CSC 216 Programming Concepts Java

Sample Test Answers

Answers to Sample Test Exercises.

Interfaces, Inheritance, Polymorphism

1.
   a. **I am a fish.**
      I am a goldfish.
   b. It is not legal because any call to a super constructor must be the first statement in the constructor. To correct the code, either remove the call to **super()** or move the call above the print statement. (Note that if there is no explicit call, then the compiler automatically inserts a call to the parent class's null constructor.)
2. Assume the following interface and classes have been defined.

```
public interface Alertable {
   void alert();
}

public abstract class InfoAlert implements Alertable {
   public void alert() {
      System.out.println(this.toString());
   }

   public abstract String getDetails();

   public String toString() {
      return "Information: " + getDetails();
   }
}

public class SystemAlert extends InfoAlert {

   public void alert() {
      System.out.println(super.toString());
   }
```

```
    public String getDetails() {
       return "Wolfware is down."
    }

    public String toString() {
       return super.toString + getDetails();
     }
 }

 public class WeatherAlert extends InfoAlert {

    public String getDetails() {
       return "School canceled due to snow."
    }

    public String toString() {
       return "Weather: " + getDetails();
    }
 }
```

a.
   1. Illegal. You cannot instantiate an interface.
   2. Legal
   3. Legal
   4. Illegal. You cannot instantiate an abstract class.
   5. Illegal. Type mismatch. You cannot convert from **`SystemAlert`** to **`WeatherAlert`**.

b. **`Weather: School canceled due to snow.`**

c. **`Information: Wolfware is down.`**
   `Information: Wolfware is down.Wolfware is down.`

3. This question is all about polymorphism! All of the code compiles. But the second code fragment results in a ClassCastException.
   a. **`Weather: School canceled due to snow.`**
      `School canceled due to snow.`
      `School canceled due to snow.`
   b. **`Information: Wolfware is down.`**
      `Wolfware is down.`
      [ClassCastException: class SystemAlert cannot be cast to class WeatherAlert]

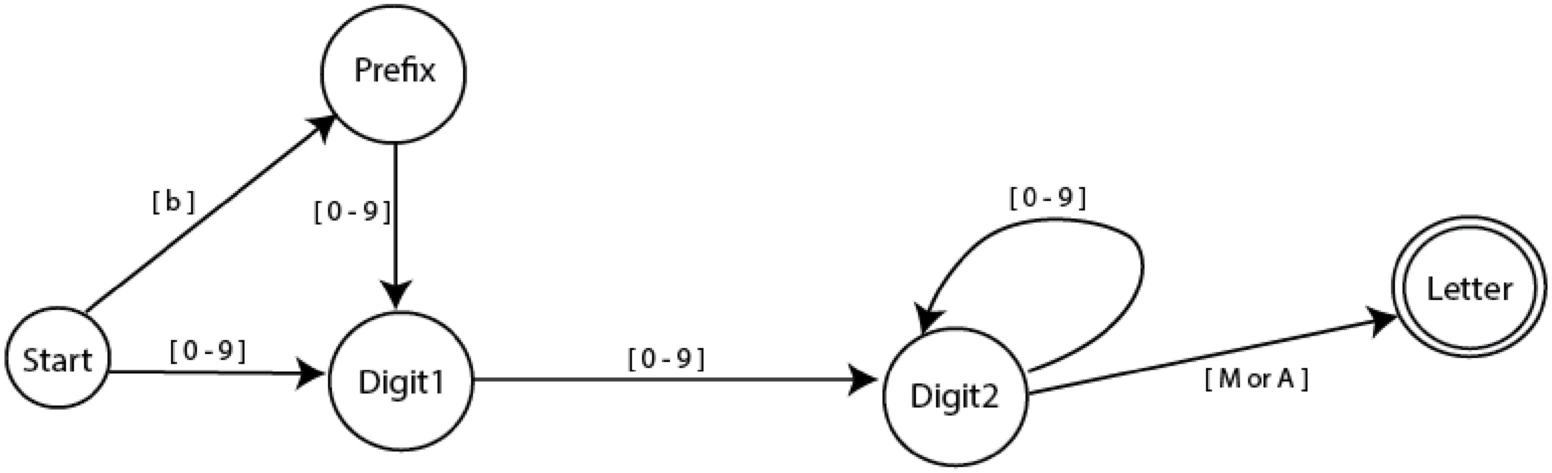
Testing

1. Terminology
   a. Unit testing
   b. Acceptance testing
   c. Regression testing

2. The 8 best values for testing are at the boundaries:
   a. -1 at the boundary of values that are not legal because they are too small
   b. 0 smallest legal value
   c. 2000 largest value for $350 charge
   d. 2001 smallest value for $500 charge
   e. 3500 largest value for $500 charge
   f. 3501 smallest value for $600 charge
   g. 4000 largest value for $600 charge
   h. 4001 at the boundary of illegal values that are too large
3. Black box testing.
4. Not catching exceptions can help you see where the code is running into problems as you test it. Each exception gives a stack trace so you can pinpoint the offending line when you try to debug your code.

Finite State Machines

1.
   a. Every character in the input corresponds to a transition. Any transition out of a state that is not marked on the diagram is illegal. If processing the final character in a string results in the state being anything except Letter, then the input string is illegal.



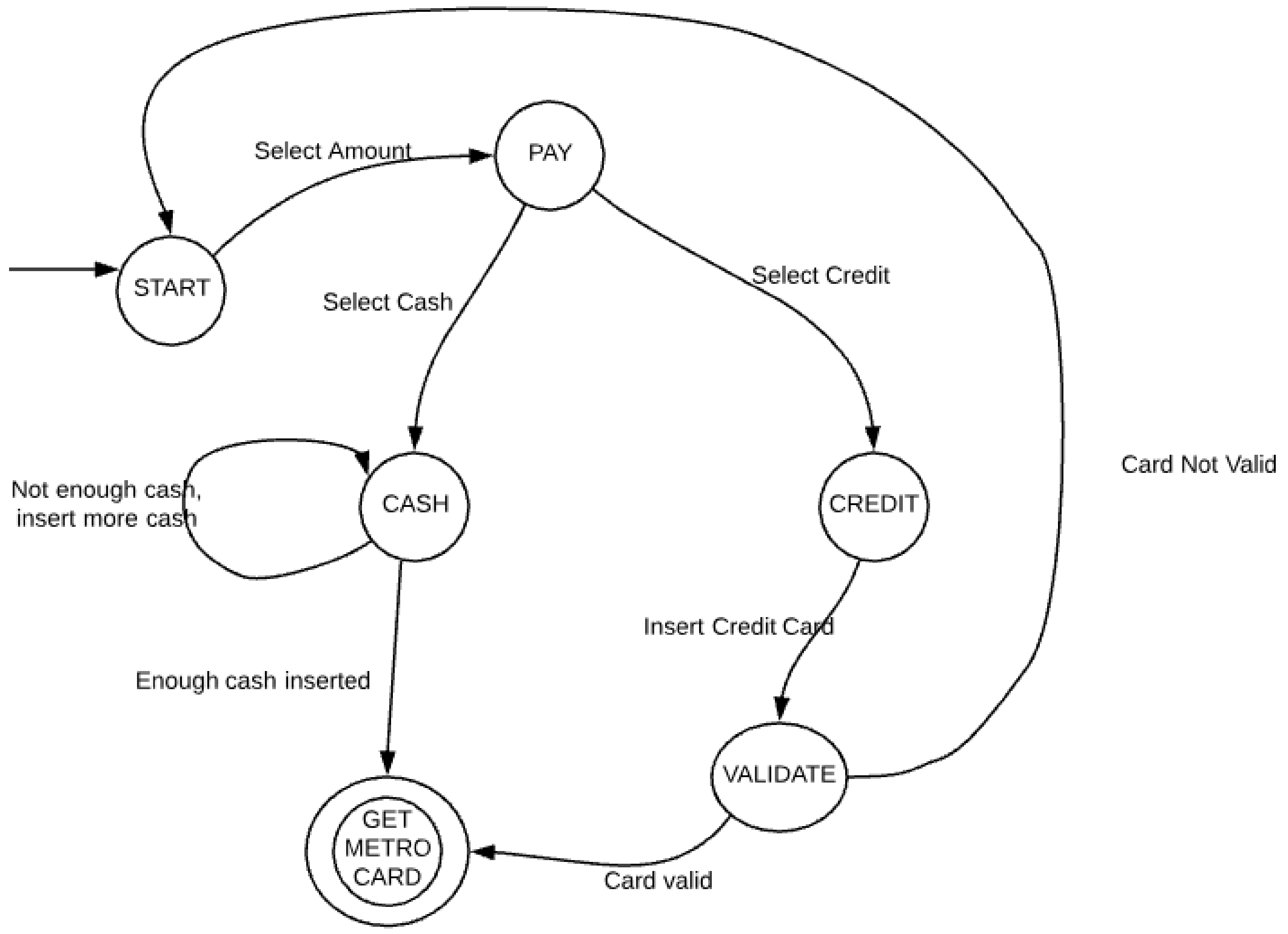
   b.

```
// Define your states
private final int START = 0;
private final int PREFIX = 1;
private final int DIGIT1 = 2;
private final int DIGITS = 3;
private final int LETTER = 4;

/**
```

```
    * Returns the brand if the Product ID is valid, or “invalid” if Product ID is not
    * @param input the Product ID
    * @return the brand of the Product: “M” for Microsoft, “A” for Apple
    * @throws IllegalFSMTransitionException if the transition is invalid
    */
  public String checkID(String input) throws IllegalFSMTransitionException {

    char ch;       // The current character being examined
    int index = 0; // The current index of the input string you're examining
    String brand = “invalid”; // Product brand - invalid by default
    // Define the state variable
    int state = START;
    while(index < input.length()) {
       ch = input.charAt(index);
       // switch statement goes here
       switch(state) {
          case START:
             if (Character.isDigit(ch))
                state = DIGIT1;
             else if (ch == 'x')
                state = PREFIX;
             else
                throw new IllegalFSMTransitionException();
             break;
          case PREFIX:
             if (Character.isDigit(ch))
                state = DIGIT1;
             else
                throw new IllegalFSMTransitionException();
             break;
          case DIGIT1:
             if (Character.isDigit(ch))
                state = DIGITS;
             else
                throw new IllegalFSMTransitionException();
             break;
          case DIGITS:
             if (ch == 'A' || ch == 'M') {
                state = Letter;
                brand = "" + ch;
             }
             else if (!Character.isDigit(ch))
                throw new IllegalFSMTransitionException();
             break;
          case LETTER:
             throw new IllegalFSMTransitionException();
```

```
            default:
               throw new IllegalFSMTransitionException();
          }

         index++;
       }   // end while
       return brand;
     }  // end isValid()
```

2.



ArrayLists and Computational Complexity

1.
```
public int removeEven() {
            int numRemoved = 0;
            for (int k = 0; k < size; k++) {
                    if (list[k] %  2 == 0) {
                            for (int j = k; j < size - 1; j++)
                                    list[j] = list[j + 1];
                            size--;
                            k--;  // Decrement k in case an even number shifted here
                            numRemoved++;
                    }
            }
            return numRemoved;
    }
```

2.

a.
```
public DifferentArrayList() {
        list = new String[100];
        for (int k = 0; k < 100; k++)
                list[k] = null;
}
```

b.
```
public int getSize() {
        int size = 0;
        for (int k = 0; k < 100; k++) {
                if (list[k] != null)
                        size++;
                else
                        return size;
        }
        return size;  // Occurs only if the list is at capacity
}
```

c. O(n). To determine the size , you need to go through the list one item at a time, counting as you go.

LinkedLists and Computational Complexity

1.
```
public int removeEvens() {
        int numRemoved = 0;
        Node p = head;
        Node q = null;
        while (p != null) {
                if (p.data %2 == 0) { // p's node is even. Remove it
                        if ( p == head) {  // Head node is a special case
```

```
                                head = head.link;
                                p = head;
                        }
                        else {  // Not at the head node. q should have a value by now
                                q.link = p.link;  // Drop p's node from the list
                                p = p.link;
                        }
                        numRemoved++;
                } // end handling even data
                else { // p's node wasn't even. Push p and q down the list.
                        q = p;
                        p = p.link;
                }  // end handling non-even data
        } // end while
        return numRemoved;
}
```

2.
   a. The code fails on a list with exactly one item, and it fails if the list is empty. (Both cases would generate a NPE.)
   b. The code works if the list has two or more items.

3.
   a.

```
public void addToEnd(int x) {
        if (head == null) {
                head = new Node(x, null);
                tail = head;
        }
        else {
                tail.link = new Node(x, null);
                tail = tail.link;
        }
}
```

   b. O(1), which is constant time, independent of the size of the list. (Since you have a reference to the end of the list, you don't have to travel through all the list items to find it.)

Stacks and Queues

1.
   a. stack contents: BOTTOM [-100, -70, 15, 20, 300] TOP
   b. returned queue contents: FRONT [-70, -100] BACK

2.
   a. queue contents: FRONT [-6, 0, -12] BACK
   b. returned stack contents: BOTTOM [2, 4, 10] TOP

Linked List Variations

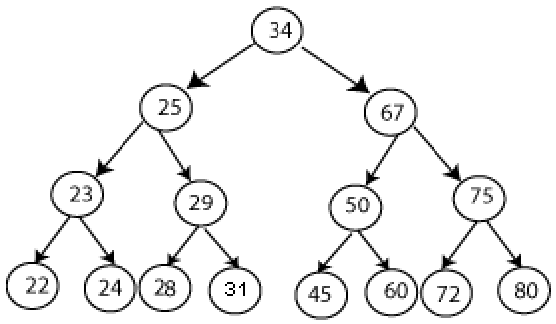
1.
```
public E removeLast() {
   if (head == null)  // List is empty
      return null;
   E value = tail.data;
   if (head == tail) {  // List has only 1 item
      head = null;
      tail = null;
   }
   else  // List has more than 1 item
      tail = tail.prev;
   return value;
}
```

Recursion

1.
```
public class LinkedListRS {
  public List head;    // List is the Node type!

  private class List {
     // .........

     public void add(String s) {
        if (next == null || next.data.compareTo(s) > 0)
           next = new List(s, next);
        else
           next.add(s);
    }

    // ......................
  }  // End definition of List

  public void add(String item) {
     if (head == null || head.data.compareTo(item) > 0)
         head = new List(item, head);
     else
         head.add(item);
  }

  // More LinkedListRS methods here
  }
```

Binary Trees

1.



2. Maximum number of comparisons required to search for an item in the binary search tree = 4. If the items were in a linked list instead of a tree, the maximum number of comparisons required to search = 15.
3.