9 パッケージと修飾子

9.1 package と import

9.1.1 package とは何か

Java では，クラスを整理整頓するための機能がついています．それが，package です．package とは，簡単に言えばフォルダやディレクトリのような存在です．

皆さんが何かファイルを作成した場合，全部 MyDocument の直下に置いたりせずに，ちゃんとフォルダやディレクトリに分けて入れますよね．音楽ファイルなら MyDocument¥Music 以下に入れたり，画像ファイルだったら，MyDocument¥ My Pictures に入れたり．

同じように，携帯電話のプログラムを作ったら，携帯電話関連ディレクトリにまとめたいですし，アドレス帳の個人データクラスを保存するのであれば，アドレス帳クラスと同じディレクトリに整理整頓しておきたいですよね．

パッケージはこのディレクトリ構造を Java 上で実現しているものだと思ってください．

ディレクトリと同様に，パッケージは階層構造によって表現されます．図は，Java にもともと付属している JavaAPI のパッケージ構造を示しています．

まず，java というパッケージがあり，その下に util というパッケージがあり，util の中に Vector,List,HashMap というクラスが存在しています．さらに util パッケージの下には regex パッケージも存在し，その中に Maches,Pattern というクラスが格納されています．

このように，ディレクトリのような階層構造をとることで，Matches と Pattern は同じ仲間であることが分かりますし，Vector,List，HashMap も同じ仲間であることが一目でわかるようになっています．

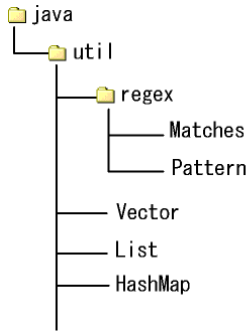

図 パッケージ構造

クラスの正式名は，これらのパッケージを利用して記述されます．

例えば，HashMap クラスの正式名は，

```
java.util.HashMap
```

となります．これを見れば分かると思いますが**パッケージの区切りは「.」**です．ディレクトリの区切りである¥や/などとは違うので注意して下さい。

9.1.2 package の作成

さて，前章で作った個人データクラス PersonalData はどのパッケージに入れるのが適切でしょうか．一応携帯電話用のアドレス帳のデータですので，cellphone.address というディレクトリに整理することにしましょう．

そのような場合，Java ではまず，クラスファイルの**先頭**に以下のような記述を追加します．

```
package cellphone.address;

public class PersonalData {
```

これだけで Clock クラスは cellphone.address パッケージの中に格納されます．必ず先頭に書くんですよ。クラス宣言の後などに書いては駄目です。一番先頭にパッケージは書いてください。

・・・といいたいところですが，実はこれだけではパッケージはうまく動きません．先ほどパッケージはディレクトリと同じようなものだというお話をしましたが、パッケージは実際にディレクトリと同期している必要があります。

たとえば、cellphone.address というパッケージに入っているクラスは、cellphone¥address ディレクトリの中になければいけないのです。

ちょっと分かりづらいですね。

では、まず Java のプログラムを書いている場所を仮に、C:¥java¥の下だとしましょう。もちろんこれは Windows での例であり、Linux などで Java を書いている方は /home/*username*/java でプログラムを書いているかもしれませんね。

普通にパッケージを作らずに Java プログラムを書いてコンパイルした場合、作業ディレクトリに class ファイルが作成されます。これは、第二章ですでにやりましたよね。

このとき、パッケージに保存されているクラスは、パッケージと同じ名前のディレクトリの中に入っていなければいけません。つまり、C:¥java¥の下にプログラムを作っているとしたら、Clock クラスは、C:¥java¥cellphone¥address¥PersonalData.java に記述し、C:¥java¥cellphone¥address¥PersonalData.class というクラスファイルを作成しなければいけないのです。

コンパイルのコマンドは、

```
C:¥java¥> javac cellphone¥address¥PersonalData.java
```

となります。

以上、パッケージを利用する方法をまとめると、以下のようになります。

1. java ファイルの先頭に package 対象パッケージ名;と記述する。

2. java ファイルを、パッケージと同名のディレクトリに置いてコンパイルする

9.1.3 package の利点

このようなパッケージを使う利点としては以下のようなものがあります.

1. クラスを一まとめに整理整頓することで，関係が分かりやすくなる
2. 同一のクラス名を異なる package で利用できる
3. 同一 package 内のクラスは import する必要が無い
4. アクセス制限をつけることが出来る

まず利点 1 については，先ほどから述べているとおりですね．同じような機能を持つクラスが一まとめになっていれば，使うときも使いやすくなるというものです．ちなみに，後述しますが, 同じ package に存在するクラスはまとめて import 出来るという利点があります.

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。

次に，利点 2 ですが，これはまったく同じ名前のクラスを別のパッケージで利用可能であるというものです．つまり，PersonalData というクラスを今回作成しましたが，同じ Clock という名前のクラスは本来は作れませんでした．そりゃそうですよね．同じ名前が二つあったら，どっちのクラスを指しているのか分からなくなってしまいますから．しかしながら，package を上手に利用すると，まったく同じ名前のクラスを二つ作成することが出来るのです．これは，ディレクトリが違えば同じ名前のファイルを作成できるのと同じです.

利点 3 については次の import について説明しないと分かりづらいかもしれませんが，実は Java において，**同一 package にあるクラスと，java.lang パッケージにあるクラスしかそのままでは使うことが出来ない**という決まりがあります．なぜこれまで説明していなかったのかといえば，これまでは同一パッケージ内にあるクラスしか使ってこなかったため，このルールを意識する必要がなかったためです．これまで唯一使ってきた String クラスも java.lang パッケージに入っているため，やはり意識する必要はありませんでした.

これまでは一生懸命意識しないで済むようにしてきましたが、これからはそうもいきませんので、次節では早速異なるパッケージに存在するクラスの利用方法をお送りします。

最後、利点 4 についても次節の import について学ばないと分かりづらいかもしれませんが、簡単に言うと「あるクラスを同一パッケージからしか利用することができなくさせられる」という機能です。

この辺のアクセス制限についてはなぜ必要なのか、後ほど詳しく説明していきますが、上手な Java プログラミングにおいて重要な要素の一つですので気をつけて置いてください。

9.1.4 import の使い方

さきほど PersonalData クラスを cellphone.address パッケージに収めましたよね。そこで、次にパッケージに収めたクラスを利用してみましょう。

以下のようなプログラムを書いてコンパイルしてみます。さて、うまく動くでしょうか・・・？

```
public class Test {
          static public void main(String[] args){
                    PersonalData pd = new PersonalData("マリオ");
                    pd.call();
          }
}
```

どうでしょうか。きっとコンパイルに失敗してしまったことと思います。

```
C:¥java¥>javac Test.java
Test.java:5: シンボルを見つけられません。
シンボル: クラス PersonalData
場所    : Test の クラス
                PersonalData pd = new PersonalData("マリオ");
                ^
Test.java:5: シンボルを見つけられません。
シンボル: クラス PersonalData
場所    : Test の クラス
                PersonalData pd = new PersonalData("マリオ");
                                  ^
```

PersonalData クラスは確かに作っていたはずなのに・・・どこかへ消えてしまったのでしょうか?

そ～ではありません．先ほど書いたとおり、Java では「**無条件で使えるクラスは同じパッケージにあるクラスだけ**」というルールがありますので、無条件で使えないだけなのです。ではどうすればいいのでしょうか？お話は簡単で、「このクラスを使うよ！」という宣言をすればよいだけです。

```
import cellphone.address.PersonalData;

public class Test {
          static public void main(String[] args){
                    PersonalData pd = new PersonalData("マリオ");
                    System.out.println(pd.name+"のデータ");
          }
}
```

ここで、**import** という新しいキーワードが出てきました。これが、**他のパッケージにあるクラスを使うよ**、という宣言なのです。

今度はちゃんとコンパイルして実行できました。

マリオに電話しました

このように、**異なるパッケージに存在するクラスを使う場合は、そのクラスを import する**ということを忘れずに行ってください。

ちなみに，あるパッケージにあるクラスを全部使いたい場合，

```
import cellphone.address.PersonalData;
import cellphone.address.CompanyData;
・・・
```

などいちいち全部書くのは面倒な場合があります.

そのような場合，

```
import cellphone.address.*;
```

とすることで，同一パッケージに存在するすべてのクラスを import したのと同じ効果が得られます.

たくさんのクラスを一度に import するときは，活用しましょう.

import 文は、package 文の直後に書くという決まりがあります。すなわち、通常のクラス宣言の場合、

```
package　パッケージ名;
import インポートするクラス名;
import インポートするクラス名;
・・・

class クラス名{
・・・
}
```

という書き方になります。

このように，imoprt はいくつでも書くことができますが，package は一つのクラスに一つしか書けません。さて、なぜだかは分かりますよね？

これは，同じファイルが一つのディレクトリにしか存在できないのと同じで，一つのクラスは一つのパッケージにしか存在できないからなんですね.

ところで、import を使う場合に注意点があります。たとえば、標準で利用可能な、JavaAPI というクラス集には Date というクラスは、java.util.Date と、java.sql.Date というパッケージ中に存在します。したがって、ただ単に Date とだけ書くと、どちらのパッケージに含まれる Date なのか混乱してしまうことになります。そこで、あるクラスを使う場合、どの package にあるクラスを使うのかを、あらかじめ import によって指定しておかなければいけません。

imoprt 命令は、クラス宣言の前に書きます。クラス宣言の先頭には package 命令が来ることが決まりですが、今回は package 命令を使っていないので、ファイルの先頭に書くことになります。というわけで、ファイルの先頭に、

```
import java.sql.Date;
```

と書けば「java.sql.Date を利用するよ」と宣言できます。これによって、このファイル内の Date は全て java.sql.Date として扱われるようになります。

ところで,先ほど照会した用に,

```
import java.util.*;
```

と書けば,「java.util にある全てのクラスを利用する」ことができます。このように宣言すれば、Date と書くだけで自動的に java.util.Date だと判断されますし、それ以外の java.util にあるクラスも全部使えるようになります。

しかし、この宣言をしてしまった後、「やっぱり java.sql.Date も使いたいな」と思ったときはどうすればよいのでしょうか？　そんなときは、

```
static public void main(String[] args){
          Date date; //java.util.Date
          java.sql.Date date2; //java.sql.Date;
}
```

このように、package 名からすべて書けば、java.sql.Date も使うことが出来ます。楽勝！

さて、こんな決まりを教えられると、ものぐさな我々は、「どうせなら java.util と java.sql にあるクラス全部使えるようにしたいや」と思ってしまい、

```
import java.util.*;
import java.sql.*;

・・・

static public void main(String[] args){
          Date date; //←どっち？
}
```

なことをしてしまいます。このときの Date がどちらを意味するのか…。残念ながらそれは誰にもわかりません。実は、こういうことをしようとすると、Date がどちらか分からなくなってしまうため、エラーが発生し、コンパイルすることが出来なくなるからです。

```
 Test.java:14: Date の参照はあいまいです。java.sql の クラス java.sql.Date と
java.util の クラス java.util.Date が両方適合します。
                         Date date;      //←どっち？
                         ^
 エラー 1 個
```

このように、「型 Date はあいまいです」と怒られた場合は、素直に明示的に Date を宣言するようにしましょう。

ところで、両方 import すると**あいまい**になるんだったら、両方 import しないとどうなるでしょうか？もうすでに知っていますよね．両方 import しなければ，どちらのクラスも使うことが出来ません．結構忘れがちですが，重要な機能 import 必ず忘れずに宣言するようにしましょう．

9.2アクセス修飾子

9.2.1 メソッド・フィールド修飾子

クラスに修飾子がつけられるように、メソッドとフィールドにも修飾子をつけることが可能です。

まず、最も重要なのが**アクセス修飾子**と呼ばれる、public/protected/private です。これらの修飾子はメソッドやフィールドがどこから使うことができるかを指定します。

と書くとどういうことか良く分からないかもしれませんが，メソッドとフィールドは，どのクラスから利用できるかというのを制限することが出来るのです．どのクラスから，といってもクラス一つ一つを書いていくわけではなく，どういう関係にあるクラスから使えるかを指定するのです.

同じクラスからしか使えなかったり，同じパッケージにあるクラスからしか使えなかったりと色々制限ができます.

まず、**public** は、どこからでも利用することができるメソッドやフィールドを作成するために使います。

次に、**protected** なメソッドやフィールドは、派生クラスまたは同一パッケージ内のクラスからのみ使うことが可能です。

そして、**private** メソッドやフィールドは、同一クラスからしか利用することができません。

さらに、**アクセス修飾子がない**メソッドとフィールドは、同じパッケージ内のクラスから利用可能になります。

と、言葉で書いても分かりづらいでしょうから、表にしてみましょうか。

アクセス修飾子	意味
public	すべてのクラスから利用可能
protected	派生クラスか同一パッケージ内から利用可能
修飾子なし	同一パッケージからのみ利用可能

private	同一クラスからのみ利用可能

それぞれのアクセス修飾子について具体例を挙げながら説明していきましょう。

なお，説明のためアクセス制限が厳しい方から順番に説明していきますね.

9.2.2 private

private 修飾子がついたメソッドやフィールドは，**同一クラスからのみアクセスできる**ようになります．例を見てみましょう.

```
package access;

public class PrivateClass {
	private void privateMethod(){
		System.out.println("同一クラスからしか使えないメソッド");
	}

	private String privateField = "同一クラスからしか見えないフィールド";
}
```

ここでは，privateMethod という private 修飾子のついたメソッドと，privateField という private 修飾子のついたフィールドを作成しました.

さあ，これを別のクラスから利用してみましょう.

```
import access.PrivateClass;

public class AccessTest {

	public static void main(String[] args) {
		PrivateClass privateClass = new PrivateClass();
		privateClass.privateMethod();
		System.out.println(privateClass.privateField);

	}
}
```

このクラスでは，private 修飾子をつけたメソッドとフィールドを呼び出しています.

さあ，これをコンパイルしようとすると・・・

```
AccessTest.java:9: privateMethod() は access.PrivateClass で private アクセスされます。
		privateClass.privateMethod();
		            ^
AccessTest.java:10: privateField は access.PrivateClass で private アクセスされます。
		System.out.println(privateClass.privateField);
		                                ^
エラー 2 個
```

というわけで，private なメソッドとフィールドを利用しようとしたプログラムは失敗に終りました.

このように，private なメソッドとフィールドは他のクラスからは利用することが出来ま

せん.

private なメソッドとフィールドを利用する場合は，以下のようにします．まず，後述するどこのクラスからでも利用可能な public メソッドを一つ作成します.

```
public class PrivateClass {
	private void privateMethod(){
		System.out.println("同一クラスからしか使えないメソッド");
	}

	private String privateField = "同一クラスからしか見えないフィールド";


	public void privateTest(){
		privateMethod();
		System.out.println(privateField);
	}
}
```

そして，このメソッドを呼び出します.

```
		PrivateClass privateClass = new PrivateClass();
		privateClass.privateTest();
```

こうすることで,

```
同一クラスからしか使えないメソッド
同一クラスからしか見えないフィールド
```

private なメソッドやフィールドを利用することが可能となります.

9.2.3 アクセス修飾子なし

private の次にアクセス制限が厳しいのが「修飾子無し」です．これは，デフォルトのアクセス制限として使われるものです.

アクセス修飾子無しでメソッドやフィールドを作成すると，そのメソッドとフィールドは，**同一クラス及び同一パッケージからのみアクセスできる**ようになります.

これも例を見てみましょうか.

```
package access;

public class NoModifierClass {
	void noModifiereMethod(){
		System.out.println("同一パッケージからしか使えないメソッド");
	}

	String noModifiereField = "同一パッケージからしか見えないフィールド";
}
```

このクラスのメソッドとフィールドはアクセス修飾子がついていません．さあ，これを別パッケージのクラスから利用してみましょう.

```
import access.NoModifierClass;

public class AccessTest {

	public static void main(String[] args) {

		NoModifierClass noModifierClass = new NoModifierClass();
		noModifierClass.noModifiereMethod();
```

```
                System.out.println(noModifierClass.noModifiereField);
        }

}
```

ここでは，private の場合と同様デフォルトパッケージから利用しようとしています．さあ，この結果は・・・

```
AccessTest.java:11: access.NoModifierClass の noModifiereMethod() は public ではありません。パッケージ外からはアクセスできません。
                noModifierClass.noModifiereMethod();
                                ^
AccessTest.java:12: access.NoModifierClass の noModifiereField は public ではありません。パッケージ外からはアクセスできません。
                System.out.println(noModifierClass.noModifiereField);
                                                  ^
エラー 2 個
```

このように，コンパイルに失敗し，パッケージ外からは利用できませんでした．ここまでは private をつけた場合と同じですね．

では，private との差を見るために，同一パッケージのクラスから呼び出してみましょう．

先ほどの呼び出しクラス AccessTest を access パッケージに移動してみます．

```
package access;

public class AccessTest {
        public static void main(String[] args) {
                NoModifierClass noModifierClass = new NoModifierClass();
                noModifierClass.noModifiereMethod();
                System.out.println(noModifierClass.noModifiereField);
        }
}
```

パッケージを移動するには，クラス宣言の最初に package をつけて，.java ファイルのディレクトリを access に移動すれば Ok ですよね．

このように，AccessTest クラスと NoModifierClass クラスが同一パッケージに移動しておくと，コンパイルが成功します．

```
C:¥java >javac access¥AccessTest.java

C:¥java >java access.AccessTest
同一パッケージからしか使えないメソッド
同一パッケージからしか見えないフィールド
```

実行も成功しました．これで同一パッケージからなら利用可能なことが確認できました．

9.2.4 protected

protected 修飾子のついたメソッドとフィールドを作成すると，**同一クラス，同一パッケ**

ージ及び派生クラスから利用できる**ようになります.

ようするに，修飾子なし+派生クラスから利用できるわけですね．では，早速 protected なメソッドとフィールドを持つクラスを作成してみましょう.

```
package access;

public class ProtectedClass {
	protected void protectedMethod(){
		System.out.println("派生先から利用可能なメソッド");
	}

	protected String protectedField = "派生先から参照可能なフィールド";
}
```

そして，毎度おなじみ別パッケージのクラスから呼び出しテストをしてみましょう.

```
import access.ProtectedClass;

public class AccessTest {

	public static void main(String[] args) {
		ProtectedClass protectedClass = new ProtectedClass();
		protectedClass.protectedMethod();
		System.out.println(protectedClass.protectedField);
	}

}
```

コンパイルしようとすると・・・

```
AccessTest.java:23: protectedMethod() は access.ProtectedClass で protected アクセスされます。
		protectedClass.protectedMethod();
		              ^
AccessTest.java:24: protectedField は access.ProtectedClass で protected アクセスされます。
		System.out.println(protectedClass.protectedField);
		                                  ^
エラー 2 個
```

というわけで，例のごとくコンパイル失敗です．ここまでは想定内，private や修飾子無しの場合と一緒ですね.

では,次に別パッケージで ProtectedClass を継承した派生クラスを作成してみましょう.

```
import access.ProtectedClass;


public class ExProtectedClass extends ProtectedClass {

	public void protectedTest(){
		protectedMethod();
		System.out.println(protectedField);
	}
}
```

こんなクラスを作成してみます．このとき，派生クラス ExProtectedClass では，

protectedMethod や protectedField を作成していないので，継承元クラスである ProtectedClass のメソッドとフィールドを利用していることに注意してください．

そして，このクラスを作成して protectedTest メソッドを呼び出してみます．

```
                ExProtectedClass exProtectedClass = new ExProtectedClass();
                exProtectedClass.protectedTest();
```

これを実行すると・・・

```
派生先から利用可能なメソッド
派生先から参照可能なフィールド
```

というわけで，派生したクラス ExProtectedClass からは protected なメソッドとフィールドを利用することが出来ることが確認できました．

ちなみに，private なメソッドは利用することが出来ません．ためしに，PrivateClass の派生クラスを作って同じように private なメソッドとフィールドを利用しようとしてみます．

```
import access.PrivateClass;


public class ExPrivateClass extends PrivateClass {
        public void privateTest(){
                privateMethod();
                System.out.println(privateField);
        }
}
```

これをコンパイルすると，

```
ExPrivateClass.java:6: シンボルを見つけられません。
シンボル: メソッド privateMethod()
場所    : ExPrivateClass の クラス
                privateMethod();
                ^
ExPrivateClass.java:7: privateField は access.PrivateClass で private アクセスされます。
                System.out.println(privateField);
                                   ^
エラー 2 個
```

となり，コンパイルに失敗します．

これが，private と protected の最大の違いです．

9.2.5 public

まず，public は全てのクラスから自由に使うことが可能なアクセス修飾子です．非常にわかりやすいですね．わざわざ例を出すまでもないのですが・・・

こんなクラスを作ってみましょう.

```
package access;

public class PublicClass {
	public void publicMethod(){
		System.out.println("どこからでも利用可能なメソッド");
	}

	public String publicField = "どこからでも参照可能なフィールド";
}
```

publicMethod というメソッドと，publicField というフィールドが用意されています.また，access というパッケージに保存されていることに注意してくださいね.

次に，パッケージ指定無し(デフォルトの位置)でこんなクラスを作成して実行しましょう.

```
import access.PublicClass;

public class AccessTest {

	public static void main(String[] args) {
		PublicClass publicClass = new PublicClass();
		publicClass.publicMethod();
		System.out.println(publicClass.publicField);
	}

}
```

ポイントは，このクラスがデフォルトパッケージにある点です．つまり，異なるパッケージにあるわけですね.

これを実行すると，

```
どこからでも利用可能なメソッド
どこからでも参照可能なフィールド
```

となり，どこからでも参照できることが分かりました.

9.2.6 アクセス修飾子まとめ

メソッドとフィールドのアクセス修飾子をまとめると，以下のようになります.

	同一クラス	同一パッケージ	派生クラス	その他のクラス
private	○	×	×	×
修飾子なし	○	○	×	×
protected	○	○	○	×
public	○	○	○	○

表中の○が利用可能，×が利用不可能をあらわします.

このように，private⇒修飾子なし⇒protected⇒public と進むにしたがって利用可能なクラスが増えていくことがわかると思います.

大事なデータほどアクセス制限が厳しい private などの修飾子を使うようにすると良いでしょう.

9.3クラスアクセス修飾子

Java でクラスを作るときに、class の前に public と書いていたのを覚えているでしょうか？これまではおまじないのように書いていましたが，この public もメソッドなどにつけていたものと同じく，アクセス修飾子になります.

ただし，クラス修飾子は public か修飾子無しのどちらかしか使うことが出来ません.

それぞれがどういう意味になるか，説明して行きたいと思います.

9.3.1 修飾子なし

まずは，修飾子無しの場合です．クラス作成時に public 修飾子をつけなかった場合，そのクラスは**同一パッケージ内からのみ利用することができます**.

ためしに，以下のようなクラスを作成してみましょうか.

```
package classaccess;

class NoModifiereClass {
          public void check(){
                    System.out.println("作成成功");
          }

}
```

NoModifiereClass のクラス宣言の前に public と書いていないことに注目して下さい.

さて，このクラスは，classaccess パッケージにあります．そこで，ためしにこのクラスをデフォルトパッケージにあるクラスから利用してみようと思います.

```
import classaccess.NoModifierClass;

public class ClassAccessTest {

          public static void main(String[] args) {
                    NoModifierClass noModifierClass = new NoModifierClass ();
          }

}
```

さて，どうなるでしょうか？

```
ClassAccessTest.java:1: classaccess の classaccess.NoModifierClass は public で
はありません。パッケージ外からはアクセスできません。
import classaccess.NoModifierClass;
                   ^
ClassAccessTest.java:6: classaccess の classaccess.NoModifierClass は public で
はありません。パッケージ外からはアクセスできません。
                NoModifierClass noModifierClass = new NoModifierClass ();
                ^
ClassAccessTest.java:6: classaccess の classaccess.NoModifierClass は public で
```

```
はありません。パッケージ外からはアクセスできません。
                NoModifierClass noModifierClass = new NoModifierClass ();
                                                  ^
ClassAccessTest.java:6: classaccess.NoModifierClass の NoModifierClass() は
public ではありません。パッケージ外からはアクセスできません。
                NoModifierClass noModifierClass = new NoModifierClass ();
                                                  ^
エラー 4 個
```

というわけで，import に失敗してしまいました．原因は，public ではないクラスだからと表示されていますね．このように，public 修飾子をつけていないクラスは，他のパッケージからは使えないのです．

念のため，同一パッケージでは使えるかどうか試してみましょうか．

```
package classaccess;

public class ClassAccessTest2 {

          public static void main(String[] args) {
                    NoModifierClass noModifierClass = new NoModifierClass();
                    noModifierClass.check();
          }

}
```

今度のクラスは，classaccess パッケージ，つまり NoModifierClass と同じパッケージにあることに注意してください．

これをコンパイルすると，無事コンパイルし，実行できます．

```
作成成功
```

このように，同一パッケージからしか利用できないようなクラスを作成したいとき，修飾子無しのクラスを作成することになります．

9.3.2 public

public 修飾子は、すでに説明した通りどのパッケージから利用可能なのかを示したものです。public と書いてあればどのクラスからでもそのクラスを利用することが可能です。

```
package classaccess;

public class PublicClass {
                    public void check(){
                    System.out.println("作成成功");
          }

}
```

こんなクラスを作成すれば，

```
import classaccess.PublicClass;

public class ClassAccessTest {
```

```
	public static void main(String[] args) {
		PublicClass publicClass = new PublicClass();
		publicClass.check();
	}

}
```

これを実行すると，

```
作成成功
```

このように，異なるパッケージのクラスからでもインスタンスを作成することが出来ます.

基本的にクラスは public で作ることが多いでしょう．ですが，時にはあまり人には見せたくない内緒のクラスを作る場合もありますので，そのような場合は修飾子無しで作成するようにします.

9.3.3 クラスアクセス修飾子まとめ

では，最後にクラスのアクセス修飾子をまとめます.

	同一パッケージ	そのほかのクラス
修飾子なし	○	×
public	○	○

表中の○が利用可能，×が利用不可能をあらわします.

クラスアクセス修飾子は，public か修飾子なししかありませんので，その点を注意してください.

9.4static 修飾子

Java にはアクセス修飾子以外にも修飾子が用意されています。その代表的なもののひとつが static 修飾子です。修飾子は static です。static 修飾子がついたフィールドやメソッドは、クラスフィールド、クラスメソッドと呼ばれるものとなり、インスタンスに関係なく使えるという意味です。

9.4.1 static フィールド

まず、static フィールドについて説明しましょう。

フィールドに static をつけると、そのフィールドは**クラス変数**と呼ばれるようになります。このクラス変数はインスタンスによらず値が一定となります。

次のような例を見てみましょう。

```
class StaticTest{
	static int data = 1;
}
```

さらに、main メソッドでこんな記述をしてみましょう。

```
	StaticTest test1 = new StaticTest();
```

```
        StaticTest test2 = new StaticTest();
        test1.data = 0;
        System.out.println(test2.data);
```

さあ、この結果はどうなるでしょうか？本来ならば、test1 インスタンスのフィールド data と、test2 インスタンスのフィールド data は異なるものなので、test1.data を 0 にしても、test2.data の値は 1 のままのような気がしますが・・・

```
0
```

と表示されました。

このように、static フィールドは、すべてのインスタンスで共通の値を持つようになります。

9.4.2 定数

ちなみに、public static final なフィールドのことを**定数**と呼びます。これは、誰でも使うことができるクラスに依存した特定の値ということになり、特別の扱いとなります。後述するインターフェースでも通常のフィールドは作成できませんが、定数だけは指定することが可能です。

なお、定数名はすべて大文字で書くのが一般的です。

```
class StaticTest{
        public static final int STATIC_DATA = 1;
}
```

なお、クラス変数や定数は、インスタンスがなくても利用することが可能です。インスタンスなしに暮らす変数を利用する場合は以下のようにします。

```
StaticTest.data = 10;
```

このように、

```
クラス名.クラス変数名
```

とすることで、クラス外からでもインスタンスなしで直接クラス変数を扱うことができます。覚えて置いてください。

9.4.3 static メソッド

次に、static なメソッドですが、これはまた特殊なメソッドになります。

static メソッドは基本的にインスタンスと関係なく利用可能なメソッドということになります。そのため、クラスメソッドと呼ばれます。

たとえば、以下のようなクラスを作ってみましょう。

```
class StaticClass{
        static void method(){
                System.out.println(“メソッド呼び出し”);
        }
}
```

さらに、main においてこんな記述をします。

```
        StaticClass.method();
```

クラス名から直接メソッドを呼んでいて奇妙な感じがしますが、これをコンパイルするとちゃんとコンパイルできて、実行もできます。

```
メソッド呼び出し
```

というわけで、ちゃんと実行できました。このように、static メソッドはインスタンスとは無関係に呼び出すことができます。

なお、static メソッド呼び出しの構文は、

```
クラス名.メソッド名(引数);
```

となります。

ところで、インスタンスとは無関係に呼び出せる static メソッドでは、フィールドの値はどうなるのでしょうか？先ほどの static メソッドのクラスをちょっと変更してみましょう。

```
class StaticClass{
        int data = 0;
        static void method(){
                System.out.println("メソッド呼び出し");
                System.out.println("data の値は"+data);
        }
}
```

としてみましょうか。さあ、どうなるでしょうか！？

```
StaticClass.java:5: static でない 変数 data を static コンテキストから参照するこ
とはできません。
                System.out.println("data の値は"+data);
                                                ^
エラー 1 個
```

というわけで、static メソッドではフィールド変数を使うことはできません。

ただし、static なフィールド、つまりクラス変数だけは別です。先ほどのクラスも、フィールド data を static な変数に変更すれば、利用することが可能となります。

```
class StaticClass{
        static int data = 0;
        static void method(){
                System.out.println("メソッド呼び出し");
                System.out.println("data の値は"+data);
        }
}
```

このメソッドを呼び出してみます.

```
        public static void main(String[] args) {
                StaticClass.method();
        }
```

すると,

```
メソッド呼び出し
data の値は 0
```

となり，data 変数を呼び出すことが出来ました.

これは，static なメソッドは全インスタンスで共通の動きをしなければいけないため，インスタンスごとに異なる値を持つフィールドは利用できなかったのですが，static フィールド，つまりクラス変数であれば全インスタンスで同じ値を持つことが保障されていますの

で，利用可能となるわけです.

9.4.4 static まとめ

static 修飾子をまとめると以下のようになります.

	フィールド	メソッド
static なし	インスタンスごとに異なる値	インスタンスから呼び出し
static	全てのインスタンスで共通の値	クラス名から呼び出し

static とついていたら，その時点でインスタンス全部で共通に使えるものなんだな，と思うと分かりやすいかもしれません.

また，わざわざ new してインスタンスを作成しなくても使うことができるのがちょっとしたポイントの一つです.

9.4.5 main メソッド

ところで、Java プログラムを実行するさい、まず main メソッドからスタートすることは一番最初にお話しました。

そのメソッドをもう一度見てみましょう。

```
class Start{
          public static void main(String[] args) {
          }
}
```

これを見ると、main メソッドは static メソッドであることが分かります。さらに、その引数として、String 配列が与えられています。

Java プログラムを実行するときは、実行する main メソッドの在るクラス名を書いていましたよね。

```
java Start
```

実は、Java プログラムはここで引数としてかかれたクラスの main メソッドを呼び出していたのです。

簡単に言えば、

```
Start.main(引数);
```

という形で main メソッドを呼び出していたんですね。

どうでしょうか。static メソッドまで学ぶと、なんとなく Java プログラムも得体の知れない動きをするわけではなく、ちゃんとルールに従って動いているんだなあ、という印象を受けたのではないでしょうか？

ちなみに、main メソッドを呼び出すときの引数 args には、main メソッドがかかれているクラス名の後に書かれた文字列が入ることになっています。

たとえば、

```
java Start hello world
```

と書けば、args の中身は、

```
args[0];//hello
args[1];//world
```

となります。

main への引数は条件を変えてプログラムを動かすときに色々と役立ちますので、是非利用してください。

9.4.6 System.out.println の説明

さて、さきほど main について説明しましたが、次は System.out.println メソッドについてです。

もうなんとなく検討がついているかもしれませんが、ここには static がフィールドが使われています。分かりますか？

そう、最初の System.out の部分です。

実は、System というクラスに、out という public static なフィールドが存在するのです。大体こんな感じで書いてあると思ってください。

```
public class System{
          public static PrintStream out;
          //その他の機能
          //・・・
}
```

つまり、System.out によってあるインスタンス out を取得するのです。そして、out のメソッド println を最後に呼び出しています。

out のメソッド println が画面に出力するようにしているんですね。

どうでしょうか？

これまでは「画面に文字を出力するためのおまじない」としか見えなかった System.out.println の意味がわかったのではないでしょうか？詳しく知れば知るほど、今まで分からなかった謎が解けてきて楽しくないですか？筆者はこれに気づいたときに簡単の声をあげたものです。あ、でも隣の友人に自慢したら「何をいまさら」といわれてしまいましたが。

自慢する相手は選んだ方がいいようです。

9.4.7 その他の修飾子

本当はこれ以外にも修飾子はあるのですが、当分の間は使う機会はないでしょう。もっと上級 Java プログラマになったときに改めて勉強すればよいと思いますので、本書では割愛いたします。ちなみに筆者もあんまり使ったことがありません。

9.5修飾子の利用例

せっかく本章で修飾子を勉強したのですから，ちょっとこの修飾子を使ったクラスの例をお見せしましょう.

今回目指すのは，PersonalData クラスの保護です.

よく，携帯電話のアドレス帳なんかに，パスワードを入れないと見られないようにする機能がありますよね．他人に勝手にアドレス帳を見られると恥ずかしいことも良くありますからね.

そこで，PersonalData クラスを拡張して，パスワードを入れない限りデータを読み込めないクラスを作成したいと思います.

そんなことが果たして出来るんでしょうか・・・？

9.5.1 アクセス制限を使った秘密アドレスクラス

まず，基本となる PersonalData について説明しましょう．今回利用する PersonalData はこれまで説明してきたものと似ていますが，簡単のため機能を大分限定したものにしています.

```
public class PersonalData {

          protected String name;
          protected String address;
          protected String phoneNumber;

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

          public String getAddress() {
                    return address;
          }

          public String getName() {
                    return name;
          }

          public String getPhoneNumber() {
                    return phoneNumber;
          }
}
```

今回扱うデータは，名前，住所，電話番号だけです.

また, アクセス制限を加えて, 直接データを利用することは出来ません. ただ, getAddress などのメソッドを用意しておいて，外部からはこれらのメソッドを使ってデータを利用することにします.

ちなみに，このようにフィールドを直接外部から使わせない手法をカプセル化といいます. ちょっと面倒なことをしているようですが, 第 7 章で callCount が変な値にならないようにする手法をちょこっと説明したように，データを保護するという意味で非常に重要な手法です．こうすることで, name などの値は派生クラスや同パッケージのクラスから以外は変更できないわけですから，データはそう簡単に変更されてしまう心配がありません.

さて，この PersonalData はこんな感じで使うことになります．

```
	public static void main(String[] args) {
		PersonalData personalData = new PersonalData("ピーチ", "キノコ王国", "090-1234-5678");

		System.out.println("名前は"+personalData.getName());
		System.out.println("住所は"+personalData.getAddress());
		System.out.println("電話番号は"+personalData.getPhoneNumber());

	}
```

これを実行すれば，

```
名前はピーチ￥
住所はキノコ王国
電話番号は 090-1234-5678
```

こんな感じで名前などを自由に取り出せます．

さあ，この PersonalData クラスから新しいクラスを派生させることによって，これまでは自由に読み出せた名前，住所などを秘密にしてしまいましょう．

そのために，こんな派生クラスを作成します．

```
package chapter9.section5example;

public class SecretPersonalData extends PersonalData {

	private String key;

	public SecretPersonalData(String n, String a, String p, String k) {
		super(n, a, p);
		key = k;
	}

	public String getAddress() {
		return "秘密です";
	}

	public String getName() {
		return "秘密です";
	}

	public String getPhoneNumber() {
		return "秘密です";
	}


}
```

さあ，かなりマニアックなクラスになりました．

さて，このクラスを作成します．コンストラクタが PersonalData から変更されていることに注意して下さい．４つ目の引数には，この SecretPersonalData を読むために必要なパスワードを入力しておきます．

```
SecretPersonalData secretPD = new SecretPersonalData("デイジー", "サラサ・ランド", "090-0000-0000", "naisho");
```

さて，これで普通に住所を取得してみるとどうなるでしょうか？

```
		System.out.println("名前は"+secretPD.getName());
		System.out.println("住所は"+secretPD.getAddress());
```

```
System.out.println("電話番号は"+secretPD.getPhoneNumber());
```

さあ，これを実行してみましょう．

```
名前は秘密です
住所は秘密です
電話番号は秘密です
```

おっと，全部秘密になりました！これでこのデータを誰かに読まれる心配はありません．これでピーチ姫にアドレスを見られても，デイジー姫と浮気していることはばれません．うひひ．

こんな事が出来たポイントは，SecretPersonalData で getAddress などを下記のようにオーバーライドしたことにあります．

```
	public String getAddress() {
		return "秘密です";
	}

	public String getName() {
		return "秘密です";
	}

	public String getPhoneNumber() {
		return "秘密です";
	}
```

これによって，これまで住所などを取得するために唯一利用できていた getAddress メソッドが「秘密です」としか返さなくなったので，住所，名前などを取得する方法が失われたのです．

もし，address などを public なフィールドにしていたら，

```
System.out.println("名前は"+secretPD.name);
```

とすることで，一発で秘密にしておきたい名前がばれてしまいます．マリオピンチ．

というわけで，フィールドを protected にしておくことで，名前や住所を隠蔽することに成功しました．なかなか便利です．

しかし，このままでは誰にも利用出来ないままで終わってしまいますよね．そこで，パスワードを入力すれば住所などの情報を取り出せるようにしましょう．

```
public class SecretPersonalData extends PersonalData {

	private String key;

	public SecretPersonalData(String n, String a, String p, String k) {
		super(n, a, p);
		key = k;
	}

	public String getAddress() {
		return "秘密です";
	}

	public String getName() {
		return "秘密です";
	}
```

```
	public String getPhoneNumber() {
		return "秘密です";
	}


	public String getAddress(String k) {
		if(key.equals(k)){
			return super.getAddress();
		}
		return "秘密です";
	}

	public String getName(String k) {
		if(key.equals(k)){
			return super.getName();
		}
		return "秘密です";
	}

	public String getPhoneNumber(String k) {
		if(key.equals(k)){
			return super.getPhoneNumber();
		}
		return "秘密です";
	}

}
```

こんなメソッドを追加しました．このメソッドは，各種フィールドを取得するメソッドに，引数としてパスワードを入力させるようにしているものです．

使い方はこんな感じ．

```
		System.out.println("名前は"+secretPD.getName("naisho"));
		System.out.println("住所は"+secretPD.getAddress("naisho"));
		System.out.println("電話番号は"+secretPD.getPhoneNumber("naisho"));
```

ここで，最初に個人データを作ったときに４番目の引数として設定した「naisho」を引数としています．

これを実行すると・・・

```
名前はデイジー
住所はサラサ・ランド
電話番号は 090-0000-0000
```

というわけで，パスワードさえ分かっていれば，ちゃんと住所も取得できます．これはすごい．

ポイントは，

```
	public String getAddress(String k) {
		if(key.equals(k)){
			return super.getAddress();
		}
		return "秘密です";
	}
```

この部分ですね．コンストラクタで指定したパスワードが key フィールドに保存されていますので，これと引数 k が一致した場合のみ住所を返すようにしています．

このように，フィールドのアクセス制限をすることによって，秘密個人情報クラスを簡単に作れるようになりました.

9.5.2 パスワードの統一

さて，先ほどはアクセス制限によって秘密個人情報クラスを作成することに成功しましたが,今度は static を使ってさらに秘密個人情報クラスを便利にすることを考えましょう.

先ほどの秘密個人情報クラスは，個人情報ごとにパスワードをかけることになっています．これはちょっと面倒くさいですよね．どうせなら秘密個人情報クラス全体でパスワードを共有したいものです.

というわけで，ここは static の出番ですね！

SecretPersonalData クラスをこんな風に改良します.

```
public class SecretPersonalData2 extends PersonalData {

	static private String key;

	public SecretPersonalData2(String n, String a, String p) {
		super(n, a, p);
	}

	static public void setKey(String k){
		key = k;
	}

	public String getAddress() {
		return "秘密です";
	}

	public String getName() {
		return "秘密です";
	}

	public String getPhoneNumber() {
		return "秘密です";
	}


	public String getAddress(String k) {
		if(key.equals(k)){
			return super.getAddress();
		}
		return "秘密です";
	}

	public String getName(String k) {
		if(key.equals(k)){
			return super.getName();
		}
		return "秘密です";
	}

	public String getPhoneNumber(String k) {
		if(key.equals(k)){
			return super.getPhoneNumber();
```

```
                    }
                    return "秘密です";
          }

}
```

違いは，key フィールドがクラス変数に変更されたこと，そして，static な setKey メソッドが作成されたこと，そして，コンストラクタからパスワードを示す引数が削除されたことです.

このクラスを利用してみましょう.

```
                    SecretPersonalData2.setKey("secret");
                    SecretPersonalData2 secretData2 = new SecretPersonalData2("デイジー", "サラサ・ランド
", "090-0000-0000");
                    SecretPersonalData2 secretData3 = new SecretPersonalData2("クッパ", "8-4",
"090-0000-0000");

                    System.out.println("名前は"+secretData2.getName());
                    System.out.println("住所は"+secretData2.getAddress());
                    System.out.println("電話番号は"+secretData2.getPhoneNumber());


                    System.out.println("名前は"+secretData3.getName());
                    System.out.println("住所は"+secretData3.getAddress());
                    System.out.println("電話番号は"+secretData3.getPhoneNumber());
```

ここでは，まず SecretPersonalData2 の static なメソッド，setKey を呼び出してパスワードを指定しています．static なメソッドはクラス名から呼び出すんでしたよね.

そして，その後秘密データを入力しています．さて，ちゃんと秘密のデータになっているか確認してみましょう.

```
名前は秘密です
住所は秘密です
電話番号は秘密です
名前は秘密です
住所は秘密です
電話番号は秘密です
```

お，ちゃんと二人分秘密のデータとなりました.

次に，パスワードつきの getName などのメソッドを呼び出して，パスワードを入れればちゃんと読み取れることを確認しましょう.

```
                    System.out.println("名前は"+secretData2.getName("secret"));
                    System.out.println("住所は"+secretData2.getAddress("secret"));
                    System.out.println("電話番号は"+secretData2.getPhoneNumber("secret"));


                    System.out.println("名前は"+secretData3.getName("secret"));
                    System.out.println("住所は"+secretData3.getAddress("secret"));
                    System.out.println("電話番号は"+secretData3.getPhoneNumber("secret"));
```

今回のパスワードは setKey で設定したとおり，「secret」です.

さあ，この結果は・・・

```
名前はデイジー
```

```
住所はサラサ・ランド
電話番号は 090-0000-0000
名前はクッパ
住所は 8-4
電話番号は 090-0000-0000
```

おっと，ちゃんとデータを読み取ることが出来ました．また，デイジーのデータを読む場合もクッパのデータを読む場合も統一されたパスワードで読めました．

もし，パスワードがピーチ姫にばれそうになったらさっと変更してしまえば，Ok です．

```
SecretPersonalData2.setKey("kinoko");
System.out.println("名前は"+secretData2.getName("secret"));
System.out.println("住所は"+secretData2.getAddress("secret"));
System.out.println("電話番号は"+secretData2.getPhoneNumber("secret"));


System.out.println("名前は"+secretData3.getName("secret"));
System.out.println("住所は"+secretData3.getAddress("secret"));
System.out.println("電話番号は"+secretData3.getPhoneNumber("secret"));
```

これで，実行すれば，

```
名前は秘密です
住所は秘密です
電話番号は秘密です
名前は秘密です
住所は秘密です
電話番号は秘密です
```

さっきまでは読めていた secret というパスワードではもう読み出すことが出来ません．

もちろん正しいパスワードを入れれば読み出せますよ．

```
System.out.println("名前は"+secretData2.getName("kinoko"));
System.out.println("住所は"+secretData2.getAddress("kinoko"));
System.out.println("電話番号は"+secretData2.getPhoneNumber("kinoko"));
System.out.println("名前は"+secretData3.getName("kinoko"));
System.out.println("住所は"+secretData3.getAddress("kinoko"));
System.out.println("電話番号は"+secretData3.getPhoneNumber("kinoko"));
```

パスワードを kinoko に変更して読み出しにチャレンジです．

```
名前はデイジー
住所はサラサ・ランド
電話番号は 090-0000-0000
名前はクッパ
住所は 8-4
電話番号は 090-0000-0000
```

ほら読めた．

というわけで，key フィールドを static にすることで，パスワードを毎回設定する必要がなくなりました．

このように，同じクラスであれば同じ値を持つフィールドは，static にすると便利なことが分かったのではないでしょうか.

9.6間違えやすいパッケージと修飾子

9.6.1 import は下のディレクトリを読み込まない

他のパッケージのクラスを使う場合は，import を使いますよね．このとき，＊記号を使うことで，同一パッケージにあるクラスを全部いっぺんに import できる技があります.

```
import cellphone.address.*;
```

とすることで, cellphone.address パッケージにある全てのクラスを利用可能になります.

しかしながら，このとき注意して欲しいのが,

```
package cellphone.address.hoge.MoreClass
```

などクラスは import されたことにならないということです.

ようするに，同一パッケージにあるクラスはすべて import されますが，下位パッケージにあるクラスは import されないんですね.

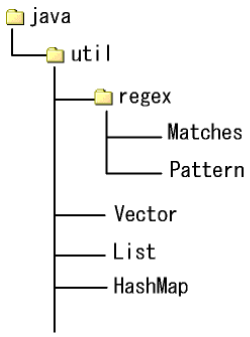

図 パッケージ構造

図のようなパッケージ構造があった場合,

```
import java.util.*
```

とすることで利用可能になるのは,

Vector, List, HashMap であり，Matches, Pattern は利用可能にならないのです.

Matches,Pattern も利用しようとしたら,

```
import java.util.regex.*
```

という import 文も追加する必要があります.

＊でまとめて import するのは便利なのですが，下位パッケージまで呼んでくれると思いがちですので，気をつけましょう.

9.6.2 かぶらない package 名

ここまでパッケージ名はわりと適当につけていましたが，実はパッケージ名をつける際には，推奨されているつけ方があります.

というのも，世界中で同じパッケージ名がいくつもあるのは望ましくないからです.

もし，仮に皆さんがすごくいいプログラムを書いて，世界中に公開したとしましょう．そのとき，そのすばらしいプログラムのパッケージを，

```
package mypackage;
```

などという名前にしてしまったとしましょう．

このとき，このプログラムを使おうと思っていた人が，もし同じ名前のパッケージを別のクラスを入れるのに利用していたらどうなるでしょうか？

混乱が起きてしまいますよね．

そこで，我々が作成するパッケージも念のため世界中に公開しても唯一のパッケージ名になるように生成するのが望ましいといわれています．

とはいえ，パッケージ名を適当につけていたのでは，どこかで使われている可能性が０ではありません．

そこで，Java ではパッケージ名の推奨されているパッケージ名のつけ方を用意しています．それは，**所属する組織のドメイン名を逆順に並べて行ったものをパッケージ名とする**，というほう方です．

たとえば，本書を出版している秀和システム株式会社でプログラムを作成したとしましょう．秀和システムのドメインは， shuwasystem.co.jp ですので，パッケージは，

```
package jp.co.shuwasystem;
```

にすることが推奨されます．

いくつかのパッケージを使い分けたいときは，

```
package jp.co.shuwasystem.cellphone.address;
```

などのように，ドメイン名を逆順にしたものの後に記述していくことになります．

もちろん，ドメイン名以後のパッケージが被らないように組織内で調整する必要はありますが・・・

ちなみに，どこの組織にも所属していない方は，お持ちのメールアドレスを逆順に並べても良いかもしれません．

```
package jp.hogehoge.mokemoke.at.tori
```

見たいな感じで．

まあ，これは推奨されている方法ではないので，あまり使わない方が良いと思いますが．

いずれにせよ，しばらくの間はプログラムを公開することもないと思いますので，あまり気にしなくても良いのですが，こういうパッケージのルールがあるということは覚えておいてください．

9.6.3 デフォルトパッケージは他から呼べない

パッケージを勉強する前は，基本的に全てのクラスを package 指定無しで作成していました．このような package 指定なしの場合を，デフォルトパッケージと呼びます．

このようなデフォルトパッケージで作成したクラスは，他のパッケージから利用するこ

とが出来ません.

```
public class NoPackage {
}
```

こんなクラスを作った場合,こうすればimportして使うことが出来そうですよね,一見.

```
import NoPackage;
```

しかしながら，これはできません.

何で出来ないのかはよく分かりませんが，とにかく出来ないのです．昔は出来たんですけどね・・・

9.6.4 アクセス修飾子は低くできない

あるクラスにpublicなメソッドを追加した後，そのクラスから派生したクラスで，メソッドのアクセス修飾子をprotectedなどにすることは出来ません.

```
public class SuperClass {
	public void check(){
		System.out.println("publicなメソッド");
	}
}
```

このように，publicメソッドcheckがあるSuperClassを継承した派生クラスExtendClassを作成します.

```
public class ExtendClass extends SuperClass {
	private void check() {
		System.out.println("protectedなメソッド");
	}
}
```

ポイントは,オーバーライドしたcheckメソッドのアクセス修飾子がprivateになっていることです.

これをコンパイルしようとすると・・・

```
ExtendClass.java:5: ExtendClass の check() は SuperClass の check() をオーバーライドできません。スーパークラスでの定義より弱いアクセス特権 (public) を割り当てようとしました。

	private void check() {
	             ^
エラー 1 個
```

というわけで，コンパイルできませんでした.

このように，アクセス修飾子は，範囲が広いメソッドをオーバーライドして範囲を狭くすることは出来ません.

public>protected>修飾子なし>privateの順で範囲が狭くなっていきますので注意してください.

ところで，なぜアクセス修飾子の範囲を狭めてはいけないのでしょうか.

それは，継承元クラスの変数に派生クラスのインスタンスを代入できることにポイントがあります.

もし，アクセス修飾子の範囲を狭めることが出来たとすると，こんなことがおきてしまいます.

```
SuperClass superClass = new ExtendClass();
superClass.check();
```

本来 SuperClass の check メソッドは public なので，どこからでも使えるため変数 superClass からも利用できるはずです．しかし，もし ExtendClass の check メソッドが private だとすると，check メソッドは使えないことになりますよね.

このような事態が生じないために，アクセス修飾子は範囲を広げることはできても，狭めることは出来ないのです.

9.6.5 getter と setter の勧め

アクセス修飾子に public や private などがあると，ついつい「全部 public にすれば楽じゃないか」と考えてしまいがちです.

しかしながら，全て public にしてしまうクラスは，設計上好ましくありません．これは，すでに 9.5 節の SecretPersonalData でなんとなく分かってもらえたと思います.

そこでフィールドは public 以外にすることが望ましいといえます．しかし，フィールドの中には外部から直接いじりたいものも数多くあります.

住所録の電話番号などは，取得もするし，電話番号が変更されればデータも書き直さなければいけませんよね．その場合，フィールドが public でないと手も足も出ないことになります.

そこで，そのような場合は，getter と setter というメソッドを作成することが推奨されます.

getter と setter は以下のようなものになります.

```
public class GetterSetterClass {
	private int value;

	public int getValue() {
		return value;
	}

	public void setValue(int v) {
		value = v;
	}

}
```

このクラスでは，private なフィールド value に対して，getValue というメソッドで，フィールドをそのまま返して，setValue というメソッドでフィールドの値を書き換えます.

利用方法としてはこんな感じ.

```
		GetterSetterClass gs = new GetterSetterClass();
```

```
gs.setValue(10);
System.out.println("value="+gs.getValue());
```

こうすることで,

```
gs.value = 10;
System.out.println("value="+gs.value);
```

と書くのと同じ効果が得られます.

では，なぜこれが嬉しいのでしょうか？

ここで,例として value の値は必ず正であるという条件を突然加えたくなったとします.

もし，後者の書き方をしていると,

```
gs.value = -10;
```

とかかれるのを防ぐことができません.

しかしながら，setter を使っていれば,

```
public void setValue(int v) {
        if(v < 0){
                return;
        }
        value = v;
}
```

とすることで，マイナスの値が value に与えられることはなくなります．これで，こんなことをしても

```
GetterSetterClass gs = new GetterSetterClass();
gs.setValue(5);
gs.setValue(-10); //マイナスで上書き
System.out.println("value="+gs.getValue());
```

結果は,

```
value=5
```

となりました．これで，value がマイナスになるのを防ぐことが出来ます.

このように，getter と setter を使うと色々便利なことができるようになりますので，是非 getter,setter を利用してください.

なお，一般的に getter と setter は,

```
get フィールド名;
set フィールド名;
```

となります.

フィールド名が name なら,

```
String getName();
void setName(String n);
```

となります.

9.6.6 private は abstract と一緒に使えない

private 修飾子は、abstract 修飾子とは同時に使えません。

```
abstract class AbstractClass{
        private abstract void abstractMethod(); //これはできない
}
```

abstract メソッドは継承先で中身を書きます。しかしながら、private メソッドにしてしまうと、そのメソッドは継承先で使うことができません。つまり、継承先で使えないのに、継承しなければいけない・・・という自己矛盾に陥ってしまいます。そのため、private と abstract は同時に使えないのです。ご注意を。