# 12 Java の高度なお話

さて長かった Java のお話もいよいよ本章でラストとなります．本章では割と高度なテクニックである Exeption,そしてファイルの入出力についてお話ししたいと思います．

また，最後にはより良い Java プログラミングを行うために必要ないくつかのアドバイスをお送りしたいと思っています．

では，後一踏ん張り，頑張りましょう．

## 12.1　　例外処理 try～catch

### 12.1.1 例外処理とは

プログラムを作るということは，バグと戦うことであると言い換えることができるかもしれません．バグとは，ようするに思うとおりにプログラムが動いてくれないことです．

ところで，このバグには３種類あります．一つは，プログラムの構造上のミスです．例えば，

```
if(a == b{//閉じ忘れ！
          a =  a+b;
}
else{
          a = 0;
}
```

このようなプログラムを書くと，最初の if 文で（）を閉じていないため，コンパイルを行おうとしたときにエラーが発生します．

このコンパイル時にエラーが起きて，コンパイルが出来ない場合，**構造上のエラーがある**といいます．

次に，こんな場合もエラーが発生します．

```
int[] data = {1,2};
int d = data[2]; //data の添え字は１まで！
```

このようなミスは，構造上は間違っていないため，コンパイル時に発見することが出来ません．そのため，プログラムの実行中に途中で停止することになります．このような場合を**実行上のエラー**と呼びましょう．

そして，最後に予想外の構造上も実行上もエラーが無いけど，ただ単にプログラムを書き間違えていておきるエラーがあります．

```
//4 週間にわたって毎日 100 円ずつ支払いをするプログラム
for(int i = 0; I < 4;i++){
          for(int j = 0; j < 7; i++){ //j と i を書き間違えた
                    //100 円の支払い処理
          }
}
```

上記の例だと，二つ目の for 文では j++と書かなければいけないのに,間違えて i++と書いてしまっています．for 文の終了条件が満たされずに for 文が無限に繰り返されることになってしまいます．このままでは永久に毎日 100 円支払いを続けなければいけなくなってしまいます！これは大変．

でも，このようなエラーはコンパイル時には発見できませんし，実行時にエラーが発生して止まることもありません．

このようなエラーを**処理上のエラー**と呼びます．処理上のエラーは，間違えに気づきにくいので注意が必要です．このようなエラーは実行結果や途中経過を見ることによって，人間が間違いを見つけて修正しなければいけません．プログラム自体は正しく動いていますのでJava君が間違いを発見してはくれないのです．プログラミングをしていると，この処理上のエラーに何度となく悩まされることになると思います．覚悟しておいてください．

って,話が横にそれましたね．さて，本章でターゲットとするのは，実行上のエラーについてです．

つまり，配列で存在しないはずの添え字を使おうとしたり，0で割る計算をしてしまったり，と明らかに間違っているとJava君にも分かるけど，実際にプログラムを動かしてみるまではそうなるかどうかが分からないエラーです．

これらのエラーが発生すると，Java君は「変なことがおきているよ！」と教えてくれますので，それに対して処理を行うやり方を学んでいきましょう．

### 12.1.2 tryとcatch

JAVAにおいて重要な考えの一つに，例外処理というものがあります．例外処理とは，ようするに何かやってはいけないことをプログラム上でやってしまったときに，それを処理するための機能です．

分かりづらいですので，まずは例を見てみましょう．

```
        String[] dateOfWeek = {"月", "火", "水", "木", "金", "土", "日"};

        String sunday = date[7]; //dateの添え字は0-6まで！
        System.out.println(sunday);
        System.out.println("終了！");
```

ここの例は，おそらく最もやりがちな間違いの一つ，配列の添え字違反です．一週間は7日ありますので，日曜日を取得しようとしたら，ついつい

```
dateOfWeek[7]
```

とやりたくなってしまいますが，配列は0からスタートするので，添え字が7のデータは存在しません．この場合，どうなるでしょうか？もうすでに何度かやらかしてしまっているので，おわかりかと思いますが・・・

```
> java TryCatch
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
>
```

というわけで,エラーが起きてプログラムが終了してしまいました.一番最後の「終了！」が表示されていないことからもプログラムが終ってしまったことが分かりますよね．

これまでは,「エラーが起きちゃった．修正しなきゃ！」と思っていただけですが，今回は，そこから一歩踏み出してみましょう．

このようなエラーが発生してもプログラムを終了させずにそれなりの対応をするようにしたいと思います．

```
try{
        String[] date = {"月", "火", "水", "木", "金", "土", "日"};

        String sunday = date[7]; //date の添え字は 0-6 まで！
        System.out.println(sunday);
}catch(ArrayIndexOutOfBoundsException e){
        System.err.println("添え字が変です");
        System.err.println("気をつけてね");
}

System.out.println("終了！");
```

このプログラムを動かすと，

```
> java TryCatch
添え字が変です
気をつけてね
終了！
>
```

と，本来存在しないはずの配列の添え字を利用してもプログラムが勝手に終了せずに，ちゃんと「終了！」と表示してから終っています．

その秘密が，try～catch なのです．

try～catch の基本的な機能は，

**「try ブロックの中で発生した Exception(エラー)を catch でチェックして処理を行う」**

というものです．

今回の例だと，try ブロックの中で配列の添え字違反が発生して，それを catch でチェックして「添え字がおかしい」という警告メッセージを出力するようになっています．

try～catch の基本構文は以下のとおりです．

```
try{
        //処理
        //エラーが発生する処理
}
catch(Exception e){
        //Exception 発生時の処理
}
```

処理の流れは，

try 以降の処理を行っていき，エラーが発生した時点で catch のブロックへ処理が移動します．

ちなみに，エラーが発生することを Java では「**例外をスルーする**」とか「**例外を投げる**」といいます．ようするに「こんなエラーが起きたよ！」という情報を投げるんですね．ぽいっと．そして，それを catch ブロックが受け止めるという図式です．

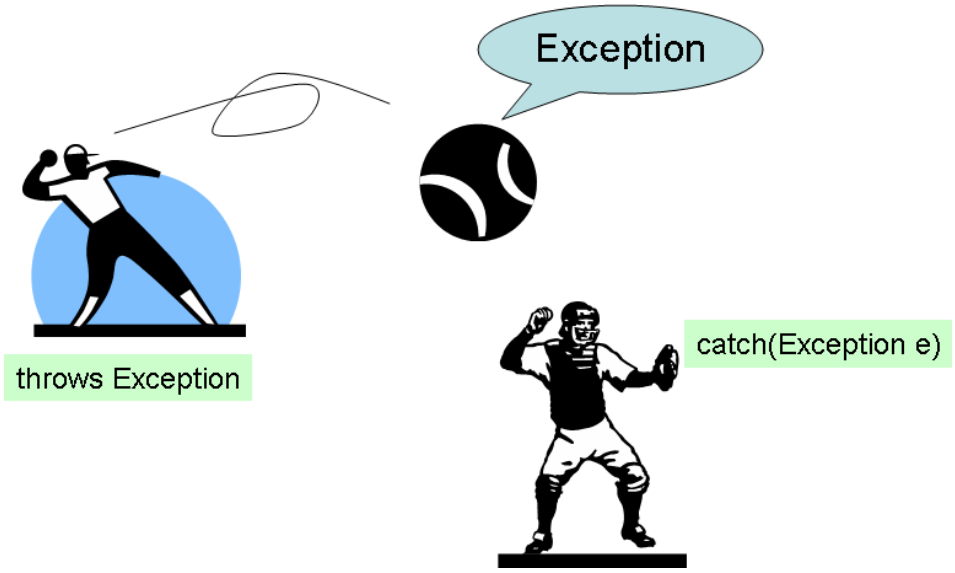



**図 投げた例外をキャッチする**

で，このキャッチできる守備範囲が try で囲まれた範囲ということなのです．

この try～catch をうまく使えば，プログラムの途中でエラーが発生してもプログラムを続行させることが出来ます．もしかしたらエラーが起きるかもしれないけど，なんとかリカバーできる場合などに ty～catch を有効に利用してください．

### 12.1.3 例外は変数である

エラーが発生したときは，Exception が投げられるという話をしました．このとき投げられる Exception は実は変数です．ある Exception を表すクラスを用意しておいて，そのクラスのインスタンスを作成してスルーするんです．

ちなみに，その投げられた Exception のインスタンスは，catch ブロックで取得することが出来ます．

```
                }catch(ArrayIndexOutOfBoundsException e){
```

と書いてありましたが，この e が ArrayIndexOutOfBoundsException 型の変数になります．Java では**エラーメッセージまでクラスで作ってしまう**んです．なかなか徹底しているとは思いませんか？

もちろん，Exception がクラスということは，継承したり，メソッドを使ったりすることも出来ます．

例えば, Exception 型にあるメソッドの一つに, printStackTrace というものがあります．これは，画面にエラーメッセージを表示してくれる機能です．

```
        }catch(ArrayIndexOutOfBoundsException e){
                e.printStackTrace();
        }
```

とすると，画面にエラーメッセージを表示してくれます．

通常エラーが起きたらエラーメッセージが表示されてプログラムが停止してしまいますが，catch することによってプログラムが停止することを押さえることが出来ます．しかし

ながら，その一方でエラーメッセージも表示されなくなってしまいます.

まあ，プログラムを動かす上ではそれでもいいのですが，どこでどんなエラーが発生したかをちゃんと把握しておきたい場合はちょっと不便です.

そんなときは，この printStackTrace を使って，画面上にはエラーメッセージを表示する物の，プログラムは停止せずにそのまま動かし続けることが出来るようになります.

割と便利で，扱いやすいテクニックですので，是非覚えておいてください.

ただし，Exception が発生すると言うことは，何かプログラム的に間違えているということですので，原因をはっきりさせて対処しましょうね.

### 12.1.4 複数エラーのキャッチ

ところで，このとき catch のあとの()内には

```
catch(ArrayIndexOutOfBoundsException e)
```

と，まるで引数のようにクラス名，変数名が書かれています.（ArrayIndexOutOfBounds はクラス名ですよ）

じつは，catch ブロックでは，どのようなエラーが発生したかを仕分けして，エラーの種類ごとに処理を帰ることも出来るのです.

ここでは，ArrayIndexOutOfBoundsException という名前のクラスをキャッチしていますが，それ以外のエラーが起きた場合は別の処理を行うことが出来ます.

たとえば，以下のように変更してみましょうか．プログラムを変更してみましょうか.

```
try{
        int k = 101/0; //0 で割ってはいけません！

        String[] date = {"月", "火", "水", "木", "金", "土", "日"};

        System.out.println(date[7]); //date の添え字は 0-6 まで！


}catch(ArrayIndexOutOfBoundsException e){
        System.err.println("添え字が変です");
        System.err.println("気をつけてね");
}catch(ArithmeticException e){
        System.err.println("やってはいけない計算をしていますよ");
        System.err.println("気をつけてね");
}
```

ここでは，0 で割り算を行うという暴挙に出ています．つまり，数学的にやってはいけないことをやっているわけです.

さらに, 下の方には ArithmeticException をキャッチする catch ブロックが存在します.

さあ，どうなるでしょうか？予想は付きますけど，念のためやってみましょう.

```
やってはいけない計算をしていますよ
気をつけてね
終了！
```

AttayIndexOutOfBoundsException のブロックを通り越して，ArithmeticException のブロックが実行されていることが分かりますね．ちなみに，この場合 ArrayIndexOutOfBoundsException は発生しません．なぜなら，その前にプログラムが ArithmeticException のブロックに飛んで(コメント：①，②など番号を振っておきたい)しまうためプログラムが問題の地点に到達しないためなんですね．

ちなみに，もしこのキャッチブロックが無ければ，

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

というメッセージと共にプログラムが終了することになります．

このように，どんな Exception が発生したかによって処理を変化させることで，より多くのエラーに対応することが出来るようになります．

ちなみに，どんな Exception が発生してもとりあえず catch したい場合は，以下のようにします．

```
try{
        //Exception 通常処理
}catch(Exception e){
        e.printStackTrace();
        //Exception 発生時の処理
}
```

このようにすることで Exception から派生したすべてのクラスのインスタンスをキャッチすることが出来ます．すべての Exception クラスは，Exception クラスから派生していますので，これで全部キャッチできるわけです．

本当は一つ一つの Exception をちゃんと吟味してプログラミングしなければいけませんが，デバッグ中などで面倒くさいときは一気にすべての Exception をキャッチしてしまうのも一つの手です．

ただし，この場合は**必ず printStackTrace を使ってどんな Exception が発生したかは表示する**ようにしましょう．これをやらないと，重大なエラーを見逃してしまうかもしれませんからね．

### 12.1.5 finally

さて，プログラムで不正なことを行うと Exception が投げられるのですが，Exception が投げられようと投げられまいと，行いたいことというのもあります．そんなときに使うのが finally です．

```
try{
        int k = 101/0;

        String[] date = {"月", "火", "水", "木", "金", "土", "日"};

        System.out.println(date[7]); //date の添え字は 0-6 まで！
```

```
                }catch(ArrayIndexOutOfBoundsException e){
                        System.err.println("添え字が変です");
                        System.err.println("気をつけてね");
                }catch(ArithmeticException e){
                        System.err.println("やってはいけない計算をしていますよ");
                        System.err.println("気をつけてね");
                }
                finally{
                        System.out.println("いずれにせよ実行");
                }

                System.out.println("終了！");
```

さきほどまでのプログラムに，finally ブロックを追加してみます．これを実行してみると，どうなるでしょうか？

```
やってはいけない計算をしていますよ
気をつけてね
いずれにせよ実行
終了！
```

ちゃんと finally が実行されましたね．

逆に Exception が発生しない場合はどうでしょうか？

```
        public static void main(String[] args) {
                try{
                        int k = 101/0;
                        String[] date = {"月", "火", "水", "木", "金", "土", "日"};
                        System.out.println(date[0]); //date の添え字は 0-6 まで！

                }catch(ArrayIndexOutOfBoundsException e){
                        System.err.println("添え字が変です");
                        System.err.println("気をつけてね");
                }catch(ArithmeticException e){
                        System.err.println("やってはいけない計算をしていますよ");
                        System.err.println("気をつけてね");
                }
                finally{
                        System.out.println("いずれにせよ実行");
                }

                System.out.println("終了！");


        }
```

Exception が発生する可能性のある処理をすべて消してしまいます．これでどうなるでしょうか？

```
いずれにせよ実行
終了！
```

この場合も，finally ブロックの中身が実行されました．

このように，finally ブロックを使えば，どんな Exception が発生しても必ず実行される文を作ることが出来ます．

この機能が特に役立つのは，throws を使って Exception を投げた場合です．通常，catch

されない Exception は全て上位メソッド，つまり Exception が発生したメソッドを呼び出したメソッドへ渡されます．

このとき，問答無用で上のメソッドに戻ってしまうため，Exception が発生した後に書かれている処理はすべて無視されてしまいます．そのため，何があってもどうしても処理しておきたい物があった場合，finally ブロックを活用する必要が出てきます．

もっとも， finally ブロックは割と高度なテクニックですので，しばらくは使うことはあまりないでしょう．いずれ必要となったときに思い出してあげてください．

### 12.1.6 throws

プログラムを書いていると，Exception が発生しても，そのメソッドではどうすればいいのか判断が付かない場合があります．

そのような場合は，throws を使います．ここでは，なつかしの個人データクラスを使った例をお見せしましょう．今回変更するのは getAddress メソッドです．アドレスを設定していないのにアドレスを取得しようとしたときに Exception が投げられるようにします．

```
	public String getAddress() throws Exception{
		if(address == null){
			throw new Exception("アドレスが設定されていません");
		}

		return address;
	}
```

まず，ポイントとなるのがメソッドの宣言の後に throws Exception と書かれている点です．この宣言は，「このメソッドからは Exception が投げられる可能性があるから，必ず処理して頂戴ね」という意味を持ちます．

今回は Exception という**すべての Exception のスーパークラス**を書いていますが，通常は個々の Exception を記述します． ArrayIndexOutOfBoundsException や， NullPointerException，IOException などを記述します．

なお，基本形は以下のとおりです．

```
返り値 メソッド名(引数) throws Exception名{
	処理
}
```

次に，不正があった場合の処理です．

不正な値が与えられた場合，このメソッドは Exception を投げることになります．それが，この部分になります．

```
		if(address == null){
			throw new Exception("アドレスが設定されていません");
		}
```

何をやっているのかといえば，Exception 型の新しいインスタンスを作成して，throw しているのです.

このように,

```
throw Exception のインスタンス;
```

と記述することで，任意の場所から例外を投げることが可能です.

では，実際に使ってみましょう.

```
        public static void main(String[] args) {
                PersonalData mario = new PersonalData("マリオ");
                String address = mario.getAddress();
                System.out.println(address);
        }
```

こんなプログラムを書いてコンパイルしてみましょう.

```
PersonalData.java:7: 例外 java.lang.Exception は報告されません。スローするにはキャッチまたは、スロー宣言をしなければなりません。
                String address = mario.getAddress();
                                                   ^
エラー 1 個
```

と，こんなメッセージが出てコンパイルに失敗してしまいました.

実は、throws が書いてあるメソッドを呼び出す場合は、必ず try～catch によって，その Exception を処理しなければいけないという決まりがあります。ようするに、メソッドが「こういう Exception を投げる可能性があるからね」と言った場合、その Exception をどうするかをあらかじめ決めておかなければいけないわけです。

では、このコンパイルが通るようにするためにはどうすればよいでしょうか？方法は二つあります。

ひとつは、ひとつは、try～catch 文を書くこと、さらに上のメソッドに throw する方法となります。try～catch の方法は先ほど述べた方法と同じで、try 文で囲んで、catch 文で対象となる Exception を捕まえることになります。

```
                try{
                        PersonalData mario = new PersonalData("マリオ");
                        String address = mario.getAddress();
                        System.out.println(address);
                }catch(Exception e){
                        e.printStackTrace();
                }
```

また、もう一つの方法は，さらに上のメソッドに throw する方法です。

これは、非常に簡単で、ようするにこのメソッドでは処理しきれないので、このメソッドを呼び出したメソッドに処理を任せてしまうという、丸投げ方法になります。呼び出し側メソッドにはいい迷惑です。

```
        public void makeAddress() throws Exception{
                PersonalData mario = new PersonalData("マリオ");
```

```
                String address = mario.getAddress();
                System.out.println(address);
        }
```

以上のように、メソッドに throws 文を書きます。ただし、この方法をとる場合は、このメソッドを呼び出すメソッドで同じように throw される Exception の処理を行わなければいけなくなってしまいます。throws があるメソッドを呼び出すときは，必ず throw される Exception に対する処理をしなければいけないのですからね.

ちなみに、全てのメソッドが次々と Exception を呼び出しメソッドに任せていってしまうと、最終的に main メソッドまでたどり着いてしまいます。じゃあ、もし main メソッドでも throws 文を書いてしまうとどうなるでしょうか？

```
        public static void main(String[] args) throws Exception{
                makeAddress();
        }

        static public void makeAddress() throws Exception{
                PersonalData mario = new PersonalData("マリオ");
                String address = mario.getAddress();
                System.out.println(address);
        }
```

この場合、もし Exception が発生するとこのように出力されプログラムが停止します。

```
Exception in thread "main" java.lang.Exception: アドレスが設定されていません
        at PersonalData.getAddress(PersonalData.java:52)
        at PersonalData.makeAddress(PersonalData.java:21)
        at PersonalData.main(PersonalData.java:16)
```

これは、配列の添え字違反などの問題が発生したときと同じですね。ようするに、何か問題が発生したときに main メソッドでも解決方法を提示しなければ、プログラムはそれ以上続けることができなくなって終了してしまうのです。

まあ、通常プログラムを書いている分にはこれでもよいのですが、自作のプログラムを誰かに使ってもらおうという場合は、エラーメッセージが出て途中でプログラムが止まってしまっては大変ですから、throws をメインメソッドまで持っていくのはやめたほうが良いでしょう。

### 12.1.7 Exception 型と RuntimeException 型

throws によって Exception を投げた場合、必ず try～catch をするか、さらに上のメソッドに丸投げしなければいけないと書きましたが、実は必ずしも try～catch しなくてもよい Exception があります。それが、RuntimeException です。

この RuntimeException の代表例は、ArrayIndexOutOfBoundsException です。これは、配列の添え字に存在しない値を書いてしまった場合などに発生する Exception でしたよね。

```
int ary = new int[100];
ary[200] = 10; //添え字は 99 まで！ArrayIndexOutOfBoundsException 発生!
```

このような例外の場合、ちゃんと気をつけていれば発生しない Exception なのですから、必ず try～catch しなければいけないとなると、非常に面倒ですよね。配列を使うたびに try～catch を書かなければいけないのですから。

```
try{
        int ary = new int[100];
        ary[0] = 10; //絶対安全とわかっているのに・・・
}catch(ArrayIndexOutOfBoundsExceptione e){
        //いちいち try～catch を書くのは面倒だなあ
}
```

でも、ご安心を。Java ではこんな面倒くさいことはさせません。このような、必ずしも毎回例外が発生していないか確認する必要もないような場合は、try～catch も throws も書く必要がないのです。

もちろん、添え字違反が起こる可能性が高く、万が一のときはちゃんと処理をしておきたいという場合は、try～catch を書いて、例外を捕まえて問題を処理することもできます。

そういう意味では、Exception よりも RuntimeException の方がプログラムを書く上では柔軟性があるといえるかもしれません。

そう考えると，すべての Exception クラスが RuntimeException クラスを継承したクラスだったらいいのに,と思ってしまうかも知れません．しかし，そうではないのです.

例えば， IOExceptin という Exception があります。これは、ファイル関係の例外で、派生クラスに FileNotFoundException などがあります.

この FileNotFoundException はプログラム中になんらかのファイルを開こうとしたときに、そのファイルが見つからなかった場合などに発生します。このような例外というのは、**どんなにプログラムをきちんと書いても、発生する可能性はあります**よね。何しろ、原因がプログラムそのもの以外に存在するのですから。このような例外については、もし何も処理を考えておかないと、プログラムは正しいのにファイルが存在しなかったがためにエラーが発生してプログラムが停止してしまう可能性が出てきてしまいます。これは、利用者にとっては嬉しくないですよね。そのため、このようなプログラムの書き方以外に原因がある問題についてはあらかじめ処理方法をプログラマが考えておかなければいけない、というのが Java の基本スタンスになります。

一方、RuntimeException を継承したは、同じく Exception を継承したクラスの一種となりますが、必ずしも try～catch をする必要がない例外という例外の中の例外(わかりづらいですが・・・)ということになります。

この RuntimeException は、プログラムをちゃんと書いておけば起こりえない例外に良く使われます。

ArrayIndexOutOfBoundsException などは添え字に正しい文字さえ入れておけば問題は発生し得ないわけですから、万が一 RuntimeException が発生したら、それはプログラムを書き換えるべき問題なわけです。したがって、このような問題はプログラム中で万が一発生したときの処理を書かなくても、プログラムを停止してしまって、問題点を書き直した方が良かったりもするわけです。

もちろん、予想外の問題が起きるかもしれない、ということを考慮して try～catch しておくこともできますので、RuntimeException はかなり柔軟に使うことができます、。

最後に、Exception と RuntimeException の関係を図にまとめておきたいと思います。

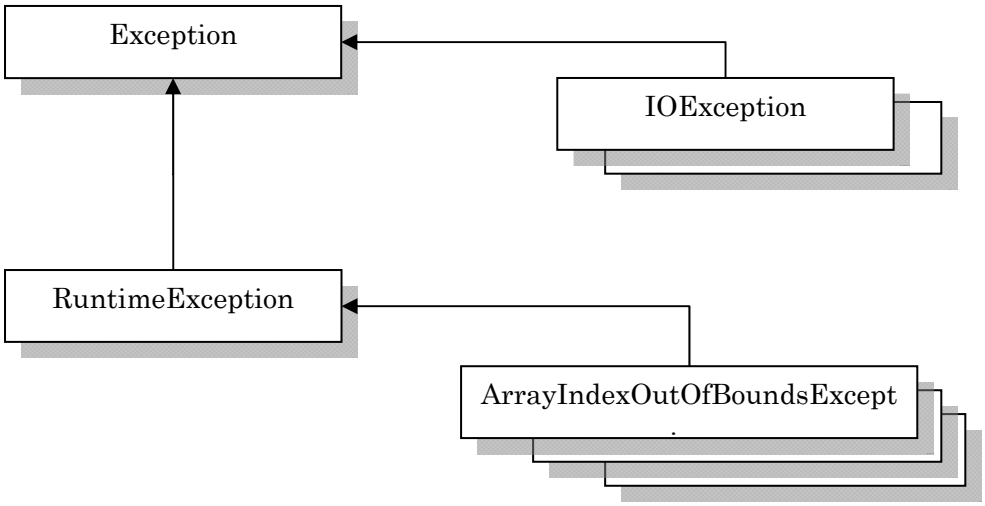



**図　Exception たちの関係図**

通常，Exception から直接は制した Exception クラスは，常に発生する可能性があり，発生したときの処理をあらかじめ考えておくべき Exception です.

一方，RuntimeException を継承したクラスは，通常は発生しないけれどもプログラム上のミスや予想外の動作によって発生する例外になります.

上手に使い分けて，予想外の動作をしたときに適切な対処ができるようにしておきましょう.

### 12.1.8　自作 Exception の作り方

Exception は、Java の API にもいくつか用意されています。これまでにも何度も出てきている ArrayIndexOutOfBoundsException もそのひとつですし、NullPointerException もよくお目見えする Exception です。また、ファイルの入出力を始めると、毎日のように IOException とお付き合いをしなければいけません。

一方で、自分で Exception を作りたい場合というのもあります。たとえば、時計クラスを作った場合は、時計に関する例外を ClockException という名前で設定するのは良いアイデアです。設定してはいけない時間、たとえば 80 時とか、-14 分などが設定されたときには、ClockException を throw するようにしたいですよね。

というわけで、そんな Exception を作成してみましょうか。

今回は、ClockException は主にプログラム内のミスで発生すると思われる Exception ですので、RuntimeException として作成します。

```
```

RuntimeException を作成するためには、単純に RuntimeException クラスを継承すればそれで完成です。簡単ですね。

次に、その ClockException を投げるように時計クラスを変更してみましょうか。

```
```

これで完成です。

どうでしょうか、自作 Exception を作るのは意外と簡単です。

### 12.1.9 例外処理の使い方

例外処理は、これまでに説明してきたものに比べて、かなりわかりづらい部分があったかと思います。特に、どのような利用方法があるのか、直感的にはわかりづらい機能です。

たとえば、配列の添え字違反がありそうなら，

```
		int[] ary = new int[100];

		int i = 10;
		if(i >= 0 && i < ary.length){
			System.out.println(i);
		}
```

このようにあらかじめ範囲を確認すれば Exception は発生しません.

また、throws を使う場合も、返り値をうまく使えば，問題ないような気がします.

例えば，下記の例では，price が正だったら画面に表示して true を返し，そうでなければ不正な値ということで false を返すことにしています.

```
	boolean showPrice(int price){
		if(price < 0){
			return false;
		}
		else{
			System.out.println(price);
			return true;
		}
	}
```

このようにすればわざわざ Exception を使うまでもなくエラーが発生したかどうかは判断できそうです。ということは、別に例外処理なんてしなくてもなんとかなりそうですよね。

確かに、そのとおりでプログラムを書いていると最初のうちは Exception という仕組みは面倒くさいだけであまり役立つ気がしません。

しかしながら、あるレベルに達すると急に Exception が非常に便利であるということがわかってくると思いますので､そのときまで Exception のことを忘れないでいてください。

ここでは、そんな「便利だなあ」と思う例をひとつだけあげてみましょうか。

では、例として生まれた年から年齢を計算するメソッドというのを考えて見ましょうか、

```
int birthYearToAge(int year){
	int thisYear = Calendar.getInstanse().get(Calendar.YEAR); //今年は何年か取得
	if(year > thisYear){//まだ生まれていない！
		return -1;//とりあえず-1 でも返しておくか・・・
	}
	return thisYear-year;
}
```

こんなメソッドを作ってみました。これで、生まれた年を西暦で入れればその人の年齢が取得できます。誕生日については考えていないので、ちょっと誤差がありますが、まあ、気にしないでおいてください。

このメソッドは、年齢を返してくれるわけですが、year に来年以降の値を入れてしまった場合について考えなければいけません。まだ生まれていない人の年齢はどうするべきでしょうか？0 歳じゃないですよね｡かといって､マイナス 14 歳というのもおかしな話です。そこで、このメソッドを作った人は、面倒なので生まれていない人は全員-1 歳ということ

にしてしまいました。

この場合 birthYearToAge を使う人は**「不正な値を入力した場合、-1 が返ってくる」**というルールを覚えていなければ行けません。そして、-1 が返って来た時の処理を書いておかなければいけません。

覚えていればいいのですが、たまに忘れてしまうことがあります。というか、忘れます。よく忘れます。自分で作った関数でも 1 年もすれば間違いなく忘れているでしょう。筆者はしょっちゅう忘れます。もし、使い方を忘れてこんなことをしてしまったら大変です。

```
int doraemon = 2112;

int age = birthYearToAge(doraemon);
System.out.println("ドラえもんの年齢は"+age+"才ですよ");
```

このプログラムの結果が

```
ドラえもんの年齢は-1 才ですよ
```

となってしまいます．ドラえもん若すぎ．

一方例外処理を使うと忘れることなく不正な値を入力したときの処理を書くことが出来ます。

```
/**
* 未来時間を入れたときの例外
*/
class FutureYearException extends Exception{
        //中身はなくても Ok
}

・・・

/**
* 生まれ年から年齢を計算するクラス(例外処理版)
* @param year 生まれ年
*/
int birthYearToAge(int year)throws FutureYearException{
        int thisYear = Calendar.getInstanse().get(Calendar.YEAR); //今年は何年か取得
        if(year > thisYear){//まだ生まれていない！
                throw new FutureYearException();          //未来の時間を入れましたよ！という例外を投げる
        }
        return thisYear-year;
}
```

このように書くことで、もし、year が 2005 以上の値だった場合 FutureYearException というクラスが例外クラスとしてこのメソッドを呼び出した側に投げられます。

ちなみに、throw というキーワードからも分かるように、例外が発生したとき Exception が送られてくることを「例外が投げられた」といいます。

さて、このような場合、

```
        int a = birthYearToAge(1976);  //コンパイルエラー
```

という書き方は出来ませんよね。かならず、try～catch ブロックで囲み、間違えて未来生まれの人の年齢を知ろうとしてしまったときの処理が必要です。

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

```
                try{
                        int year=Integer.parseInt(args[0];
                        int age = birthYearToAge(year);
                        System.out.println(year+"年生まれの人の年齢は"+age+"才ですよ");
                }catch(FutureYearException e){
                        //FutureYearException が投げられたときの処理
                        System.out.println("未来に生まれる人の年齢は計算できません・・・");
                }finally{
                        //例外があってもなくても必ず行う処理
                        System.out.println("処理終わり");
                }
        }
```

これで、引数に西暦を入れると、その年に生まれた人の年齢を返してくれるプログラムが完成です。

これを使ってドラえもんの年齢を調べてみましょうか。ドラえもんは 2112 年生まれなんですよ。知ってました？

```
$ java ExceptionTest 2112
未来に生まれる人の年齢は計算できません・・・
処理終わり
```

というわけで未来の時間である西暦 2112 年生まれのドラえもんの年齢は計算できないようになりました。のびたくんもがっかり。

でも、以前のメソッドだと-1 才といわれたはずなので、それに比べればましかもしれません。

というわけで、あるメソッドで問題が発生するが、返値でそれを知らせるのが困難な場合に Exception を利用すると問題がわかりやすくなります。

これは、特に他人に自作のメソッドを使ってもらう場合などに有効な技術です。たとえば、メソッドを使う人がメソッドの中身をよく理解できていなかったとしても、Exception を throw することによって、どんな問題が発生しているのかが一目でわかります。

先ほどの例だと、FutureYearException が投げられるということは、未来時間を引数に入れてはいけないんだな、と一目でわかります。

プログラムを書いていると、色々な問題が必ず発生してきます。プログラムの記述と例外はセットのようなものです。

最初のうちは自作 Exception 使う機会はないかもしれませんが、API の Exception とはしょっちゅうお付き合いすることになります。

try～catch の使い方だけでもぜひマスターしておいてください。

## 12.2　　ファイルの入出力

これまでは，プログラムを動かした結果は，画面に表示するだけでした．しかし，これだと一回プログラムを動かすと一回しか確認できません．

もしかして，二回プログラムを動かせばいいとか思っていませんか？そりゃあ，一回動かすのに 1 秒もかからないプログラムなら何度動かしても苦になりませんが，もし一回動かすのに 1 時間かかるプログラムがあったらどうでしょう？しかもその結果を見逃してしまったりしたら・・・ビデオのように巻き戻しなんてことはできませんから，大変です．

また，プログラムを動かした結果が膨大なデータを出力するものだったらどうでしょうか？一画面には収まりきらないような文章が出力されるプログラムの場合，その場で読むのではなくテキストファイルに入れておいて，後でメモ帳などを使ってじっくりと読みたいですよね．

というわけで，ファイルへの入出力について学びましょう．

### 12.2.1 ファイルからの入力

文字列をファイルから入力する場合に利用するのが、 BufferedReader と FileReader です。

以下に，ファイルの中身を全て読み込んで画面に出力する場合の基本形を例示しましょうか．

```
import java.io.*;
public class ReadFile{
          static public void main(String[] args){
                    try{
                              Reader reader = new FileReader(args[0]);
                              BufferedReader br = new BufferedReader(reader);
                              String data = null;
                              while((data = br.readLine()) != null){
                                        System.out.println(data);
                              }
                              br.close();
                              reader.close();
                    }catch(IOException e){
                              e.printStackTrace();
                    }
          }
}
```

これが，ファイルからのデータ読み込みの基本となります．これによって，String 型変数 fileData にファイルの中身が全て保存されます．

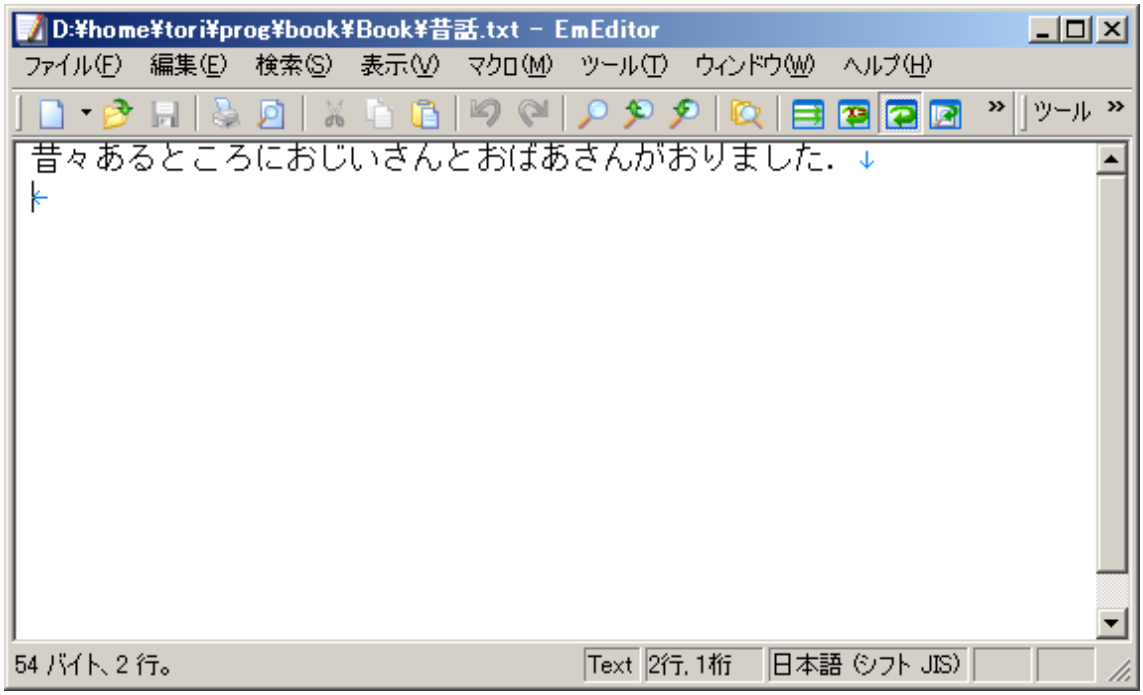
では，実行してみましょう．このプログラムを動かす場合は，

```
>java ReadFile fileName
```

のように，ファイル名を入力して実行します．

ちなみに，やってくれることは，指定したファイルの中身を読み出して全部画面上に書き出してくれる，ただそれだけです．

さあ，実行してみましょうか．ちなみに，昔話.txt の中身はこんな感じです．




さあ，無事 Java は昔話を語りだすことが出来るでしょうか？

```
>java ReadFile 昔話.txt
昔々あるところにおじいさんとおばあさんがおりました.
>
```

というわけで，無事 Java に昔話をさせることに成功しました．すごい．これで Java プログラミング中に急に昔話を読みたくなったときも安心ですね．

ところで，このプログラムを見ると，たかがファイルを読み込むだけで偉い騒ぎです．

が，これだけ複雑なのにも実はちゃんと理由があるのです．順を追って説明していきましょうか．

まず，読み込み部分について説明しましょう．

```
        Reader reader = new FileReader(args[0]);
        BufferedReader br = new BufferedReader(reader);
        String data = null;
        while((data = br.readLine()) != null){
                System.out.println(data);
        }
        br.close();
        reader.close();
```

まず，読み込みで重要なのは太字部分です．

どうせだったら，こんな複雑なことをしなくてもファイル名を指定すればファイルの中

身を全部読み込んでくれるクラスがあれば便利なのですが，Java ではあえてクラスを二つに分けて読み込むようにしています．

それは，読み込みの機能を分離しておくことで色々な場所からの読み込みを統一的に扱うことが出来るようにするためなのです．

ちょっと順番が逆になりますが，先に BufferedReader について説明したいと思います．

このクラスは，**入力ストリーム**から効率的に文字列を読み込むためのクラスです．入力ストリームというのは分かりにくい概念かもしれませんが，ある Java プログラムに向けてデータが送られてきている状態だと思ってください．

今回はファイルからデータが送られてきましたが，時にはキーボードからテキストが打ち込まれるかもしれませんし，インターネットからテキストを読み込むのかもしれません．もしかしたら，一見文字列には見えない形でデータが来るかもしれません．色々な方法を使ってあちこちからプログラムに向けてデータは送られてきます．そして，送り元に依存せずに統一したやり方で文字列を取得するために BufferedReader が用意されているのです．ようするに，

- どこからきているのか
- どうやってきているのか

は私達には分からなくても BufferedReader がなんとかしてくれて，文字列だけを取得しているように見せかけてくれるのです．

BufferedReader には readLine というメソッドが用意されています．このメソッドによって，（どこから送られてきたかはわからないけど）文字列を一行ずつ読み込むことが出来るのです．

```
while((data = br.readLine()) != null){
        //処理
}
```

それが，この部分です．ここでは，手を抜いて一行で書いていますが，ちゃんと機能一つ一つを書くとこのようになります．

```
while(true){
        String data = br.readLine();
        if(data == null){
                break;
        }
        //処理
}
```

ようするに，BufferedReader によって読み出した文字列一行を文字列の変数 data に代入し，null だったら終了し，null じゃなかったら処理を行う，という作業をします．

なぜ null かどうかをチェックしているかというと，文字列が送られてきていなかった場合や，すべて読み込み終わってしまった場合は null が返されると決まっているからです．今回の場合だと null が来たらファイルの中身を全部読みきったんだな，と分かるため処理を終了することになります．

さて，これで BufferedReader の使い方は分かりましたよね？じゃあ，いったい BufferedReader はどこからやってきたデータを読み込めばいいのでしょうか？

今回それを指定しているのが，

```
Reader reader = new FileReader(args[0]);
```

この部分です．

利用しているのは，FileReader クラスです．もう名前から見て分かりますよね．FileReader クラスは，ファイルからデータを呼び出すクラスです．つまり，FileReader クラスによって任意のファイルからデータを送り出す準備をします．そして，そのデータを BufferedReader に渡すことによってファイルからデータを読み込むことが出来るのです．

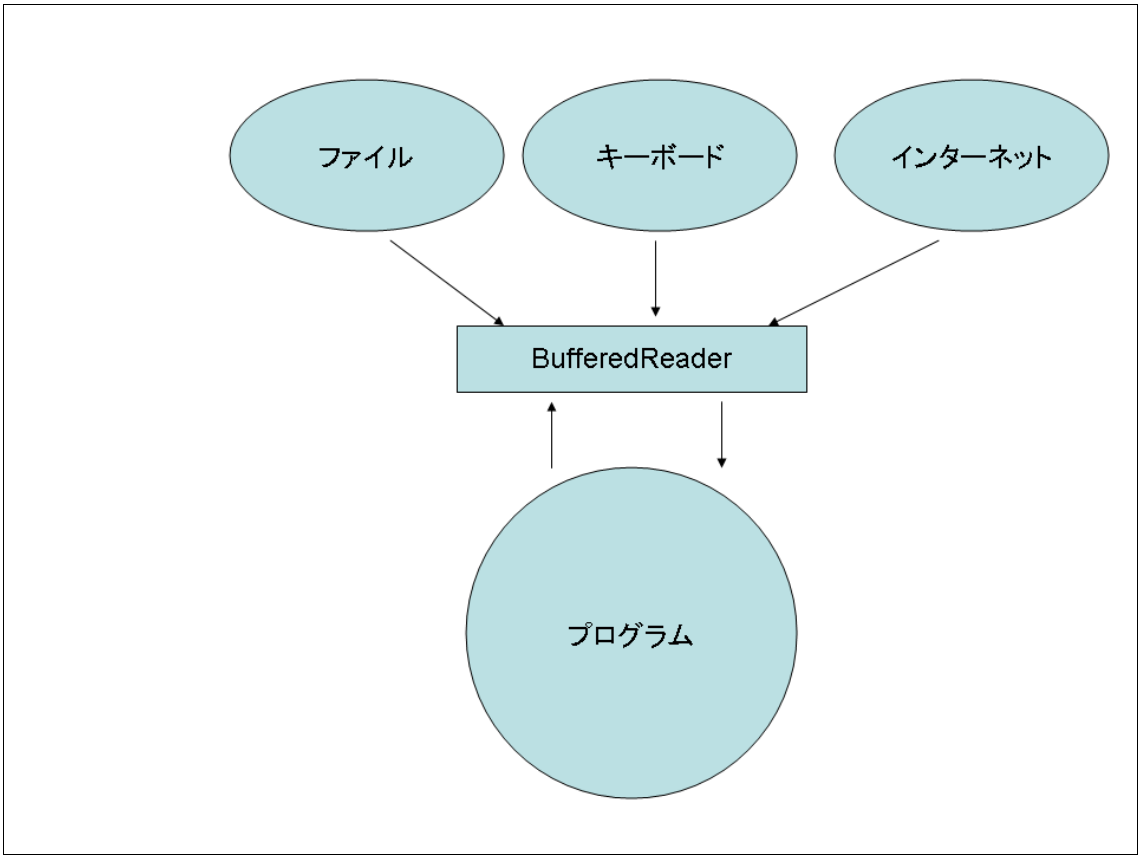



**図 どこから文字列がきても同じ扱い**

では，ためしにキーボードからの入力を取得してみましょうか．

```
Reader reader = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(reader);
String data = null;
while((data = br.readLine()) != null){
        System.out.println(data);
}
br.close();
reader.close();
```

ここで作成しているのはキーボードから入力するための Reader クラスです．まあ，こういうものだと覚えておいてください．

では，実行してみましょうか.

```
>java ReadFile
ヤッホー ←キーボードからの入力
ヤッホー
まねするな！←キーボードからの入力
まねするな！
>
```

おっと，見事鸚鵡返しで書いた内容を出力してくれるプログラムが出来ました.書いたことをそのまま繰り返すだけなので，ちょっとうざったいですが，とにかく一行変えただけでキーボードからの入力も処理できるプログラムになりました.

同じように Reader を変更するとインターネットからデータを読み込むことすら可能となります．便利ですね.

ちなみに，最後に

```
        br.close();
        reader.close();
```

という記述がありました.

これは，一種のおまじないのようなものなのですが「これ以上は読み込まないよ」という意味になります．簡単に言うと，メモ帳でファイルを開いていたのを，メモ帳を閉じてもう読まなくなるようなものです. close をして BufferedReader と FileReader を閉じてしまえば，それ以上はデータを読み込むことが出来なくなります.

なんでこんな機能が必要かといえば，Windows などではどこかで開いているファイルは削除したり出来ないという決まりがあります．そのため,close を使ってファイルを閉じて，他のプログラムで同じファイルを使えるようにしてあげる必要があります.

まあ，これはおまじないのようなもので，こうするものだと覚えておいてください.

### 12.2.2 ファイルへの出力

さて，ファイルから入力する方法が分かったら，順番から言っても次はファイルへ出力する番ですよね.

```
import java.io.*;

public class WriteFile {

        public static void main(String[] args) {
                try{
                        Writer writer = new FileWriter("output.txt");
                        BufferedWriter bw = new BufferedWriter(writer);

                        bw.write("ファイルへ書き込み");

                        bw.close();
                        writer.close();
```

```
                    }catch(IOException e){
                            e.printStackTrace();
                    }
            }

}
```

ぱっと見ると，ほとんど先ほどの読み込みと同じですね．違いは Reader が Writer になって，readLione が write になったことくらいでしょうか．

本当にそれだけか？と疑問になってしまうかも知れませんが，本当にそれだけなのです．

とはいえ，説明を全く省くわけにはいきませんので，まずは書き出す部分から説明しましょう．

```
BufferedWriter bw = new BufferedWriter(writer);

bw.write("ファイルへ書き込み");
```

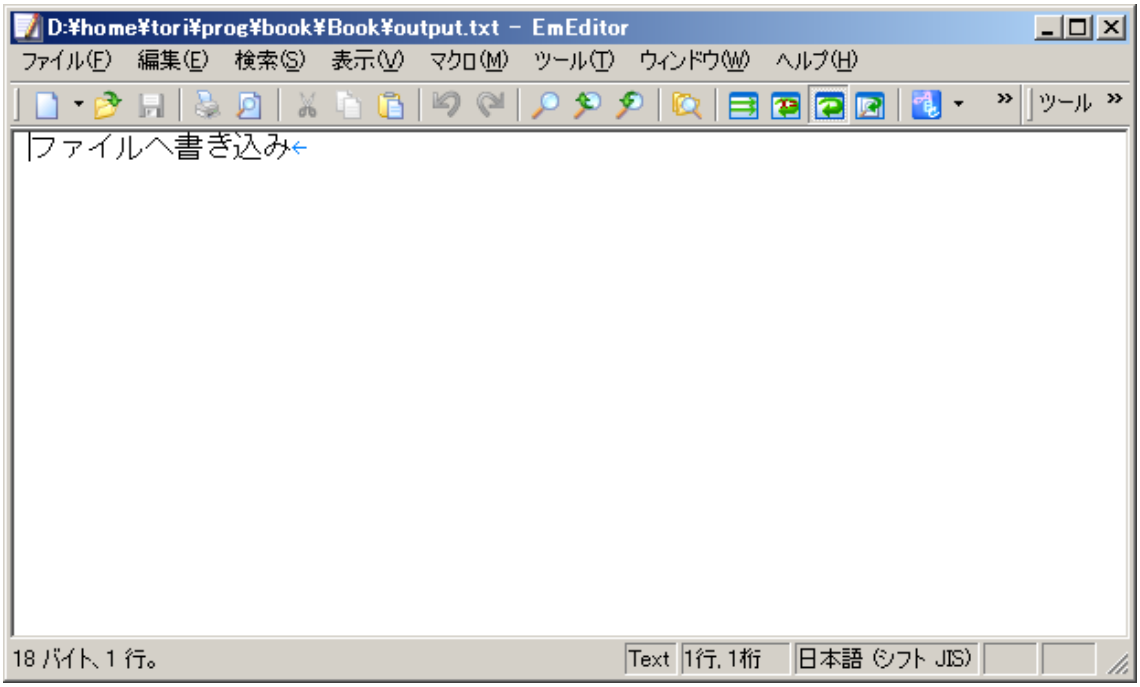
出力で重要なのはこの部分です．

入力の場合と大名時用に，BufferedWriter というクラスが文字列の出力を一手に担っています．ここに「どこへ出力するのか」を指定してあげれば，出力装置のできあがりです．

もうおわかりかと思いますが，出力先は

```
Writer writer = new FileWriter("output.txt");
```

に書いてあるとおり，FileWriter を使っている，すなわちファイルへの出力を行うよ，ということになっています．で，write メソッドによって，引数に書かれている文字列がそのままファイルへ出力されます．ここでは，output.txt というファイルに出力されます．



ところで，ファイルからの入力の時は Reader を変更することでキーボードからの入力に変更できました．出力のときも同じように出力先を変更できるのでしょうか？

もちろんできます．例えば，標準出力，つまり普通に画面に出力するには以下のように Writer を変更します．

```
Writer writer = new OutputStreamWriter(System.out);
```

これで，BufferedWriter の write メソッドを使ったとき，文字列を画面に出力してくれます．まあ，普通は

```
System.out.println("ファイルへ書き込み");
```

と書きますけどね・・・

### 12.2.3 例外処理

ところで，ファイルから文字列を読み込もうとしたとき，指定したのが存在しないファイルだったらどうなるでしょうか？

当然読み込むべきファイルが無いわけですから困ってしまいます．

読み込むべきファイルが無いのですから，br.readLine()でいきなり null が返ってくるというのも悪く無い方法ですが，それですと，

- ファイルはあるけど何も書いてない
- ファイルが無い

のどちらだか分からなくなってしまいます．

そこで，もし読み込むべきファイルが存在しなかった場合，Java は Exception を throw することによって読むべきファイルがなかったことを知らせてくれます．

このとき throw する Exception が FileNotFoundException です．もし，これが throw されたらファイルが無いんだな，とすぐにわかります．

これと同様に，ファイルの入出力関係では何か問題が発生した場合は Exception を throw することになっています．throw される Exception はすべて IOException の継承クラスです．

IOException は RuntimeException ではないので，必ず catch しなければいけません．ですので，ちょっと面倒ですが**ファイル操作をする場合は必ず try-catch をしなければいけない**，と覚えておいてください．

### 12.2.4 ファイルのコピーをする

せっかくファイルの入出力を習ったのですから，一つファイルをコピーするプログラムというのを作ってみましょうか．

ここでは，特に新しいことを学ぶわけではありませんが，どんな風に使えるのかを理解してもらうために作成してみましょう．

```
import java.io.*;
```

```
public class Copy {

	public static void main(String[] args) {
		String inputFile = args[0];
		String outputFile = args[1];

		try{
			Reader reader = new FileReader(inputFile);
			BufferedReader br = new BufferedReader(reader);
			Writer writer = new FileWriter(outputFile);
			BufferedWriter bw = new BufferedWriter(writer);


			String inputString;
			while((inputString = br.readLine()) != null){
				bw.write(inputString);
				bw.write("¥n");
			}

			br.close();
			reader.close();

			bw.close();
			writer.close();

		}catch(IOException e){
			e.printStackTrace();
		}

	}
}
```

このプログラムのポイントは,reader と writer を同時に開き, 読み込んだデータをすぐに writer で書き込んでいる点です.

これによって, プログラムは常に一行分のデータしか保存しなくてもすべてのデータをコピーすることが出来るのです.

### 12.2.5 CSV 形式の読み込み

次に, もう少し実用的なところで, CSV 形式のファイルの読み込みを行ってみましょう.

ＣSV 形式とは, データを,(カンマ)で並べたデータのことです.

```
10,20,30,40,15,46,15
```

**CSV 形式の例**

エクセルなどの表計算ソフトでも簡単に読み込むことができるため, 大量のデータをほぞんしたりするのによく使われます.

ここでは, 整数が書かれている CSV ファイルを読み込んで, 順番に ArrayList に入れていくプログラムを作ってみましょう.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;
import java.util.ArrayList;

public class CsvReader {
```

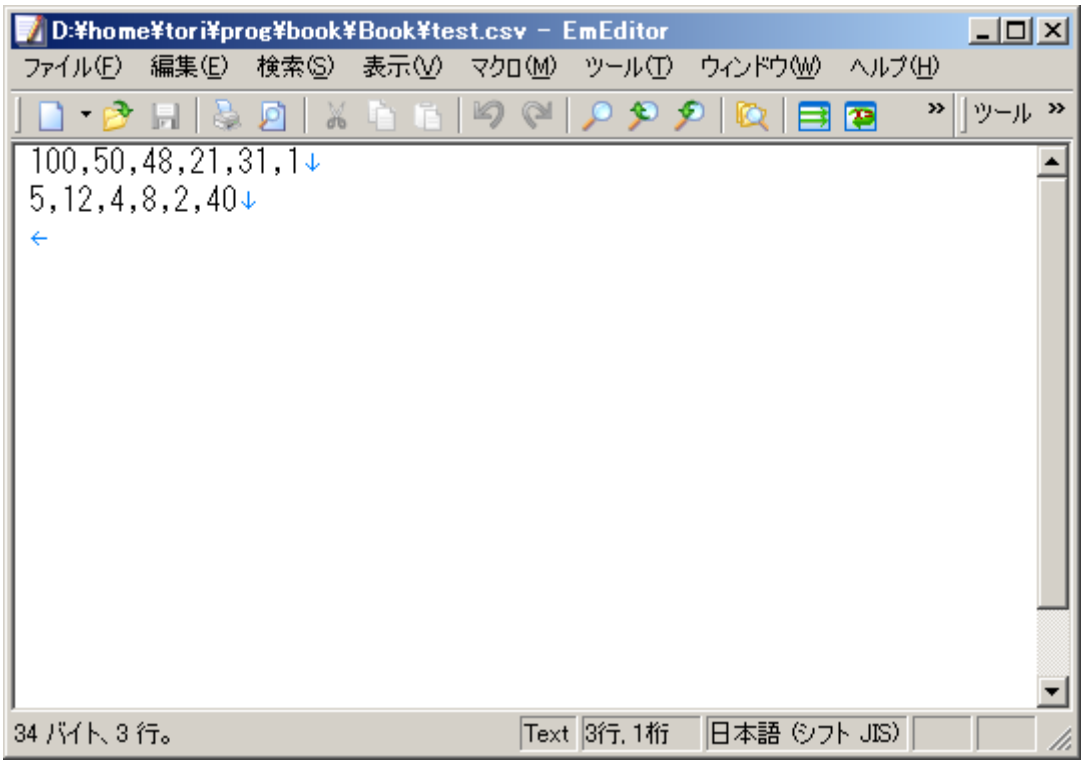
```
	public static void main(String[] args) {
		ArrayList<Integer> list = new ArrayList();
		try{
			Reader reader = new FileReader(args[0]);
			BufferedReader br = new BufferedReader(reader);
			String line = null;
			while((line = br.readLine()) != null){
				String[] data = line.split(",");
				for(int i = 0; i < data.length; i++){
					int d = Integer.parseInt(data[i]);
					list.add(d);
				}
			}
			br.close();
			reader.close();

		}catch(IOException e){
			e.printStackTrace();
		}
		int sum = 0;
		for(int data:list){
			sum += data;
		}
		System.out.println("合計="+sum);
	}

}
```

これで，読み込んだ CSV ファイルのデータをすべて ArrayList に入力することが出来ます．それだけだとつまらないので，どうせだからその合計値も求めてみました．

では，このプログラムで以下のような CSV ファイルを読み込んでみましょうか．




```
> java CsvReader test.csv
```

```
合計=322
```

というわけで，無事 CSV ファイルを読み込んでその合計値を出すことに成功しました．合計値が正しいかどうかはみなさんで確かめてみてください．

さて，ここでやっていることは基本的にはファイルの入力だけですが，二カ所だけ補足説明をしておきたいと思います．

まず，

```
String[] data = line.split(",");
```

この部分です．

個々で使っているのは String クラスの split というメソッドです．このメソッドは，指定された文字で文字列を分離して配列に保存します．

例えば，

```
String name = “部屋とＹシャツと私”;
String[]items= name.split(“と”);
for(int i = 0; i < items.length; i++){
          System.out.println(items[i]);
}
```

とすると，splitted 配列に，順番に「部屋」「Y シャツ」「私」という文字列が代入されます．これで，歌うときに Y シャツだったか T シャツだったか混乱する心配がありません．

```
部屋
Ｙシャツ
私
```

次に，

```
int d = Integer.parseInt(data[i]);
```

です．これはすでにラッパークラスのところで説明しましたよね．

ここでは，Integer クラスの static メソッドを parseInt 使って数字が書いてある String 変数を int 型の変数に直しています．

すでに説明したとおり，Java では文字列と数値は厳密に分けられています．つまり，”10”という文字列は，1 と 0 の組み合わせであって，数字の 10 ではないのです．

しかし，それだと色々不便です．

今回のようにファイルから BufferedReader#readLine を利用して文字列を読み込むと，本来数字のはずのデータも文字列としてしか取得できません．

したがって，合計を求めるのも大変です．”10”+”10”は”20”ではなくて，”1010”に鳴ってしまうのですから．

そこで，数字が書いてあると分かっている文字列についてのみ，parseInt を使えば，文字列を解析して数値に直してくれるわけです．

ただし，数字じゃない文字列を入れた場合，エラーが発生しますのでご注意を．

```
int d = Integer.parseInt(“五十”);
```

この結果は・・・

```
  Exception in thread "main" java.lang.NumberFormatException: For input string: "五十"
```

というわけで，Java は漢数字は読めないようです.

CSV ファイルは，エクセルのデータなどからも変換できますので，Excel に記録してあるデータを Java で高度な操作を行いたい場合などに是非使ってみてください.

# 12.3　　よりよい Java プログラミングのために

## 12.3.1 良いプログラムを書くためのテクニック

ここでご紹介するのは，必ずしも使わなくても良いテクニックの数々です.

守らなくても良いなら別にいいじゃん！

と思う方も多いかもしれませんが，これらのテクニックは，先人達が「より良いプログラム」を実現するために培ってきた知恵の数々です.

より良いプログラムとは，以下のような性質を持っているものであると考えます.

- プログラムのミスが少なくなる
- バグが発生したら原因がすぐ分かる
- 他人が読んですぐに理解が出来る

この三点に気を使ってプログラミングすることで，良いプログラムが作成可能となります.

１点目と２点目は比較的分かりやすいと思いますが，問題は３点目ですね．最初のうちは「他人にプログラムなんて見せないよ」と思うかもしれません．しかし，ここでいう「他人」とは．他の人ではなく**明日のあなた**だと思ってください．今日一生懸命プログラミングをしたところ，果たして明日もちゃんと覚えているでしょうか？いや，一日くらいなら覚えていられるかもしれませんが，１年後となったら同でしょう？１年前のプログラムを見て，すぐに全容を思い浮かべることが出来る人はあまり多くありません．少なくとも筆者には無理です.

そんなとき，良いプログラムを書いておけば，来年のあなたが久しぶりにプログラムを見ても，すぐにその使い方が分かるようになるわけです.

これは，意外と重要なことです.

是非，良いプログラムを書くためのテクニックを学んで，来年のあなたが迷うことなくプログラミングを再開できるようなものを作成しましょう.

### 12.3.2 命名ルール

Java のプログラミングをする時に，色々なクラスや変数などの名前を付けなければ行けません．もちろん，a とか b とかいう名前でも同じ動きをするのですが，人間が読んでわかりやすいネーミングを行うことはよりよいプログラミングを行う上で非常に重要です．

```
int a = 100;
```

と書くよりも，

```
int price = 100;
```

と書いた方がこの変数が何に使われるのか一目で分かりますよね．

このように，間違えにくいプログラミング実現のための命名法について述べていきたいと思います．

#### 12.3.2.1 予約キーワード

さて，Java では自由に変数名やクラス名を付けられますが，いくつかこの名前だけは付けちゃ駄目だよ，という名前があります．それが，予約キーワードです．

ようするに，Java のプログラミングをする上で，構文を記述するのに使うキーワード立ちのことです．

例えば，if という名前の変数を作成してしまうと，プログラムは if 文を表したいのか，if という変数を操作したいのか分からなくなってしまいます．

```
if(if == 0){ //ん！？
}
```

というわけで，Java プログラムで使うことが決まっているいくつかの名前は変数名，クラス名，メソッド名，いずれにも使うことが出来ません．

これら，利用できないキーワードを以下に示します．これらのキーワードは名前に使わないように注意しましょう．

**表．予約キーワード**

| | | | |
|---|---|---|---|
| abstract | double | int | strictfp |
| assert | else | interface | super |
| boolean | extends | long | switch |
| break | false | native | synchronized |
| byte | final | new | this |
| case | finally | null | throw |
| catch | float | package | throws |
| char | for | private | transient |
| class | goto | protected | true |
| const | if | public | try |
| continue | implements | return | void |
| default | import | short | volatile |

| | | | |
|---|---|---|---|
| do | instanceof | static | while |

#### 12.3.2.2 パッケージの命名ルール

パッケージの命名ルールについてはすでに 9 章で述べました.

基本的に，**所属する組織のドメイン名を逆順に並べて行ったもの**をパッケージ名としましょう.

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;
```

となります.

ただし，簡単なプログラムを書いていて，他人に公開する可能性がないのであればどんなパッケージ名でもかまいません．ただし，パッケージ名は小文字が推奨されています.

また，java,javax から始まるパッケージ名は付けてはいけません．これは，標準でついてくる java のパッケージが java,javax であるためです.

以上の点だけ気をつけてパッケージ名を作成しましょう.

例えば，携帯電話のアドレス帳関連クラスのパッケージであれば,

```
package phone.mobile.address
```

などになるでしょう.

#### 12.3.2.3 クラス・インターフェースの命名ルール

クラスの名前は,「正しくフルスペルで、最初の 1 文字は大文字とし、複数の単語をつなげるときは、続く単語の最初の 1 文字も大文字とする」ことが望ましいとされています. (Gosling,Joy,Steele,1996;Sandvik,1996;Ambler,1998a)。

また，クラス名は名詞であった方がわかりやすいでしょう．修飾語を付けてもかまいません.

例えば,

```
class Phone
class MobilePhone
class PersonalHandyPhoneSystem
```

などです.

```
class PHS
```

などの略語は使わない方がよいでしょう.

また，abstract クラスの場合，最初に Abstract と書くと良いかも知れません.

```
AbstractPhone
```

こうしておいて，実装する際には Abstract を取った名前を利用することが可能となります.

```
class Phone extends AbstractPhone
```

よく使うクラスの名前ほどわかりやすい名前にすることが望ましいので，一般の電話を表す Phone というクラス名は実体として作成可能なクラスに取っておいて，そのスーパークラスには Abstract を付けるというルールです.

次に，インターフェースの命名ルールですインターフェースもクラスと同様に，最初の 1 文字は大文字とし、複数の単語をつなげるときは、続く単語の最初の 1 文字も大文字とします．また，インターフェースの場合は，

```
interface Callable;
interface Ringable;
```

などの副詞も使われます.

また，どうしてもすでにある実体の名前をインターフェースにしたい場合は，I を頭に付けるという方法がとられることがあります.

```
interface Iclock;
interface Icamera;
```

もちろんこの I は Interface の I です.

#### 12.3.2.4 メソッドの命名ルール

メソッドの名前にはルールがあります。基本的には変数と同じように数字やアルファベットを使って書くことになります。

一般的なルールとして、メソッド名は**小文字からはじめ，複数単語からなるメソッド名は、途中に出てくる単語の最初の文字を大文字にする**，というものがあります．また一般的に動詞を使うというルールがあります.

```
void call();
void sendEmail();
```

などです.

また，フィールドの取得や設定を行うメソッドはアクセッサメソッドと呼び，特殊な命名法を適用します.

値を取得するアクセッサメソッドを getter(ゲッター)と呼びます．boolean 以外の値を取得する場合は，get の後にフィールド名を付加し，boolean 型の場合は is の後にフィールド名を付加したものをメソッド名とします.

```
int getBatteryPower();
boolean isMovable()
```

一方，値を設定するアクセッサメソッドは setter(セッター)と呼びます．setter はフィールド名の前に set を付けたものとなります.

```
void setMyPhoneNumber(String phoneNumber);
void setIsOpen(boolean isOpen);
```

また，アクセッサメソッドではありませんが，複数のオブジェクトを加えることができる setter の場合，フィールド名の前に add を付けるのが一般的です.

```
void addPersonalData(PersonalData newAddress);
```

通常メソッド名はそのメソッドを呼ぶことによって生じる現象を性格に述べたものを用いるのが望ましいですが，あまりにも長い場合は多少省略した方が良いでしょう．

#### 12.3.2.5 変数の命名ルール

変数はその種類によって命名方法を変化させます．

まず，**フィールド**はメソッドと同じく**小文字からはじめ，複数単語からなるメソッド名は、途中に出てくる単語の最初の文字を大文字にする**というルールを適用します．

```
int callPrice;
PersonalData personalData;
```

などです．

ただし，Collection フレームワークや配列を使う場合は，それに準じた名前を最後に付けるとわかりやすいでしょう．

```
int[] dayPriceArray;
HashMap<String, PersonalData> addressMap;
```

また，boolean フィールドに限っては名前の最初に is を付けることを推奨します．

```
boolean isBatteryCharging;
```

また，フィールドの中でも特殊な例である**定数**は，**全て大文字で書き，複数単語から鳴る場合は単語を_(アンダーバー)で区切る**というルールになります．

```
static final int MAX_ADDRESS_NUM = 100;
static final String TELEPHONE_CARRIER = "HardBank";
```

次に，ローカル変数の場合です．

ローカル変数の場合は同一メソッド内でしか使われないので，あまり細かく規定する必要はないかも知れませんが，一般的な命名法があります．

まず，ループ変数の名前です．ループ変数とは，for ループでループ回数を制御するために用いる変数です．これらの変数は通常 i,j,k などを用います．

```
for(int i = 0; i < 100; i++){
          for(int j = 0; j < 100; j++){

          }
}
```

また，while 文の制御にはフラグ f 等を用いることもあるようです．

```
boolean f = true;
while(f){

}
```

次に，try～catch で Exception を catch する場合，Exception 変数として e を用いるのが

一般的です.

```
try{

}catch(Exception e){

}
```

このとき，Exception の種類によらず，e を用います.

これら以外のローカル変数には短めの名前をつけることが多いようです．またあるクラスの変数を作成する場合は，クラス名の最初の大文字を小文字にした名前をつけると分かりやすいでしょう.

```
Calendar calendar = Calendar.getInstance();
```

また，配列の名前としては ary などを使うこともあります.

```
int[] ary;
```

コレクションフレームワークの場合，List 型，Set 型，Map 型でそれぞれ以下のように名づけることが多いようです.

```
ArrayList<String> list;
HashSet<String> set;
HashMap<String, Integer> map;
```

ただし，これらはメソッドの中で利用方法が明確で他に重複したクラスのインスタンスが存在しない場合に限ります.

同じクラスのインスタンスが存在した場合は，フィールドにつける場合と同じようにどのようなインスタンスなのかを示した名前をつけることが望ましいでしょう.

```
Calendar today;
Calendar yesterday;
ArrayList<String> nameList;
ArrayList<Integer> priceList;
```

#### 12.3.2.6 名前の重複

プログラムを書いていると，フィールドとローカル変数で名前が重複しそうになることが良くあります．特に，setter を作った場合，引数をそのままフィールドに代入することが多いので，両方同じ名前にしてしまいそうになることが良くあります.

しかしながら，このようなプログラムを書いてしまうと・・・

```
        int price;
        public void setPrice(int price) {
                price = price;
        }
```

メソッド内の price は引数の price とフィールドの price どちらになるのでしょうか？

実は，このように**ローカル変数とフィールドの名前が重複した場合，ローカル変数が優先される**というルールがあります．したがって，ここではローカル変数に price にローカル変数 price の値を代入していることになります．フィールド price は蚊帳の外.

混乱を避けるためフィールドとローカル変数の名前は異なるものにしておいたほうが無難ですが，明示的にフィールドを呼び出す方法も用意されています．それが，キーワード **this** です．

あるインスタンスのフィールドを呼び出すときは，インスタンス名に.をつけてフィールド名を書けばよかったですよね．

```
personalData.name;
```

これと同じように，インスタンス名を書けばフィールドを呼び出すことが出来るのですが，インスタンス自信は自分の名前がわかりません．そこで，Java では特別に this という仮想的なインスタンスが用意されており，this からアクセスされたフィールドなどは自分自身のフィールドになるのです．

これを利用すると，ローカル変数とフィールドに重複した名前があったとしても，

```
	int price;
	public void setPrice(int price) {
		price; //ローカル変数
		this.price; //フィールド
	}
```

二つの変数を明確に分けることが出来ます．

したがって，この setter で引数をフィールドに代入するのであれば，

```
	int price;
	public void setPrice(int price) {
		this.price = price;
	}
```

とすればよいのです．

変数名を考えるというのは意外と面倒な作業なので，同じ変数名を使いまわすこのテクニック，覚えておいて損はありません．

以上，命名法について簡単に説明してきました．必ずしもこのような命名法を守る必要はありませんが，分かりやすいプログラミングを心がけるためにも，是非これらの命名法を利用してください．

### 12.3.3 コーディングスタイルの統一

Java プログラムは結構柔軟で，規約さえ守っていればどんな書き方をしてもかまいません．

しかしながら，やはり読みやすい書き方というのは存在します．

例えば，こんなプログラムどう思いますか？

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

int
price;
	price
	=
100;price+=5;
```

```
                    if(price >= 100)
                    System.out.println("100円以上");
                    else{System.out.println("100円以下");
}
          }
}
```

プログラムとしては間違えていませんが，非常に読みづらいことこの上ありません．

プログラムとしては，price 変数に 100 を代入して，さらに５を足した後１００円以上かどうか判断しているのですが・・・

そんなわけで，プログラムを書くときは，ある程度書き方を決めておいた方が誰にでも見やすくなります．そのようなプログラムの書き方をコーディングスタイルといいます．

プログラムを書くときの注意点をいくつかここで述べたいと思います．

#### 12.3.3.1　一行一命令

Java では，一つの命令は;までということが決まっています．そのため，一行にたくさんの命令を詰め込むことも出来ますし，逆に一つの命令を複数行に分けることも出来ます．

しかし，そうすることによってプログラム全体の長さが短くなったり長くなったりはしますが，実効時間には変化ありませんし，プログラムが読みにくくなる可能性があります．

そこで，一行に一命令を書くことをお勧めします．

先ほどの分かりづらいプログラムをこのルールに従って直すと，こうなります．

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

int price;
          price = 100;
price+=5;
                    if(price >= 100)
                    System.out.println("100円以上");
                    else{
System.out.println("100円以下");
}
          }
}
```

どうでしょう？

先ほどよりは大分みやすくなったような気がします．

#### 12.3.3.2 ブロックごとにインデントする

インデントとは，各行の先頭に空白，またはタブを入れることをいいます．これによって，ブロックの関係が分かりやすくなります．

先のプログラムをブロックごとにインデントしてしまいましょう．

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

                              int price;
                              price = 100;
                              price+=5;
                              if(price >= 100)
                                        System.out.println("100円以上");
```

```
                    else{
                        System.out.println("100円以下");
                    }
                }
}
} //←あれあれ？
```

ちゃんとブロックごとにインデントの結果がこうなりました.

インデントをきちんと書くと，自動的に

- クラス宣言はインデントなし
- フィールドやメソッドの宣言はインデント一つ
- メソッドの中身はインデント二つ

となります.

こうすることによって，プログラム全体の見通しがかなり良くなります.

しかも，もう皆さんも気づいていると思いますが，実は最初のプログラム，{}の数があっていなかったんですね．つまり，正しくブロックが作られていなかったのです.

最後に，一つ余分な{があまってしまっているのが分かります.

このように，インデントを正しく作ることで，{} の閉じ忘れや余分な}などが一目で分かるようになるのです.

インデントをちゃんとつけて無駄なミスがなくなるようにしましょう.

#### 12.3.3.3　１行でもブロックを

if 文などを書いたとき,条件にマッチした場合,ブロックの中身が実行されます.ただし,ブロックがない場合直後の１行だけ実行されるというルールがあります.

したがって,

```
                if(price >= 100)
                    System.out.println("100円以上");
```

この部分はブロックがなくても実行されるわけです．これは，if 文だけではなく for ループと while ループにおいても適用されるルールです.

しかしながら，このような書き方はあまり推奨されません．なぜなら，もしこの if 文に対してさらに命令を追加しようと思った場合に困ってしまうからです.

```
                if(price >= 100)
                    System.out.println("100円以上");
                    System.out.println("まだ余裕"); //①
```

と書こうとすると，{}無しの場合は一行しか if 文の対象とならないため，price が 100 未満であったとしても①の行が実行されてしまうのです．つまり，このように書いてある場合と一緒ですね.

```
                if(price >= 100){
                    System.out.println("100円以上");
                }
                System.out.println("まだ余裕");
```

このようなことがないように，if 文や for 文，while 文を書く場合は必ずブロックを指定

するようにしましょう.

```
public class StyleMain {

		public static void main(String[] args) {

			int price;
			price = 100;
			price+=5;
			if(price >= 100){
				System.out.println("100 円以上");
				System.out.println("まだ余裕");
			}
			else{
				System.out.println("100 円以下");
			}
		}

}
```

以上を考慮してプログラムを書き直すとこうなります.

大分みやすくなりましたね.

#### 12.3.3.4 ブロック {} の位置

ブロックを示す{}の位置は比較的自由につけることが可能です. 例えば, こんなことも可能です.

```
			if(price >= 100)


			{
				System.out.println("100 円以上");
				System.out.println("まだ余裕");
			}
```

間に何もなければ, 空白は無視されます. しかし, これは読みづらいですよね. 空白を作る意味もありませんし.

当然のようにブロック{}は if 文の直後に書くのが望ましいでしょう.

ただし, このときに種類の書き方があります.

```
			if(price >= 100)
			{
				System.out.println("100 円以上");
				System.out.println("まだ余裕");
			}
```

とかく場合と

```
			if(price >= 100){
				System.out.println("100 円以上");
				System.out.println("まだ余裕");
			}
```

と書く場合です. どちらも意味は同じです. 好きな方で書いてかまいません. ただし, 一般的には後者の書き方のほうが多いようです.

### 12.3.4 アクセッサの勧め

フィールドは private にしてアクセッサを利用する事がお勧めです．これは，プログラムを作成しているうちにフィールドの値に何らかの制限を加えたくなったときなどに，プログラム全体を変更しなくてもフィールドを持つクラスのアクセッサを変更すればすむようになるからです．

```
        private int price;

        public int getPrice() {
                return price;
        }

        public void setPrice(int price) {
                if(price < 0){
                        return;
                }

                this.price = price;
        }
```

例えば，こうすることで確実に price の値がマイナスになることが防げます．

あるいは，set メソッドを最初から作成しなければ，途中で値を変更することができないクラスにすることも可能です．

最初のうちはいちいち set～と get～というメソッドを作るのが面倒・・・と思うかもしれませんが，カプセル化というオブジェクト指向プログラミングの考え方ではもっとも基本的な考え方といっても良いものですので，積極的に使うようにしてください．

### 12.3.5 コメントを書こう

通常プログラムを書く場合，それぞれの場所で何をやっているのか，コメントを書くことがあります．コメントを書くことによって，「将来的にいったいこのプログラムは何をやっているんだ？」ということを思い出しやすくすることができます．

Java では，クラスやインターフェース，そしてフィールドとメソッドについては，「このようにコメントを書きましょう」という決まりがあります．

実は，この決まりに従ってコメントを書いておくと，コマンド一つで各クラスの仕様書を自動的に作成してくれる機能があるのです．具体的にどうすれば仕様書を作成できるかは次節で紹介するとして，ここではコメントの書き方だけ簡単に説明します．

#### 12.3.5.1 クラス・インターフェースのコメント

クラスとインターフェースに関するコメントを各場合は，以下のようにします．

ここでは，PersonalData クラスを例にとっています．

```
/**
  アドレス帳に載せる個人データ一人分を表すクラスです．<br>
  名前のほかに，住所，電話番号などを指定可能です．
  @author tori
```

```
 */
public class PersonalData {
}
```

このような書き方でコメントを書きます.

通常のコメントは

```
/*コメント*/
```

のように書きますが，クラスの仕様書を書くためのコメントは,

```
/**
コメント
*/
```

と記述します.

また，改行する場合は<br>と書きます．これは改行を示す記号ですのでそのまま覚えておいてください. HTML を勉強したことがある人は改行マークだとすぐわかると思います.

このとき，クラスについてできるだけ詳しい説明を載せると良いでしょう．ここに書いた内容がそのまま仕様書となります.

このとき，クラスやインターフェースに関するコメントは，**クラス・インターフェース宣言の直前に書かなければいけません**.

これは重要ですので，覚えておいてください.

また,

```
  @author tori
```

は作者を示す部分です.

@author の後に空白を入れて，名前を書けば，このクラスを誰が書いたのかが一目でわかるようになります．これは，そういう決まりですので，クラスを一つ作ったら自分の名前，または自分だとわかる印を残しておいてください.

では，クラス・インターフェースに関するコメントの書き方をまとめますね.

```
/**
  クラスの説明
  @author 作者名
 */
public class クラス名 {
}
```

#### 12.3.5.2 フィールドのコメント

フィールドに関するコメントは，フィールドの定義の直前に記述します.

```
public class PersonalData {
          /**
            名前
           */
          String name;
```

クラス・インターフェースのコメントと同様,

```
/**
コメント
*/
```

と記述します.

特に難しいことはありませんので，**フィールド定義の直前に記述しなければいけない**ということだけ覚えておいてください.

#### 12.3.5.3 メソッドのコメント

メソッドのコメントはちょっとだけ書くことが他よりも多いです.

```
/**
  住所を取得します
  @return 住所
 */
public String getAddress() {
          return address;
}


/**
  住所を設定します.
  @param address 住所
 */
public void setAddress(String address) {
          this.address = address;
}
```

ここでは,二つのメソッドを使って説明したいと思います.

メソッドのコメントも，フィールドなどと同様**メソッド定義の直前に記述牛なければいけません**.

また，そのときの

```
/**
コメント
*/
```

のように特殊なコメント記号で囲みます.

次に，まず getAddress メソッドのコメントを見ると，

```
/**
  住所を取得します
  @return 住所
 */
```

住所を取得する，という説明の後に，@return という記述があります．これは，返り値が何であるかの説明を記述する部分になります．ここでは，住所を返しますので，返り値の説明は「住所」となります.

次に，setAddress メソッドのコメントを見ると，

```
/**
  住所を設定します.
  @param address 住所
 */
```

となっています．ポイントは，@param address ですね.

これは，引数の説明になります．setAddress メソッドには，address という引数があります．この引数には何を入れればよいのかをここでは説明します．この場合，住所を引数としてあたえるので，「住所」と書いてあります．

もし，複数の引数がある場合は，それぞれの引数についてコメントを書かなければいけません．順番は問いませんが，通常引数が書かれている順番に書いた方がよいでしょう．

まとめると，メソッドのコメントは下記のように書くことになります．

```
/**
メソッドの説明
@param 引数A 引数Aの説明
@param 引数B 引数Bの説明
・・・
@return 返り値の説明
*/
int method(int 引数A, String 引数B,…){
}
```

なんでこんな面倒な決まりがあるんだろう？と思うかもしれませんが，次の JavaDoc の項を読めば謎はすぐ解けると思います．もう少し待ってください．

## 12.4　調べ物をするときは

長かった Java の旅もいよいよ終わりに近づいてきました．

最後に，本書では扱え切れなった様々な情報をどのようにすれば手に入れることが出来るかを簡単にお伝えしたいと思います．

### 12.4.1 JavaDoc

ArrayList などの JavaAPI にはご紹介したものよりもはるかに多くの機能がついていたりします．それらの機能を調べるのにはどうすればよいでしょうか？

Java ではあるクラスに関する仕様書はJavaDoc と呼ばる HTML で公開されます. HTML で書かれていることからもわかるとおり，JavaDoc はホームページ形式になっておりブラウザを使って簡単に参照できます．

特に，付属の API の JavaDoc は全て Sun の HP で公開されていますので，ArrayList や HashSet などJavaAPI に属するクラスの詳しい情報などは簡単に取得することが出来ます．

2007 年 4 月現在, 最新版の Java である Java6 の JavaDoc を例にとって解説していきましょう．Java6 の JavaDoc は

http://java.sun.com/javase/ja/6/docs/ja/api/index.html

にあります.

このページを開くと，図１のようなページが開きます.


**図 1 JavaDoc**

ここで，左下に注目すると「すべてのクラス」と書いてありますよね．ここに JavaAPI に含まれる全てのクラスが記述されています.

そのため，量が多いのですが，ページ内検索を使って ArrayList を探してみてください．見つかりましたか？見つかったら，クリックしてください．右側のフレームが ArrayList の説明に変わると思います.

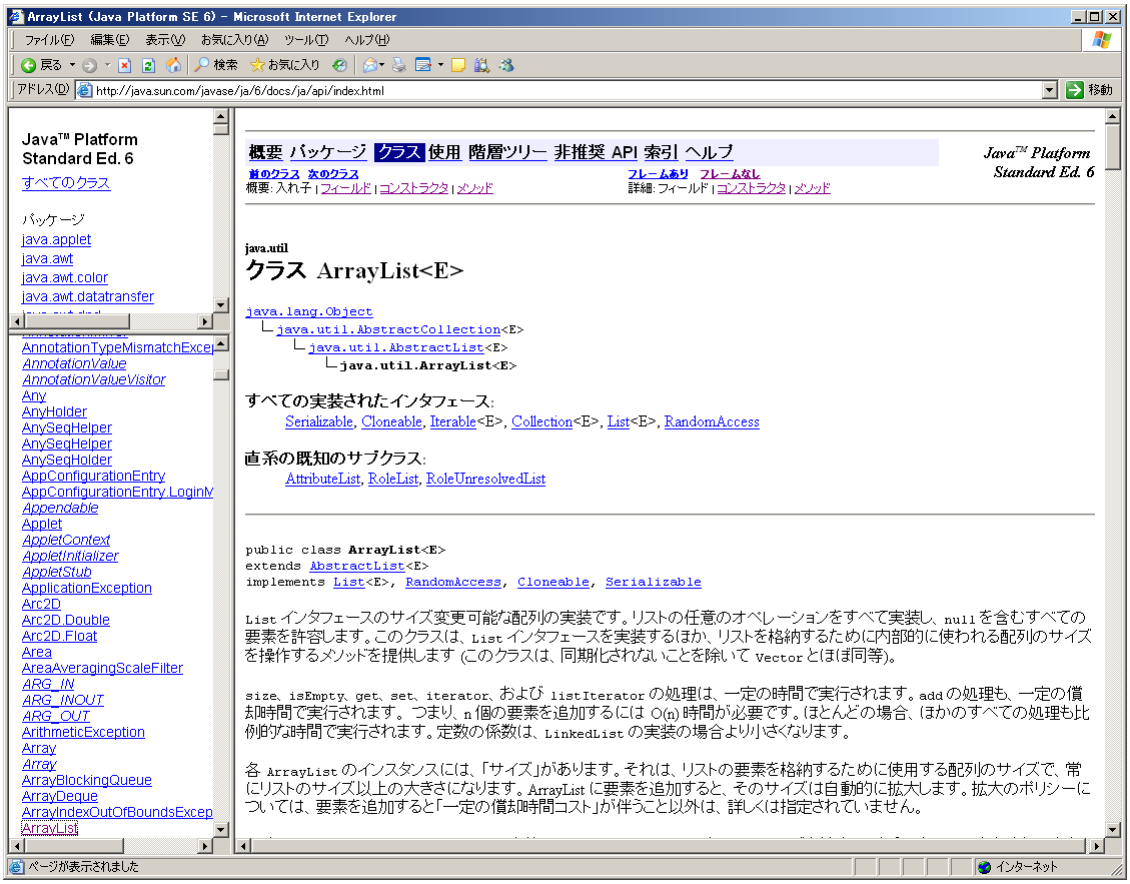



図 2 ArrayList の表示

この右側のフレームに ArrayList に関する詳しい情報が詰まっています.

JavaDoc にはそのクラスの仕様に関する詳しい情報が詰まっています．ここを読めばクラスの基本的な使い方は分かるはずですので，その読み方を学んでみましょう.

#### 12.4.1.1 継承関係

JavaDoc では最初に継承関係についての記述があります．あるクラスの継承関係が知りたい場合は，まずここから見ていきましょう.

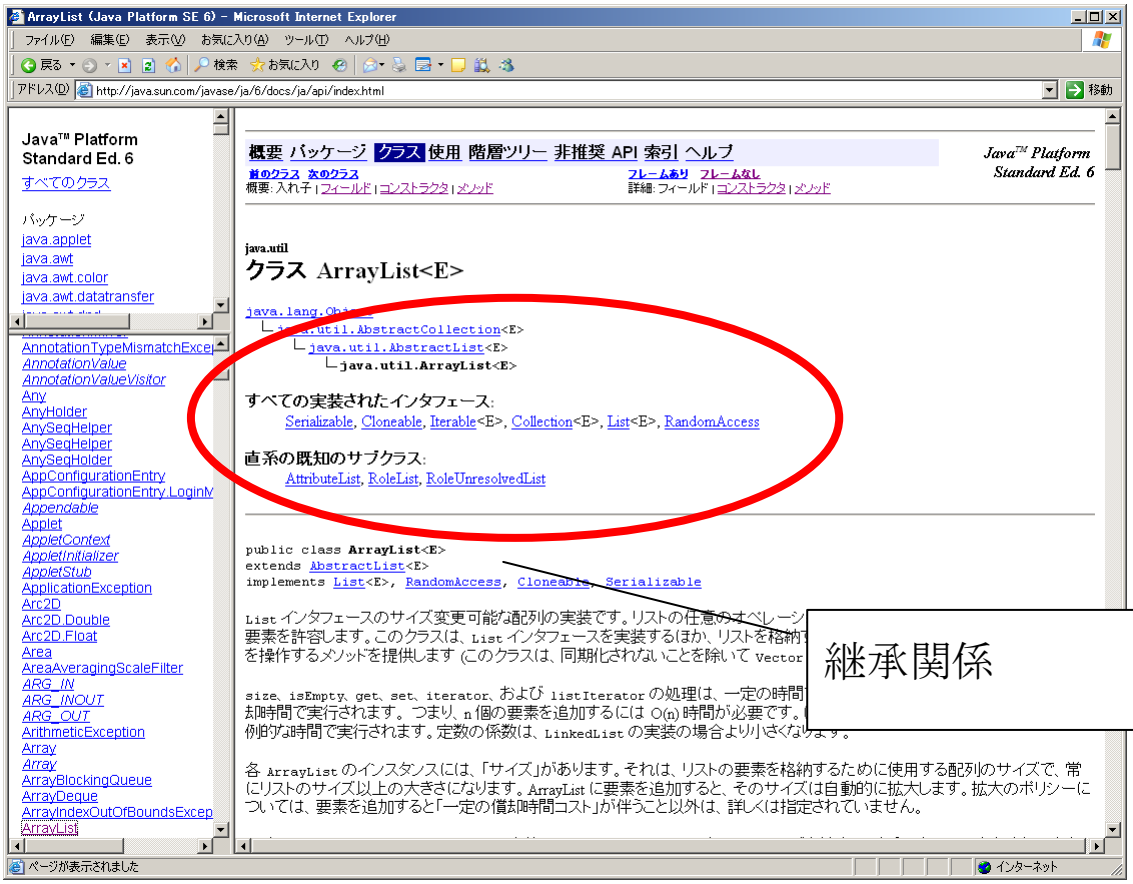



**図 3 継承関係**

まず，クラス名の直後，この部分を見てみましょう．

```
java.lang.Object
  └java.util.AbstractCollection<E>
      └java.util.AbstractList<E>
          └java.util.ArrayList<E>
```

これは，ArrayList の継承関係を示しています．この場合 ArrayList は Object クラスから派生した AbstractCollection クラスから派生した AbstractList クラスから派生していることを示しています．

また，その下を見ると，

すべての実装されたインタフェース:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

とあります．ここに，ArrayList が実装しているインターフェースをすべて記述されています．

ArrayList はこれらの全てのインターフェースを実装しているわけです． ArrayList が書いてあるソースファイルを読めば，

```
public class ArrayList<E> extends AbstractList<E>
        implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess {
          //中略
}
```

などと書いてあるかもしれません．

更にしたには，

**直系の既知のサブクラス:**
AttributeList, RoleList, RoleUnresolvedList

このような記述があります．ここには，ArrayList クラスを継承したクラスが書いてあります．

```
public class AttributeList extends ArrayList{
}
```

などの関係にあるクラス達ですね．

このように，JavaDoc を読めばそのクラスの継承関係がほとんど分かるようになっています．

#### 12.4.1.2 クラスの具体的な説明

継承関係の次に記述されているのが，クラスの具体的な説明です．ここを読めば，どのような場合にこのクラスを使えばよいのか，使うときの注意点などが書いてあります．

まずはここを読んで何に使うクラスなのかを把握するのが良いでしょう．

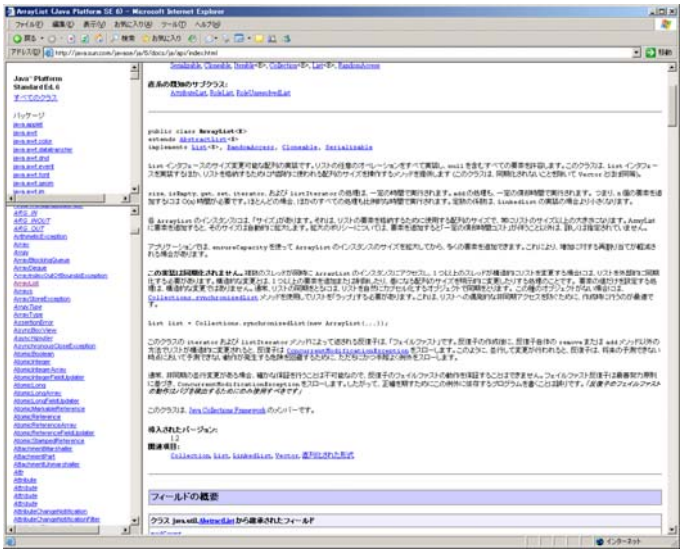


図 4 クラスの説明

### 12.4.1.3 フィールド，コンストラクタ，メソッドの一覧

クラスの説明の後には，フィールド，コンストラクタ，メソッドの一覧があります．この一覧から，どんなメソッドやフィールドを利用可能かが分かります．なお，フィールド名，コンストラクタ名，メソッド名をクリックすると，それぞれの詳細な説明に移動します．

また，引数や返り値のクラス名をクリックすれば，それぞれのクラスの説明に移動します．

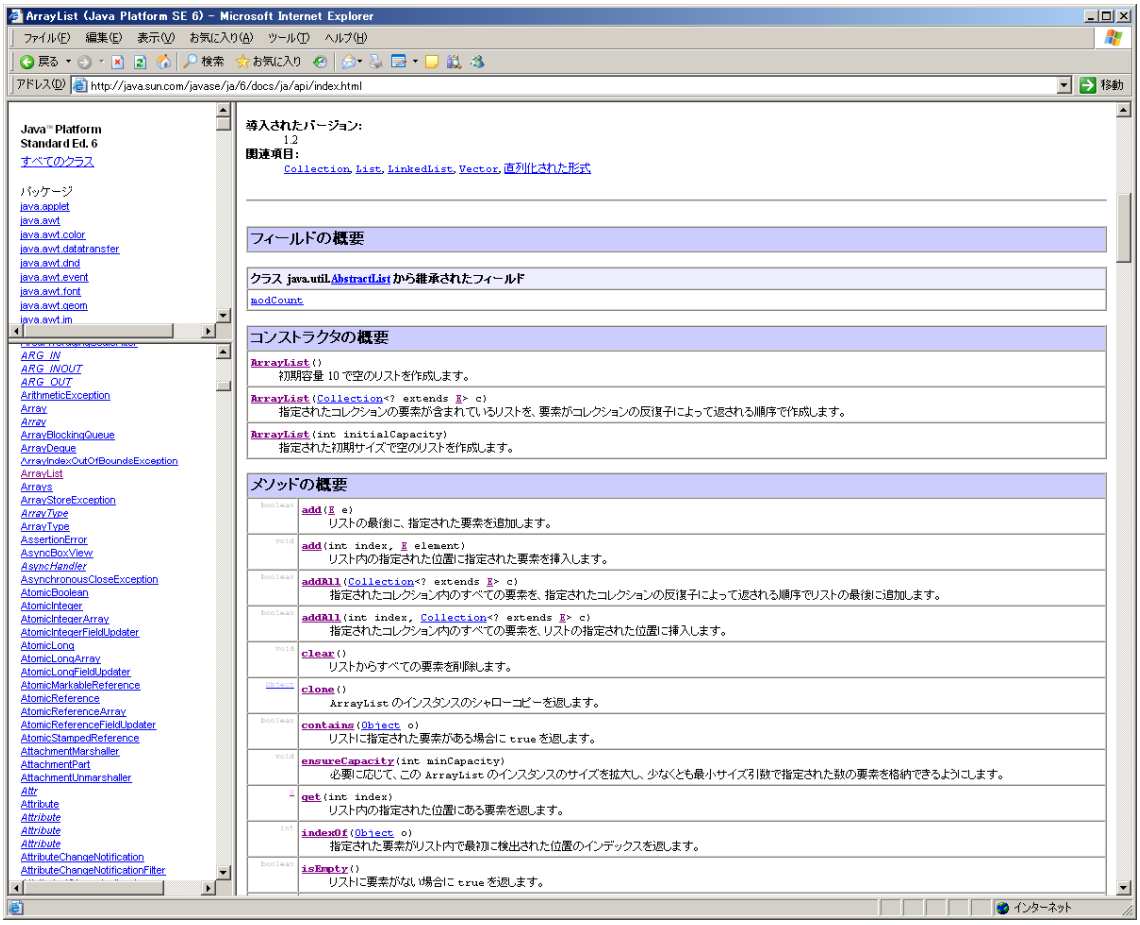



図 5 フィールドとメソッドの一覧

### 12.4.1.4 フィールドの詳細

フィールドの詳細は，それぞれのフィールドがどんな型で，何を意味するのかが書いてあります．

フィールド名，型名，説明の順で書いてあります．例えば，Calendar クラスの YEAR フィールドであれば，

**YEAR**
public static final int **YEAR**
get および set のための、年を示すフィールド値です。これはカレンダ固有の値です。サブクラスのドキュメントを参照してください。

**関連項目:**
**定数フィールド値**

のように書かれています.

ここから，Calendar.YEAR は static で final かつ public な整数型のフィールドであり Calendar クラスで，get，set するときに「年」をあらわすために利用することが分かります.

#### 12.4.1.5 メソッド・コンストラクタの詳細

フィールドの詳細の次に各コンストラクタ，メソッドの詳細情報があります. ここでは，ArrayList の remove メソッドを例にとって説明します.

## remove

public E **remove**(int index)
**リストの指定された位置にある要素を削除します。後続の要素は左に移動します（インデックス値から 1 を減算)。**
**定義:**
**インタフェース** List<E> **内の** remove
**オーバーライド:**
**クラス** AbstractList<E> **内の** remove
**パラメータ:**
index - **削除される要素のインデックス**
**戻り値:**
**リストから削除した要素**
**例外:**
IndexOutOfBoundsException - **インデックスが範囲外の場合** (index < 0 || index > size())

これは ArrayList の中から任意の添え字に対応する要素を削除するメソッドです.

まず，最初にこの**メソッドの定義**が書かれています. ちなみに,

```
public E remove(int index)
```

の E は ArrayList を作成したときの,

```
ArrayList<String> list = new ArrayList<String>();
```

<>内に囲まれたところに書いた，このリストに挿入可能なクラスとなります. この例だと String になります.

```
public String remove(int index)
```

と読み替えることが出来ます. これは実際にプログラムに何を書くかによって変更され

るわけですから，JavaDoc では E という文字で代用しているわけですね．

そして，定義の次には**説明**が書いてあります．この説明と第 10 章に図入りで説明してありますので見比べてください．同じ事が書いてあるはずです．

その次にある**定義**と**オーバーライド**はどのインターフェースやクラスでこのメソッドが定義されているかを示しています．定義にはこのメソッドが書かれているインターフェース名が書かれていますし，オーバーライドはどの継承元クラスのメソッドをオーバーライドしているかが書いてあります．

次に**パラメータ**ですが，ここには引数の説明が書いてあります．定義の所で，

```
public E remove(int index)
```

と書かれていますよね．つまり，このメソッドには int 型の index という引数があるわけです．パラメータには，この引数 index についての説明が書いてあります．この index で指定した添え字の要素が削除される訳です．

なお，引数が複数ある場合は，パラメータには全ての引数についての説明が書かれます．

次に，**戻り値**はこのメソッドを利用したときの返り値が書いてあります．通常メソッドの「返り値」と呼ぶのですが，なぜか JavaDoc では戻り値という言い方をします．二つは同じものですので，間違えないようにして下さい．

ここでは，リストから削除した要素そのものが返されることが分かります．

最後に，**例外**です．ここでは，発生する例外とその条件が書いてあります．このメソッドの場合は，ArrayList が持っている要素数以上の添え字を指定したり，0 未満の添え字を指定したりすると IndexOutOfBoundsException という Exception を投げることが分かります．

このように，JavaDoc を読めば各クラスの仕様が分かるようになっています．

もし，JavaAPI を使っていて分からないことがあったら JavaDoc を読んでみると解決するかも知れません．

ただし，JavaDoc は Java についてそれなりに理解がある人向けに書いてあります．最初の内は何を書いてあるのかよく分からないことが多いかも知れません．ですが，分かるようになってくれば，これほど便利なものはありませんので JavaDoc の読み方に慣れておくようにしましょう．

#### 12.4.1.6 JavaDoc の作成

さて，JavaAPI などの JavaDoc はどのようにして作っているのでしょうか？実は，人の手で一生懸命作っているわけではなく，Java のプログラムから自動生成しているのです.

さて，前節でクラスやメソッドについてコメントを書くときの書き方を勉強しましたよね．あれは、実は JavaDoc を自動生成するためだったんです。

では、早速作り方をご紹介します。JavaDoc を作成するコマンドは、そのまんま javadoc といいます。

javadoc コマンドに、ディレクトリと JavaDoc を作成するクラスのソースファイルを指定します。こんな感じ。

```
javadoc -d doc PersonalData.java
```

ここでは、doc というディレクトリに PersonalData.java の JavaDoc ファイルを作成します。

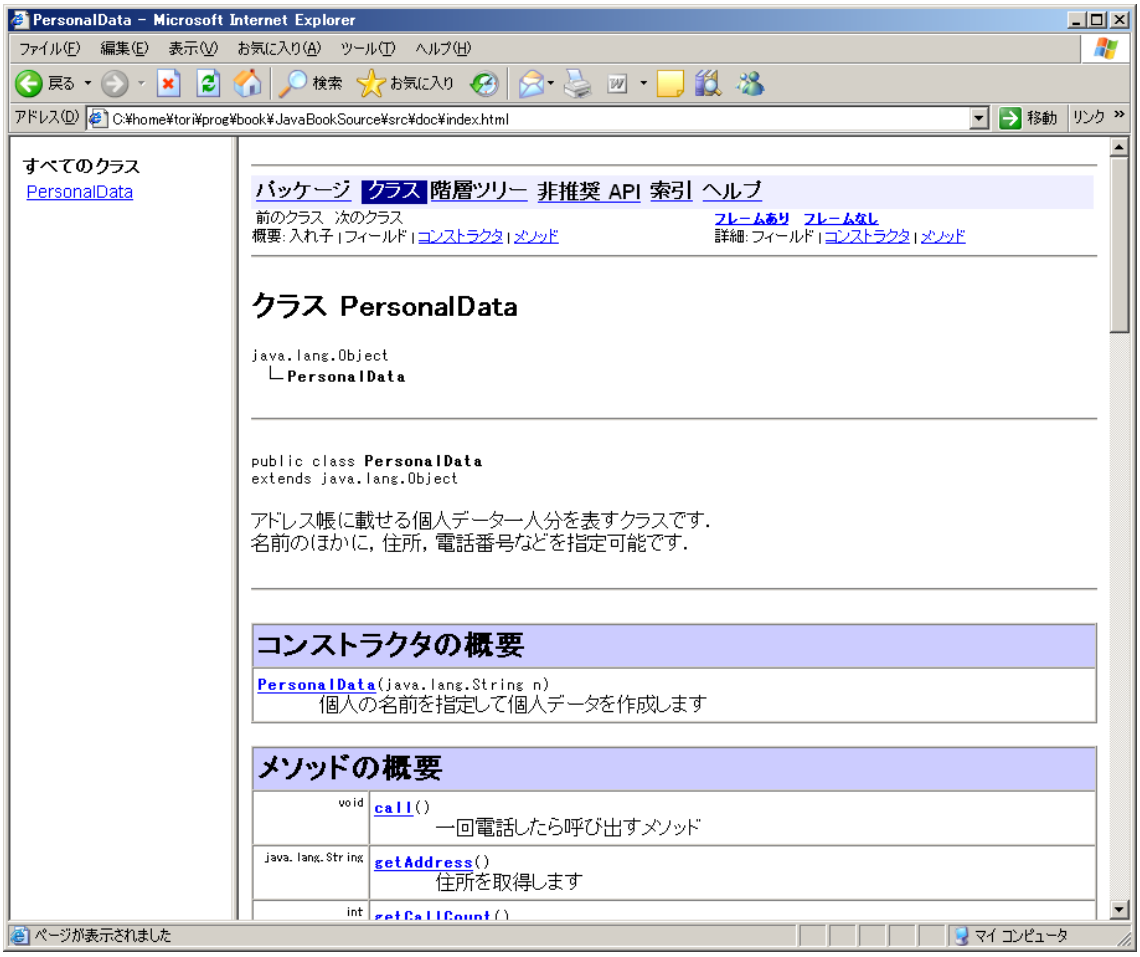
すると、

```
C:¥java>javadoc -d doc PersonalData.java
宛先ディレクトリの作成中: "doc¥"
ソースファイル PersonalData.java を読み込んでいます...
Javadoc 情報を構築しています...
標準 Doclet バージョン 1.6.0
全パッケージとクラスの階層ツリーを作成しています...
doc¥PersonalData.html の生成
doc¥package-frame.html の生成
doc¥package-summary.html の生成
doc¥package-tree.html の生成
doc¥constant-values.html の生成
全パッケージとクラスのインデックスを作成しています...
doc¥overview-tree.html の生成
doc¥index-all.html の生成
doc¥deprecated-list.html の生成
全クラスのインデックスを作成しています...
doc¥allclasses-frame.html の生成
doc¥allclasses-noframe.html の生成
doc¥index.html の生成
doc¥help-doc.html の生成
doc¥stylesheet.css の生成

C:¥java>
```

これで終了です。みると、新しいディレクトリが作成され、中にいくつかの htlm ファイルがあることが分かります。

その中の、index.html を開いてみましょう。




すると、こんな感じで PersonalData の情報が取得できます。

ぱっと見た感じ JavaAPI の JavaaDoc と同じですね。このように、ちゃんとコメントを書いておけばコマンドひとつでそのクラスの使い方が分かるようになるわけです。

今後大規模なプログラムを作る機会があったら、自分のプログラムの JavaDoc を準備しておくようにしましょう。

### 12.4.2 検索の勧め

Java は世界的にも利用者の多い言語ですので、インターネットを検索するといろいろな情報がヒットします。それらの情報を検索することで、本書では触れられなかった部分や、まだ情報が不足していた部分などについて理解を深めることができるでしょう。

ここでは、情報検索のコツをいくつかご紹介します。

まず、何か Java でしたいことがあった場合です。その場合は、

| *やりたいこと*　Java |
|---|

で検索すると良いでしょう。たとえば、ファイルの入出力の方法を知りたい場合は、

| ファイル入出力　Java |
|---|

とします。

これで、ファイルの入出力の方法について書いてあるページがいくつか出てくると思います。具体的なソースコードが書かれている可能性もありますので、それらのページにあるソースコードをそのままコピー&ペーストすれば使うことができるでしょう。

また、ある JavaAPI の使い方が分からない場合は、以下のような検索をするとよいでしょう。

| *調べたいAPI名*　Java |
|---|

たとえば、HashMap についてもう少し詳しい情報が知りたければ、

| HashMap　Java |
|---|

とすれば、HashMap の使い方が詳しく出ている HP を発見できると思います。

ただし、たいていの場合一番最初に SUN の JavaDoc のページが出てくると思いますので、JavaDoc 以外で情報を手に入れたい場合はそれ以外のページでそれっぽいタイトルのページを探しましょう。

検索にはかなりの慣れが必要となりますが、インターネット情報には Java を勉強する上で有用な情報がつまっていますので、有効活用しましょう。

## 12.5　今後学ぶべきこと

長かった Java の勉強もこれにて、ついに終了です。いかがでしたでしょうか？

しかしながら、本書で学んだことは奥深い Java の世界のほんのさわりにしか過ぎません。皆さんはこれからさらに Java について勉強を進めて行くことになると思います。

そのときに進めていくと良いであろうことをここではご紹介したいと思います。

### 12.5.1 swing

今回ご紹介したプログラムはすべてコマンドプロンプトから実行して終了するだけのプログラムでした。しかし、通常アプリケーションを作れば、Window が立ち上がってグラフィカルな操作ができますよね。もちろん Java でもグラフィカルなアプリケーションを作ることができます。そのために利用するのが **swing** という JavaAPI の一種です。swing は、グラフィカルインターフェースを Java で実現するためのクラス集です。

そのため、単にコマンドプロンプト上で動くだけではない、もっと派手なプログラムを作りたいと思ったら、swing の勉強が必須となります。

swing もかなり奥が深いのでここでは詳しい説明はしませんが、興味がある方は

```
swing java
```

などで検索してみましょう。

ここでは，簡単な Swing プログラムだけご紹介します．このプログラムは，文字入力を行うことができるウィンドウを一つ表示するだけの簡単な Swing プログラムです．

```
import javax.swing.JFrame;
import javax.swing.JTextArea;

public class SwingMain{

	static public void main(String[] args){
		JFrame frame = new JFrame();
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		JTextArea textArea = new JTextArea();
		frame.add(textArea);

		frame.setBounds(0, 0, 360, 240);
		frame.setVisible(true);

	}
}
```

詳しい説明は省きますが，此を実行すると，図のようなグラフィカルなアプリケーションを作成することが出来ます．

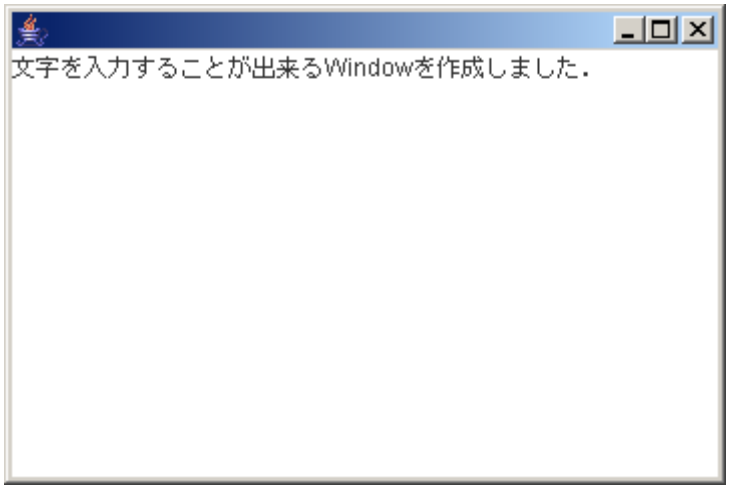



**図 6 グラフィカルな Window の作成**

もちろんこれだけでは何も出来ませんが，ここに色々な機能を追加して，アプリケーションを作成していくことが出来ます．

### 12.5.2 スレッド

スレッドは、簡単に言えば**複数の処理を同時並行して行う**方法です。

コンピュータの中のプログラムは、すべてスレッドという単位で動いています。複数のスレッドが同時平衡して動いていて、それによってコンピュータは動いているわけです。

たとえば、ワードを立ち上げればワードのスレッドが立ち上がりますし、インターネットエクスプローラを立ち上げれば、インターネットエクスプローラのスレッドが立ち上がるわけです。

これと同じように、Java プログラム内でも複数の操作を同時に行うことができるようになっています。このスレッドも勉強を始めるとかなり奥が深いものですので、興味がある方はぜひ調べてみてください。

### 12.5.3 Java アプレット

Java アプレットは HP などにおいておくとブラウザ上で Java を動かすことができる仕組みです。みなさんもインターネット上で Java のゲームを見つけてやったことがあるのではないでしょうか？

swing と似ているところがありますが､インターネットからダウンロードできるものということで､いろいろ制限がかけられています。たとえば、ファイルの読み書きができない、とか他のアプリケーションを利用できないなどです。

しかしながら、配布が気楽にできて、グラフィカルなプログラミングも容易にできますので、ゲームなどを作るのに比較的適しています。

非常に簡単なアプレットの例をご紹介しましょう.

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;

public class AppletSample extends Applet implements Runnable{

	int x;
	int y;
	int size;

	public AppletSample() {
		x = 10;
		y = 10;
		size = 20;
		setBackground(Color.GRAY);
	}

	public void start() {

		Thread thread = new Thread(this);
		thread.start();

	}

	public void paint(Graphics g) {
		g.setColor(Color.RED);
		g.fillOval(x, y, size, size);
	}

	public void run() {
```

```
                    int dx = 5;
                    int dy = 5;
                    while(true){
                              x += dx;
                              y += dy;
                              repaint();

                              if(x > getWidth()-size || x <= 1){
                                        dx = -dx;
                              }
                              if(y > getHeight()-size || y <= 10){
                                        dy = -dy;
                              }

                              try {
                                        Thread.sleep(30);
                              } catch (InterruptedException e) {
                                        e.printStackTrace();
                              }
                    }
          }
}
```

このプログラムから，AppletSample.class を作成したら，適当なディレクトリに設置して，そのディレクトリに以下のような HTML ファイルを作成してください.

```
<html>
<body>
<applet code="AppletSample" width="100" height="100"></applet>
</body>
```

そして，この HTML ファイルをインターネットエクスプローラなどのブラウザで読み込んでください.

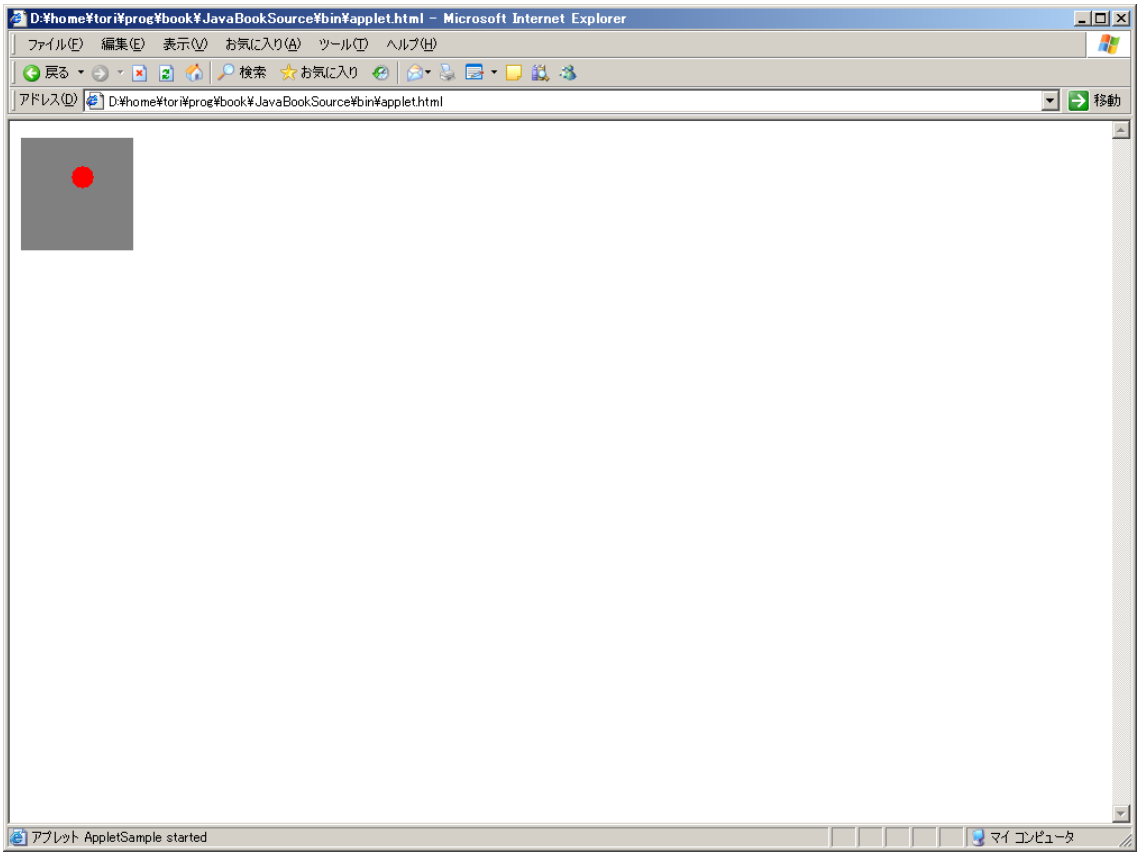



図 7 アプレット

図のような画面が立ち上がり，赤い玉が灰色のエリアを動き回るはずです.

ここで詳しい説明は省きますが，Applet クラスを継承した派生クラスを作ることで，簡単にアプレットを作成することができます.

興味がある型は，Applet，Java などで検索するとアプレットの作成方法が簡単に見つかるでしょう.

### 12.5.4 Java サーブレット

Java サーブレットはサーバーサイド Java とも呼ばれ、インターネット上の HP を動的に変更させるために使われます。

CGI などを Java によって作成するようなものだと思ってください。データベースとの親和性が高く、最近では大規模な WEB アプリケーションを作成するのに欠かせないものとなりつつあります。特に、業務用に使われることが多いので、将来皆さんがプロの Java プログラマになればサーブレットを使う機会も多々あるかもしれません。

ただし、Java サーブレットを使うには WEB サーバがないと駄目ですし、通常のレンタルサーバでは Java サーブレットの利用は認められていないことが多いようです。ですので、しばらくの間はあまり縁がないものかもしれませんね。ただ、こういったものもあるということだけ覚えておいてください。

### 12.5.5 デザインパターン

デザインパターンは、プログラムを書く上での決まりごとのようなものです。ある動作をするクラスを作ろうと思ったときに、どういうクラスを作って、どういう動作を実装すればよいのか、ということをパターン化したものです。

デザインパターンを使うことで、プロのプログラマたちが過去の経験から考えたパターンを実装することで、先人たちの知恵を拝借することがでます。また、同じデザインパターンを知っている人同士であれば、ひとつのプログラムを作るときの説明が楽になります。

「あのパターンを使って作ろう」

といえばすむようになるわけです。もし、お互いにデザインパターンを知らなければ、

「こういうクラスとこういうクラスをこう組み合わせて・・・」

といちいち話し合わなければいけません。そういった手間を減らすためにもデザインパターンは有効です。皆さんが将来的にプロ･プログラマを目指すのであれば、デザインパターンは知っていて損はありません。

ここでは一つだけ非常に簡単でわかりやすいデザインパターンの例をご紹介しましょう.

ここで考えるのは，地球をクラスとして作成してみようという試みです．ただし，地球が世の中にいくつもあっても困りますから，この世に地球インスタンスは一つしか作れないようにします．地球は一つ，という歌がありましたがそれを Java プログラムで実装してしまおうという大胆不敵な企画です．こんなときに使われるパターンが，シングルトンと呼ばれるパターンです．シングルトンとは，**そのクラスのインスタンスは一つしか作ることが出来ない**，というパターンです.

では，実際に Earth クラスを作成してみましょうか.

```
public class Earth {

	private Earth(){
	}

	static private Earth theEarth;

	static public Earth getInstanse(){
		if(theEarth == null){
			theEarth = new Earth();
		}
		return theEarth;
	}
}
```

ここでポイントとなるのが，Earth クラスのコンストラクタが private 型であるという点です．つまり，Earth クラス以外からは Earth クラスのインスタンスを作成できないわけです．でも，そうすると Earth インスタンスを作る Earth インスタンスを作れないから，Earth インスタンスは永久に作れないのでは，と思ってしまいそうですが，そこは大丈夫．Java には static メソッドというインスタンスを使わなくても実行できるメソッドがありましたよね．それを利用して getInstance というメソッドを利用して Earth クラス内部で作

成した Earth インスタンスを返しています.

ここで，getInstance が返すインスタンスを一つに限定しているので，インスタンスは一つしか存在しないことが保証されることになります.

ちなみに，この Earth クラスのインスタンスを取得する場合は,

```
Earth earth = Earth.getInstanse();
```

とします.

このように，あるパターンに沿った形でクラスを設計することである条件を必ず満たすようなプログラムを作成することが出来るわけです.

これ以外にも様々なパターンが考え出されておりますので，デザインパターンを学んで一歩上行くプログラマを目指してください.

さあ、今後 Java をどう勉強していくか決まったでしょうか？グラフィカルなアプリケーション作成を目指すのであれば，swing やアプレットを勉強するのもよいでしょう．また，WEB アプリケーションに興味があるのであれば，サーブレットの勉強が必須になります.

また，スレッドやデザインパターンはどんなプログラムを書くときでもきっと役立つ事と思います.

これから先は皆さんの目的に合わせていろいろな勉強方法があります。Java を使って実現したいことを思い浮かべながら、それに向けて勉強していってください。

本書が皆さんが Java をマスターするための第一歩として役立ったことを願いながら筆をおきたいと思います。