CMSC 206: Data Structures
Practice Exam #1

For questions 1-3, consider the following code:

```
public class Greeter
{
      private String target;

      public Greeter(String theTarget)
      {
            target = theTarget;
      }

      public void greet()
      {
            System.out.println("Hello, " + target);
      }

      public String getTarget()
      {
            return target;
      }
}

public class PoliteGreeter extends Greeter
{
      // no fields
      public PoliteGreeter(String theTarget)
      {

      }

      public void greet()
      {
            System.out.println("How are you, " +
                                    getTarget() + "?");
      }
}

public class Main
{
      public static void main(String[] args)
      {
            Greeter dave = new Greeter("Dave");
            dave.greet();

            PoliteGreeter diane = new PoliteGreeter("Diane");
            diane.greet();
      }
}
```

This program is intended to print the following:

```
Hello, Dave
Hello, Diane
How are you, Diane?
```

1. Which of the following statements is true of the class `Greeter`?

```
public class Greeter
{
      private String target;

      public Greeter(String theTarget)
      {
            target = theTarget;
      }

      public void greet()
      {
            System.out.println("Hello, " + target);
      }

      public String getTarget()
      {
            return target;
      }
}
```

A. The class compiles, but the constructor does not work as it should. Instead, it should have `theTarget = target;`.

B. The class does not compile because it does not have a constructor that takes no parameters.

C. Because the class does not have a constructor that takes no parameters, one is provided for us. The line `Greeter g = new Greeter();` would compile.

D. The method `greet` does not compile because it produces a `String`, and yet its return type is `void`.

E. The class compiles and runs as expected.

2. What statement is true of the class `PoliteGreeter`?

```
public class PoliteGreeter extends Greeter
{
      // no fields
      public PoliteGreeter(String theTarget)
      {

      }

      public void greet()
      {
            System.out.println("How are you, " +
                                    getTarget() + "?");
      }
}
```

A. The class does not compile because the implicit constructor implicitly calls a superclass constructor that does not exist.

B. The provided constructor does not compile because it implicitly calls a superclass constructor that does not exist.

C. The `greet` method overrides a method in `Greeter` and should be declared `abstract`.

D. The class does not compile because it uses the `target` field of `Greeter` but does not explicitly inherit that field. We must add `private String target;` to fix this problem.

E. The class compiles and runs as expected.

3. The program as written does not produce the output desired. If we fix any compilation errors according to the questions above, how do we make the program produce the desired output?

   A. Add the following line to the end of the `Greeter` constructor:
   ```
   greet();
   ```

   B. On the line following the construction of the `PoliteGreeter` object, add this:
   ```
   diane.super.greet();
   ```

   C. Remove the two lines in `main` that explicitly call the `greet()` method.

   D. At the beginning of `PoliteGreeter`'s `greet` method, add this:
   ```
   super.greet();
   ```

   E. On the line following the construction of the `PoliteGreeter` object, add this:
   ```
   diane.Greeter.greet();
   ```

4. Read the program below and write what would be printed if this program were compiled and run:

```
public class Robot
{
      private int difficulty;

      public Robot(int diff)
      {
            difficulty = diff;
      }

      public void whoAreYou()
      {
            System.out.println("I'm a robot!");
      }

      public int getDifficulty()
      {
            return difficulty;
      }
}

public class LegoRobot extends Robot
{
      public LegoRobot()
      {
            super(3);
      }

      @Override
      public void whoAreYou()
      {
            System.out.println("I look like Wall-E.");
      }
}

public class FirstRobot extends Robot
{
      public FirstRobot()
      {
            super(11);
      }

      @Override
      public void whoAreYou()
      {
            super.whoAreYou();
            System.out.println("I go round and round.");
      }
}
```

```
public class Main
{
    public static void main(String[] args)
    {
        Robot rob = new Robot(5);
        System.out.println(rob.getDifficulty());
        rob.whoAreYou();

        LegoRobot lego = new LegoRobot();
        System.out.println(lego.getDifficulty());
        lego.whoAreYou();

        rob = lego;
        rob.whoAreYou();

        FirstRobot first = new FirstRobot();
        System.out.println(first.getDifficulty());

        rob = first;
        rob.whoAreYou();
    }
}
```

5. Answer the following questions about the classes used in the problem above:

   a. How many distinct robot-related classes were defined? What are their names?

   b. Does `Robot` have a superclass? If so, what is it?

   c. Does `LegoRobot` have a superclass? If so, what is it?

   d. What would happen if we removed the `super(3);` line from the `LegoRobot` constructor? Why?

   e. How many methods are in the `FirstRobot` class? Do not count any methods that are not written in this practice exam. What are these methods' names?

   f. The `whoAreYou` method in `Robot` uses `System.out.println`. What would happen if this were changed to say `return` instead? That is, what would happen if we changed this line to `return "I'm a robot!";`?

6. Consider the following class and interface:

```
public interface Iface
{
     int ifaceMethod();
}

public abstract class Superclass
{
     public Superclass(int x)
     {
          System.out.println(x);
     }

     public abstract void superMethod(int x);
}
```

Fill out the following class with the minimal number of methods and/or constructors so that it compiles. The behavior of the methods and/or constructors is irrelevant, as long as they compile. You may not label the class as `abstract`.

```
public class Problem5 extends Superclass
                      implements Iface
{











}
```

7. Consider the following statements. Draw a memory diagram (that could show aliasing) after these statements are executed.

```
int x = 5;

int[] a = new int[2];
a[0] = x;
a[1] = x;
int[] b = new int[2];
b[0] = a[0];
b[1] = 8;
a[0] = 10;
int[] c = b;
c[0] = 15;
```

8. The following problem is not simply contrived for your confusion. All topics discussed are real and are used daily by appropriately qualified computer scientists. Indeed, Java version 8 has built-in support for these concepts, though that support is beyond the scope of this course. All answers to questions in this section are *short*!

   A *function object* is an object that represents a mathematical function, such as $f(x) = x^2$ or $g(x) = |x|$. Once we have function objects, we can then define operations on the, such as composition or duplication. The composition c of functions f and g is defined by $c(x) = f(g(x))$. The duplication d of a function f is defined by $d(x) = f(f(x))$. Note that a duplication is the composition of a function with itself.

   For this problem, we will develop a set of classes that implement unary functions in Java. (Unary simply means we have functions that take only one argument. We will not consider binary, trinary, or n-ary functions for this problem.)

   Here is the `Function` class that is the superclass of all functions:

```
public abstract class Function
{
     /** @return the result of the operation */
     public abstract int operate(int operand);

     /** Perform the operation on every element of a list
      *  @param operands a list of numbers to be operated upon
      *  @return a list of the numbers that are the results
      *          of the operation on the operands list; the
      *          returned list has the same number of elements
      *          as the operands list
      */
     public ArrayList<Integer> map(ArrayList<Integer> operands)
     {
          ArrayList<Integer> result = new ArrayList<>();

          for(int op : operands)
          {
               result.add(operate(op));
          }

          return result;
     }

     /** @return a duplicate of this operation */
     public Function duplicate()
     {
          // to be implemented below
     }
}
```

Here is an example class that extends `Function`:

```
public class Squared extends Function
{
     /** post: operand squared is returned
     public int operate(int operand)
     {
          return operand * operand;
     }
}
```

To illustrate the use of function objects, here is a sample `main` method:

```
public class Main
{
     public static void main(String[] args)
     {
          Squared sq = new Squared();
          ArrayList<Integer> nums = new ArrayList<Integer>();

          nums.add(1);
          nums.add(2);
          nums.add(3);
          nums.add(4);

          ArrayList<Integer> squaredNums = sq.map(nums);
          System.out.println(squaredNums);
     }
}
```

This `main` method prints the following:

```
[1, 4, 9, 16]
```

a. Write an `Abs` class that extends `Function` and implements absolute value:

b. Write a `Composition` class that extends `Function` and represents the composition of two functions:

```
public class Composition extends Function
{
  // fields here:






  /** Creates a function representing f(g(x)) */
  public Composition(Function f, Function g)
  {
       // code here:






  }

  /** @return the value of f(g(operand)) */
  @Override
  public int operate(int operand)
  {







  }
}
```

c. Write a `Duplicate` class that extends `Composition` and represents the duplication of a function. `Duplicate` extends `Composite` because duplication is really a special case of composition. It should have a constructor that takes one function as a parameter and any other fields and methods as necessary:

```
public class Duplicate extends Composition
{
















}
```

d. Write the `duplicate` method in `Function`. It should return a duplication of the function it is called on.

```
public Function duplicate()
{









}
```

e. The `Abs` class can override the `duplicate` method to be more efficient, because duplicating `Abs` does not do anything more than `Abs` itself. (In other words, if $g(x) = |x|$, then $g(g(x)) = ||\mathrm{x}|| = |\mathrm{x}| = g(x)$.) Write this more efficient version of the `duplicate` method:

```
public class Abs extends Function
{
   // other stuff you wrote above
   @Override
   public Function duplicate()
   {






   }
}
```