Immutability & Encapsulation

CS 61B Spring 2016: Discussion 6

Announcements

Homework 1 due Friday 2/26

Project 2 due Monday 3/7

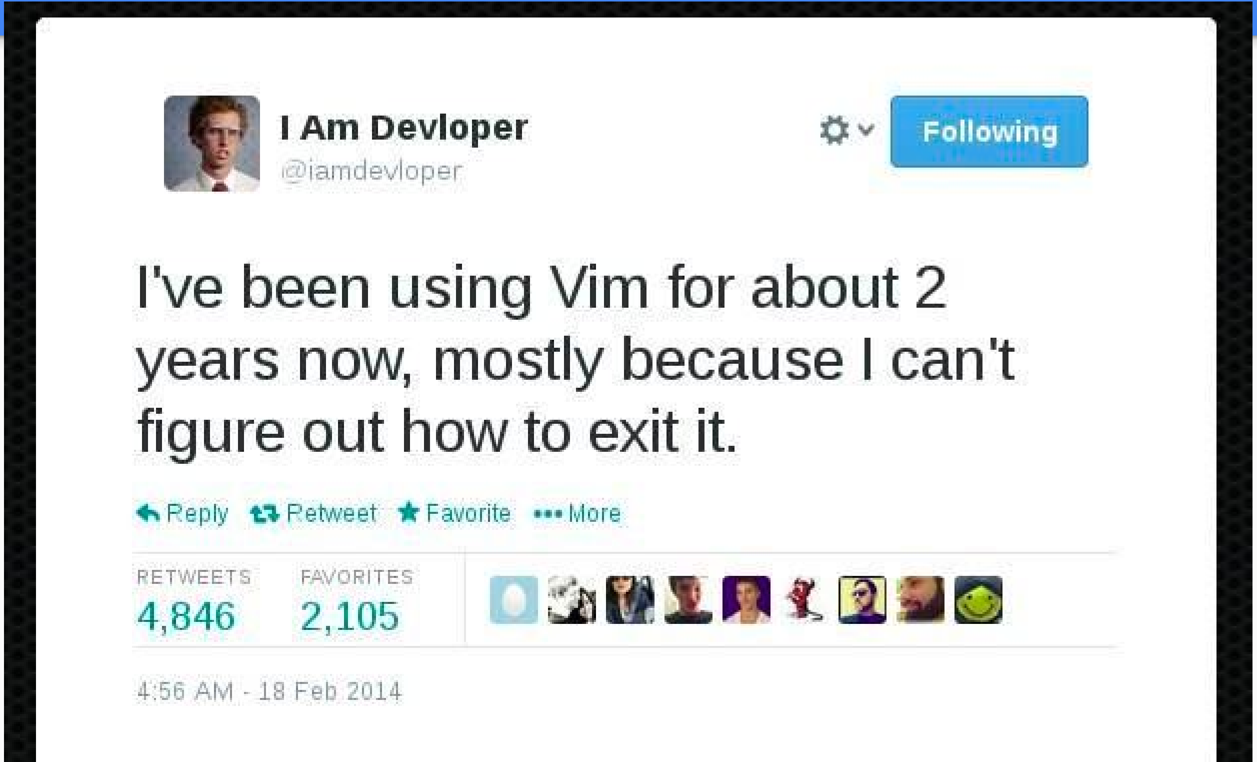
Lab 6 is out if you want to finish it early

UNIX Tip of the Day: alias

Example:
```
$ alias c=”javac *.java”
$ c     # does the same thing as javac *.java
```

More details: https://piazza.com/class/iiklg7j9ggf2vl?cid=2396

A recent tweet that surfaced on Facebook



I Am Devloper
@iamdevloper

I've been using Vim for about 2 years now, mostly because I can't figure out how to exit it.

RETWEETS 4,846 FAVORITES 2,105

4:56 AM - 18 Feb 2014

Immutability

An immutable data type is one for which an instance cannot change in any observable way after instantiation.

Mutable examples: LinkedListDeque, Planet

Immutable examples: Integer, String

Immutability

In Java, we can help ensure immutability using the keyword **final**.

A **final variable** is assigned once; after that, it cannot be reassigned.

```
public final int x = 5;

x = 6; // can’t do this!
```

Immutability

However, just declaring a variable **final** does not make it necessarily immutable. It just helps.

This is commonly the case for objects (references) that are declared **final**, as opposed to primitives.

```
public final ArrayList<Integer> list = new ArrayList<Integer>();
```

The variable `list` can never point to any different `ArrayList`, but the `ArrayList` it references can be modified (e.g. call `add()`, `remove()`, etc.).

Your turn! (Problem 1)

Problem 1

Pebble

Mutable. The weight field is public and not declared final, so we can easily reassign its value.

Problem 1

Rock

Immutable. The weight field is final, so it cannot be reassigned after the rock is initialized.

Problem 1

Rocks

Mutable. Even though the rocks[] array is final and the variable itself cannot be assigned, we can still change the values within the array.

Problem 1

SecretRocks

Mutable. The rocks variable is private, so it cannot be accessed outside of the class. However, the argument to the constructor, rox, is passed in externally and thus could be edited externally, thereby also mutating the values in rocks.

Problem 1

PunkRock

Mutable. It is possible to access and modify the contents of the Rock[] band, since the PunkRock class has a public method that returns a reference to band.

Side note: This kind of public method that allows “safe” access to private variables is often called a “getter” or “accessor” method.

Problem 1

MommaRock

Mutable. The Pebble instance within MommaRock has public variables that can be changed, as we discussed before. For example, we could do something like:

```
MommaRock mr = new MommaRock();

mr.baby.weight = 5;
```

Onto Problem 2...

Problem 2

```
class Exploiter1 {
    public static void main(String[] args) {
        try {
            // Your exploit here!

        }  catch (NullPointerException e) {
            System.out.println("Success!");
        }
    }
}
```

Problem 2

```
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!");
        }
    }
}
```

Problem 2

```java
class Exploiter2 {
    public static void main(String[] args) {
        BadIntStack trap =  new BadIntStack();
        // Your exploit here!

        while(!trap.isEmpty()) {
            trap.pop();
        }
        System.out.println("This print statement is unreachable!");
    }
}
```

Problem 2

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

Problem 2

How can we prevent these exploits from happening?

1. Check if top is null in both pop() and peek() before we actually try to access top.
2. Make the top variable within BadIntStack private, so that we don’t have direct access to it, and therefore could not re-arrange the pointers to produce the “infinite stack”.

And lastly, Problem 3...

Problem 3

Talk to your neighbors about what kinds of classes you might want to have.

Think purely in roles: what should each class be responsible for doing or keep track of?

Then, once you've got some ideas for roles, think about what types of instance variables or data structures each class might need in order to best perform its role.

Side note: API means Application Programming Interface, which is a fancy word for what kinds of classes and their respective methods are available from a provided package for use by other programmers.

Problem 3 - Sample Solution

Classes we might have:

- ParkingLot
    - int numRegSpots, int numCompSpots, int numHandiSpots, PriorityQueues for every type of Spot to get the next best Spot to park a Car
    - findSpotAndPark(Car c), freeSpot(Spot sp)
- Car
    - boolean isCompact, boolean isHandicapped, ParkingLot parkingLot, Spot parkingSpot
    - findSpotAndPark(ParkingLot lot), leaveSpot()
- Spot implements Comparable
    - String type, int proximity, Car currCar
    - parkCar(Car c), clearSpot(), compareTo(Spot other)