Question 1. (1 point) Say that you decide to make the instance variables in a superclass protected rather than private. What does this increase? Circle the right word from this list:

(coupling,) congestion, cohesion

Question 2. (4 points) If a method in a public class has the access level shown in the left column, where can that method be used? Check each box where code in the given class is allowed to call a method with the given access level.

	A class in the same package	A class in another package	A subclass in the same package	A subclass in another package
protected	**X**		**X**	**X**
public	**X**	**X**	**X**	**X**
package	**X**		**X**	
private				

Question 3. (2 points) In class we've said that you should design classes so that they minimize coupling and maximize cohesion. Give one reason why this is good advice.

There are many reasons, e.g.:
Minimize coupling to: allow internal class changes without requiring changes in other code; allow independent code development
Maximize cohesion to: allow choosing a class with just the desired functionality without having to accept unwanted functionality as well; facilitate understanding the code

Question 4. (1 point) Say that you are writing a set of classes that provide a way to read text out loud. Among them is a class called `Speech` that provides methods to do the conversion to sounds. Your friend has written a separate application for use by speechwriters -- it, too, contains a class called `Speech` that holds the text, runs the TelePrompter, etc. Your friend wants to use your classes to have speeches read aloud by the computer, but neither of you wants to change the name of your `Speech` class so your friend can use both classes. You won't have to change your class names if you each put your classes in your own (circle the right one):

superclass, interface, (package) model, name thingy

Question 5. (5 points) In the following list, specify which part of a model-view-controller architecture handles the given task. Write the appropriate letters next to each of the components listed in the left column.

The model handles **c, e**

The view handles **b, d**

The controller handles **a**

(a) user interaction
(b) displaying information
(c) keeping track of the core computation or information
(d) looking scenic
(e) making sure the vehicles don't collide

Question 6. (4 points) Consider these class declarations:

```
public class A {
  public int xyzzy(double x, double y) {...}   // method #1
  public int xyzzy(double x, String s) {...}   // method #2
}
public class B extends A {
  public int xyzzy(double w, double z) {...}   // method #3
}
```

a) (2 points) Fill in the blanks below with whichever of the following words best describes the relationship between the two methods given: overloads, extends, implements, overrides

`int xyzzy(double w, double z)` in `B` __**overrides**__ `int xyzzy(double x, double y)` in `A`

`int xyzzy(double x, String s)` in `A` __**overloads**__ `int xyzzy(double x, double y)` in `A`

b) (1 point) In the following code, which of the three `xyzzy` methods is actually called? The methods are numbered in the comments above -- circle the right number: 1 (2) 3

```
B b = new B(...);
b.xyzzy( 5.0, "howdy" );
```

c) (1 point) Say that we call `b.toString()`. In what class does Java find the `toString` method?

Object

Question 7. (2 points) Recall our favorite example class, `Employee`. Say that we also have a `Customer` class, and we notice that it has data and methods for some kinds of information that are also in `Employee`, e.g. name, address, phone. We'd like to reduce redundancy by putting that duplicated code somewhere that both `Customer` and `Employee` could use it. We'd also like to have some generic type name that we could use for either `Employees` or `Customers` when we use the common methods (maybe we'd like to put all these people in a big list and send them our monthly newsletter). But we don't want to allow making any objects of that generic type -- we want all our objects to be actual `Employees` or `Customers` (or their subclasses).

Given those requirements, should we make our new generic type an *interface*, a *concrete superclass*, or an *abstract superclass*? Briefly tell why.

The new type must be an abstract superclass: To prevent making objects of that type, it would either need to be an abstract superclass or an interface. But we also want to provide method implementations in it, and an interface can contain only declarations, not implementations.

Question 8. (6 points) Say you have a `Clock` class that counts up minutes and hours. It has a `void nextMinute()` method that tells the `Clock` to advance to the next minute, and `int getMinutes()` and `int getHours()` that report the minutes and hours. It has a constructor, `Clock(int hours, int minutes)`, that lets you set the starting minutes and hours. That is, to make a `Clock` set to 10:30, have it change to the next minute, and print the new time, you'd do:

```
Clock c = new Clock( 10, 30 );
c.nextMinute();
System.out.println( c.getHours() + ":" + c.getMinutes() );
```

The `Clock` is supposed to behave like a typical cheap desk clock: When minutes gets to 59, then the next call to `nextMinute` should go to the next hour, and the minutes should go back to zero. If the hour is 12 and minutes is 59, the next minute should have hour 1 and minute 0.

Your job is not to write code for `Clock`. Instead, you should write code to *test* `Clock`., specifically, `Clock`'s `nextMinute` method. (You don't need to test the constructor or the get and set methods -- only `nextMinute`.) Decide what cases you need to check to make sure `Clock`'s `nextMinute` method changes the hours and minutes as described. Below, there is a skeleton for a `JUnit` test. Fill in the method `testAddMinute()` to perform your tests. You can add instance variables or other methods if you need them. You can make several different `Clock` objects set to different times if that will help. Here is a short list of methods provided by class `TestCase` – you can use these or any others you remember:

```
assertEquals(int expected, int actual)
assertEquals(double expected, double actual, double delta)  // for floating-point
assertNull(Object reference)
assertNotNull(Object reference)
assertTrue(boolean condition)
assertFalse(boolean condition)
```

There are alternate versions of all of these that take a `String` as an additional first parameter that you can use to print a message, e.g.

```
assertEquals(String message, int expected, int actual)
```

Write your code on the next page.

Question 8 (cont.) Write your test code here.

```
import junit.framework.TestCase;
public class ClockTest extends TestCase {

    public void testAddMinute() {

        Clock clock;

        // We need to include a test for each distinctly different behavior -- any
        // case where the external description of the behavior is different, and
        // any case where we suspect the code might differ internally.  (In the set
        // of tests below, we only check edge cases -- we have not exhaustively
        // looked for bad behavior.  Whether this is appropriate depends on how
        // many non-edge-cases there are, and how critical the software is.)
        // Because we have no other testXXX methods to share setup with, we do all
        // setup here.

        // No minute or hour rollover:
        // State before call is: any legal hour, any legal minute except 59.
        // Expected result is: minute should go up by 1, hour stay same.
        clock = new Clock( 10, 30 );  // Any hour, any minute other than 59.
        clock.nextMinute();           // Advance the time.
        assertEqual( "Hour should not change if previous minute is not 59",
                     clock.getHour(), 10 );
        assertEqual( "Minute should go up by one if previous minute is not 59",
                     clock.getMinute(), 31 );

        // Minute rollover without hour rollover:
        // State before call is: any legal hour except 12, minute is 59.
        // Expected result is: minute should go to 0, hour should go up by 1.
        clock = new Clock( 1, 59 );   // Any hour except 12, minute is 59.
        clock.nextMinute();           // Advance the time.
        assertEqual( "Hour should go up by one if previous hour is not 12" +
                     " and previous minute is 59",
                     clock.getHour(), 2 );
        assertEqual( "Minute should go to 0 if previous minute is 59",
                     clock.getMinute(), 0 );

        // Minute rollover and hour rollover:
        // State before call is: hour is 12, minute is 59.
        // Expected result is: minute should go to 0, hour should go to 1.
        clock = new Clock( 12, 59 );  // Hour is 12, minute is 59.
        clock.nextMinute();           // Advance the time.
        assertEqual( "Hour should go to 1 if previous hour is 12" +
                     " and previous minute is 59",
                     clock.getHour(), 1 );
        assertEqual( "Minute should go to 0 if previous minute is 59",
                     clock.getMinute(), 0 );


    }
}
```

Question 9. (6 points) Consider these class definitions:

```
public class Whatzit {
  private String noise = "Huh?";
  public String makeNoise() {
    return noise;
  }
  public void newNoise( String noise ) {
    this.noise = noise;
  }
  public String makeNoise( boolean which ) {
    return makeNoise();
  }
}

public class Thingamajig extends Whatzit {
  public String makeNoise( boolean which ) {
    if ( which )
      return super.makeNoise();
    else
      return makeNoise();
  }
}

public class Deelybobber extends Whatzit {
  private String noise = "Uh-oh";
}

public class Gizmo extends Thingamajig {
  public String makeNoise() {
    return "Sproing!";
  }
}
```

What gets printed when the following code is executed?

```
Whatzit t = new Thingamajig() ;
Whatzit d = new Deelybobber() ;
Whatzit g = new Gizmo();

System.out.println( t.makeNoise() );           Huh?

System.out.println( d.makeNoise( false ) );    Huh?
System.out.println( d.makeNoise( true ) );     Huh?

System.out.println( g.makeNoise( true ) );     Huh?
g.newNoise( "Yowza!" );
System.out.println( g.makeNoise( true ) );     Yowza!
System.out.println( g.makeNoise( false ) );    Sproing!
```

Question 10. (8 points) In this question we'd like to expand the bouncing ball simulation as follows. If two or more balls collide during a cycle, we want to add new balls to the simulation whose diameter is the sum of the diameters of all of the balls that have collided.

To be more specific, when a ball's `action()` method executes, it should ask the model for a list of all the balls in the simulation, go through that list, and discover all of the balls, if any, that it overlaps. If the ball touches or overlaps one or more other balls, then its `action()` method should create a new ball whose diameter is the sum of the diameters of all the overlapping balls and add it to the model. The new ball can have whatever position and motion you like (including 0). Each of the balls involved in the collision should create a new ball in its `action()` method – don't worry about checking for whether a neighboring ball has already created a new one.

Here is some reference information about the `SimModel` and the `Ball` classes. Remember that code in class `Ball` can reference instance variables and methods in any `Ball` object.

Instance methods available in `SimModel`:

`java.util.List getThings()` -- *Return a copy of the list of the `SimThings` in the simulation at the beginning of the current cycle.*

`void add(SimThing t)` -- *Add the given `SimThing` to the world after this cycle is complete.*

`List` instance method:

`Iterator iterator()` -- *Return an `Iterator` for the `List`.*

`Iterator` instance methods:

`boolean hasNext()` -- *Return true if there are more items.*

`Object next()` -- *Return the next item.*

Static method available in `Math`:

`double Math.sqrt(double number)` -- *Return the square root.*

`double Math.pow(double number, double power)` -- *Return the given number raised to the given power.*

```
import java.awt.*;
import java.util.*;

/** A small ball object that wanders randomly around a bounded area. */
public class Ball implements SimThing {
  // instance variables
  private int x, y;           // current coordinates of this Ball
  private int dx, dy;         // current motion direction in x and y
                              //    (sign gives direction, magnitude
                              //    gives speed)
  private Color color;        // color of this ball
  private int diameter;       // size of this ball
  private int maxx, maxy;     // max x and y coordinates
  private SimModel model;     // the model object that refers to this ball


  // Add any new instance variables you want here.




  /**
   * Construct a new Ball with the given coordinates, initial direction, and model.
   * @param x initial x coordinate of center of the ball
   * @param y initial y coordinate of center of the ball
   * @param dx initial change in x value on each simulation cycle
   * @param dy initial change in y value on each simulation cycle
   * @param c color of the ball
   * @param diameter diameter of the ball
   * @param model the model that this Ball is a part of
   */
  public Ball(int x, int y, int dx, int dy,
              Color c, int diameter, SimModel model) { … }

  // Add any new methods you want here.

  private int distance(int x1, int y1, int x2, int y2) {
    int dx = x1-x2;
    int dy = y1-y2;
    return (int) Math.sqrt(dx*dx  + dy*dy);
  }
```

```
  /**
   * Perform an appropriate action on each cycle of the simulation,
   * in this case advancing by dx,dy and reversing either direction
   * if we hit an edge of the simulation.
   */
  public void action() {
    x = x + dx;
    if (x < diameter/2 || x > maxx - diameter/2) {
      dx = -dx;
    }
    y = y + dy;
    if (y < diameter/2 || y > maxy - diameter/2) {
      dy = -dy;
    }

    // Add new action code here.
    // Find balls touching or overlapping this one (excluding this one).  If any,
    // add a new ball with diameter equal to sum of their diameters.

    int diameterSum = 0;               // Diameter sum of overlapping balls
    boolean foundOverlappers = false;  // = "some ball overlaps this one"

    java.util.List balls = model.getThings(); // Ask model for list of balls.
    Iterator it = balls.iterator();
    while (it.hasNext()) {
      Ball other = (Ball) it.next();

      // Check whether we overlap each ball except ourself
      if (this != other ) {
        int centerDist = distance(x, y, other.x, other.y);

        // Increase the size of the new ball if this ball overlaps other
        if (2 * centerDist <= diameter + other.diameter) {
          diameterSum = diameterSum + other.diameter;
          foundOverlappers = true;
        }
      }
    }

    // If we found one or more overlapping balls, create a new Ball
    // and add it to the model.  (For convenience, use the current ball's
    // characteristics other than the diameter.)
    if (foundOverlappers) {
      model.add(new Ball( x, y, dx, dy, color, diameter + diameterSum, model));
    }
  }
}
```