Question 1. (16 points) Equality revisited. As we know, equality for subclasses is hard to get right. We'd like to explore the issues involved in this question.

Suppose we have the following class that represents a cup that holds liquids:

```
public class Cup {
  protected final int volume;
  protected final String color;
  protected Liquid liquid;      // contents of this cup

  public Cup(int volume, String color, Liquid liquid) {
    this.volume = volume;
    this.color = color;
    this.liquid = liquid;
  }

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Cup)) {
      return false;
    }
    Cup c = (Cup) o;
    return this.volume==c.volume && this.color.equals(c.color) &&
      this.liquid.equals(c.liquid);
  }
}
```

We now want to add a new class, `Mug`, that is a `Cup` with a distinctive handle shape:

```
public class Mug extends Cup {
   private final Shape handle; // the shape of this mug's handle

   public Mug(int volume, String color, Liquid liquid,
              Shape handle) {
      super(volume, color, liquid);
      this.handle = handle;
   }

   @Override
   public boolean equals(Object o) { ... }
}
```

You should assume that classes `Liquid` and `Shape` contain correct implementations `equals` for those classes.

Do not remove this page from the exam, but feel free to tear off the copy of this page at the end of the exam. Continue with questions about these classes on the next page.

Question 1. (cont.) (16 points, 4 each) Here are four possible implementations of method `equals` for class `Mug`. For each one, you should determine whether it satisfies the specification for an `equals` method (it must be reflexive [a=a], symmetric [a=b implies b=a], transitive [a=b and b=c implies a=c], and properly handle `null` as an argument). If the method satisfies these properties, circle OK. If it does not satisfy some property, circle **all** of the properties that are not satisfied. It *doesn't* matter whether the method computes a sensible value for `equals` – just whether the method satisfies the specification for `equals`. The methods do compile successfully with no reported errors.

(a)
```
public boolean equals(Object o) {
      return true;
  }
```

OK REFLEXIVE (SYMMETRIC) TRANSITIVE (NULLArgument)

(b)
```
public boolean equals(Object o) {
    if (!(o instanceof Cup)) {
      return false;
    } else if (!(o instanceof Mug)) {
       return super.equals(o);
    } else {
       Mug m = (Mug) o;
     return super.equals(o) && this.handle.equals(m.handle);
   }
 }
```

OK REFLEXIVE SYMMETRIC (TRANSITIVE) NULLArgument

(c)
```
public boolean equals(Object o) {
    if (!(o instanceof Mug)) {
      return false;
    }
    Mug c = (Mug) o;
    return this.volume == c.volume && this.color.equals(c.color)
           && this.liquid.equals(c.liquid);
 }
```

OK REFLEXIVE (SYMMETRIC) (TRANSITIVE) NULLArgument

(d)
```
public boolean equals(Object o) {
   if (this == null && o == null) {
     return true;
   }
   return this == o;
 }
```

OK REFLEXIVE (SYMMETRIC) (TRANSITIVE) NULLArgument

Note: many of the problems arise when comparing combinations of `Mug` and `Cup` objects.

Question 2. (10 points, 1 each) Subclasses and generics – a traditional, annoying, but apparently necessary question. Let's assume that we have the `Cup` and `Mug` classes from the previous page and an additional subclass of `Cup` (*not* a subclass of `Mug`) named `TeaCup`.

```
public class Cup { ... }
public class Mug extends Cup { ... }
public class TeaCup extends Cup { ... }
```

Assume we have the following variables declared:

```
Object o; Cup c; Mug m; TeaCup t;

List<? extends Cup> lec;
List<? super Cup> lsc;
List<?> lq;
```

For each of the following, circle OK if the statement has correct Java types and will compile without type-checking errors; circle ERROR if there is some sort of type error.

OK (ERROR) `lec.add(c);`

OK (ERROR) `lq.add(o);`

(OK) ERROR `lsc.add(c);`

(OK) ERROR `lsc.add(t);`

(OK) ERROR `lq.add(null);`

(OK) ERROR `o = lq.get(0);`

OK (ERROR) `c = lq.get(0);`

(OK) ERROR `c = lec.get(0);`

OK (ERROR) `m = lec.get(0);`

OK (ERROR) `c = lsc.get(0);`

A friend of yours is opening a new Café and they hired a Coders 'R Us® graduate to write a computer system to manage orders. When a customer places an order, the system is supposed to keep track of the customer name and what they've ordered. The code is incomplete, but here's what was delivered before the Coders graduate left for a new job. (It does compile without any reported errors.)

```
public class OrderQueue {
  private final List<String> names; // customer names
  private int size;                 // # pending orders
  // maps customer names to their orders
  private final Map<String, List<String>> orders;

  public OrderQueue(){
    names = new ArrayList<String>();
    size = 0;
    orders = new HashMap<String, List<String>>();
  }

  public OrderQueue(OrderQueue other) {
    this.names = other.names;
    this.size = other.size;
    this.orders = other.orders;
  }

  public void placeOrder(String name, List<String> contents) {
    names.add(name);
    orders.put(name, contents);
    size++;
  }

  public List<String> getNames() { return names; }

  public int getSize() { return size;}

  public Map<String, List<String>> getOrders() {
    return Collections.unmodifiableMap(orders);
  }

  public List<List<String>> getAllOrders() {
    List<List<String>> result = new ArrayList<List<String>>();
    for (String name : orders.keySet()) {
      List<String> order = orders.get(name);
      result.add(order);
    }
    return result;
  }
}
```

Do not remove this page from the exam, but feel free to tear off the copy of this page at the end of the exam. Continue with questions about this code on the next pages.

Question 3. (12 points) Class specification. Because we're behind schedule and over budget, we're going to have to use the existing code. Before going further we need to add some documentation so we have a better idea of how the class works. You should base your answers on the apparent intended behavior in the original code and comments.

(a) (4 points) Give an appropriate abstract description of the class that should appear in the JavaDoc comment right before the first line of the class.

```
/**
 *  An OrderQueue is a collection of orders o1, ..., on.
 *  Each order is a pair <name, items> where name is the
 *  name of the person placing the order and items is
 *  a list of the items in the order.
 *
 *  No two orders in an OrderQueue may have the same name.
 *
 *
 *
 */
public class OrderQueue { ... }
```

(b) (4 points) Give a suitable representation invariant for this class. You should use the existing instance variables that are already present in the code (i.e., you should use the rep that is already there).

`names` and `orders` may not be `null`.
Each entry in `names` is a key in the map `orders`, and each key in orders appears in `names`, i.e., the set of strings in `names` contains the same items as the set of strings in the of `orders` keyset. (That implies that `names` may not contain any duplicate entries.)
Variable `size` = `names.size()` = `orders.size()`.

(c) (4 points) Give a suitable abstraction function for this class. Your answer should use information from your answers to parts (a) and (b) of the question as needed.

Each <key, value> pair in `orders` corresponds to one order o_i in the `OrderQueue`. The key in each pair is a name associated with an order, and the value of the pair is the list of items in that order. The list `names` holds a second copy of all of the names that are keys in the `orders` map, stored in the sequence that the orders were added to the `OrderQueue`. Variable `size` contains the number of items in the `OrderQueue`, which is the same as the number of entries in both `orders` and `names`.

Question 4. (14 points, 2 each) Rep exposure. We want to be sure there are no rep exposure problems in this code. For each of the following methods in the existing code, circle OK if it does not create a rep exposure problem or circle PROBLEM if it does. If you circle PROBLEM, give a brief explanation of what the problem is. If you circle OK you do not need to explain your answer further.

(a) `public OrderQueue(){ ... }` (OK) PROBLEM
Explanation if problem:

(b) `public OrderQueue(OrderQueue other){ ... }` OK (PROBLEM)
Explanation if problem:

Shares fields with `other` object. Any change to either object affects the other one.

(c) `public void placeOrder(String name, List<String> contents{ ...}`
Explanation if problem: OK (PROBLEM)

Shares a list (order contents) with the caller. Even though `Strings` are immutable, the caller can modify the `List` referenced by `OrderQueue`.

(d) `public List<String> getNames(){ ... }` OK (PROBLEM)
Explanation if problem:

Returns a reference to the `names List` to the caller, which the caller could modify.

(e) `public int getSize() { ... }` (OK) PROBLEM
Explanation if problem:

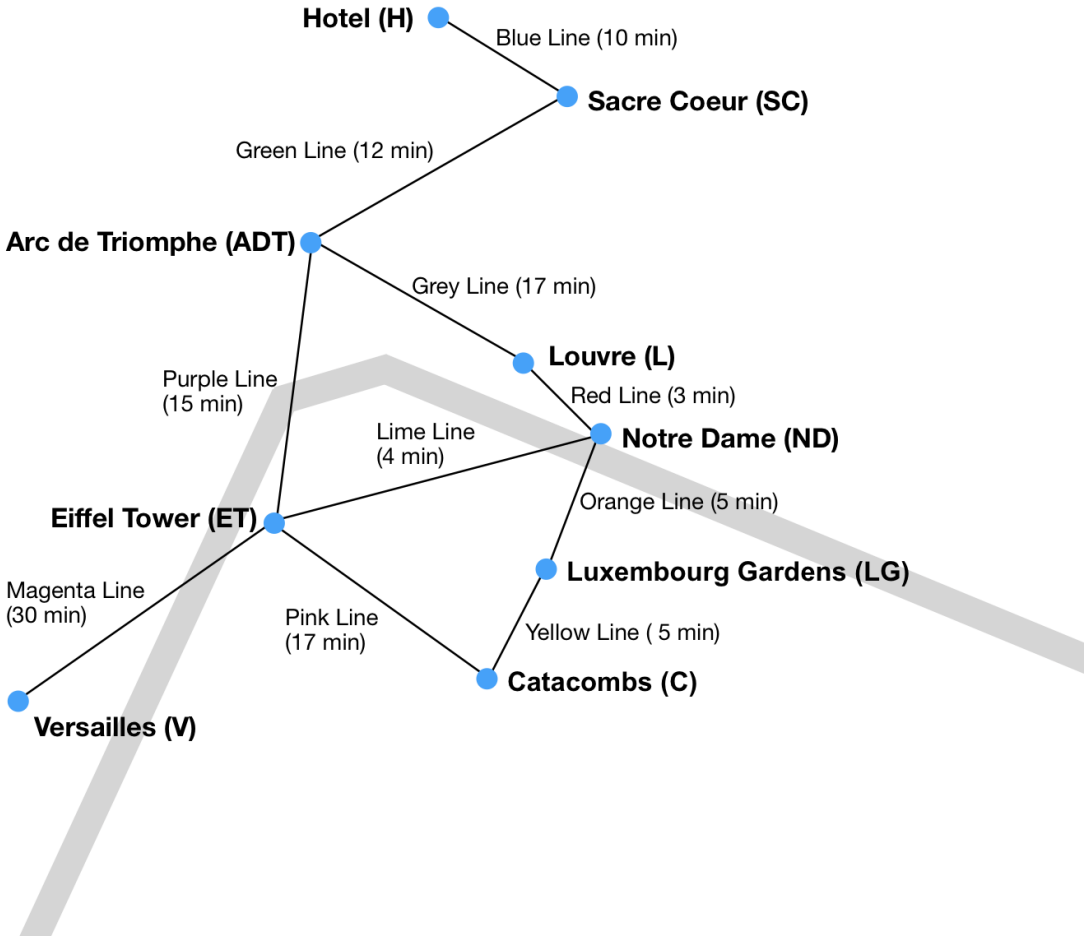
(f) `public Map<String, List<String>> getOrders(){ ... }`
Explanation if problem: OK (PROBLEM)

Although the caller cannot modify the returned `Map` directly, it can modify the List values in the `Map`'s <key,value> pairs.

(g) `public List<List<String>> getAllOrders(){ ... }`
Explanation if problem: OK (PROBLEM)

The returned value is a list of references to the same `Lists` of order items stored in the `OrderQueue`. These could be used to change the contents of any order currently in the `OrderQueue`.

Question 5. (18 points, 9 each) Graph searching. Now that the quarter is over we're going to take a trip! To Paris!! To help us plan, a friend has sent us a simplified map of the Paris Metro showing some of the main stops we'll be using. Here it is:



The map, of course, is a graph where the nodes are Metro stops and the edges are labeled with Metro line names and travel times between stops. We would like to use our knowledge of graph search algorithms to discover paths in the graph. Answer the questions on the next page.

Do not remove this page from the exam, but feel free to tear off the copy of this page at the end of the exam.

Question 5. (cont.) (a) Find the path with fewest number of transfers (i.e., fewest number of intermediate stops) between your Hotel (H) and the Luxembourg Gardens (LG) and determine how much time it takes. Indicate the algorithm used to find your answer. Fill in your answers below. If there are ties when computing the path, you should pick the "lexicographically least path" (i.e., use alphabetical ordering).

(i) Algorithm used: **BFS** Total travel time: **59 min.**

(ii) Path from H to LG with fewest transfers (show the line used – Blue, Green, etc. – and travel time for each edge):

H to SC via Blue Line, time 10 min

SC to ADT via line Green Line, time 12 min

ADT to ET via line Purple Line, time 15 min

ET to C via line Pink Line, time 17 min

C to LG via line Yellow Line, time 5 min

(b) Find the fastest path (minimum travel time) from Sacre Coeur (SC) to Catacombs (C). If there are two or more paths with the same minimum time, write down one of them. As above, indicate the algorithm used and then show the path and total time.

(i) Algorithm used: **Dijkstra's** Total travel time: **41 min.**

(ii) Fastest path from SC to C (show the line used – Blue, Green, etc. – and travel time for each edge):

SC to ADT via Green Line, time 12 min

ADT to ET via Purple Line, time 15 min

ET to ND via line Lime Line, time 4 min

ND to LG via line Orange Line, time 5 min

LG to C via line Yellow Line, time 5 min

Question 6. (18 points) A little bit of React. We've found the following sample of a React application, and we're trying to understand how it works. Here is the code; answer questions about it on the next page. Some import/export statements in the code are not shown. Note that the button text is just the literal character '+' – it does not represent string concatenation or some other operation.

```
class ShoppingCart extends Component {

  render() {
    return (
       <div>
          <Item name="Strawberry" price={1.00}/>
          <Item name="Milk" price={5.00}/>
       </div>
    );
  }
}

class Item extends Component {

  constructor(props) {
    super(props);
    this.state = {
       quantity: 1
    };
  }

  increment = () => {
    this.setState((oldState) => {
      return {
        quantity: oldState.quantity + 1
      };
    });
  };

  render() {
    let totalPrice = this.state.quantity * this.props.price;
    return (
      <p>
        <button onClick={this.increment}>+</button>
        {this.state.quantity}x {this.props.name}: ${totalPrice}
      </p>
    );
  }
}
```

Do not remove this page from the exam, but feel free to tear off the copy of this page at the end of the exam. Continue with questions about this code on the next page.

Question 6. (cont.) (a) (14 points) When the following index.html and index.js files are loaded into a web browser, what is the final HTML page that is generated in the browser? (i.e., what HTML would you see if you used the Chrome developer tools to look at the page?) For simplicity, do not include attributes, just include the tags themselves and their content as text.

index.html:

```
<html>
  <body>
    <div id="root"></div>
 </body>
</html>
```

index.js:

```
ReactDOM.render(<ShoppingCart />,
                document.getElementById('root'));
```

```
<html>
  <body>
    <div id="root">
      <div>
        <p> <button>+</button> 1x Strawberry: $1 </p>
        <p> <button>+</button> 1x Milk: $5 </p>
      </div>
    </div>
  </body>
</html>
```

(b) (4 points) The `increment` function causes the current quantity of an element to increase by one, so users can add more than one item to their cart. It uses the alternative version of `setState` that has a function parameter. That function receives, as its parameter, the most recent state object that was passed in a call to `setState`, regardless of the component lifecycle. Why is it necessary to use this version of `setState`, instead of the regular `setState` and `this.state.quantity + 1`?
Hint: Consider what might happen if a user presses the button twice, very, very quickly. Describe how it's possible for two quick presses of the button to only cause the quantity to increase once, and how the alternate version of `setState` prevents this.

`setState` is always a request to React to update the state sometime in the future. If we used the "regular" version and the button was pressed twice very quickly, it's possible that the value inside `this.state.quantity` hasn't updated from the first press, meaning the second press won't update the quantity value correctly. The alternate version, however, is guaranteed to give us the most recent `setState` call's value, which means our calculation will be accurate.

Question 7. (10 points, 2 each) Design Patterns. Listed below are a number of design problems that we've encountered in projects in this course. For each situation, we used a specific design pattern to solve the problem. Circle the name of the pattern we used in each situation. (There is a correct answer for each question.)

(a) To get a reference object that we could use to keep track of the `<canvas>`, we called `React.createRef()`.

Singleton (Factory Method) Adapter Iterator

(b) To create a `CSVToBean` object, we created another helper object and called multiple methods on it to configure the `CSVToBean` object, finally calling a method that returns an actual `CSVToBean` with all of our configurations.

Iterator Factory Method (Builder) Composite

(c) To allow the code in the provided text interface for the pathfinder application, which is the same for all students, to interact with everyone's different graph and pathfinding implementations, we used the `ModelConnector` object.

(Adapter) Composite Iterator Strategy

(d) We implemented multiple path finding algorithms (BFS, Dijkstra) separate from the implementation of the Graph that they operated on, to allow each individual use-case to choose the correct algorithm for the job.

Factory Method Adapter (Strategy) Prototype

(e) Each React component can determine its own "final look" by overriding the `render()` function, and components that contain other components use their children's `render()` functions to aid in calculating the result of their own `render()`.

Prototype Builder (Composite) Iterator

Question 8. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

Draw a picture of something that you plan to do during the rest of the summer!

Congratulations from the CSE 331 staff!
Have a great vacation and see you in the fall!!