リスト 3-3　Main.java

```
1  import java.util.*;
2
3  public class Main {
4    public static void main(String[] args) {
5      ArrayList<String> names = new ArrayList<String>();
6      names.add("湊");
7      names.add("朝香");
8      names.add("菅原");
9      Iterator<String> it = names.iterator();
10     while(it.hasNext()) {
11       String e = it.next();
12       System.out.println(e);
13     }
14   }
15 }
```

矢印を次に進められるなら繰り返す

矢印を次に進め、内容を取り出す

拡張 for 文を用いた方法と比べて構文はやや複雑ですが、イテレータを用いることが必要な場面もあります。リストから要素を順に取り出す 3 種類の方法はすべて利用できるようにしておきましょう（図 3-12）。

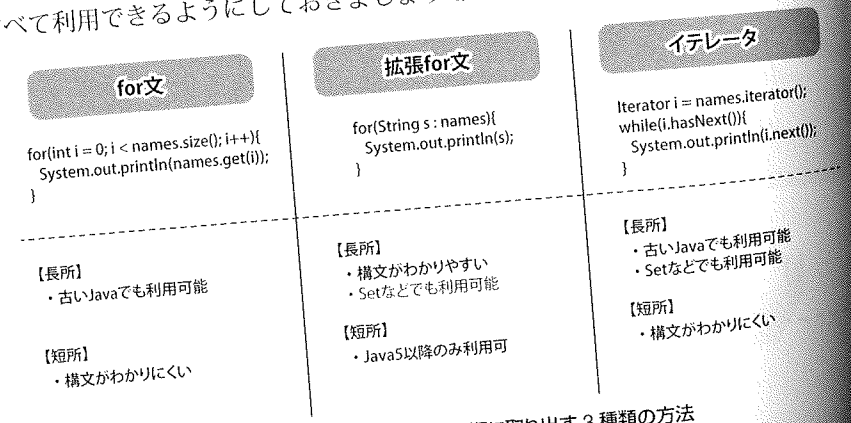

図 3-12　ArrayList の中身を順に取り出す 3 種類の方法

3.3 その他のリスト

3.3.1 LinkedList

コレクションフレームワークには、ArrayList のほかにも LinkedList というクラスが準備されています。両者はどちらもリストを実現しているクラスであり、備えているメソッドや使い方もほとんど同じです。実際、LinkedList は表 3-1（p.79）のメソッドをすべて持っており、たとえばリスト 3-3（p.86）の「ArrayList」を「LinkedList」に書き換えても同じように動作します。

同じ機能なのに、わざわざ違うクラスが準備してあるってことは、何か違いがあるはずですよね？

あぁ。小さな違いなんだけど、場合によっては動作に大きな影響を与えることもあるんだ。

ArrayList と LinkedList には、クラス内部の作り（内部実装）に違いがあります。ArrayList は Java の配列を応用して作られたものですが、LinkedList は**連結リスト**と呼ばれる構造を応用して作られています（次ページの図 3-13）。

「同様の動作を実現するものだが、内部構造が違う」という意味では、ガソリン自動車と電気自動車のようなものかもしれません。ガソリン自動車と電気自動車は、「運転する人にとってはほぼ同じもの」です。しかし長距離の移動であればガソリン自動車が、エコロジー面では電気自動車が有利なように、ArrayList と LinkedList には次ページの表 3-3 のような**内部構造（内部実装）の違いに起因した得意不得意な動作の違い**があります。

要素の挿入や削除が頻繁に発生する場合

リストの途中に要素が挿入または削除されるという処理は、ArrayList がもっ

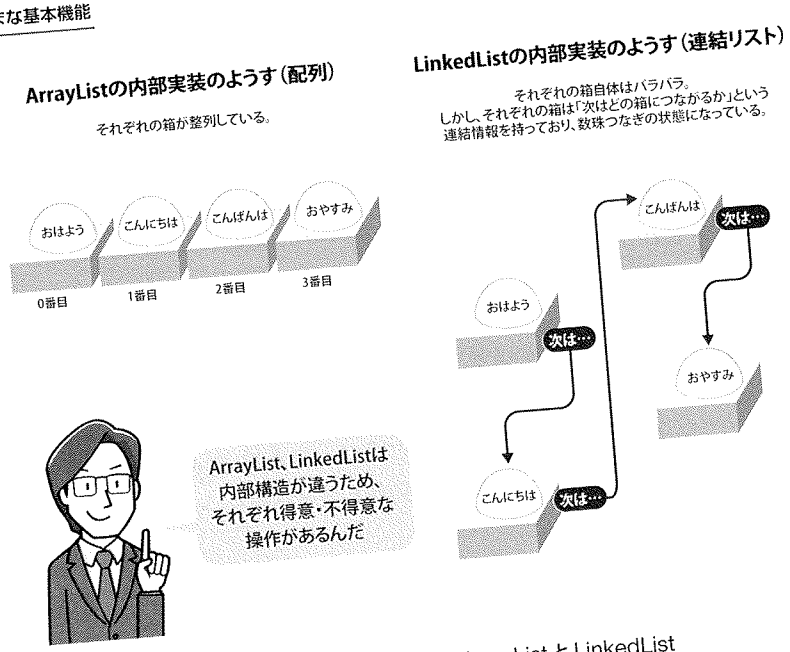

図 3-13　内部構造が異なる ArrayList と LinkedList

ArrayList		LinkedList
配列 （隙間無く並んだ箱）	内部構造	連結リスト （数珠つなぎの箱）
✕（遅い）	要素の挿入・削除 add()、remove()	✓（高速）
✓（高速）	指定位置の要素の取得 get()	✕（遅い）

表 3-3　内部構造の違いによる得意・不得意

とも苦手とする処理です。なぜなら途中の要素が削除されると、それより後ろにあった要素を「玉突き方式」で 1 つずつ前に移動させていく処理が必要になるからです。もし要素数 10,000 のリストの 2 番目の要素が削除されたら、9,998 回の玉突きコピーを行わなければなりません（次ページ図 3-14 の左側）。

一方の LinkedList にとって、要素の挿入や削除は簡単な処理です。たとえば remove() を行う場合、「削除対象の 1 つ前の要素」に対して、次の箱を示す連結情報を書き換えるだけでよいからです（次ページ図 3-14 の右側）。

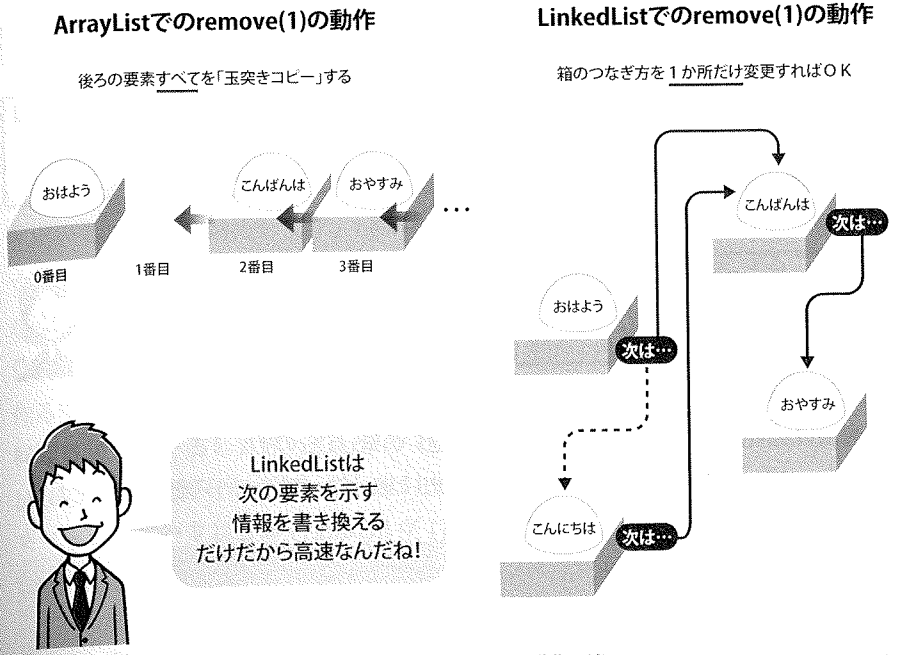

図 3-14　要素を削除した際の動作の違い

添え字を指定して要素を取り出す場合

しかし、LinkedList にも弱点があります。連結リスト方式であるため、get() メソッドを使った要素の取り出しが苦手なのです。連結リストは単なる「数珠つなぎの箱」ですので、それぞれの箱には「○番目」という番号は振ってありません。よって、get() メソッドなどで「○番目を取得せよ」と指示された場合、先頭から○番目まで数えながら辿っていく必要があります。

たとえば、10,000 個の箱が数珠つなぎになっている LinkedList の場合、「9,998 番目の要素を取得する」となれば、先頭から順に 9,998 回も辿っていかなければならないのです。

なお、要素数が多い LinkedList で末尾付近の要素を add() や remove() するときには注意が必要だ。get() 同様、要素に辿り着くまで延々と要素を辿る必要があるため、結果的に ArrayList より遅くなることもあるよ。

3.3.2 ザックリとらえれば、どちらもList

内部の構造や得意不得意が違うけれども、表面的にはとても似ている、「兄弟みたい」な関係なんですね。

APIリファレンスを調べたら、ArrayListとLinkedListは本当に兄弟だという証拠が見つかりました！

ArrayListとLinkedListは内部構造に違いがあるものの、「ざっくり見れば同じList」です。実際、両者はともにjava.util.Listインタフェースを実装しており、今まで紹介したget()やremove()、size()やiterator()などはすべてListインタフェースに定義してあるメソッドです。

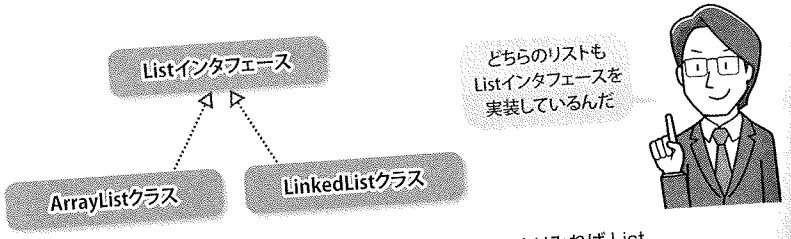

図3-15　ArrayListもLinkedListも、ザックリみればList

「ArrayList is-a List」ですし、「LinkedList is-a List」ですので、次のようなソースコードを書くことも可能です。

```
List<String> list1 = new ArrayList<String>();
List<Hero> list2 = new LinkedList<Hero>();
```

このコードでは、右辺でnewをするときに具体的なクラス名（ArrayListやLinkedList）を使い、左辺の変数の型としてはあいまいなインタフェース名（List）を利用しています。

このように、コレクションのインスタンスは極力あいまいな型で取り扱うのが「通」なやりかただよ。

リストインスタンスを格納する変数や引数・戻り値の型は、極力あいまいな型を用いることをお勧めします。newをする段階ではどの実装を利用するか意識する必要はありますが、その後の利用においては実装の違いを気にすることは少なく、**「ざっくりListとして扱う」ほうがメリットが大きい**からです。

たとえば、あるリストを受け取って、その要素のすべてを表示するメソッドを作る場合のことを考えてみましょう。次のコードのようにArrayListだけを受け付けるメソッドとして作ってしまうと、利用する側にとってはLinkedListなどほかのリストを渡せなくなってしまいます。

```
public static void printList(ArrayList<String> list) {
  for(String s : list) {
    System.out.println(s);
  }
}
```

しかし、このprintList()メソッドの仕事は「リストの中身を取り出して1つずつ表示すること」であって、リストの内部構造が配列であるか連結リストであるかは関係ありません。

このメソッドは引数をList＜String＞型に変更することで、ArrayList＜String＞やLinkedList＜String＞など、どんなリストでも受け取って中身を表示できるようになります。

インタフェース型の活用

引数・戻り値・ローカル変数には、極力あいまいな型（インタフェース型）を利用できないかを検討し、積極的に利用する。

3.4 さまざまなコレクションクラス

3.4.1 コレクションクラスの全体像

前節では、リスト構造に関する３つの代表的な型（ArrayList、LinkedList、List）を紹介しました。しかし、コレクションフレームワークでは、リスト以外にもたくさんのデータ構造に関するクラスやインタフェースを提供しています。ここでコレクションフレームワークの全体像をご覧に入れましょう（図3-16）。

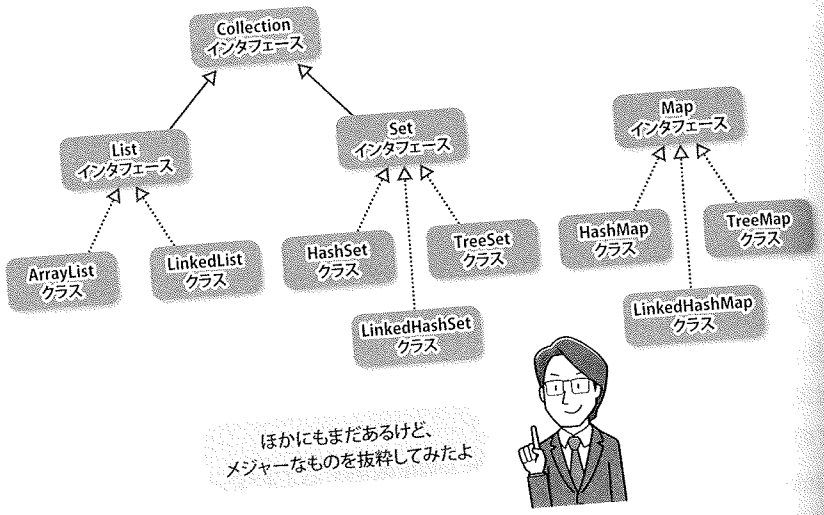

図3-16　コレクションフレームワークに含まれるさまざまなクラス


今まで学んできたのは左端の３つだけで、ほかにもいろんなクラスがあるんだなぁ。


大きく見ると、List 関連、Set 関連、Map 関連の３種類みたいね。

3.4.2 java.util.HashSet クラス

java.util.Set インタフェースを実装するコレクションクラスの中でもっとも一般的なものが、java.util.HashSet クラスです。これら Set 関連クラスは複数の情報を重複なく格納する**集合（セット）**というデータ構造を実現するためのものであり、それぞれの要素に順序がないことが一般的です。

たとえば「信号の色をひとまとめにして管理する」ことを考えましょう。おそらく要素としては「赤・青・黄」の３つの要素が含まれることになり、「赤・赤・青・黄」などと重複することはありえません。また、３つの色がどのような順番であるかは関係なく、とにかく「赤」と「青」と「黄」が含まれていればよいはずです。このように「重複は許さないが、その順番は問わない」というデータの集まりを利用したい場合に Set 関連クラスを用います（表3-4）。

表3-4　Set インタフェースが備えるメソッドの一覧

戻り値	メソッド	意味
要素を格納する		
boolean	add(●)	セットに要素を追加
要素を取り出す		
（なし）		
セットを調査する		
int	size()	格納されている要素数を返す
boolean	isEmpty()	要素数がゼロであるかを判定
boolean	contains(●)	指定要素が含まれているか判定
要素を削除する		
void	clear()	要素をすべて削除する
boolean	remove(●)	指定した内容の要素を削除する
その他		
Iterator<●>	iterator()	要素を順に処理するイテレータを返す

※●は要素を示す。

Set の基本特性

- それぞれの要素には、**重複が許されない**
- それぞれの要素には、**基本的に順序関係がない**

Set 関連クラスには、前ページの表 3-4 のようなメソッドが備わっています。ぜひ List 関連のクラスが備えるメソッド一覧(p.79 の表 3-1)と対比しながら確認してみてください。

Set は List と少し似ていますが、List と比較して注意しなければならないのは、次の 3 点です。

1. 重複した値を格納しようとすると無視される

add() メソッドを呼び出すことで要素を格納できますが、すでに同じもの(equals() で等価と判断されるもの)が格納されている場合は無視されます。

リスト 3-4　Main.java

```
1  import java.util.HashSet;
2  import java.util.Set;
3
4  public class Main {
5    public static void main(String[] args) {
6      Set<String> colors = new HashSet<String>();
7      colors.add("赤");
8      colors.add("青");
9      colors.add("黄");
10     colors.add("赤");
11     System.out.println("色は" + colors.size() + "種類");
12   }
13 }
```

重複して赤を格納しようしても無視される

実行結果

```
色は3種類
```

追加が無視された

2. set() や get() がない

リストには「○番目の要素を取得する」「○番目に要素を設定する」ために get()

メソッドや set() メソッドが存在しました。しかし、要素同士に順序がない Set には、そもそも「○番目」という概念がなく、添え字を使った操作は行えません。

get() が使えなくなったら、セットに入れた値を 1 つずつ取り出せなくなっちゃいませんか？

大丈夫よ。拡張 for 文やイテレータを使えばいいじゃない。

そのとおり。でも落とし穴があるから注意しよう。

3. 1 つずつ取り出す場合の順序は不明

セットの要素には順序がありません。よって拡張 for 文やイテレータを使ってセットの中身を 1 つずつ取り出す場合、**どのような順序で要素が取り出せるかは一切確約されていない**ことに注意が必要です。

次のリスト 3-5 の実行結果を見ればわかるように、格納した順序とはまったく異なる順序で取得することもあります。また、実行するたびに異なる順序で値を取得するかもしれません。

リスト 3-5　Main.java

```
1  import java.util.HashSet;
2  import java.util.Set;
3
4  public class Main {
5    public static void main(String[] args) {
6      Set<String> colors = new HashSet<String>();
7      colors.add("赤");
8      colors.add("青");
9      colors.add("黄");
```

赤・青・黄の順に格納

```
    for(String s : colors) {
      System.out.print(s + "→");
    }
  }
}
```

実行結果

```
青→赤→黄→
```

格納の順序と異なっている

※実行のたびに結果が異なる可能性があります。

3.4.3 Set の実装バリエーション

すでに解説したとおり、基本的にセットというデータ構造では要素同士の順序を管理することはなく、その順序性を保証しません。しかし、それでは困るという場合は、以下のルールに従って順序を保証する TreeSet や LinkedHashSet といった実装を利用することができます。

LinkedHashSet ・・・ 値を格納した順序に整列
TreeSet ・・・ 自然順序付けで整列

自然順序付けって、どんな順序なんですか？

それぞれのクラス固有の順序のことだよ。たとえば String クラスでは辞書順になるよう定義されている。詳細は第 4 章で説明しよう。

たとえば、String 型の複数の文字列を TreeSet に格納すると、簡単に「辞書順」で取り出すことができます。

リスト 3-6

Main.java

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

public class Main {
  public static void main(String[] args) {
    Set<String> words = new TreeSet<String>();
    words.add("dog");
    words.add("cat");
    words.add("wolf");
    words.add("panda");
    for(String s : words) {
      System.out.print(s + "→");
    }
  }
}
```

実行結果

```
cat→dog→panda→wolf→
```

3.5 Mapの使い方

3.5.1 ペアを格納するデータ構造

マップ(Map)とは、2つの情報をキー(key)と値(value)のペアとして格納するデータ構造です。格納した値は、キーを指定して読み書きできます。とても便利なデータ構造なので、プログラム開発において多用されます。

Javaではjava.util.Mapインタフェースおよびjava.util.HashMapクラスに代表される各種実装クラスを用いることで、手軽にマップを活用できます。このとき、「String型のデータを格納するList」をList＜String＞型と表現したように、「String型のキーとInteger型の値のペアを格納するMap」は、Map＜String, Integer＞型と表現します(図3-17)。

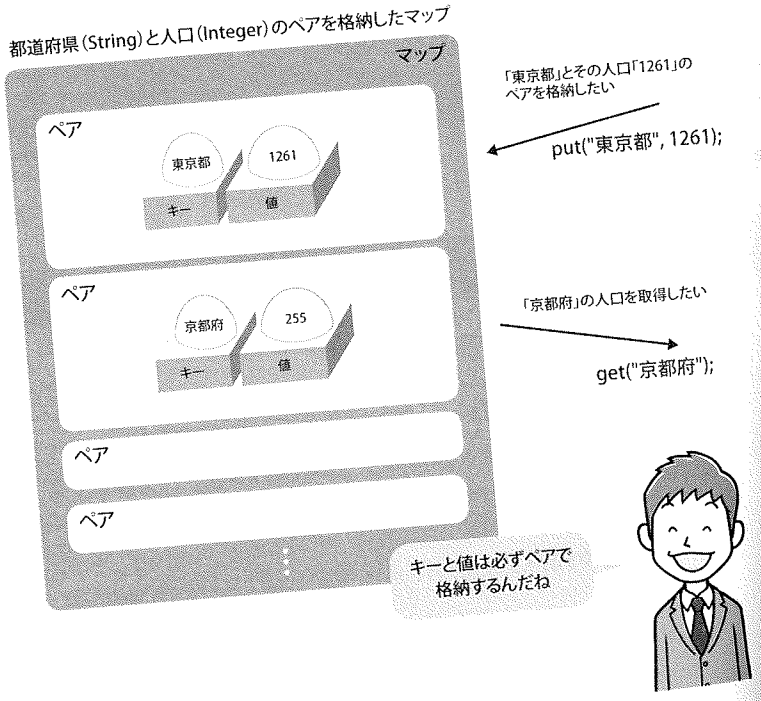

図3-17　マップの利用

3.5.2 HashMapクラスの利用

Mapインタフェースの実装の中でも特に多く利用されるのは、java.util.HashMapクラスです。HashMapのインスタンス化は次の構文で行います。

HashMapのインスタンス化

```
Map< キーの型 , 値の型 > マップ変数 =
new HashMap< キーの型 , 値の型 >();
```

また、HashMapは表3-5のようなメソッドを持っています。

戻り値	メソッド	意味
要素を格納する		
■	put(●,■)	マップに●と■のペアを格納する
要素を取り出す		
■	get(●)	キー値●に対応する値を取得(なければnull)
マップを調査する		
int	size()	格納されているペア数を数える
boolean	isEmpty()	要素数がゼロであるかを判定
boolean	containsKey(●)	指定データがキーに含まれているかを判定
boolean	containsValue(■)	指定データが値に含まれているかを判定
要素を削除する		
void	clear()	要素をすべて削除する
■	remove(●)	指定した内容の要素を削除する
その他		
Set<●>	keySet()	格納されているキーの一覧を返す

※●はキーを、■は値を示す

表3-5　HashMap <●, ■>が備えるメソッド

中でもget()とput()による読み書き、remove()による削除、size()によるペア数の取得などがよく利用されます。

なお、Mapでは値の重複は許されますが、**キーの重複は許されません**。よって、同じキーで異なる値をput()すると、値は上書きされてしまうので注意が必要で