Question 1. (4 points) When we're modifying Java code, some of the things we do will change the **coupling** between the class we're working on and other classes that it interacts with. For each of the following possible changes, circle *increase* if the change increases the coupling between this class and other classes, circle *decrease* if it decreases the coupling, or circle *no effect* if the change has no effect on the coupling with other classes, whether or not it is a worthwhile change.

(a) Change the declaration of an instance variable from `public` to `private`.

increase coupling [✓ *decrease coupling*] *no effect*

(b) Change the name of an instance variable.

increase coupling *decrease coupling* [✓ *no effect*]

(c) Move a method that is declared `public` and duplicated in several related classes to their common superclass.

increase coupling *decrease coupling* [✓ *no effect*]

(d) Change the declaration of a method from `private` to `protected`.

[✓ *increase coupling*] *decrease coupling* *no effect*

Question 2. (2 points) One of your colleagues is experimenting with Java 1.4 Swing and trying to create a simple window with a label in it. For some reason the code doesn't work. Here it is:

```
import java.awt.*;
import javax.swing.*;

public class TestFrame {
  public static void main(String[] args) {
    JFrame window = new JFrame("window title");
    JLabel label = new JLabel("label text");
    window.add(label);
    window.pack();
    window.setVisible(true);
  }
}
```

What's the problem? Show how to fix it.

The problem is that you cannot add items directly to a `JFrame`, you need to use it's `getContentPane` method instead. To fix it change the line that says `window.add(label)` to `window.getContentPane().add(label)`.

Question 3. (16 points) The obligatory inheritance question.

Consider the following interfaces and classes. (You can remove this page from the exam for reference if you want to avoid having to flip back and forth while answering the questions on the next page.)

```
public interface Red{
  public void turnRed();
}

public interface Blue{
  public void turnBlue();
}


public class A implements Red {

  public void one() {
    System.out.println( "A.1" );
  }

  public void two(){
    one();
    System.out.println(  "A.2" );
  }

  public void turnRed() {
    System.out.println("Nooooooooo!!");
  }
}


public class B extends A implements Red, Blue {

  public void one( int x){
    System.out.println( "B.1 " + x );
  }

  public void two()      {
    System.out.println( "B.2" );
  }

  public void turnBlue() {
    System.out.println("Aaaiiiiiiieeeeeee!!");
  }
}


public class C extends A {

  public void one() {
    System.out.println( "C.1" );
  }
}
```

Question 3. (cont.) For each of the following, indicate what output is produced when the code is executed. If some sort of error occurs instead, give a brief explanation of the problem.

(a)
```
A a = new B();
a.two();
```
Prints "B.2"

(b)
```
B b = new B();
b.one(4);
b.one();
```
Prints "B.1 4"

Prints "A.1"

(c)
```
Red r = new B();
r.turnRed();
```
Prints "Noooooooooo!!"

(d)
```
Blue b = new Blue();
b.turnBlue();
```
Error – can't create instances of interfaces

(e)
```
C c = new A();
c.two();
```
Error – incompatible types in assignment

(f)
```
A a = new B();
a.turnBlue();
```
Error – no turnBlue method in class A

(g)
```
A a = new C();
a.two();
```
Prints "C.1   A.2"

(h) One of your colleagues wants to extend the class hierarchy to create a new class that combines the features of both class `B` and class `C` and adds a new method. Here's the code:

```
public class D extends B, C {
   public void three() { System.out.println("D.3"); }
}
```

Will this work? Why or why not?
No, a Java class can only extend one other class, not two.

Question 4. (2 points) Here is a small program that uses inheritance.

```
public class Alpha {
   public void a() {
      System.out.println("Alpha.a");
   }
}

public class Beta extends Alpha {
   public void b() {
      System.out.println("Beta.b");
   }
}
```

Now we'd like to add an instance variable to class `Alpha` along with a constructor to initialize it as follows:

```
public class Alpha {
   private int n;
   public Alpha(int initialValue) {
      n = initialValue;
   }
   public void a() {
      System.out.println("Alpha.a");
   }
}
```

Unfortunately, after this change, these classes won't compile without errors. Why not?

Class `Beta` does not contain a constructor so a default constructor is assumed which contains a call to `super()`. This works with the original version of `Alpha`, which contains no explicit constructor definitions, but once we add an explicit constructor to `Alpha`, the zero-argument constructor is no longer automatically supplied. So the call to `super()` in `Beta` fails because no zero-argument constructor is available.

Question 5. (4 points) The example code that uses Swing event handling to react to things like mouse and button events usually starts with the following lines of code:

```
   import java.awt.*;
   import java.awt.event.*;
   import javax.swing.*;
```

(a) If we change the first line to `import java.awt.Color;` will the code continue to compile with no other changes? If no, why not; if the answer is maybe or "it depends", describe when the change would be ok and when it would cause a problem.

This will work if `Color` is the only class from `java.awt` that is referenced in the code. If any other classes from `java.awt` are used, the program will not compile.

(b) Suppose we leave the first line unchanged as `import java.awt.*;` In that case, can't we delete the second line that says `import java.awt.event.*;` ? It looks like it is not needed. Is it? If so, why?

The second import is needed if anything in `java.awt.event` is used in the code. The `import java.awt.*;` statement only imports things declared directly in `java.awt`, not things declared in subpackages.

Question 6. (4 points) Suppose we're programming a computer solitaire (card) game using a Model-View-Controller design. For each of the following, identify which MVC component should implement the appropriate action by writing "M", "V", or "C".

__**C**__ React to user mouse clicks on the screen

__**M**__ Determine if a move the user is trying to make is legal

__**M**__ Shuffle the card deck (i.e., randomize the order of the cards)

__**V**__ Show an animation of the deck being shuffled

The rest of the questions involve a simple version of the dinosaur world. The code describing the key parts of the simulation appears on this page and the next. You can remove these pages for reference if you want.

```
public interface SimThing {
   public void action();
}


import java.util.*;

public class SimModel {

   /** Add a new SimThing t to this model */
   public void add(SimThing t) { ... }

   /** Return a list of all SimThings currently in the model */
   public List getThings() { ... }

   // rest omitted...
}
```

The `Food` interface and simple `Plant` and `Dinosaur` classes are defined as follows:

```
public interface Food {
   public double getCalories();
}


public class Plant implements Food, SimThing {
   private double x, y;      // meters from the center of the safari
   private double calories;  // energy value of this plant

   public Plant(double x, double y, double calories) {
         this.x = x;
         this.y = y;
         this.calories = calories;
   }

   public double getX()        { return x; }
   public double getY()        { return y; }
   public double getCalories() { return calories; }

   public void action() {
         // plants do nothing
   }
}
```

```
public class Dinosaur implements Food, SimThing {
   private double x, y;      // meters from the center of the safari
   private double calories;  // current energy of this dinosaur;
   private SimModel world;   // the model this dinosaur belongs to

   public Dinosaur(double x, double y, SimModel m) {
      this.x = x;
      this.y = y;
      this.calories = 0.0;
      this.world = m;
   }

   public double getX()        { return x; }
   public double getY()        { return y; }
   public double getCalories() { return calories; }

   public void eat(Food lunch) {
      calories = calories + lunch.getCalories();
   }

   public void action() {
      // to be supplied
   }

   /** return the distance from point (x1,y1) to point (x2,y2) */
   public double distance(double x1, double y1,
                          double x2, double y2) {
      double dx = x2 – x1;
      double dy = y2 – y1;
      return Math.sqrt(dx*dx + dy*dy);
   }
}
```

Notice that `Dinosaurs` and `Plants` are almost exactly the same, except that the number of calories in a `Plant` never changes, regardless of how many times it is eaten or how often its `action()` method is called, and a `Plant` doesn't eat anything.

Question 7. (8 points) Provide an implementation of the `action()` method for a `Dinosaur` object so that it eats all of the `Plant` objects within 100 meters of its current position. It should not eat other `Dinosaurs`, and it does not change the `Plant` objects that it eats. Notice that a `Dinosaur` has an instance variable that is a reference to the `SimModel` controlling the simulation, and you can get a list of all the `SimThings` in the simulation from there. Also, a distance method is provided that you can use to calculate the distance between two (x,y) points.

For full credit, you **must** use an iterator to process the list of `SimThings`, not method `get(i)`.

Hint: you may not need all of the available space

```
/** Perform this Dinosaur’s action – eat all Plants within
 *  100 meters */
public void action() {

   List things = world.getThings();
   Iterator it = things.iterator();
   while (it.hasNext()) {
      Object thing = it.next();
      if (thing instanceof Plant) {
         Plant food = (Plant) thing;
         if (distance(x, y, food.getX(), food.getY()) <= 100.0) {
            eat(food);
         }
      }
   }

}
```

Question 8. (16 points) Add tests to the following JUnit framework to test the following two things about your `Dinosaur` class:

(i) Verify that the `eat(Food lunch)` method properly updates the `Dinosaur`'s energy level, and

(ii) Verify that the `action()` method as described in the previous question properly eats all `Plants`, and only `Plants`, within 100 meters of the `Dinosaur`'s location.

You may include additional instance variables and methods (like `setUp`) if you'd like, but this is not required.

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

   public void testEat() {
      Dinosaur dino = new Dinosaur(0.0, 0.0, null);
      double oldCalories = dino.getCalories();
      Plant lunch = new Plant(0.0, 0.0, 100.0);
      dino.eat(lunch);
      assertEquals(100.0, dino.getCalories() – oldCalories, 0.0);
   }

   public void testAction() {
      SimModel world = new SimModel();
      world.add(new Plant(10.0, 10.0, 1.0);
      world.add(new Plant(200.0, 200.0, 2.0);
      Dinosaur fred = new Dinosaur(20.0, 20.0, world);
      fred.eat(new Plant(0.0, 0.0, 4.0));
      world.add(fred);
      Dinosaur dino = new Dinosaur(0.0, 0.0, world);
      double oldCalories = dino.getCalories();
      dino.action();
      assertEquals(1.0, dino.getCalories() – oldCalories, 0.0);
   }
}
```

There are obviously a lot of different ways to test these methods. If your code did an adequate job it received the credit, even if it didn't look like the above code. A couple of specific notes:

- **This version of `testEat()` verifies the difference between the dinosaur's calories before and after eating something so that it would continue to work even if the initial value of a dinosaur's calories was not 0.0. But it's ok to assume that a dinosaur has 0.0 calories when it is created.**
- **A minimal test for the `action()` method would be to create a model with at least one plant that is within range (≤100.0 meters), a plant that is out of range, and one dinosaur that is within range (and shouldn't be eaten). The calorie values for the different objects should be non-zero and different enough from each other so that the calories of the things eaten can't accidentally add up to the expected value.**