1 Immutable Rocks

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  }
5  /* This class is mutable. Pebble's weight field is public, and thus a
      pebble's state can easily be changed. */
6  public class Rock {
7      public final int weight;
8      public Rock (int w) { weight = w; }
9  }
10 /* This class is immutable. Rock's weight field is final, so it cannot be
      reassigned once a rock is initialized. */
11 public class Rocks {
12     public final Rock[] rocks;
13     public Rocks (Rock[] rox) { rocks = rox; }
14 }
15 /* Though rocks cannot be reassigned, we can still change what the array
      holds and thus Rocks is mutable. */
16 public class SecretRocks {
17     private Rock[] rocks;
18     public SecretRocks(Rock[] rox) { rocks = rox; }
19 }
20 /* 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. */
21 public class PunkRock {
22     private final Rock[] band;
23     public PunkRock (Rock yRoad) { band = {yRoad}; }
24     public Rock[] myBand() {
25         return band;
26     }
27 }
28 /* It is possible to access and modify the contents of PunkRock's private
      array through its public myBand() method, so this class is mutable. */
29 public class MommaRock {
30     public static final Pebble baby = new Pebble();
31 }
32 /* 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

Below is the `SNode` class, along with a flawed implementation of the Stack ADT, `BadIntStack`.

```java
public class SNode() {
        /* These public variables will not cause any problems assuming
           BadIntStack's top Node is private. However, it is still better
           practice to declare instance variables like prev and val as
           private. */
    public Integer val;
    public SNode prev;

    public SNode(Integer v, SNode p) {
        val = v;
        prev = p;
    }
    public SNode(Integer v) {
        this(v, null);
    }
}
```

```java
public 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 an
               EmptyStackException). */
        }
        Integer ans = top.val;
        top = top.prev;
        return ans;
    }
    public Integer peek() {
        if (top == null) {
            return null; /* Or throw a meaningful exception. */
        }
        return top.val;
    }
}
```

Fill in the `Exploiter1` class so that it prints *"Success!"* by causing `BadIntStack` to produce a `NullPointerException`.

```
class Exploiter1 {
    public static void main(String[] args) {
        try {
            // Your exploit here!
            BadIntStack bad = new BadIntStack();
            b.pop();
        } catch (NullPointerException e) {
            System.out.println("Success!");
        }
    }
}
```

Now, fill in the client class `Exploiter2` so that it creates an "infinitely long" stack.

```
class Exploiter2 {
    public static void main(String[] args) {
        BadIntStack trap = new BadIntStack();
        // Your exploit here!
        trap.push(1);
        trap.top.prev = trap.top;
        while(!trap.isEmpty()) {
            trap.pop();
        }
        System.out.println("This print statement is unreachable!");
    }
}
```

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

3 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.

```
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.
```

```
/* A note on ADTs: 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 of Cars to Spots. */
```