实验名称：\_\_UML类图的Java代码实现\_\_\_

姓名：胡鑫宇 学号：2024303434

1. 实验环境

- OS：Windows 11

- 工具版本：JDK 1.8，IDEA

1. 实验过程

1.分析智能家居生态系统"HomeSphere"的需求描述，使用Java编程语言逐步编程实现给定类图（UML类图.jpg），完成各个类的Java代码实现，同时遵循Java编程规范，为撰写的类提供相应的Javadoc注释。

2.对于类图中涉及的equals方法，判断两个对象的id是否相等作为比较原则。

3.对于类图中涉及的toString方法，只需返回基本类型的属性数据，格式如下（属性值为字符串时加西文单引号）：

类名{属性1=属性1值, 属性2='属性2值',...}

例如：

Device{deviceId=1, name='Living Room AC', isOnline=true, powerStatus=true}

4.在IDE中导入junit.jar包（详见实验二附件），使用提供的HomeSphereSystemTest.java（详见实验二附件）测试程序是否正确。提供的测试文件HomeSphereSystemTest.java一共包含17个测试函数，每项功能测试逻辑如下所示：

1. testHouseholdCreation

测试逻辑：调用Household类的构造函数，验证对象实例不为空，getHouseholdId()和getAddress()方法返回结果符合预期。

1. testHouseholdRoomManagement

测试逻辑：分别调用Household类的addRoom()方法和removeRoom()方法，验证getRooms()方法返回结果是否符合预期。

1. testHouseholdUserManagement

测试逻辑：分别调用Household类的addUser()方法和removeUser()方法，验证getUsers()方法返回结果是否符合预期。

1. testManufacturer

测试逻辑：创建Manufacturer实例，验证getManufacturerId()、getName()、getProtocols()方法返回结果符合预期。

1. testHouseholdListAllDevices

测试逻辑：在隶属于Household的不同房间中分别添加设备，调用Household的listAllDevices()方法，验证返回结果是否为各房间所添加的设备。

1. testRoomDeviceManagement

测试逻辑：分别调用Room类的addDevice()方法和removeDevice()方法，验证getDevices()方法返回结果是否符合预期。

1. testDevicePowerManagement

测试逻辑：依次调用Device类的powerOn()方法和powerOff()方法，验证isPowerStatus()方法返回结果是否符合预期。

1. testAirConditionerTemperature

测试逻辑：初始设置AirConditioner类的setCurrTemp()、验证getCurrTemp()方法返回结果是否符合预期；调用setTargetTemp()方法设置目标温度，验证getTargetTemp()方法返回结果是否符合预期。

1. testLightBulbSettings

测试逻辑：调用LightBulb类的setBrightness()方法设置亮度，调用setColorTemp()方法设置色温，验证getBrightness()方法和getColorTemp()方法返回结果是否符合预期。

1. testSmartLock

测试逻辑：调用SmartLock类的setLocked()方法设置锁状态，setBatteryLevel()方法设置电量，验证isLocked()方法和getBatteryLevel()方法返回结果是否符合预期。

1. testBathroomScale

测试逻辑：调用BathroomScale类的setBodyMass()方法设置体重，setBatteryLevel()方法设置电量，验证getBodyMass()方法和getBatteryLevel()方法返回结果是否符合预期。

1. testRunningLog

测试逻辑：创建RunningLog实例，验证getDateTime()、getEvent()、getType()、getNote()方法返回结果符合预期。

1. testEnergyReporting

测试逻辑：验证AirConditioner类和LightBulb类实现了EnergyReporting接口，SmartLock类和BathroomScale类不实现EnergyReporting接口；验证AirConditioner类和LightBulb类正确实现getPower()方法，且返回结果符合预期；验证AirConditioner类和LightBulb类正确实现getReport()方法，且返回结果符合预期。

1. testDeviceAction

测试逻辑：创建DeviceAction实例，设置空调目标温度为20.0度，验证getCommand()、getParameters()、getDevice()方法返回结果符合预期；调用execute()方法，验证空调目标温度符合预期。

1. testAutomationScene

测试逻辑：创建AutomationScene实例，验证getSceneld()、getName()、getDescription()方法返回结果符合预期；在场景中添加DeviceAction实例，验证getActions()方法返回结果符合预期；调用manualTrig()方法，验证手动触发场景是否符合预期。

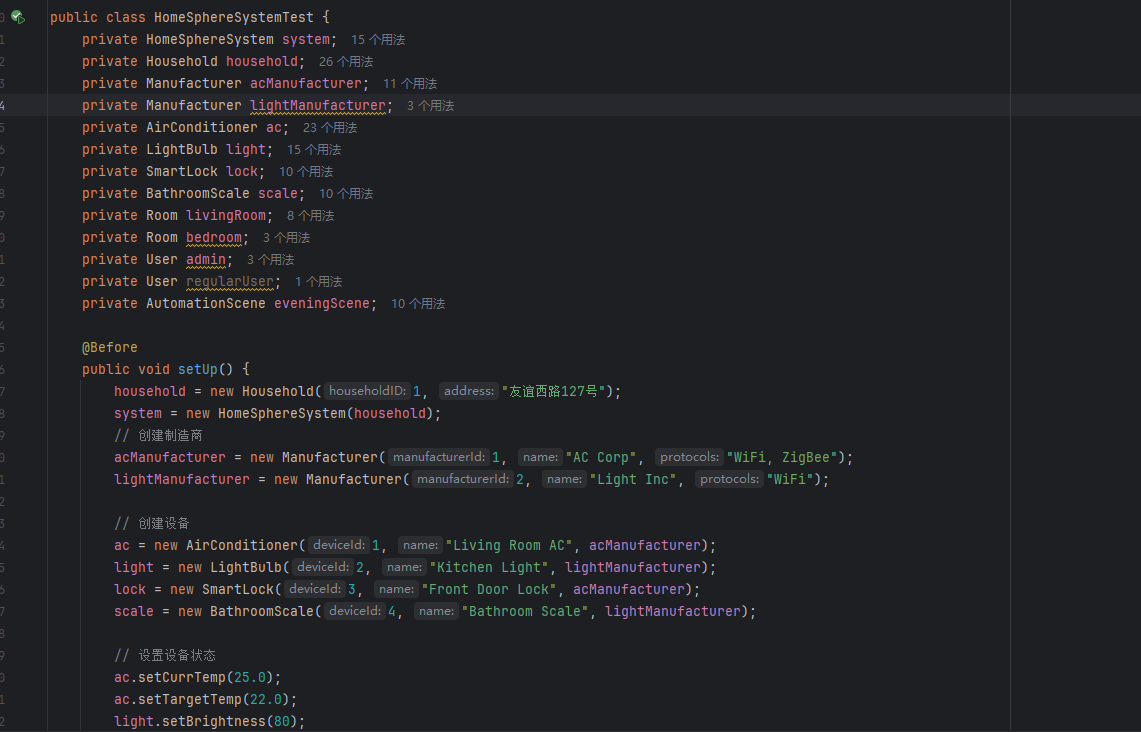
1. testHouseholdAutomationSceneManagement

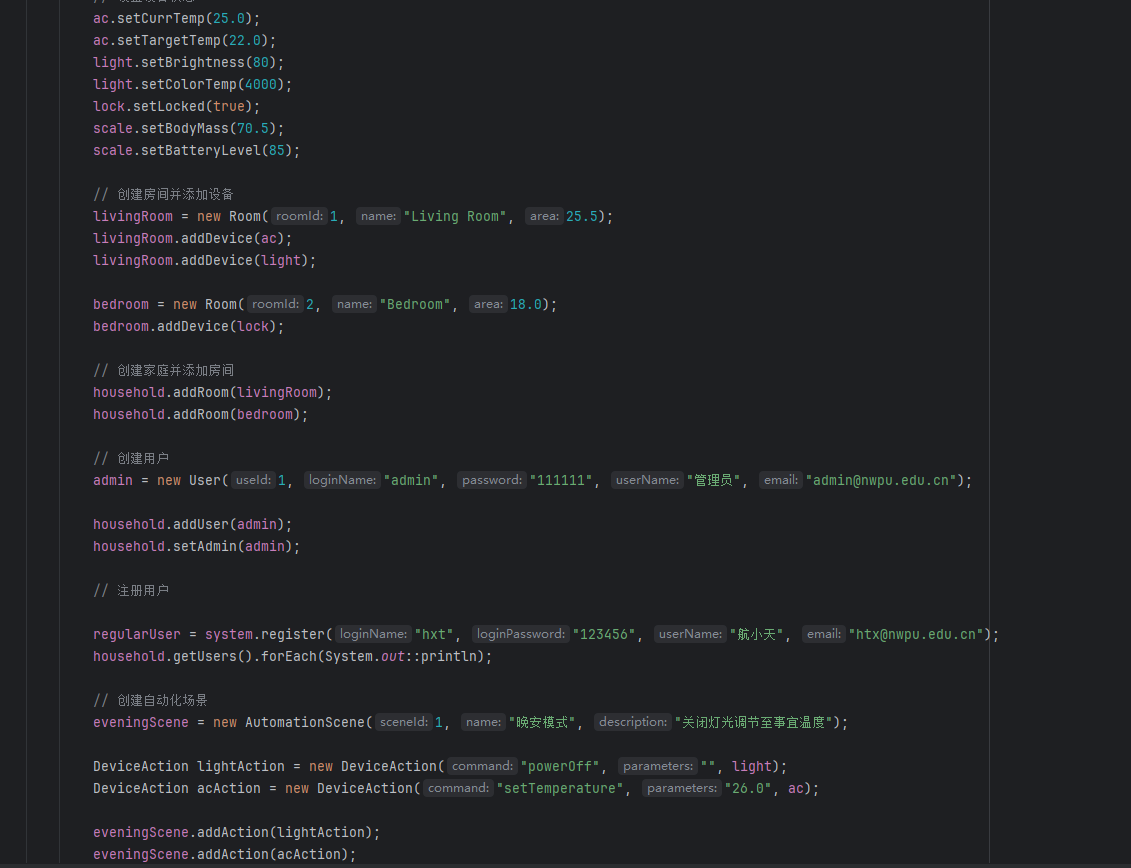
测试逻辑：分别调用Household类的addAutoScene()方法和removeAutoScene()方法，验证getAutoScenes()方法返回结果是否符合预期。

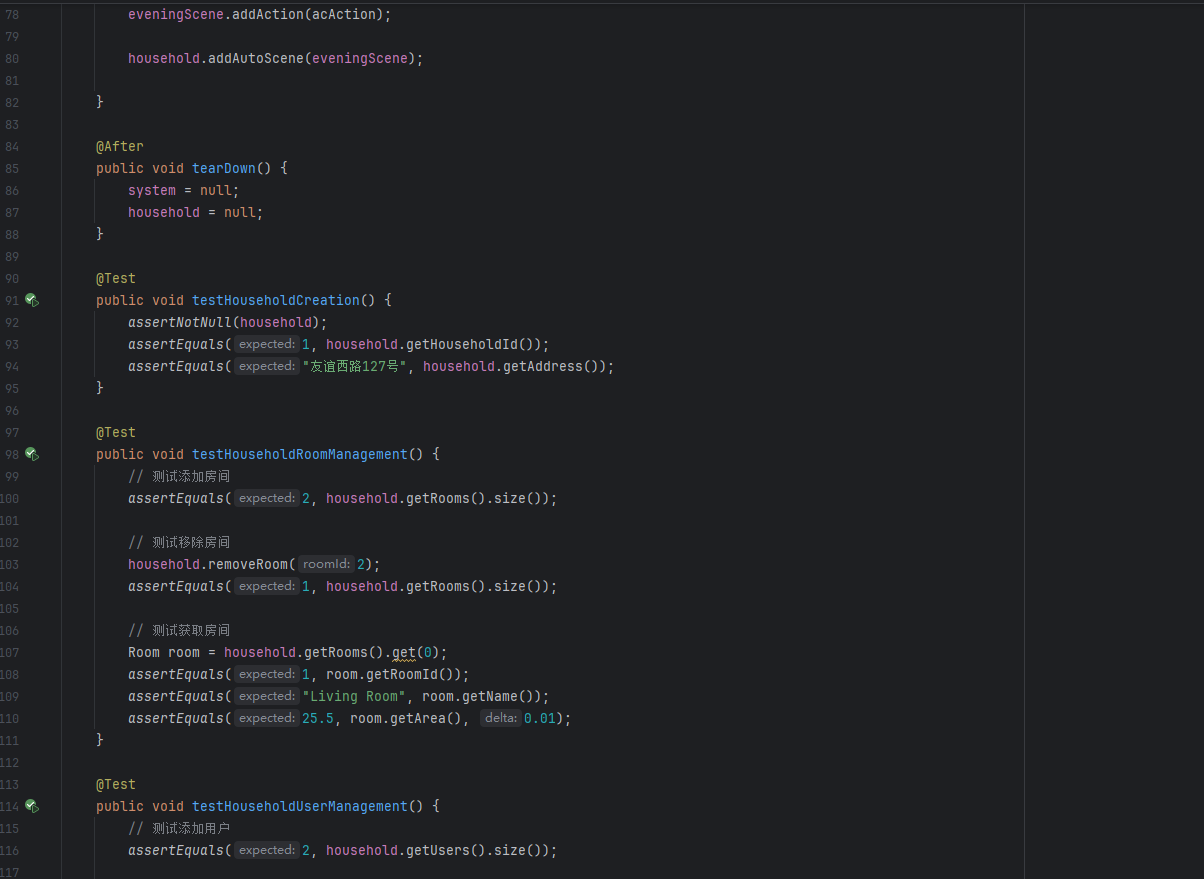
1. testHomeSphereSystem

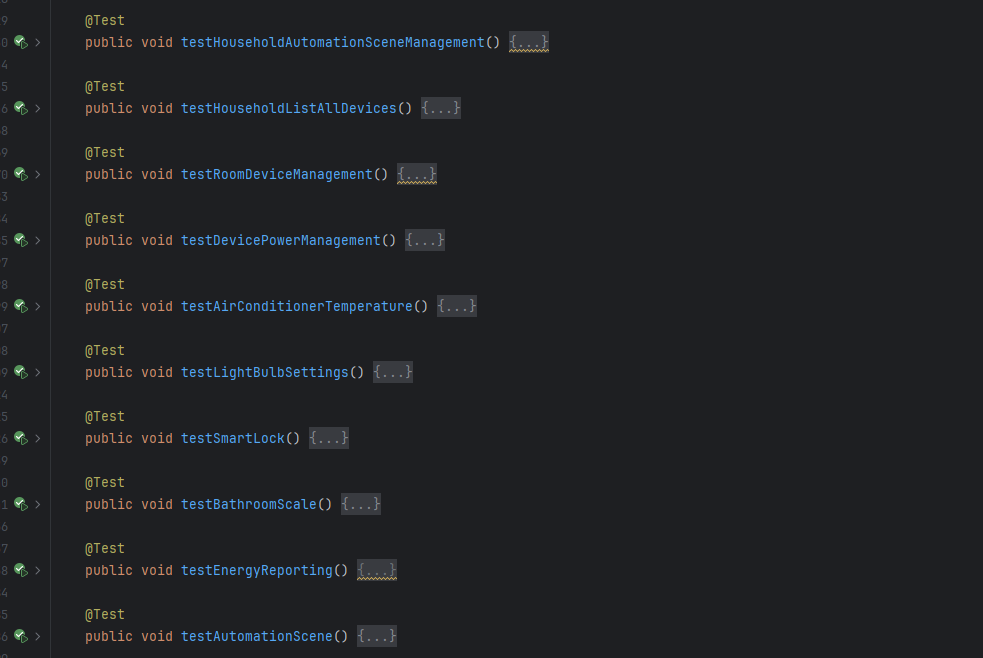
测试逻辑：调用login()、displayUsers()、displayRooms()、displayDevices()、displayAutoScenes()、displayEnergyReportings(s)、manualTrigSceneById()、logoff()方法，验证信息打印符合预期。

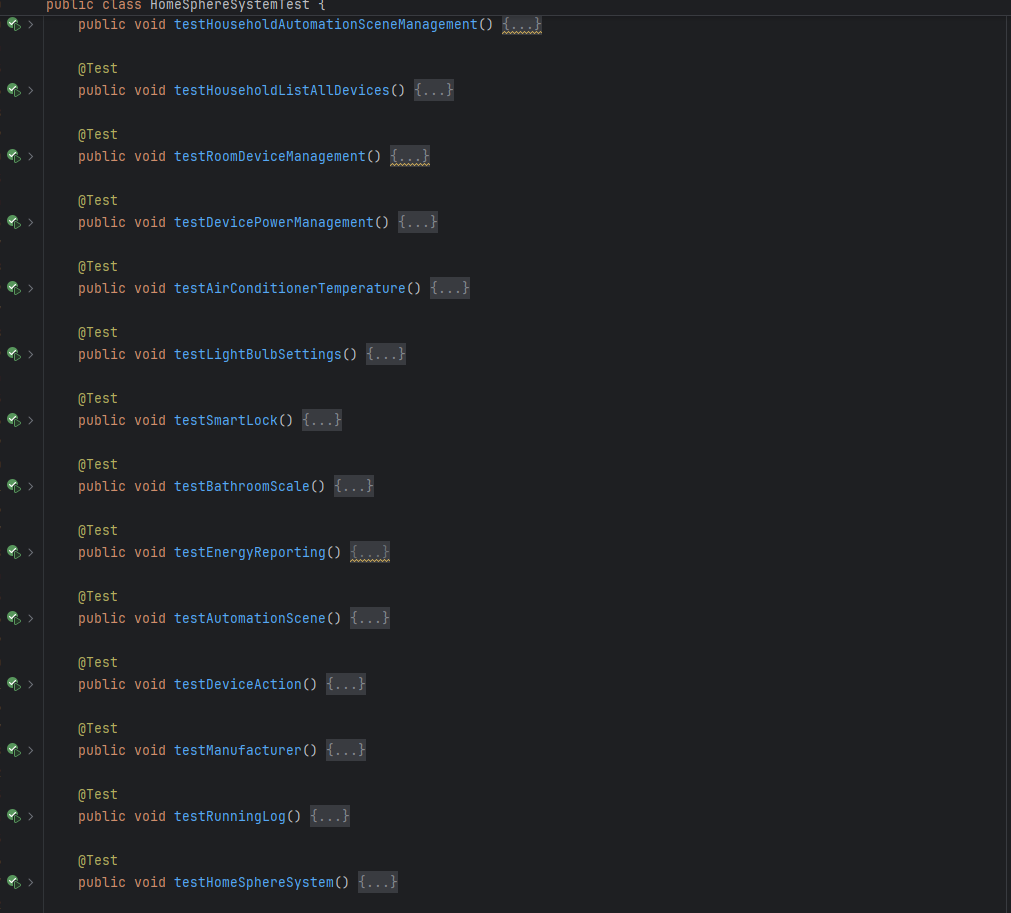
1. 实验结果及分析











4.问题与解决

- 问题1：登录时会出现空指针异常

解决：初始化依赖对象：确保household在使用前已实例化（如在构造函数中初始化）。

public class HomeSphereSystem {

private Household household;

public HomeSphereSystem() {

this.household = new Household(); // 确保非null

}

}

强制用户密码非空：在User类构造函数中校验密码，避免null。

java

运行

public User(..., String password, ...) {

if (password == null || password.trim().isEmpty()) {

throw new IllegalArgumentException("密码不能为空");

}

this.password = password.trim();

}

- 问题2：用户查找逻辑漏洞

解决：统一用户名大小写：存储和查找时均转为小写（或大写）。

Bn// Household类的findUserByLoginName方法

public User findUserByLoginName(String loginName) {

String lowerName = loginName.toLowerCase(); // 统一转为小写

for (User user : users) {

if (user.getLoginName().toLowerCase().equals(lowerName)) {

return user;

}

}

return null;

}

添加用户状态校验：在User类中增加isActive字段，登录时验证。

java

运行

// 登录方法中添加

if (!targetUser.isActive()) {

System.out.println("登录失败：用户已注销");

return;

}。

1. 拓展思考

思考1：抽象类与接口在设计与编程实现时有何异同？

解答：抽象类（Abstract Class）和接口（Interface）是面向对象编程中用于抽象和规范行为的重要机制，二者在设计目的和编程实现上既有相似性，也有显著差异。以下从设计意图、语法规则、使用场景等方面详细对比：

一、相同点

都用于抽象行为两者都不能直接实例化，需通过子类（抽象类的子类）或实现类（接口的实现类）来具体化逻辑，核心目的是定义规范而非具体实现。例如：抽象类 Shape 和接口 Drawable 都可用于规范 “图形” 的行为，但不提供完整实现。

都可包含抽象方法都能声明无具体实现的方法（抽象方法），强制子类 / 实现类必须实现这些方法，从而保证行为的一致性。

二、不同点

|  |  |  |
| --- | --- | --- |
| 维度 | 抽象类（Abstract Class） | 接口（Interface） |
| 关键字 | 使用 abstract class 定义 | 使用 interface 定义 |
| 继承 / 实现方式 | 子类通过 extends 继承，单继承（一个类只能继承一个抽象类） | 实现类通过 implements 实现，多实现（一个类可实现多个接口） |
| 方法实现 | 可以包含抽象方法（无实现）和具体方法（有实现） | JDK 8 前：只能包含抽象方法（无实现）；  JDK 8 及后：可包含 default 方法（有实现）和 static 方法（有实现） |
| 成员变量 | 可以包含各种修饰符的成员变量（public/private/protected、static/final 等），变量可被修改 | 成员变量默认是 public static final（常量），必须初始化且不可修改 |
| 构造方法 | 有构造方法（用于子类初始化时调用） | 无构造方法（接口不能实例化，无需构造） |
| 设计意图 | 体现 \*\*“is-a” 关系 \*\*（继承的层级关系），用于抽取共性实现（代码复用） | 体现 \*\*“has-a” 关系 \*\*（能力的附加），用于定义行为规范（多实现扩展） |

思考2：本实验中是否涉及多态，在设计和实现中时如何体现的？

解答：实验中涉及多态。

一、多态的核心体现场景

1. 设备控制逻辑中的多态（接口实现 + 方法重写）

在设备控制（如 execute 方法）中，不同设备（AirConditioner、LightBulb、SmartLock）都属于 Device 类型，但对同一指令（如 powerOn、setTemperature）的处理不同，这是多态的典型体现。

设计层面：定义抽象父类 Device 或接口（如 Controllable），声明通用方法（powerOn()、powerOff() 等），具体设备类（如 AirConditioner）继承 Device 并重写这些方法，实现自身特有的逻辑。

实现层面：

java

运行

// 抽象父类/接口（定义规范）

public abstract class Device {

public abstract void powerOn();

public abstract void powerOff();

}

// 具体设备类（重写方法，实现多态）

public class AirConditioner extends Device {

@Override

public void powerOn() {

System.out.println("空调开机，开始预热"); // 空调特有的开机逻辑

}

}

public class LightBulb extends Device {

@Override

public void powerOn() {

System.out.println("灯泡开灯，亮度设为默认值"); // 灯泡特有的开机逻辑

}

}

多态的使用：在 execute 方法中，通过 Device 父类引用指向不同子类对象，调用 powerOn() 时，会自动执行对应设备的实现：

java

运行

public void execute(Device device) {

device.powerOn(); // 同一方法调用，根据设备类型执行不同逻辑

}

// 调用时体现多态

execute(new AirConditioner()); // 输出“空调开机，开始预热”

execute(new LightBulb()); // 输出“灯泡开灯，亮度设为默认值”

2. 能量上报功能中的多态（接口实现）

若存在 “设备能量上报” 功能（如你之前提到的 EnergyReporting 接口），不同设备的功率计算方式不同，也会体现多态。

设计层面：定义接口 EnergyReporting 声明 getPower() 方法，不同设备类实现该接口并返回自身功率。

实现层面：

java

运行

// 接口定义规范

public interface EnergyReporting {

double getPower(); // 声明功率获取方法

}

// 空调实现接口（多态实现1）

public class AirConditioner extends Device implements EnergyReporting {

@Override

public double getPower() {

return 1500.0; // 空调功率

}

}

// 灯泡实现接口（多态实现2）

public class LightBulb extends Device implements EnergyReporting {

@Override

public double getPower() {

return 60.0; // 灯泡功率

}

}

多态的使用：通过接口引用集合管理不同设备，统一调用 getPower() 即可获取各自功率：

java

运行

List<EnergyReporting> devices = new ArrayList<>();

devices.add(new AirConditioner());

devices.add(new LightBulb());

for (EnergyReporting device : devices) {

System.out.println("功率：" + device.getPower()); // 自动调用对应设备的实现

}

3. 用户操作中的多态（继承 + 角色差异）

若系统存在不同用户角色（如 Admin、RegularUser 继承自 User），同一操作（如 manageDevice()）的权限不同，也会体现多态。

设计层面：父类 User 声明 manageDevice() 方法，子类 Admin 拥有全部权限，RegularUser 权限受限。

实现层面：

java

运行

public abstract class User {

public abstract void manageDevice(Device device);

}

public class Admin extends User {

@Override

public void manageDevice(Device device) {

System.out.println("管理员：完全控制设备"); // 管理员权限

}

}

public class RegularUser extends User {

@Override

public void manageDevice(Device device) {

System.out.println("普通用户：仅能开关设备"); // 普通用户权限

}

}

多态的使用：通过 User 父类引用调用 manageDevice()，根据用户类型执行不同权限逻辑：

java

运行

User user1 = new Admin();

User user2 = new RegularUser();

user1.manageDevice(ac); // 管理员逻辑

user2.manageDevice(ac); // 普通用户逻辑

二、多态在设计中的意义

简化代码逻辑：如 execute 方法无需针对每种设备写单独的控制逻辑，只需通过 Device 父类调用方法，降低代码冗余。

提高扩展性：新增设备（如 SmartCurtain）时，只需让其继承 Device 并重写方法，无需修改现有控制逻辑（符合 “开闭原则”）。

统一接口规范：无论设备类型如何，都通过统一的父类 / 接口方法交互，便于团队协作和代码维护。