Name: Solutions
Student ID Number: ________________________________
Signature: _________________________________________

Directions:

• This exam is closed book and closed notes. No electronic devices are permitted.

• Please check that you have pages 1 through 8. Write your name on every page.

• Do your own work. No discussion or collaboration with other students is permitted.

• If a question seems ambiguous, please walk to the front of the room and ask me about it quietly, or write an explanation of how you interpreted the question.

• The exam is divided into parts, corresponding to Modules 1 through 5, plus optional extensions for Modules 2 and 4. Complete only those parts for which you earned less than 85% on the corresponding quiz. Leave the rest blank.

• You have two hours to complete the exam.

• When you finish: If fewer than 10 minutes remain, please do not turn in your exam early, since getting up may disturb other students who are trying to finish.

Bonus question: At undergraduate research day during class on Friday, October 12, which of the following was NOT a topic of research presented (circle one)?

- Genetic testing for algorithmic mutations
- N-body simulation
- Physics engines for K-12 education
- Wireless networks for structural health monitoring
- Extracting information automatically from photographs
1. Complete the following table by filling in the value and type of each expression.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>“The sum is ” + 5 + 5</td>
<td>&quot;The sum is 55&quot;</td>
<td>String</td>
</tr>
<tr>
<td>(5 &lt; 7) != (3 &gt; 5)</td>
<td>true</td>
<td>boolean</td>
</tr>
<tr>
<td>1 / 2</td>
<td>0</td>
<td>int</td>
</tr>
<tr>
<td>7 / (1 + 1.0)</td>
<td>3.5</td>
<td>double</td>
</tr>
<tr>
<td>2 + 3 * 2 + 3</td>
<td>11</td>
<td>int</td>
</tr>
</tbody>
</table>

2. Draw an expression tree for the following expression.

\[ \neg \left( \frac{(4 \times 3)}{2} - 3 \times 2 > \text{Math.sqrt}(16) \right) \]

3. Beside each operator in the above expression tree, show the value that would result from carrying out the evaluation at that point in the computation.
Module 2
Recursion

```java
int foo(int j) {
    if (j < 2)
        return j;
    else
        return foo(j-2) + foo(j-1);
}
```

1. Using the substitution model, show the execution of `foo(4)`. Circle the return value.

   ![Substitution Model Diagram]

2. In the execution of `foo(4)`, draw the execution stack as it would look just before the first time line 3 is executed. Remember that on line 5, Java will execute the first recursive call to completion before making the second call.

   ![Execution Stack Diagram]

3. Vin Cent Mortgage offers you $d$ dollars at a monthly interest rate $r$. You won’t make any payments for the first $n$ months! But each month the loan balance is increased by $r$ times the previous loan balance. Write a recursive method called `balloon` that computes how much money you’ll owe after $n$ months.

   ```java
   double balloon(double d, double r, int n) {
       if (n == 0)
           return d;
       else
           return balloon(d*(1+r), r, n-1);
   }
   ```

   Bonus (1 point): Is your method tail recursive? Yes. Explain: there is no combining step (the recursive call returns the final answer)
Suppose \( f(x) \) is a continuous function, and that \( f(x) \) is positive value for all values of \( x \). (That is, the curve is always above the \( x \) axis.)

Let \( A(a, b, w) \) be an approximation for the area under the curve \( f(x) \) between \( x=a \) and \( x=b \). The approximation is obtained by summing the areas of rectangles, as shown. Each rectangle has integer width \( w \). The first rectangle has height \( f(a) \) and the last rectangle has height \( f(b-w) \).

Write a correct Java method named `areaUnderTheCurve` with the following specification.

**Use iteration. Do not use recursion.**

Parameters: \( a \) and \( b \), the left and right ends of the region (both integers)
\( w \), the width of each rectangle (an integer)

Assume: \( w > 0 \).
\( b-a \) is divisible by \( w \).
There already exists a method `double f(int x)` that returns \( f(x) \).

Return: \( A(a, b, w) \)

```java
    double areaUnderTheCurve(int a, int b, int w) {
        double sum = 0;
        int x = a;
        while (x < b) {
            sum = sum + w * f(x);
            x = x + w;
        }
        return sum;
    }
```

Optional Extension:

a. State a sensible loop invariant for your implementation above.

\[ \text{sum} = A(a, x, w) \]

b. Use the loop termination condition and your loop invariant to argue that the method returns the correct value.

\[ \text{On termination, } x = b, \text{ so substituting into the loop invariant, we have } \text{sum} = A(a, b, w). \]
Consider the following Java code:

```java
public class Account {
    private int balance;
    public Account(int openingBalance) {
        balance = openingBalance;
    }
    public int getBalance() {
        return balance;
    }
    public void deposit(int dollars) {
        balance = balance + dollars;
    }
    public boolean withdraw(int dollars) {
        if (dollars <= balance) {
            balance = balance - dollars;
            return true;
        } else {
            return false;
        }
    }
    public boolean transfer(int dollars, Account dest) {
        if (withdraw(dollars)) {
            dest.deposit(dollars);
            return true;
        } else {
            return false;
        }
    }
    public String toString() {
        return ("$" + balance + ".00");
    }
}
```

```java
public void testAccount() {
    Account alan, betty, cathy, elaine;
    alan = new Account(50);
    betty = new Account(25);
    cathy = null;
    Account david = betty;
    System.out.println("Alan has " + alan);
    System.out.println("Betty has " + betty);
    System.out.println("Cathy has " + cathy);
    System.out.println("David has " + david);
    betty = alan;
    alan = cathy;
    cathy = new Account(betty.getBalance());
    betty.transfer(15, cathy);
    System.out.println("Now Alan has " + alan);
    System.out.println("Now Betty has " + betty);
    System.out.println("Now Cathy has " + cathy);
    System.out.println("Now David has " + david);
    //Be careful!
    Account fred = new Account(0);
    fred.deposit(-5);
    System.out.println("OK? " + fred.withdraw(-9));
    betty = new Account(fred.getBalance() +
        cathy.getBalance());
    System.out.println("Finally...");
    System.out.println("Alan has " + alan);
    System.out.println("Betty has " + betty);
    System.out.println("Cathy has " + cathy);
    System.out.println("David has " + david);
    System.out.println("Fred has " + fred);
}
```

1. How many Account reference variables are declared within the method testAccount? _____6_____

2. How many Account objects are created during execution of the method testAccount? _5___

3. Hand simulate the execution of the testAccount() method. Keep track of which object each variable refers to, and keep track of the values in each object. **Draw the variables and objects in the space below. Show each variable as a labeled rectangle and each object as a circle. Draw arrows from variables to the objects they reference.** As you go, fill in the blanks to complete the output that would be printed.

```
Alan has $50.00
Betty has $25.00
Cathy has null
David has null
Now Alan has null
Now Betty has $25.00
Now Cathy has $35.00
Now David has $25.00
OK? true
Finally...
Alan has null
Betty has $67.00
Cathy has $35.00
David has $25.00
Fred has $4.00
```
Module 5  
Modular Design

Suppose you are given classes Widget, Machine, and Truck. Widgets have a color and a weight, and are sometimes defective. Machines can transform widgets. When a machine transforms a widget that is passed to it, the method returns the same widget back as the return value. Some machines can make a widget become defective, or fix a defective widget. Machines may break when transforming widgets. Trucks can be loaded with widgets, one at a time. Widgets are loaded and unloaded at the back of the truck, so the widget that gets loaded first will be the last widget to be unloaded. So, if you load widgets a, b, and c into a truck, then you’ll unload them in the order c, b, and a. If a truck is full, you can’t load more widgets into it. If a truck is empty, you can’t unload widgets from it.

1. Write a method that takes as parameters two unbreakable machines m1 and m2.
   • One of m1 or m2 occasionally makes widgets defective, and the other one never does.
   • Your method should return the machine that never makes widgets defective.
   • Assume widgets are not defective when created. Create as few widgets as possible.
   Hint: To ensure that your algorithm terminates in all cases, alternate between testing the machines.

   ```java
   public Machine goodMachine(Machine m1, Machine m2) {
       Widget w = new Widget();
       Machine choice = null;
       while (!w.isDefective()) {
           if (m1.transform(w).isDefective())
               choice = m2;
           else if (m2.transform(w).isDefective())
               choice = m1;
       }
       return choice;
   }
   ```

2. Complete the following method that returns the lightest widget in the given truck.
   • Assume the given truck contains at least one widget, and any truck you create is sufficiently large.
   • At the end, all widgets (except the lightest one) should be back in the original truck.
   Hint: As you empty the truck, remember the lightest widget encountered so far.

   ```java
   public Widget lightest(Truck t) {
       // Assume all widgets in t have different weights.
       Truck spare = new Truck();
       Widget lightest = null;
       while (!t.isEmpty()) {
           Widget w = t.unload();
           if (lightest == null || lightest.getWeight() > w.getWeight())
               lightest = w;
           spare.load(w);
       }
       while (!spare.isEmpty()) {
           Widget w = spare.unload();
           if (w != lightest)
               t.load(w);
       }
       return lightest;
   }
   ```
Complete the following table to demonstrate your understanding of the iterative nature of test-driven development. Developer code should be the simplest implementation that will pass the test case on its left.

<table>
<thead>
<tr>
<th>Test Code</th>
<th>Implementation</th>
</tr>
</thead>
</table>
| @Test
  public void test() {
  Container c = new Container(5);
  }

  public class Container {
    public Container (int x) {
    }
  }

<table>
<thead>
<tr>
<th>Test Code</th>
<th>Implementation</th>
</tr>
</thead>
</table>
| @Test
g  public void test() {
  Container c = new Container(5);
  c.add(3);
  assertEquals(5, c.max());
  }

  public class Container {
    public Container (int x) {
    }
    public void add(int y) {
    }
    public int max() { return 5; }
  }

  OR

  public class Container {
    int x;
    public Container (int x) {
      this.x = x;
    }
    public void add(int y) {
      if (y > x) {
        x = y;
      }
    }
    public int max() { return x; }
  }

<table>
<thead>
<tr>
<th>Test Code</th>
<th>Implementation</th>
</tr>
</thead>
</table>
| @Test
g  public void test() {
  Container c = new Container(5);
  c.add(3);
  c.add(7);
  c.add(2);
  assertEquals(7, c.max());
  }

  public class Container {
    int x;
    public Container (int x) {
      this.x = x;
    }
    public void add(int y) {
      if (y > x) {
        x = y;
      }
    }
    public int max() { return x; }
  }