CS 61B Spring 2018

Selecting ADTs

Discussion 6: February 19, 2018

1 Immutable Rocks

Access control allows us to restrict the use of fields, methods, and classes.

- `public`: Accessible by everyone.
- `protected`: Accessible by the class itself, the package, and any subclasses.
- *default (no modifier)*: Accessible by the class itself and the package.
- `private`: Accessible only by the class itself.

1.1 A class is immutable if nothing about its instances can change after they are constructed. Which of the following classes are immutable?

```
public class Pebble {
    public int weight;
    public Pebble() { weight = 1; }
}
```

```
public class Rock {
    public final int weight;
    public Rock (int w) { weight = w; }
}
```

```
public class Rocks {
    public final Rock[] rocks;
    public Rocks (Rock[] rox) { rocks = rox; }
}
```

```
public class SecretRocks {
    private Rock[] rocks;
    public SecretRocks(Rock[] rox) { rocks = rox; }
}
```

```
public class PunkRock {
    private final Rock[] band;
    public PunkRock (Rock yRoad) { band = {yRoad}; }
    public Rock[] myBand() {
        return band;
    }
}
```

```
public class MommaRock {
    public static final Pebble baby = new Pebble();
}
```

2 Breaking the System

2.1 Below is a flawed implementation of the stack ADT.

```
1  public class BadIntegerStack {
2      public class Node() {
3          public Integer value;
4          public Node prev;
5
6          public Node(Integer v, Node p) {
7              value = v;
8              prev = p;
9          }
10     }
11     public Node top;
12
13     public boolean isEmpty() {
14         return top == null;
15     }
16
17     public void push(Integer num) {
18         top = new Node(num, top));
19     }
20
21     public Integer pop() {
22         Integer ans = top.value;
23         top = top.prev;
24         return ans;
25     }
26     public Integer peek() {
27         return top.value;
28     }
29 }
```

(a) Exploit the flaw by filling in the `main` method below so that it prints "Success" by causing `BadIntegerStack` to produce a `NullPointerException`.

```
1  public static void main(String[] args) {
       try {
           BadIntegerStack stack = new BadIntegerStack();
           stack.pop();
       } catch (NullPointerException e) {
           System.out.println("Success");
       }
   }
```

(b) Exploit another flaw in the stack by completing the `main` method below so that the stack appears infinitely tall.

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

BadIntegerStack stack = new BadIntegerStack();
stack.push(1);
stack.top.prev = stack.top;
while (!stack.isEmpty()) {
stack.pop()
}
System.out.println("Unreachable");
}

(c) How can we change the `BadIntegerStack` class so that it won't throw `NullPointerException`s or allow ne'er-do-wells to produce endless stacks?

Node top → private

check that top is not null before pop

3 Design a Parking Lot!

3.1 Design a `ParkingLot` package that allocates specific parking spaces to cars in a smart way. Decide what classes you'll need, and design the API for each. Time permitting, select data structures to implement the API for each class. Try to deal with annoying cases (like disobedient humans).

- Parking spaces can be either regular, compact, or handicapped-only.
- When a new car arrives, the system should assign a specific space based on the type of car.
- All cars are allowed to park in regular spots. Thus, compact cars can park in both compact spaces and regular spaces.
- When a car leaves, the system should record that the space is free.
- Your package should be designed in a manner that allows different parking lots to have different numbers of spaces for each of the 3 types.
- Parking spots should have a sense of closeness to the entrance. When parking a car, place it as close to the entrance as possible. Assume these distances are distinct.