Java　学習テキスト

コレクションフレームワーク

目次

配列とは

Java の配列を思い出して下さい。

配列は沢山の変数を 1 つずつ定義しなくても、まとめて大量の変数を扱いたい時に、非常に手軽に作成・使用出来る便利機能でした。

Java で配列を作成するときは、以下の 2 通りの作り方をする事が多いでしょう。

```
int[] ary1 = new int[5];
int[] ary2 = { 1, 2, 3, 4, 5 };
```

しかし、Java の配列は、一度要素数を決めたら、後から要素数を変更することが出来ません。

つまり、以下のようなコードはコンパイルエラーとなってしまいます。

```
ary1[5] = 6;
```

ary1 配列は 5 つ作成した為、index は 0～4 の 5 つ分。index:5 は存在しないため、コンパイルエラーが発生する(6 個目の要素を作成してはくれない)。

後から要素数を変更することが出来ない理由として、まずはプログラミング言語全体に於ける配列の種類が 2 つあることを紹介します。

配列の大別

配列には、大きく分けて以下の 2 つの種類があります。

1. 固定長配列(静的配列)
2. 可変長配列(動的配列)

1 の固定長配列は、配列の作成時に要素数を決定する方法です。後から要素数の変更が行われない為、以下のようなメリットとデメリットがあります。

【固定長配列のメリット】
・決められた要素数を、他者が勝手に変更出来ないようにする。
・コンピュータがプログラムを実行する時、要素数が決まっているので処理が速くなる。

【固定長配列のデメリット】

・要素数の変更が出来ないので、配列を作成する時に要素数を決めておく(調べる)必要がある。

2 の可変長配列は、いつでも配列の要素数を変更する事が出来る方法です。後から要素数の変更ができる為、以下のようなメリットとデメリットがあります。

【可変長配列のメリット】
・いつでもどこでも要素数が変更できるので、必要な要素数を調べなくてもいい。

【可変長配列のデメリット】
・他者が要素数を変更してしまい、プログラムに不具合が出る事がある。
・コンピュータがプログラムを実行する時、要素が変更されると保存場所(メモリ内)を都度確保する必要がある為、処理が遅くなる。

それぞれにメリットとデメリットがあり、用途やプログラムの状況を把握してどちらを使用するか決定をします。
プログラミング言語によっては両方使用出来るものもありますが、Java は固定長配列(静的配列)しか用意されていませんので、可変長配列(動的配列)は存在しません。

COLUMN

プログラムにおける「他者」とは誰でしょうか。
この言葉には、2 つの意味が込められています。
1 つ目は単純に「他の人」です。ソースコードの前半を自分が記述し、後半を別の人が書いた時や、自分の作ったソースコードを他の人が編集する場合など、後からソースコードを記述・編集した人は、固定長配列であれば存在しない要素にアクセスした場合、実行時エラーで直ぐにミスに気づけます。
2 つ目の意味は「外部」です。つまり、自分の作ったプログラムを呼び出して利用するプログラムを作成している「他の人」です。
通常カプセル化によって勝手に変数にアクセスできないようにしますが、配列を引数で受取り、要素数を固定したい場合などは、固定長配列を使用します。

コレクションフレームワークとは

前項で、Java には可変長配列は存在しない。と断言しました。
しかし、プログラムの状況により、可変長配列を必要とするシーンは多々あります。
そういった時の為に、Java では「可変長配列っぽい何か」を別途用意してくれています。
この「可変長配列っぽい何か」は**コレクションフレームワーク**と呼ばれます。
コレクションフレームワークを強引に言ってしまえば、「Java には可変長配列が無いから、可変長配列みたいに使えるプログラムを作成した」という事です。

コレクションという言葉には「収蔵」などの意味があります。
その為、プログラミングの分野では、データなどをまとめて格納するための構造体をコレクションと呼びます(似たような言葉に「コンテナ」もあります)。
広義の意味では配列もコレクションの一つですが、Java では「コレクション」は「コレクションフレームワークの一部」を指しています。
Java のコレクションフレームワークでは、幾つかのコレクション(全てデータなどを一纏めにして扱いますが、要素の管理方法や取得方法が異なります)を用意しています。
それら一つ一つのコレクションに対し、コレクションフレームワークでは様々なクラスやインターフェースが用意されています。
その全てを網羅することは難しいので、このテキストではよく利用されるクラスとメソッドについて解説を行っていきます。

リスト

Java で最も利用される「可変長配列っぽい何か」の代表格が java.util.ArrayList<E>です。

ArrayList は List インターフェースを実装していて、「**リスト**」と呼ばれるコレクションを扱います。

言うなれば、この「リスト」こそが代表的な「可変長配列っぽい何か」です。

<E>など今まで見たことのない表記もありますが、下記のコードで使い方を見てみましょう。

尚、コレクションフレームワークは java.util パッケージに格納されているので、忘れずに import するようにしましょう。

【Sample1_1.java】

```
import java.util.ArrayList;

public class Sample1_1 {
    public static void main(String[] args) {
        //ArrayList の定義
        ArrayList<Integer> al = new ArrayList<Integer>();

        //ArrayList に要素を追加
        al.add(10);
        al.add(20);

        //ArrayList から要素を取得
        int a = al.get(0);

        System.out.println(a);
    }
}
```

実行結果:

```
10
```

ArrayList の宣言

宣言部分を分解して見ていきましょう。

```
//ArrayList の定義
ArrayList<Integer> al = new ArrayList<Integer>();
```

リストを使うときは幾つかのルールがあり、配列とは大きく異なる点があります。
下記のポイントを押さえておきましょう。

1.<E>に要素の型を記述する。

型の宣言を見ると、ArrayList<Integer>となっています。この<Integer>の部分を「**ジェネリクス(総称型)**」と言い、リストに格納する要素の型を宣言します。

使用時の注意点としては、ジェネリクスは「基本型」が使用できない。というルールです。

コードを ArrayList<int>などに変更すると、コンパイルエラーが発生します。

その為、リストで基本型要素を持った配列のように使いたい時は、ラッパークラスを使用します。

ラッパークラスとは、基本型をクラス型として定義しているクラスの事です。

以下の対応表を確認して、使う時の参考にして下さい。

基本型	ラッパークラス
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

※基本的には先頭文字を大文字にする。
int と char だけ表記が変わるので注意。

2.インスタンス化

ArrayList はクラスとして提供されているので、使用する時は new ArrayList<Integer>でインスタンス化します。

実際に各要素を代入するには、メソッドを使用して行います。

配列のように初期値を代入して宣言することは出来ませんので、注意してください。

また、型の宣言で<>内に指定した型と、インスタンス化で指定する型が一致する必要があります。

今回は ArrayList<Integer>で型の宣言を行った為、インスタンス化も new

ArrayList<Integer>()となります。

リストの要素を増やす

続いて要素の追加を見ていきましょう。

```
//ArrayList に要素を追加
al.add(10);
al.add(20);
```

要素を増やすときは、add メソッドを呼び出して、引数に値を渡します。

今回の例では 2 回 add メソッドを呼び出している為、2 要素分のリストが作成されています。

index	値
0	10
1	20

このように、add メソッドは後から好きなだけ要素の追加を行ってくれます。

リストから要素を取得する

要素を取得する場合は、get メソッドを使用します。

```
//ArrayList から要素を取得
int a = al.get(0);
```

get メソッドの引数には index を指定します。

配列と同様に index は 0 から始まるので、今回は最初の要素である「10」を取得しています。

コードをよく読むと、int 型の変数に値を代入しています。しかし、リストの要素は型を Integer に指定しています。

このように、ラッパークラスは対応する基本型へはキャストなしでそのまま代入する事が可能です。

当然、以下のように直接 println()メソッドに値を渡すことも可能です。

```
System.out.println(al.get(0));
```

尚、存在しない index を指定すると、配列と同様に ArrayIndexOutOfBoundsException という実行時エラーがスローされます。

COLUMN

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

上記の変数宣言とインスタンス化を見て、「長いなあ…」と思われた方も少なくないと思います。

実は、Java には「型推論」という機能があります。

これはどこでも使えるわけではありませんが、ジェネリクスは型推論に対応しています。

どのように使うかというと、型の宣言で指定した要素の型は、インスタンス化では省略出来るという仕組みです。

つまり、以下のように記述すると、JVM が型を読み取って、インスタンス化するクラスの要素型を推論して設定してくれます。

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

但し、型の宣言時は省略できませんので注意しましょう。

このような書き方は、非公式ですが<>の見た目から、ダイヤモンド演算子と呼ばれています。

練習問題

問1.　Practice1.java を作成し、以下の設問通りにコーディングしてください。

1.ランダムな数を 5～10 の範囲で生成して、その数分だけリストを作成してください。

2.要素は整数で 0 から順番に格納してください。

3.全ての要素をコンソールに出力してください。

実行例(実行するたびに 5～9 で変化します):

```
0 1 2 3 4 5 6 7 8
```

リストの便利なメソッド

リストには、配列と同様に要素数を調べたり、要素を検索して index 値を取得したりする事が出来ます。

ArrayList で良く使うメソッドは以下の通りです。

戻り値	メソッド	説明
void	add(int index, E element)	リストの index で指定された位置に要素を挿入します。尚、その index に既に要素がある場合は、以降の要素を次の index へ移動させます。
boolean	add(E e)	リストの最後に要素を追加します。
E	set(int index, E element)	リストの index で指定された要素を指定された要素で置き換えます。
E	get(int index)	index で指定された位置にある要素を返します。
E	remove(int index)	リストの index で指定された要素を削除します。
int	indexOf(Object o)	指定された要素がリスト内に検出された、最初の位置(index)を返します。尚、要素が検出されなかった場合は、-1 を返します。
boolean	contains(Object o)	指定された要素がこのリストに含まれている場合に true を返します。
int	size()	このリスト内にある要素の数を返します。

これらのメソッドを利用すると、リストの様々な操作を自由に行うことが出来ます。

サンプルプログラムを作成して動き方を確認してみましょう。

【Sample1_2.java】

```
import java.util.*;

public class Sample1_2 {
    public static void main(String[] args) {
        //ArrayList のインスタンス化
        List<Integer> list = new ArrayList<>();

        //List に要素を追加
        list.add(0); list.add(1); print("要素を追加:", list);

        //List の index:1 に要素を挿入
```

```
        list.add(1, 2); print("要素を挿入:", list);

        //list の index:1 の要素を削除
        list.remove(1); print("要素を削除:", list);

        //list の index:0 の要素を変更
        list.set(0, 10); print("要素を変更:", list);

        //list の要素を検索
        int index = list.indexOf(10);
        System.out.println("要素を検索:" + index);

        //list に要素が含まれるか確認
        boolean chk = list.contains(10);
        System.out.println("要素の確認:" + chk);

        //list の要素数
        int indexNum = list.size();
        System.out.println("要 素 数:" + indexNum);
    }

    //出力用メソッド
    static void print(String text, List<Integer> list) {
        System.out.print(text);
        //コレクションは拡張 for も使用可能
        for(int val : list) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}
```

実行結果:

```
要素を追加:0 1
要素を挿入:0 2 1
要素を削除:0 1
要素を変更:10 1
```

```
要素を検索:0
要素の確認:true
要　素　数:2
```

List 型の ArrayList

Sample1_2.java の変数宣言を確認してみましょう。

```
List<Integer> list = new ArrayList<>();
```

型は List インターフェース型にし、インスタンスを ArrayList にしています。

この書き方は、java でよく使用される記述方法です。

インターフェース型にクラスをインスタンス化する使い方ですが、リストでは特にこの書き方が推奨されています。

理由としては以下の 2 点です。

1. 基本的なリスト操作は全て List インターフェースで定義されている為、List 型で操作が足りなくなる事が少ない
2. ArrayList 以外の List インターフェースを実装したクラス(LinkedList クラス、Vector クラスなど)でもインスタンス化が可能

どちらも基本的にはメソッドの引数や戻り値の部分で大きな効力を発揮します。

メソッド内では ArrayList 型にしても大きな問題は起こりにくいですが、特別な理由(ArrayList クラスにしか存在しないメソッドを利用したい)がなければ、慣習的に List 型にしておく事をお勧めします。

練習問題

問1　Practice2.java を作成し、以下の設問通りにコーディングしてください。

1.要素を 1 文字のリストを作成し、それぞれ「あ」「い」「う」「え」「お」の文字を格納して下さい。

2.リストから、3 文字目の「う」を削除してください。

3.リストから、4 文字目に「が」を挿入してください。

4.リストを検索し、「あ」があれば 1 文字目を削除してください。(if 文を使います)

5.リストから「い」の文字を検索し、その文字があれば削除してください。(if 文を使います)

6.リストの内容を全て一行でコンソールに出力し、全体の文字数を一緒に出力してください。

実行例:

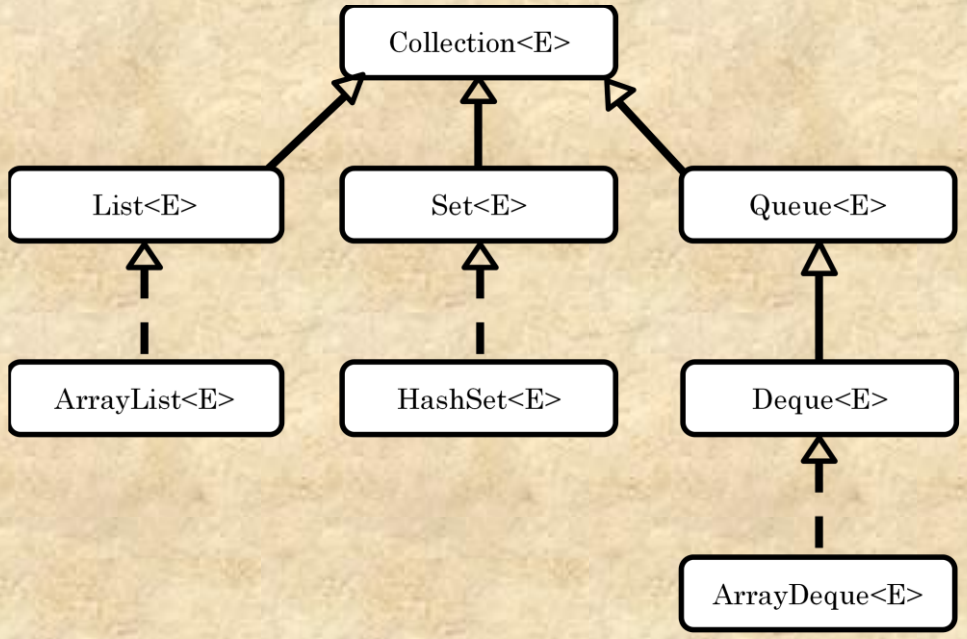
```
え が お 文字数:3
```

COLUMN

Java はオブジェクト指向言語ですので、様々なクラスやインターフェースを利用しています。

Java でコレクションフレームワークを「コレクション」と表現するのには、コレクションフレームワークの頂点のインターフェース名から来ています。

厳密には Object クラスや他クラス・インターフェースの継承などもありますが、コレクションフレームワークをシンプルにクラス図で表現してみましょう。



Collection インターフェースを継承した、List インターフェース、Set インターフェース、Queue インターフェースがそれぞれ「可変長配列っぽい何か」のひな型となります(使い方がそれぞれ異なります)。

それぞれよく利用されるインスタンス化クラスは ArrayList<E>、HashSet<E>、ArrayDeque<E>となります。

配列とリストの相互活用

ここまでリストを触ってみて、配列とは違う使い勝手の良さを感じた事と思います。

しかし､膨大なJavaのライブラリの中には､メソッドの引数として配列を受け取ったり、戻り値として配列を返したりと、配列を利用したプログラムも数多くあります。

また、過去に作成したプログラムを再利用した際に、配列が使用されている事も多々あります。

そういった場合は、配列とリストを相互に変換する事が可能です。

但し、いくつかの制約や制限が付くことがある為、細かいルールをしっかりと把握しておきましょう。

配列とリストの変換

配列とリストを相互に変換するには、それぞれ異なるクラスを使用します。

1. リストを配列に変換する
   java.util.List<E>.toArray()
2. 配列をリストに変換する
   java.util.Arrays.asList(T... a)

List インターフェースには、リストを配列に変換するメソッドが用意されています。

また、Arrays クラスには、リストを配列に変換したり、配列に様々な操作を行ったりするメソッドが提供されています。

Arrays クラスについての詳細は後述しますが、全てのメソッドが static で作成されている為、インスタンス化をせずに「Arrays.メソッド名()」で直接利用する事が可能です。

クラスを調べる

List<E>や Arrays クラスの使い方を確認する前に、変数の中身が、今現在どのクラスなのかを確認してみましょう。

クラスを確認したい時は Object.getClass メソッドを使用します。

Object クラスは全てのクラスのスーパークラスですので、クラス型であればどのクラスからも呼び出すことが可能です。

サンプルプログラムを作成して動き方を見てみましょう。

【Sample1_3.java】

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

```
public class Sample1_3 {
    public static void main(String[] args) {

        //配列を用意する
        int[] ary = { 1, 2, 3};

        //リストを用意する
        List<Integer> list = new ArrayList<>();

        System.out.println("ary 変数:" + ary.getClass());
        System.out.println("list 変数:" + list.getClass());
    }
}
```

実行例:

```
ary 変数:class [I
list 変数:class java.util.ArrayList
```

配列は少し特殊な出力の仕方をします。

「[」が「配列」という意味を持ち、「I」は要素の型を表します。I であった場合は int を指しますので、これは int 型の配列である。という意味です。

ArrayList は見た通り、パッケージ名を含めた FQCN で表記されます。

クラス型であればクラス名を確認する事が出来ますので、条件式に組み合わせるといった利用方法もあります。

実際に変換をしてみよう

クラスの確認方法が分かった所で、実際に変換を行ってみましょう。

今回はリストを作成し、配列に変換し、またリストに戻します。

サンプルプログラムを作成し、実行確認をしてみましょう。

また、今回は要素 0 で行っていますが、それぞれが持っている要素もコピーもされます。

【Sample1_4.java】

```
import java.util.*;
public class Sample1_4 {
    public static void main(String[] args) {
        //リストを作成する
```

```
        List<Integer> list = new ArrayList<>();
        list.add(1); list.add(2);
        System.out.println("list 変数のクラス:" + list.getClass());
        System.out.println("list 変数の中身 :" + list);

        //リストを配列に変換
        Integer[] list2Ary = list.toArray(new Integer[list.size()]);
        System.out.println("list2Ary 変数のクラス:" + list2Ary.getClass());
        System.out.println(
            "list2Ary 変数の中身 :" + Arrays.toString(list2Ary));

        //配列をリストに変換
        List<Integer> ary2List = Arrays.asList(list2Ary);
        System.out.println("ary2List 変数のクラス:" + ary2List.getClass());
        System.out.println("ary2List 変数の中身 :" + ary2List);
    }
}
```

実行例:

```
list 変数のクラス:class java.util.ArrayList
list 変数の中身 :[1, 2]
list2Ary 変数のクラス:class [Ljava.lang.Integer;
list2Ary 変数の中身 :[1, 2]
ary2List 変数のクラス:class java.util.Arrays$ArrayList
ary2List 変数の中身 :[1, 2]
```

リストの作成

以下のコードは前項でも解説済みの部分です。

```
//リストを作成する
List<Integer> list = new ArrayList<>();
list.add(1); list.add(2);
System.out.println("list 変数のクラス:" + list.getClass());
System.out.println("list 変数の中身 :" + list);
```

特筆すべき点としては println メソッドの中に、リスト変数をそのまま渡している所です。

インスタンス化された ArrayList は Object クラスの toString メソッドをオーバーライドしている為、このように変数名だけを指定するとリストの中身が一覧表記([1, 2])で出力されます。

実行結果を見ると以下の通りです。

実行結果:

```
list 変数のクラス:class java.util.ArrayList
list 変数の中身 :[1, 2]
```

リストを配列へ変換

リストを配列に変更する時は、リスト変数の toArray メソッドを使用します。

```
//リストを配列に変換
Integer[] list2Ary = list.toArray(new Integer[list.size()]);
System.out.println("list2Ary 変数のクラス:" + list2Ary.getClass());
System.out.println(
    "list2Ary 変数の中身 :" + Arrays.toString(list2Ary));
```

リストを配列に変換する式を、1 つずつ分解して動き方を見ていきましょう。

まず、**リストと同じ型**の配列変数を用意します。

`Integer[] list2Ary`

次に、list.toArray メソッドを呼び出します。

`list.toArray()`

最後にメソッド内の内容を分解してみましょう。

`new Integer[list.size()]`

少し複雑に見えますが、通常の配列の宣言を Integer 型で行うと以下の通りとなります。

```
Integer[] ary = new Intger[2];
```

いつもの配列変数の宣言と同じである事が分かります。

このインスタンス化の部分が、今回の toArray メソッド内で行っている処理になります。

配列の数を決める[]内は、list.size()でリストの要素数を取り出しています。

また、toArray メソッドは、リスト内の要素を配列に格納してくれます。

これらの処理を通常の式で表現すると以下のようになります。

```
Integer[] list2Ary = new Integer[2];
list2Ary[0] = 1; list2Ary[1] = 2;
```

一見複雑に見える処理であっても、分解して理解すれば今までやってきた内容と何ら変わりありません。

また、リストのように「[1, 2]」という形式で配列の中身を一覧表記したい時は、Arrays クラスを利用します。
Arrays.toString メソッドに配列を渡せば、リストと同じ形式での出力が可能という事を覚えておいてください。
配列.toString メソッドでは違う値が表示される為、注意が必要です。

最後に、実行結果を見ると、配列のクラス名が[Ljava.lang.Integer;となっています。

```
list2Ary 変数のクラス:class [Ljava.lang.Integer;
list2Ary 変数の中身 :[1, 2]
```

「[」は配列という意味でした。「L」は要素がクラス型である事を表し、その後にどのクラスかが表記されています。

配列をリストへ変換

最後に、配列をリストに変換します。

```
//配列をリストに変換
List<Integer> ary2List = Arrays.asList(list2Ary);
System.out.println("ary2List 変数のクラス:" + ary2List.getClass());
System.out.println("ary2List 変数の中身 :" + ary2List);
```

配列をリストに変換するには、先ほども出てきた Arrays クラスを利用します。
同じ要素型を持つリストに対し、Arrays.asList メソッドに配列を渡せば、同じ要素を持ったリストに変換する事が可能です。
但し、一点だけ注意しなければならないルールがあります。
実行例を見てみましょう。

```
ary2List 変数のクラス:class java.util.Arrays$ArrayList
ary2List 変数の中身 :[1, 2]
```

クラスを確認すると java.util.Arrays$ArrayList となっています。
この Arrays$ArrayList となっている場合は、リストの要素が変更できない事を表します。

つまり、配列から Arrays.asList を使用した場合は、add メソッドによる要素の追加や変更、remove メソッドによる要素の削除等、変更が出来ません。

これは、ArrayList クラスが「可変長のリスト」を作成するクラスなのに対して、Arrays$ArrayList クラスは「固定長のリスト」を作成する**別のクラス**だから起こる問題です。

元が配列で、リストとして扱いたい場合などには Arrays.asList メソッドはとても便利ですが、固定長のリストになるという事は忘れずに覚えておきましょう。

配列を可変長リストに変換する

前項では配列をリストに変換すると固定長リストになる。という内容でした。

では、配列を可変長リストに変換できないのかというと、そんな事はありません。

Arrays.asList メソッドを使用して、可変長リストを作成することも可能です。

以下のサンプルコードでその動き方を見てみましょう。

【Sample1_5.java】

```
import java.util.*;
public class Sample1_5 {
    public static void main(String[] args) {
        //配列を作成
        Integer[] ary = {1, 2};
        System.out.println("クラス名:" + ary.getClass());
        System.out.println("中　身:" + Arrays.toString(ary));

        //配列をリストに変換
        List<Integer> list = new ArrayList<>(Arrays.asList(ary));
        System.out.println("クラス名:" + list.getClass());
        //要素を追加
        list.add(3);
        //中身を表示
        System.out.println("中　身:" + list);
    }
}
```

実行例:

```
クラス名:class [Ljava.lang.Integer;
中　身:[1, 2]
クラス名:class java.util.ArrayList
```

```
中　身:[1, 2, 3]
```

ArrayList のコンストラクタを利用する

特筆すべき点は以下のコードです。

```
List<Integer> list = new ArrayList<>(Arrays.asList(ary));
```

ArrayList のインスタンス化時に、コンストラクタへ固定長リストを渡しています。

実は、ArrayList のコンストラクタには、リストを指定すると、要素を全てコピーして新しいリストを生成する。というプログラムが記述されています。

その為、実行例を見ると「Arrays$ArrayList」ではなく「ArrayList」になっている事が分かります。

このようにすれば、配列を可変長リストに変換する事が可能です。

但し、このやり方は新しいインスタンスを生成する数が増える為、処理時間や一時的にメモリ使用量が増加するというデメリットもあります。

要素の変更が不要である場合、配列→固定長リストで 2 つのインスタンスとコピー処理で済みますが、要素の変更が必要である場合は、配列→固定長リスト→可変長リストと 3 つのインスタンスとコピー処理が必要です。

配列をリストに変換する時は、後から要素の追加が必要かどうかをしっかりと考えてから行いましょう。

練習問題

問1　Practice3.java を作成し、以下の設問通りにコーディングしてください。

1.static メソッド doIt(List<String> list)を定義してください。引数で受け取った「list」変数のクラス名と、リストの要素を全て一覧表示します。

2.String 型の配列を作成し、それぞれ「えい」「えい」「おー」と 3 つの要素を格納して下さい。

3.String 型のリストを作成し、「右」「左」「右」と 3 つの要素を追加してください。

4.「1.」で作成した doIt メソッドを利用して、「2.」の配列と「3.」のリストを引数に渡して画面表示を行ってください。

実行例:

```
クラス名は:class java.util.Arrays$ArrayList
中　身は:[えい, えい, おー]
クラス名は:class java.util.ArrayList
中　身は:[右, 左, 右]
```

Arrays クラス

これまで触ってきた Arrays クラスには、配列を操作する様々な便利なメソッドを数多く提供しています。

この項目では、Arrays クラスの便利なメソッドを学んでみましょう。

Arrays クラスの便利なメソッド

Arrays クラスは配列を操作するクラスです。

メソッドは全て static で定義されており、インスタンス化しなくてもいつでも使えるという特徴があります。

Arrays クラスのよく利用されるメソッドは下記の通りです。

戻り値	メソッド	説明
List<T>	asList(T... a)	指定された配列に連動する固定サイズのリストを返します。
基本型[]	copyOfRange(*基本型*[] original, int from, int to)	指定された配列の指定された範囲を新しい配列にコピーします。
void	sort(*型*[] a)	指定された配列を数値の昇順でソートします。
void	sort(*型*[] a, int fromIndex, int toIndex)	指定された範囲の配列を昇順にソートします。
String	toString(*型*[] a)	指定された配列の文字列表現を返します。

※「基本型」や「型」となっているのは、対応した型全てにオーバーライドされている事を表します。

これらのメソッドを利用すると、配列の操作が非常に便利になります。

サンプルプログラムを作成して動き方を確認してみましょう。

【Sample1_6.java】

```
import java.util.*;

public class Sample1_6 {
    public static void main(String[] args) {
        //配列を用意する
        int[] ary = { 5, 4, 3, 2, 1 };
```

```
        System.out.println("大本の配列:" + Arrays.toString(ary));

        //昇順にソートする
        Arrays.sort(ary);
        System.out.println("並び変え後:" + Arrays.toString(ary));

        //配列の index2～4 をコピーする
        int[] aryCp = Arrays.copyOfRange(ary, 2, 5);
        System.out.println("index2～4 のコピー:" + Arrays.toString(aryCp));
    }
}
```

実行結果:

```
大本の配列:[5, 4, 3, 2, 1]
並び変え後:[1, 2, 3, 4, 5]
index2～4 のコピー:[3, 4, 5]
```

sort

配列の並び替えは、Arrays.sort メソッドに配列を引数で指定するだけです。
引数で指定された配列が直接並び替えされます。

```
//昇順にソートする
Arrays.sort(ary);
System.out.println("並び変え後:" + Arrays.toString(ary));
```

原則として Arrays.sort メソッドで並び替え出来るのは昇順だけです。また、配列の要素がクラス型の場合は、その要素の自然順序付けに従って昇順にソートされます。

COLUMN

自然順序付けとは、アルファベット順を基本として、複数桁の数字をまとめて一つの数値として扱って順序付けられるような照合規則を指します。

しかし、Java では、**並び替え時に指定される値を、クラス毎に変更する事が可能です。**

これはクラス設計時に予め決めておく物で、クラスによって値は異なります。

その為、ソート結果はクラスによって変動します(Integer[]などのラッパークラスであれば、内部の整数値で順序付けしているので、int[]と同じ結果となります)。

copyOfRange

配列のコピーを作成する時は、配列そのものを丸々コピーする方法と、配列の一部をコピーする方法があります。

配列を丸々コピーするという事はあまりありませんので、このテキストでは配列の一部をコピーする方法を取り扱います。

```
//配列の index2～4 をコピーする
int[] aryCp = Arrays.copyOfRange(ary, 2, 5);
System.out.println("index2～4 のコピー:" + Arrays.toString(aryCp));
```

Java の配列をコピーする時に特に注意が必要なのは、index の範囲の指定方法です。

```
Arrays.copyOfRange(ary, 2, 5)
```

1 つ目の引数は、コピー元となる配列を指定します。

2 つ目の引数は、開始位置を**「index」で指定します**。

3 つ目の引数は、終了位置を**「個数」で指定します**。

つまり、今回の指定方法は「ary 配列」の「index:2」から「要素数 5 個目」までをコピーしてね。という意味になります。

COLUMN

配列のコピー操作は、Arrays クラスだけでなく System クラスを使用する方法もあります。

その場合は、以下のように記述します。

```
System.arraycopy(コピー元の配列，コピー元配列の開始位置，コピー先の配列，コピー先配列の開始位置，コピー元配列の要素数)
```

System.arrycopy メソッドを使用する場合は、貼り付ける配列の index も指定ができます。

そういった利便性から、System.arraycopy もよく使用されます。

練習問題

問1　Practice4.java を作成し、以下の設問通りにコーディングしてください。

1.9～0 までの 10 要素を持つ配列を作成し、7～4 の要素を持つ配列をコピーで作

成してください。

2.コピーした配列を昇順にして、一覧を画面に表示してください。

実行例:

```
[4, 5, 6, 7]
```

シャローコピーとディープコピー

Java のようなオブジェクト指向言語の配列のコピーは 2 種類に分類されています。
それが、シャローコピー(shallow copy)とディープコピー(deep copy)です。
基本型の配列をコピーする時は意識しませんが、Java では様々なクラスが用意されており、それらのクラスを配列やリストとして扱うことも数多くあります。
その時に意識すべき注意点が、この 2 つのコピーの違いです。
この違いを意識して、間違えないようにする事はとても大切です。
なぜなら、「見た目は一緒」だけど「内部が異なる」為、実際にプログラムを実行した場合に全く違う結果になるからです。
プログラムのバグの元になりやすく、且つ、意識していなければ気付き難い内容になる為、しっかりと内容を理解しておきましょう。

シャローコピー

日本語では「浅いコピー」と表現されます。
クラス型を使用した場合は「参照値」をコピーします。

それでは、シャローコピーのやり方と動きを、サンプルプログラムを作成して確認してみましょう。

Sample1_7.java

```
//テスト用のクラスを作成
class Shallow {
    //文字列を保持するフィールド
    private String value;
    //セッター
    public void setValue(String str) { this.value = str; }
    //ゲッター
    public String getValue(){ return this.value; }
}

public class Sample1_7 {
    public static void main(String[] args) {
        //Shallow クラスをインスタンス化
        Shallow shallow1 = new Shallow();
```

```
        //shallow1 に"1"をセット
        shallow1.setValue("1");
        //shallow2 に shallow1 をシャローコピー
        Shallow shallow2 = shallow1;
        //shallow2 に"2"をセット
        shallow2.setValue("2");
        //shallow1 と shallow2 の値を出力
        System.out.println("shallow1:" + shallow1.getValue() + "|shallow2
:" + shallow2.getValue());
    }
}
```

実行結果:

```
shallow1:2|shallow2:2
```

シャローコピーは参照値のコピー

実行結果を見ると、shallow2 変数の値だけ変更したのに、shallow1 変数の出力も同様に変更されています。

これは、「参照値のコピー」が行われた為起こる現象です。

Java がプログラムを実行している時のメモリ内を確認してみましょう。

まず、基本型やソースコードに宣言した変数は、**スタック領域**に保存されます。

そして、インスタンス化したクラスは全て**ヒープ領域**に保存されます。

ヒープ領域には、特定のインスタンスを指定できるようにアドレスが割り振られていて、スタック領域の変数にはアドレス(参照値)が保存されます。

今回の場合は、以下のような関係になっています。(アドレスは長い番号になるので、省略して表記します)

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
クラス型変数 cls1【#参照値】 クラス型変数 cls2【#参照値】 ※基本型は値が直接入る int 型変数 i 【10】	001	クラス名 クラス内変数 クラス内メソッド

では、実際のコード実行に合わせて、上記のメモリ管理表に値を差し込んでいきましょう。

まず、以下のコードを実行した段階でのメモリ内の情報です。

```
//Shallow クラスをインスタンス化
Shallow shallow1 = new Shallow();
```

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
Shallow 型変数 shallow1【#001】	001	Shallow String 型変数 value【""】 void setValue(String str) String getValue()

スタック領域内に Shallow 型変数 shallow1 が作成され、「new Shallow()」を実行すると、ヒープ領域に Shallow クラスをインスタンス化し、=でその参照値を shallow1 変数に代入します。

次のコードを実行してみましょう。

```
//shallow1 に"1"をセット
shallow1.setValue("1");
```

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
Shallow 型変数 shallow1【#001】	001	Shallow String 型変数 value【"1"】 void setValue(String str) String getValue()

shallow1.setValue("1")を分解すると、「参照値#001」番の「setValue()」メソッドに、引数「"1"」を渡す。となります。

こうする事で、ヒープ領域 001 番の Shallow インスタンスの変数 value に”1”が代入されました。

次に、シャローコピーを行ったコードを見てみましょう。

```
//shallow2 に shallow1 をシャローコピー
Shallow shallow2 = shallow1;
```

クラス型変数を別のクラス型変数に直接代入を行っています。
この場合、shallow1 変数も shallow2 変数もスタック領域に保存されているデータです。
その為、この代入式は「shallow1 の中身」を「shallow2 に上書き」する動き方をします。
スタック領域内で「何の値」がコピーされたかを確認してみましょう。

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
Shallow 型変数 shallow1【#001】 Shallow 型変数 shallow2【#001】	001	Shallow String 型変数 value【”1”】 void setValue(String str) String getValue()

見ての通り、shallow1 変数に入っていた「参照値」が、shallow2 に上書きコピーされました。

この状態で以下のコードを実行するとどうなるでしょうか。

```
//shallow2 に"2"をセット
shallow2.setValue("2");
```

shallow2 変数の中身は、「参照値#001」です。つまり、「参照値#001」番の「setValue()」メソッドに、引数「”2”」を渡す。となります。
その結果、メモリ内は以下のようになります。

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
Shallow 型変数 shallow1【#001】 Shallow 型変数 shallow2【#001】	001	Shallow String 型変数 value【”2”】 void setValue(String str) String getValue()

参照値#001 番の Shallow インスタンスの value 変数が「”2”」に上書きされました。

shallow1 変数も、shallow2 変数も同じインスタンスを対象としている為、このような事態が起きます。

このように「参照値」をコピーする方式を「シャローコピー」と言います。

ディープコピー

日本語では「深いコピー」と表現されます。

クラス型を使用した場合は「インスタンスそのもの」をコピーします。

それでは、ディープコピーのやり方と動きを、サンプルプログラムを作成して確認してみましょう。

Sample1_8.java

```
//テスト用のクラスを作成
class Deep {
    //文字列を保持するフィールド
    private String value;
    //セッター
    public void setValue(String str) { this.value = str; }
    //ゲッター
    public String getValue(){ return this.value; }
}
public class Sample1_8 {
    public static void main(String[] args) {
        //Deep クラスをインスタンス化
        Deep deep1 = new Deep();
        //deep1 に"1"をセット
```

```
        deep1.setValue("1");
        //deep2 に deep1 をディープコピー
        Deep deep2 = new Deep();
        deep2.setValue(deep1.getValue());
        //deep2 に"2"をセット
        deep2.setValue("2");
        //deep1 と deep2 の値を出力
        System.out.println("deep1:" + deep1.getValue() + "|deep2:" + deep
2.getValue());
    }
}
```

実行結果:

```
deep1:1|deep2:2
```

ディープコピーはインスタンスのコピー

実行結果を見ると、deep1 変数と deep2 変数の出力が異なるものになっています。

また、コードを見ると、インスタンスを新規作成してから、インスタンス内の変数に対して値をコピーしているのが分かります。

このように、**別インスタンスを新規作成し、内部の値のみを上書きコピーする**方法をディープコピーといいます。

細かく Java のメモリ内の動きも見ていきましょう。

```
//Deep クラスをインスタンス化
Deep deep1 = new Deep();
//deep1 に"1"をセット
deep1.setValue("1");
```

まず、Deep インスタンスを作成し、値を代入しています。

この時のメモリ内部の状態は以下の通りです。

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス

Deep 型変数 deep1【#001】	001	Deep String 型変数 value【”1”】 void setValue(String str) String getValue()

続いて、下記のコードを実行しました。

```
//deep2 に deep1 をディープコピー
Deep deep2 = new Deep();
```

新しく Deep インスタンスを作成しています。

この時、ヒープ領域の状態と、deep2 変数の中身は以下のようになります。

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
Deep 型変数 deep1【#001】 Deep 型変数 deep2【#002】	001	Deep String 型変数 value【”1”】 void setValue(String str) String getValue()
	002	Deep String 型変数 value【””】 Void setValue(String str) String getValue()

ヒープ領域にもう 1 つ Deep インスタンスが作成され、参照値#002 が割り振られます。

そして、deep2 変数に参照値#002 が代入されました。

参照値#001 と参照値#002 はどちらも Deep インスタンスですが、別々に存在している事が分かります。

見た通り、この 2 つのインスタンスは全くの別物で、なんの関わりもありません。

これでは「コピー」とは言えないので、参照値#002 の Deep インスタンスに、参照値#001 の Deep インスタンスの値をコピーします。

```
deep2.setValue(deep1.getValue());
```

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
Deep 型変数 deep1【#001】 Deep 型変数 deep2【#002】	001	Deep String 型変数 value【”1”】 void setValue(String str) String getValue()
	002	Deep String 型変数 value【”1”】 Void setValue(String str) String getValue()

これで、2 つのインスタンスがそれぞれ持っている値が同一になりました。

それでは最後に、deep2 変数の持つインスタンスの値を変更してみましょう。

```
//deep2 に"2"をセット
deep2.setValue("2");
```

ディープコピーした deep2 の参照値#002 インスタンスの値が変更されると、下記のようになります。

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
Deep 型変数 deep1【#001】 Deep 型変数 deep2【#002】	001	Deep String 型変数 value【”1”】 void setValue(String str) String getValue()
	002	Deep String 型変数 value【”2”】 Void setValue(String str) String getValue()

参照値#002 のインスタンスの value 変数の値だけが変更されました。

このように、「別のインスタンスを作成」し、「内部の値だけを上書きコピー」する方法をディープコピーといいます。

これであれば、deep1 変数と deep2 変数はそれぞれ別のインスタンスを参照する為、個別に操作を行うことが可能となります。

基本型配列のコピー

ここまでは、通常のクラス型変数でのシャローコピーとディープコピーを見てきました。

これが配列になると、また少し内容が変わってきます。

まずは、Java のプログラム上でどのインスタンスを参照しているかを確認する為、以下のコードを確認してみましょう。

Sample1_9.java

```java
public class Sample1_9 {
    public static void main(String[] args) {
        //基本型配列を作成
        int[] ary = { 0, 1, 2 };
        //ary 変数のハッシュコードを出力
        System.out.println("ary 変数のハッシュコード:" + ary.hashCode());
    }
}
```

実行結果(数値は異なる可能性があります):

```
ary 変数のハッシュコード:925858445
```

配列もクラス型

今まで基本型のように扱ってきた配列も、厳密にはクラス型になります。

配列は要素を連続したアドレスに保存する。という性質を持っているので厳密には異なりますが、イメージとしてメモリ管理表を表現するのなら、以下のような形となります。

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
int 型配列変数 ary【#001】	001	[I index0:【0】

		index1:【1】 index2:【2】

ハッシュコード

ここまで、メモリ管理表を用いてきましたが、実際に Java のコードで参照値を出力する事が出来れば、確認もしやくなります。

そういった場合に利用されるのが Object クラスで定義されている hashCode メソッドです。

hashCode メソッドの厳密な使い方は少し意味合いが異なりますが、**オーバーライドをしていなければ**、参照値を元に整数値を生成しています。

つまり、出力される値が同一かどうかで、同じインスタンスか、異なるインスタンスかを確認することが可能です。

配列は hashCode()メソッドをオーバーライドしていませんので、ハッシュコードの値でどのインスタンスなのかを確認できます。

また、配列そのもののハッシュコードは、index 番号を付けずに変数名.hashCode()で取得する事が可能です。

基本型配列のシャローコピーとディープコピー

それでは、基本型配列のシャローコピーとディープコピーについてみていきましょう。

まずは、基本型配列を Arrays.copyOfRange()メソッドを利用してコピーをしてみましょう。

この時の動き方を、以下のサンプルコードで確認しましょう。

Sample1_10.java

```
import java.util.*;

public class Sample1_10 {
    public static void main(String[] args) {
        //配列を作成
        int[] ary1 = {0, 1, 2};
        //配列をコピー
        int[] ary2 = Arrays.copyOfRange(ary1, 0, ary1.length);
        //ary2 の値を変更
        ary2[0] = 3; ary2[1] = 4; ary2[2] = 5;
        //ary1 と ary2 のハッシュコードと値を出力
```

```
        System.out.println("ary1:" + Arrays.toString(ary1) + "#" + ary1.h
ashCode());
        System.out.println("ary2:" + Arrays.toString(ary2) + "#" + ary2.h
ashCode());
    }
}
```

実行結果(数値は異なる可能性があります):

```
ary1:[0, 1, 2]#925858445
ary2:[3, 4, 5]#798154996
```

基本型配列のメソッド利用したコピーは「ディープコピー」

実行結果からみる通り、基本型配列を用いた配列のコピーはディープコピーです。
つまり、実行時のメモリ内部は以下の通りです。

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
int 型配列変数 ary1【#001】 int 型配列変数 ary2【#002】	001	[I index[0]:【0】 index[1]:【1】 index[2]:【2】
	002	[I index[0]:【3】 index[1]:【4】 index[2]:【5】

COLUMN

基本型配列でシャローコピーをしたい場合はどうしたらいいでしょうか？

シャローコピーというのは、「参照値」の上書きコピーです。

つまり、以下のようにすれば簡単に実装する事が可能です。

```
int[] ary1 = { 0, 1, 2 };
int[] ary2 = ary1;
```

注意点としては、メソッドの引数や戻り値も、原則としてこのシャローコピーになるという事です。

もし引数や戻り値にディープコピーした配列を指定したい場合は、引数呼び出し前にディープコピーをしておきましょう。

シャローコピーとディープコピーを意識せずにクラス型変数や配列・リストを引数で渡すと、想定しない動き方をするので気を付けて下さい。

クラス型配列のコピー

ここまでは、基本型配列をメソッドでコピーした場合、ディープコピーになる事を確認しました。

これがクラス型配列になると、また少し内容が変わってきます。

まずは、クラス型配列を利用した時の動き方を確認しておきましょう。

Sample1_11.java

```
//テスト用のクラスを作成
class ArrayCopy {
    //文字列を保持するフィールド
    private String value;
    //セッター
    public void setValue(String str) { this.value = str; }
    //ゲッター
    public String getValue(){ return this.value; }
}

public class Sample1_11 {
    public static void main(String[] args) {
        //ArrayCopy 型配列を宣言し、インスタンスを 2 つ作成
```

```
        ArrayCopy[] ary = new ArrayCopy[2];
        ary[0] = new ArrayCopy();
        ary[1] = new ArrayCopy();

        //ary の要素をハッシュコードで表示
        System.out.println("ary[0]の中身:" + ary[0].hashCode());
        System.out.println("ary[1]の中身:" + ary[1].hashCode());
    }
}
```

実行結果(数値は異なる可能性があります):

```
ary[0]の中身:798154996
ary[1]の中身:681842940
```

要素のハッシュコード

上記のコードは、クラス型配列を作成し、2 つの要素としてそれぞれインスタンス化を行っています。

配列に index を指定すると、そこに入っているのはインスタンスの参照値です。

メモリ管理表のイメージとしては下記のようになります。

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
ArrayCopy 型配列変数 ary【#001】	001	[LArrayCopy index[0]:【#002】 index[1]:【#003】
	002	ArrayCopy String 型変数 value【""】 Void setValue(String str) String getValue()
	003	ArrayCopy String 型変数 value【""】 Void setValue(String str) String getValue()

クラス型配列のメソッドを利用したコピー

それでは、実際にクラス型配列のコピーを見ていきましょう。

今回も Arrays.copyOfRange()メソッドを利用して、シャローコピーなのか、ディープコピーなのかを確認してみましょう。

Sample1_12.java

```
import java.util.*;

//テスト用のクラスを作成
class ArrayCopy {
    //文字列を保持するフィールド
    private String value;
    //セッター
    public void setValue(String str) { this.value = str; }
    //ゲッター
    public String getValue(){ return this.value; }
}
public class Sample1_12 {
    public static void main(String[] args) {
        //ArrayCopy 型配列を作成し、インスタンスを 2 つ生成
        ArrayCopy[] ary1 = { new ArrayCopy(), new ArrayCopy() };
        //Arrays.copyOfRange()メソッドでコピーを作成
        ArrayCopy[] ary2 = Arrays.copyOfRange(ary1, 0, 2);
        //それぞれのハッシュコードを出力
        System.out.println("ary1:" + ary1.hashCode() + "|ary1[0]:" + ary1
[0].hashCode() + "|ary1[1]:" + ary1[1].hashCode());
        System.out.println("ary2:" + ary2.hashCode() + "|ary2[0]:" + ary2
[0].hashCode() + "|ary2[1]:" + ary2[1].hashCode());
    }
}
```

実行結果(数値は異なる可能性があります):

```
ary1:798154996|ary1[0]:681842940|ary1[1]:1392838282
ary2:523429237|ary2[0]:681842940|ary2[1]:1392838282
```

Arrays クラスの配列コピーは、要素はシャローコピー

実行結果から見てわかる通り、配列そのものはディープコピーですが、要素内はシャローコピーです。

メモリ内部の管理表イメージとしては、以下の通りです。

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
ArrayCopy 型配列変数 ary【#001】 ArrayCopy 型配列変数 ary【#004】	001	[LArrayCopy index[0]:【#002】 index[1]:【#003】
	002	ArrayCopy String 型変数 value【""】 Void setValue(String str) String getValue()
	003	ArrayCopy String 型変数 value【""】 Void setValue(String str) String getValue()
	004	[LArrayCopy index[0]:【#002】 index[1]:【#003】

ここまでの動作をまとめると、以下のような動き方になることが分かります

・基本型配列

・メソッドを使うと「ディープコピー」

クラス型配列

・メソッドを使うと配列そのものは「ディープコピー」

・メソッドを使うと要素は「シャローコピー」

クラス型配列のディープコピー

では、実際にクラス型配列でディープコピーをしたい場合はどのようにしたらよいでしょうか。

答えはいたってシンプルです。
以下のコードを確認してみましょう。

Sample1_13.java

```
//テスト用のクラスを作成
class ArrayCopy {
    //文字列を保持するフィールド
    private String value;
    //セッター
    public void setValue(String str) { this.value = str; }
    //ゲッター
    public String getValue(){ return this.value; }
}
public class Sample1_13 {
    public static void main(String[] args) {
        //クラス型配列を用意
        ArrayCopy[] ary1 = {new ArrayCopy(), new ArrayCopy()};
        ary1[0].setValue("0"); ary1[1].setValue("1");

        //クラス型配列のディープコピー
        ArrayCopy[] ary2 = new ArrayCopy[ary1.length];
        for (int i = 0; i < ary1.length; i++) {
            ary2[i] = new ArrayCopy();
            ary2[i].setValue(ary1[i].getValue());
        }
        System.out.println("ary1:" + ary1.hashCode() + "|ary1[0]:" + ary1
[0].hashCode() + "|ary1[1]:" + ary1[1].hashCode());
        System.out.println("ary2:" + ary2.hashCode() + "|ary2[0]:" + ary2
[0].hashCode() + "|ary2[1]:" + ary2[1].hashCode());
    }
}
```

実行結果

実行結果(数値は異なる可能性があります):

```
ary1:798154996|ary1[0]:681842940|ary1[1]:1392838282
ary2:523429237|ary2[0]:664740647|ary2[1]:804564176
```

上書き処理をコード化する

クラス型配列のディープコピーはメソッドが用意されていませんので、自分で作成する必要があります。そう、答えは「自分で作る」です。

必要な数だけインスタンス化を行い、それぞれインスタンスに値を代入していきます。

最終的なメモリ管理表は以下のようになります。

【Java のメモリ管理表(イメージ)】

スタック領域	ヒープ領域	
	アドレス (参照値)	インスタンス
ArrayCopy 型配列変数 ary【#001】 ArrayCopy 型配列変数 ary【#004】	001	[LArrayCopy index[0]:【#002】 index[1]:【#003】
	002	ArrayCopy String 型変数 value【”0”】 Void setValue(String str) String getValue()
	003	ArrayCopy String 型変数 value【”1”】 Void setValue(String str) String getValue()
	004	[LArrayCopy index[0]:【#005】 index[1]:【#006】
	005	ArrayCopy String 型変数 value【”0”】 Void setValue(String str) String getValue()
	006	ArrayCopy String 型変数 value【”1”】 Void setValue(String str) String getValue()

なぜ、クラス型配列でディープコピーが行えるメソッドが用意されていないのでしょうか。

簡単に言ってしまえば、クラス型配列に於けるディープコピーは実装する事ができないからです。

Java でプログラムを作成する場合、プログラマーは様々なクラスを作成します。

そして、クラス毎に持つ属性や操作は、そのクラス毎に全て異なるものとなります。

ディープコピーをしようと思ったら、それぞれの属性の数や種類、どの操作で値の代入を行うのか、取得するのか。という情報がクラス毎に個別に必要となります。

皆さんが作成するオリジナルのクラスに対応させようとしても、メソッドを提供するプログラムの作成段階でまだ存在していないクラスに対応させる事は、事実上不可能なのです。

練習問題

問1　Practice5.java を作成し、以下の設問通りにコーディングしてください。

1.テスト用の Test クラスを作成し、以下の要件通りにして下さい。

・String 型のフィールドを 1 つ持つ

・void setValue(String str)メソッドを呼び出すと、フィールドに値を代入する。

・String getValue()メソッドを呼び出すと、フィールドの値を戻す

2.要素として Test クラスを 3 つ持つクラス型配列を作成してください。

3.上記クラス型配列を、メソッドを使って先頭から 2 要素分、シャローコピーしてください。

4.2 で作成したクラス型配列を、コードを実装してディープコピーしてください。

5.作成した 3 つのクラス型配列のハッシュコードを全て出力してください。

実行例:

```
ary1:798154996|ary1[0]681842940|ary1[1]1392838282|ary1[2]:523429237
ary2:664740647|ary2[0]681842940|ary2[1]1392838282
ary3:804564176|ary3[0]1421795058|ary3[1]1555009629|ary3[2]:41359092
```

Collections クラス

Arrays クラスが配列の操作に使える便利なクラスであったように、リストには Collections クラスが用意されています。

Collections クラスを利用するとリストに対しての様々な操作が簡単に行えます。

この項目では、Collections クラスについて学んでいきましょう。

Collections クラスの便利なメソッド

Collections クラスはリストを含むコレクション全般を操作するクラスです。

Arrays クラスと同様に、**メソッドは全て static で定義されており**、インスタンス化しなくてもいつでも使えるという特徴があります。

Collections クラスのよく利用されるメソッドは下記の通りです。

戻り値	メソッド	説明
Boolean	replaceAll(List<T> list, T oldVal, T newVal)	リスト内に出現する指定された値をすべてほかの値に置き換えます
Void	reverse(List<?> list)	指定されたリストの要素の順序を逆にします。
Void	shuffle(List<?> list)	デフォルトの乱数発生の元を使用して、指定されたリストの順序を無作為に入れ替えます。
Void	sort(List<T> list)	指定されたリストを、その要素の自然順序付けに従って昇順にソートします。

それでは、それぞれのメソッドを実際に使用してみましょう。

Sample1_14.java

```
import java.util.*;
public class Sample1_14 {
    public static void main(String[] args) {
        //リストを作成
        List<Integer> list = new ArrayList<>(Arrays.asList(0, 1, 2, 3));
        System.out.println("元リスト:" + list);

        //要素の置換
        Collections.replaceAll(list, 0, 4);
        System.out.println("置換:" + list);
```

```
        //要素の逆順(index の反転)
        Collections.reverse(list);
        System.out.println("逆順:" + list);

        //要素の並び替え(ランダム)
        Collections.shuffle(list);
        System.out.println("ランダム:" + list);

        //要素の並び替え(昇順)
        Collections.sort(list);
        System.out.println("昇順:" + list);
    }
}
```

実行結果:

```
元リスト:[0, 1, 2, 3]
置換:[4, 1, 2, 3]
逆順:[3, 2, 1, 4]
ランダム:[2, 4, 1, 3]
昇順:[1, 2, 3, 4]
```

可変長リストを一行で作る

リストの宣言部を確認してみましょう。

```
//リストを作成
List<Integer> list = new ArrayList<>(Arrays.asList(0, 1, 2, 3));
```

ArrayList のコンストラクタに Arrays.asList を使用する方法は解説済みですが、実は Arrays.asList の引数に要素を直接指定する事が可能です。

実行時にメモリ消費の増加とインスタンス化の処理が発生しますが、メモリや処理速度に余裕がある場合に、add メソッドを呼び出すのが手間な時は、この方法を活用しましょう。

replaceAll

replaceAll メソッドは指定した要素を置換します。

```
Collections.replaceAll(list, 0, 4);
```

第 1 引数が対象となるリスト、第 2 引数が検索する要素、第 2 引数が置換する要素です。
今回は 0 の要素を 4 に変更しているので、出力結果が以下の通りとなっています。

```
置換:[4, 1, 2, 3]
```

reverse

reverse メソッドは、リストを逆順に並び替えます。

```
Collections.reverse(list);
```

ここで言う逆順とは、index の値をひっくり返すという事です。
今回の要素は[4, 1, 2, 3]と並んでいるので、これを逆順にすると[3, 2, 1, 4]となります。

```
逆順:[3, 2, 1, 4]
```

昇順ソートのように要素を基準として並び替えるのではない点に注意してください。

shuffle

リストの中身をランダムに入れ替えます。

```
Collections.shuffle(list);
```

リスト内の要素をランダムに並び替えたい時に利用します。
ランダムに要素を出力したい時に便利です。

sort

要素の順序で昇順に並び替えを行います。

```
//要素の並び替え(昇順)
Collections.sort(list);
```

shuffle した後に戻したり、ランダムに入力されたデータを並び替える時に使用します。
Arrays.sort のリスト版として覚えておきましょう。

練習問題

問1　　Practice6.java を作成し、以下の設問通りにコーディングしてください。

1.String 型のリストを作成し、以下の要素(1 行 1 要素とします)を追加してください。

public
static
main
String

2.「String」の要素を「int」に置換してください。

3.リストをランダムに並び替えてください。

4.リストの要素数で乱数を作成し、ランダムな要素を画面に出力してください。

5.ユーザの入力を 1 行で受け付け、入力された値が表示された値と一致するか判定し、一致すれば「OK」、不一致の場合は「NG」と出力してください。

実行例:

```
public
public
OK
```

マップ

Java のコレクションフレームワークの中で、リストの次に最もよく利用されているのが java.util.Map<K, V>です。そのまま「マップ」と呼ばれます。

Map は「キー」と「値」を 1 セットで格納する事が出来ます。

値を取得する時や代入する時は、キーを指定して操作を行います。

配列やリストがインデックス番号を 0 から振るのに対して、Map はインデックス番号の代わりにキーを使用します。更に、何をキーにするかは自由に設定する事が可能です。

尚、Map は「キー」と「値」で対象を特定する性質上、データの順番は保持しません。キーを指定せずに出力した場合、順番はランダムになってしまうので注意してください。

Map の基本操作

まずは Map を実際のコードで書いて、動き方を見ていきましょう。

Map の大きな特徴は、「キー」と「値」を別々の型にする事が出来るという点です。

但し、リストと同じように使用できる型はクラス型に限定されます。

Sample1_15.java

```
import java.util.*;

public class Sample1_15 {
    public static void main(String[] args) {
        //Map を宣言
        Map<Integer, String> map = new HashMap<>();

        //Map にキーと値を追加
        map.put(100, "A");
        map.put(80, "B");
        map.put(60, "C");

        //Map の値をキーから取得
        System.out.println(map.get(100));
        System.out.println(map.get(80));
        System.out.println(map.get(60));
    }
}
```

実行結果:

```
A
B
C
```

Map の宣言

Map はインターフェースになります。

実際に使用するインスタンスでよく利用されるのは java.util.HashMap<K, V>です。

そして、リストと同様に、Map インターフェース型に HashMap インスタンスを生成するのが望ましいとされています。

```
//Map を宣言
Map<Integer, String> map = new HashMap<>();
```

ジェネリクスで指定するのは、「キー」と「値」です。

Map<キー, 値>で必ず 2 つ宣言する必要がある点に注意をしてください。

尚、今回は<Integer, String>としましたが、<String, String>や<String, Integer>など、クラス型であれば組み合わせは自由となります。

要素の追加

Map に要素を追加する時に使用するメソッドは put メソッドです。

要素を追加する時は、必ずキーと値をセットにして指定します。

```
//Map にキーと値を追加
map.put(100, "A");
map.put(80, "B");
map.put(60, "C");
```

要素を追加する時は、必ずジェネリクスで指定したキーと値、その順番に注意してください。

また、**キーは一意でなければなりません**。もし既に追加したキーを指定して新しい値を指定した場合は、そのキーの値が上書きされます。

要素の取得

要素の取得は get メソッドを使用します。

```
//Map の値をキーから取得
System.out.println(map.get(100));
System.out.println(map.get(80));
System.out.println(map.get(60));
```

get メソッドの引数に「キー」を指定します。
尚、存在しないキーを指定した場合、エラーにはならず「null」という値が帰ってきます。

Map の要素を一覧表示する

Map の要素を一覧で表示する事が可能です。
それぞれ、「キーと値のペアの一覧」「キーの一覧」「値の一覧」と複数パターンが用意されています。
それぞれの動き方を見てみましょう。

Sample1_16.java

```
import java.util.*;
public class Sample1_16 {
    public static void main(String[] args) {
        //Map の宣言
        Map<Character, String> map = new HashMap<>();
        //要素の代入
        map.put('さ', "さとう");
        map.put('し', "しお");
        map.put('す', "す");
        map.put('せ', "せうゆ");
        map.put('そ', "そーす");
        //キーと値の一覧
        System.out.println("キーと値の一覧:" + map.entrySet());
        //キーの一覧
        System.out.println("キーの一覧:" + map.keySet());
        //値の一覧
        System.out.println("値の一覧:" + map.values());
    }
}
```

実行結果:

```
キーと値の一覧:[さ=さとう, し=しお, す=す, せ=せうゆ, そ=そーす]
```

```
キーの一覧:[さ, し, す, せ, そ]
値の一覧:[さとう, しお, す, せうゆ, そーす]
```

entrySet

entrySet メソッドは、「キー=値」の形式で一覧表示します。

マップが持っている要素の一覧を確認したい時に利用しましょう。

keySet、values

keySet はキーの一覧を、values は値の一覧を取得します。

これらは単体で一覧表示されることは少ないですが、拡張 for と組み合わせる事でその効力を発揮します。

拡張 for

Map は拡張 for を使って、一気に要素を取得する事が可能です。以下のコードで動き方を確認してみましょう。

Sample1_17.java

```java
import java.util.*;

public class Sample1_17 {
    public static void main(String[] args) {
            //Map の宣言
            Map<Character, String> map = new HashMap<>();
            //要素の代入
            map.put('さ', "さとう");
            map.put('し', "しお");
            map.put('す', "す");
            map.put('せ', "せうゆ");
            map.put('そ', "そーす");

            //拡張 for でキーの一覧を取得する
            for (Character key: map.keySet()) {
                //キーを使って値を取得して表示する
                String value = map.get(key);
                System.out.println(key + ":" + value);
            }
```

```
    }
}
```

実行結果:

```
さ:さとう
し:しお
す:す
せ:せうゆ
そ:そーす
```

key で取得する

拡張 for のコードを確認してみましょう。

```
for (Character key: map.keySet()) {
```

実際に取得している値はキーになります。

このように keySet メソッドを利用すると、キーの一覧を 1 つずつ取得する事が可能です。

キーを 1 つずつ取得し、get メソッドにそのキーを指定する事で、全ての値を取り出しています。

この使い方はマップでは基本的なやり方ですので、しっかりと覚えておきましょう。

Map の便利なメソッド

Map インターフェースには、マップを操作するためのメソッドが多数用意されています。

要素の形式がリストとは違う為、使い方も含めよく確認しておきましょう。

戻り値	メソッド	説明
void	clear()	マップからマッピングをすべて削除します(オプションの操作)。
boolean	containsKey(Object key)	指定されたキーのマッピングがこのマップに含まれている場合に true を返します。
boolean	containsValue(Object value)	このマップが 1 つまたは複数のキーと指定された値をマッピングしている場合に true を返します。
static <K,V>	copyOf(Map<? extends K,?	指定された Map のエントリを

Map<K,V>	extends V> map)	含む「変更不可能なマップ」を返します。
Set<Map.Entry<K, V>>	entrySet()	このマップに含まれるマッピングの Set ビューを返します。
V	get(Object key)	指定されたキーがマップされている値を返します。そのキーのマッピングがこのマップに含まれていない場合は null を返します。
Set<K>	keySet()	このマップに含まれるキーの Set ビューを返します。
V	remove(Object key)	このマップからキーのマッピング(ある場合)を削除します(オプションの操作)。
default boolean	remove (Object key, Object value)	指定された値に指定されたキーが現在マッピングされている場合にのみ、そのキーのエントリを削除します。
default V	replace(K key, V value)	指定されたキーがなんらかの値に現在マッピングされている場合にのみ、そのキーのエントリを置換します。
default boolean	replace (K key, V oldValue, V newValue)	指定されたキーが指定された値に現在マッピングされている場合にのみ、そのキーのエントリを置換します。
int	size()	このマップ内のキー値マッピングの数を返します。
Collection<V>	values()	このマップに含まれる値の Collection ビューを返します。

キーと値の関係上、使用するメソッド数も相応に数が用意されています。

全てを確認するのは大変なので、基本的な操作部分をサンプルコードで確認していきましょう。

Sample1_18.java

```java
import java.util.*;
public class Sample1_18 {
    public static void main(String[] args) {
        //マップを宣言
        Map<String, String> map = new HashMap<>();

        //要素を追加
        map.put("東京都", "新宿区");
        map.put("千葉県", "千葉市");
        map.put("神奈川県", "横浜市");
        map.put("埼玉県", "さいたま市");

        //キーが存在するか確認する
        System.out.println("東京都:" + map.containsKey("東京都"));
        //値が存在するか確認する
        System.out.println("新宿区:" + map.containsValue("新宿区"));

        //キーを削除する
        map.remove("埼玉県");
        System.out.println("埼玉県:" + map.containsKey("埼玉県"));

        //キーを置き換える
        map.replace("神奈川県", "栃木県");
        System.out.println("栃木県:" + map.containsKey("栃木県"));

        //マップの要素数を確認する
        System.out.println("要素数:" + map.size());
    }
}
```

実行結果:

```
東京都:true
新宿区:true
埼玉県:false
栃木県:false
要素数:3
```

containsKey、containsValue

引数に渡したキーまたは値がマップ内の要素に存在するかを調べます。

結果は boolean 型で戻るので、if 文に直接組み込むことも可能です。

remove

キーを元に要素を削除します。

同じ値が複数ある場合で特定のキーと値の組み合わせを削除したい時などは、引数の数で動作を指定します。

replace

キーの置換が出来ます。別のキーにしたい時に使用します。

size

要素数を戻します。

マップの要素数は「キーと値のセット」が 1 要素となります。

よって、今回の例では「3」が戻ってきます。

練習問題

問1　Practice7.java を作成し、以下の設問通りにコーディングしてください。

1.String 型のキーと String 型の値を持つマップを宣言してください。

2.マップに以下の要素を追加します。(左側がキー:右側が値です)

競泳:金

アーチェリー:銅

ウエイトリフティング:銅

フェンシング:金

レスリング:銀

3.「柔道」というキーがあれば、その値を出力し、なければ「結果はまだ出ていません」と出力してください。

4.「フェンシング」というキーがあれば、その値を出力し、なければ「結果はまだ出ていません」と出力してください。

5.「金」という値が何個あるか、画面に出力してください。

実行例:

```
柔道の結果:結果はまだ出ていません
フェンシングの結果:金
金メダルの数:2
```