1. Provide definitions for the following terms. How does each of the terms apply to OO notion of classes? Provide examples of both good and bad uses of the terms in a design of a class or a set of classes.

   (a) (15 points) Abstraction

   (b) (15 points) Encapsulation

   (c) (15 points) Cohesion

   (d) (15 points) Coupling

(a) Abstraction is the process of hiding implementation details and exposing only the relevant features of an object. It allows a class to present an interface without revealing the internal complexities.

Good

```
abstract class Shape {
    abstract double calculateArea();
}
class Circle extends Shape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

Define a clear interface, hides unnecessary details

Bad

```
class Circle {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
    public void draw() {
        System.out.println("draw a circle with radius" + radius);
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public static void main(String[] args) {
        Circle circle = new Circle(5.0);
        circle.draw();
        circle.printDetails();
    }
}
```

The class does too much. It should focus on representing a circle, but it also handles drawing and printing, violating the single responsibility principle and reducing abstraction.

(b) Encapsulation is the bunding of data (variables) and methods (functions) that operate on the data into a single unit (class) while restricting direct access to some details of the object.

Good

```
class BankAccount {
    private double balance;
    public BankAccount(double initialBalance) {
        this.balance = Math.max(initialBalance, 0);
    }
    public void deposit(double amount) {
        if (amount > 0) balance += amount;
    }
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) balance -= amount;
    }
}
```

uses private fields with controlled access methods

Bad

```
class BankAccount {
    public double balance;
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }
```

Allows direct modification of internal fields.

(c) Cohesion measures how closely related and focused the responsibilities of a class are. High cohesion means a class is well-defined and performs a single function effectively. Low cohesion means a class handles too many unrelated tasks.

Good

```
class order {
  private List<String> items = new ArrayList<>();
  public void addItem (String item){
    items.add (item);
  }
  public int getTotalItems (){
    return items.size();
  }
}
```

Keeps classes focused on a single responsibility

Bad

```
class Order {
  private List<String> items = new ArrayList<>();
  public void addItem (String item){
    items.add (item);
  }
  public int getTotalItems () {
    return items.size();
  }
  public void sendEmailConfirmation() {
    System.out.println ("Email sent!");
  }
}
```

Mixes unrelated responsibilities in one class.

(d) Coupling refers to the degree of dependency between classes. Low coupling is preferred because it reduces the impact of changes in one class on others, making the system more modular and easier to maintain.

Good

```
class EmailService {
  public void sendEmail (String message){
    System.out.println ("Sent Email = " + message);
  }
}
class Order {
  private EmailService emailService;
  public Order (EmailService emailService){
    this.emailService = emailService;
  }
  public void confirmOrder (){
    system.out.println ("Order confirmed");
  }
}
EmailService emailService = new EmailService();
Order order = new Order(emailService);
order.confirmOrder ();
```

Use interfaces and dependency injection.

Bad

```
class Order {
  public void confirmOrder() {
    System.out.println ("Order confirmed!");
    System.out.println ("Sent Email...");
  }
}
```

Directly depends on specific implementations.