3-10.コレクションフレームワーク

提出する 評定を受ける

■コレクションフレームワーク

はじめに

この章では、java.utilパッケージをimportして利用できる便利なクラスを学んで行きます。
java.utilパッケージには、配列操作、イベント・モデル、日付および時間など、
さまざまな、 **基本的** かつ **便利** かつ **現場** で使いまくりの機能があります。
このような「便利な機能を提供する仕組み」を総じて「フレームワーク」と読びます。
コレクション（複数要素の集まり） + フレームワーク（便利な機能を提供する仕組み） ということです！

この章では、その中でも、 **配列操作** に着目して学んでいきましょう！

Step1: 順序を持ったコレクション【List】

配列を操作する便利なクラスを紹介します！
ArrayListクラス と **LinkedListクラス** です。
配列の操作とは言いましたが、 少し前にやった配列 とは少々異なります。
配列は、最初にサイズを決めるため、後からサイズを拡張することはできません。
しかし、Listクラスは要素を追加した分だけ自動的にサイズが拡張される動的な配列を作ることができます。
また、Listクラスの様々なメソッドを使うことにより値を追加したり途中に挿入したり、削除することも可能です。

定義方法

```
// リストインターフェース名<型> インスタンス名 = new コンストラクタ名<型>();

// ArrayList
List<String> sampleList = new ArrayList<String>();

// LinkedList
List<String> sampleList = new LinkedList<String>();
```

補足

Java1.7以降では、 <> （ダイヤモンド演算子）を用いて以下のように記述することが可能です。

```
// ArrayListに<型>を指定しない
List<String> sampleList = new ArrayList<>();
```

・仮型引数による型の保証
左辺の型が明記されている場合は、 `ArrayList` が持つ **ジェネリクス** という機能が働きます。
ArrayListの型を仮の型引数とし、Listで指定された型を実際の型としてコンパイルします。
（※ジェネリクスについては紹介程度しておきます。

ArrayListクラス／LinkedListの使用例

?

```
//utilパッケージのArrayListクラスをimport
import java.util.ArrayList;

//utilパッケージのLinkedListクラスをimport
import java.util.LinkedList;

//utilパッケージのListクラスをimport
import java.util.List;

public class ArrayMain {

    public static void main(String args[]) {
        // LinkedListを使用する場合は以下のように書きます
        // List<String> sampleList = new LinkedList<String>();


        List<String> sampleList = new ArrayList<String>();

    // パターン1
        sampleList.add("ABC");
        sampleList.add("DEF");

        // パターン2
        sampleList = new ArrayList<String>() {
            {
                add("ABC");
                add("DEF");
            }
        };

        // リストに格納した全要素を順番に出力
        for (int i = 0; i < sampleList.size(); i++) {
            System.out.println(sampleList.get(i));
        }
    }
}
```

解説

生成したArrayListクラスのインスタンスを、Listインタフェース型の変数sampleListに保持しています。
Listインタフェースでは、要素の追加は **addメソッド** で行います。
また、パターン1と2は、意味合いとして同じになります。

値を取り出すときは、 **getメソッド** を使用します。

List系クラスの特徴と使い分け

クラス	長所	短所	使い所
ArrayList	特定の要素にアクセスするスピードが早い	要素を追加するスピードが遅い	配列内の要素に対してランダムなアクセスを必要とし、配列内の要素に対して挿入／削除の操作があまり必要ない場合
LinkedList	要素を追加したり削除するスピードが早い	特定の要素にアクセスするスピードが遅い	配列内の要素に対してランダムなアクセスを必要とせず、配列内の要素に対して挿入／削除の操作を頻繁に行う場合

ArrayListクラス は配列でリストを実装しており、「添え字による要素へのアクセス」が高速です。

要素の「追加」に関しても、新たな要素を末尾へ加えるだけなので問題なく行うことができます。
しかし、配列の途中の位置への要素の「挿入」や「削除」に関しては、 挿入・削除した位置以降の
全ての要素の位置を移動させるという処理を行う必要があるため、低速です。

LinkedListクラス はリスト構造を使用して実装しています。
このため、「添え字による要素へのアクセス」は、毎回先頭から順番に要素をたどっていきながら
目的の位置を探す（添え字の番号まで移動していく）必要があるため、低速です。

各クラスの特徴を踏まえて、用途に応じてクラスを選択することが Listインタフェースを使いこなす
ポイントとなります。

単純に「動的にサイズが拡張される配列」としてコレクションを使用したい `ArrayListクラス`
要素数が多くて、且つ要素の挿入・削除を頻繁に行うことが予想 `LinkedListクラス`

Step2: キーと値とのマッピングを保持するコレクション【Map】

Listと似たようなもので、連想配列を取り扱うものに **Map** というコレクションがあります。
Mapは二つの要素から成ります。

一つ目は **「キー」** と呼ばれる値です。
二つ目は **「値」** になります。

「キー」は簡単にいえば、値につける名前だと思ってもらえば大丈夫です。

値ひとつひとつに「キー」が存在しており、「キー」と「値」がペアになっているのがMapの特徴です。

Map系クラスの特徴

それぞれに特徴がありますので簡単に見ていきましょう。

クラス	概要
HashMap	ハッシュを使ったMapのデフォルト。 要素数によらない高速な検索ができます。 順序が保持されていないので目的の要素のキーを指定してそれに紐づく値を取得するのが早い
TreeMap	順序がキーの順番になっている。キーが数値の場合は小さい順に要素が保持されます。 キーが文字列の場合は、文字コードの順（辞書順、アルファベット順）に要素が保持される。
LinkedHashMap	キーの挿入順を保持する。 コンストラクタの引数の指定によって、 挿入順ではなくアクセス順を保持することもできます。デフォルトは挿入順です。 要素を追加（put）した順番に、そのままの順番で要素が保持されます。 または、コンストラクタの引数に順序付けモードを指定して、アクセス順（getした順番）に保持されるようにすることもできます。

定義方法

こちらもArrayList等と同様の記述となります。
（※ダイヤモンド演算子を用いての記述も可能です。

```
// マップインターフェース名<型> インスタンス名 = new コンストラクタ名<型>();

// HashMap（Mapの代わりに左辺を同じHashMapにしても可）
Map<String, Integer> hashMap = new HashMap<String, Integer>();

// TreeMap（Mapの代わりに左辺を同じTreeMapにしても可）
Map<String, Integer> treeMap = new TreeMap<String, Integer>();

// LinkedHashMap（Mapの代わりに左辺を同じLinkedHashMapにしても可）
Map<String, Integer> treeMap = new LinkedHashMap<String, Integer>();
```

Map系クラスの使用例

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

public class HashMapAddition {

    public static void main(String[] args) {

        // パターン1
        Map<String, String> sampleHashMap = new HashMap<String, String>();
        sampleHashMap.put("apple", "りんご");
        sampleHashMap.put("orange", "みかん");
        sampleHashMap.put("peach", "もも");

        // パターン2
        sampleHashMap = new HashMap<String, String>() {
            {
                put("apple", "りんご");
                put("orange", "みかん");
                put("peach", "もも");
            }
        };

        // キー値を指定して取得
        System.out.println(sampleHashMap.get("apple"));
        System.out.println(sampleHashMap.get("orange"));
        System.out.println(sampleHashMap.get("peach"));
        System.out.println(sampleHashMap.get("mango"));
    }
}
```

【出力結果】

```
りんご
みかん
もも
null
```

解説

生成したHashMapクラスのインスタンスをMapインタフェース型の変数sampleHashMapに保持しています。
Listインタフェースでは、要素の追加はaddメソッドを使用しましたが、
Mapインタフェースにはaddというメソッドは存在しません。
その代わりに **put** という名前のメソッドで要素の追加を行います。

putメソッドには引数を２つ指定します。
第１引数がキー、第２引数がキーに紐づく値です。
値を取り出すときは、 **getメソッド** を使用します。

補足

・ `get()` **の引数は** `Object`

`Object` を引数に取るため、あらゆる型やクラスのキーに対応可能です。
（※Objectについては後述します。
しかし、`HashMap` の **変数を定義した際のキー値と一致しないキー値が指定された場合** は
エラーにはならないものの、ワーニングが表示されます。

・ `Object` **クラス**

`Object` はクラスとして定義されており、全ての型やクラスの親です。
継承 については学びましたが、実は、すべてのクラスは **Objectクラスから成り立っています。**
そのため、`Object` で定義された型（クラス）の変数や、メソッドの引数は あらゆる値を格納することができます。

【サンプル: getに指定する引数】

```
Map<Integer, String> map = new HashMap<>();
map.put("値");

// 正しい指定
String strValue = map.get(1);

// 誤った指定
String strValue = map.get("1");
```

上記の `map.get("1");` は指定は可能ですが、ワーニングが表示され、
変数定義時にマッピングしたキー値の型と不一致であるため、取得結果がnullとなります。

Step3: 重複要素のないコレクション【Set】

この他に全く新しく登場してきたのが、Setインタフェースを実装するHashSetクラスやTreeSetクラスです。
値の格納方法は、ListやMapの格納方法の中間のような形式です。
Setは、Mapのようなキーと値の関連付けを行わず、Listのように値だけを格納しますが、
インデックスを使って格納されているわけではありません。
また、 **重複して同じ値を格納することもできません。**

クラス	要素の並び	備考
HashSet	2, 9, 8, 1, 5	ランダム
TreeSet	1, 2, 5, 8, 9	昇順に並ぶ
LinkedHashSet	5, 8, 1, 9, 2	追加した順に並ぶ

定義方法

こちらもList、Map等と同様の記述となります。
（※ダイヤモンド演算子を用いての記述も可能です。

```
// セットインターフェース名<型> インスタンス名 = new コンストラクタ名<型>();

// HashSet（Setの代わりに左辺を同じHashSetにしても可）
Set<String> hashSet = new HashSet<String>();

// TreeSet（Setの代わりに左辺を同じTreeSetにしても可）
Set<String> treeSet = new TreeSet<String>();

// LinkedHashSet（Setの代わりに左辺を同じLinkedHashSetにしても可）
Set<String> linkedHashSet = new LinkedHashSet<String>();
```

Set系クラスの例

`HashSet TreeSet LinkedHashSet` の例がありますので、
下記をコピペして、ご自身のPCで試してみてください！

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

public class Curriculum {

    public static void main(String[] args) {

        HashSet<String> sampleSet1 = new HashSet<String>();

        //適当に値を入れていく
        sampleSet1.add("A");
        sampleSet1.add("BA");
        sampleSet1.add("ABC");
        sampleSet1.add("ABCDE");
        sampleSet1.add("CD");

        System.out.println("HashSetなのでランダムで表示される");

        for (String s1 : sampleSet1) {
            System.out.println(s1);
        }

        TreeSet<String> sampleSet2 = new TreeSet<String>();

        //適当に値を入れていく
        sampleSet2.add("A");
        sampleSet2.add("D");
        sampleSet2.add("C");
        sampleSet2.add("B");
        System.out.println("TreeSetを使っているので、昇降順に表示される");

        for (String s2 : sampleSet2) {
            System.out.println(s2);
        }

        LinkedHashSet<String> sampleSet3 = new LinkedHashSet<String>();

        sampleSet3.add("ABCD");
        sampleSet3.add("ABCDEFGHIJK");
        sampleSet3.add("ABCDE");
        sampleSet3.add("ABCDEFGHIJ");
        System.out.println("LinkedHashSetなので追加した順に表示");

        for (String s3 : sampleSet3) {
            System.out.println(s3);
        }

        TreeSet<String> sampleSet4 = new TreeSet<String>();

        //適当に値を入れていく
        sampleSet4.add("A");
        sampleSet4.add("D");
        sampleSet4.add("D");
        sampleSet4.add("D");
        sampleSet4.add("B");
        sampleSet4.add("C");
        System.out.println("同じ値は格納できないから「D」は一つとして考えるよ");

        for (String s4 : sampleSet4) {
            System.out.println(s4);
        }
    }
}
```

【出力結果】

```
HashSetなのでランダムで表示される
A
ABCDE
CD
ABC
BA
TreeSetを使っているので、昇降順に表示される
A
B
C
D
LinkedHashSetなので追加した順に表示
ABCD
ABCDEFGHIJK
ABCDE
ABCDEFGHIJ
同じ値は格納できないから「D」は一つとして考えるよ
A
B
C
D
```

解説

このようにして、配列の並びを操ることができます。
では、いつ使用すれば良いのか。。。
例えば、スーパーのシステムを作っていて、
経営者から「うちの商品を安い順からリストアップしたい」とお願いされたとします。
「どうしたものか…項目を整理してくれるものはないだろうか…そうだ！ `TreeSet` を使ってみよう」
となるわけですね。

拡張for文

例題の中に見慣れないfor文あったかと思います。

```
for (String s : sampleSet) {
    System.out.println(s);
}
```

これを、 **拡張for文** と呼びます。
これまで学んできたfor文との差は、 **要素の指定** ができないということです。

```
for (int i = 0; i < sampleList.size(); i++) {
    System.out.println(sampleList.get(i));
}
```

上記のようにすれば、要素の指定ができるわけですが、
リストの **何番目が必要** という要件がなければ、拡張for文を活用することが多いです。
拡張for文は、記述内容がシンプルでコーディングもしやすく、可読性も優れているので現場では重宝されます。
特に、通常のfor文と違い記述ミスが軽減される点で、通常のfor文よりも需要は大きいです。

> **オマケ**

Mapの要素を取得する場合には、
いくつか特別な書き方がありますので簡単に紹介します。
（以下は **Map系クラスの使用例** の続きとします

```
        //Map.Entryをインポート
        import java.util.Map.Entry;
...
        System.out.println("----- entrySet() -----");

        // entrySetを使用してMapに含まれる要素の「キー／値」のセットを取得
        for (Entry<String, String> entry : sampleHashMap.entrySet()) {
            System.out.println("キー: " + entry.getKey() + "／値: " + entry.getValue());
        }

        System.out.println("\n----- keySet() -----");

        // keySetを使用してキー値を取得
        for (String key : sampleHashMap.keySet()) {
            System.out.println("キー: " + key);
        }
```

【出力結果】

```
----- entrySet() -----
キー: orange／値: みかん
キー: apple／値: りんご
キー: peach／値: もも

----- keySet() -----
キー: orange
キー: apple
キー: peach
```

・**entrySet()**
Setと、Mapの応用です。
Mapのエントリー（キーと値）を型としたSetコレクションになり、
戻り値は `Set<Map.Entry<K,V>>` と定義されています。

上記のように拡張for文と併せて記述することで、 `sampleHashMap` が保持する要素をリストのように扱うことが可能となります！
また、`keySet()` も同様ですが、`sampleHashMap` が保持する要素の **キー値のみ** を返しますので、受け皿はStringクラスとなります。

コレクションは機能が豊富で覚えることが多いですが、
現場では一番使用率が高いと言っても過言ではありません！
是非使いこなせるようになりましょう！

課題

インポートした3-10のフォルダの中にプロジェクトがありますので、
指示通りにコーディングして、ファイルを提出して下さい。

評定概要

学生から秘匿	No

参加者	75
提出	51
要評定	2