データ構造とプログラミング：スタック・キュー

情報工学科 山本哲男

テストケースの実装

- テストケース1つにつき1メソッドで記述
- 1メソッドに複数の事項を記述するのはさけるべき
  - 通常のメソッドも同様

スタック

- データの挿入・削除をリストの先頭に限定したデータ構造
- LIFO：Last In First Out（最後に入ったものが最初に出る）

優先度の高い仕事が入ってきたらまず実行し、終わったら元の仕事に戻る

スタック（抽象データ型）

スタックを抽象データ型で表すと

- 値
  - あるデータ型Tの要素を０個以上並べたもの
- 操作
  - 挿入：スタックの先頭に要素を追加
  - 削除：スタックの先頭の要素を削除
  - 取得：スタックの先頭の要素を取得
  - 空判定：スタックが空か判定
  - ・・・

スタックの操作

- 挿入：要素xを追加

```
void push(E x)
```

- 削除：要素を削除し，取得

```
E pop()
```

- 取得：要素を取得するが削除しない

```
E peek()
```

- 空判定：空かどうか判定

```
boolean empty()
```


スタックのインタフェース

```
public interface Stack<E> {
    void push(E x);
    E pop() throws EmptyException;
    E peek() throws EmptyException;
    boolean empty();
}
```

スタックが空の場合例外をスローする

キュー

- データの挿入をリストの最後・取得と削除をリストの最初に限定したデータ構造
- FIFO：First In First Out（最初に入ったものが最初に出る）

リストにため込んだ順番に処理をする

キュー（抽象データ型）

スタックを抽象データ型で表すと

- 値
  - あるデータ型Tの要素を０個以上並べたもの
- 操作
  - 挿入：キューの最後に要素を追加
  - 削除：キューの先頭の要素を削除
  - 取得：キューの先頭の要素を取得
  - 空判定：キューが空か判定

・・・

キューの操作

- 挿入：要素xを追加

```
void enqueue(E x)
```

- 削除：要素を削除し，取得

```
E dequeue()
```

- 取得：要素を取得するが削除しない

```
E peek()
```

- 空判定：空かどうか判定

```
boolean empty()
```


キューのインタフェース

```
public interface Queue<E> {
    void enqueue(E x);
    E dequeue() throws EmptyException;
    E peek() throws EmptyException;
    boolean empty();
}
```

キューが空の場合例外をスローする

スタックとキューの実装

- すべて自前で実装する方式
- リストを利用して実装する方式
  - 合成集約を使用する方法
    - List型のインスタンス変数を利用
  - 継承を利用する方法
    - Listを実装しているクラスを継承して利用

クラス間の関連

- クラス間の関連（has-a関係）
  - 集約（アグリゲーション）
    - クラスが他のクラスの組み合わせで構成されている関係
  - 合成集約（コンポジション）
    - 含んでいる側が消滅すると含まれている側も消滅し，一方が消滅するとオブジェクトが機能しなくなるような関係
  - 集約でも合成集約でもない関連
    - インスタンス変数に他クラスのオブジェクトを保持しているが，集約や合成集約のような包含関係にない関連

合成集約によるスタックの実装

- pushやpopメソッドの処理は，list変数のオブジェクトに対してaddやremoveすることで実現

```
public class StackImpl<E> implements Stack<E> {
    // フィールド
    private List<E> list;
    ...
    // コンストラクタ
    public StackImpl() { listの初期化 }
    // インタフェースの実装
    public void push(E x) { ... }
    public E pop() throws EmptyException { ... }
    ...
}
```


継承

- 既存のクラスの性質を引き継いだ新しいクラスを定義すること
- 既存のクラスをスーパークラス（親クラス）と呼び，新しいクラスをサブクラス（子クラス）と呼ぶ
- スーパークラスの持つフィールドとメソッドをそのまま引き継ぐ

  ※アクセス修飾子に依存

アクセス修飾子

- private
  - サブクラスからアクセス不可
  - 任意のクラスからアクセス不可
- public
  - サブクラスからアクセス可
  - 任意のクラスからアクセス可
- protected
  - サブクラスからアクセス可
  - 任意のクラスからアクセス不可

継承によるスタックの実装

- pushやpopメソッドの処理は，スーパークラスで定義されているaddやremoveメソッドを呼び出すことで実現

```
public class StackImpl<E> extends LinkedList<E> implements Stack<E> {
    // コンストラクタ
    public StackImpl() { ... }
    // インタフェースの実装
    public void push(E x) { ... }
    public E pop() throws EmptyException { ... }
    ...
}
```

super

- Javaではコンストラクタの先頭で自動的に親クラスのコンストラクタが呼び出される
  - 引数を与えて呼び出したい場合等は明示的にsuper() を利用

```
class SuperClass {
  SuperClass() {
    ...
  }
  SuperClass(int a) {
    ...
  }
  ...
}
```

```
class SubClass extends SuperClass {
  SubClass(int a) {
//    super(); 明示しないとこのようになる
      super(a);
      ...
  }
  ...
}
```


継承の注意点

- 継承関係はis-a関係で利用するべき
  - クラスＡを拡張してクラスＢを作る前に，「すべてのＢは本当にＡであるか（is-a関係か）？」を確認
- 親クラスの実装に依存するため，親クラスの実装を熟知していないと罠にはまる可能性（依存が高い）
  - 親クラスのメソッドのうち必要のないメソッドまで継承してしまう

継承

- 親クラスとして振る舞うことが可能
- ポリモーフィズムにより実態のメソッドが呼び出される

```
class Animal {
  void Cry() {
    System.out.println("Cry");
  }
}
class Dog extends Animal {
  void Cry() {
    System.out.println("ワン");
  }
}
```

```
class Cat extends Animal {
  void Cry(){
    System.out.println("ニャー");
  }
}
class Test {
  public static void m(void) {
    Animal animal1 = new Dog();
    Animal animal2 = new Cat();
    animal1.Cry();
    animal2.Cry();
  }
}
```


文字列の分割

- Stringクラスにsplitメソッドが存在
  - 指定された正規表現に一致する位置で分割してくれる
- 正規表現を使いこなすことで様々な区切りを実現可能

\sは空白を表す特殊文字
一文字以上の空白なので\s+
\をエスケープしないといけないので\\s++

例）

```
String str[] = String.split("\\s+");
```

正規表現

- 文字列のパターンマッチによく使われる
- メタ文字を利用してマッチ

主なメタ文字

メタ文字	意味
.	任意の一文字
+	直前の文字の1個以上の繰り返し
*	直前の文字の0個以上の繰り返し
[]	括弧内に含まれる一文字にマッチ
^	行の最初にマッチ
$	行の最後にマッチ