Abstraction and Polymorphism in Java

To demonstrate **Abstraction** and **Polymorphism** in Java, we can use a simple example of an **Employee Management System**. In this example:

1. **Abstraction** is implemented by using an abstract class called `Employee` which defines general characteristics of an employee.
2. **Polymorphism** is shown by creating different types of employees (`FullTimeEmployee` and `PartTimeEmployee`) and treating them as their abstract type (`Employee`).

Here's how we can set this up.

Code Example: Employee Management System

```
// Abstract class representing a generic Employee
abstract class Employee {
    protected String name;
    protected int employeeId;

    // Constructor to initialize common properties of employe
es
    public Employee(String name, int employeeId) {
        this.name = name;
        this.employeeId = employeeId;
    }

    // Abstract method to calculate salary (to be implemented
by subclasses)
    public abstract double calculateSalary();

    // Method to display employee details
    public void displayDetails() {
        System.out.println("Employee ID: " + employeeId);
        System.out.println("Name: " + name);
        System.out.println("Salary: " + calculateSalary());
    }
}

// Class representing a Full-Time Employee
class FullTimeEmployee extends Employee {
    private double monthlySalary;

    // Constructor for FullTimeEmployee
    public FullTimeEmployee(String name, int employeeId, doub
le monthlySalary) {
        super(name, employeeId);
        this.monthlySalary = monthlySalary;
    }

    // Overridden method to calculate salary for FullTimeEmpl
oyee
    @Override
    public double calculateSalary() {
        return monthlySalary;
    }
}

// Class representing a Part-Time Employee
class PartTimeEmployee extends Employee {
    private double hourlyWage;
    private int hoursWorked;

    // Constructor for PartTimeEmployee
    public PartTimeEmployee(String name, int employeeId, doub
le hourlyWage, int hoursWorked) {
        super(name, employeeId);
        this.hourlyWage = hourlyWage;
        this.hoursWorked = hoursWorked;
    }

    // Overridden method to calculate salary for PartTimeEmpl
oyee
    @Override
    public double calculateSalary() {
        return hourlyWage * hoursWorked;
    }
}

// Main class to demonstrate Abstraction and Polymorphism
public class EmployeeManagement {
    public static void main(String[] args) {
        // Create instances of FullTimeEmployee and PartTimeE
mployee
        Employee fullTimeEmp = new FullTimeEmployee("Alice",
101, 5000);
        Employee partTimeEmp = new PartTimeEmployee("Bob", 10
2, 20, 80);

        // Display details using polymorphism
        System.out.println("Full-Time Employee Details:");
        fullTimeEmp.displayDetails();

        System.out.println("\\nPart-Time Employee Details:");
        partTimeEmp.displayDetails();
    }
}
```

Explanation

1. **Abstraction**:

    - The `Employee` class is an **abstract class** that represents the general concept of an employee. It has an abstract method `calculateSalary()` which must be implemented by any subclass. This abstract method forces subclasses to define their specific way of calculating salary.

- This `Employee` class provides a blueprint for creating different types of employees but does not implement the specifics of calculating the salary.

2. **Polymorphism**:

- We create instances of `FullTimeEmployee` and `PartTimeEmployee`, but they are both referenced as `Employee` objects.
- Using polymorphism, we call the `displayDetails()` method on each `Employee` object, and it correctly invokes the `calculateSalary()` method defined in each subclass.
- This approach allows treating all types of employees in a uniform way while ensuring that each type calculates its salary differently.

Sample Output:

```
Full-Time Employee Details:
Employee ID: 101
Name: Alice
Salary: 5000.0

Part-Time Employee Details:
Employee ID: 102
Name: Bob
Salary: 1600.0
```

Key Concepts Demonstrated:

1. **Abstraction** allows us to define a general template (`Employee`) that can be applied to all employees, but the specifics (like how salary is calculated) are implemented in each subclass.
2. **Polymorphism** enables treating `FullTimeEmployee` and `PartTimeEmployee` objects as `Employee` objects. The correct `calculateSalary()` method is called based on the actual object type at runtime, showing dynamic behavior.

This example is simple yet effective for illustrating the foundational OOP principles of **Abstraction** and **Polymorphism** in Java.