**1. Define a Service Interface (Abstraction/Polymorphism)**

public interface TelecomService {

void start(); // start the service

void stop(); // stop it

void billCustomer(int units);

}

* **Why use interface?** It abstracts the *what* without the *how*, enabling different service types under one.

**2. Abstract Base Class for Shared Logic (Abstraction/Inheritance)**

public abstract class BaseService implements TelecomService {

protected String serviceName;

protected double ratePerUnit;

public BaseService(String serviceName, double rateRatePerUnit) {

this.serviceName = serviceName;

this.ratePerUnit = rateRatePerUnit;

}

@Override

public void start() {

System.out.println(serviceName + " started.");

}

@Override

public void stop() {

System.out.println(serviceName + " stopped.");

}

@Override

public void billCustomer(int units) {

double amount = units \* ratePerUnit;

processPayment(units, amount);

}

// Abstract: subclasses provide their own payment routing logic

protected abstract void processPayment(int units, double amount);

}

* **Abstract class** gives shared implementation and defines an abstract method (processPayment) for specialized behavior.

**3. Concrete Implementations (Inheritance, Polymorphism)**

public class SMSService extends BaseService {

public SMSService(double rate) {

super("SMS Service", rate);

}

@Override

protected void processPayment(int units, double amount) {

System.out.printf("Billing SMS: %d msgs @ ₹%.2f = ₹%.2f%n", units, ratePerUnit, amount);

}

}

public class VoiceCallService extends BaseService {

public VoiceCallService(double rate) {

super("Voice Call Service", rate);

}

@Override

protected void processPayment(int units, double amount) {

System.out.printf("Billing Calls: %d mins @ ₹%.2f = ₹%.2f%n", units, ratePerUnit, amount);

}

}

* Both **inherit** shared logic.
* **Polymorphism**: You can treat both as TelecomService.

**4. Service Manager Demonstrates Polymorphism**

public class TelecomApp {

public static void main(String[] args) {

TelecomService sms = new SMSService(1.5); // ₹1.5 per SMS

TelecomService voice = new VoiceCallService(2.0);// ₹2.0 per minute

runService(sms, 10);

runService(voice, 5);

}

private static void runService(TelecomService service, int units) {

service.start();

service.billCustomer(units);

service.stop();

System.out.println();

}

}

**Output:**

SMS Service started.

Billing SMS: 10 msgs @ ₹1.50 = ₹15.00

SMS Service stopped.

Voice Call Service started.

Billing Calls: 5 mins @ ₹2.00 = ₹10.00

Voice Call Service stopped.

* Notice **runService** works generically with any TelecomService—that's runtime polymorphism.

**Concepts Recap:**

| **OOP Principle** | **Where It Appears** |
| --- | --- |
| **Abstraction** | TelecomService interface hides implementation detail |
| **Inheritance** | BaseService → SMSService, VoiceCallService for shared behavior |
| **Polymorphism** | runService(TelecomService...) handles different services |
| **Abstraction via Abstract Class** | BaseService.processPayment(...) is partially abstract |

**Extensions You Could Add:**

* A new service type, e.g., DataService, just by subclassing BaseService.
* Use multiple interfaces, e.g., Taxable or Discountable.
* Introduce the **Open/Closed Principle**: add new service type **without** modifying existing code.

This code gives you a clean, modular telecom-services simulation in Java—ideal for demonstrating key OOP concepts in your presentation. If you’d like, I can expand it with more features like configuration, unit tests, or even a simple billing UI.

Here’s a refined Java example simulating a telecom service system, showcasing meaningful use of **interfaces**, **abstraction**, **inheritance**, and **polymorphism**:

**1. Define the Common Interface (Full Abstraction & Polymorphism)**

public interface TelecomService {

void start();

void stop();

double billCustomer(int units);

}

* **Interface** enforces a contract for all services, abstracting implementation and enabling polymorphism.

**2. Abstract Base Class (Partial Abstraction & Shared Logic)**

public abstract class BaseService implements TelecomService {

protected final String serviceName;

protected final double ratePerUnit;

protected BaseService(String name, double rate) {

this.serviceName = name;

this.ratePerUnit = rate;

}

@Override

public void start() { System.out.println(serviceName + " started."); }

@Override

public void stop() { System.out.println(serviceName + " stopped."); }

@Override

public double billCustomer(int units) {

double amount = units \* ratePerUnit;

processPayment(units, amount);

return amount;

}

protected abstract void processPayment(int units, double amount);

}

* Leverages **inheritance** for shared logic.
* **Abstract method** processPayment(...) ensures derived classes implement billing logic specifically.

**3. Concrete Service Classes (Inheritance & Polymorphism)**

public class SMSService extends BaseService {

public SMSService(double rate) {

super("SMS Service", rate);

}

@Override

protected void processPayment(int units, double amount) {

System.out.printf("Billing %d SMS @ ₹%.2f each = ₹%.2f%n",

units, ratePerUnit, amount);

}

}

public class VoiceCallService extends BaseService {

public VoiceCallService(double rate) {

super("Voice Call Service", rate);

}

@Override

protected void processPayment(int units, double amount) {

System.out.printf("Billing %d minutes calls @ ₹%.2f = ₹%.2f%n",

units, ratePerUnit, amount);

}

}

* Both **inherit** and extend BaseService.
* **Polymorphism** means code treating services uniformly but invoking unique behaviors.

**4. Using Services in the Client (Runtime Polymorphism)**

public class TelecomClient {

public static void main(String[] args) {

TelecomService[] services = {

new SMSService(1.0), // ₹1/SMS

new VoiceCallService(2.5) // ₹2.5/min

};

for (TelecomService svc : services) {

svc.start();

double amount = svc.billCustomer(10);

System.out.printf("Total: ₹%.2f%n", amount);

svc.stop();

System.out.println();

}

}

}

**Output:**

SMS Service started.

Billing 10 SMS @ ₹1.00 each = ₹10.00

Total: ₹10.00

SMS Service stopped.

Voice Call Service started.

Billing 10 minutes calls @ ₹2.50 = ₹25.00

Total: ₹25.00

Voice Call Service stopped.

* **Runtime polymorphism** picks the correct processPayment(...) based on actual service type.

**Concept Recap**

| **OOP Principle** | **Where It's Used** |
| --- | --- |
| **Abstraction** | TelecomService interface hides implementation details |
| **Inheritance** | SMSService, VoiceCallService extend BaseService |
| **Polymorphism** | TelecomService[] services works with all types using same calls |
| **Abstract class** | Shared logic + abstract method in BaseService |

**Potential Extensions**

* Add **DataService** for billing GB usage.
* Implement **discounts**, **taxes**, or permission via additional interfaces (e.g., Discountable).
* Use a **factory pattern** to instantiate services—supporting Open/Closed principle.
* Build a **billing engine** that reads units from usage logs and invoices customers.

This example demonstrates a clean, modular telecom simulation in Java—ideal for presentations or assignments to illustrate core OOP principles in a real-world domain. Would you like to see tests, exception handling, or expanded features next?