Java演習
第12回

2024/7/3

横山大作

講義前・後の質問

- 横山まで
- dyokoyama@meiji.ac.jp

提出課題10: 複製（再掲）

- 箱詰めのチョコレートを表現しようとしたクラスが作られている
  - class Chocolate: 1粒のチョコレート、味がintで表現されている
  - class Box: チョコが何粒か入った箱。Chocolateインスタンスがいくつか配列で覚えてある
- 3個詰めセットのサンプルとなるsampleインスタンスが用意されていた
  - 味は[0, 1, 2]だった
- これを使って、1個だけ味を変更したboxインスタンスを作りたい
  - sampleをコピーしてboxを作り、boxの0番目の味を9に変える(set_flavor)
  - 味が[9, 1, 2]になるはず
- このような処理が問題なく（sampleに影響を与えることなく）行えるようにプログラムを完成してほしい

やりたいこと

- 絵に描けますか？

- Deep Copyが必要ですね
  - チョコレートインスタンスも複製しないとだめ

コピーコンストラクタで複製

```
class Chocolate {
    private int flavor;
    Chocolate(int f) { flavor = f; }
    Chocolate(Chocolate c) { flavor = c.flavor; }
}

class Box {
    Chocolate piece[];
    Box(Chocolate[] c) {
        piece = c;
    }
    Box(Box org) {
        piece = new Chocolate[org.piece.length];
        for (int i = 0; i < org.piece.length; i++) {
            piece[i] = new Chocolate(org.piece[i]);
        }
    }
}
```

ソースは適宜
抜粋しています

コピーコンストラクタ
使用

コピーコンストラクタで複製（続き）

```
    public static void main(String[] args) {
        Chocolate[] choco = {new Chocolate(0), new Chocolate(1),
new Chocolate(2) };
        Box sample = new Box(choco);

        // boxには最初、sampleと同じチョコが入っているようにする（コピーする）
        Box box = new Box(sample);

        // boxの0個目のチョコについて、味を9に変更
        box.piece[0].set_flavor(9);

        // box, sampleを表示するとどうなるか？
        System.out.println(box);
        System.out.println(sample);
    }
```

コピーコンストラクタ使用

clone()で複製（参考）

```
class Chocolate implements Cloneable {
    int flavor;
    public Chocolate clone() {
        Chocolate ret = new Chocolate(flavor);
        return ret;
    }
}

class Box implements Cloneable {
    Chocolate piece[];
    public Box clone() {
        Chocolate[] p = new Chocolate[piece.length];
        for (int i = 0; i < piece.length; i++) {
            p[i] = piece[i].clone();
        }
        Box ret = new Box(p);
        return ret;
    }
}
```

ここでは元からあったコンストラクタを使っている

clone()使っている

ここも元からあったコンストラクタ

clone()で複製（参考）（続き）

```
    public static void main(String[] args) {
        Chocolate[] choco = {new Chocolate(0), new Chocolate(1),
new Chocolate(2) };
        Box sample = new Box(choco);

        // boxには最初、sampleと同じチョコが入っているようにする（コ
ピーする）
        Box box = sample.clone();

        // boxの0個目のチョコについて、味を9に変更
        box.piece[0].set_flavor(9);

        // box, sampleを表示するとどうなるか？
        System.out.println(box);
        System.out.println(sample);
    }
```

ここが変わるだけ

この資料の内容

- コレクション
  - Listインタフェース
  - Setインタフェース
  - Collectionインタフェース
- ジェネリクス

- 提出課題11

コレクション(p.574)

- データをまとめて処理するときには配列を利用していた
  - もっと便利に使いたい
    - 例えば、（途中への）追加削除をもっと簡単にするには？
  - 求める機能が色々ある
    - 全機能入りのスーパー配列、みたいなものはできない
- 機能ごとに「袋」オブジェクトを用意する
  - コレクション
  - オブジェクトをまとめるオブジェクト
- Javaは機能の異なるいくつかのコレクションを提供
  - ここでは3種類を見ていく

抽象データ型

- これまでの学習ではデータをまとめる方法は具体的
  - データの置き方が決まっている
    - クラス（構造体）：ひとかたまり
    - 配列：データがつながって配置されている
- 必ずしも具体的に決める必要はない
  - 名前の一覧だけ覚えておきたい
  - 各人がどの科目を取ったかだけ覚えておきたい
  - …
- 「こういうことがしたい」という性質だけを決めておくのが「抽象データ型」
  - コレクションは抽象データ型と具体的な実装の集まり

袋を大きく分類

- 順序を保つ必要がある
  - List
- 順序には(あまり)興味がない
  - 要素だけで良い
    - Set
  - 対応関係が必要
    - Map

3種類の袋の正体

- 3つのインタフェース
  - こういう特性の（こういうメソッドを持った）袋、という定義

- java.util.List
- java.util.Set
- java.util.Map

インタフェースと実装クラス

- コレクションの性質を規定しているのはインタフェース
  - Listとは何か、を宣言
- 実際に使うのは実装クラス
  - new ArrayList(); として使う
- 実装クラスはより細かい要求に従って選ぶ
  - 使い方によって性能、ふるまいが異なる

代表的な実装クラス

- List
  - ArrayList, LinkedList
- Set
  - HashSet, TreeSet
- Map
  - HashMap, TreeMap

混乱しがちなこと

- 中身のデータとコレクションの構造をごっちゃにしないように

- 袋：コレクション
- 袋の中身：コレクションの要素のデータ型

- 袋: 配列
- 袋の中身: String　とか

- 袋にもいろいろな種類がある、もちろん中身にも

List

- 配列はnewするときに要素数が決まっていた
- 後から足したくなったら？
- あるいは、要素を減らしたくなったら？

- 自由に出し入れできるような袋を考えたい

Listインタフェース

- リストとは、以下のような操作ができるもの
  - add(E obj)
  - E get(int index)
  - E remove(int index)
  - int size()
  - iterator()
  - toArray()

- 出し入れ、削除
  - 何番目の要素、と指定できる
  - 配列の代わりにも使えそう

Listインタフェース

- まずはジェネリクスを使わない状態での使い方を見てみよう

- 出し入れするのはObject型のインスタンス
  - 入れるときはObject型への代入になるのでそのまま書ける(親クラス型への代入)
  - 出すときはObject型として出てくるのでキャストが必要(子クラス型への代入)

```
ArrayList l = new ArrayList();
l.add(“東京”);
l.add(“ロンドン”);

String item = (String)l.get(1);
```

型引数の指定 (p.608)

- 「String型が入るList」「Integer型が入るリスト」のように、要素の型を指定することができる
- これを指定すれば
  - 要素を追加するときに違う型を入れるとエラー
  - 要素を取り出すときに決まった型で出てくる
    - のでキャストの必要なし

```
ArrayList<String> l = new ArrayList<String>();
l.add(“東京”);
l.add(“ロンドン”);

String item = l.get(1);
```

ジェネリクス (p.608)

- List<String>のように、「何か別のクラスを利用したクラス」や「何かの型専用のメソッド」などが存在する
- これをJavaで書けるようにしたのがジェネリクス

- ある型の情報を含んだ型、と思ってよい
  - ある型によって専用化された型

何かの型専用になった型

- List<Integer>
  - 要素としてIntegerしか入らないリスト
- Comparator<String>
  - Stringを2つ受け取って、大小（前後）関係を判定してくれるような「比較器」

- Comparatorは説明ではComparator<T>と書いてある
  - int compare(T o1, T o2)
  - T型のデータを2つ取って、intを返す関数を定義せよ

Comparator<T>の例（課題9を再掲）

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

        // . . . 文字列としてソートするのに続いて . . .

        Arrays.sort(dat, new Comparator<String>() {
            public int compare(String lo, String ro) {
                int l = Integer.parseInt(lo);
                int r = Integer.parseInt(ro);
                return l - r;
            }
        });
        for (String s : dat) {
            System.out.println(s);
        }
    }
}
```

キャストがいらなくなる

ArrayListの使い方

```
ArrayList<String> l = new ArrayList<String>();
l.add(“東京”);
l.add(“ロンドン”);

String item = l.get(1);
```

- newするとき、型パラメータ部分(<>の中)はコンパイラに推測させられる

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

コレクションと基本データ型

- コレクションには「オブジェクト」しか入れられない
- intなどの基本データ型はオブジェクトではない
- List<int>などは作れない

- どうする？

ラッパークラス (p.578)

- 基本データ型を「くるんで」オブジェクト化するようなクラス
- 基本データ型ごとに対応したクラスあり

```
Integer n = new Integer(3);
int num = n.intValue();

ArrayList<Integer> l = new ArrayList<Integer>();
l.add(new Integer(5));
```

ラッパークラスの機能

- 文字列との変換
- 定数がある
  - NaNとか
  - SIZE、BYTESとかの情報もあり
- オートボクシング(p.579)
  - 基本データ型と自動変換が可能
  - うっかりするとバグの原因になる
    - nullかもよ？

```
int num2 = Integer.parseInt(”356”);
```

```
Integer n = new Integer(3);
int num = n;  // 自動的に変換してくれる

ArrayList<Integer> l = new ArrayList<Integer>();
l.add(5); // 自動的に変換してくれる
```

インタフェースで使う(p.593)

- 具体クラスではなく、インタフェース型の変数で利用することが多い
  - それで十分だから
  - 抽象度が高いまま使うと、詳細を変更しても影響が起きにくい
    - 具体的には、実装クラスを変えたくなった時にソースの変更箇所が少ない

```
List<String> l = new ArrayList<String>();
l.add(“東京”);
l.add(“ロンドン”);

String item = l.get(1);
```

拡張for文で走査する (p.585)

- 拡張for文は抽象度が高い処理
  - インデックスに依らず、すべてのデータを処理する
  - コレクションと相性が良い
- 実は配列でも使えてた
- Listだけでなく、SetやMapでも使える

```
List<String> l = new ArrayList<String>();
l.add(“東京”);
l.add(“ロンドン”);

for (String item : l) {
    System.out.println(item);
}
```

ArrayListとLinkedList (p.590)

- 実装が異なる
- 配列で実装されているか、リンクで実装されているか

- 途中での挿入削除が多いならLinkedList
- データをなめるだけならArrayList

Set (p. 595)

- 集合
  - 順序がない
  - 重複がない
- 順序がないということは、「3番目」とかの要素が取れないということ
- 主なメソッド
  - 出し入れ
  - 含まれるかのチェック(contains())
    - 高速にチェックできることが想定されている

Collectionインタフェース (p.605)

- List, SetはCollectionインタフェースを継承している
  - 一般的な「集合」を表すもの
  - Iterableインタフェースが先祖にある
  - 拡張for文が使える

- Collectionインタフェースの範囲で使っていれば、実装をArrayListからHashSetに差し替えることも可能

HashSetの構造

- まずハッシュ値で分ける
  - Bucket(バケツ)に入れる、という表現をする
- 同じバケツに入ったものはリスト管理
  - equals()で比較して見つける

自作のクラスをコレクションに入れる(p.609)

- 自分で定義したオブジェクトを袋に入れて管理したい

- コレクションの管理に必要なメソッドを理解しよう
  - equals()
  - hashCode()

自作クラスでequals()を作る

- equals()を定義すれば等価チェックに対応できる
  - public boolean equals(Object obj) をそのクラスに実装する
    - 比較するobjが自分と「等価」であるときにtrue、それ以外はfalseを返す
  - 実装しなくてはならないメソッドの返り値、引数などは決まっている
    - Object.equals()をオーバーライドして使うため

equals()

- オーバーライドして使う
- 引数を間違えてオーバーロードになっちゃってる場合がありがち
  - コレクションの中などではObjectとの比較が呼ばれるから、オーバーロード版が呼ばれない
  - @Override書くと気づける

```
class Point {
  int x;
  int y;

  public boolean equals(Object o) {
    // こっちが正しい
  }
  public boolean equals(Point p) {
    // こう書きがちだがやめよう。呼ばれないことも。
  }
}
```

equals()の作り方

- 流れはだいたい決まっている
  - インスタンスとして同じならreturn true
  - nullとの比較だったらreturn false
  - 違うクラスのオブジェクトとの比較だったらreturn false
  - 相手を自分のクラスにキャストして、中身を比較
- 手で書くのは割と面倒
- Eclipseでひな形作ってくれるのでそれを活用しよう

自作クラスでequals()を作る時

- equals()に加えて、hashCode()メソッドを定義することを強くおススメ
  - コレクションの実装によっては必要になる
  - equals()とhashCode()が違う挙動をすると気づきにくいバグが起きる
- Eclipseで「ソース」->「hashCode()とequals()を追加」とかで自動生成されるくらい、一緒に定義するべき
  - この方法で生成されるコードは結構複雑なことをやっているので、見てみると面白い

hashCode()メソッド

- ハッシュを使うときに必要
  - 中身の型の話
- Objects.hash()メソッドが簡単　(Java 7より)
  - 引数の数が可変
  - ハッシュに使いたい要素を並べる

```
import java.util.Objects;

class Point {
  int x;
  int y;

  public int hashCode() {
    return Objects.hash(x, y);
  }
}
```

equals()とhashCode()

- 「等しい」ならば同じハッシュでないとおかしい
  - equalsだけ実装すると、この要求が満たせなくなる
- 同じハッシュでも等しくない場合はある

Map

- は次回に回します

「リストの配列作りたいんですけど…」

- List<String>[] data = new ArrayList<String>[100];
  - やりたくなるよね
- Cannot create a generic array of ArrayList<String> というコンパイルエラーが消えない
- ジェネリクスを使っているクラスの配列は作れない
  - そう決まっている
- なぜ？

こんなことしたくなるよね

```
String[] sa = new String[10];
Object[] oa = sa;
```

一般的な型で扱いたい

```
void consume_o(Object[] o) { ... }


String[] sa = new String[10];
consume_o(sa);
```

一般的な型で作られた関数に
具体的な型のデータを渡したい

こんな危険がある

```
String[] sa = new String[10];
Object[] oa = sa;
oa[0] = new Integer(3);
```

Stringの配列なのに
要素にInteger入ってしまう

```
void consume_o(Object[] o) {


String[] sa = new String[10];
consume_o(sa);
```

コンパイル時にはチェック
できない
ランタイムに例外発生
java.lang.ArrayStoreException

配列とジェネリクスの違い

```
String[] sa = new String[10];
Object[] oa = sa;


List<String> sl = new ArrayList<String>();
List<Object> ol = sl; // コンパイルエラー
```

- Listも配列みたいに代入できたほうが便利そうだけど…

参考: イレイジャ

- Javaのジェネリクスはコンパイル時にのみ型を意識する
  - コンパイルが終わったら型情報を消す
  - 実行時はList<String>もList<Integer>もすべてList扱い
    - List<?>扱い
  - instanceofとかで区別がつかなくなっている

```
List<String> sl = new ArrayList<String>();
List<Object> ol = sl; // コンパイルエラー…にならないとすると

ol.add(new Integer(3));
```

実行時でもエラーがわからない
slがString用だったことを忘れて
しまっているので

参考: 変性(variance)

今、ObjectはStringの親クラスである。この時、

- 共変(covariant)
  - List<Object>はList<String>の親クラス
  - Javaの配列はこれ
- 不変(invariant)
  - List<Object>とList<String>は関係がないクラス
  - Javaのジェネリクスはこれ
- 反変(contravariant)
  - 共変の逆

要するに

- Javaの配列は安全性を気にせず共変(covariant)で扱うことにした
  - 安全性は実行時にチェックすることにした
- ジェネリクスは互換性の都合上イレイジャで実装した
- 安全性の担保のために、不変(invariant)で扱うことにした

- 変性が違うので、混ぜて使えない

とはいえ

```
void consume_o(List<Object> o) { ... }


List<String> sa = new ArrayList<String>();
consume_o(sa);
```

- こんなこともしたくなるよね

型の制約を付けられる

```
void consume_o(List<? extends Object> o) { ... }


List<String> sa = new ArrayList<String>();
consume_o(sa);
```

- <>の中の型パラメータに制約が付けられる
  - T extends B: Bか、Bの子孫
  - T super B: Bか、Bの先祖
  - ?: 制約を満たすあるクラス
- 「これは使って大丈夫」というときにextendsやsuperで指定できるようにした

- あるデータ構造を「使う」ときと「使わせる」ときで要求される関係が逆になるため
  - List<Object>を使う人はList<String>も使える、List<String>を使う人はList<Object>を渡されても困る
  - List<String>を出す人はList<Object>を出すと取り扱ってもらって構わない、List<Object>を出す人はList<String>として使われると困る
- どのような意味があるかをよく考えないと、正しく指定できない

- このあたりを考えなければならないのはジェネリクスで汎用なツールを「作る側」
  - 「使う側」だと実はあまり知らなくても良い（正しく作られていれば）

- このあたりの話は新しい言語だともう少し扱いやすくなっている
  - ScalaとかC#とか…
- 型に厳しい言語を勉強し始めたら思い出してみよう

使うときに制約をつける例

- ListはList<String> l; のように使うのが普通
- 型が不明な時には
  - List l;
    - ジェネリクスを使わない状態になる。
  - List<?> l;
    - 制約のない何かの型、になる
    - List<Object>でもないので注意

```
List<?> l = new ArrayList<String>();
int x = l.size();     // これはOK
l.add(“hoge”);  // これはコンパイルエラー
l.add((Object)o);  // これもコンパイルエラー
```

  - List<? extends Point> l;
    - Pointかその子孫のリスト

参考文献

- Joshua Bloch, Effective Java 第3版, 丸善出版
- 谷本心ら、Java本格入門、技術評論社

ジェネリクスなどが気になったら読んでみよう

提出課題11: 点の重複排除

- 座標を表すPointクラスがある
- Pointクラスには、テスト用の点を出力するPoint.testPoint(int i)というメソッドがある
- testPoint(1)からtestPoint(10)までで得られる10点について、実質的にはいくつの点があるだろうか？重複を考慮し、2回以上現れる点は1回のみ表示するようにして、全部の点を表示しよう。
  - 「点が等しい」とは、座標の値が等しいこと
  - Point.toString()が用意されているので、異なる点を1行に1つずつ、すべての点についてprintln()する

要求

- PointTestクラスのmainを呼び出したら、
  - まず、相異なる点（同じ点が出てきたら2度目以降は無視した点）の総数を整数で表示し、
  - 次の行から相異なる点を1行に1つずつ表示する
    - 表示順序は問わない
- ソースファイルはPointTest.java
- 同じファイルにPointクラスを入れておくこと
  - Pointクラスにもコードを書き加える必要あり
- ひな形は配布してある

- 今回はコレクションの使い方の練習なので、そう思って解いてください

ヒント

- Pointクラスに、「等しい」ことの判断機能が必要
  - equals()ですね
- 重複を排除するときはコレクションを使うと簡単
  - 色々方法がある
- コレクションによってはhashCode()を使うので定義が必要
- Eclipseのひな形作成ツールを使っても良い
- コレクションを使うときはジェネリクスを使おう
- testPoint()の中身を読めば、答えはどうなるかわかるはず。よくチェックしよう。

提出物

- 提出物はPointTest.java
  - 先頭に「**組番号、名前**」と、出力された文字列をコメントで記入
    - 採点ミスを減らすための用心。ご協力ください。
  - Package javalec11　とする
  - PointTest.main()を呼び出したら課題の結果が表示されるようにする

- ㋈切は7/9(火) 17:00