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?

```
1 public class Pebble {
2     public int weight;
3     public Pebble() { weight = 1; }
4 }
```

This class is mutable. `Pebble`'s weight field is public, and thus a pebble's state can easily be changed.

```
1 public class Rock {
2     public final int weight;
3     public Rock (int w) { weight = w; }
4 }
```

This class is immutable. `Rock`'s weight field is final, so it cannot be reassigned once a rock is initialized.

```
1 public class Rocks {
2     public final Rock[] rocks;
3     public Rocks (Rock[] rox) { rocks = rox; }
4 }
```

Though rocks cannot be reassigned, we can still change what the array holds and thus `Rocks` is mutable.

```
1 public class SecretRocks {
2     private Rock[] rocks;
3     public SecretRocks(Rock[] rox) { rocks = rox; }
4 }
```

The rocks variable is private, so no outside variable can reassign it or its elements. However, `rox` can be edited externally after it is passed in, so `SecretRocks` is technically mutable. This class can be made immutable by using `Arrays.copyOf`.

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

It is possible to access and modify the contents of `PunkRock`'s private array through its public `myBand()` method, so this class is mutable.

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

This class is mutable since `Pebble` has public variables that can be changed. For instance, given a `MommaRock mr`, you could mutate `mr` with `mr.baby.weight = 5`.

2 Breaking the System

2.1 Below is a flawed implementation of the stack ADT.

```
public class BadIntegerStack {
    public class Node() {
        public Integer value;
        public Node prev;

        public Node(Integer v, Node p) {
            value = v;
            prev = p;
        }
    }
    public Node top;

    public boolean isEmpty() {
        return top == null;
    }

    public void push(Integer num) {
        top = new Node(num, top));
    }

    public Integer pop() {
        Integer ans = top.value;
        top = top.prev;
        return ans;
    }
    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`.

```
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.

```
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("This print statement is 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?

Make `top` **`private`**.

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.

`public class Car`

- `public Car(boolean isCompact, boolean isHandicapped)`: creates a car with given size and permissions.
- `public boolean isCompact()`: returns whether or not a car can fit in a compact space.
- `public boolean isHandicapped()`: returns whether or not a car may park in a handicapped space.
- `public boolean findSpotAndPark(ParkingLot lot)`: attempts to park this car in a spot, returning true if successful.
- `public void leaveSpot()`: vacates this car's spot.

`private class Spot`

- The `Spot` class can be declared private and encapsulated by the `ParkingLot` class. Though it is private, and therefore not a part of our parking lot API, its methods are described here to give you an idea of how a `Spot` class might be implemented.
- `private Spot(String type, int proximity)`: creates a spot of a given type and proximity.
- `private boolean isHandicapped()`: returns true if this spot is reserved for handicapped drivers.
- `private boolean isCompact()`: returns true if this parking space can only accomodate compact cars.

`public class ParkingLot`

- `public ParkingLot(int[] handicappedDistances, int[] compactDistances , int[] regularDistances)`: creates a parking lot containing `handicappedDistances .length` handicapped spaces, each with a distance corresponding to an element of `handicappedDistances`. Also initializes the appropriate compact and regular spaces.
- `public boolean findSpotAndPark(Car toPark)`: attempts to find a spot and park the given car. Returns false if no spots are available.
- `public void removeCarFromSpot(Car toRemove)`: records when a spot has been vacated, and makes the spot available for parking again.

Prioritization of closeness in parking space selection can be handled using several priority queues (one for each kind of parking space). Occupied spaces (which are dequeued when they are assigned) can be tracked with a `Map<Car, Spot>`.