにかくそういうルールになっているんです。

結構間違えやすいポイントです。

7.4.3 配列とクラス

さて、クラスのインスタンスは変数と同じようなものだという説明をしましたが、同じように配列の型としてクラスを使うことができます。

クラスの配列を使う場合は、以下のようにします。

```
PersonalData[] personalDataAry = new PersonalData[100];
```

これで 100 個の PersonalData インスタンス分の配列を確保することができました。これによって、100 人分のデータを手に入れました！

・・・と思いがちですが、実はこれでは 100 人分のデータは手に入れていません。

「なぜに！？」

と思ってしまうかもしれませんが、ちょっと思い出してみてください。インスタンスを作るにはどのようにしていたでしょうか？

そう。new というキーワードを使ってインスタンスを作るよ！という宣言をしていましたよね。

```
PersonalData pd;
pd = new PersonalData();
```

ちょうどこんな感じで、最初に

```
PersonalData pd;
```

と書いた時点ではインスタンスは作成されていません。この時点では、PersonalData を指すことができる変数を確保しただけです。PersonalData そのものを作成するには、

```
pd = new PersonalData();
```

として、実際に個人データを作成しなければいけませんよね。

これと同様に、配列の場合も

```
PersonalData[] personalDataAry = new PersonalData[100];
```

だけでは「100 個の PersonalData インスタンスを格納する配列を作るよ」という意味にしかならず、100 個の PersonalData インスタンスはまた別に作成しなければいけません。

たとえば、こんな感じ。

```
PersonalData[] personalDataAry = new PersonalData[100];
personalDataAry[0] = new PersonalData("マリオ");
personalDataAry[1] = new PersonalData("ルイージ");
```

```
personalDataAry[2] = new PersonalData("ピーチ");
personalDataAry[3] = new PersonalData("クッパ");
・・・
```

これで、スーパーマリオ一族のアドレス帳が完成するわけです。これで、いつピーチ姫が誘拐されても、すぐマリオに連絡が取れます。

配列を作った場合、new によって**それぞれの要素にデータをいれなければいけない**ということは最初のうちは忘れがちですので、気をつけてください。

ちなみに、new によってインスタンスを代入していない配列には何も入っていません。Java ではこのように何も入っていない変数は **null** と表現されます。ま、これについてはいずれお話しましょう。

7.4.4 クラスを使うわけ

ところで、本章で紹介した PersonalData クラスですが、結局のところ名前や住所、電話番号などを記録しているだけのクラスです。

前章でお話した通り、これだけならば、以下のようにしても同じことができてしまいます。

```
String name="マリオ";
String address = "きのこ王国";
String phoneNumber = "99";
```

では、なぜわざわざクラスにしているのでしょう？それは、**プログラムのミスをできるだけ減らすため**です。

よく、「Java はオブジェクト指向プログラミングである」と言われます。オブジェクト指向プログラミングというのは、ものを基準にプログラムを作っていく方法のことを言います。

たとえば、クラスを使わない場合、こんなことができてしまいます。

```
String name="マリオ";
String address = "きのこ王国";
String phoneNumber = "99";

int callCount = -1;

System.out.println(name+"には"+callCount+"回電話しました");
```

これを実行するとどうなるでしょうか？

```
マリオには-1 回電話しました
```

おおっと、ありえない表示がされてしまいました。確かに、初代スーパーマリオブラザーズにはマイナス面なんてものもありましたが、-1 回の電話なんてことは聞いたことがありません。

でも、これはクラスの場合も同じことができてしまう・・・と思うでしょ？実は、クラスを上手に使うと、このようなありえない電話回数が設定されることを避けることができるのです。

そのために、PersonalData クラスをこんな風に変えてみましょう。

```
public class PersonalData {
	String name;
	String address;
	String phoneNumber;
	private int callCount;

	public PersonalData(String n){
		name = n;
		callCount = 0;
	}


	/**
	 * 一回電話したら呼び出すメソッド
	 */
	public void call(){
		callCount++;
	}

	public int getCallCount() {
		return callCount;
	}
}
```

まず、ポイントとなるのが、

```
	private int callCount;
```

です。ここにある private というのはアクセス修飾子と呼ばれるもので，このフィールドを使える場所を限定するために利用するものです. private というアクセス修飾子を使うと，同一クラス以外からはこのフィールドを使うことが出来なくなります. この辺については、第 9 章で詳しくお話ししますのでそう言う機能がクラスにはあると思ってください.

例えば,

```
		PersonalData pd = new PersonalData("マリオ");
		pd.callCount = -1;
```

というようなことをしようとすると,

```
CallPrivate.java:6: callCount は PersonalData で private アクセスされます。
		pd.callCount = -1;
		   ^
エラー 1 個
```

というエラーが発生しまて，直接 callCount フィールドを変更できなくなります。これで callCount フィールドは勝手にマイナスなどに変更される心配はなくなりました。

こんなことはクラスを使わないとできませんよね。

しかし、これでは callCount の値を確認することもできません。

```
		System.out.println(pd.callCount);
```

これもできませんから。

そこで、どうするかというと、こんなメソッドを作ります。

```
	public int getCallCount() {
		return callCount;
	}
```

これは、単に callCount の値を int 型として返すだけです。が、これでクラス以外からは

見ることができなかった callCount の値を取得することができるようになります。

さて、こうすることで、電話回数の値を変な値にされる心配がなくなります。コンストラクタで callCount は 0 にしていますし、それ以降は、call メソッドが呼ばれたときに callCount が 1 増えるだけです。

いきなり 100 にされたり、-100 などの変な値にされることは絶対にありません。もちろん、クラス内で変な値にしてしまえば別です．しかしながら，一つのクラスは一人の人間が責任を持って作成するだろうという仮定の下，同一クラス内ならばどんなことでもできるように Java は設計されています.

このように、単に int 型の変数では実現不可能な特殊な機能(電話回数の記録としてみれば、極あたりまえの機能)をクラスを使うことで簡単に実現可能となったのです。

え？

「気をつければいいじゃん」

だって？

もちろん、そのとおりなんですが、プログラムをしばらくやっているうちに、きっと皆さんもある事実に気が付くと思います。

プログラムを書くとき、誰しもが必ずどこか間違える

という事実を。

どんなに注意深くプログラムを書いたとしても、どんなに凄腕のプロが書いたとしても、プログラムにミスがないことは非常に稀です。作者もこれまでに何百というプログラムを作成してきましたが、まったくミスなしにプログラムが完成したことは記憶にありません。

しかしながら、クラスを使うことによって,そのミスを大きく減らすことができるようになるのです。100 個のミスを見つけて修正するのと、10 個のミスを見つけて修正するのでは天と地ほどの差がありますよね。

たとえば、今回紹介した電話回数の設定を間違えずにすむ方法も、これを実装しておかなければ、プログラムが大きくなったときに変な電話回数をいつのまにか設定してしまって混乱してしまうことがきっとあります。

「そんなことないよ」

と思うかもしれませんが、可能性は残されています。

でも、ちゃんとクラスを使えば、そんなことは「100%ない」ということが保証されるのです。これは、嬉しい。

これは、クラスの便利さをあらわすほんの一端でしかありませんが、これ以外にもクラスを使ってできることはたくさんあります。

そして、その多くが「ミスを減らす」「プログラミング労力を減らす」「プログラムを見直したときに読みやすくなる」ことに大いに役立ちます。

ぜひ効率のよいプログラミングを目指してクラスの使い方をマスターしてください。