Question 1

a) What is Object-Oriented Programming? How does it differ from Procedure-Oriented Programming?

Object-Oriented Programming (OOP):
OOP is a programming paradigm based on the concept of "objects," which are instances of classes. These objects can contain data in the form of fields (attributes or properties) and code in the form of methods (functions or procedures). OOP aims to implement real-world entities like inheritance, polymorphism, encapsulation, and abstraction.

Differences from Procedure-Oriented Programming (POP):

- **Design Approach:**
  - **OOP:** Focuses on objects that encapsulate data and behavior.
  - **POP:** Focuses on functions and procedures that operate on data.
- **Data Handling:**
  - **OOP:** Data is encapsulated within objects and protected through access modifiers.
  - **POP:** Data is generally global and can be accessed by any function.
- **Modularity:**
  - **OOP:** Encourages modularity through classes and objects.
  - **POP:** Functions are the primary units of modularity.
- **Reusability:**
  - **OOP:** Promotes reusability through inheritance and polymorphism.
  - **POP:** Reusability is achieved by writing reusable functions, but it is less flexible compared to OOP.

b) Define the following terminologies as used in Object-Oriented Programming and provide an example of each:

1. **Instantiation:**
   Instantiation is the process of creating an object from a class. When a class is defined, no memory is allocated until an object of that class is created.
   - **Example:** `MyClass obj = new MyClass();`

2. **Instance Variable:**
   An instance variable is a variable defined in a class for which each instantiated object of the class has a separate copy.
   - **Example:**

```
class MyClass {
    int instanceVariable;
}
MyClass obj = new MyClass();
obj.instanceVariable = 5;
```

3. **Method Overriding:**
   Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The overridden method in the subclass should have the same name, return type, and parameters as in the superclass.
   - **Example:**

```
class Animal {
    void makeSound() {
        System.out.println("Animal sound");
    }
}
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Bark");
    }
}
```

4. **Single Inheritance:**
   Single inheritance is a feature of OOP languages where a class can inherit properties and methods from only one superclass.
   - **Example:**

```
class ParentClass {
    void display() {
        System.out.println("Parent Class");
    }
}
class ChildClass extends ParentClass {
    // Inherits display() method from ParentClass
}
```

5. **Serialized Object:**
   Serialization is the process of converting an object's state into a byte stream, so it can be saved to a file, sent over a network, or stored in a database. Deserialization is the reverse process, converting the byte stream back into an object.
   - **Example:**

```
import java.io.*;

class MyClass implements Serializable {
    int instanceVariable;
}

public class SerializeDemo {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.instanceVariable = 5;

        try {
            FileOutputStream fileOut = new FileOutputStream("object.s
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(obj);
            out.close();
            fileOut.close();
        } catch (IOException i) {
            i.printStackTrace();
        }
    }
}
```

c) Write a Java method to receive the Gross Salary of an employee as argument, calculate and return the Tax Payable for an employee.

```java
public class TaxCalculator {

    public static double calculateTax(double grossSalary) {
        double tax;
        if (grossSalary < 25000) {
            tax = 0;
        } else if (grossSalary <= 50000) {
            tax = grossSalary * 0.10;
        } else {
            tax = grossSalary * 0.15;
        }
        return tax;
    }

    public static void main(String[] args) {
        double grossSalary = 30000; // Example salary
        double taxPayable = calculateTax(grossSalary);
        System.out.println("The tax payable is: " + taxPayable);
    }
}
```

Question 2

a) Why is an array such an inefficient data structure for a dynamic sorted list?

An array is inefficient for a dynamic sorted list for several reasons:

1. **Fixed Size:** Arrays have a fixed size, which means that if the array is full and you need to add more elements, you must create a new, larger array and copy all the

elements to the new array, which is time-consuming.

2. **Insertion:** Inserting an element into a sorted array requires shifting elements to maintain order, leading to a time complexity of O(n) in the worst case.
3. **Deletion:** Similarly, deleting an element from a sorted array also requires shifting elements, resulting in O(n) time complexity.

b) If linked lists are so much better than arrays, why are arrays used at all?

Despite their limitations, arrays are used because:

1. **Direct Access:** Arrays allow direct access to any element using its index, providing O(1) time complexity for accessing elements.
2. **Memory Efficiency:** Arrays have a lower memory overhead compared to linked lists since they do not require additional memory for pointers.
3. **Cache Performance:** Arrays have better cache locality, which can lead to significant performance improvements in certain applications.

c) Complete the following method to print all the elements of a collection using the Iterator interface.

```
void print(Collection c) {
    Iterator iterator = c.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

d) Trace the following code, showing the contents of the stack after each invocation:

```
Stack stack = new Stack();

stack.push(new Character('A')); // Stack: A
stack.push(new Character('B')); // Stack: A, B
stack.push(new Character('C')); // Stack: A, B, C
stack.pop();                    // Stack: A, B
stack.pop();                    // Stack: A
stack.push(new Character('D')); // Stack: A, D
stack.push(new Character('E')); // Stack: A, D, E
stack.push(new Character('F')); // Stack: A, D, E, F
stack.pop();                    // Stack: A, D, E
stack.push(new Character('G')); // Stack: A, D, E, G
stack.pop();                    // Stack: A, D, E
stack.pop();                    // Stack: A, D
stack.pop();                    // Stack: A
```

Trace:

1. After `stack.push(new Character('A'));` - Stack: [A]
2. After `stack.push(new Character('B'));` - Stack: [A, B]
3. After `stack.push(new Character('C'));` - Stack: [A, B, C]
4. After `stack.pop();` - Stack: [A, B]
5. After `stack.pop();` - Stack: [A]
6. After `stack.push(new Character('D'));` - Stack: [A, D]
7. After `stack.push(new Character('E'));` - Stack: [A, D, E]
8. After `stack.push(new Character('F'));` - Stack: [A, D, E, F]
9. After `stack.pop();` - Stack: [A, D, E]
10. After `stack.push(new Character('G'));` - Stack: [A, D, E, G]
11. After `stack.pop();` - Stack: [A, D, E]
12. After `stack.pop();` - Stack: [A, D]
13. After `stack.pop();` - Stack: [A]

Question 3

a) Write a Java program to create a base class Animal and declare two (2) data members.

```
class Animal {
    String name;
    int age;

    // Constructor
    Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Member function
    void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

b) Use the class created to demonstrate inheritance. In order to do this, derive a class called Dog from class Animal and then another class called Labrador from class Dog. Give data members, constructor(s), and member functions as necessary.

```
// Dog class inheriting from Animal class
class Dog extends Animal {
    String breed;

    // Constructor
    Dog(String name, int age, String breed) {
        super(name, age);
        this.breed = breed;
    }

    // Member function
    void displayDogInfo() {
        displayInfo();
        System.out.println("Breed: " + breed);
    }
}

// Labrador class inheriting from Dog class
class Labrador extends Dog {
    String color;

    // Constructor
    Labrador(String name, int age, String breed, String color) {
        super(name, age, breed);
        this.color = color;
    }

    // Member function
    void displayLabradorInfo() {
        displayDogInfo();
        System.out.println("Color: " + color);
    }
}
```

c) Use the Main() method to demonstrate object creation, access to data members, and invoking member functions.

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Labrador class
        Labrador labrador = new Labrador("Buddy", 3, "Labrador Retriever", "Yellow");

        // Accessing data members and invoking member functions
        labrador.displayLabradorInfo();
    }
}
```

Explanation:

1. **Animal Class:** The base class `Animal` is created with two data members: `name` and `age`. A constructor is defined to initialize these data members, and a member function `displayInfo` is provided to display the information.
2. **Dog Class:** The `Dog` class inherits from `Animal` and adds an additional data member `breed`. A constructor is provided to initialize all data members, and a member function `displayDogInfo` is defined to display the dog's information.
3. **Labrador Class:** The `Labrador` class inherits from `Dog` and adds an additional data member `color`. A constructor is provided to initialize all data members, and a member function `displayLabradorInfo` is defined to display the Labrador's information.
4. **Main Method:** In the `Main` class, an object of the `Labrador` class is created. The data members are accessed, and the member functions are invoked to demonstrate inheritance and polymorphism.

Question 4

a) What is an interface? (3 marks)

An interface in Java is a reference type, similar to a class, that can contain only constants,

method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or methods that provide implementation. They are used to specify a set of methods that a class must implement, thereby providing a way to achieve abstraction and multiple inheritance in Java.

b) Give the differences between an interface and a class. (6 marks)

1. Method Implementation:
    - Class: Can have both method declarations and method implementations.
    - Interface: Can only have method declarations (abstract methods) by default, although it can have default methods and static methods with implementations.
2. Multiple Inheritance:
    - Class: Supports single inheritance (a class can inherit from only one class).
    - Interface: Supports multiple inheritance (a class can implement multiple interfaces).
3. Fields:
    - Class: Can have instance fields (variables).
    - Interface: Can only have static final fields (constants).
4. Access Modifiers:
    - Class: Can have constructors and can use various access modifiers (public, protected, private).
    - Interface: Does not have constructors. All methods and fields are implicitly public (methods are public and abstract by default, fields are public, static, and final).
5. Inheritance:
    - Class: Uses the `extends` keyword for inheritance.
    - Interface: Uses the `implements` keyword for a class to implement an interface, and `extends` keyword if an interface is inheriting another interface.
6. Instantiation:
    - Class: Can be instantiated using the `new` keyword.
    - Interface: Cannot be instantiated directly; must be implemented by a class to create objects.

c) The following figure shows the documentation for the ActionListener interface.

i. Explain what information the Interface Declaration section is giving. (3 marks)

The Interface Declaration section provides the signature of the interface `ActionListener`. It indicates that `ActionListener` is a public interface that extends another interface called `EventListener`. This means `ActionListener` inherits the characteristics of `EventListener` and is meant to handle events, specifically to define the `actionPerformed` method that needs to be implemented by any class that uses this interface.

ii. Explain what information the Interface Method section is giving and show an example how you would use it. (3 marks)

The Interface Method section describes the method signature of the `actionPerformed` method. It indicates that any class implementing the `ActionListener` interface must define this method. The method takes a single parameter of type `ActionEvent` and returns void. It is invoked when an action occurs.

Example:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

public class MyActionListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ActionListener Example");
        JButton button = new JButton("Click Me");

        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button was clicked!");
            }
        });

        frame.add(button);
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

d) The program below shows the use of an Interface. Complete the method CompareTo so that the ArrayList is displayed in REVERSED order. (10 marks)

To display the ArrayList in reversed order, we can modify the `compareTo` method in the `Student` class to reverse the natural ordering.

Modified Code:

```
class Student implements Comparable<Student> {
    int rollno;
    String name;
    int age;

    Student(int rollno, String name, int age) {
        this.rollno = rollno;
        this.name = name;
        this.age = age;
    }

    public int compareTo(Student st) {
        // Reverse the natural order by switching the comparison
        return st.rollno - this.rollno;
    }

    public static void main(String args[]) {
        ArrayList<Student> al = new ArrayList<Student>();
        al.add(new Student(101, "Vijay", 23));
        al.add(new Student(106, "Ajay", 27));
        al.add(new Student(105, "Jai", 21));

        Collections.sort(al);
        for (Student st : al) {
            System.out.println(st.rollno + " " + st.name + " " + st.age);
        }
    }
}
```

In this code, the `compareTo` method is implemented to compare `Student` objects based on their `rollno` in reversed order by subtracting `this.rollno` from `st.rollno`. When the `Collections.sort(al)` method is called, it sorts the `ArrayList` in descending order based on `rollno`.