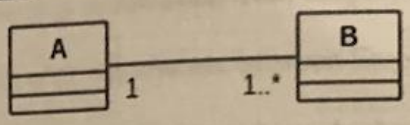
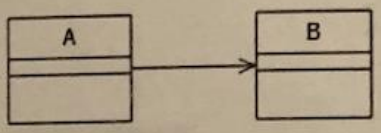
Part I: Short Answer (2 points each)

___ 1. (Fill in the Blanks) To implement the class diagram below in code, a Association of type B objects must be placed inside the class A.



___2. (Fill in the Blanks) If the class diagram below has been implemented in code, the following must be true at runtime: When an instance of class A is created, it keeps a reference to class B.



___3. (Fill in the Blanks) A Sequence Diagram Diagram shows the flow of communication between the running objects of the system, driven by the use cases of the system.

___4. What happens when the main method in the following code is executed? Explain

```
public class Base extends Extension{
    public static void main(String[] args) {
        Extension s = new Base();
        s.print();
    }
    @Override
    public void print(String s){
        System.out.println("From Extension");
    }
}
```

```
public class Extension {
    void print() {
        System.out.println("From Extension");
    }
}
```

A. There is a compiler error. → the method is throwing it, Incorrect Signature

B. There is a runtime error.

C. "From Base" is printed to the console.

D. "From Extension" is printed to the console.

___5. In the following code, which of the following is correct regarding the relationship between Employer and Gardener? Circle one letter.

A. There is a dependency from Employer to Gardener

B. There is a one-way association from Employer to Gardener

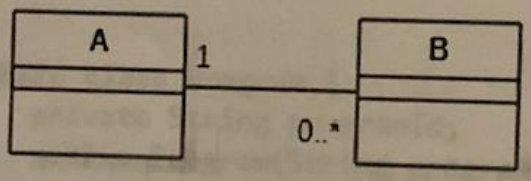
C. There is a two-way association between Employer and Gardener

D. Not possible to determine from the code shown

```
public class Employer {
    private Gardener gardener= new Gardener();

    public void employ(){
        gardener.garden();
    }
}
```

```
public class Gardener {
    public void garden() {
        //do gardening
    }
}
```

___6. Consider the following class diagram:



Which of the following statements is (are) correct? Circle all that are correct.

A. Each instance of the class B contains a list of instances of A.

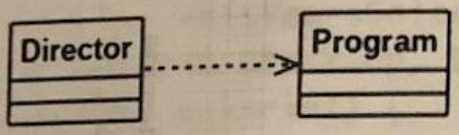
B. Each instance of the class A contains a list of instances of B.

C. A is a property of B.

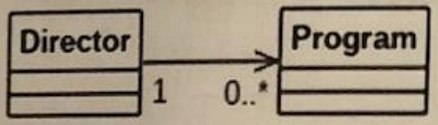
D. If an instance of A has been created, at least one instance of B has also been created.

7. Which of the following UML diagrams correctly models the relationship between Director and Program? The code for Director and Program is shown below.

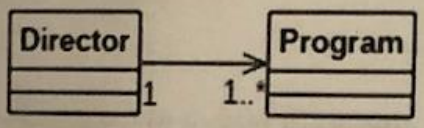
A.



B.



C




```
public class Program {
    private String programId;
    public Program(String programId) {
        this.programId = programId;
    }
}
```

```
public class Director {
    private String name;
    private List<Program> programs = new ArrayList<>();
    public Director(String name, String programId) {
        this.name = name;
        addProgram(new Program(programId));
    }
    public void addProgram(Program p) {
        programs.add(p);
    }
}
```

8. Consider the following Customer class. It contains instance variables of type Account, LocalDate, and List<Double>. (The Account class is also shown.)

```
public class Customer {
    private Account checkingAccount;
    private LocalDate birthdate;
    private List<Double> thisYearsSalaries;
    public Customer(Account checking, LocalDate bdate,
            List<Double> salaries) {
        this.checkingAccount = checking;
        this.birthdate = bdate;
        this.thisYearsSalaries = salaries;
    }
}

public class Account {
    private double balance;
}
```

If this Customer class is modeled in a class diagram:

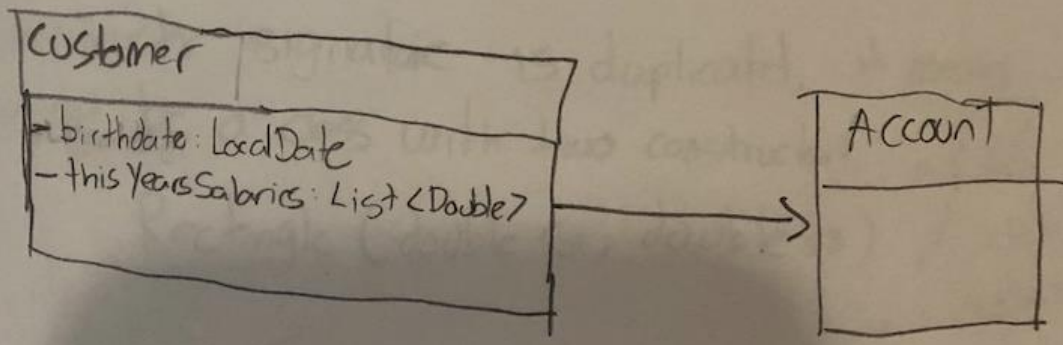
a. Which instance variables should be modeled as *attributes*?

birthdate and thisYearsSalaries

b. Which instance variables should be modeled as *associations*?

Checking Account

Explain your answer.



Because you must diagram separated entities the compounded attributes and put on the diagram primitive ones (put inside the class)

1. [10 points] A rectangle can be specified by specifying two sides joined at an endpoint (in other words, length, and width), but it can also be specified by specifying one side and a diagonal.

A. The following code attempts to implement a Rectangle class and provide support for the two ways of constructing a Rectangle. The code does not compile. Why is there a compiler error? (Write your answer below.)

```
public class Rectangle {
        double side1, side2, diagonal;
        public Rectangle(double s1, double s2) {
                this.side1 = s1; this.side2 = s2;
                diagonal = Math.sqrt(side1 * side1 + side2 * side2);
        }
        public Rectangle(double s1, double diagonal) {
                this.side1 = s1;
                this.diagonal = diagonal;
                side2 = Math.sqrt(diagonal * diagonal - side1 * side1);
        }
        public double computeArea() {
                return side1 * side2;
        }
}
```

The constructor signature is duplicated. It means, there cannot be a class with two constructors with the same signature. Using a factory method or a Builder pattern, this compile error can be resolved.

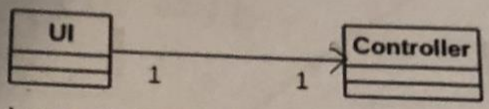
B. In the space provided below, rewrite the code for Rectangle (from Part A) so that it supports the two ways of constructing a rectangle. Use a technique described in the course (Factory Method Pattern)

```
public class Rectangle {
        double side1, side2, diagonal;

        private Rectangle() {

        }
        public static Rectangle createWithSide(double s1, double s2) {
                Rectangle rSRectangle = new Rectangle();
                rSRectangle.side1 = s1;
                rSRectangle.side2 = s2;
                rSRectangle.diagonal = Math.sqrt(rSRectangle.side1 * rSRectangle.side1 + rSRectangle.side2
* rSRectangle.side2);
                return rSRectangle;
        }
```

```
	public static Rectangle createWithDiagonal(double s1, double diagonal) {
		Rectangle rDRectangle = new Rectangle();
		rDRectangle.side1 = s1;
		rDRectangle.diagonal = diagonal;
		rDRectangle.side2 = Math.sqrt(rDRectangle.diagonal * rDRectangle.diagonal -
rDRectangle.side1 * rDRectangle.side1);
		return rDRectangle;
	}

	public double computeArea() {
		return side1 * side2;
	}
}
```

1. [20 pts]

A. The diagram below shows that (for a particular application) there is a one-one bidirectional association between a UI class and a Controller class.



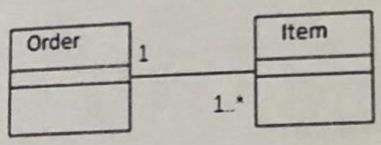
In the space provided below, write Java code that implements this diagram. Assume that UI and Controller are the only classes in a particular package. Your code must meet the following requirements:

a. The UI class owns the relationship, so it should not be possible to create an instance of Controller independently of an already existing UI class
b. The code must show relevant instance variables, constructor implementations, and methods, sufficient to implement this model. (Show only those instance variables that are implied by the diagram. Getters for instance variables should be provided.) Note: Using this diagram, the only instance variables will be those that are implied by the bidirectional 1-1 relationship.
c. All classes, properties, methods, and constructors must be given appropriate visibility qualifiers (private, protected, public, or package level).

```
public class UI {
	private Controller controller;
	public UI() {
		controller = new Controller();
		controller.setUi(this);
	}
	public Controller getController() {
		return controller;
	}
```

```
    }

    public class Controller {
        private UI ui;
        Controller(){}
        public void setUi(UI ui) {
            this.ui = ui;
        }
    }
```

B. The diagram below shows that (for a particular application) there is a one-many bidirectional association between an Order class and an Item class.



In the space provided below, write Java code that implements this diagram. Assume that Order and Item are the only classes in a particular package. The Order class owns the relationship, so it should not be possible to create an instance of Item independently of an already existing Order class.

```
public class Order {
    private List<Item> items;
    public Order() {
        items = new ArrayList<Item>();
        addItem(new Item(this));
    }
    public void addItem(Item item){
        items.add(item);
    }
}

public class Item {
    private Order order;
    Item(Order order) { this.order = order; }
    public Order getOrder() { return order; }
}
```

3. [10 pts] In the problem description below, a properties management system is described. As a first step in analysis for providing a solution to this problem, a very simple class diagram is given below. For this problem, develop the class diagram further using inheritance and include associations (with multiplicities) and some operations for your classes. Your new diagram should use the new class Property for the purpose of inheritance.

The code provided gives implementations of all the classes in the diagram shown below. Your objective is to update those implementations so that they correspond to the new version of the class diagram. Note that the Admin and Driver classes that have been provided for you have implementations that produce correct outputs, but the method computeTotalRent in Admin performs its computation by checking the types of different rental properties. You need to refactor the implementation of computeTotalent so that the inheritance you have introduced in your new diagram is used and computation is performed using polymorphism. To do this you will need to implement and make use of the (unimplemented) class Property.

Note: This problem is asking for 4 things:

(i) An improved class diagram that shows associations, multiplicities, inheritance (and uses the Property class)

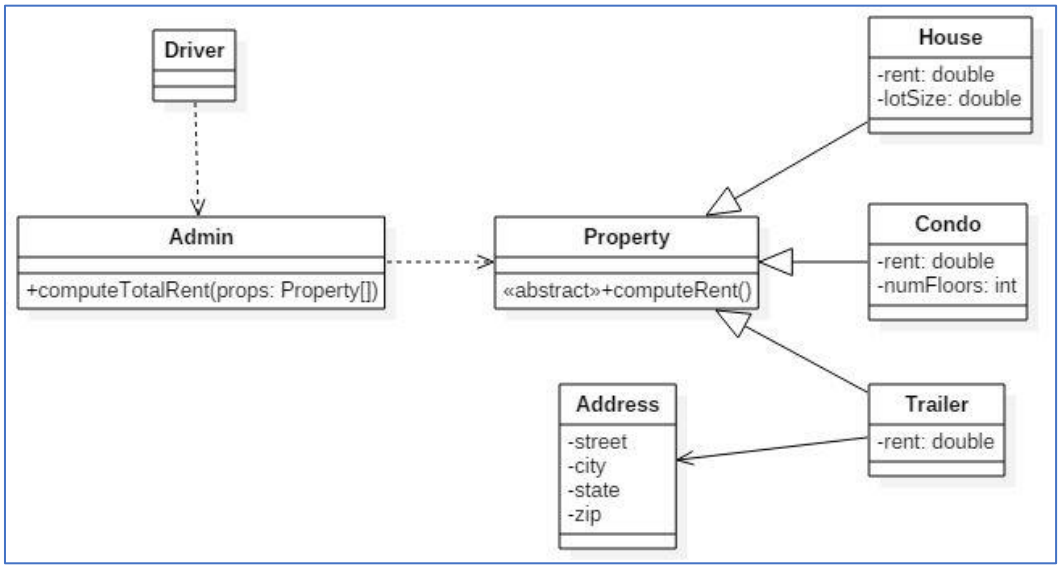
(ii) Refactored code in Admin and Driver; in particular, a rewritten version of computeTotalRent0. (Note: You do not need to implement the functionality of listing all properties in a certain city that is mentioned in the Problem Description below.)

(iii) Implementation of the Property class

(iv) Updates of the other classes provided so that they correspond to your new class diagram.

The Admin and Driver classes must be completely rewritten, and an implementation of Property needs to be done (use the space provided for each of these). For the other classes, just make small modifications of the classes provided rather than rewriting the code for these.

(i)



(ii) Admin & Driver Code:

```
import java.util.List;

public class Admin {
    public static double computeTotalRent(Property[] properties) {
        double totalRent = 0;
        for(Property p: properties) {
            totalRent += p.computeRent();
        }
        return totalRent;
    }
}

public class Driver {
    public static void main(String[] args) {
    Property[] properties = {
                new House(1200.0),
                new Condo(2),
                new Trailer()
        };
    double totalRent = Admin.computeTotalRent(properties);
    System.out.println("Total Rent: "+totalRent);
    }
}
```

(iii) Property class

```
abstract public class Property {
    abstract double computeRent();
}
```

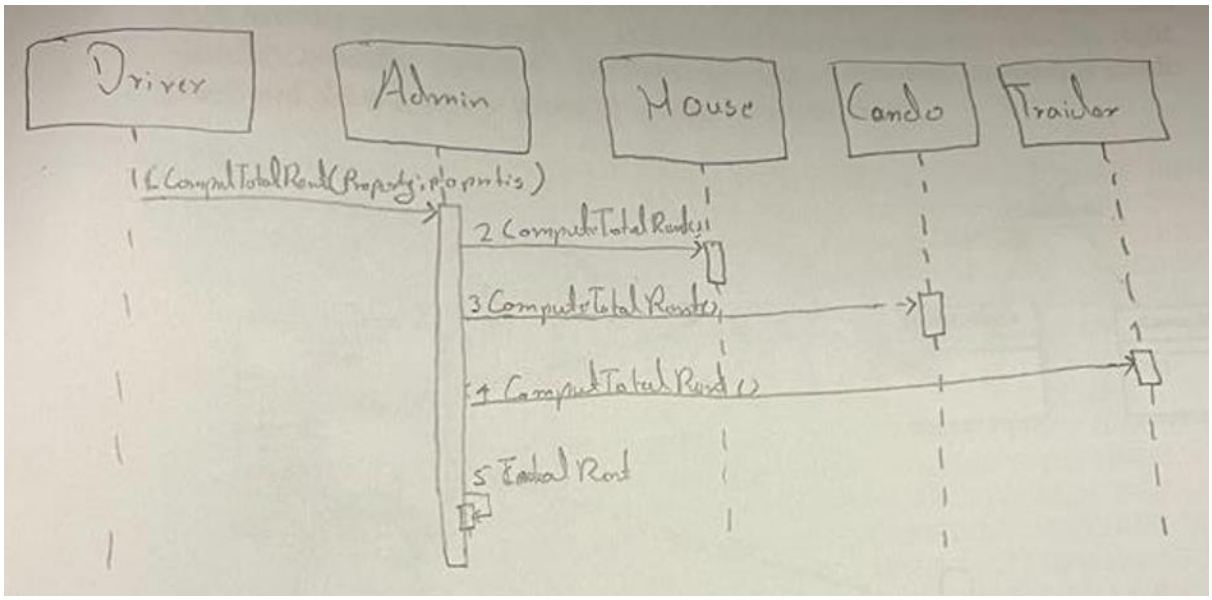
(iv) Changes/Implementation of other classes

```
public class Condo extends Property {
//internal code unchanged
}
```

```
public class House extends Property {
//internal code unchanged
}
```

```
public class Trailor extends Property {
//internal code unchanged
}
```

4. [4 points] Create a sequence diagram to model the dynamics of the problem given in Problem 3. Provide a diagram only for the main flow. In this case, the main flow will be the flow in which three properties - one house, one condominium, and one trailer - are passed to the computeTotalRent method in Admin (as shown in the Driver class implementation). Remember: sequence diagrams are concerned with run-time objects only.



5. [10 points] On the following two blank pages implement the UML design below into Java code. Show how the abstract Duck class uses the IFlyBehavior and IQuackBehavior interfaces to implement its fly and quack methods.

Implement the relationships properly. The inheritance between concrete ducks and the abstract duck superclass. The association between the abstract duck and the interfaces and the different interface realizations. Use the right visibility modifier for the FlyBehavior and OuackBehabior properties inside Duck and show how the concrete ducks initialize them.

```
public interface FlyBehavior {
	public void fly();
}

public interface QuackBehavior {
	public void quack();
}

public class FlyWithWings implements FlyBehavior {
	public void fly() {
		System.out.println("  fly with wings");
	}
}

public class CannotFly implements FlyBehavior {
	public void fly() {
		System.out.println("  cannot fly");
	}
}

public class Quack implements QuackBehavior {
	public void quack() {
		System.out.println("  quacking");
```

```
	}
}


public class MuteQuack implements QuackBehavior {
	public void quack() {
		System.out.println("  cannot quack");
	}
}


abstract public class Duck {
	private FlyBehavior flyBehavior;
	private QuackBehavior quackBehavior;
	abstract public void display();

	public void fly() {
		flyBehavior.fly();
	}
	public void quack() {
		quackBehavior.quack();
	}
	protected void setFlyBehavior(FlyBehavior b) {
		flyBehavior = b;
	}
	protected void setQuackBehavior(QuackBehavior b) {
		quackBehavior = b;
	}

	public void swim() {
```

```
            System.out.println("  swimming");
        }
}


public class MallardDuck extends Duck {
        public MallardDuck() {
            setQuackBehavior(new Quack());
            setFlyBehavior(new FlyWithWings());
        }
        @Override
        public void display() {
            System.out.println("  display");
        }
}


public class DecoyDuck extends Duck {
        public DecoyDuck() {
            setQuackBehavior(new MuteQuack());
            setFlyBehavior(new CannotFly());
        }
        @Override
        public void display() {
            System.out.println("  displaying");
        }
}
```