8 継承とインターフェース

8.1継承

8.1.1 継承とはなにか

オブジェクト指向の重要な考え方の一つに、継承があります。継承とは、あるクラスの機能に変更や拡張を加えて、新しいクラスを作成することを言います。

本節では、**派生クラス**と**継承元クラス**という言葉が出てきます。名前を見れば分かるとは思いますが、派生クラスとは継承して新しく作られたクラスで、継承元クラスとは派生するために利用した土台となるクラスです。なお、派生クラスのことを**サブクラス**、継承元クラスについては、**スーパークラス**という呼び方もしています。

- 派生クラス＝サブクラス
- 継承元クラス＝スーパークラス

だと覚えて置いてください。

なんとなくスーパークラスのほうが強そうだから、派生したクラスっぽいですけど、逆ですよ。

さて継承です。ここで、アドレス帳の個人データについて考えて見ましょう。個人データには、名前、住所、電話番号を記録できるように作っていました。

しかしながら、最近の携帯電話でこれだけしか情報をもてない個人データなんてありえませんよね。ほら、大事なものを忘れています。そう、E-Mail アドレスです。最近の携帯電話で E-Mail アドレスがないのは、おじいちゃん用のらくらくフォンくらいなものです。

というわけで、E-Mail アドレスも保存できる拡張クラスを作ることにしましょう。

ここで、直接 PersonalData クラスに E-Mail アドレスのフィールドを追加するという考え方もありますが、ここでは継承の勉強もかねて、これまでの PersonalData とはちょっと異なる E-Mail 管理機能つき PersonalData を作成することにします。

ぱっと思いつく方法はこんな感じですね。

```
public class PersonalData2 {
        String name;
        String address;
        String phoneNumber;
        String emailAddress;
        private int callCount;

        public PersonalData2(String n){
                name = n;
                callCount = 0;
        }



        /**
         * 一回電話したら呼び出すメソッド
         */
```

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

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

なんのことはない、PersonalData クラスにただ Email アドレスをあらわす String を追加しただけです。

しかし、これだと追加部分以外はまったく同じ動きをするわけで、ちょっともったいない気がしますよね。なんかもったいない気がします。

実は、ほとんど同じだけど、若干機能が異なるクラスを作りたい場合は、継承という作業を行うことができます。

E-Mail アドレスを扱える PersonalData の継承クラスを作るには､以下のようにします。

```
public class MailPersonalData extends PersonalData {
	String mailAddress;

	public MailPersonalData(String n) {
		super(n);
	}
}
```

さあ，これで E-Mail アドレスを扱える個人データクラスが完成しました.

え？これだけかって？

とりあえずは，これだけなんです.

本当にこれで使えるのか確認してみましょうか。

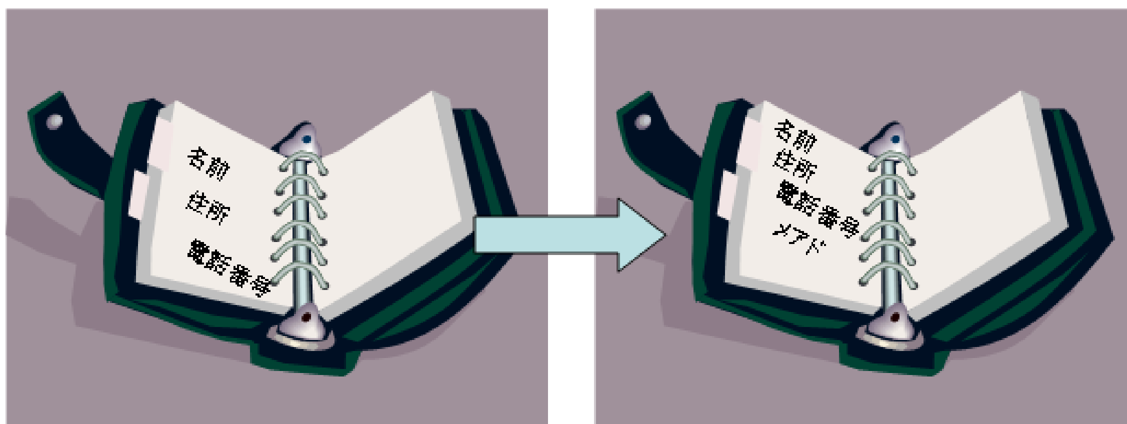
```
	static public void main(String[] args){
		MailPersonalData mpd = new MailPersonalData("マリオ");
		mpd.mailAddress = "mario@kinoko";
		System.out.println(mpd.name+"のメールアドレスは"+mpd.mailAddress+"です");
	}
```

MailPersonalData には name なんてフィールドは無いにもかかわらず利用しようとしている点に注目してください。

さあ、この実行結果は・・・

```
マリオのメールアドレスは mario@kinoko です
```

おお、ちゃんと名前も取得して E-Mail アドレスを使うこともできました。これでピーチ姫はいつでもマリオにメールを送れます。メールのやり取りする二人の姿はイマイチ見たくないような気もしますが。



実は、継承したクラスは元のクラス（スーパークラスと呼びます）の機能を基本的に使うことができるのです。

これで、PersonalData のフィールドやメソッドを二度書かなくても利用できるわけですね。

そんな便利な派生クラスを作成するには、以下のようにします。

```
public class 派生クラス名 extends スーパークラス名{
        //コンストラクタ
}
```

ポイントとなるのは，皆さんが予想したとおり extends というキーワードです.

派生クラスを作る場合は,クラス名の指定の後に extends というキーワードを書いて，更にその後に継承元のクラス名を書きます．それだけで Ok です。これで継承元クラスの機能を使うことができるクラスの完成です。

ただし、コンストラクタは継承元クラスのものを使うことができませんので、派生クラスで作らなくてはいけません。この辺のルールについては後で説明しますが、そういうものだと思ってください。

ちなみに、

```
        public MailPersonalData(String n) {
                super(n);
        }
```

このコンストラクタは、PersonalData クラスの引数が文字列のコンストラクタ、

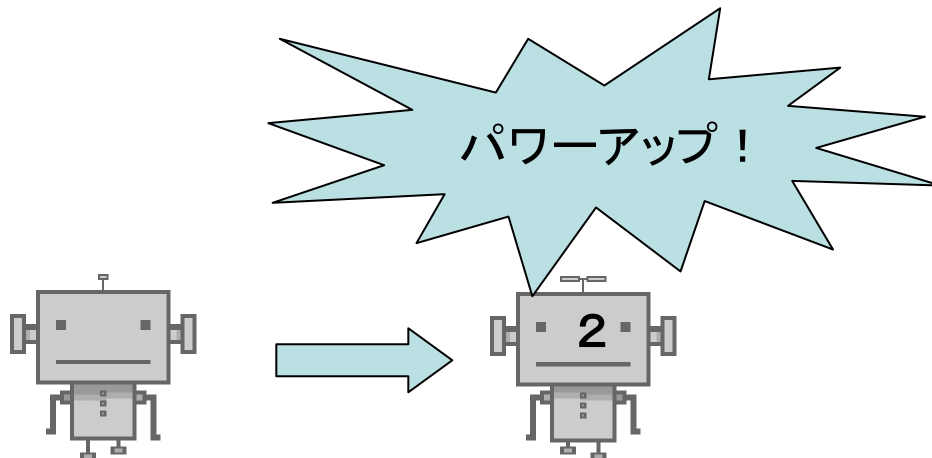
```
        public PersonalData(String n){
                name = n;
                callCount = 0;
        }
```

を呼び出すための構文です。

これについても後で説明します。

このように，あるクラスを継承することによって必要最低限の情報だけを書けばクラス

として機能するクラスを作成することができます。世の中便利ですね.



図，継承元クラスから派生クラスを作成

8.1.2 フィールドの追加

PersonalData クラスを継承した MailPersonalData クラスは何も書かなくても PersonalData クラスのフィールドを使うことが出来ました.

また、それに追加して mailAddress フィールドを追加しました。

```
public class MailPersonalData extends PersonalData {
          String mailAddress;

          //中略
}
```

このように、派生クラスでは継承元クラスに無かった新しいフィールドを自由に追加することが可能です。

さらに、E-Mail を送信した回数を記録したいと思えば、

```
public class MailPersonalData extends PersonalData {
          String mailAddress;
          int mailCount;

          public MailPersonalData(String n) {
                    super(n);
                    mailCount = 0;
          }
}
```

てな感じでさらに追加すればよいでしょう。ちなみに、初期化はコンストラクタで行っています。

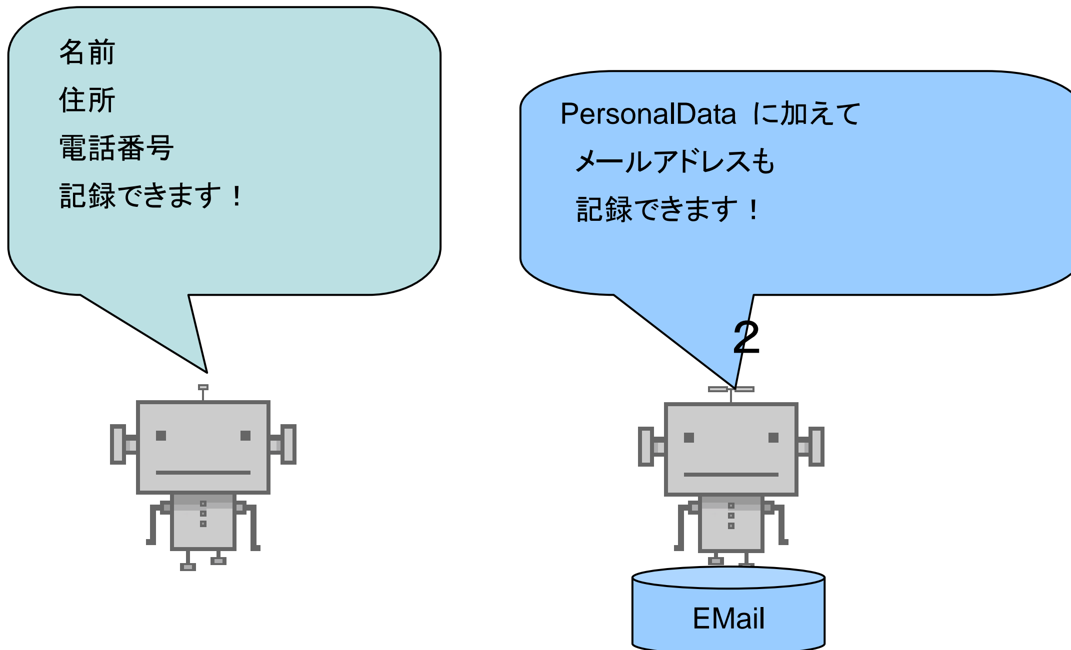


図　継承でフィールド追加

8.1.3　メソッドの追加

メソッドの追加も、フィールドと同様に簡単に行えます。

今回は、E-Mail を送ったら E-Mail 送信回数を増やすメソッドを追加してみましょうか。

```
public class MailPersonalData extends PersonalData {

        String mailAddress;
        int mailCount;

        static public void main(String[] args){
                MailPersonalData mpd = new MailPersonalData("マリオ");
                mpd.mailAddress = "mario@kinoko";
                System.out.println(mpd.name+"のメールアドレスは"+mpd.mailAddress+"です");
        }

        public MailPersonalData(String n) {
                super(n);
                mailCount = 0;
        }

        void sendMail(){
                System.out.println(name+"にメールを送りました");
                mailCount++;
        }

}
```

フィールドを追加したときと同じで、メソッドを追加すればもう利用可能になります。

```
        static public void main(String[] args){
                MailPersonalData mpd = new MailPersonalData("マリオ");
                mpd.mailAddress = "mario@kinoko";
                mpd.sendMail();
```

```
		mpd.sendMail();
		mpd.sendMail();
		System.out.println(mpd.name+"には"+mpd.mailCount+"回メールを送りました。");
	}
```

これを実行すると、

```
マリオにメールを送りました
マリオにメールを送りました
マリオにメールを送りました
マリオには 3 回メールを送りました。
```

となり、メールを送った回数をちゃんと記録することができました。

8.1.4 オーバーライド

さて、これまではあるクラスを継承した派生クラスに機能を追加していくという作業を行っていましたが、継承にはもう一つ**機能の変更**という重要な性質があります。具体的には、**同じメソッドを利用するときでも、元のクラスと継承したクラスで違う結果をもたらす**ようにすることを言います。このような作業を Java ではオーバーライドと呼びます。では、オーバーライドの例を見てみましょう。

まずは，PersonalData にこんなメソッドを追加してみましょうか.

```
	public String getContact(){
		return name+"と連絡を取るには，"+phoneNumber+"に電話をすればいいよ";
	}
```

これは，ある人物と連絡を取る方法を簡単に教えてくれるメソッドです.

```
	public static void main(String[] args) {
		PersonalData pd = new PersonalData("マリオ");
		pd.phoneNumber = "090-0000-0000";
		System.out.println(pd.getContact());
	}
```

こんなプログラムを実行すれば,

```
マリオと連絡を取るには，090-0000-0000 に電話をすればいいよ
```

となり，連絡方法が一目瞭然です.

一方で，MailPersonalData クラスに以下のメソッドを加えてみましょう.

```
	public String getContact() {
		return name+"と連絡を取るには，"+mailAddress+"にメールをすればいいよ";
	}
```

引数無しの getContact というメソッドで，これは PersonalData にあるメソッドと全く一緒です．同じ引数の同じ名前のメソッドは二つ作ることは出来ないと，オーバーロードの章で説明しましたよね.

では，これもだめでしょうか？

実は，これは許されるのです.

派生クラスでは継承元クラスと全く同一のメソッドを作成することが可能です．では,

このメソッドを呼び出したらどちらのメソッドが呼び出されるのでしょうか？
実行してみましょう．

```
static public void main(String[] args){
        MailPersonalData mpd = new MailPersonalData("マリオ");
        mpd.mailAddress = "mario@kinoko";
        mpd.phoneNumber = "090-0000-0000";
        System.out.println(mpd.getContact());
}
```

さあ，この場合電話番号とメールアドレスどちらが連絡先として表示されるでしょうか？

```
マリオと連絡を取るには，mario@kinoko にメールをすればいいよ
```

おお．
メールアドレスが表示されましたね．ということは，MailPersonalData のメソッドが呼ばれたと言うことです．

このように派生クラスでは，継承元クラスのメソッドを上書きすることが出来るのです．上書きすることで，同じメソッドを呼んでいるようで，まったく違う動作を行わせることが出来るわけですね．

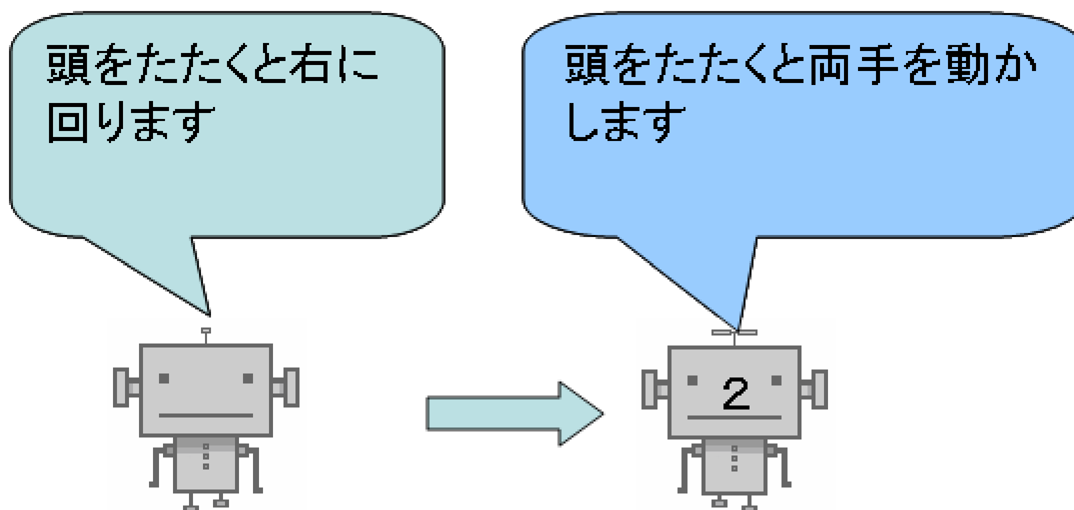


図 同じメソッドで違う効果

ちなみに、どうしてもオーバーライドしたメソッドではなくて継承元クラスのメソッドを呼び出したい場合は、次のようにします。

```
super.getContact();
```

super というキーワードを使うと、強制的に継承元クラス=スーパークラスのメソッドが呼び出されることになります。もちろん、継承元クラスでそのメソッドが定義されていなければ，呼び出すことは出来ませんが。

8.1.5 派生クラスのインスタンス作成

派生クラスは、継承元クラスの機能を持っています。そのため、派生クラスのインスタンスは継承元クラスの変数に代入することができます。

```
MailPersonalData mpd = new MailPersonalData("マリオ");
mpd.mailAddress = "mario@kinoko";
mpd.phoneNumber = "090-0000-0000";

PersonalData pd = mpd;
```

これによって、個人データとしてメールつき個人データも保存できるわけです。

これは、派生したクラスは元のクラスの一種であると考えられるからです。いうなれば、時計に目覚し機能がついた目覚まし時計が、時計として使われるようなものですね。

このとき，変数 pd は一見すると PersonalData クラスのインスタンスのように見えますが，実体は MailPersonalData のインスタンスである，というちょっと奇妙な状態になります．このようにして作成したインスタンスで，オーバーライドしたメソッドを呼び出すとどうなるでしょうか？

```
MailPersonalData mpd = new MailPersonalData("マリオ");
mpd.mailAddress = "mario@kinoko";
mpd.phoneNumber = "090-0000-0000";

PersonalData pd = mpd;
System.out.println(pd.getContact());
```

さて、マリオにはどう連絡を取ればよいでしょうね。

```
マリオと連絡を取るには，mario@kinoko にメールをすればいいよ
```

メールで連絡するのは，オーバーライドした MailPersonalData のメソッドですよね．ということは，PersonalData クラスの変数からメソッドを呼んだのに，実行されたのはオーバーライドされたメソッドだと言うことが分かります．

なぜこんなことになるかというと、pd に代入されているインスタンスは MailPersonalData ですので、そのインスタンスに対して「getContact メソッドを使うよ」という命令をすれば、当然メールを連絡手段として教えてくれるわけです。なぜなら、インスタンス自身は、自分がどの変数に代入されているか知らないからなんですね。だから、インスタンス自身はあくまでも自分が行うべきメソッドを動かすだけなわけです。

たとえわれわれが PersonalData のメソッドを使ってほしいと思っても無駄なんですね。

じゃあ, 次に MailPersonalData クラスのメソッドを呼び出してみるとどうなるでしょうか？

```
pd.sendMail();
```

さあ、実行してみましょうか。

```
SetInstanceTo.java:16: シンボルを見つけられません。
```

```
シンボル: メソッド sendMail()
場所    : PersonalData の クラス
              pd.sendMail();
                ^
エラー 1 個
```

っと，実行しようとする前に，コンパイルで失敗してしまいました.

エラーメッセージを読むと，PersonalData クラスには sendMail というメソッドがないので，実行できないよ！といっています.

つまり，継承元クラスの変数として派生クラスのインスタンスを作ってしまうと，派生クラスで新たに追加したメソッドは利用できないのです.

これは、Java 君はコンパイルするときに、変数の型しか見ておらず、実際にその変数に入っているインスタンスの型がなんであるかはチェックしないからなのです。

「pd は PersonalData 型だな。おっと、PersonalData 型には sendMail なんてメソッドはないぞ。じゃあ駄目だ！」

となるわけです。

ちょっと注意してほしいのは、コンパイルするときと、プログラムを実行するときで利用されるクラスが変わるという点です。

コンパイルのときは、変数の型が使われ、実行するときはインスタンスの型が使われるわけですね。

これは、コンパイル時にはどんなインスタンスが作成されるか分からないため、本当に使われるインスタンスの機能を使うことができないためです。

今回は、MailPersonalData 型が使われると分かっているので、PersonalData 型の変数に代入した MailPersonalData インスタンスのメソッドを使えることをわれわれは知っていますが、Java のコンパイラは、pd に代入されるのが PersonalData 型のインスタンスなのか MailPersonalData 型のインスタンスなのか分からないわけです。

したがって、コンパイラは安全策をとって PersonalData 型だと判断して MailPersonalData 型のメソッドの利用を許さないわけですね。

さて，最後に逆のパターンをやってみましょう．継承元クラスのインスタンスを派生クラスの変数に入れてみます.

```
MailPersonalData mpd = new PersonalData(″ピーチ″);
```

さあ、どうなるでしょうか。

```
SetInstanceTo.java:18: 互換性のない型
検出値  : PersonalData
```

```
期待値  : MailPersonalData
                MailPersonalData mpd = new PersonalData("ピーチ");
                                       ^
エラー 3 個
```

これまた実行する前に，コンパイルエラーが発生しました．

派生クラスの変数には継承元クラスのインスタンスを入れることは出来ませんでした．

これは，変数の型を呼び名として，作ったインスタンスを実体として考えてみれば，当たり前のことだと分かります．例えば，目覚まし時計のことを「時計」と呼ぶことはあっても，時計のことを「目覚まし時計」と呼ぶことはありませんよね．それと同じ事なんですね．

以上，まとめると以下のようになります．

- スーパークラスの変数に，派生クラスのインスタンスを代入できる
- 派生クラスでメソッドをオーバーライドしていたら，派生クラスのメソッドが使用される
- スーパークラスで定義されていないメソッドは使えない
- 派生クラスの変数に，スーパークラスのインスタンスは代入できない

ところで、こんなことができて何が嬉しいんでしょうか？役に立たない機能があってもしょうがないですよね。

そこで、うれしい例を二つほどお見せしましょうか。まず、一つ目は**配列に格納できる**という点、二つ目は**異なるクラスを同じように扱うことができる**という点です。

まず、配列への格納を考えて見ましょう。

アドレス帳を作成する際に、メールアドレスがある人とない人を区別してアドレス帳に保存する意味はあんまりありませんよね。

そのため、ひとつにまとめて配列で記録したいところです。しかしながら、二つのクラスを一つの配列にまとめることなんてできません。ということは、メールアドレスを持っている人ともっていない人でそれぞれ配列を作らなきゃいけないわけです。これはちょっと面倒くさいですね。

```
MailPersonalData[] mpdArray = new MailPersonalData[100];
PersonalData[] pdArray = new PersonalData[100];
```

しかしながら、継承関係にあるクラス同士なら二つのクラスのインスタンスを一つの配列に入れることができるわけです。こんな感じで。

```
MailPersonalData mpd = new MailPersonalData("マリオ");
mpd.mailAddress = "mario@kinoko";
mpd.phoneNumber = "090-0000-0000";
PersonalData pd = new PersonalData("ルイージ");
```

```
			pd.phoneNumber = "03-1234-5678";

			pdArray[0] = mpd;
			pdArray[1] = pd;
```

これは、**派生クラスは継承元クラスの変数として扱うことができる**、という Java のルールがあってこそのテクニックです。これで、メール機能を持っていない携帯電話しか持っていないルイージにもピーチ姫がさらわれたらすぐに連絡できますね。

また、二つ目のポイントについても確認しましょう。

```
			for(int i = 0; i < 2; i++){
				System.out.println(pdArray[i].getContact());
			}
```

これで、pdArray の最初と二つ目の要素、すなわちマリオとルイージの連絡先を確認することになります。

さあ、どうなるでしょうか？

```
マリオと連絡を取るには，mario@kinoko にメールをすればいいよ
ルイージと連絡を取るには，03-1234-5678 に電話をすればいいよ
```

おお！二つ異なるインスタンスにもかかわらずまったく同じ作業で両方のメソッドを利用することができました。

このように、継承元クラスと派生クラスで同じように扱いながら異なる結果を得ることができるわけで、これはプログラミングをする上でかなり便利な機能となります。このような機能のことを**ポリモーフィズム**といいます。

まあ、最初のうちはなかなかその便利さを味わうことはないかもしれませんが、オブジェクト指向プログラミングをやっていると、いずれこの便利さが身にしみる日が来ると思います。

8.2継承を制御する修飾子

8.2.1　クラス継承修飾子

あるクラスを作ったときに，その継承関係を制御することができます．それに利用するのが，継承修飾子です．

クラスには､public または無印という利用可能な範囲を指定する修飾子がありましたが、それ以外に継承に関係する修飾子が存在します。それが、**abstract** と **final** です。

abstract 修飾子のついたクラスは new することができず、そのクラスを元に継承したクラスを利用することが前提のクラスです。

一方 final 修飾子のついたクラスは、継承クラスを作成できないクラスです。

8.2.1.1　abstract

abstract 修飾子のついたクラスは、**抽象クラス**と呼ばれ**継承されることが前提**となりま

す。

つまり、abstract クラスのインスタンスは作ることができないのです。

例を見てみましょうか。

```
public abstract class CannotMake{
                    void showMessage(){
                    System.out.println(“作成不可能クラス”);
          }
}
```

というクラスを作成してから、このクラスのインスタンスを作成してみましょう。

```
CannotMake cannotMake = new CannotMake();
```

これをコンパイルしようとすると、

型 CannotMake のインスタンスを生成できません。

というエラーメッセージが出て、コンパイルできません。

じゃあ、こんなクラスを作っても意味がない！と思われるかもしれませんが、このクラスの場合、継承することによって使うことができるようになるのです。

```
public class CanMake extends CannotMake{
}
```

というクラスを作成し、

```
CanMake canMake = new CanMake();
canMake.showMessage();
```

を実行すると、

```
作成不可能クラス
```

と表示され、作成不可能クラスが継承されて作成可能クラスとなりました。

このように、abstract 修飾子のついたクラスは継承しないとインスタンスを作成できませんので注意して管さい。

ただし、クラス変数としては使えますよ。

```
CannotMake cannotMake = new CanMake();
```

という書き方は可能です。

8.2.1.2 final

先ほどの abstract は継承しないと使えないクラスであることを意味していましたが、final 修飾子は**継承することができないクラスである**ことを意味します。

つまり、final と書いてあるクラスを継承したクラスは作れないのです。

簡単な例を見てみましょうか。

```
final class FinalClass{
}
```

この FinalClass クラスは、final 修飾子がついているので、継承できません。ためしに継承しようとしてみましょうか。

```
class ExtendClass extends FinalClass{
}
```

これをコンパイルしてみようとすると・・・

```
型 ExtendClass は final クラス FinalClass をサブクラス化できません。
```

というわけで、継承できませんでした。

final は誰にも継承されたくないようなクラスを作った場合に利用します。まあ、当分の間は皆さんが使うことはないと思いますが、こういう修飾子もあるということを覚えて置いてください。

ちなみに、文字列を扱う String はクラスの一種ですが、String クラスは final 修飾子がついていますので、継承することができません。暇な時間があったら継承できないことを確認してみてください。

ところで、abstract は継承することが前提のクラスであることを意味し、final は継承不可能であることを意味するというお話をしました。

ということは、両方同時に使うとどうなるのでしょうか・・・？

試し見ましょうか。

```
abstract final class AbstractFinalclass{

}
```

さあ、コンパイルしてみましょう。

```
Test.java:14: 修飾子 abstract と final の組み合わせは不正です。
abstract final class AbstractFinalclass{
                     ^
エラー 1 個
```

ありゃりゃ。予想通り怒られてしまいました。まあ、両方の修飾子を一度につけたくなるようなことは好奇心以外では起こりえないのですが、間違えてつけてしまうことがないようにコンパイル時に注意してくれるんですね。Java ってやさしいですね。

まあ、このやさしさに甘えず皆さんは abstract と final を同時にクラスにつけることがないように注意してください。

ちなみに、public と abstract、あるいは public と final は組み合わせることが可能です。

8.2.2 メソッド・フィールド継承修飾子

8.2.2.1 abstract

abstract 修飾子は、クラスにつけた場合は継承することが前提のクラスという意味でしたが、メソッドにもつけることができます。メソッドにつけた場合、継承クラスで必ずオーバーライドしなければいけないメソッドということになります。

たとえば、

```
abstract class AbstractClass{
        abstract void abstractMethod();
}
```

のように記述します。

ここでポイントなのが、abstract メソッドには具体的な動きを記述しないという点です。通常の場合、メソッドを書いたら、{}で囲んでそのメソッドが呼ばれたときの動作を記述しますよね。abstract メソッドの場合はそれがありません。つまり、このメソッドが呼ばれたときにどういう動きをするかは、継承先のクラスにすべて任されているのです。

ところで、もしこのクラスのインスタンスを作成して abstractMethod を呼んでしまったらどうなるでしょうか？動作が記述されていないから、プログラムは何をすればよいのかわからなくなってしまいますよね。

そのため、**abstract メソッドを持つクラスは abstract クラスになる**という決まりがあります。もし、abstract メソッドを持っているクラスが abstract クラスでないと、コンパイル不可能になりますのでご注意ください。

なお，abstract はフィールドにはつけることができません。メソッドにしか使えませんのでご注意ください。

8.2.2.2 final

final 修飾子は、メソッドとフィールドでそれぞれ意味が異なります。

まず、フィールドに final をつけた場合変更不可能な変数となります。通常のフィールドはインスタンス内でメソッドを利用したり、private 以外の修飾子がついていれば、他のクラスからでも変更することが可能ですが、final がついているフィールドは変更できません。

たとえば、以下のようなクラスを作っていましょうか。

```
class Husband{
        final int wife = 1;
}
```

これで、wife 変数は 1 で固定され、それ以外の値に変化させることはできません。これで奥さんは一人きり。浮気はもうできません。

もし仮に、

```
class Husband{
        final int wife = 1;
        FinalClass(){
                wife = 2;
        }
}
```

とすると、

```
Husband.java:6: final 変数 wife に値を代入することはできません。
                data = 2;
                ^
エラー 1 個
```

となり、コンパイルエラーとなります。奥さんを二人持つことは永遠にできそうもありません。これで鬼嫁も安心。

ちなみに、final 修飾子で最初に値を与えないこともできます。

```
class Husband{
        final int wife;
}
```

この場合、wife の値を必ずコンストラクタで指定する必要があります。

```
class Husband{
        final int wife;
        FinalClass(){
                if(日本){
                        wife = 1;
                }
                else if(アラビア){
                        wife = 4;
                }
        }
}
```

これで日本だと wife は一人が限界ですが、アラビアなら一夫多妻制で 4 人の奥さんを持つことができるようになりました。男の夢がここに実現します。すごい。

ただし、一度コンストラクタで一度していたらもう二度と変更できませんので、ご注意ください。

次に、メソッドに final をつけた場合です。この場合、そのメソッドはオーバーライド不可能となります。

クラスに final をつけた場合継承不可能なクラスになるように、メソッドはオーバーライド禁止となるのです。

ためしに、こんなクラスを作ってみましょうか。

```
public class FinalClass {
        final void method(){

        }
}
```

さらに、これを継承したクラスを作って見ます。

```
class ExtendClass extends FinalClass{
        void method() {

        }
}
```

これをコンパイルしようとすると

```
FinalClass.java:17: ExtendClass の method() は FinalClass の method() をオーバー
ライドできません。オーバーライドされたメソッドは final です。
        void method() {
             ^
エラー 1 個
```

というわけで、final なメソッドはオーバーライドできないことが分かったと思います。

以上で継承の説明は終わりです。

8.3インターフェースと実装

8.3.1　インターフェースとはなにか

Java にはインターフェースという機能があります．これは，**あるクラスに対して行うことが出来る操作**を表現するための機能です．

簡単に言うと，あるクラスがどんなメソッドを持っているか，どんなメソッドを持っていなければいけないかが書いてあるものです．

インターフェースの例を現実社会のものに置き換えて考えて見ましょう．

ここで，カメラとデジタルカメラの関係について考えて見ましょう．それぞれ，シャッターを押して写真を撮るという操作を行なうことができますよね．

これは，カメラの会社が違っても基本的に同じ操作をすれば同じ効果が得られます．操作を統一しておけば，利用者が混乱しないですみますよね．

このように，統一された機能を持っていることを保障するためにインターフェースという考え方があるのです．

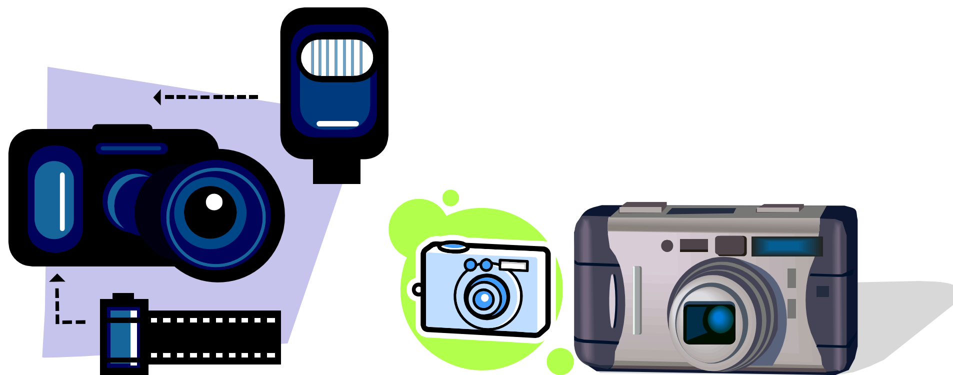


図 機能は違っても写真を撮る操作は皆同じ

Java でも，同じインターフェースを持ったクラス同士は，区別しないで扱うことが出来るのです．A 社のカメラと B 社のカメラで使い方に差がないように，携帯インターフェースを実装したクラスは同じ使い方ができるのです．

ところで，写真を撮るという機能は，カメラだけについているものではありませんよね．最近は携帯電話で写真をとる人のほうが多いくらいじゃないでしょうか？

図 カメラ・・・？

携帯電話出写真とをとる場合も，ボタンを教えて写真を撮るという操作にはなんら変わりはありません.

しかしながら，携帯電話はカメラではありませんよね．もちろん，カメラとしての機能は持っていますが，もともとは電話です.

ということは，携帯電話をクラスで表そうとしたらカメラクラスを継承するよりも，電話クラスを継承したほうが良いでしょう．しかし，そうすると今度はカメラ機能を持っていることが表現できなくなる・・・

このように，Java ではひとつのクラスしか継承できないために，全ての機能を網羅して表現することが出来ない場合があります.

このような状況を表現する方法がインターフェースです.

インターフェースとは，**同じ操作ができるもの**をまとめたものと考えてください.

8.3.2 Java におけるインターフェースの必要性

携帯電話を使っているとき，いろんなデータを入れすぎてメモリ不足になったことはありませんか？携帯電話で使えるデータ量というのは限られていますよね．そこで，アドレス帳のデータがどのくらいのデータ量を使っているのか調べることにしましょう.

そこで，PersonalData クラスに以下のようなメソッドを追加しました.

```
public int getSize(){
        int s = 0;
        s += name.length();
        if(address != null){
                s += address.length();
        }
        if(phoneNumber != null){
                s += phoneNumber.length();
        }
        return s;
}
```

これは，名前やアドレス情報を記述するために必要なメモリ量を全部足した値を返してくれるメソッドです(*正確にはこれでは計算し切れませんが・・・*).

これは，MailPersonalData クラスでは，Email アドレスも記録していますので，ちょっ

と変更してこうなります．

```
	public int getMemory(){
		int s = super.getMemory();
		if(mailAddress != null){
			s += mailAddress.length();
		}
		return s;
	}
```

まあ，難しいことは余り考えずに，これでメモリ量が計算できると思ってください．

さて，これでアドレス帳に記録されている分はメモリ量が計算できそうですね．

```
		//pdArrayは個人データが入った配列
		PersonalData[] pdArray = new PersonalData[2];
		int size = 0;
		for(int i = 0; i < pdArray.length; i++){
			size += pdArray[i].getMemory();
		}
		System.out.println("総メモリ容量は"+size);
```

とすれば，各 PersonalData(または，MailPersonalData)のインスタンスの getMemory メソッドを呼び出して，全個人データで利用するメモリ容量が分かります．

```
総メモリ容量は 44
```

というわけで，どのくらいのデータを使ったかが分かりました．

しかしながら，携帯電話でメモリを使用するのはアドレス帳だけではありません．着メロもメモリを使用しますし，写真データもメモリを使用します．ど～でもいい写真を撮りすぎて，肝心なときに写真を撮れなくなった思い出は誰に出もあるのではないでしょうか．

そこで，これらのデータについてもメモリをどのくらい利用するのか調べておかないと，携帯電話のメモリが残り何メガくらいあるのか把握できなくなってしまいます．

しかし，いちいち写真データの容量，着メロの容量と別々に計算するのは面倒くさいですよね．PersonalData と MailPersonalData と同様に，全データを一度に配列に入れて getMemory メソッドを使って一気に計算してしまいたい物です．

```
		for(int i = 0; i < dataArray.length; i++){
			size += dataArray[i].getMemory();
		}
```

ですが，このとき，dataArray はどんな配列ならいいでしょうか？

一つアイデアとして考えられるのは，getMemory メソッドを持ったクラスを作成して，全データがそのクラスを継承するという手ですね．仮にそのクラスを MobilePhoneData とすれば，

```
public abstract class MobilePhoneData {
	abstract public int getMemory();
}
```

となります．メモリの計算方法はデータによって異なるので，abstract クラスにして，どう計算するかは派生クラスで作成する点に注意してください．

さて，このクラスを継承すれば，メモリ量を計算するという目的は達成できます．しか

し，着メロデータなどが，MobilePhoneData を継承するのはどうなんでしょう．どちらかといえば，音楽データクラスを継承したいところです．

しかし，Java では二つのクラスを同時に継承することは出来ません．こりゃ困った．

そんなときに，interface 機能が役立ちます．

8.3.3 interface の構文

まずは，使用メモリ量を取得できるクラスに付加する interface，MemoryCountable を作ってみましょう．

```
public interface MemoryCountable {
        int getMemory();
}
```

これでインターフェースの作成は終わりです．簡単ですね．

これを作ったところで，まだ何が役に立つのか分かりませんが，とにかくこういう interface を作ったことを覚えておいてください．

この interface を持っているクラスは，必ず getMemory というメソッドを持っていることが保証されます．どう役に立つのかは，次節で解説します．

インターフェース宣言の基本構文は以下のとおりです。

```
public interface インターフェース名 {
        返値 メソッド名(引数);
}
```

クラスのように宣言しますが，class と書かれていた部分が interface になり，メソッドの宣言でその中身を書かないのがポイントです．クラスではメソッドの中身を書かないと，abstract クラスになってしまいますが，interface ではどんなときもメソッドの中身は書きません。中身はすべて実装クラスで書くことに決まっています。インターフェースはあくまでも「こういう機能があるよ」ということだけを書いておき、その中身については実装クラスにすべて任せてしまうという、ちょっと投げやりな機能なのです。

また，フィールドを作成できないのもインターフェースの特徴の一つです．必要なフィールドはやはり実装クラスで記述することになります．

簡単に言えば，全部のメソッドが abstract でフィールドを持っていないクラスを作るような物だと思って下さい．

8.3.4 実装クラスの作成

interface では持っている機能を並べます。ただし、その中身については書きません。機能については、インターフェースを組み込んだ**実装クラス**で書くことになります。

実装クラスとは，インターフェースを持っているクラスのことを言います．

ここでは，着メロを記録するクラスが MemoryCountable を実装しているとして，記述

しましょう.

今回作る ChakuMero クラスは着メロをあらわしますから、きっと「音楽」を継承しているでしょう。さらに，どのくらいメモリを使うかを調べるための機能があることが保証されています.

そこで、まず音楽を表す Music クラスを継承し，かつインターフェース MemoryCountable を実装していることを表現します.

なお，Music クラスについては，今回は割愛しますよ.

```
public class ChakuMero extends Music implements MemoryCountable{

}
```

しかし，これでは保障されるはずの getMemory メソッドが存在しません．これでは，約束が違います.

コンパイルしようとすると，

```
ChakuMero.java:3: ChakuMero は abstract でなく、MemoryCountable 内の abstract メソッド getMemory() をオーバーライドしません。
public class ChakuMero extends Music implements MemoryCountable{
       ^
エラー 1 個
```

と，怒られてしまいます.

そこで，ちゃんと getMemory メソッドを実装しましょう.

```
public class ChakuMero extends Music implements MemoryCountable{

        public int getMemory() {
                return musicSize;
        }

}
```

musicSize は音楽データの容量だと思ってください.

このようにメソッドを追加することで，ChakuMero クラスは利用するメモリ容量を返すことができるようになりました.

このように、MemoryCountable インターフェースを実装した時点で、そのクラスには必ず MemoryCountable インターフェースで指定されている getMemory メソッドがかかれていることが保証されます。

なお，あるインターフェースの実装クラスを作成する構文は以下のとおりです.

```
class クラス名 implements インターフェース名{

}
```

また，インターフェースは複数実装できるので，複数のインターフェースを実装したク

ラスを作成する場合は,

```
class クラス名 implements インターフェース1, インターフェース2{

}
```

のように,「,」区切りで羅列していくことになります.

もちろん,インターフェース１,インターフェース２で指定されているメソッドは全部記述してある必要がありますけど.

8.3.5 インターフェースのルール

さて、具体例をあげてインターフェースについて説明してきましたが、ここでインターフェースのルールを覚えていきましょう。

- インターフェースには機能を羅列する
- 機能の中身は記述しない
- 機能はすべて public 型
- フィールドを持つことはできない
- 定数を持つことができる
- インターフェースを継承することができる
- インターフェース型変数を作れる

8.3.5.1 インターフェースは機能を羅列する

インターフェースは基本的に実装クラスがどのようなメソッドを持っているかをあらわす機能です。

たとえば、Java に最初から付随している JavaAPI には List というインターフェースが用意されています。この List は高度な配列を扱うクラスで実装されているインターフェースなのです。

この List には、size というメソッドが存在します。これは、配列の長さを返すメソッドです。あるいは、配列に値を挿入する add というメソッドなどが定義されます。したがって、List インターフェースを実装しているすべてのクラスでは、配列の長さを調べる size メソッドが必ず存在することが保証されることになります。

このように、そのインターフェースを実装したクラスに特定のメソッドがあることを保証する、そのためにインターフェースは存在していると思ってください。

8.3.5.2 機能の中身は記述しない

インターフェースはあくまでもどんな機能を持っているかを保証させるためにあるため、その中身は記述しません。したがって、クラス側でどう中身を書くかは決めることに鳴ります。

ちなみに、インターフェースで決められるのは、メソッドの引数と返り値だけです。し

たがって、そのメソッドの動作は適当に決めることができてしまいます。たとえば、Horolインターフェースで adjust というメソッドを定義した場合、当然時計の時刻を合わせる機能だと期待しますが、必ずしも時刻を合わせるメソッドであるとは限りません。これはクラスを作成する人にすべてゆだねられています。

そのため、もし皆さんが今後あるインターフェースを実装するようなクラスを作成しなければいけない場合は、必ずそのメソッドの条件に合った機能を実装するように注意しましょう。

8.3.5.3　機能はすべて public

インターフェースで指定する機能は、すべて pulbic 型であるという決まりがあります。ようするに、どのクラスからでもその機能を利用可能なのです。

そのため、インターフェースを書くときにメソッドの前に public を書いても書かなくてもかまわないというルールがあります。MemoryCountable インターフェースの場合ですと、

```
public interface MemoryCountable {
	ublic String getMemory();
}
```

と書いても，

```
public interface MemoryCountable {
	String getMemory();
}
```

と書いても，インターフェースの機能は変わりません.

筆者はいちいち public と書くのが面倒なので下の書き方をしていますけどね.

8.3.5.4　フィールドを持つことはできない

インターフェースは機能を記述したものです。したがって、メソッドの中身を掛けないのと同様に、フィールドを持つことはできません。

```
public interface MemoryCountable {
	String getMemory();
	int parameter;
}
```

というようなことはできません。実際にこのような記述をすると、コンパイルなどはとおりますが、自動的に後述する定数となって扱われることになります。

一つ一つのインスタンスで異なる値を持つことができる、フィールドが必要な場合は、必ずインターフェースを実装したクラスで定義してください。

8.3.5.5　定数を持つことができる

インターフェースはフィールドを持つことはできません。が、定数を持つことはできます。

定数とは、どのインスタンスでも同じ値を持ち、かつ変更不可能な値のことです。static かつ final なフィールドのことを言います。詳しいことは後ほどの章で説明します.

ここでは、例として最大メモリ量を記録してみましょう。

```
public interface MemoryCountable {
	String getMemory();
	static final int MEMORY_SIZE = 1000000;
}
```

このようにすることで，MEMORY_SIZE という定数を指定することが出来ます.

この値は，一見変数のように見えますが，その値を変更することが出来ません．最初に指定した 1000000 以外の値を取ることは出来ないのです.

なお、インターフェースで定義するメソッドがすべて自動的に public になるように、フィールドは書けば自動的に定数になります。

つまり、

```
public interface MemoryCountable {
	String getMemory();
	int MEMORY_SIZE = 1000000;
}
```

と書いても、自動的に public static final であるとして処理されます。これは結構重要ですので覚えて置いてください。

なお，通常定数はすべて大文字で書くという決まりがありますので，インターフェースに定数を付加するときは，大文字で書くようにしましょう.

8.3.5.6 インターフェースの継承

インターフェースはクラスと同じように、継承を行うことができます。インターフェースの継承は，通常のクラスの継承と同じように extends を使います.

ためしに，先ほどの MemoryCountable インターフェースを拡張したインターフェースを作成してみましょう.

```
public interface ExMemoryCountable extends MemoryCountable{
	void compression();
}
```

このように，書くことで，MemoryCountable で，かつ compression，つまり圧縮機能を持ったメモリデータのインターフェースを作ることができます.

ちなみに，圧縮機能がどういうものかは実装クラスに任されますが.

なお,このインターフェースを実装したクラスは compression と getMemory の両方のメソッドを完備していないと駄目ですよ.

```
public class Picture implements ExMemoryCountable {

	int memorySize;

	public void compression() {
		//圧縮操作
	}

	public int getMemory() {
		return memorySize;
	}
}
```

インターフェースの拡張は，以下の構文に従って記述します．

```
public interface インターフェース名 extends 元となるインターフェース1，元となるインターフェース2{
        追加メソッド();
}
```

面白いことに，インターフェースの継承は複数のインターフェースを同時に継承することができます．クラスが一つのクラスしか継承できないのとは異なることを覚えて置いてください．

タイプ	継承可能数
クラスの継承	一つしか継承できない
インターフェースの継承	複数の継承が可能

8.3.5.7　インターフェース型変数が作れる

最後に紹介するのは、インターフェース型変数のお話です。

これは、インターフェースの機能の中でもかなり重要な機能の一つなので是非覚えておいて活用してください。

派生クラスは，継承元クラスの変数に代入できましたが，同様にインターフェース型の変数を作成することができます．

つまり，

```
MemoryCountable mc = new PersonalData("マリオ");
```

ということが出来るわけです．

インターフェースは，変数という点からはクラスとほとんど同じように扱われます．ですので，単に変数を作るだけではなく，配列を作成することも出来ます．

```
MemoryCountable[] mcArray = new MemoryCountable[100];
mcArray[0] = new PersonalData("ルイージ");
mcArray[1] = new ChakuMero("マリオのテーマ");
```

これによって，ぜんぜん違うクラスを継承しているデータであっても，同じ配列に代入することが出来ました．

このとき，注意しなければいけない点として，mcArray 配列の要素は，中身がなんであったとしても，MemoryCountable インターフェースで指定されているメソッド以外は使うことが出来なくなっているという点です．

本来，PersonalData クラスには，call というメソッドがありましたよね．しかしながら，

```
mcArray[0].call();
```

という使い方は出来ません．

これは，継承元クラスの変数に派生クラスの変数を代入したときに，派生クラスのメソッドが使えないのと同じように，インターフェース型の変数に実装クラスを代入した場合，インターフェースに指定されている機能だけ使うことが出来，実装クラス特有の機能は使

うことが出来ません.

8.3.6 クラスとインターフェースの違い

クラスとインターフェースは比較的よく似ていますが，いくつか決定的な違いがあります．特に，継承と実装には大きな違いがあります.

それをまとめておきましょう.

表 1 クラスとインターフェースの違い

クラス		インターフェース
extends	継承・実装キーワード	implements
一つしか継承できない	派生・実装クラス	複数を実装可能
基本的に，中身を記述する	メソッド	中身を記述しない
持てる	フィールド	持てない
基本的に作成可能	インスタンス	作成不可能
作成可能	変数	作成可能

8.3.7 インターフェースの利用例

ここまで説明してきて，インターフェースをどう作成するかは分かっていただけたと思いますが，どう利用すればよいのかは分からなかったのではないでしょうか？

そこで，インターフェースを利用した例を示したいと思います.

今回のミッションは，携帯電話内にある全データの総メモリ量を取得して，残りのメモリに比べてどの程度残っているかを調査することとします.

携帯電話に登録されているデータは以下の３種類としましょう.

- アドレス帳
- 写真データ
- 着メロ

これらがそれぞれ配列に収められているとします.

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

        int maxMemorySize = 1200;

        MemoryCountable[] allDataArray = new MemoryCountable[dataNum];
        //memoryCountableArray は全データを保有

        int memorySize = 0;
        for(int i = 0; i < allDataArray.length; i++){
                memorySize += allDataArray[i].getMemory();
        }
        System.out.println("全データの容量は"+memorySize+"です．");
```

```
			double useRate = (double)memorySize/maxMemorySize;
			System.out.println("全データの"+useRate*100+"%利用しています．");
		}
```

ここで，ポイントは全データが allDataArray に格納されているという点です．このような配列を一つ作っておくことで，一回の for ループを使うだけで全メモリを簡単に取得することが出来ます．

```
全データの容量は 1540 です．
全データの 128.33333333333334%利用しています．
```

もちろん，それぞれのデータの配列でメモリを一つ一つ調べていってもいいんですが，こうするとプログラムを書くときにちょっと楽ですよね．

8.4間違えやすい継承

8.4.1 継承元クラスのメソッドから派生クラスのメソッドを呼ぶ

派生クラスでは，継承元クラスのメソッドをオーバーロードできるということを学びました．

つまり，あるメソッドがオーバーロードされていれば，呼ばれるのはオーバーロードされたメソッドという事になるわけです．

さて，ここで問題継承元クラスからオーバーロードされたメソッドを呼んだらどうなるでしょうか？呼ばれるのは，継承元クラスのメソッドでしょうか？それとも，派生クラスのメソッドでしょうか？

確かめてみましょうか．

まず，こんなクラスを作ってみましょう．このクラスは，showMessage メソッドを呼ぶと，getMessage メソッドで返してくる文字列を表示します．

```
public class SuperClass {

		void showMessage(){
			System.out.println(getMessage());
		}

		String getMessage(){
			return "継承元クラスのメソッドが呼ばれました";
		}
}
```

このクラスを使ってこんなプログラムを実行してみましょう．

```
			SuperClass sc = new SuperClass();
			sc.showMessage();
```

その結果はこうなります．

```
継承元クラスのメソッドが呼ばれました
```

まあ，これは当然ですね．

では，次にこんな派生クラスを作ってみます．

```
public class ExtendedClass extends SuperClass {

	String getMessage(){
		return "派生クラスのメソッドが呼ばれました";
	}
}
```

この派生クラスでは，showMessage メソッドがありませんので，showMessage メソッドを呼ぶと，継承元クラスの showMessage メソッドが利用されることになります．では，showMessage メソッドは，継承元クラスのメソッドと，派生クラスのメソッドどちらを呼ぶのでしょうか？

確認のため，次のプログラムを実行してみます．

```
		ExtendedClass ec = new ExtendedClass();
		ec.showMessage();
```

この結果は，どうなるでしょうか？

```
派生クラスのメソッドが呼ばれました
```

おっと．呼ばれる getMessage メソッドが変化しました．このように，継承元クラスの別メソッドから呼ばれたとしても，オーバーロードされたメソッドは，必ずオーバーロードされたのメソッドを呼び出すことになります．

したがって，SuperClass クラスを作成しているときに，getMessage を呼び出せば必ず「継承元クラスのメソッドが呼ばれました」という文字列が返ってくると期待していると，大変なことになるかもしれませんよ．もし，いたずら好きな人が，こんなクラスを作ってしまっていたら・・・

```
public class TrickClass extends SuperClass {
	String getMessage(){
		return "致命的なバグが発生しました．あなたのパソコンが壊れました";
	}
}
```

そして，こっそりと SuperClass の変数にこの TrickClass のインスタンスを混ぜてしまったらどうなるでしょうか．

```
		//こっそり忍ばせる
		SuperClass sc2 = new TrickClass();

		・・・

		//気づかずに使ってしまった！
		sc2.showMessage();
```

そうすると，

```
致命的なバグが発生しました．あなたのパソコンが壊れました
```

という世にも恐ろしいメッセージが表示されてしまうことになります．

メソッドを作成するときはオーバーロードされる危険性を考えながら作成しましょう．

8.4.2 フィールドはオーバーロードできない

メソッドはオーバーロードすることが出来ます．では，フィールドはオーバーロードできるのでしょうか？

答えは簡単．

念のため試してみましょうか．

前節で試した getMessage メソッドをやめて，直接 message というフィールドを作成したとします．

```
public class SuperClass {
        String message = "継承元クラスの変数";

        void showMessage(){
                System.out.println(message);
        }
}
```

このクラスに対して，

```
                SuperClass sc = new SuperClass();
                sc.showMessage();
```

これを実行すると・・・

```
継承元クラスの変数
```

というわけで，まずはちゃんとフィールドの値をそのまま出力していることが確認できました．

では，次に，

```
public class ExtendedClass extends SuperClass {
        String message = "派生クラスの変数";
}
```

こんな派生クラスを作成します．メソッドのようにオーバーロードされていれば，

```
                ExtendedClass ec = new ExtendedClass();
                ec.showMessage();
```

このプログラムで出力されるのは「派生クラスの変数」のはずですが・・・

```
継承元クラスの変数
```

実行すると，やっぱり継承元の変数が表示されました．

つまり，継承元で呼ばれるフィールドは，継承元のフィールドであり，派生クラスのフィールドにはなりません．

継承元クラスにあるフィールドと同じ名前のフィールドを追加することもできますが、使い分けをするためにちょっと工夫が必要となってしまいます。

それに利用するキーワードが this と super です．

その使い方を見るために，ExtendedClass に以下のようなメソッドを追加して実験してみましょう．

```
public class ExtendedClass extends SuperClass {
        String message = "派生クラスの変数";
```

```
	void showFieldMessage(){
		System.out.println(message);

		System.out.println(this.message);
		System.out.println(super.message);
	}

}
```

さあ，この showFieldMessage メソッドを呼び出すと，どう出力されるでしょうか？

```
		ExtendedClass ec = new ExtendedClass();
		ec.showFieldMessage();
```

この結果を見てみると・・・

```
派生クラスの変数
派生クラスの変数
継承元クラスの変数
```

このようになりました.

まず，普通にフィールドを呼び出した場合，派生クラスのフィールドが呼ばれます.

次に，this キーワードをつけても，やはり派生クラスのフィールドが呼ばれます.

最後に，super キーワードをつけると，継承元クラスのフィールドが呼ばれます.

このように，this または super をつけることで，継承元，派生先どちらのフィールドを呼ぶかを制御することが可能となります.

なお，派生クラスにないフィールドの場合，キーワードに関係なく継承元クラスのフィールドが呼ばれます.

まとめると，このようになります.

キーワード	派生クラスにフィールドがある場合	派生クラスにフィールドがない場合
キーワードなし	派生クラスのフィールド	継承元クラスのフィールド
this	派生クラスのフィールド	継承元クラスのフィールド
super	継承元クラスのフィールド	継承元クラスのフィールド

8.4.3 継承とコンストラクタの問題

継承を行った場合、コンストラクタを作成するのには少し注意が必要です。

派生クラスのコンストラクタはかなり複雑なルールが存在しますので，そのルールをきちんと守ったプログラムを作る必要があります.

ここでは，場合わけして詳しく説明して行きたいと思います.

8.4.3.1　継承元クラスにコンストラクタがない場合

継承元クラスにコンストラクタがない場合は，非常に単純です．

この場合は，派生クラスでは自由にコンストラクタを作成することが可能です．

下記のクラスは，この条件に当てはまる物です．

```
public class NoConstractor {

}
```

このようなクラスから，派生したクラスではどのようなコンストラクタでも自由に作ることが出来ます．

また，**コンストラクタなしのクラス**を作成することが出来ます．

このコンストラクタ無しのクラスを作れるところがポイントです．

8.4.3.2　継承元クラスに引数無しのコンストラクタがある場合

継承元クラスに引数なしのコンストラクタがある場合も，比較的単純です．この場合も，派生クラスでは自由にコンストラクタを作成することが可能です．

このとき，注意して欲しいのが，継承元クラスに引数無しのコンストラクタさえあれば，引数有りのコンストラクタがあっても，派生クラスでは自由にコンストラクタを作ることが出来るという点です．

```
public class VoidConstractor {

        public HasVoidConstractor() {
                System.out.println("引数なしで継承元クラスを作成");
        }


        public HasVoidConstractor(String arg) {
                System.out.println("引数["+arg+"]で継承元クラスを作成");
        }

}
```

このようなクラスから派生したクラスは，自由にコンストラクタを作ることが出来ます．

また，**コンストラクタなしのクラス**を作成することが出来ます．

なお，継承元クラスに引数無しのコンストラクタがあって，派生クラスにコンストラクタがなかった場合，自動的に継承元の引数無しコンストラクタが呼ばれることになります．

たとえ，以下のような派生クラスを作成したとします．

```
public class ExVoidConstractor extends VoidConstractor{
}
```

このクラスのインスタンスを以下のように作成しましょう．

```
                ExVoidConstractor evc = new ExVoidConstractor();
```

この場合，

```
引数なしで継承元クラスを作成
```

となり，継承もとクラスの引数無しのコンストラクタを自動的に呼び出します.

さて，ここで派生クラスにコンストラクタを作成してみましょう.

```
public class ExVoidConstractor extends VoidConstractor{
          public ExVoidConstractor() {
                    System.out.println("引数なしで派生クラスを作成");
          }
}
```

さて，このクラスのインスタンスを作成するとどうなるでしょうか.

```
引数なしで継承元クラスを作成
引数なしで派生クラスを作成
```

あれ？継承元のコンストラクタまで呼ばれてしまいました.

実は，派生クラスを作成すると，自動的に**継承元クラスのコンストラクタが呼ばれる**というルールが Java には存在するのです.

継承元クラスにコンストラクタが無かった場合何も表示されなかったのは，単に何もしないデフォルトコンストラクタが呼ばれたため継承元のコンストラクタが呼ばれたことに気づいていなかっただけなんですね.

派生クラスを作成すると，継承元クラスのデフォルトコンストラクタが呼ばれるということを忘れずにいてください.

ちなみに，派生クラスのコンストラクタよりも先に継承元のコンストラクタが呼ばれます．順番が重要な処理を行う場合は，一応注意しておいてください.

8.4.3.3　継承元クラスに引数ありのコンストラクタのみあった場合

継承元に引数ありのコンストラクタしかなかった場合，そのクラスから派生したクラスは**コンストラクタ無しで作成することが出来ません**.

具体例を見てみましょう.

```
public class SuperClass {
          public SuperClass(String arg) {
                    System.out.println("引数["+arg+"]で継承元クラスを作成");
          }

}
```

まず，こんな継承元クラスを作成します．次に，

```
public class ExtendedClass extends SuperClass {
}
```

こんなコンストラクタ無しの派生クラスを作成します.

これをコンパイルしようとすると・・・

```
ExtendedClass.java:3: シンボルを見つけられません。
シンボル: コンストラクタ SuperClass()
```

```
場所　　：SuperClass の クラス
public class ExtendedClass extends SuperClass {
        ^
エラー 1 個
```

このようにエラーが発生してコンパイルに失敗します．

派生クラスを作成すると，必ず継承元クラスのコンストラクタが呼ばれるルールから，派生クラスのインスタンスを作る際に，継承元クラスの引数無しコンストラクタを呼び出そうとします．このとき，**引数有りのコンストラクタがあった場合，暗黙のデフォルトコンストラクタは呼ばれない**というルールがあったことを思い出してください．

つまり，継承元クラスの引数無しコンストラクタが存在しないのに呼び出そうとしたと判断され，コンパイルエラーが発生するのです．

派生クラスを作成するときに継承元クラスのデフォルトコンストラクタを呼び出さないようにするためには，明示的にコンストラクタを呼び出すようにしなければ行けません．

そこで，派生クラスをこんな風に改良しましょう．

```
public class ExtendedClass extends SuperClass {

        public ExtendedClass() {
                super("継承元コンストラクタ呼び出し");
                System.out.println("引数なしで派生クラスを作成");
        }
}
```

これで，派生クラスを作成してみます．

```
                ExtendedClass ec = new ExtendedClass();
```

さあ，実行してみましょう．

```
引数[継承元コンストラクタ呼び出し]で継承元クラスを作成
引数なしで派生クラスを作成
```

ちゃんと派生クラスのインスタンスを作成することが出来ました．

ここで，ポイントは

```
                super("継承元コンストラクタ呼び出し");
```

です．

すでに説明したとおり，super というのは継承元クラスのメソッドやフィールドを呼び出すためのキーワードです．その super を使って，

```
                super();
```

とすれば，継承元クラス(スーパークラス)のコンストラクタを呼び出すことができます．

継承元クラスにデフォルトコンストラクタがない場合は，このようにして，派生クラスでは，明示的に継承元クラスのコンストラクタを呼び出す必要があります．

ちなみに，継承元クラスのコンストラクタの呼び出しは，派生クラスのコンストラクタの最初に書かなければ行けません．

```
public class ExtendedClass extends SuperClass {

	public ExtendedClass() {
		System.out.println("引数なしで派生クラスを作成");
		super("継承元コンストラクタ呼び出し");
	}
}
```

こんな書き方はできませんので，注意しましょう.

8.4.3.4　コンストラクタまとめ

派生クラスのコンストラクタのルールは簡単にまとめると以下のようになります.

- 派生クラスのコンストラクタは継承元クラスのコンストラクタを呼ばなければいけない
- 継承元クラスにデフォルトコンストラクタがあれば省略しても，デフォルトコンストラクタを呼び出す
- 継承元クラスにデフォルトコンストラクタがなければ，明示的にコンストラクタを呼び出す必要がある
- 継承元クラスのコンストラクタの呼び出しは，派生クラスのコンストラクタの最初で行わなければ行けない

8.4.4　すべてのクラスは Object

クラスの継承についていろいろ学んできましたが、ここでちょっと衝撃的な話をしたいと思います。

実は、Java において、すべてのクラスは Object というクラスを暗黙的に継承しています。

つまり、

```
public class SuperClass extends Object{
```

とかいてあるのとまったく同じなのです。

そのため、Object 型の変数を作っておくと、どんなクラスのインスタンスでも入れることが可能になります。

```
Object obj = new SuperClass();
obj = new Integer(0);
```

結構重要なテクニックですので、ぜひ覚えておいてください。特に，Object クラスの配列を作成すると，クラスのインスタンスならなんでも格納可能なハイブリッドな配列も作成できてしまいます.

```
Object[] objArray = new Object[100];
objArray[0] = new Integer(0);
objArray[1] = "文字列だってこの通り";
```

また，Object クラスにはいくつかのメソッドが用意されています.