10 便利なクラス JavaAPI

10.1　JavaAPI とは何か？

さて、これまで Java の基本的な構文について勉強してきました。もちろんこれだけで Java プログラミングは十分可能です。が、Java には便利なクラス集がはじめから添付されています。

もし、C プログラミングをしたことがある方がいれば、文字列処理に苦労した思い出があるのではないでしょうか？C などでは、基本的な関数くらいしか用意されていなかったので、ちょっと高度なことをやろうとすると頑張って色々な関数を書かなければいけませんでした。しかし、Java は我々のような面倒くさがり屋さんのことをちゃんと考えてくれたのか、いろいろ便利なクラスを準備してくれています。

例えば、文字列一つとっても、C では char 型のポインタを操作しなければならず、メモリ関係のエラーに悩まされる人も多かったことでしょう。しかし、Java ならそんな問題に悩まされることなく簡単に文字列処理をすることが出来ます。

また、同じ Java においても配列は一度確保するとその大きさを変えることが出来なかったりして、不便です。しかし Java では、配列の大きさを自由に変えることができる便利なクラスも用意されています。

このように、さまざまなクラスが用意されているというのは、ただプログラミングが楽になるということだけが利点ではありません。JavaAPI のクラスを使ってプログラミングすれば、世界中の人が使い方が分かっているクラスを使えるという利点があります。つまり、自作のクラスを使ったプログラムを他の人が使おうとした場合は、そのクラスについて勉強しなければ使うことができません。どのメソッドで何が起きて、どのフィールドが何を意味しているのか分からなければ使えませんよね。それに対して、JavaAPI のクラスを使えば世界中の人が共通してしっているクラスですので、いちいち説明しなくても使ってもらうことができます。

また、もう一つの利点として、JavaAPI のクラスを使えばプログラムが間違っている心配がないということがあります。自分でクラスを作った場合、もしかしたらどこかでプログラムが間違っていてとんでもないエラーが発生するかもしれませんよね。絶対にエラーは発生させないぜ！という自信のある方がもしいたら、すばらしいですが、ちょっと高い鼻を低めにしておいた方が今後の Java 人生を送る上では無難だと思います。たいがい、プログラムミスをしてそのデバッグにプログラミングの時間の大半を取られるでしょうから。話がそれましたが、そんな自作クラスではどうしても発生してしまうバグですが、JavaAPI のクラスを使えば、少なくともそのクラス内にはバグがないことが保証されます。

たとえば、まったくでたらめな数字を次から次へと出してくる乱数という概念があります。簡単に言えば、サイコロを振って出た目を返してくれるメソッドを作るようなもので

す。一般にコンピュータでは本当の乱数を出すことは難しく、擬似乱数を発生させることが多いです。この擬似乱数を作成するクラス、作れますか？適当な数字を返すメソッドを作るのは結構難しいことなのです。一般的に知られているアルゴリズムもありますが、それらが本当の意味でバラバラの数字を返してくれることを保証するのは非常に困難です。もし、サイコロを振って一見バラバラの芽が出ているようで、実は偶数と奇数が交互にしか出なかったりしたらイヤですよね。そのため、擬似乱数を正確に作りたい。でも、そんなことに時間を割きたくない。そんな場合には、JavaAPI にある Random というクラスを使えば、えらい数学者の人が考え出したアルゴリズムによって乱数を返してくれるメソッドが準備されていて、たいていの場合はそれを使えば十分にバラバラな数値を返してくれることが保証されています。

このように、JavaAPI は

- 便利なクラスを大量に用意してくれている
- 世界中の人が同じクラスを扱える
- バグがないことを保証してくれる

という嬉しい三点盛りで皆さんをお待ちしています。

本章ではそんな便利な JavaAPI のクラスたちのうちよく使ういくつかをご紹介したいと思います。JavaAPI を学んで一歩上行くプログラマーを目指しましょう。

10.2　文字列操作～String～

10.2.1 String クラス

Java では文字列を扱うために便利なクラスが準備されています。すでにご紹介した String です。「おはよう」「こんにちは」「私ピーちゃん」など文字はすべて文字列クラスとして扱われます。そういう意味では、String クラスは Java の中でもかなり特殊な部類に入ります。

後で分かりますが、通常 JavaAPI を使う場合そのクラスを import しなければ扱うことができません。しかしながら、JavaAPI のなかでも java.lang パッケージにあるクラスだけは imoprt しなくても扱うことができます。簡単に言えば、何も書かなくても、

```
import java.lang.*
```

と書いてあることとして扱われます。そして、String クラスもこの java.lang パッケージ内に存在するのです。

では、文字列クラスを具体的に使っていって見ましょう。

10.2.2 String クラスの作成

文字列クラスは、Java のクラスの中でも特殊な部類に入ります。それは、作成方法にも現れています。

通常のクラスであれば、インスタンスは new キーワードを使って作成します。

```
String str;
str = new String("I am a pen");
```

これで、自分がペンであることを主張する素敵な文字列が完成しました。

が、String クラスはもっとお手軽に作成することができるのです。

```
String str;
str = "He is an apple";
```

これでもプログラムはちゃんと動いて、彼がリンゴであるという衝撃的事実を教えてくれる文字列が完成しました。

さて、なぜこんなことができるのでしょうか？

実は、""で囲まれた範囲は、自動的に String クラスに直されるという決まりがあるのです。したがって、わざわざ new と書かなくても String クラスのインスタンスを作成することができるのです。

ようするに、String クラスのインスタンスは以下の二種類の作り方ができると覚えて置いてください。

```
String str1 = new String(文字列);
String str2 = 文字列;
```

ところで、C 言語など古いプログラミング言語では、日本語を使うには色々と工夫が必要でした。これは、ほとんどのプログラミング言語を作ったのがアメリカ人など英語圏の人たちであったため、英語以外の国の言葉をあまり考慮していなかったためです。

しかしながら、Java は世界中の人に使ってもらうことを目的としているため、最初から日本語が使えるように設計されています。

その証拠を以下に示しましょう。

```
String japanese = “ワタシハ日本人でーす”;
```

ほら、こんな偽外人っぽい言葉でも Java にかかれば簡単に利用することができます。すばらしいですね。

このように、String クラスでは日本語も英語も(多分アラビア語だって)同じように扱うことができますので、英語が苦手なあなたも、アラビア人の彼女がいるあなたも安心です。

10.2.3 文字列の連結

Java の String クラスには、文字列を連結する機能がついています。

通常クラスのインスタンスはメソッドを通して操作するのですが、文字列クラス String に限っては、+演算子や+=演算子を使うことができます。

例としてこんなプログラムを書いてみましょう。

```
String kosaku = "島耕作";

String kachou = "課長"+kosaku;
String buchou = "部長";
buchou += kosaku;
```

```
System.out.println(kachou);
System.out.println(buchou);
```

```
課長島耕作
部長島耕作
```

このとおり島耕作さんの出世街道をあらわすことができちゃいました。

ちなみに、使える演算子は+と+=だけです。-とか*とか/とかは使えません。

```
String hoge = ” hoge” *3;
```

と書いて、

```
“hogehogehoge”
```

になるなんてことはありません。

一般に画面に文字を出力したりする場合に文字列を使いますが、文字列の中に数値を含めたくなることがあります。このような要求も、文字列の連結を利用すれば簡単に実現できます。

```
int num = 3;
String ishi = ″石の上にも″+num+″分″;
System.out.println(ishi);
```

```
石の上にも３分
```

こらえ性のない人が見事表現できました。このように、+で数字をつないでも Java は自動的に数字を文字列の中に代入してくれます。なんと便利。

これを利用すると、数字を簡単に文字列に変換することができます。

```
double pi = 3.14;
int pi2 = 3;

String piString = ″″+pi
String piString2 = ″″+pi2;
```

このようにすることで、”3.14”という文字列と”3”という文字列を手軽に作成することができます。このテクニックは結構便利なので覚えて置いてください。

10.2.4 String 型のさまざまな機能

Java では文字列の色々な操作をすることが可能です．例えば，

- 文字列の長さを調べる
- 文字の置き換え
- 文字列の分割
- 文字列の比較

などが可能です。

```
public static void main(String[] args) {
          String str = new String("部長・島耕作");
          System.out.println(str);

          int length = str.length();
          System.out.println("文字列の長さは"+length);

          str = str.replaceAll("部長", "取締役");
          System.out.println(str);

          String[] twoString = str.split("・");
          for(int i = 0; i < twoString.length; i++){
                    System.out.println(i+":"+twoString[i]);
          }
}
```

これを実行すると、

```
部長・島耕作
文字列の長さは6
取締役・島耕作
0:取締役
1:島耕作
```

となります。

10.2.4.1 文字列の長さ

まず、最初は文字列の長さを取得しましょう。文字列の長さを取得するメソッドは,

```
文字列.length()
```

です.

```
          String str = new String("部長・島耕作");
          int length = str.length();
          System.out.println("文字列の長さは"+length);
```

「部長・島耕作」には文字が 6 つ含まれているので、文字列の長さは 6 になります。

ところで，日本語は通常全角文字と言われ，半角アルファベットの二倍の大きさで表示されます．しかしながら，日本語でも一文字は一文字なのでちゃんと「部長・島耕作」で 6 文字と数えてくれます．もちろん，間にある「・」だって一文字として数えてくれます．

では，こんな場合はどうでしょうか？

```
                    String str = "空白文字入り の場合";
                    System.out.println(str.length());
```

ここでは，間に空白文字が入っています．この場合どうカウントされるかというと・・・

```
10
```

となり，空白文字も一文字として数えられています．

10.2.4.2 文字列の置き換え

次に、ついに島耕作氏が取締役まで出世したことをあらわしてみましょう。

```
str = str.replaceAll("部長", "取締役");
System.out.println(str);
```

この結果は,

```
取締役・島耕作
```

となります.

```
文字列.replaceAll(変更する文字列, 変更後文字列);
```

この replaceAll というメソッドを使うと、ある文字列の中で一つ目の引数に与えられている部分文字列を、二つ目の引数であたえられている文字列に変換した文字列を返します。今回の例ですと、「部長」という肩書きを「取締役」に置き換えてしまっているのです。

このとき，同じ文字が何文字もあれば，すべて置き換えられるます.

```
String bird = "にわにわにわにわとりがいる";
String noBird = bird.replaceAll("にわ", "");
System.out.println(noBird);
```

ここでは,「にわ」を全部消してしまっています．その結果として,

```
とりがいる
```

味気のない文になってしまいました.

ちなみに，ほぼ同じ動きで，最初に一致した文だけを変更するメソッドとして，replaceFirst も用意されています.

```
String aBird = bird.replaceFirst("にわ", "");
System.out.println(aBird);
```

これを実行すると,

```
にわにわにわとりがいる
```

というわけで，庭に鶏がいることが分かりました．二羽いるかどうかは別問題.

というわけで，文字列の置き換えメソッドが二つありますので，両方とも覚えておくと便利です.

```
文字列.replaceAll(変更する文字列, 変更後文字列); //すべて置換
文字列.replaceFirst (変更する文字列, 変更後文字列); //最初に一致したものだけ置換
```

10.2.4.3 文字列の分割

次に紹介するのが，文字列の分割して配列にする方法です。

```
String[] twoString = str.split("・");
```

によって,「取締役・島耕作」が「取締役」と「島耕作」に分割されます.

split メソッドは、引数に与えられた文字列を区切り文字と考えて、もともとの文字列を分割します。この例ですと「・」を区切り文字にしていますので、「取締役」という文字列と「島耕作」という文字列の二つに分割して、順番に格納された配列を返します。

ちなみに、区切り文字として与えられた文字は配列には含まれませんので注意してくだ

さい。例えば,「,」カンマで区切られた文字列を分割して配列にするには以下のようにします.

```
String friends = "マリオ,ルイージ,クッパ";
String[] friendsArray = friends.split(",");
```

また，空白文字で区切られた文字列を分割する場合はちょっと特殊ですが，以下のようにします.

```
String friends = "マリオ ルイージ　クッパ";
String[] friendsArray = friends.split("¥¥s+");
for(int i = 0; i < friendsArray.length; i++){
        System.out.println(friendsArray[i]);
}
```

¥¥s+という特殊な記号を使うと,スペースが自動的に区切り文字になります.このとき，複数の空白が合っても自動的に一つの区切り文字として扱われます．したがって，この実行結果は,

```
マリオ
ルイージ
クッパ
```

となるわけです．めでたい.

文字列を配列に直す方法は以下のとおりです.

```
String[] 配列 = 文字列.split(区切り文字);
```

10.2.4.4 文字列の比較

文字列を扱う場合、その文字列同士が等しいかどうかを判断する必要があることがあります。そんなときは、注意が必要です。以下のようなコードを見てみましょうか。

```
public static void main(String[] args) {
  String java1 = new String("JAVA");
  String java 2 = new String("JAVA");

  if(java 1 == java 2){
    System.out.println("同じだよ");
  }
  else{
    System.out.println("違うよ");
  }
}
```

両方「JAVA」と書いてあるのですから、当然

```
同じだよ
```

と出力されそうなものです。しかしながら、動かしてみると意外なことに

```
違うよ
```

と怒られてしまいます。あまりに意外な結果に怒りたいのはこっちです。

しかし、これには実は深い意味があるのです。それについては、次章で説明しますので、今は==では同じ文字列かどうか判断できないということだけ覚えて置いてください。

同じ文字列かどうか判断する場合は、

```
public static void main(String[] args) {
  String java 1 = new String("JAVA");
  String java 2 = new String("JAVA");

  if(java 1.equals(java 2)){
    System.out.println("同じだよ");
  }
  else{
    System.out.println("違うよ");
  }
}
```

このように equals メソッドを使いましょう。すると、ちゃんと文字列の中身を検査して、

```
同じだよ
```

と出力してくれます。というわけで、文字列が同じかどうかを検査したいときは、==ではなく equals を使うと覚えておきましょう。

ちなみに、この equals という命令、いろいろなところで出てきます。出てきたときは、==とはちょっとだけ違うんだということに注目してみてみましょう。

10.3 プリミティブ型をカバー～ラッパークラス～

10.3.1 ラッパークラスとは

Java では，クラスの変数を参照型，int や double といった基本的な数字の変数をプリミティブ型と呼び，厳密に区別しています.

次章で説明するようにメモリ管理からして別物になっています.

特に大きな問題になるのが，次節以降で説明する Collection フレームワークと呼ばれるクラスを利用する場合です．これらのクラスは配列のような動きをするのですが，プリミティブ型を要素にすることができません．つまり，クラス型の変数しか入れられない便利な配列といった位置づけなのです.

しかし，それらのクラスにも数値を入れたいという要求はどうしてもあります.

そこで，そのような要求に応えるために，**ラッパークラス**と呼ばれる，プリミティブ型

をそのまんまクラスにしたようなクラスが Java には用意されています.

ここでは，それらについて簡単に説明します.

10.3.2 ラッパークラス一覧

ラッパークラスはプリミティブ型をクラスにしたようなものだという話しをしましたが，各プリミティブ型ごとにラッパークラスは用意されています.

まずはその一覧をご紹介します.

プリミティブ型	ラッパークラス
int	Integer
short	Short
long	Long
float	Float
double	Double
boolean	Boolean
byte	Byte
char	Character

int 型のラッパークラスが Integer に，char 型の Character 型になった以外は，プリミティブ型の名前の最初の一文字が大文字になっただけのものです.

10.3.3 ラッパークラスの作成

ラッパークラスは，基本的にコンストラクタに元となるプリミティブ型を指定して作成します.

```
Integer i = new Integer(10);
Boolean b = new Boolean(true);

double a = 1.5;
Double d = new Double(a);
```

また，こうしなくても，プリミティブな整数型を代入するとそれに見合ったラッパクラスを自動的に作成してくれます.

```
Integer wrapperInt = 50;
```

10.3.4 ラッパークラスの特徴

ラッパークラスは基本的には元となるプリミティブ型のをあらわすクラスという事になります．ラッパークラスに対しては，ほとんど全てのプリミティブ型で利用可能な操作を行うことが出来ます.

```
Integer i = new Integer(10);
Boolean b = new Boolean(true);

double a = 1.5;
Double d = new Double(a);

if(b){
        i++;
        System.out.println(i+5);
```

```
                }
                if(i == 11){
                        System.out.println("同じだよ");
                }
                else{
                        System.out.println("ちがうよ");
                }
```

このようなプログラムであれば,

```
16
同じだよ
```

と出力されます.

このように,ラッパークラスはほとんどの場合でプリミティブ型と一見同じように動きます.

10.3.5 ラッパクラスとプリミティブクラスの交換

ラッパクラスとプリミティブクラスの変換は自動的に行われます. したがって,

```
Integer wrapperInt = new Integer(10);
int primitiveInt = wrapperInt;
wrapperInt = 51;
System.out.println(primitiveInt);
System.out.println(wrapperInt);
```

は期待通りの動きをします.

```
10
51
```

このように，自動的にラッパクラスとプリミティブ型とを相互変換してくれる機能を Autoboxing と呼びます。まあ、名前は覚える必要ないのですが。

ちなみに，この機能は Java5 から登場したものです.

Java1.4 までの時代はこのような交換はできませんでした.

万が一 Java1.4 以前のものを利用する場合は，以下のようにしてラッパクラスとプリミティブ型を交換します.

```
Integer wrapperInt = Integer.valueOf(50);
int primitiveInt = wrapperInt.intValue();
```

Integer にある static な valueOf メソッドを利用すると，int 型を Integer 型に変換してくれます．また，intValue メソッドで，Integer 型変数が保持している整数を int 型に直して返します.

同様に，Double 型には doubleValue メソッドがあります.

ラッパークラス	プリミティブ⇒ラッパクラス	ラッパクラス⇒プリミティブ
Integer	Integer.valueOf(int)	*Integer*.intValue();
Short	Short.valueOf (short)	*Short*.shortValue();
Long	Long.valueOf (long)	*Long*.longValue();
Float	Float.valueOf t(flaot)	*Float*.floatValue();
Double	Double.valueOf (double)	*Double*.doubleValue();
Boolean	Boolean.valueOf (boolean)	*Boolean*.booleanValue();

Byte	Byte.valueOf (byte)	*Byte*.byteValue();
Character	Character.valueOf (char)	*Character*.charValue();

10.3.6 文字列からの変換

ラッパークラスは，ただ単にプリミティブな数値と同じものではなく，もう少し高機能です．その機能で最もよく使うのが，この文字列からの変換でしょう．

ラッパークラスを使うことで，数字が書かれた文字列を数字型の変数に変換することが出来るのです．

```
String price = "1160";
int priceInt = Integer.parseInt(price);
double taxIncluded = priceInt*1.05;
System.out.println(taxIncluded);
```

例えば，このプログラムでは，ある商品の値段が，文字列として「1160」と与えられています．これを数値に直して priceInt という変数に代入します．さらに，その後 priceInt を 1.05 倍して税込価格を計算しました．

この結果は，

```
1218.0
```

無事 1218.0 という数字になり，文字列で与えられた値段から税込価格を計算することに成功しています．

このとき，ポイントは文字列を整数型に直すメソッド，

```
int 整数 = Integer.parseInt(文字列)
```

です．

今回は整数の場合を紹介しましたが，これ以外にも，Double，Boolean など全てのラッパクラスに文字列をプリミティブ型に直すメソッドが用意されています．

ラッパークラス	変換メソッド
Integer	Integer.parseInt(*文字列*)
Short	Short.parseShort(*文字列*)
Long	Long.parseLong(*文字列*)
Float	Float.parseFloat(*文字列*)
Double	Double.parseDouble(*文字列*)
Boolean	Boolean.parseBoolean(*文字列*)
Byte	Byte.parseByte(*文字列*)
Character	Character.parseChar(*文字列*)

ちなみに，返値はすべてプリミティブ型となります．

ところで，このとき引数に数字以外を入れたらどうなるでしょうか？

```
int value = Integer.parseInt("5000 円");
```

```
                System.out.println(value);
```

これを実行してみると・・・

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "5
000 円"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at StringToNumberMain.main(StringToNumberMain.java:14)
```

というわけで，エラーが発生してプログラムが終了してしまいました.

例え，数字が含まれていてもそのまま数字として扱えないものだと，利用できませんのでご注意下さい.

10.4　順番に並べる ArrayList

Java の配列は、長さを取得できたり、廃棄しなくて良かったりと、高度な機能を持っていました。しかし、世の中上には上がいるもので、配列よりもさらに高度な機能を持った配列のようなクラスが存在します。それが、ArrayList です。

基本的に ArrayList は配列とほぼ同じ機能を持っています。配列のように、値を入れたり、値を出したり、長さを調べたりすることが出来ます。

では、とりあえず ArrayList が配列に比べてどれだけ便利なのかを如実に示す例をご紹介しましょう。今回説明に使うのも、例のごとく、アドレス帳で PersonalData をまとめたものです。これまでは配列で行っていましたが，今回は ArrayList を使って作成します.

通常ならば配列を用意すれば十分と思いがちです。しかし、配列の場合あらかじめいくつ個人データを保存するかを決めておかなければいけません。したがって、万が一予想よりも友達が多くなってしまった場合，アドレス帳に全員分のデータを保存しておくことが出来ません。せっかく合コンを開いてもあの子の携帯番号を記録できないなんて！

そんな悲劇に見舞われたくない方にお勧めするのが、今回紹介する ArrayList です。

10.4.1 基本的な使い方

ArrayList の最も基本的な使い方は以下のとおりです.

```
import java.util.ArrayList;

import cellphone.address.PersonalData;

public class ArrayListMain {

        public static void main(String[] args) {
                ArrayList<PersonalData> addressList = new ArrayList<PersonalData>();

                PersonalData mario = new PersonalData("マリオ");
                PersonalData luigi = new PersonalData("ルイージ");
                addressList.add(mario);
                addressList.add(luigi);
                for(int i = 0; i < addressList.size(); i++){
                        PersonalData pd = addressList.get(i);
                        System.out.println(pd.getName());
```

```
                }
        }

}
```

では，このプログラムを詳しく見て行きましょうか.

まず，ArrayList を利用するためには java.util.ArrayList を import する必要があります.

```
import java.util.ArrayList;
```

これは, ArrayList が java.util というパッケージに入っているからなんですね. もちろん,

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

としてもかまいませんが，いずれにせよ必ず ArrayList を import するようにしましょう.

次に、ArrayList 型の addressList というインスタンスを作成します。

```
 ArrayList<PersonalData> addressList = new ArrayList<PersonalData>();
```

通常、変数を宣言する場合は、

```
int a = 0;
```

のように書いていましたが，ここでは<PersonalData>というヘンなものがついています。これは「この ArrayList の中身は PersonalData だよ」ということを宣言しているものです。

配列の場合は最初に「何の配列か」を指定して作成しましたよね。

```
PersonalData[] personalDataArray;
```

それと同じようなものだと思ってもらえれば結構です。

このような宣言をすることで、addressList の中に PersonalData 型の変数を入れることが出来なるようになります. 逆にいうと addressList の中に PersonalData 型以外を入れることは出来ません。

このように ArrayList の中身を<>で指定することを **Generics** と呼びます. 実は，中身が何であるかを指定しないで ArrayList を作成する方法というのもあります. それについては後ほど説明したいと思います.

さて addressList インスタンスが出来たら、addressList の中へ個人データを挿入します。

```
                    PersonalData mario = new PersonalData("マリオ");
                    PersonalData luigi = new PersonalData("ルイージ");
                    addressList.add(mario);
                    addressList.add(luigi);
```

この add という命令、「ArrayList の最後に新しい値を入れろ」という命令なのです。そのため、最初は長さ 0 の addressList ですが、その一番下、つまり 0 番目に mario の値が追加されます。また，その直後に今度は luigi が与えられます.

ここでポイントとなるのが，ArrayList は配列のようなものですが，大きさの指定を全くしていません. また，どこにデータを入れるのかも指定していません. ただ，最後に入れるよ，ということだけを指定しています.

さて、こうしてまんまと addressList の中にはマリオとルイージの個人データが保存されることになりました．ま，今回は名前しかデータはありませんが・・・

では，せっかく挿入したデータを確認してみることにしましょう。配列の場合と同様に、for 文を使って，ArrayList の要素を全部取得して名前を表示していきましょう．

```
for(int i = 0; i < addressList.size(); i++){
        PersonalData pd = addressList.get(i);
        System.out.println(pd.getName());
}
```

配列では、length によって長さを調べられましたが、ArrayList の場合は size()を利用します。これで、addressList の長さ分だけ i の値が変化します。

```
PersonalData pd = addressList.get(i);
```

そして、get(*i*)メソッドを利用すること出，*i* 番目の要素を取り出せます。これによって、無事 PersonalData 型のデータを取得できましたので、名前を表示します。

```
マリオ
ルイージ
```

これで、マリオとルイージのデータを保存に成功したことが分かりました．

何度も書きますが，配列との最大の違いは最初にいくつのデータを入力するかを決めておかなくても良いという点です．add メソッドによって PersonalData を挿入すれば好きなだけデータを追加していくことが出来るのです．

さて、こんな感じで ArrayList を使うと、データがいくつ入るのか分からないときでも、配列を利用することができるわけです。

10.4.2 配列との比較

ArrayList は配列と似ています．基本的に，ArrayList では配列で行うことが出来る操作はほとんど全部行うことが可能です．

配列の操作とそれに対応する ArrayList のメソッドを以下にまとめます．ここで，配列の変数を array, ArrayList の変数を list として比較していきましょう．

配列	操作	ArrayList
String[] array	変数宣言	ArrayList<String> list
new String[10]	作成	new ArrayList<String>()
array.length	長さの取得	list.size()

`val = array[0]`	**要素の取得**	`val = list.get(0)`
`array[0] = val`	**要素の設定**	`list.set(0, val)`
出来ない	**要素の挿入**	`list.add(val)`

なお，要素の挿入は配列では行うことが出来ません．

10.4.3 数値の保存

ArrayList では，中に入れる要素の種類を<>で囲んで指定します．

しかしながら，ここでプリミティブ型は宣言する事ができません．

```
ArrayList<int> intList = new ArrayList<int>();
```

このようなことができないのです．

そこで，このような場合に利用するのが，前節でやったラッパークラスです．整数を入れる ArrayList を作成する場合は，以下のようにします．

```
ArrayList<Integer> intList = new ArrayList<Integer>();
```

これによって整数をずんずん入れられる ArrayList が完成します．

```
intList.add(1);
intList.add(new Integer(2));
intList.add(2+2);
intList.add(3, 2*2*2);
```

このとき，本来 ArrayList に代入できるのは Integer 型ですが，ラッパクラスの自動変換機能を使って，普通の整数を入れた場合も自動的に Integer 型に変換して ArrayList に挿入してくれます．

また，同様に取得も以下のように行うことが出来ます．

```
int val1 = intList.get(0);
Integer val2 = intList.get(1);
```

取り出す場合も挿入する場合と同様，int⇔Integer は自動的に変換してくれます．

ただし，このとき注意しなければいけないのが，以下のような操作が出来ないという点です．

```
intList.get(1)++;
intList.get(2)+=5;
intList.get(3)=16;
```

これら 3 つの操作は，いずれも代入を含む操作です．このような代入を含む操作を行う場合は，set メソッドと併用して，以下のように行わなければいけません．

```
intList.set(1, intList.get(1)+1);
intList.set(2, intList.get(2)+5);
intList.set(3, 16);
```

一度**取り出した要素に対して代入操作を行う場合は，set で改めて挿入しなおさなければいけない**，ということを覚えておいてください．

10.4.4 Generics を使わない ArrayList

さて，今回紹介した ArrayList は中に入れる型を<>で指定していました．このように<>

で中に入れる型を指定することを Generics といいます. 実は<>を省略することが出来ます.

今回 addressList は Generic を使って PersonalData 型を入れるものとして宣言しましたが、Generics を使わないことで、どんなものでもいれられる ArrayList にすることも可能です。

```
ArrayList wholelist = new ArrayList ();
```

このように宣言することで、wholeList には参照型ならなんでも入れることが出来るようになります。例えば、String の次に Double を入れ、その後に Date を入れるといった荒業も可能です。

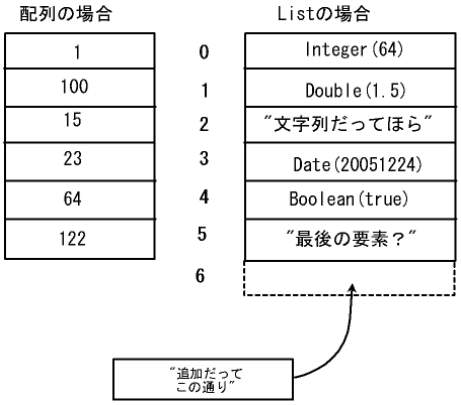


図 1　なんでも入る wholeList に 4 次元ポケットもビックリ

しかし、この場合 wholeList から get を使ってデータを取り出したとき、そのデータ型が何か分からなくなってしまうという欠点があります。そのため、プログラマがそのデータの型をはっきりと教えてあげなければいけません。その場合、

```
Double value = (Double) wholeList.get(i);
```

のように中身の型を知らせてあげる必要があります。(Doulbe)です。このような操作を「明示的なキャスト」といいます。このとき、もし wholeList の i 番目の要素が Double でないと ClassCastException が発生しますので、気をつけましょう。

ちなみに、<>によって中身を指定していない場合に、(Double)を付け忘れるとどうなるでしょうか.

```
Double value = wholeList.get(i);        //(Double)つけ忘れ！
```

この場合,

```
NoGenericsMain.java:13: 互換性のない型
検出値　:java.lang.Object
期待値　:Double
                Double d = wholeList.get(0);
                                         ^
エラー 1 個
```

このように，エラーが発生して怒られてしまいます。気を付けましょう。

10.4.5 ArrayList の添え字

ArrayList でも配列と同様に get できるのは、「0～要素の数-1」となります．つまり、3 つのデータ（要素）を ArrayList に追加場合、0～2 までの要素番号（インデックス）を持つ ArrayList が作成されます。

要素番号	中身
0	111
1	108
2	112

ここでもし、3 番目以降のデータを取ろうとすると、配列のときと同様 ArrayIndexOutOfBounds というエラーが発生してプログラムが止まってしまいます。もちろん、-1 より前のインデックスを指定しても同じエラーが発生します。

ところで、要素は追加するばかりではなく途中で変更したい場合もあります。そんな場合は、set を使います。これは配列にも普通にある機能です。ので、あんまりビックリするようなことはありませんが、一応説明しておきます。

```
valueList.set(0, 5); // a[0] = 5;と同じ
```

このように、set メソッドを使うことで、a[0]=5 と同じ動作が可能になります。ただし、このとき添え字に入れられる数は、すでにある要素数と同じだけという制限があります。あ、ちなみに要素数＝配列数と考えて下さい。

したがって、要素数が 5 しかない配列に対して、

```
valueList.set(10, 5);
```

という動作は出来ないのです。この場合もやっぱり ArrayIndexOutOfBoundsException が発生します。

さて、この要素数、一番最初は 0 です。したがって、最初は set することはできず、add することしかできません。これは不便ですよね。というわけで、最初から要素数を決めておくことも出来ます。

```
ArrayList<Double> valueList = new ArrayList(10);
```

こうすることで、最初から要素が 10 個ある ArrayList が完成です。ただし、一番最初は各要素の中身は空(null)になります。

なお、ArrayList に格納できる要素数はすでに紹介した通り、

```
int size = valueList.size(); //a.length と同じ
```

のように、size()を使います。

10.4.6 ArrayList の更なる機能

さて、ArrayList には配列ではできなかった便利な機能が満載です。

- ArrayList の途中に要素を挿入
- 特定のデータが ArrayList 中に在るか検索
- 要素を削除
- 全要素を削除

10.4.6.1 途中に要素を挿入

ArrayList では，最後に要素を追加するために add メソッドがありましたが，実は引数を変えた add メソッドを使うことで，List の途中にもデータを挿入することが出来ます．

```
ArrayList<PersonalData> addressList = new ArrayList<PersonalData>();

PersonalData mario = new PersonalData("マリオ");
PersonalData luigi = new PersonalData("ルイージ");
addressList.add(mario);
addressList.add(luigi);

PersonalData kuppa = new PersonalData("クッパ");
addressList.add(0, kuppa);
for(int i = 0; i < addressList.size(); i++){
        PersonalData pd = addressList.get(i);
        System.out.println(pd.getName());
}
```

このように，add メソッドに挿入する場所の添え字を引数として渡すと，

```
クッパ //←添え字の 0 番目にクッパのデータが入りました
マリオ
ルイージ
```

このように，任意の位置にデータを挿入することが出来ます．

ただし，このときデータが存在しない場所には挿入することが出来ません．もし無理矢理挿入しようとするとどうなるでしょうか？

```
PersonalData peach = new PersonalData("ピーチ");
addressList.add(5, peach);
```

ここでは，3 つしかデータがない，つまり添え字は 2 までしかない状態で添え字 5 の位置に無理矢理ピーチ姫のデータを入れようとしています．そうすると・・

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 5, Size: 3
        at java.util.ArrayList.add(Unknown Source)
        at InsertMain.main(InsertMain.java:28)
```

というわけで，エラーが発生してしまいました．

このように，通常の配列のように ArrayList といえども，最大添え字以上の添え字を指定しての操作はできませんのでご注意を．

10.4.6.2 要素を削除

配列の場合，要素の削除を実現するのは結構大変です．addressArray という配列から添え字 5 の要素を削除するには，

```
PersonalData[] addressArray = new PersonalData[10];
PersonalData[] tmpArray = new PersonalData[addressArray.length-1];

int idx = 5;
System.arraycopy(addressArray, 0, tmpArray, 0, idx);
System.arraycopy(addressArray, idx+1, tmpArray, idx, addressArray.length-idx-1);
addressArray = tmpArray;
```

という作業が必要になります．非常に面倒くさいですね．

これが，ArrayList だと非常に簡単に終ります．

```
ArrayList<PersonalData> addressList = new ArrayList<PersonalData>();

PersonalData mario = new PersonalData("マリオ");
PersonalData luigi = new PersonalData("ルイージ");
addressList.add(mario);
addressList.add(luigi);

addressList.remove(0);
for(int i = 0; i < addressList.size(); i++){
          PersonalData pd = addressList.get(i);
          System.out.println(pd.getName());
}
```

この中で，実際に削除を行っているのは，

```
addressList.remove(0);
```

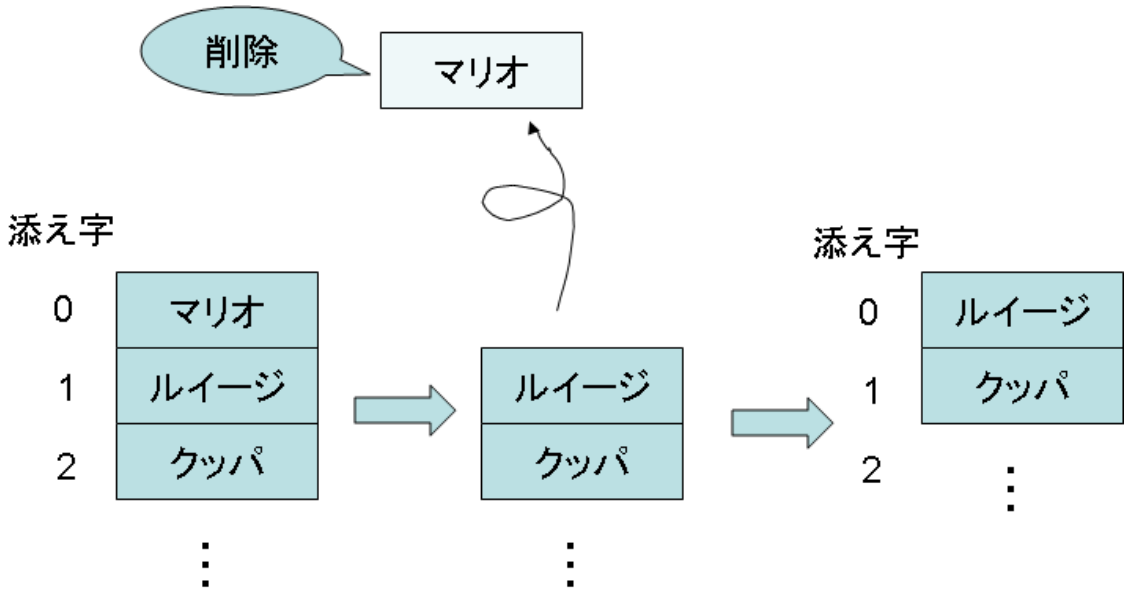
の部分だけです．

これを実行すると，

```
ルイージ
```

となり，添え字 0 にあったマリオのアドレスは削除されています．

ちなみに，このとき添え字 0 以降のデータは自動的に前に詰められます．



このように，ArrayList を使った場合，remove メソッドを使うことで，簡単に要素を削除できます．

```
List 名.remove(添え字);
```

ちなみに,ArrayList には一つの要素を削除するメソッドだけではなく,全要素を一度に削除するメソッドも用意されています．これも使い方は非常に簡単です．

```
addressList.clear();
System.out.println("要素数は"+addressList.size());
```

このように clear メソッドを使うと，全要素が自動的に全て削除されます．実行すると，

```
要素数は 0
```

というわけで，全データが削除されました．

10.4.7 配列と ArrayList の交換

ArrayList と配列が非常に似ていることは、分かっていただけたと思います。ということは、配列にも直せるんじゃないか思った貴方は大正解。ArrayList から配列に直すメソッドも用意されています。ただ、少し使い方に特徴があるので注意しましょう。

```
ArrayList<PersonalData> addressList = new ArrayList<PersonalData> ();

addressList.add(new PersonalData("マリオ"));
addressList.add(new PersonalData("ルイージ"));

PersonalData[] addressArray = addressList.toArray(new PersonalData[0]);

for(int i = 0; i < addressArray.length; i++){
    System.out.println(addressArray[i].getName());
```

```
                }
```

このようにすると ArrayList から配列へ変換できます。

```
マリオ
ルイージ
```

ここでのポイントは,

```
PersonalData[] addressArray = addressList.toArray(new PersonalData[0]);
```

この部分こそが、List から配列に直す部分ということは分かると思います。使うのは、toArray メソッドです。この toArray メソッドは、List の中身をすべて配列にして返してくれます。

また、最後に引数が new PersonalData[0]となっていますが、これはおまじないだと思ってください。これを書かないと、うまくいかないことがあります。

以上、まとめると、List から配列に直す場合は、

```
配列のクラス名[] インスタンス名 = List のインスタンス名(new 配列のクラス名[0]);
```

と書けばよいということになります。

また，逆に配列から ArrayList にする場合は以下のようにします.

```
ArrayList<PersonalData> addressList2 = new ArrayList<PersonalData>(Arrays.asList(addressArray));
```

ただし，このとき java.util.Arrays を import しておく必要があります.

あまり難しいことは説明してもしょうがないので，こうすることで配列→ArrayList が変換可能だと覚えておいてください.

```
ArrayList<要素の型> List 名 = new ArrayList<要素の型>(Arrays.asList(配列名));
```

10.5　一つずつ集める集合クラス～HashSet～

10.5.1 HashSet とは何か？

配列よりも便利なクラスとして，ArrayList をご紹介しましたが，ArrayList とはちょっと異なる「集めるクラス」が Java には用意されています.

その一つが，本章で紹介する HashSet です.

HashSet は，ArrayList とは異なり，入れたデータが入れた順番には並びません．ただ，雑多に HashSet の中に入れられていると思ってください.

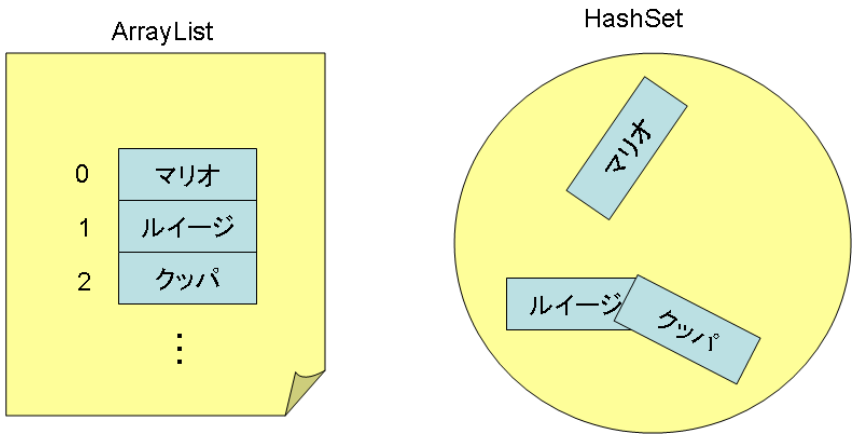


図 2 ArrayList と Hash

しかし，ただ雑多に入れられただけというクラスではあまり価値がありませんよね．もっと決定的な違いが HashSet と ArrayList にはあります．それが，**唯一性**です．

ArrayList には同じデータをいくつも入れることが出来ましたが，HashSet には**同じデータは一つしか入れられない**という決まりがあります．逆に言えば，HashSet の中には重複データがないことが保障されるわけです．

どんな場合に役立つかはこれからみていくとして，その違いを把握しておいてください．

コラム

さて，ArrayList と HashSet の違いを理解するために，こんな例を出して見ましょうか．

皆さんは，１万円札を５枚と 1000 円札を 10 枚持っていたとします．このとき,手持ちのお金をそのまま持っているのは不安なので， ArrayList か HashSet かのどちらかで作られた財布に入れることにしました．

さあ，どちらの財布に入れたほうがよいでしょうか？ちょっと考えてみてくださいね・・・

では，ArrayList なさ畏怖に入れた場合と HashSet な財布に入れた場合，それぞれどうなるのか，考えて見ましょう．

まず,ArrayList の中に入れれば，後で取り出すときも１万円札が５枚，その後 1000 円札を 10 枚，合計 6 万円取り出すことができます．

しかし，もし HashSet な財布に入れておいたら・・・5 枚の 1 万円札は HashSet の重複を許さないというルールから，1 枚の 1 万円札になってしまいます．そして，10 枚の 1000 円札も 1 枚の 1000 円札になってしまいます．したがって，11000 円に目減りしてしまいます．ショック

こんなショックな状態に陥らないですむように,ArrayList と HashSet の違いをきちんと理解しましょう．

10.5.2 HashSet の基本

HashSet も ArrayList のように,指定されたデータをいくつか集めるために利用します.

```
import java.util.HashSet;

public class HashSetMain {

	public static void main(String[] args) {
		HashSet<PersonalData> addressSet = new HashSet<PersonalData>();

		PersonalData mario = new PersonalData("マリオ");
		PersonalData luigi = new PersonalData("ルイージ");
		addressSet.add(mario);
		addressSet.add(luigi);
		for(PersonalData personalData:addressSet){
			System.out.println(personalData.getName());
		}
	}
}
```

これは，ArrayList の最初の例と同じようにマリオ，ルイージの住所を addressSet という名前のアドレス帳に保存しています.

データの取得方法がちょっと ArrayList とは違いますが，それ以外はほとんど一緒名点に注目してください.

さて，この実行結果はというと・・・

```
マリオ
ルイージ
```

というわけで，ArrayList の場合とほとんど変わりません.

さて，詳しく中身を見て行きましょう.

まず、HashSet は java.util パッケージにありますので、インポートしないと使うことができません。

```
import java.util.HashSet;
```

もちろん、面倒ならば以下のようにすることで他の java.util パッケージにあるすべてのクラスを使うことができるようになります。

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

ArrayList や HashMap も同じパッケージにありますので、こちらのほうが何かと便利かもしれませんね。

```
		HashSet<PersonalData> addressSet = new HashSet<PersonalData>();

		PersonalData mario = new PersonalData("マリオ");
		PersonalData luigi = new PersonalData("ルイージ");
		addressSet.add(mario);
		addressSet.add(luigi);
```

この部分は，ArrayList が HashSet に変わっただけで，それ以外には何も変化がありません．ArrayList を作成したときと同じようにデータを順次追加していくことになります.

一方，HashSet に溜め込んだデータを取り出すには，ちょっと特殊な方法が必要です.

ArrayList ならば添え字を指定すれば添え字に対応したデータが返されますが，HashSet の場合添え字が存在しないので，特定の要素を取り出すことが出来ません.

その代わりに用意されているのが，**iterator** という機能です.

iterator とは**反復子**と呼ばれるもので，これを利用することで，全ての要素に順番に同じことを繰り返し行うことができるようになります.

その iterator を使う方法の一つがこれです.

```
for(PersonalData personalData:addressSet){
        System.out.println(personalData.getName());
}
```

for を使っていますが，一般の for 文の書き方とちょっと違いますよね.

HashSet で iterator を使うための for の構文は以下のとおりです.

```
for(変数型 変数名:HashSet 名){
        //変数名に対する処理
}
```

これによって，HashSet 内にあるすべての要素に対して順番に処理を行うことが可能になります．先の例ですと，HashSet 内のすべての PersonalData の名前を取得して表示しているわけです.

ちなみに、この書き方は Java5 以降でしか使えません。Java1.4 以前の Java を使う場合は、以下のような書き方をします。

```
for(Iterator<PersonalData> it = addressSet.iterator(); it.hasNext(); ){
        PersonalData personalData = it.next();

        System.out.println(personalData.getName());
}
```

ちょっと分かりづらいかと思いますが、こうするものだと思って覚えてください。

その結果として，マリオとルイージの名前が表示されたわけですね.

ということは,

```
PersonalData mario = new PersonalData("マリオ");
PersonalData luigi = new PersonalData("ルイージ");
addressSet.add(mario);
addressSet.add(luigi);

PersonalData kuppa = new PersonalData("クッパ");
addressSet.add(kuppa);

for(PersonalData personalData:addressSet){
        System.out.println(personalData.getName());
}
```

としてクッパのデータを追加してあげれば,

```
マリオ
ルイージ
クッパ
```

となり，ちゃんと名前が表示されるようになります.

ちなみに，実は，ArrayList でも HashSet と同じように for ループによってすべてのデ

ータをあますことなく取得することが出来ます.

```
ArrayList<PersonalData> addressList = new ArrayList<PersonalData>();

//データ挿入

for(PersonalData pd:addressList){
        System.out.println(pd.getName());
}
```

ちょっとしたテクニックですが覚えておいて損はない機能です.

10.5.3 同じデータが重複しない

さて，HashSet の使い方を示しましたが，このままではただの任意のデータを取得できない ArrayList みたいです.

そこで，HashSet の特徴を利用したプログラムを一つご紹介しましょう.

```
HashSet<PersonalData> addressSet = new HashSet<PersonalData>();

PersonalData mario = new PersonalData("マリオ");
PersonalData luigi = new PersonalData("ルイージ");
addressSet.add(mario);
addressSet.add(luigi);

addressSet.add(mario); //←間違えて二回登録
for(PersonalData personalData:addressSet){
        System.out.println(personalData.getName());
}
```

ここでは，間違えてマリオの個人データを二回登録してしまっています．まあ，二回登録したからといって困ったことがおきるわけではありませんが，できれば同じ個人データは一つしか登録されていない方が望ましいですよね.

さて，困ってしまいましたが，一応アドレス帳がどうなっているか見てみましょう.

```
マリオ
ルイージ
```

あれれ，二回登録したにもかかわらず，マリオの個人データは一つしか記録されています.

そう，すでに説明したとおり，HashSet では同じデータは一つしか保有しないという決まりがありますから，同じ人のデータを二回登録しようとすると登録を却下してくれるのです．これは便利.

10.5.4 HashSet のいろいろな機能

HashSet には様々な機能が用意されています．そのうち良く使うものをいくつかご紹介しましょう.

10.5.4.1 存在するかどうかの確認

HashSet では，ある要素が HashSet の中にあるかどうかを簡単に調べる方法が用意されています．それが，contains です．

次のような例を見てみましょう．

```
HashSet<PersonalData> marioAddressSet = new HashSet<PersonalData>();
PersonalData deiji = new PersonalData("デイジー");

marioAddressSet.add(deiji);

//浮気調査開始

if(marioAddressSet.contains(deiji)){
        System.out.println("浮気発見！");
}
```

ピーチ姫がマリオのアドレス帳を盗み見て，デイジー姫の個人情報がアドレス帳に存在するかどうかを確認しています．その結果は・・・

```
浮気発見！
```

おっと，あっさりとデイジー姫の個人情報があることを発見されてしまいました．

このように，HashSet を利用すると，あるインスタンスが HashSet に存在するかどうかを簡単に把握することが出来ます．

HashSet 内にあるかどうかの確認を行うメソッドは以下のように使います．

```
HashSet.contains(変数);
```

存在すれば true が，存在しなければ false が返ってきます．

10.5.4.2 要素の削除

さて，次は要素の削除です．

マリオの浮気を発見したピーチ姫，ここで当然のようにデイジー姫の個人情報を消しにかかりました．さあ，どうやって消すのかというと・・・

```
if(marioAddressSet.contains(deiji)){
        System.out.println("浮気発見！");
        marioAddressSet.remove(deiji);
        System.out.println("削除！");
}

for(PersonalData personalData:marioAddressSet){
        System.out.println(personalData.getName());
}
```

これで，簡単に削除できます．結果を見てみると・・・

```
住所録一覧
クッパ
ルイージ
デイジー
浮気発見！
削除！
クッパ
```

```
ルイージ
```

まんまとデイジーの個人データが消されてしまいました．とほほ．

というわけで，特定のデータを HashSet から削除する構文は以下のとおりです．

```
HashSet.remove(変数);
```

なお，remove メソッドの返値は boolean で，削除する対象があれば true，なければ false が返ってきます．

10.5.4.3 まとめて追加

さて，アドレス帳などを管理するときに一度に何人ものアドレスを追加したい場合もありますよね．HashSet ではそれが可能となっています．

ここでは，クッパにさらわれたピーチが，クッパのもっているアドレス帳を自分のところにもコピーしようとしたと想定しましょうか．

```
PersonalData mario = new PersonalData("マリオ");
PersonalData luigi = new PersonalData("ルイージ");
PersonalData kino = new PersonalData("キノピオ");
PersonalData noko = new PersonalData("ノコノコ");


HashSet<PersonalData> peachSet = new HashSet<PersonalData>();
peachSet.add(mario);
peachSet.add(luigi);

HashSet<PersonalData> kuppaSet = new HashSet<PersonalData>();
kuppaSet.add(kino);
kuppaSet.add(noko);
kuppaSet.add(mario); //重複

peachSet.addAll(kuppaSet);
for(PersonalData personalData:peachSet){
        System.out.println(personalData.getName());
}
```

この addAll メソッドを使うと，ある HashSet の中身をすべて別の HashSet に移すことが可能です．

この結果はというと，

```
マリオ
ルイージ
キノピオ
ノコノコ
```

となり，ピーチ姫のアドレス帳には全員分のデータが移されました．ちなみに，重複していたマリオの個人データは当然一つしか登録されません．

10.5.5 順番がぐちゃぐちゃ

さて，HashSet ではデータに特に順番付けがされません．そのため，必ずしも入れた順

番にデータが出てくるとは限らないことに注意してください.

例えば，こんなプログラムを考えてみましょう．これは，トランプのカードを順番に cardSet に入れていったものだと想像してください.

```
HashSet<String> cardSet = new HashSet<String>();

cardSet.add("1");
cardSet.add("2");
cardSet.add("3");
cardSet.add("4");
cardSet.add("5");
cardSet.add("6");
cardSet.add("7");
cardSet.add("8");
cardSet.add("9");
cardSet.add("10");
cardSet.add("J");
cardSet.add("Q");
cardSet.add("K");

for(String card:cardSet){
        System.out.println(card);
}
```

これを実行すると・・・

```
3
K
7
2
Q
1
6
10
5
J
9
4
8
```

おっと，勝手にシャッフルされたものが出てきました.

このように，HashSet では for 文で取り出そうとしたときに，**データが入れた順番に出てくるとは限らない**ということに注意してください.

しかも，毎回同じ順番で出るとも限りませんし，**逆にシャッフルされていないとも限らない**のです． HashSet に入れるデータは順番が関係ないものに限るように気をつけましょう．順番が気になるデータは ArrayList に入れた方が無難です.

しかしながら，順番が気になるけど重複しては困る，そんな場合もありますよね．そんなときは，ちょっとしたテクニックを使うと，重複しないけど入れた順番に出てくるようにすることが出来ます．それは以下のようにする方法です.

```
HashSet<String> cardSet = new LinkedHashSet<String>();

cardSet.add("1");
cardSet.add("2");
cardSet.add("3");
cardSet.add("4");
```

```
cardSet.add("5");
cardSet.add("6");
cardSet.add("7");
cardSet.add("8");
cardSet.add("9");
cardSet.add("10");
cardSet.add("J");
cardSet.add("Q");
cardSet.add("K");

for(String card:cardSet){
        System.out.println(card);
}
```

new で作るインスタンスを LinkedHashSet というクラスにしています. こうするだけで,

```
1
2
3
4
5
6
7
8
9
10
J
Q
K
```

ほら順番どおり出てきました.

今回はあまり詳しい話は割愛しますが, LinkedHashSet は HashSet を継承した派生クラスで，入れた順番にデータを出すことを保証しているクラスです.

どうしても順番どおりに重複無しにデータを使いたい場合，このクラスを使ってみるのも一つの手かもしれません.

10.6 関連付ける～HashMap～

10.6.1 HashMap とは何か

いろいろな値を保存しておきたいときは、配列か ArrayList，または HashSet を使うということはすでに学びました。しかし、配列とリストは 0 番目は何、1 番目は何、…といった具合に数字と値とを対応付けることしか出来ません。HashSet にいたっては，どんな値とも対応付けることが出来ません.

しかし、同じように値を保存する場合でも、数字以外のものと対応付けたい場合があります。例えば、ある人の個人データを取得するときに，名前を元に検索できたら便利ですよね．そう、できればこんな感じにデータを保存したいわけです。

```
PersonalData mario = new PersonalData("マリオ");
PersonalData luigi = new PersonalData("ルイージ");
```

```
                PersonalData[] personalDataArray = new PersonalData[10];
                ArrayList<PersonalData> personalDataList = new ArrayList<PersonalData>();

                personalDataArray["マリオ"]=mario;
                personalDataList.add(["ルイージ"], luigi));
```

しかしながら，配列も ArrayList もこんなことはできません。こんなプログラムを書いたらすぐにエラーが起きて怒られてしまいます。融通の利かないやつらです。

ところが、Java の開発者は我々の要望がよく分かっています。こういう便利な機能を持ったクラスが Java には用意されています。それは **HashMap** です。配列が

```
PersonalData mario = personalDataArray[0];
```

として、0 番目の値を取得できたように、HashMap を使うと，

```
PersonalData mario = hashMap.get("マリオ")
```

という書き方ができるのです．なんと HashMap では、文字列、数字を問わず、あらゆるクラスインスタンスに対応付けて、値を持つことが出来るのです。

10.6.2 HashMap の基本

では、早速 HashMap の使い方を学びましょう。今回は、各個人の個人データを名前と関連付けて保存するプログラムをご紹介します。

なお，HashMap では ArrayList でいえば添え字に当たるものを**キー(key)**と呼び，要素にあたるものを**バリュー(value)**または**データ**と呼びます．

ここで紹介するプログラムは、最初に文字列クラスをキー，と個人データクラスをバリューとして関連付けて HashMap クラスを作成します．次に，各個人データを HashMap に保存します。その後、名前を元にデータを取得します．

```
import java.util.HashMap;
public class HashMapMain {
        public static void main(String[] args) {
                HashMap<String, PersonalData> addressMap = new HashMap<String, PersonalData>();
                PersonalData mario = new PersonalData("マリオ");
                PersonalData luigi = new PersonalData("ルイージ");
                addressMap.put("マリオ", mario);
                addressMap.put("ルイージ", luigi);

                // マリオの住所を取り出す
                PersonalData data = addressMap.get("マリオ");
                System.out.println("マリオの住所データ:" + data.getName());
        }
}
```

これを実行すると，

```
マリオの住所データ:マリオ
```

となり，ちゃんとマリオの個人データが取得できていることがわかります.

HashMapの動きは基本的にArrayListとほぼ同じだと思って間違いありません.ただし，添え字が数字ではなくてなんでも良いというのが違いなだけです.

まず、HashMap は、java.util パッケージにありますので、java.util.HashMap をインポートします。

```
import java.util.HashMap;
```

次に，HashMap を作成する場合は，添え字としてつかうクラスと，データとして入れるクラスを指定します.

```
HashMap<String, PersonalData> addressMap = new HashMap<String, PersonalData>();
```

ArrayList や HashSet では保存するデータの型だけを<>で囲んでいましたが，Map の場合はキーとバリューの両方の型を指定しなければけません．ここでは，文字列をキーにして，個人データを保存するので,

```
HashMap<String, PersonalData>
```

とします.

次に，データを入力します.

```
addressMap.put("マリオ", mario);
```

ArrayList では，添え字を指定してデータを保存する add メソッドがありました.

```
addressList.add(0, kuppa);
```

これが，HashMap の場合は put というメソッドになると思ってください.

これで,「マリオ」という文字列をキーとして HashMap にデータが入力されました．あとは，取り出してみるだけですね．取り出す場合は，ArrayList の場合と同じく get メソッドを使います．引数は取り出したいデータと結びつけたキーです.

```
PersonalData data = addressMap.get("マリオ");
```

これで，PersonalData 型の変数 data にはマリオの個人データが代入されます.

以上が，簡単な HashMap の基本的な操作方法です.

10.6.3 HashMap の構造

HashMap は，キーとバリューの結合によってデータが管理されています.

つまり，あるキーに対して，１対１でデータが一つ対応付けられているわけです.ArrayList の場合は，ある数字一つに対してデータが一つ対応付けられていましたが，それが数字ではなくて何でも良くなったものだと思ってください.

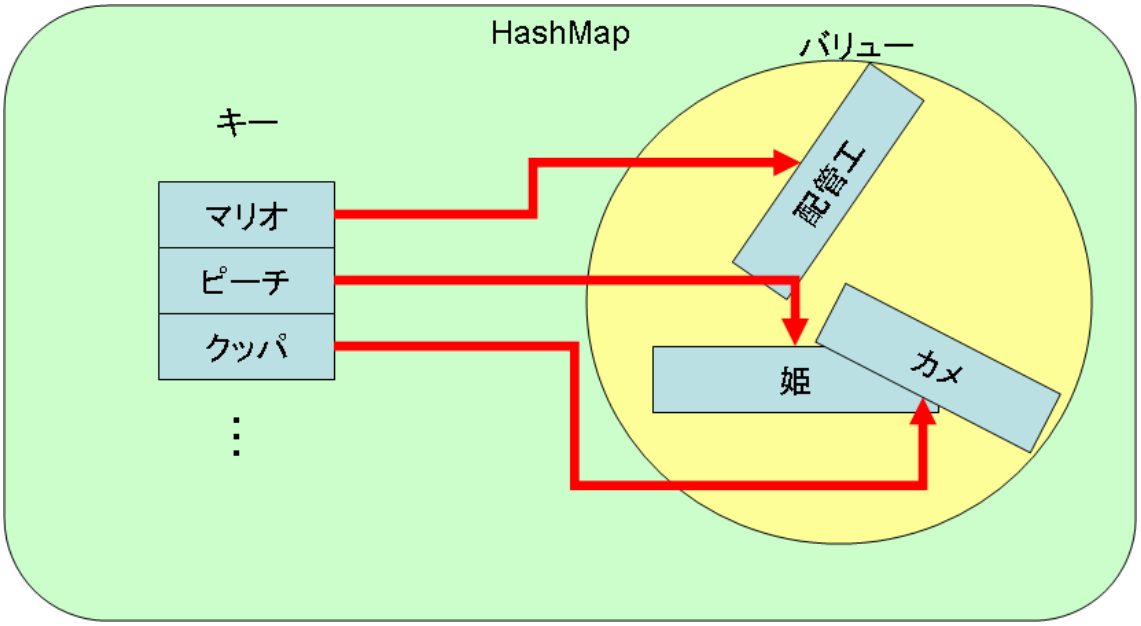


図 3 HashMap の構造

図３の HashMap の構造を見てもらうと，キーとして「マリオ」「ルイージ」「クッパ」などがあるのが分かります．そして，それぞれが指すデータが，配管工，姫，カメとそれぞれの職業になっています．いや，カメが職業がどうかは良くわかりませんが．

このようにデータを整理しておくことで，「マリオの職業は何だっけ？」と思ったときに，マリオというキーさえ入れればすぐに配管工というデータが取り出せるようになるわけです．

これは，辞書を作る際に，単語をキーにして，その意味をバリューとして持つのと同じようなものだと思ってください．

10.6.4 HashMap の全データの取得方法

ArrayList は HashMap でいうキーが添え字として、数字で 0 から順番につけられるため、データ数さえ分かればデータをすべて取り出すことができました。

```
for(int i = 0; i < addressList.size(); i++){
        PersonalData pd = addressList.get(i);
        System.out.println(pd.getName());
}
```

一方、HashMap は添え字となるデータが任意のクラス型として指定できるため、どんなキーと結びつけてデータを作ったのかわかりません。

そこで、HashMap にある全キーをすべて取り出す方法をご紹介しましょう。HashMap のキーをすべて取得するには、keySet メソッドを利用します。

```
Set<String> keySet = addressMap.keySet();
```

このとき、返値は Set 型です。Set 型ははじめて出てきましたが、実はこれインターフェースです。インターフェースを忘れた方はちょっと戻って第 8 章を読み直してみてくださ

いね。

で、Set がどんなインターフェースかといえば、HashSet の機能をもつクラスが持つものなんです。HashSet 自体も Set インターフェースを実装したものですので、

```
Set<String> set = new HashSet<String>();
```

こんなことだってできてしまいます。

まあ、とにかく、HashMap の keySet メソッドを使えば、HashSet が返ってくるようなものだと思って使ってもらえば問題ありません。

したがって、変数 keySet に addressMap のキーがすべて入っているわけです。

ということで、登録されている全データを表示したいのであれば、

```
// 全部取り出す
Set<String> keySet = addressMap.keySet();
// 全部取り出す
for (String name :keySet) {
        PersonalData personalData = addressMap.get(name);
        System.out.println(name + "の住所データ:" + personalData.getName());
}
```

とすることで、HashMap 内の全データを取得することが可能となります。

```
ルイージの住所データ:ルイージ
マリオの住所データ:マリオ
```

ただし、キーが Set 型で返ってくることから分かるように、順番は入れた順番とは無関係ですので、これまたご注意を。

ちなみに、いちいち Set 型の変数を作らなくても、

```
for (String name :addressMap.keySet()) {
        PersonalData personalData = addressMap.get(name);
        System.out.println(name + "の住所データ:" + personalData.getName());
}
```

こうしてしまっても同じです。

このテクニックはかなり使い勝手がよいので、ぜひ覚えておいてください。

10.6.5 存在しないキーを指定した場合

HashMap では、キーを指定すれば対応したデータをとることができます。では、指定したキーに関連したデータが存在しない場合はどうなるでしょうか？

ちょっと確認してみましょう。

```
HashMap<String, PersonalData> addressMap = new HashMap<String, PersonalData>();
PersonalData mario = new PersonalData("マリオ");
PersonalData luigi = new PersonalData("ルイージ");
addressMap.put("マリオ", mario);
addressMap.put("ルイージ", luigi);

//存在しない住所を取り出す
PersonalData data = addressMap.get("クッパ");
System.out.println("クッパの住所データ:" + data.getName());
```

ここでは、マリオとルイージの個人データしか入力していないのにもかかわらず、クッパのデータを取得しようとしています。さあ、実行するとどうなるでしょうか？

```
Exception in thread "main" java.lang.NullPointerException
        at  NoData.main(NoData.java:21)
```

おっと、エラーメッセージが発生してプログラムが終了してしまいました。

ただし、このときエラーが発生したのはデータを取得した瞬間ではなくて、変数 data を利用しようとしたときであるということに気をつけてください。

実は、存在しないデータを取得しようとすると、null という特殊な値が返ってくることが決まっています。null については次章で詳しく説明しますが、簡単に言えば「何もない」ことを知らせる特殊な値だと思ってください。

ここで覚えておくべきことは、**データが存在しなければ null が返ってくる**、という点です。

でも、それでは万が一存在しないデータを取得しようとしたときにちょっと困ってしまいますよね。

そこで、存在しない可能性があるキーをもとにデータを取り出すときはまず、そのキーが HashMap 内に存在するかどうかをチェックすると良いでしょう。

そのためのメソッドが containsKey です。

```
if(addressMap.containsKey("クッパ")){
        PersonalData data = addressMap.get("クッパ");
        System.out.println("クッパの住所データ:" + data.getName());
}
else{
        System.out.println("クッパの住所データはありません");
}
```

containsKey は、引数としてキーを渡して、そのキーが存在すれば true を、存在しなければ false を返してくるメソッドです。

これを実行すれば、

```
クッパの住所データはありません
```

となり、エラーが発生することなくデータが存在しないことが確認できました。

10.6.6 データの削除

HashMap ではいらなくなったデータを削除することができます。そのためのメソッドが remove です。

remove では、あるキーを指定すると、そのキーとそれに対応付けられていたデータが削除されることになります。

その例として以下のようなものを見てみましょう。ここでは、例のごとくマリオの住所録から浮気相手の名前を探し出して削除しています。

```
for (String name :addressMap.keySet()) {
        PersonalData personalData = addressMap.get(name);
        System.out.println(name + "の住所データ:" + personalData.getName());
}

System.out.println("いらないデータを削除します");
PersonalData uwakiaite = addressMap.remove("デイジー");

for (String name :addressMap.keySet()) {
        PersonalData personalData = addressMap.get(name);
        System.out.println(name + "の住所データ:" + personalData.getName());
```

```
                }
```

ポイントは、remove メソッドでキーを引数にしている点ですね。

この実行結果は・・・

```
デイジーの住所データ:デイジー　←浮気発見！
ルイージの住所データ:ルイージ
マリオの住所データ:マリオ
いらないデータを削除します
ルイージの住所データ:ルイージ　←データが消されました
マリオの住所データ:マリオ
```

というわけで、無事データを削除することができました。ちなみに、remove メソッドの返値は、そのキーに関連付けられていたデータです。ここでは、デイジーの個人情報が返り値として戻ってきます。

HashSet の場合と大きく異なっているのが、HashSet の場合、データを削除するためには削除したいデータを指す変数を持っていなければいけませんでした。つまり、デイジーの個人データを削除したければ PersonalData 型のインスタンスがわかっていなければいけなかったのです。

しかしながら、HashMap の場合はキーが分かっていればインスタンスそのものは輪からなくても削除することができます。これは結構大きな差ですので、HashMap を有効利用するのに役立つと思います。

なお、全データを消去する方法は以下のとおりです。

```
addressMap.clear();
```

10.7　Collection フレームワーク

10.7.1 ArrayList,HashSet、HashMap の比較

さて、ここまで ArrayList,HashSet,HashMap について説明してきました。これらのクラスは、すべてデータを保存するためのクラスでしたよね。

ある意味似たような機能を持っているクラスということができます。

実はこれらのクラス Java においては、Collection フレームワークという名前で、同じグループのクラスであると考えられています。

その証拠に、すべて java.util パッケージに所属していたり、似たようなメソッドで使うことができたりしますよね。

そんな良く似た三つのクラスの機能をここで一覧にしたいと思います。違いが分からなくなったらここを見ると比較的良く分かるかもしれません。

	ArrayList	HashSet	HashMap
データの挿入	add(データ)	add(データ)	put(キー,データ)
任意の位置への入力	set(添え字, データ)	使用不可	put(キー,データ)
任意のデータ取得	get(添え字)	使用不可	get(キー)
データの削除	remove(添え字)	remove(データ)	remove(キー)
全データ削除	clear()	clear()	clear()
データ量	size()	size()	size()
データの重複	可能	不可	可能
キー(添え字)の重複	不可		不可
キー(添え字)の種類	int 型のみ		任意の参照型

10.7.2 ArrayList の仲間

ArrayList クラスは、List というインターフェースを実装しています。

List インターフェースを実装しているクラスは他にもいくつかありますが、それらはすべて ArrayList と同じメソッドを持っており、それ以外に ArrayList とちょっと異なる機能を持っています。そんな List インターフェースを実装した ArrayList の仲間を簡単にいくつか紹介しましょう。

- LinkedList

  LinkedList は、ほとんど ArrayList と同じですが、ただデータの挿入にかかる時間が非常に短いのが特徴です。

  通常 ArrayList ではデータを挿入には結構時間がかかることが知られています

```
list.add(データ)
```

  それに対して、LinkedList はこの作業がかなり高速化されています。

  ただし、逆に、

```
list.get(0)
```

  のような取得には ArrayList よりも時間がかかるという欠点があります。データの挿入が非常におおいデータを扱うときに有効利用が可能でしょう。

- Stack

  Stack はスタックと呼ばれるデータ構造を実装するために使われるクラスです。

  Stack とは、後入れ先出しを実現するためのメソッドがいくつか用意されています。

```
Stack<String> stack = new Stack<String>();
//先頭にデータを入力
stack.push("データ入力");
//先頭のデータを取得して、そのデータを削除
stack.pop();
```

10.7.3 HashSet の仲間

HashSet は Set インターフェースを実装したクラスです。Set インターフェースを実装したほかのクラスは HashSet のもつほとんどの機能を持っている上に、さらに高機能だったりします。

そういえば、HashMap の keySet メソッドを使ったときの返値も Set 型でしたよね。基本的に集合をあらわすクラスは大抵 Set インターフェースを実装していますので、その仲間を覚えておくと便利かもしれません。

- LinkedHashSet

  LinkedHashSet はすでにちょっと紹介しましたが、入れた順番に出力されることが保障されている Set です。

  通常 HashSet は入れた順番とは無関係にデータが取得されますので、データを入れた順番が重要な場合は LinkedHashSet を使うと良いでしょう。

- TreeSet

  TreeSet は LinkedHashSet と似ていて、出力される順番が一意に決定されます。ただし、それは入れた順番ではなくて、データをソートした順番になります。

  たとえば、以下のように TreeSet 型に適当な順番で数字を入れていったとしましょう。

```
import java.util.Set;
import java.util.TreeSet;

public class TreeSetMain {
	public static void main(String[] args) {
		Set<Integer> treeSet = new TreeSet<Integer>();
		treeSet.add(34);
		treeSet.add(1);
		treeSet.add(100);
		treeSet.add(-100);

		for(int i:treeSet){
			System.out.println(i);
		}
	}

}
```

これを実行してみると、

```
-100
1
34
100
```

数字が小さいほうから順番に並べられて出力されました。

このように、TreeSet を使うことで自動的にデータを並べて出力させることができるようになります。このようなデータの並べ替えを**ソート**といい、いろんな場面で使う機会が多いと思います。今回は詳しい説明は省きますが、こういう機能を持った Set、TreeSet があるということを覚えておいてください。

ちなみに TreeSet には、ある値以上のデータを削除する、とかある値以上のデータ

だけを取得する、といったこともできます。この辺については JavaDoc というクラスの説明書を読んでもらえればいろいろな情報があると思います。

10.7.4 HashMap の仲間

HashMap にもいくつか仲間がいます。これらは皆 Map インターフェースを実装しています。Map インターフェースを実装していれば、HashMap と同じ機能を持ちながら、さらに高度な機能を持っているものだと思ってください。

- LinkedHashMap

  LinkedHashMap は keySet メソッドによって Set を取得したときに、その Set が LinkedHashSet とほぼ同じで入れた順番にキーが取得ができるようになっているものです。

- TreeMap

  TreeMap は、keySet によって取得したキーの Set が TreeSet になっていて、自動的にソートされているものになります。ようするに、キーがソートされた状態で取得できるわけです。

  ここで注意してほしいのが、あくまでもソートされているのはキーであって、データではないという点ですね。データをソートしたい場合は、TreeSet を使わなければいけません。

10.7.5 データの相互変換

ArrayList(を含めた List インターフェースを実装したクラス)と、HashSet(を含めた Set インターフェースを実装したクラス)は、互いのデータを相互に変換することができます。

その方法は非常に簡単で、コンストラクタに引数として入れるだけです。

```
ArrayList<PersonalData> addressList = new ArrayList<PersonalData>();
PersonalData mario = new PersonalData("マリオ");
PersonalData luigi = new PersonalData("ルイージ");

HashSet<PersonalData> addressSet = new HashSet<PersonalData>(addressList);
```

これによって、マリオとルイージの個人データが入った HashSet を簡単に作ることができます。

もちろん、逆に ArrayList のコンストラクタに Set の変数を入れれば Set の中身が入った ArrayList を作ることも可能です。

また、一度作ったデータに対して addAll メソッドを使ってもデータを追加可能です。

```
HashSet<PersonalData> addressSet = new HashSet<PersonalData>();
addressSet.addAll(addressArray);
```

こうすることで、addressSet に addressArray の全データが追加されます。

10.7.6 ArrayList と HashSet の応用例

これを利用すると、ちょっと面白いことができてしまいます。その例を以下に示しましょう。

問題：以下に与える配列の中に、何種類の漢字が含まれているのかを数え上げよ

```
String[] kanjiArray = {
                    "鱧", "鯏", "鮫", "鯖", "鱧", "鯊", "鮎", "鯡", "鮭", "鯖",
                    "鯵", "鮭", "鯊", "鮎", "鯖", "鯖", "鮭", "鱚", "鰐", "鮑",
                    "鮫", "鯡", "鱧", "鯏", "鯱", "鮎", "鯊", "鯊", "鱧", "鮪"
};
```

この問題を解くのに、いちいち漢字の種類を調べていっていたのでは時間がかかってしまいます。しかしながら、一気にこの問題を解決する方法もあるのです。それが、以下のコードです。

```
List<String> kanjiList = Arrays.asList(kanjiArray);
HashSet<String> kanjiSet = new HashSet<String>(kanjiList);
for(String kanji:kanjiSet){
        System.out.println(kanji);
}
System.out.println("全部で"+kanjiSet.size()+"種類");
```

長いようですが、実際に作業をしているのは、太字にしたわずか 2 行だけです。それ以外は最後の確認を行う部分です。

さあ、これを実行すると、

```
鰐
鮎
鯊
鯏
鮫
鱧
鯖
鮑
鯡
鮭
鮪
鱚
鮫
鯱
鯵
全部で 15 種類
```

と出力され、重複した漢字はすべて取り除かれ、全部で何種類の漢字が与えられたかだけを取り出すことができました。これで、寿司屋の湯飲みを作ったときに重複した漢字を書く心配はありません。

10.8 Calendar

10.8.1.1 Calendar の基本

Calendar は、その名前とは裏腹に、時間を表すクラスです。年月日だけではなく、時間、分、秒、ミリ秒まで表現することが出来ます。

Calendar クラスも java.util パッケージにありますので、まずインポートするところからはじめます。

```
import java.util.Calendar;
```

次に、カレンダー情報を使っていくのですが、通常、クラスのインスタンスを取得するには new を使うのにたいして、Calendar のインスタンスを取得する方法はちょっと変わっています。

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

ちょっと特殊ですが、こういうものだと思ってください。
では次に、年月日を Calendar から取得してみましょう。

```
	public static void main(String[] args) {
		Calendar calendar = Calendar.getInstance(); // カレンダー作成
		int year = calendar.get(Calendar.YEAR); // 年を取得
		int mon = calendar.get(Calendar.MONTH) + 1; // 月を取得
		int date = calendar.get(Calendar.DATE); // 日を取得

		System.out.println(year+"/"+mon+"/"+date);

	}
```

これを実行すれば、

```
2007/4/1
```

となり、ちょっと嘘っぽい日を取得することができました。

Calendar から年月日などを取得するときは、get を利用します。 get メソッドでは、指定した引数に関する情報を取得できます。具体的には、年月日時分秒などを取得可能です。それぞれを取得するときの引数を、以下の表にまとめます。

引数	対応する値
Calendar.YEAR	年
Calendar.MONTH	月
Calendar.DATE	日
Calendar.HOUR	時
Calendar.MINUTE	分
Calendar.SECOND	秒

ただし、このとき注意しなければいけないことがあります。**月だけは本来の月から 1 を引いた値になる**のです。例えば、1 月ならば 0 が返りますし、5 月なら 4 が返ります。なぜこんな仕様になっているのかは謎ですが、こういうものなので気をつけましょう。

```
calendar.set(Calendar.YEAR, 2005);        //年を設定
calendar.set(Calendar.MONTH, 5);          //月を設定(6 月を指定)
calendar.set(Calendar.DATE, 20);          //日を設定
```

次に、日付の設定方法です。日付の設定には set を使います。get と同様に年月日を表す引数と、そこに与える値を設定します。ただし、月に関しては「現実の月－1」の値を設定する必要があるので、ご注意を。

10.8.2 Calendar のその他の機能

日付の設定方法と取得方法が分かったので、他のメソッドもどんどん紹介しましょう。

```
long milliTime = calendar.getTimeInMillis();        //時間をミリ秒で取得
```

getTimeInMillis は、1970 年 1 月 1 日から何ミリ秒たったかを返します。例えば、さっき調べてみたら、1119277243546 を指していました。もう一回やってみたら、1119277317406 を指しました。これらの差を取れば 73860 ミリ秒、つまり 73.860 秒経過したことが分かります。このように、getTimeInMillis により得られた経過時間の差の絶対値をとることで、Calendar 同士の差を取得することが可能です。

```
Calendar targetCalendar = Calendar.getInstance();   //カレンダー作成
calendar.after(targetCalendar);//targetCalendar の後かどうか
calendar.before(targetCalendar);          //targetCalendar より前かどうか
```

after と before はその名の通り、対象となる Calendar より先か後かを返します。二つのカレンダーを比較するときに利用します。

```
calendar.add(Calendar.DATE, 35);
```

最後に日付を加算する add メソッドをご紹介します。このメソッドでは、get, set と同じように「年月日を表す引数」、それから「加算する値」を渡すことで、その値の分 Calendar の日付を移動させます。この例では、35 日カレンダーの日付を進めています。マイナスの値を入れれば、日付を戻すことも出来ますし、add の結果月をまたぐ変更がおこれば、ちゃんと月も変化してくれます。

```
//2005 年 3 月 20 日=立春に設定
```

```
calendar.set(Calendar.YEAR, 2005);          //年を設定
calendar.set(Calendar.MONTH, 2);            //月を設定
calendar.set(Calendar.DATE, 20);            //日を設定

calendar.add(Calendar.DATE, 88);            //88 日後=6 月 16 日
```

これで、八十八夜を見逃すことはありません。お茶摘みのタイミングを逃すこともなく、静岡の皆さんも安心。

10.8.3 Calendar を使った実行時間測定

Calendar クラスには、.getTimeInMillis()という 1970 年 1 月 1 日から何ミリ秒か取得できます。これを利用すると、ある処理にかかった時間を調べることができます。

調べるには以下のようにします。

```
long startTime = Calendar.getInstance().getTimeInMillis();
//処理
long endTime = Calendar.getInstance().getTimeInMillis();

long time = endTime-startTime;
System.out.println("処理に"+time+"ミリ秒かかりました");
```

こうすることで、処理にどのくらい時間がかかったのか調べられます。

```
処理に 687 ミリ秒かかりました
```

もし、プログラムを作っていて終了までに時間がかかるようなことがあったら、このテクニックを使ってどの処理に時間がかかっているのか調べてみるといいでしょう。

10.9 数学関連

10.9.1 Math

10.9.2 Random

余裕があれば書きます。

10.10 間違えやすい API

10.10.1 ArrayList は要素数を決めて作成できない

配列は、最初から要素数を決めて作成することができました。

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

一方で、ArrayList は要素数をあらかじめ決めて作成することができません。ArrayList を使う場合は、add を使って要素を追加していくしかありません。

ですので、いきなりこんなことはできないのです。

```
static public void main(String[] args){
        ArrayList<String> list = new ArrayList<String>();
        list.set(0, "テスト");
}
```

すでに、要素がある添え字にだけ set が可能ですので注意してください。

ArrayList は自動的に配列の長さを変更してくれるため、どこにデータを入れてもいいと思い込みがちですが、あくまでもデータを入れることができるので、すでに別のデータが入っている添え字に限られます。

間違えやすいポイントなのでご注意を。

10.10.2 for の途中で削除できない HashSet、HashMap

HashSet では、for 文を使って全データにアクセスします。そのとき、途中にあるデータを消したい場合はどうすればよいでしょうか？

remove を使えばよいような気がしますが・・・

```
HashSet<Integer> set = new HashSet<Integer>();

for(int i = 0; i < 100; i++){
        set.add(i);
}

for(Integer i: set){
        if(i % 2 == 0){
                set.remove(i);
        }
}
```

これによって、set 内にある 2 で割り切れる数字、つまり偶数だけ削除しようとしてしまおうとしています。

しかしながら、これを実行すると、

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at
UnableToRemoveMain.main(UnableToRemoveMain.java:18)
```

こんなエラーが発生しました。

実は、for 文で全データを利用しようとしているときに、remove を使うことはできないという決まりがあるのです。

では、どうすればよいのでしょうか？

HashSet を勉強したときに、Java1.4 以前では for 文の書き方が異なるというお話をしましたよね。覚えていますか？

```
for(Iterator<Integer> it = set.iterator(); it.hasNext();){
        Integer i = it.next();
        //処理
}
```

こちらを使うと、削除を行う方法があるんです。

それが、こちらです。

```
for(Iterator<Integer> it = set.iterator(); it.hasNext();){
        Integer i = it.next();
        if(i % 2 == 0){
```

```
                    it.remove();
                }
            }
```

ポイントは、it.remove()という部分ですね。

ちょっと理解が難しいところかもしれませんが、こういうものだと思ってください。これによって、特定のデータを削除することができます。

あるいは、こんな方法もあります。

```
HashSet<Integer> removeDataSet = new HashSet<Integer>();
for(Integer i: set){
        if(i % 2 == 0){
                removeDataSet.add(i);
        }
}
set.removeAll(removeDataSet);
```

ここでは、削除したいデータをあらかじめ removeDataSet という HashSet に入れておいて、あとで、removeAll メソッドを使って削除を行っています。

ちなみに、removeAll は引数として与えた Set と一致するデータをすべて削除するというメソッドです。

このように、HashSet の中身を for 文で見ながらデータを削除するにはちょっとしたテクニックが必要になってきますので、ご注意ください。

10.10.3 同一性の判断

HashSet では、同一性の判断が非常に重要になります。

HashSet は重複を許しませんので、同じデータかどうかを判断する必要があるわけです。また、remove や contains を使う場合も同一性の判断は重要となります。さらに、HashMap においては、キーが一致しているかどうかも同一性の判断ということになります。

この同一性はどう判定されるのでしょうか？

実は、**equals メソッド**と、**hashCode メソッド**の返値の同一性によって判断されるのです。これらは二つとも Object クラスのメソッドですのですべてのクラスで利用できます。そのすべてのクラスで利用可能な二つのメソッドによって同一性を確認します。詳しく見ていきましょうか。

通常二つのオブジェクト同士が一致するかどうかを判断する材料として、equals と==という二種類のものがあります。

String 型だと、この二つで大きな違いがあったことを覚えているかと思います。

```
String boss = new String("クッパ");
String koreanFood = new String("クッパ");
```

```
if(boss == koreanFood){
        System.out.println(boss+" == "+koreanFood);
}
else{
        System.out.println(boss+" != "+koreanFood);
}

if(boss.equals(koreanFood)){
        System.out.println(boss+" equals "+koreanFood);
}
else{
        System.out.println(boss+" not equals "+koreanFood);
}
```

このようなプログラムを書くと、

```
クッパ != クッパ
クッパ equals クッパ
```

二つのインスタンスは==では同一ではないと判断され、equals では同一と判断されます。

では、HashSet の場合どうか確かめて見ましょう。

```
HashSet<String> set = new HashSet<String>();
set.add(boss);
set.add(koreanFood);

System.out.println("データ数は"+set.size());
```

もし、同一と判断されれば、ひとつしかデータは入っていなく、同一ではないと判断されれば二つデータが入っているはずです。

```
データ数は 1
```

というわけで、二つのクッパは同じものと考えられていることが分かりました。

ここから分かるとおり、HashSet において同一性は equals で true かどうかによって判断されます。

しかし、実はそれだけではありません。先ほども述べたとおり、hashCode というメソッドも重要な鍵を握っています。

hashCode とは int 型を返すメソッドで、

あるインスタンス A,B があったとき、

```
A.equals(B)
```

が true ならば、

```
A.hashCode()==B.hashCode()
```

が成り立つような値を返さなければいけないという決まりがあるメソッドです。

これは、通常 JavaAPI を使っている分には意識する必要はないのですが、自作クラスを作る場合は注意してください。

たとえば、PersonalData クラスで、同一性を名前によって判断するように変更したとしましょう。

```
public class PersonalData {

        //中略
```

```
	public boolean equals(Object obj){
		if (obj instanceof PersonalData) {
			PersonalData pd = (PersonalData) obj;
			return name.equals(pd.name);

		}
		return false;
	}
}
```

これは、equals の引数が PersonalData の場合、name の equals の返値を返すクラスです。ポイントは、

```
		if (obj instanceof PersonalData) {
```

です。

これによって、obj が PersonalData 型のインスタンスならば true、そうでなければ false となります。

この PersonalData を使って、以下のようなプログラムを作って見ます。

```
		HashSet<PersonalData> addressMap = new HashSet<PersonalData>();
		PersonalData boss = new PersonalData("クッパ");
		PersonalData kuppa = new PersonalData("クッパ");

		addressMap.add(boss);
		addressMap.add(kuppa);

		for(PersonalData personalData: addressMap){
			System.out.println(personalData.getName());
		}
		System.out.println("データ数は"+addressMap.size());
```

equals で name が一致していれば true を返すようにしていますから、これで addressSet の中にはクッパのデータはひとつしか入らないはずですが・・・

```
クッパ
クッパ
データ数は 2
```

おっと、残念ながら二つのクッパは同じデータと認められなかったようです。これは、先ほどから説明しているとおり、hashCode の値が一致していないからなのです。

ためしに、二つのクッパの個人データの hashCode を確認してみましょう。

```
		System.out.println("boss の hashCode は"+boss.hashCode());
		System.out.println("kuppa の hashCode は"+kuppa.hashCode());
```

これを実行してみると、

```
boss の hashCode は 17510567
kuppa の hashCode は 27744459
```

うん、確かに二つの hashCode は異なっているようです。これでは、equals は true を返しても、hashCode は一致していないので、HashSet ではうまく同一と判断してくれないのです。

では、PersonalData で hashCode もオーバーライドして、名前が同じなら同じ値を返すように変更します。

```
public class PersonalData {
```

```
        //中略
        public int hashCode(){
                return name.hashCode();
        }
}
```

ポイントは、**equals の判断で利用する変数の hashCode をそのまま使ってしまう**点です。これを守っていれば、equals が true なれば hashCode も自動的に一緒になるからです。

このように変更してから再び HashSet に入れてみましょう。

```
                HashSet<PersonalData> addressMap = new HashSet<PersonalData>();
                PersonalData boss = new PersonalData("クッパ");
                PersonalData kuppa = new PersonalData("クッパ");

                addressMap.add(boss);
                addressMap.add(kuppa);

                for(PersonalData personalData: addressMap){
                        System.out.println(personalData.getName());
                }
                System.out.println("データ数は"+addressMap.size());

                System.out.println("boss の hashCode は"+boss.hashCode());
                System.out.println("kuppa の hashCode は"+kuppa.hashCode());
```

この結果は

```
クッパ
データ数は 1
boss の hashCode は 12376413
kuppa の hashCode は 12376413
```

というわけで、二つのクッパは同じものとして扱われることになりました。

自作クラスを作成する場合、equals についてはちゃんと考慮するのですが、hashCode についてはついつい忘れがちなので、hashCode についても忘れないように気をつけましょう。