1. A **Money** object has an amount and a currency, which are just values. Draw a sequence diagram of what happens when **sale.computeVat(m)** is invoked with **Money** reference **m**. Show "computeVat(m)" as the "found" message.

```
class Sale {
     public Money computeVat( Money money ) {
          double amount = money.getAmount( );
          String currency = money.getCurrency();
          // create new object to represent VAT tax
          Money tax = new Money(amount * 0.07, currency);
          return tax;
     }
```

2. Draw a Sequence Diagram to show what happens when max(a, b) is invoked, where a and b refer to Double objects. "max(a,b)" is the "found message".

```
public Double max( Double x, Double y) {
     double vx = x.getDoubleValue( );
     double vy = y.getDoubleValue( );
     if (vx >= vy) return x;
     else return y;
}
```

3. Fill in the blanks to show how to use an *Iterator* to sum a *Collection* of coins.

```
public double sum( Collection<Coin> coins ) {
     _____________ iterator = coins._____________;
     double sum = 0.0;
     while ( ____________________ ) {
          sum += __________. ____________ . ___________________ ;
     }
     return sum;
}
```

4. Draw a UML sequence diagram for the previous exercise. You can use one Coin object to represent the coins that the iterator returns (even though in fact they would be different). Note that in a sequence diagram you don't have to show operations performed inside a method (like sum = sum + 1) but you *can* show them by writing the operation inside an oval next to the activation box.

Here is some code for a university Course Management application.

```
public class Registrar { /* a Singleton class */
     private static Registrar instance;
     private CourseCatalog catalog;
     private Registrar( ) {
          // code to initialize registrar and course catalog
     }
     public static Registrar getInstance( ) {
          return instance;
     }
     public CourseCatalog getCourseCatalog( ) {
          return catalog;
     }
}
public class CourseCatalog {
     private List<Course> courses;
     public Course getCourse(String courseId) {
          // code to find course in catalog is not shown
          Course course = . . .
          return course;
     }
}
public class Course {
     private String courseId;
     private int credits;
     private String name;
     public String getName() { return name; }
     public int credits( ) { return credits; }
     public String getCourseId() { return courseId; }
}
```

5. Draw a UML class diagram showing the relationships between classes.

6. Write Java code to get the name and number of credits for course 01204499. Be sure to test whether this course really exists!

7. Draw a Sequence Diagram of your code from previous problem.

The Course Management application also has these classes:

```
/** Enrollment represents a student enrolled in one course. */
public class Enrollment {
     private Course course;
     private Grade grade;
     private boolean dropped; // true if has dropped the course
     public Course getCourse( ) { return course; }
     public Grade getGrade( ) { return grade; }
     public boolean hasDropped( ) { return dropped; }
}
/** The possible grades and their grade point values. */
public enum Grade {
     A(4.0),
     BPLUS(3.5),
     B(3.0),
     CPLUS(2.5),
     C(2.0),
     DPLUS(1.5),
     D(1.0),
     F(0.0),
     W(0.0);
     private final double gradePoints;
     ________  Grade( _______________ ) {
          ________________________________________;
     }
     public double getGradePoints() { return gradePoints; }
}
```

8. Complete the code for the Grade enum.

9. A student has a List of Enrollment objects.

```
List<Enrollment> enrolled =
            Registrar.getInstance().getEnrollment("57105411111");
```

Write Java code to compute how many credits the student is taking. Don't count dropped courses.

10. Afer the semester is over, how would you compute the semester GPA for an enrollment?

```
public double computeGpa( Enrollment enrolled ) {
```

11. Draw a class diagram of relationships between all the classes in the course management system. You don't need to show attributes, just relationships. Include *names* and *multiplicity* where it makes sense.

12. We want the Grade enum to print grades like "A", "B+" rather than "BPLUS". Write a **`toString`** method for Grade using *only one statement* to create a String and replace "PLUS" with "+" where it occurs. Use the Enum's name() method and String replace(old,new) method.

KU Pizza Shop

The KU Pizza Shop sells 2 kinds of products: Pizza and Drinks. Both pizza and drink have a size. A pizza also has some toppings.

```
public class Pizza {
      private int size;        // 0=nothing, 1=small, 2 = medium, 3=large
      private double [] prices = {0, 120, 160, 200}; // price of each size
      private List<String> toppings;
      public Pizza(int size) { ... }
      public double getPrice() {
            // price depends on the size and number of toppings (30. each)
            return prices[size] + 30*toppings.size();
      }
      public void setSize(int size) { this.size = size; }
      public void addTopping(String topping) { ... }
      public String toString() { /* describes this pizza */ }
}
```

```
public class Drink {
      private int size;       // 1=small, 2 = medium, 3=large
      private String flavor; // "Lime", "Cola", etc.
      private double [] prices = {0, 16, 22, 28}; // price of each size
      public Drink(String flavor) { ... }
      public double getPrice() {
            return prices[size]; // price depends only on the size
      }
      public void setSize(int size) { this.size = size; }
      public String toString() { /* describes this drink */ }
}
```

A customer's order is put in a FoodOrder object like this:

```
public class FoodOrder {
      // pizzas in this order
      private List<Pizza> pizzas;
      // drinks in this order
      private List<Drink> drinks;
      public FoodOrder() {
            pizzas = new ArrayList<Pizza>( );
            drinks = new ArrayList<Drink>( );
      }
      public void addPizza( Pizza pizza ) { pizzas.add( pizza ); }
      public void addDrink( Drink drink ) { drinks.add( drink ); }
      public double getTotal() {
            double total = 0;
            for(Pizza p: pizzas) total += p.getPrice();
            for(Drink d: drinks) total += d.getPrice();
            return total;
      }
      public void printOrder() {
            for(Pizza p: pizzas) System.out.println( p.toString() );
            for(Drink d: drinks) System.out.println( d.toString() );
            System.out.println("Total price: " + getTotal() );
      }
}
```

Your friend, Miss Poly Morphism, thinks that the FoodOrder code is too complex. It handles pizza and drinks separately, but actually it invokes exactly the same methods on both Pizza and Drink.

Apply some O-O principles to simplify the code.

13. Define an Interface named OrderItem for the essential behavior of both Pizza and Drink. Draw a UML diagram of the design.

14. How would you modify the Pizza class to use this interface? Write only the line(s) you must change in Pizza.

15. Write the new code for FoodOrder than uses the interface and polymorphism.

```
public class FoodOrder
{     ____________________________

      public void addOrderItem (                          ) {



      }

      public double getTotal (     ) {




      }
      // code for printOrder omitted
}
```

16. Miss Polly's friend, Ab Strack, thinks you can make the code even better -- there is still *duplicate code* in Pizza and Drink classes. Design an *abstract superclass* for Pizza and Drink and move common code to the abstract class. Name the class ***`AbstractItem`***..

Draw a **UML class diagram** of the new design showing the relationships between *`OrderItem`*, *`AbstractItem`*, `Pizza`, `Drink`, and `FoodOrder`.

17. Write Java code for **`Drink`** that uses ***`AbstractItem`***.. The code should be much simpler.

18. Write Java code for this sequence diagram.