CS 61B Discussion 6: Test Review Spring 2020

1 Inheritance Practice

```
public class Q {
   public void a() {
      System.out.println("Q.a");
   }
   public void b() {
      a();
   }
   public void c() {
      e();
   }
   public void d() {
      e();
   }
   public static void e() {
      System.out.println("Q.e");
   }
}

public class R extends Q {
   public void a() {
      System.out.println("R.a");
   }
   public void d() {
      e();
   }
   public static void e() {
      System.out.println("R.e");
   }
}

public class S {
   public static void main(String[] args) {
      R aR = new R();
      run(aR);
   }
   public static void run(Q x) {
      x.a();      /* Output: ____________________ */
      x.b();      /* Output: ____________________ */
      x.c();      /* Output: ____________________ */
      ((R)x).c(); /* Output: ____________________ */
      x.d();      /* Output: ____________________ */
      ((R)x).d(); /* Output: ____________________ */
   }
}
```

In `run`, write what gets printed next to each line when it is called from `main`.

2 Reduce

We'd like to write a method `reduce`, which uses a `BinaryFunction` interface to accumulate the values of a `List` of integers into a single value. `BinaryFunction` can operate (through the `apply` method) on two integer arguments and return a single integer. Note that `reduce` can now work with a range of binary functions (for example, addition and multiplication). Write two classes `Adder` and `Multiplier` that implement `BinaryFunction`. Then, fill in `reduce` and `main`, and define types for `add` and `mult` in the space provided.

```
import java.util.ArrayList;
import java.util.List;
public class ListUtils {
    /** If the list is empty, return 0.
     *  If it has one element, return that element.
     *  Otherwise, apply a function of two arguments cumulatively to the
     *  elements of list and return a single accumulated value.
     *  Does not modify the list. */
    public static int reduce(BinaryFunction func, List<Integer> list) {



    }
    public static void main(String[] args) {
        ArrayList<Integer> integers = new ArrayList<>();
        integers.add(2); integers.add(3); integers.add(4);
        ___________ add = ____________________;
        ___________ mult = ____________________;
        reduce(add, integers); // Should evaluate to 9
        reduce(mult, integers); // Should evaluate to 24
    }
}

interface BinaryFunction {
    int apply(int x, int y);
}

// Add additional classes below:
```

3 Interleaving IntLists

Implement `interleave(IntList A, IntList B)` so that it returns an IntList whose contents are the result of interleaving IntLists A and B, beginning with the the first item in A if possible. This method should interleave the items in-place and should therefore be destructive. For example, if A is (1 -> 3 -> 5 -> 7) and B is (2 -> 4), then calling `interleave(A, B)` should return the list (1 -> 2 -> 3 -> 4 -> 5 -> 7). Because this process is destructive, both A and B may become modified in the process. A and B are not guaranteed to be the same length and may be null.

```
public IntList interleave(IntList A, IntList B) {
   if (A == null) {
      return B;
   } else if (B == null) {
      return A;
   }

   IntList curr = A;
   IntList other = B;
   IntList save;

   // Add code here















    return A;
}
```

4 Inheritance Infiltration

Access modifiers are critical when it comes to security. Look at the PasswordChecker and User classes below.

```
public class PasswordChecker {
    /** Returns true if the provided login and password are correct. */
    public boolean authenticate(String login, String password) {
        // Does some secret authentication stuff...
    }
}

public class User {
    private String username;
    private String password;

    public void login(PasswordChecker p) {
        p.authenticate(username, password);
    }
}
```

Even though the username and password variables are private, the login and authenticate methods are both public. We can use inheritance to take advantage of this and extract the password of any given User object. Complete the PasswordExtractor class below so that calling extractPassword returns the password of a given User. You may not modify the provided classes (i.e. you may not change the implementations of PasswordChecker or User).

```
public class PasswordExtractor extends ____________________ {
   String extractedPassword;

   public String extractPassword(User u) {





   }

   // Are there any other methods that we need to implement?









}
```

Hint: The login method of User passes in the username and password fields as parameters to the authenticate method of a given PasswordChecker. Think about how we can take advantage of method overriding to gain access to the password.