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`.

```
1  class SNode() {
2      Integer val;
3      SNode prev;
4
5      public SNode(Integer v, SNode p) {
6          val = v;
7          prev = p;
8      }
9      public SNode(Integer v) {
10         this(v, null);
11     }
12 }
13
14 class BadIntStack {
15     SNode top;
16
17     public boolean isEmpty() {
18         return top == null;
19     }
20     public void push(Integer num) {
21         top = new SNode(num, top));
22     }
23     public Integer pop() {
24         Integer ans = top.val;
25         top = top.prev;
26         return ans;
27     }
28     public Integer peek() {
29         return top.val;
30     }
31 }
```

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.

```
1  public class Pebble {
2      public int weight;
3      public Pebble() { weight = 1; }
4  }
5  public class Rock {
6      public final int weight;
7      public Rock (int w) { weight = w; }
8  }
9  public class Rocks {
10     public final Rock[] rocks;
11     public Rocks (Rock[] rox) { rocks = rox; }
12 }
13 public class SecretRocks {
14     private Rock[] rocks;
15     public SecretRocks(Rock[] rox) { rocks = rox; }
16 }
17 public class PunkRock {
18     private final Rock[] band;
19     public PunkRock (Rock yRoad) { band = {yRoad}; }
20     public Rock[] myBand() {
21         return band;
22     }
23 }
24 public class MommaRock {
25     public static final Pebble baby = new Pebble();
26 }
```

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.