# CS 61B Discussion 7 Spring 2015

## 1 Breaking the System

Below is a bad implementation of a stack. Write a client class `Exploiter1` for which `BadIntStack` causes a `NullPointerException`. Then write a client class `Exploiter2` that creates an infinitely long stack. Note that these two classes are in the same default package as `BadIntStack`. Then make any changes necessary so that the BadIntStack will not throw `NullPointerExceptions` or allow evil tamperers to do mean things like `Exploiter2`.

```java
class SNode() {
    These package-private variables will not cause any problems if
       BadIntStack's top Node is private. Though it is still better practice
       to keep it private. Normally, other classes in the same directory
       will be placed in the same 'default package' and thus able to access
       package-private variables. '
    Integer val;
    SNode prev;
    public SNode(Integer v, SNode p) {
        val = v;
        prev = p;
    }
    public SNode(Integer v) {
        this(v, null);
    }
}

class BadIntStack {
    private SNode top;
    public boolean isEmpty() {
        return top == null;
    }
    public void push(Integer num) {
        top = new SNode(num, top);
    }
    public Integer pop() {
        if (top == null) {
            return null; // Or throw a meaningful exception. Like
                EmptyStackException.
        }
        Integer ans = top.val;
        top = top.prev;
        return ans;
    }
    public Integer peek() {
        if (top == null) {
            return null; // Or throw a meaningful exception. Like
                EmptyStackException.
        }
        return top.val;
    }
}
```

```java
public class Exploiter1 {
    public static void main(String[] args) {
         BadIntStack b = new BadIntStack();
         b.pop();
    }
}

public class Exploiter2 {
    public static void main(String[] args) {
         BadIntStack b = new BadIntStack();
         b.push(1);
         b.top.prev = b.top;
         while(!b.isEmpty()) {
           b.pop();
         }
    }
}
```

## 2 Immutable Rocks

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

```
//Pebbles weight variable is public and thus its state can easily be changed.
public class Pebble {
    public int weight;
    public Pebble() { weight = 1; }
}

//Rock's weight cannot be reassigned once it is initialized.
public class Rock {
    public final int weight;
    public Rock (int w) { weight = w; }
}

The rocks variable can still change what elements it holds and thus the class
    is mutable.
public class Rocks {
    public final Rock[] rocks;
    public Rocks (Rock[] rox) { rocks = rox; }
}

The rocks variable here is private so no outside classes can reassign it or
    its elements. However, it is still considered mutable due to the ability
    to change the rox argument you passed in. To make it truly immutable, use
    Arrays.copyOf.
public class SecretRocks {
    private Rock[] rocks;
    public SecretRocks(Rock[] rox) { rocks = rox; }
}

Getter method allows you to access the private array and thus change its
    contents.
public class PunkRock {
    private final Rock[] band;
    public PunkRock (Rock yRoad) { band = {yRoad}; }
    public Rock[] myBand() {
        return band;
    }
}

This class is mutable since Pebble has public variables that can be changed.
    For instance, you could write mr.baby.weight = 5; (given mr is a
    MommaRock).
public class MommaRock {
    public static final Pebble baby = new Pebble();
}
```

## 3 DIY: Design a Parking Lot

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.

`Results may vary. One valid solution to be posted.`