CS 61B Discussion 6 Spring 2017

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?

```java
public class Pebble {
    public int weight;
    public Pebble() { weight = 1; }
}
/* This class is mutable. Pebble's weight field is public, and thus a
   pebble's state can easily be changed. */
public class Rock {
    public final int weight;
    public Rock (int w) { weight = w; }
}
/* This class is immutable. Rock's weight field is final, so it cannot be
   reassigned once a rock is initialized. */
public class Rocks {
    public final Rock[] rocks;
    public Rocks (Rock[] rox) { rocks = rox; }
}
/* Though rocks cannot be reassigned, we can still change what the array
   holds and thus Rocks is mutable. */
public class SecretRocks {
    private Rock[] rocks;
    public SecretRocks(Rock[] rox) { rocks = rox; }
}
/* 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 Assorted ADTs

Below are some sketches of ADTs (not real Java code). It's not important to understand the details of how these work right now; just try to understand how each one can be used conceptually.

```
List {
    insert(item, position);       // inserts item into the list at the position
    get(position);              // returns the item in the list at the position
    size();                          // returns the number of items in the list
}

Set {
    add(item);                  // puts item in the set. Does not add duplicates
    contains(item);             // returns whether or not the item is in the set
    items();              // returns a List of all items in some arbitrary order
}

Stack {
    push(item);                                       // puts item onto the stack
    pop();                       // removes and returns the most recently put item
    isEmpty();                                 // returns whether the stack is empty
}

Queue {
    enqueue(item);                                    // puts item into the queue
    dequeue();                  // removes and returns the least recently put item
    isEmpty();                                 // returns whether the queue is empty
}

PriorityQueue {
    enqueue(item, priority);       // puts item into the queue with a priority
    dequeue();             // removes and returns the item with highest priority
    peek();       // returns but does not remove the item with highest priority
}

Map {                                          // like a dictionary from python
    put(key, value);       /* puts key into the map and associates it with the
       given value. If key is already in the map, replaces its existing
       value with the given value */
    get(key);                               // returns value associated with key
    keys();              // returns a List of all keys in some arbitrary order
}
```

3 Solving Problems with ADTs

Consider the problems below. Which of the ADTs given in the previous section might you use to solve each problem? Although in principle any of the ADTs might be used to solve any of the problems, think about which ones will make code implementation easier or more efficient.

1. Given a news article, find the frequency of each word used in the article.

```
Use a map. When you encounter a word for the first time, put the key
    into the map with a value of 1. For every subsequent time you
    encounter a word, get the value, and put the key back into the map
    with its value you just got, plus 1.
```

2. Given an unsorted array of integers, return the array sorted from least to greatest.

```
Use a priority queue. For each integer in the unsorted array, enqueue
    the integer with a priority equal to its value. Calling dequeue will
    return the largest integer; therefore, we can insert these values
    from index length-1 to 0 into our array to sort from least to
    greatest.
```

3. Implement the forward and back buttons for a web browser.

```
Use two stacks, one for each button. Each time you visit a new web page,
    add the previous page to the back button's stack. When you click the
    back button, add the current page to the forward button stack, and
    pop a page from the back button stack. When you click the forward
    button, add the current page to the back button stack, and pop a page
    from the forward button stack. Finally, when you visit a new page,
    clear the forward button stack.
```

4 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. */
```