HomeNest

An Intelligent Home Automation System

Presented by :
Arij Khlifi | Rouida Hentati | Takwa Dalensi | Lina Smiri | Siwar Jerbi

Project Overview

SmartNest is a comprehensive *object-oriented Java* application for designing, controlling, and managing smart home environments

- **Design:** Create rooms and add smart devices

- **Control:** Operate devices in real-time

- **Manage :** Monitor energy and schedule tasks

Automate : Implement intelligent behaviors

SYSTEM ARCHITECTURE

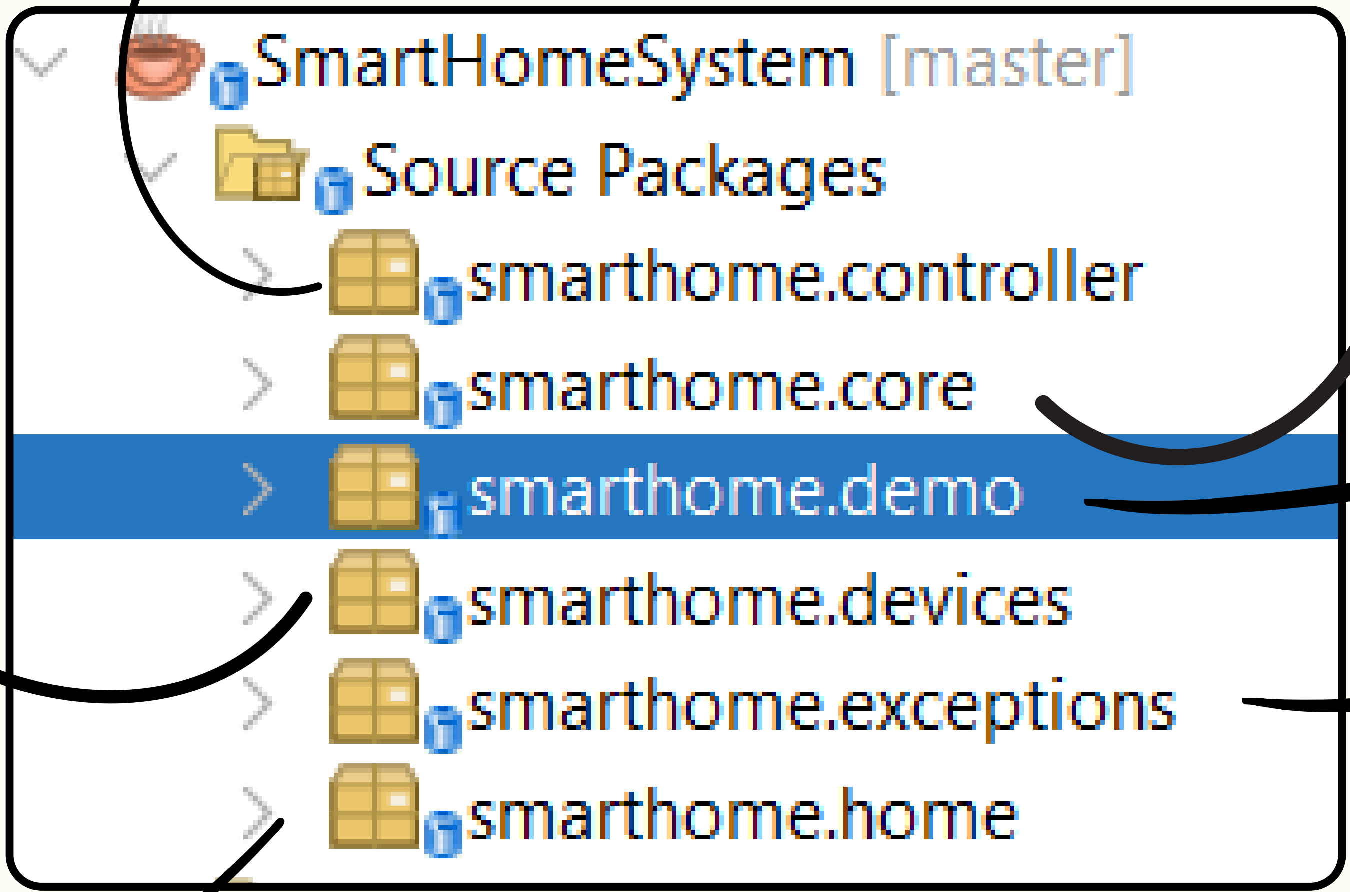
The Brain

- CentralController logic
- ScheduledTask automation
- Coordinates everything

The Foundation

- SmartDevice abstract class
- Controllable & EnergyConsumer interfaces

Devices that extends our SmartDevice.



SmartHomeSystem [master]
Source Packages
smarthome.controller
smarthome.core
smarthome.demo
smarthome.devices
smarthome.exceptions
smarthome.home

The Interface -
User interaction layer

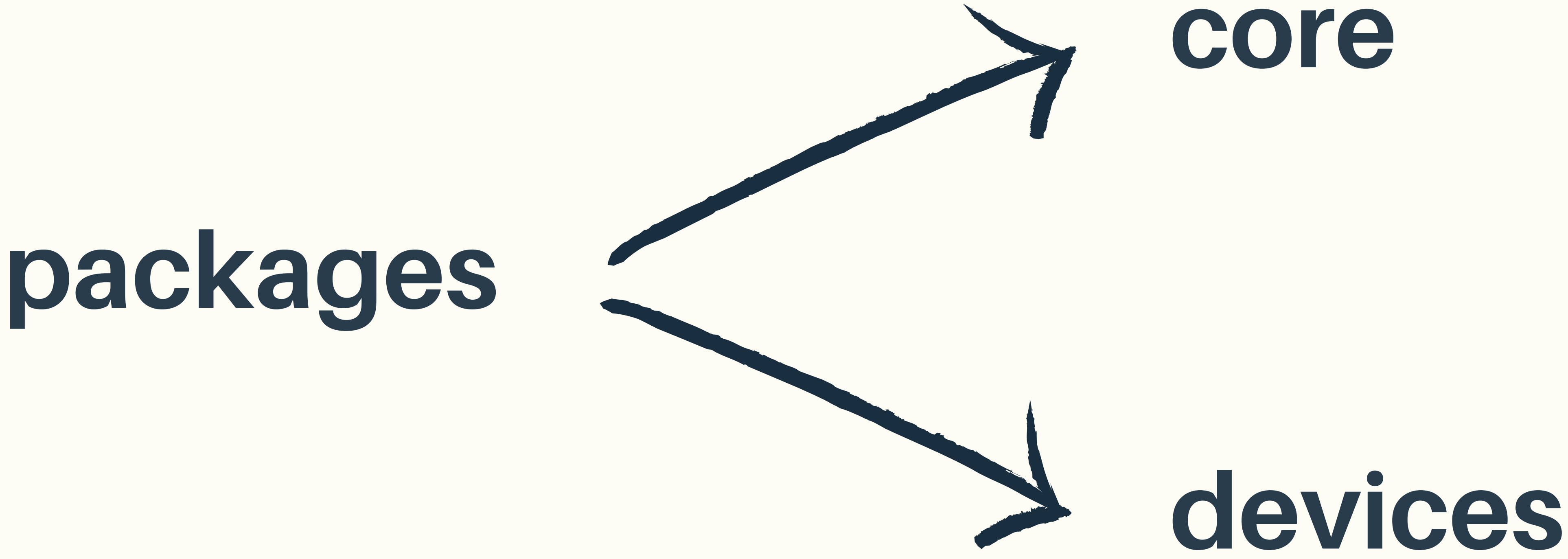
The Structure

- Home & Room classes
- Organizes devices physically
- Room capacity management

The Safety Net

make failures graceful and informative.

core device architecture



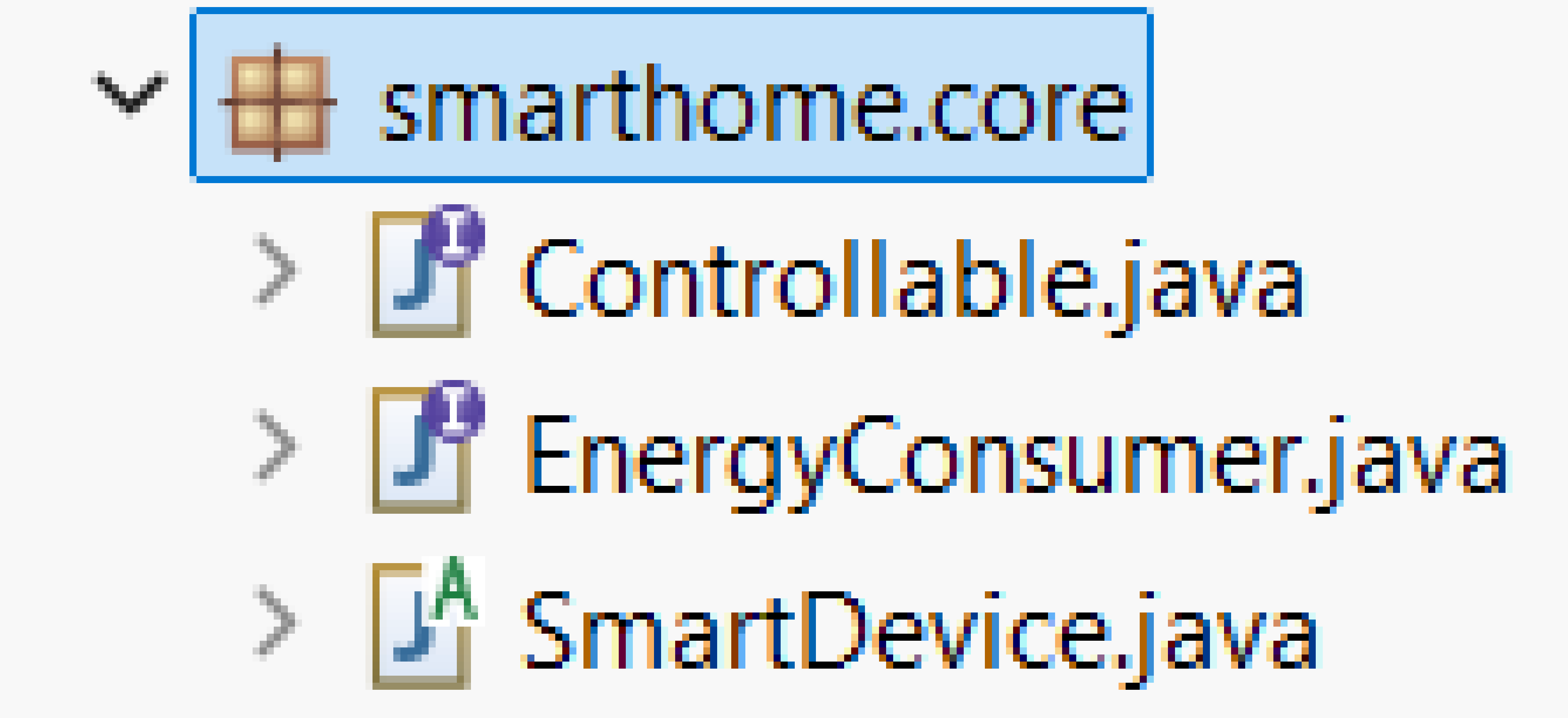
packages

core

devices

core package :

→ Contains generic abstractions that define what a smart device is and what it can do, without any device-specific logic.



smarthome.core

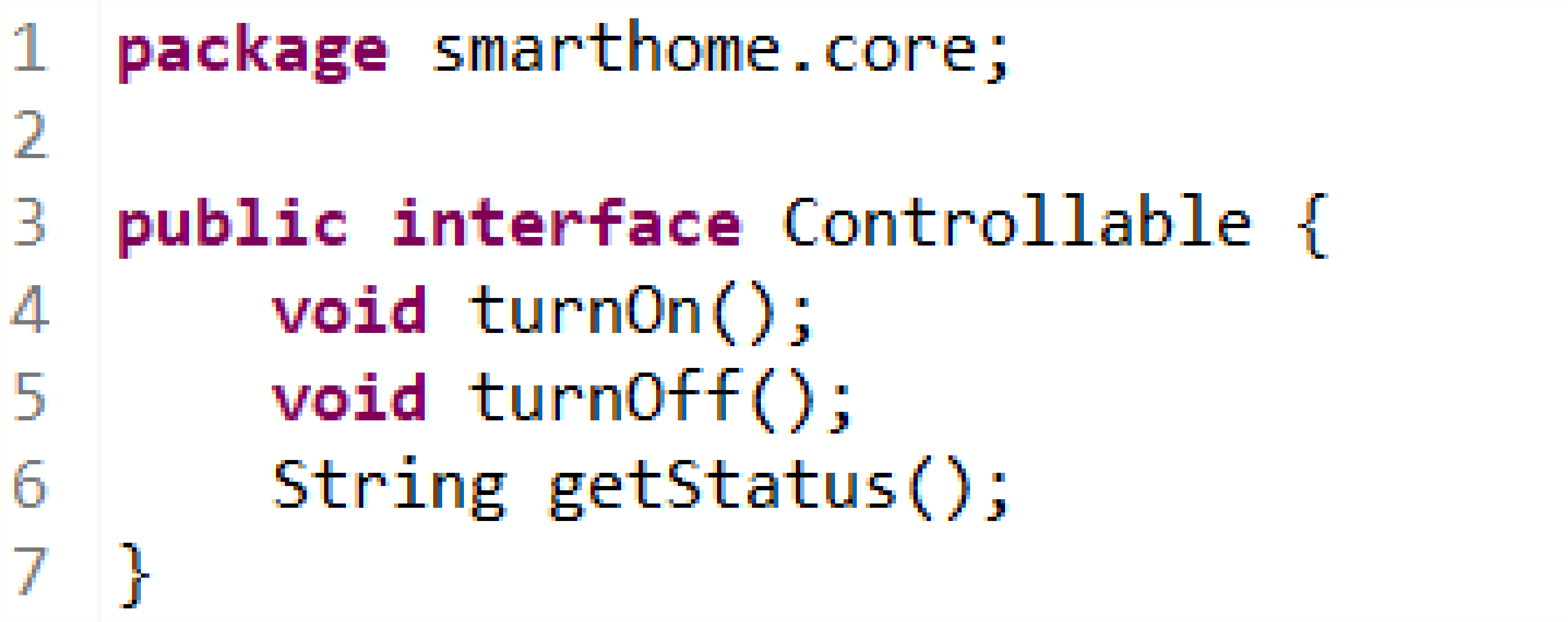
Controllable.java

EnergyConsumer.java

SmartDevice.java

Controllable Interface

→ defines basic control actions

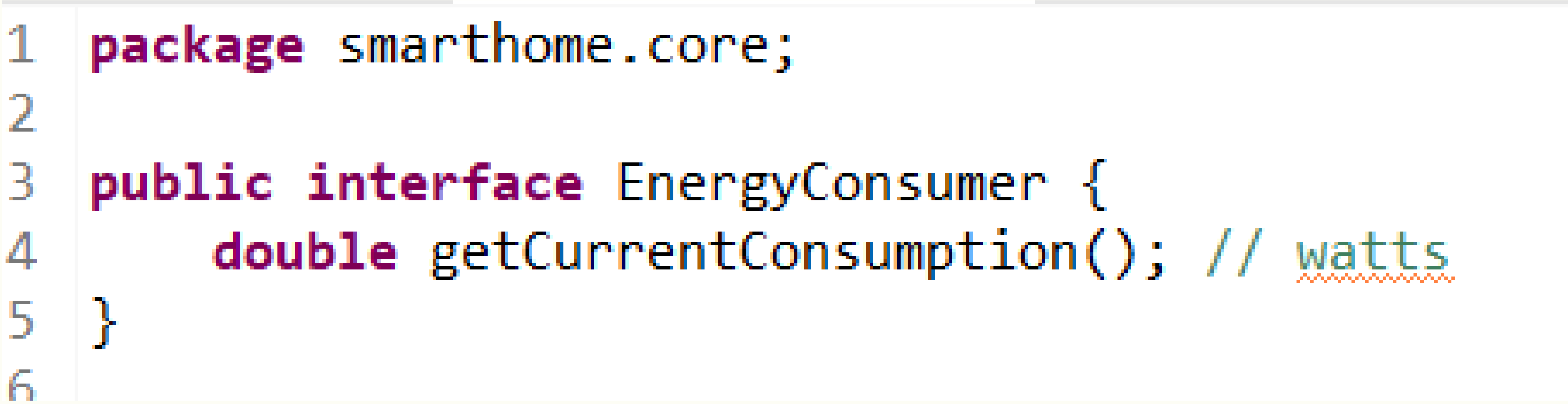


```java
package smarthome.core;

public interface Controllable {
    void turnOn();
    void turnOff();
    String getStatus();
}
```

Energy Consumer

→ Models power usage

→ It allows the system to calculate energy consumption independently from the device type.

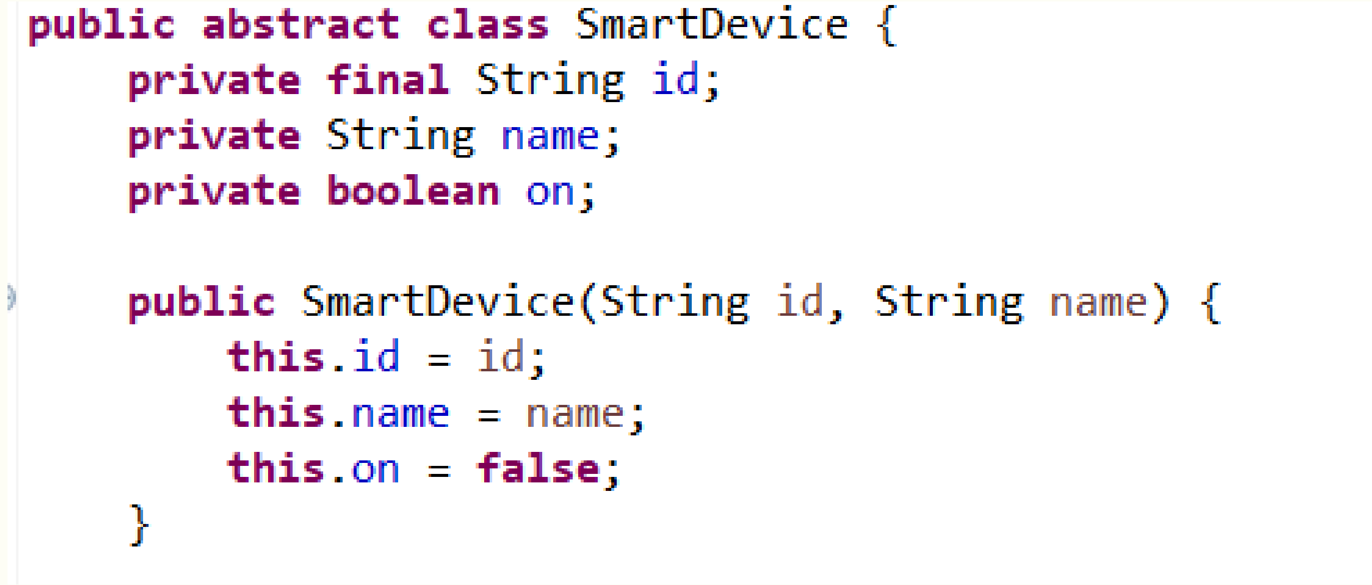


```java
package smarthome.core;

public interface EnergyConsumer {
    double getCurrentConsumption(); // watts
}
```

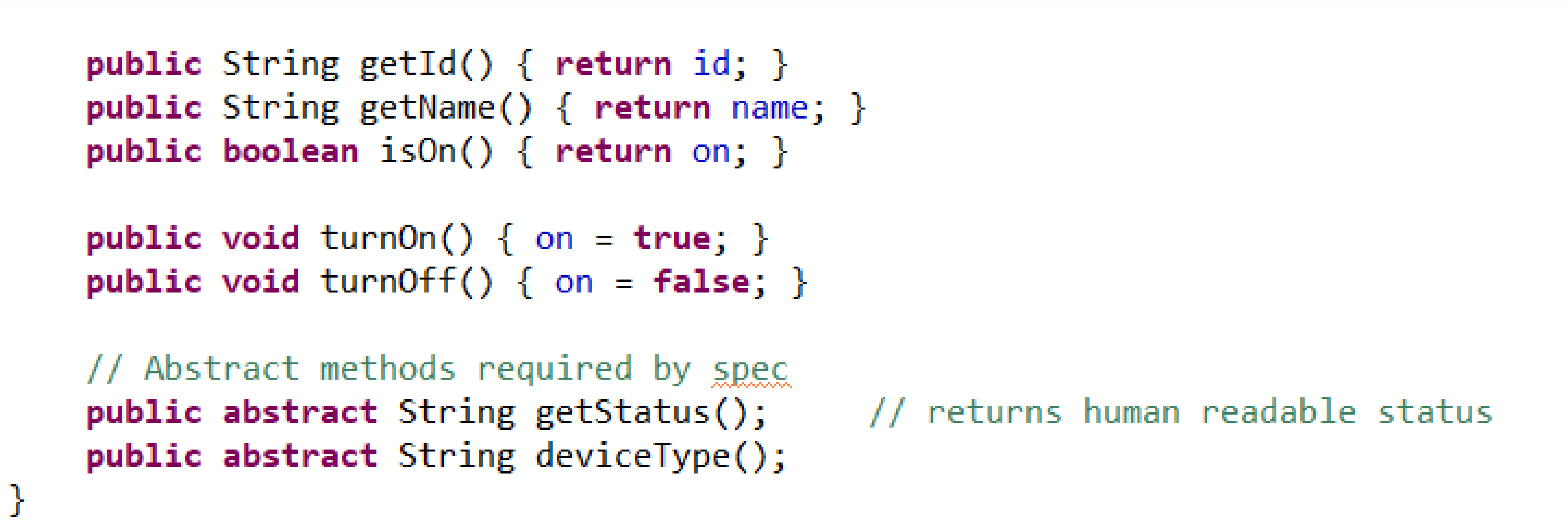
Smart Device

→ abstract base class for all devices

→ It centralizes shared state like device ID and power status, while forcing subclasses to define their own status and type



```
public abstract class SmartDevice {
    private final String id;
    private String name;
    private boolean on;

    public SmartDevice(String id, String name) {
        this.id = id;
        this.name = name;
        this.on = false;
    }
```



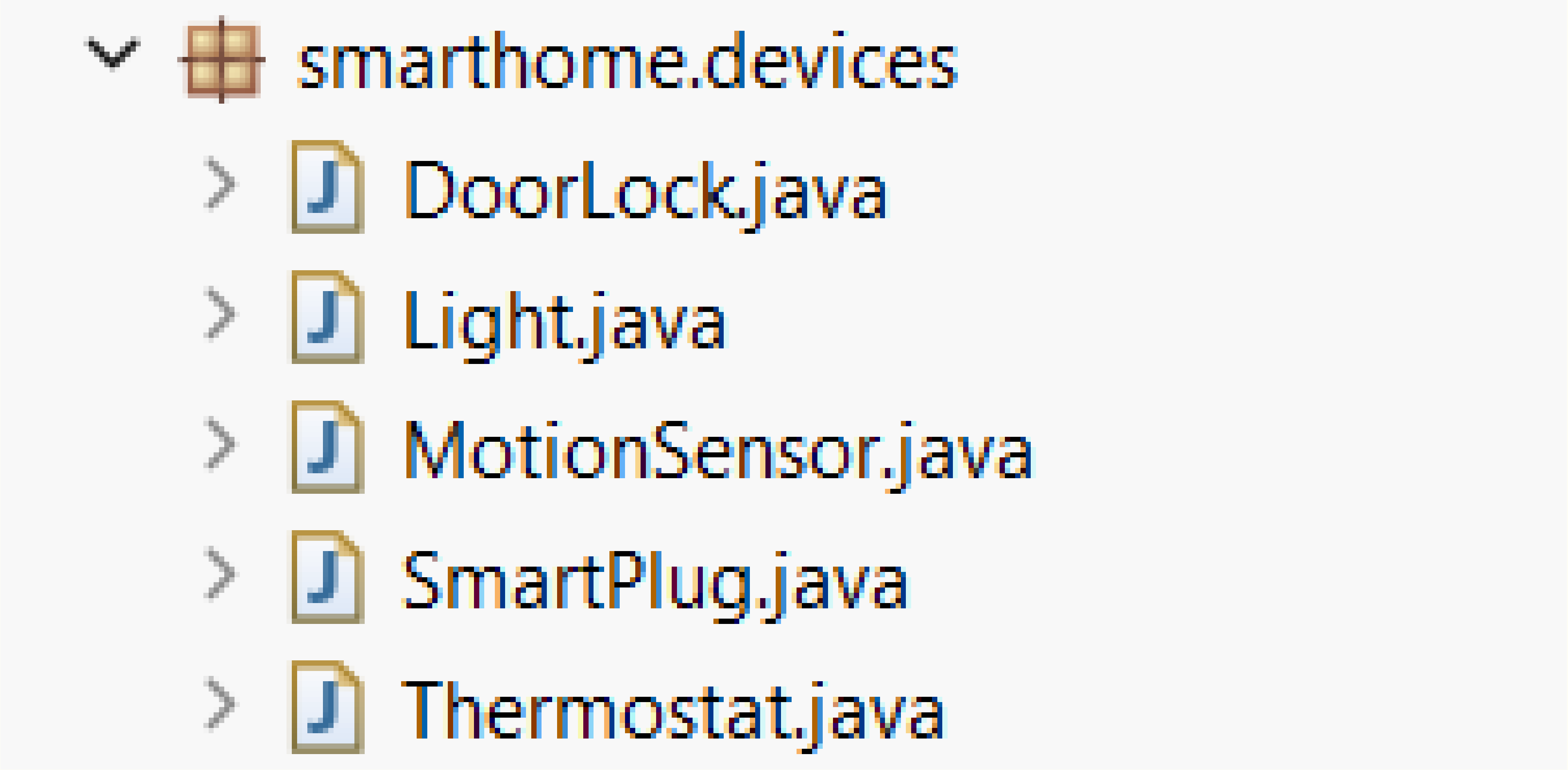
```
    public String getId() { return id; }
    public String getName() { return name; }
    public boolean isOn() { return on; }

    public void turnOn() { on = true; }
    public void turnOff() { on = false; }

    // Abstract methods required by spec
    public abstract String getStatus();     // returns human readable status
    public abstract String deviceType();
}
```

Why Abstract Class + Interfaces?

Abstract class → shared state

Interfaces → shared capabilities

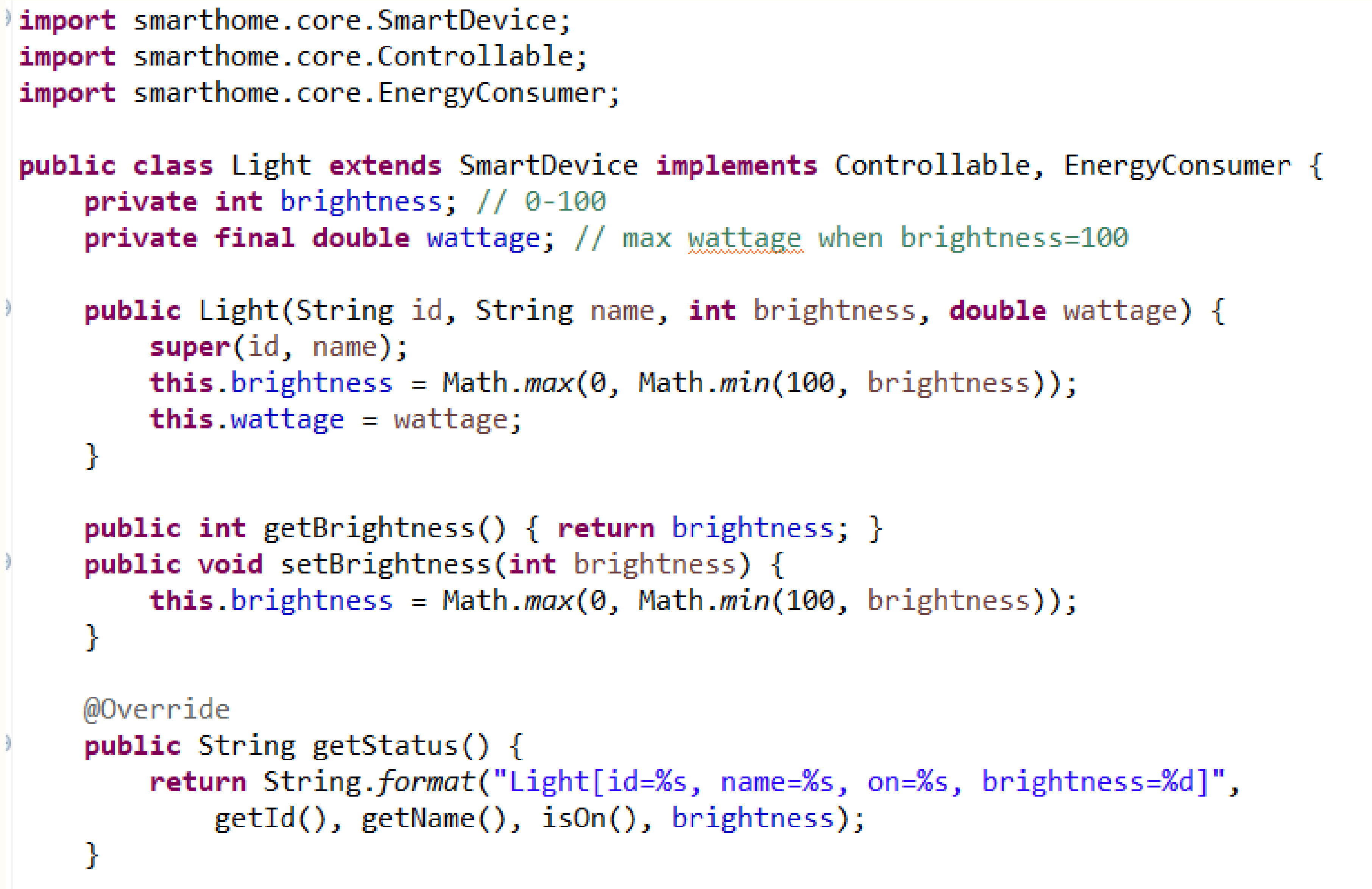
devices package:



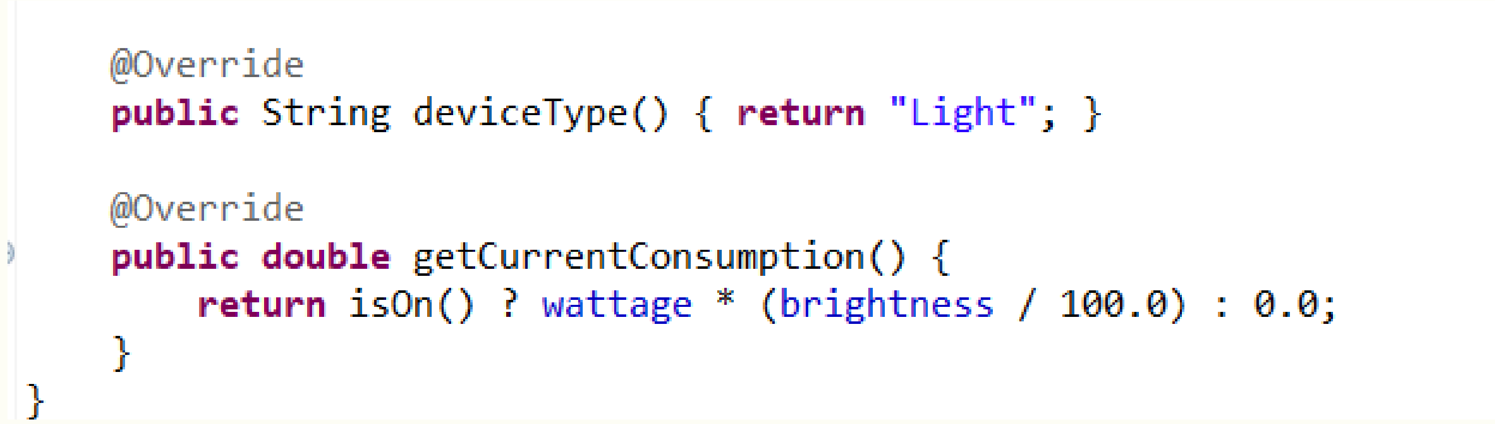
smarthome.devices
- DoorLock.java
- Light.java
- MotionSensor.java
- SmartPlug.java
- Thermostat.java

The Light device

→ both controllable and an energy consumer



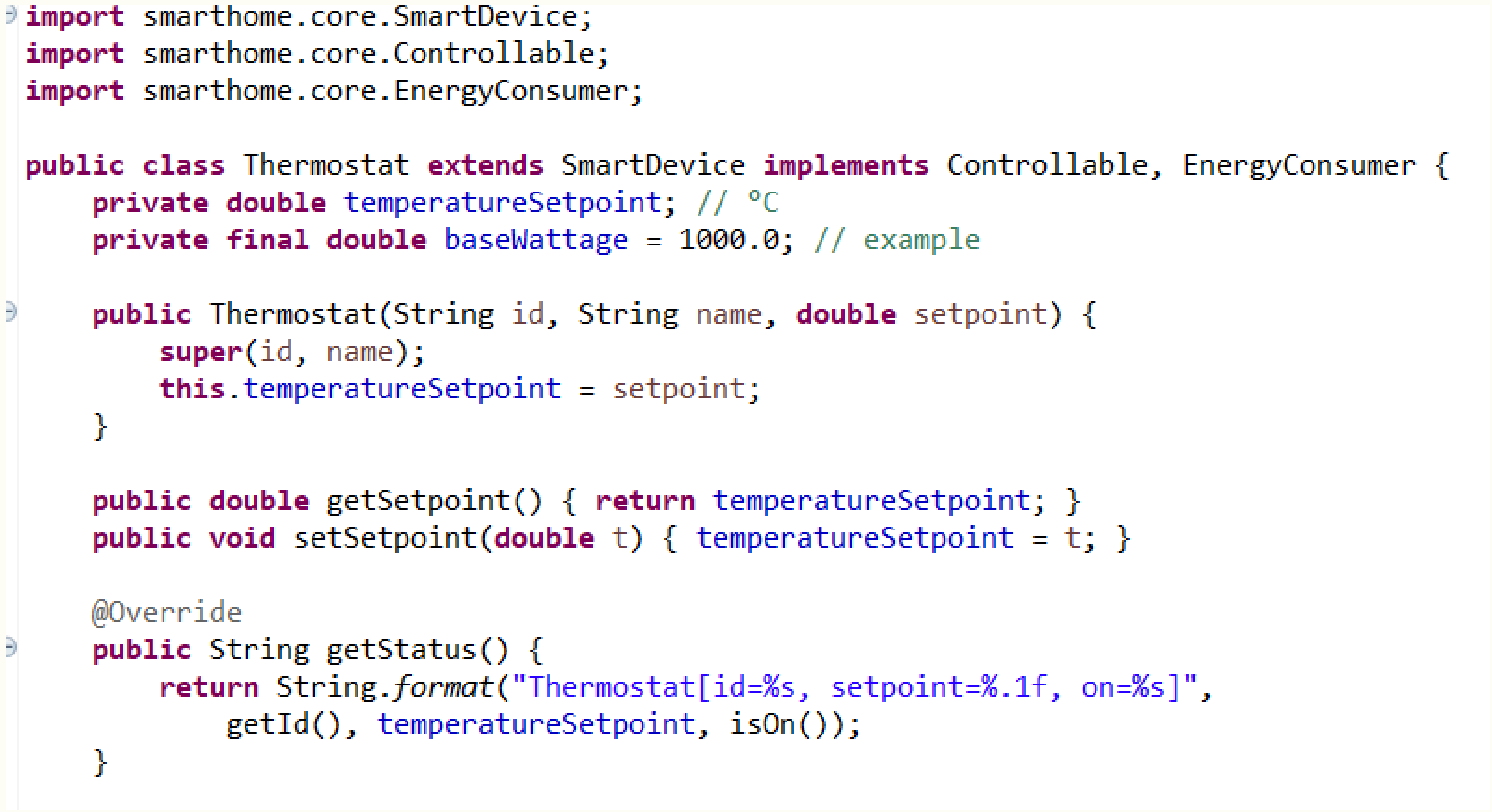
```java
import smarthome.core.SmartDevice;
import smarthome.core.Controllable;
import smarthome.core.EnergyConsumer;

public class Light extends SmartDevice implements Controllable, EnergyConsumer {
    private int brightness; // 0-100
    private final double wattage; // max wattage when brightness=100

    public Light(String id, String name, int brightness, double wattage) {
        super(id, name);
        this.brightness = Math.max(0, Math.min(100, brightness));
        this.wattage = wattage;
    }

    public int getBrightness() { return brightness; }
    public void setBrightness(int brightness) {
        this.brightness = Math.max(0, Math.min(100, brightness));
    }

    @Override
    public String getStatus() {
        return String.format("Light[id=%s, name=%s, on=%s, brightness=%d]",
            getId(), getName(), isOn(), brightness);
    }
```



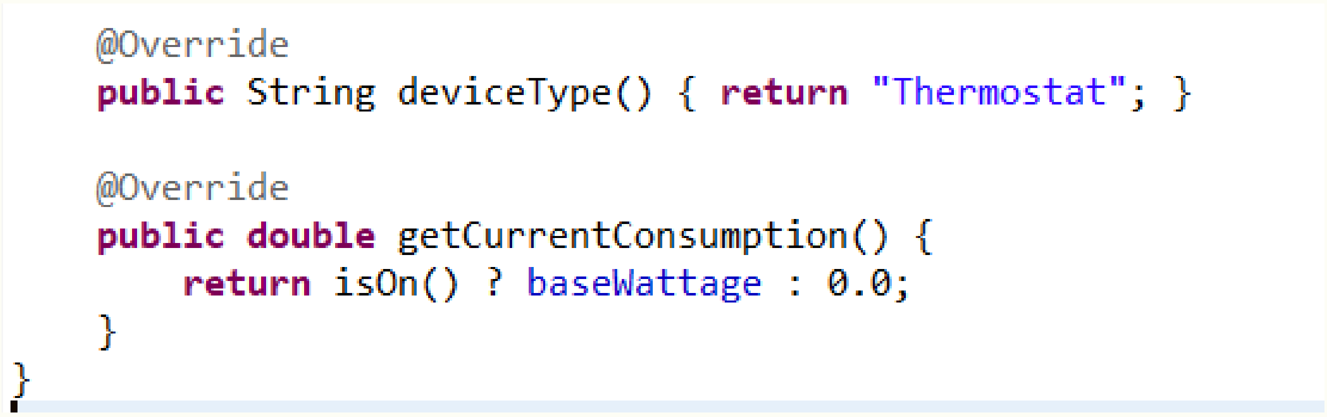
```java
    @Override
    public String deviceType() { return "Light"; }

    @Override
    public double getCurrentConsumption() {
        return isOn() ? wattage * (brightness / 100.0) : 0.0;
    }
}
```

The Thermostat

→ includes a temperature setpoint and consumes energy only when active, representing heating or cooling.



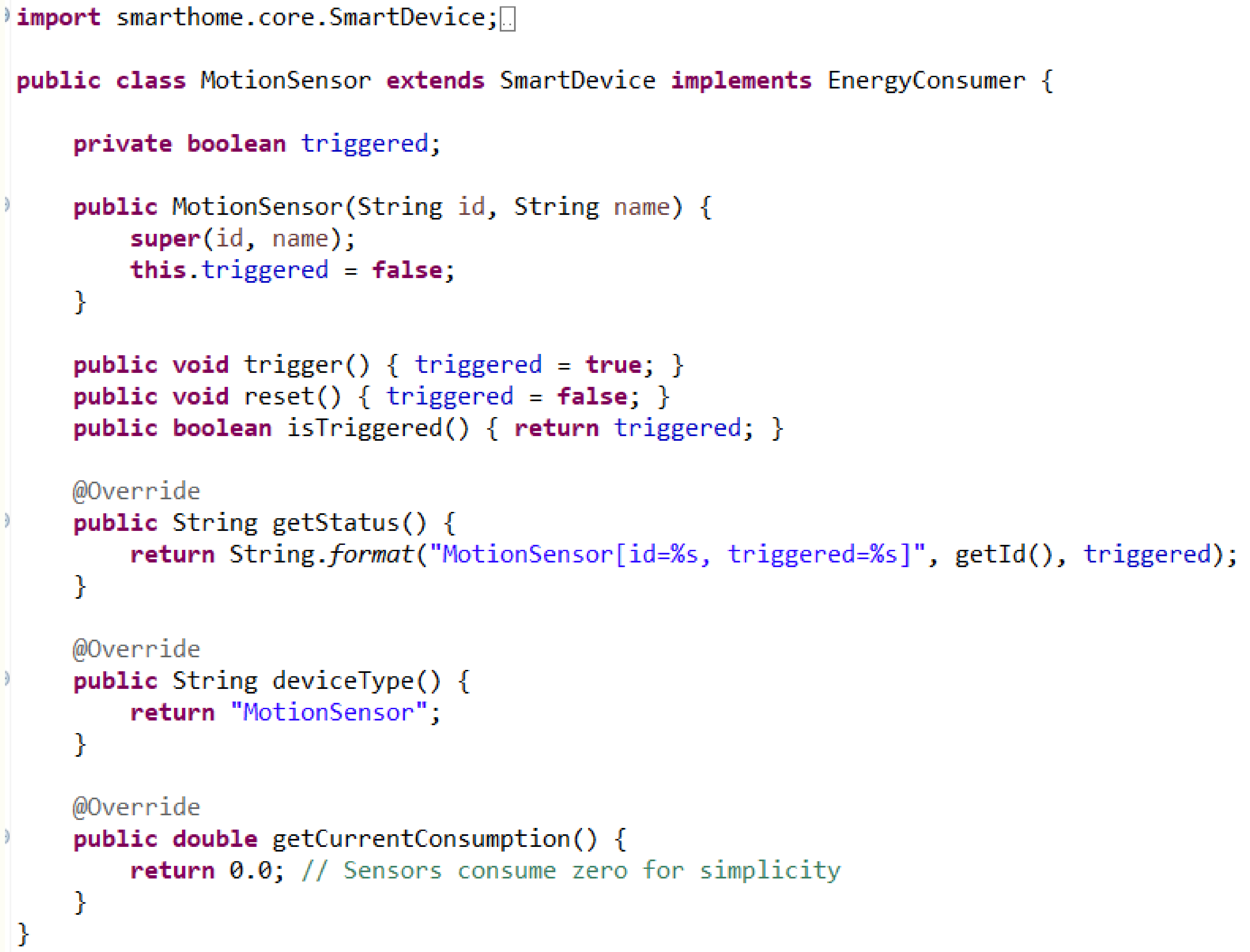
```java
import smarthome.core.SmartDevice;
import smarthome.core.Controllable;
import smarthome.core.EnergyConsumer;

public class Thermostat extends SmartDevice implements Controllable, EnergyConsumer {
    private double temperatureSetpoint; // °C
    private final double baseWattage = 1000.0; // example

    public Thermostat(String id, String name, double setpoint) {
        super(id, name);
        this.temperatureSetpoint = setpoint;
    }

    public double getSetpoint() { return temperatureSetpoint; }
    public void setSetpoint(double t) { temperatureSetpoint = t; }

    @Override
    public String getStatus() {
        return String.format("Thermostat[id=%s, setpoint=%.1f, on=%s]",
            getId(), temperatureSetpoint, isOn());
    }
```



```java
    @Override
    public String deviceType() { return "Thermostat"; }

    @Override
    public double getCurrentConsumption() {
        return isOn() ? baseWattage : 0.0;
    }
}
```

The MotionSensor

→ not controllable



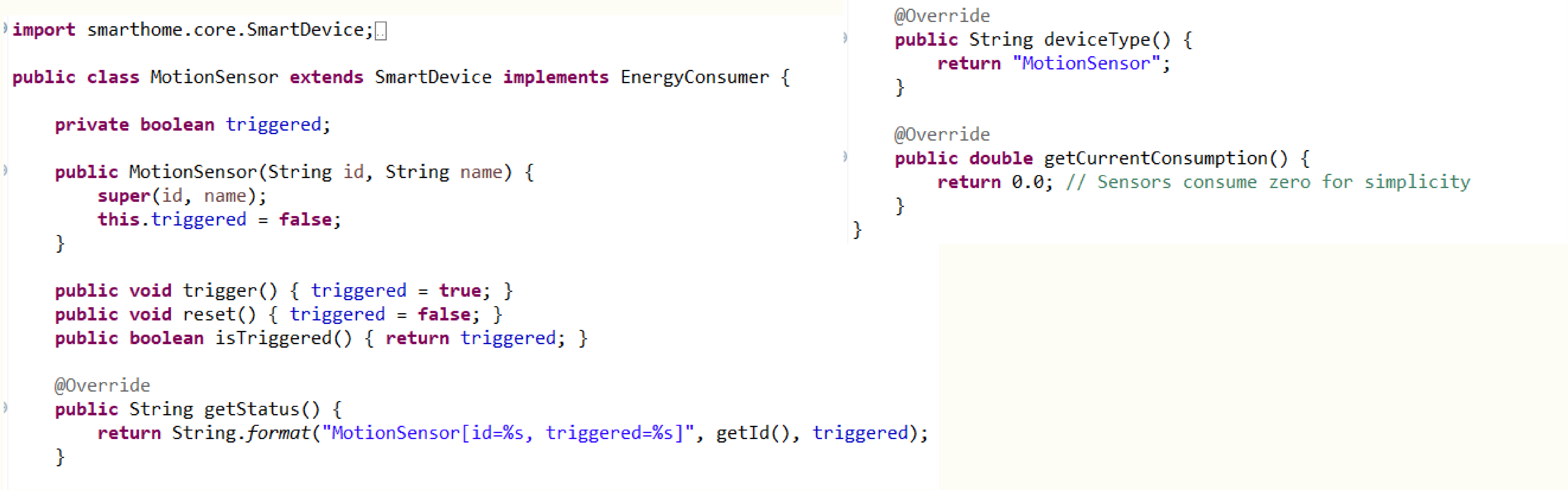
```java
import smarthome.core.SmartDevice;

public class MotionSensor extends SmartDevice implements EnergyConsumer {

    private boolean triggered;

    public MotionSensor(String id, String name) {
        super(id, name);
        this.triggered = false;
    }

    public void trigger() { triggered = true; }
    public void reset() { triggered = false; }
    public boolean isTriggered() { return triggered; }

    @Override
    public String getStatus() {
        return String.format("MotionSensor[id=%s, triggered=%s]", getId(), triggered);
    }

    @Override
    public String deviceType() {
        return "MotionSensor";
    }

    @Override
    public double getCurrentConsumption() {
        return 0.0; // Sensors consume zero for simplicity
    }
}
```

The SmartPlug

→ both Controllable and EnergyConsumer.

It represents a generic smart outlet that consumes energy only when turned on

→ Controls external appliances

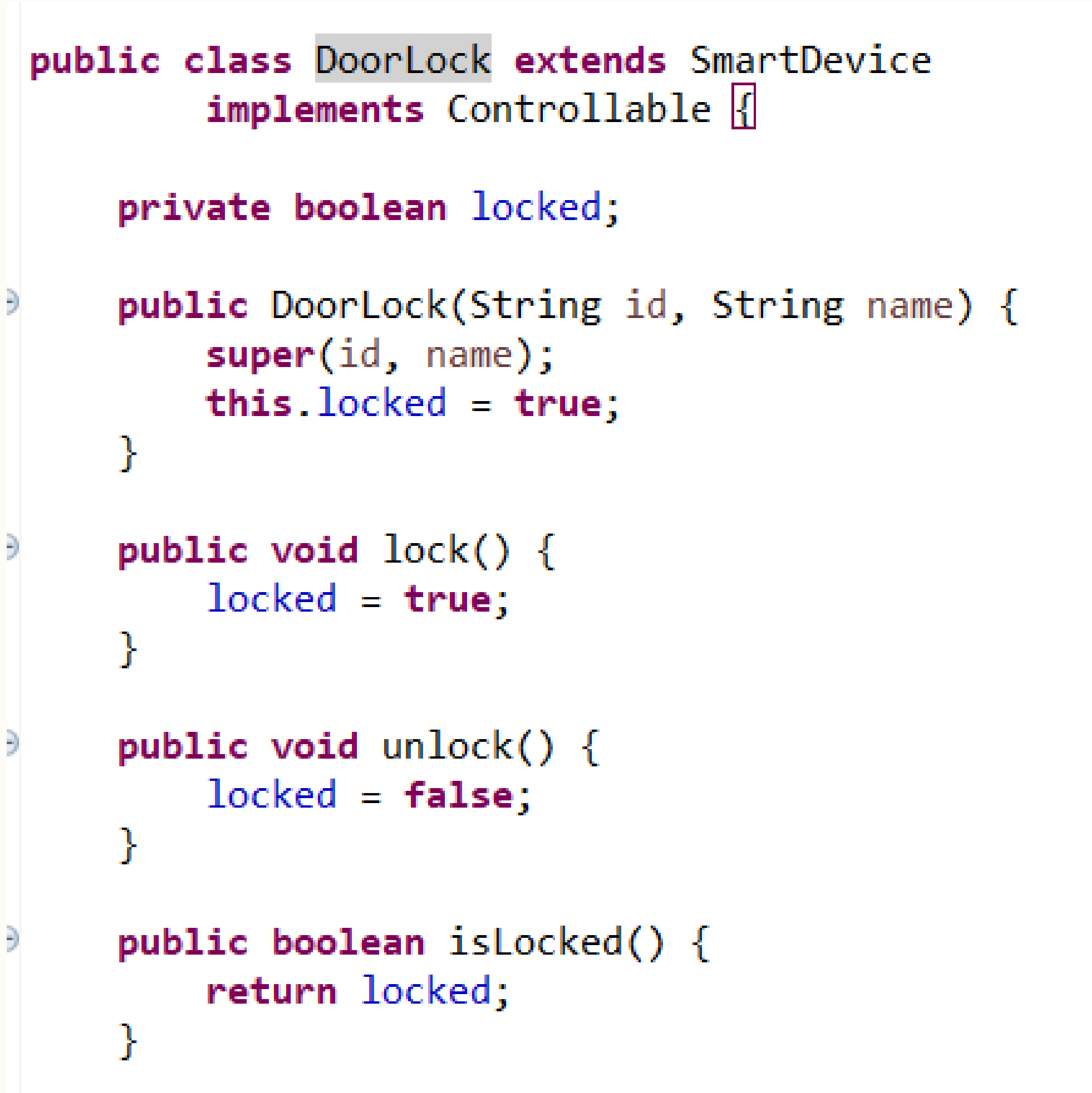


```java
import smarthome.core.SmartDevice;

public class MotionSensor extends SmartDevice implements EnergyConsumer {

    private boolean triggered;

    public MotionSensor(String id, String name) {
        super(id, name);
        this.triggered = false;
    }

    public void trigger() { triggered = true; }
    public void reset() { triggered = false; }
    public boolean isTriggered() { return triggered; }

    @Override
    public String getStatus() {
        return String.format("MotionSensor[id=%s, triggered=%s]", getId(), triggered);
    }
```

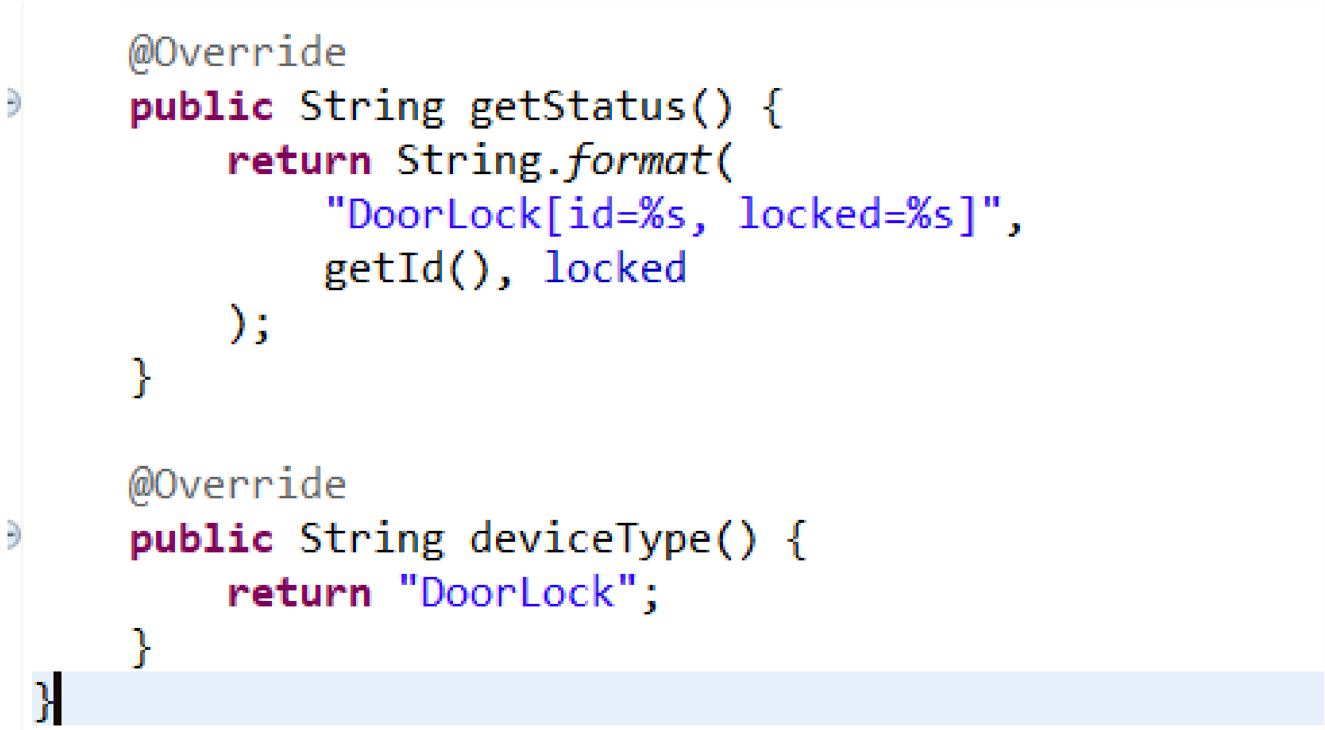
```java
    @Override
    public String deviceType() {
        return "MotionSensor";
    }

    @Override
    public double getCurrentConsumption() {
        return 0.0; // Sensors consume zero for simplicity
    }
}
```

The DoorLock

→ The DoorLock is controllable but does not implement EnergyConsumer.



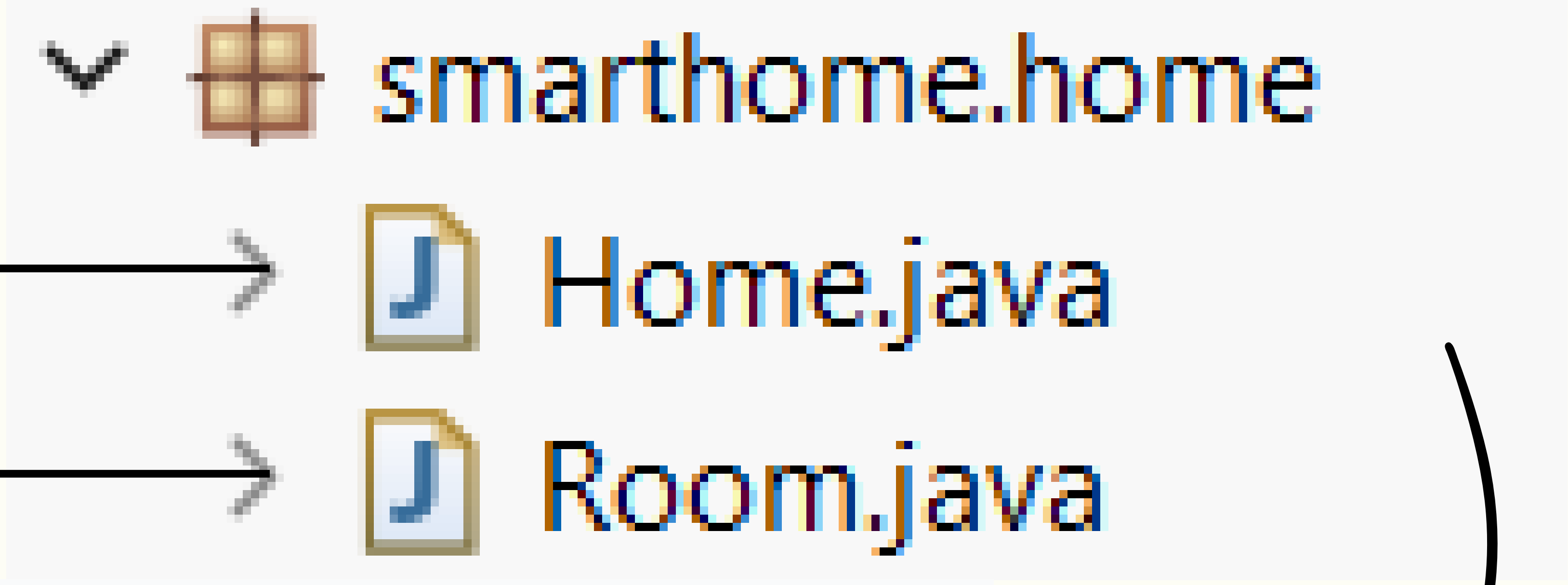
```java
public class DoorLock extends SmartDevice
        implements Controllable {

    private boolean locked;

    public DoorLock(String id, String name) {
        super(id, name);
        this.locked = true;
    }

    public void lock() {
        locked = true;
    }

    public void unlock() {
        locked = false;
    }

    public boolean isLocked() {
        return locked;
    }
```

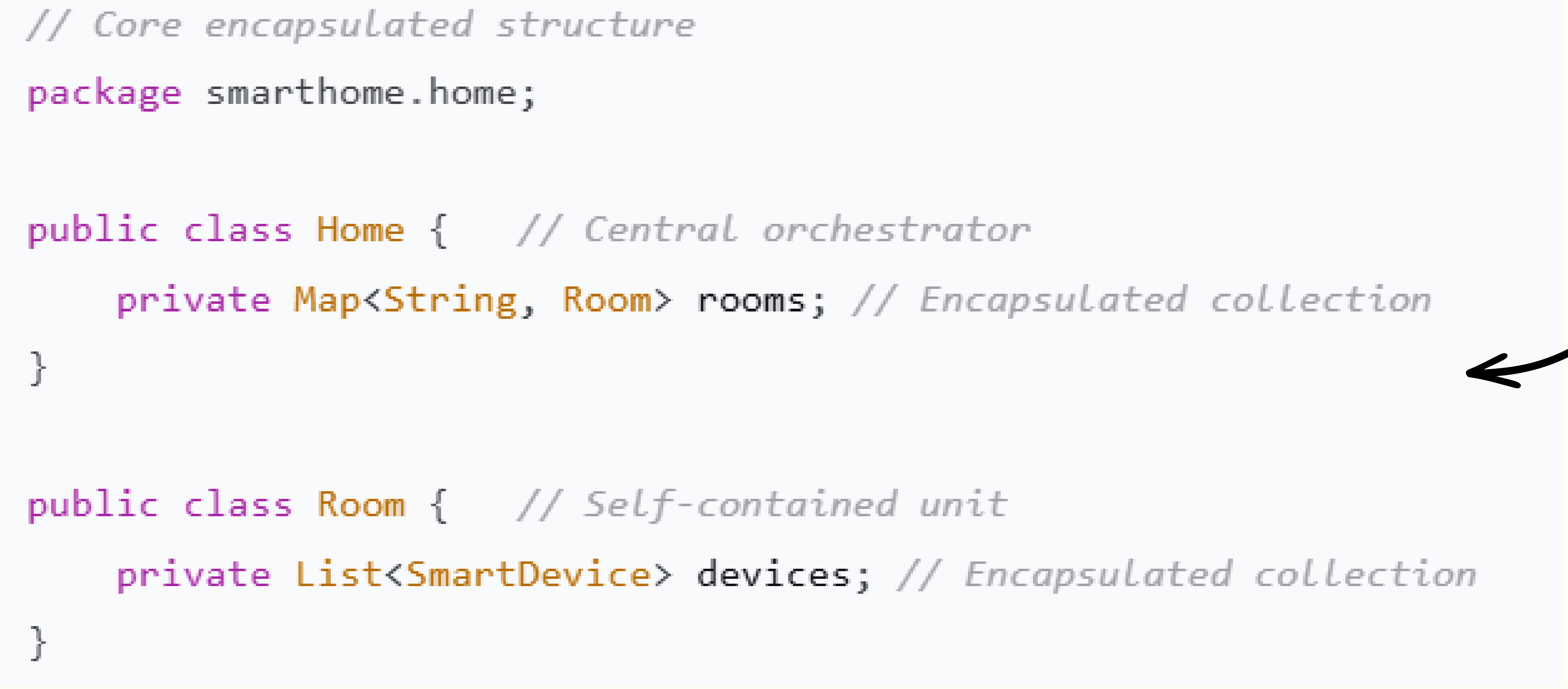


```java
    @Override
    public String getStatus() {
        return String.format(
            "DoorLock[id=%s, locked=%s]",
            getId(), locked
        );
    }

    @Override
    public String deviceType() {
        return "DoorLock";
    }
}
```

Smart Home Foundation

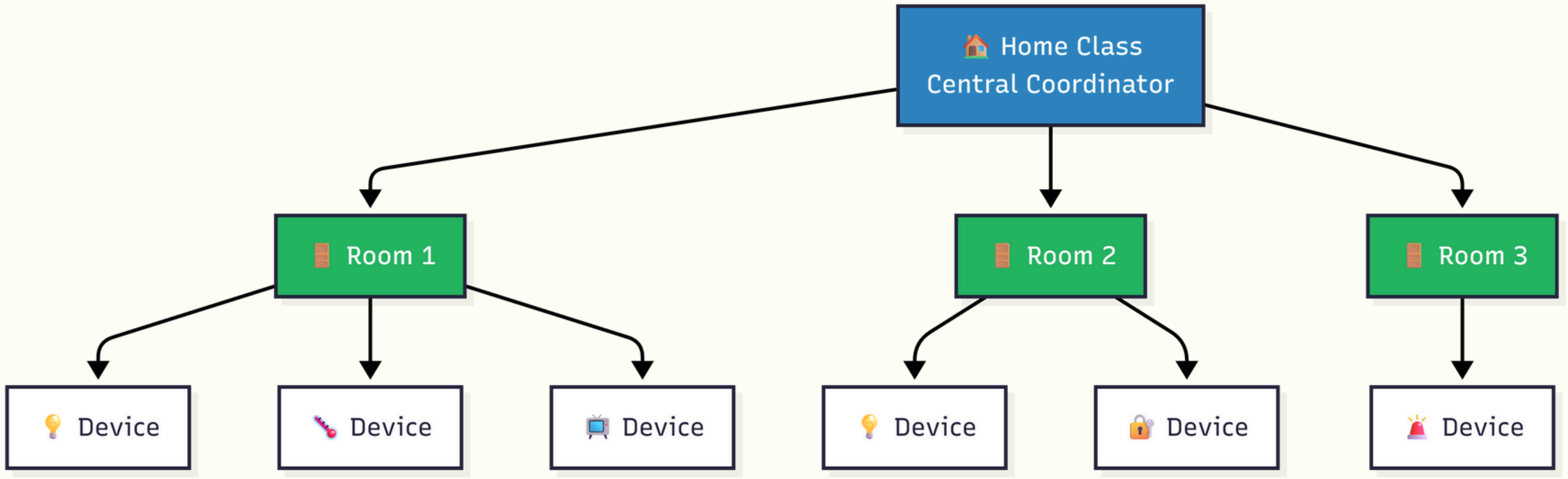
- Manages all Rooms
- Manages devices per room





```
// Core encapsulated structure
package smarthome.home;

public class Home {   // Central orchestrator
    private Map<String, Room> rooms; // Encapsulated collection
}

public class Room {   // Self-contained unit
    private List<SmartDevice> devices; // Encapsulated collection
}
```

Architectural Overview



🏠 Home Class
Central Coordinator

🚪 Room 1

🚪 Room 2

🚪 Room 3

💡 Device

🌡️ Device

📺 Device

💡 Device

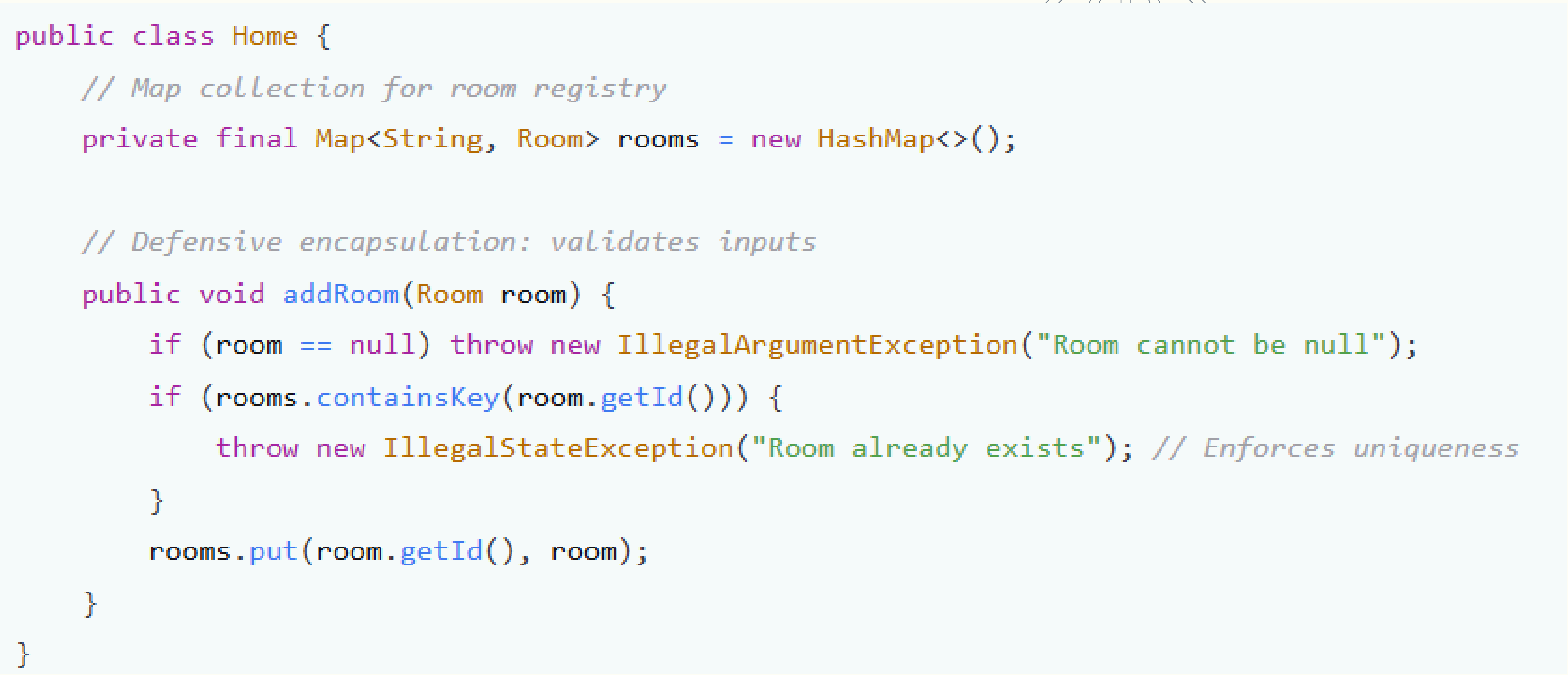
🔐 Device

🚨 Device

- **KEY RESPONSIBILITIES:**
  - **Home:** Central registry, device tracking, search interface
  - **Room:** Device storage, capacity management, energy tracking

Home Structure

Home Manages Multiple Rooms

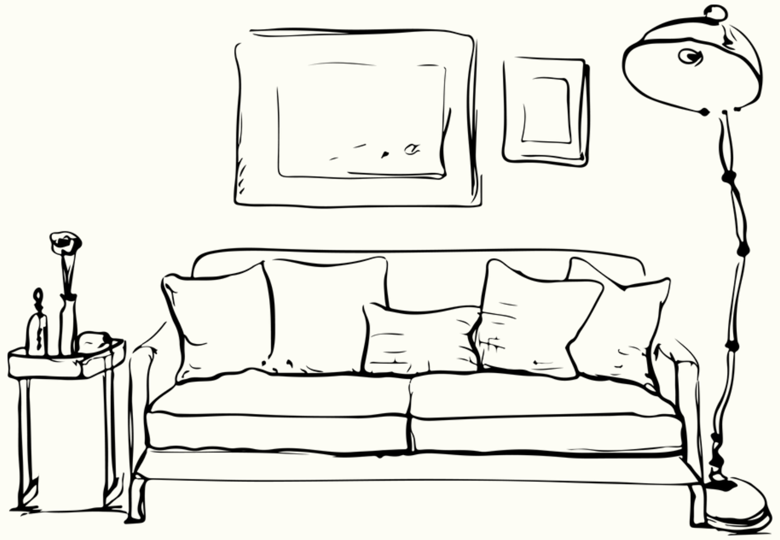


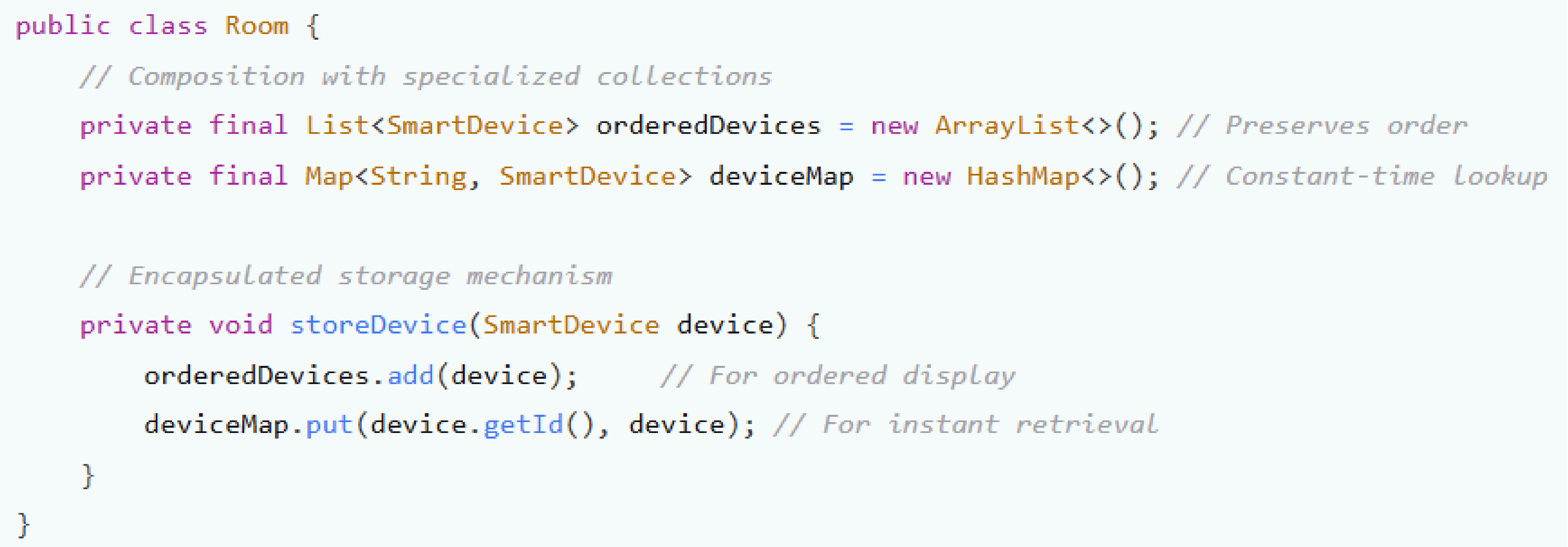
```java
public class Home {
    // Map collection for room registry
    private final Map<String, Room> rooms = new HashMap<>();

    // Defensive encapsulation: validates inputs
    public void addRoom(Room room) {
        if (room == null) throw new IllegalArgumentException("Room cannot be null");
        if (rooms.containsKey(room.getId())) {
            throw new IllegalStateException("Room already exists"); // Enforces uniqueness
        }
        rooms.put(room.getId(), room);
    }
}
```

- Home can store unlimited rooms
- Each room has a unique identifier
- Instant room retrieval by ID

Room Structure

Room Manages Multiple Devices



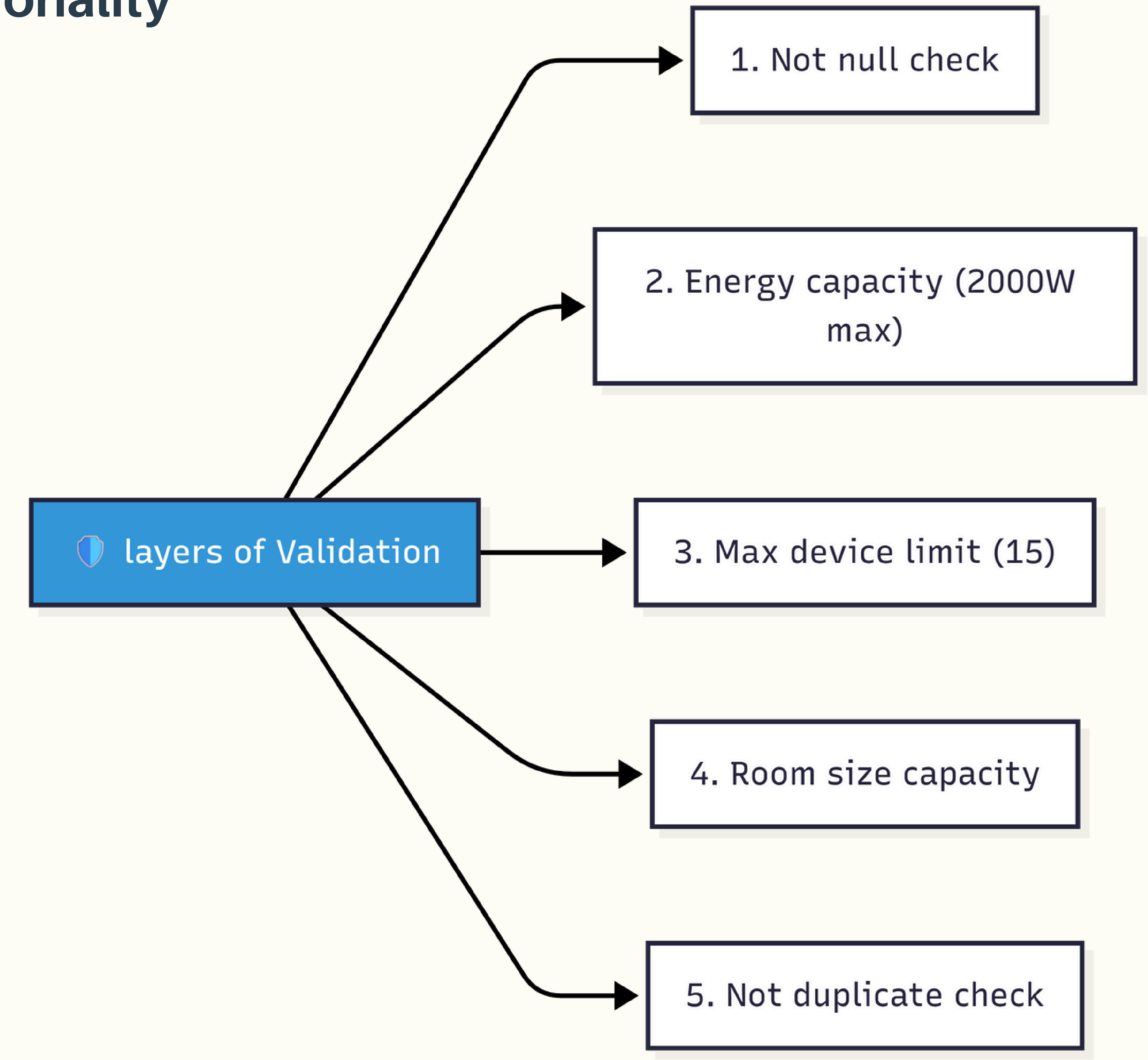


```java
public class Room {
    // Composition with specialized collections
    private final List<SmartDevice> orderedDevices = new ArrayList<>(); // Preserves order
    private final Map<String, SmartDevice> deviceMap = new HashMap<>(); // Constant-time lookup

    // Encapsulated storage mechanism
    private void storeDevice(SmartDevice device) {
        orderedDevices.add(device);     // For ordered display
        deviceMap.put(device.getId(), device); // For instant retrieval
    }
}
```

- Rooms store multiple devices
- Devices know their room location
- Fast device lookup by ID
- Maintains device display order

Complete Device Management

Add/Remove Functionality



🛡️ layers of Validation

1. Not null check

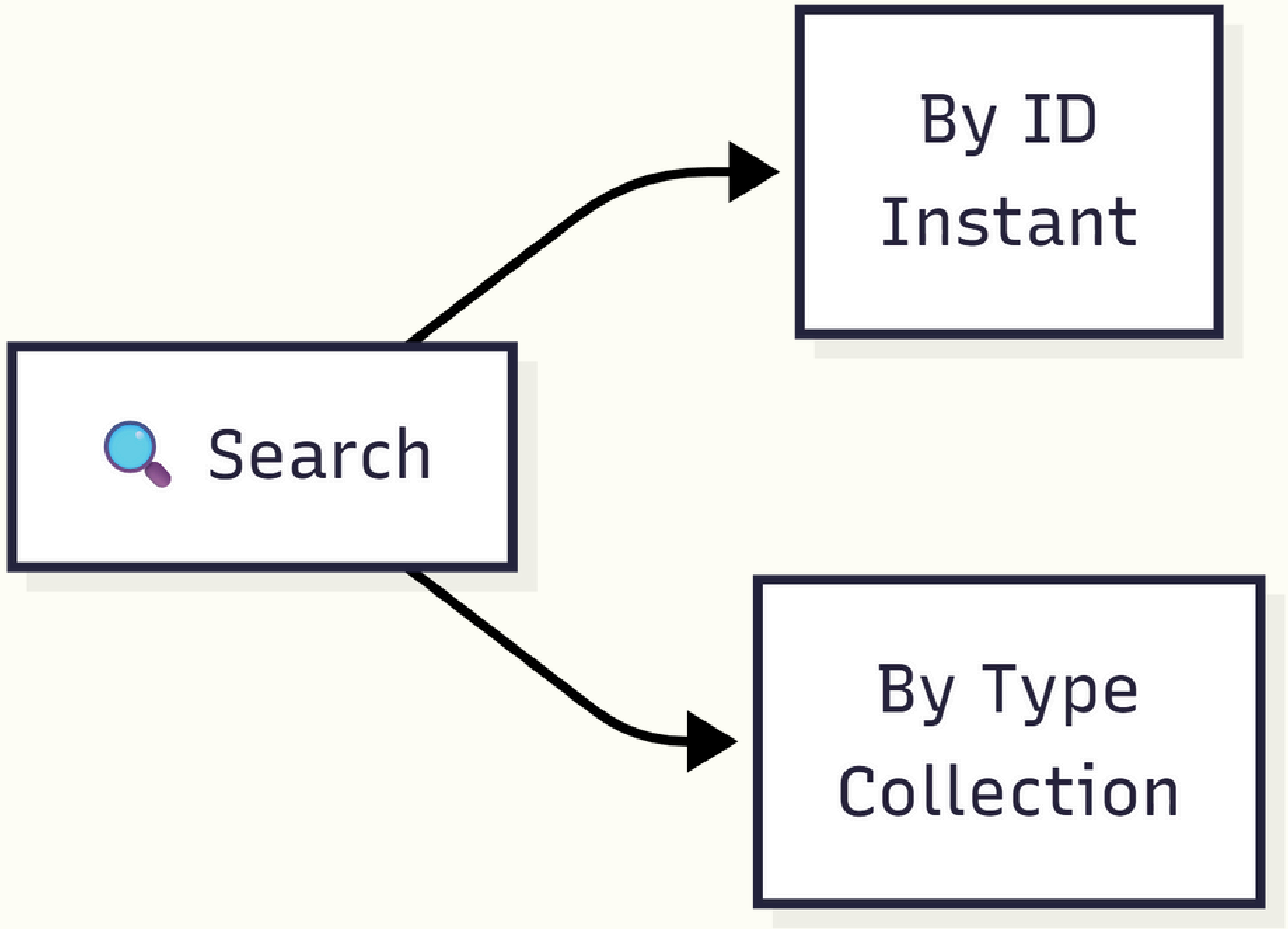
2. Energy capacity (2000W max)

3. Max device limit (15)

4. Room size capacity

5. Not duplicate check

Multiple Search Options



🔍 Search

By ID
Instant

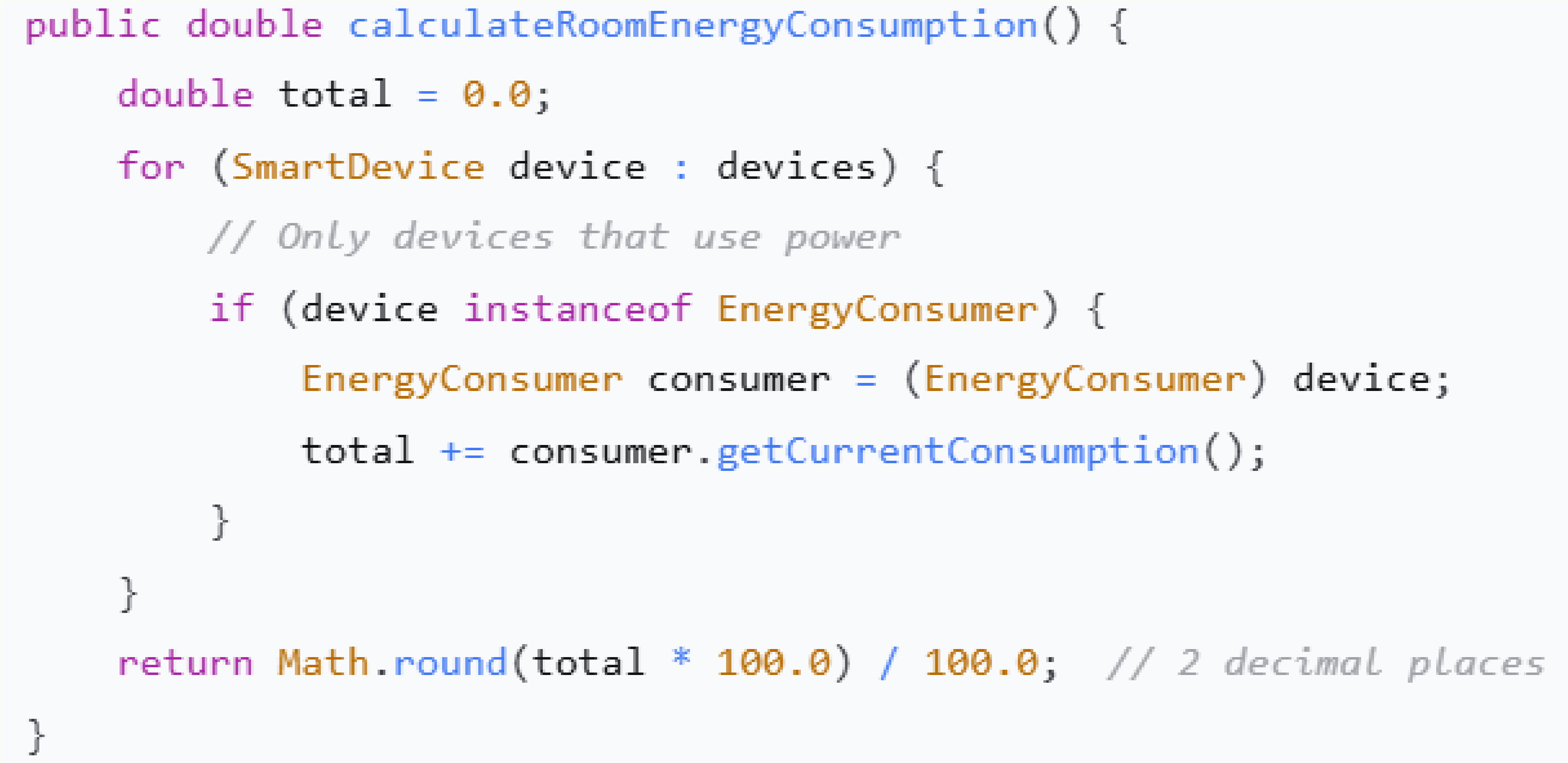
By Type
Collection

```java
public SmartDevice findDeviceById(String id) {
    return deviceMap.get(id); // Constant-time direct access
}
```

```java
public List<SmartDevice> findDevicesByType(String type) {
    List<SmartDevice> results = new ArrayList<>();
    for (SmartDevice device : devices) {
        if (device.getDeviceType().equalsIgnoreCase(type)) {
            results.add(device);  // Case-insensitive
        }
    }
    return results;
}
```

Energy Management

REAL-TIME TRACKING

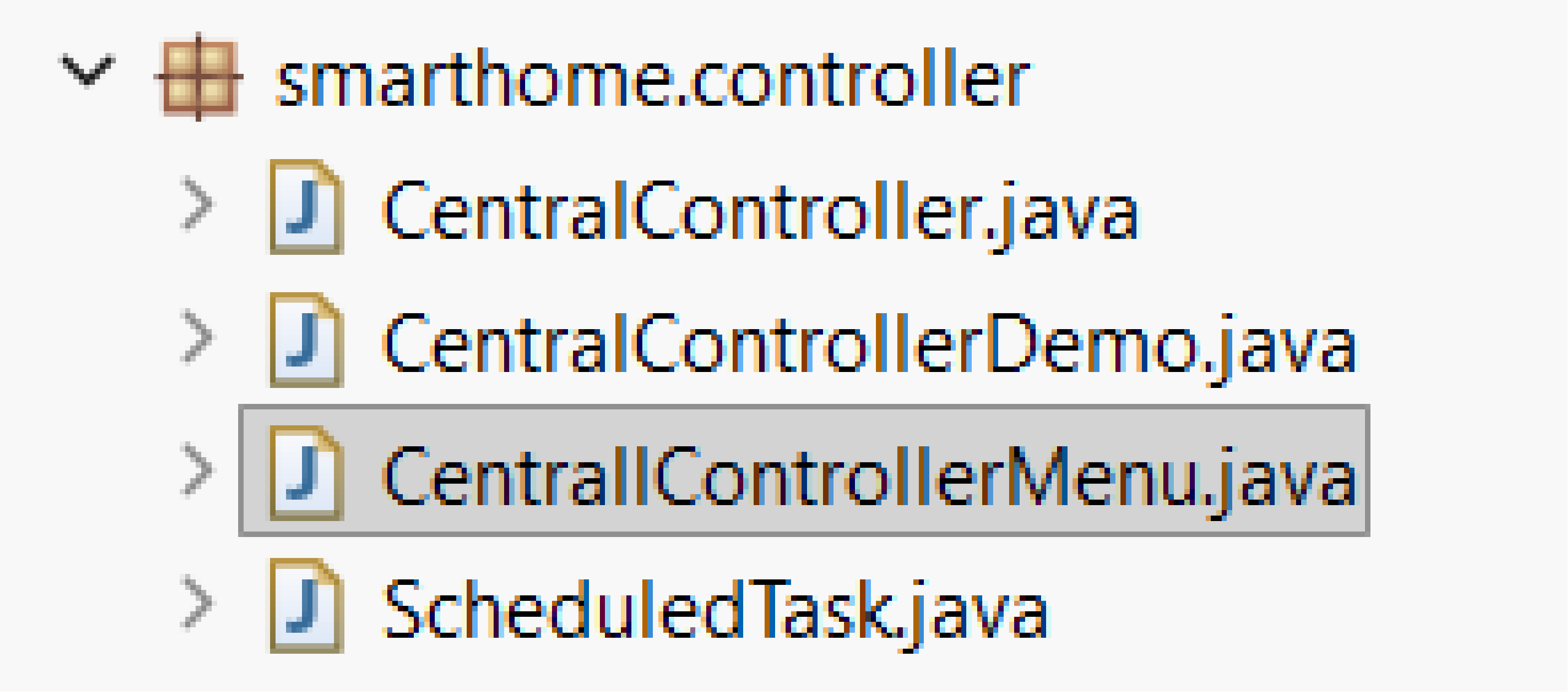


```java
public double calculateRoomEnergyConsumption() {
    double total = 0.0;
    for (SmartDevice device : devices) {
        // Only devices that use power
        if (device instanceof EnergyConsumer) {
            EnergyConsumer consumer = (EnergyConsumer) device;
            total += consumer.getCurrentConsumption();
        }
    }
    return Math.round(total * 100.0) / 100.0;  // 2 decimal places
}
```

- Calculates real-time usage
- Only includes power-consuming devices
- Commercial precision (2 decimals)
- Prevents adding devices that exceed limits

Central Controller

→ The Central Controller is a component that acts as the brain of the smart home by coordinating : devices, rooms, energy management, scheduling, and user interaction.



smarthome.controller
- CentralController.java
- CentralControllerDemo.java
- CentrallControllerMenu.java
- ScheduledTask.java

Core Responsibilities

- **Controls all smart devices through Home**
- **Provides:**

  - Device listing (by room, type, or ID)

  - Bulk actions (ON/OFF)

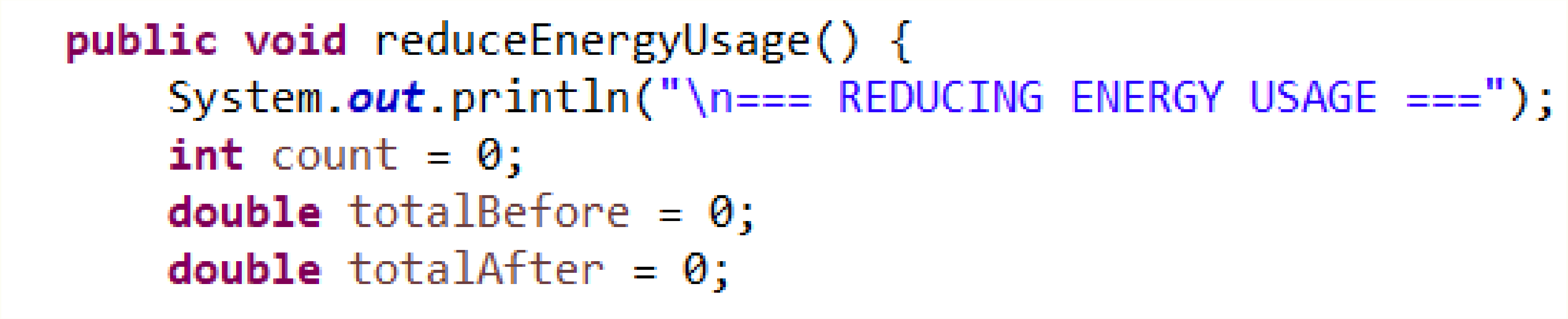
  - Energy monitoring & optimization
- **Managesautomation tasks** (scheduling)

→ **The CentralController does not control devices directly. Instead, it communicates with the Home class, which ensures good separation of concerns and clean architecture.**

Device & Energy Management

- **Turn ON/OFF:** (All devices / Devices by room / Devices by type)
- **Energy features:** (Total consumption calculation / Room-level energy report/ Automatic energy reduction for lights)

→ For energy optimization, I implemented a **method** that **automatically reduces light brightness** if it exceeds a threshold. This simulates a real smart-home energy-saving strategy.



```java
public void reduceEnergyUsage() {
    System.out.println("\n=== REDUCING ENERGY USAGE ===");
    int count = 0;
    double totalBefore = 0;
    double totalAfter = 0;
```

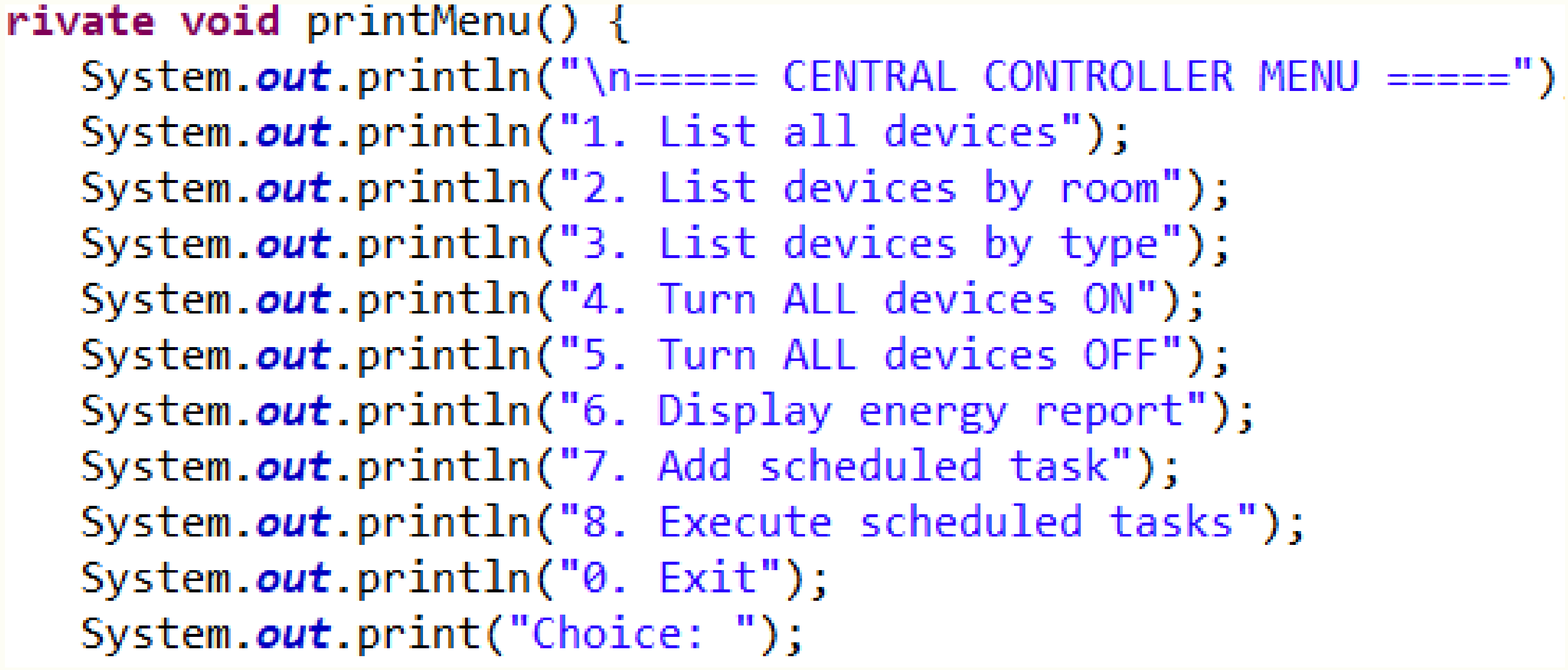
Scheduled Task

- **Represents an automated action**
- **Attributes:**(Time, Action(ON,OFF,SET_TEMPERATURE...)**,** **Target** (device, room, or type))
- **Supports parameters** (temperature, brightness)

→ Each ScheduledTask knows what action to perform, when to perform it, and on which target. This makes automation **flexible** and **scalable**.

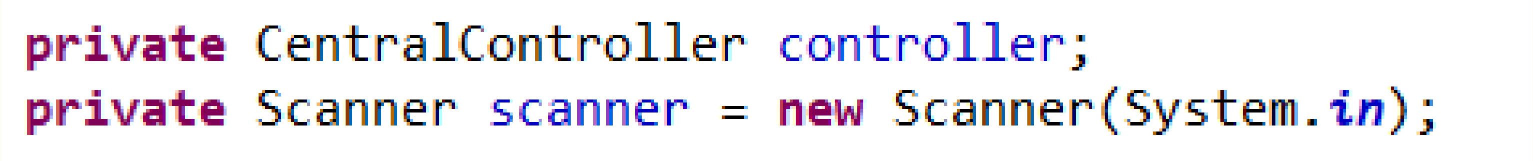
User Interaction – CentralControllerMenu

- Console-based menu
- Allows user to:
  - List devices
  - Control devices
  - View energy report
  - Add & execute scheduled tasks



```
rivate void printMenu() {
    System.out.println("\n===== CENTRAL CONTROLLER MENU =====")
    System.out.println("1. List all devices");
    System.out.println("2. List devices by room");
    System.out.println("3. List devices by type");
    System.out.println("4. Turn ALL devices ON");
    System.out.println("5. Turn ALL devices OFF");
    System.out.println("6. Display energy report");
    System.out.println("7. Add scheduled task");
    System.out.println("8. Execute scheduled tasks");
    System.out.println("0. Exit");
    System.out.print("Choice: ");
```

- Uses Scanner for input

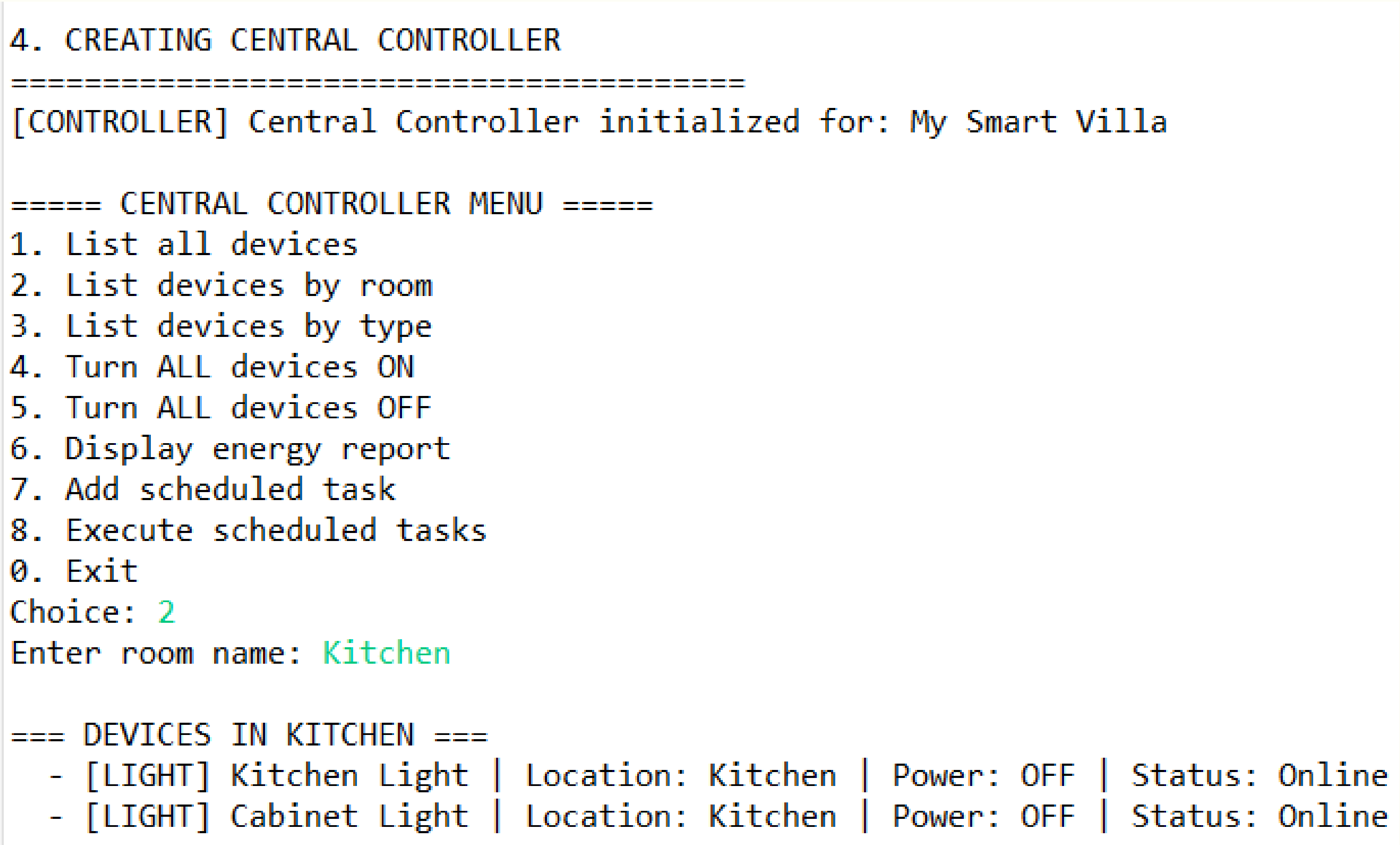


```
private CentralController controller;
private Scanner scanner = new Scanner(System.in);
```

→ I implemented a console menu so the system can be used interactively, similar to a real smart-home dashboard.

Demo Execution & Conclusion

- **Demo shows:**
  - Device creation & assignment
  - Controller initialization
  - Automation execution by time
- **Benefits:**
  - Modular design
  - Easy extension
  - Real-world simulation



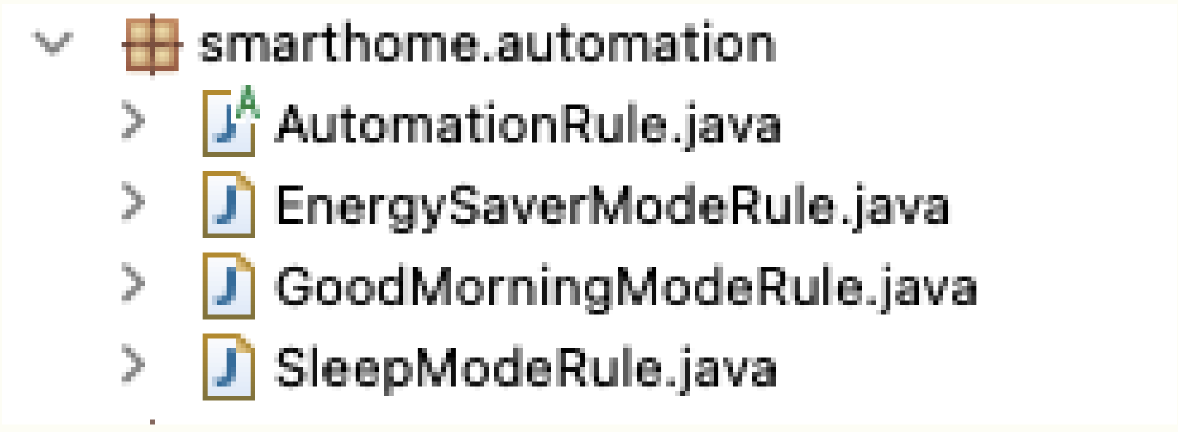
```
4. CREATING CENTRAL CONTROLLER
========================================
[CONTROLLER] Central Controller initialized for: My Smart Villa

===== CENTRAL CONTROLLER MENU =====
1. List all devices
2. List devices by room
3. List devices by type
4. Turn ALL devices ON
5. Turn ALL devices OFF
6. Display energy report
7. Add scheduled task
8. Execute scheduled tasks
0. Exit
Choice: 2
Enter room name: Kitchen

=== DEVICES IN KITCHEN ===
  - [LIGHT] Kitchen Light | Location: Kitchen | Power: OFF | Status: Online
  - [LIGHT] Cabinet Light | Location: Kitchen | Power: OFF | Status: Online
```

→ **In conclusion, my part ensures centralized control, automation, and scalability. The design follows good object-oriented principles and can easily be extended with new devices or actions.**

Automation System

AutomationRule - Abstract Base Class

```
abstract boolean checkCondition(Home home)
// Evaluates if rule should trigger

abstract void executeAction(Home home)
// Performs the automation actions

boolean evaluate(Home home)
// Complete evaluation cycle
```



smarthome.automation
- AutomationRule.java
- EnergySaverModeRule.java
- GoodMorningModeRule.java
- SleepModeRule.java

☀️ Good Morning ModeRule

Triggers:

- **Time-based:** 6:00 AM (customizable)
- **Manual activation anytime**

Actions:

- 💡 **Bedroom lights** → 80% (gradual)
- 💡 **Bathroom lights** → 100% (full)
- 💡 **Kitchen/Living** → 60% (medium)
- 🌡️ **All thermostats** → 22°C (comfort)

🌙 Sleep Mode Rule

Triggers:

- **Time-based:** 11:00 PM (customizable)
- **Manual activation anytime**

Actions:

- 💡 **Bedroom lights** → 10% (night light)
- 💡 **All other lights** → OFF
- 📺 **All TVs** → OFF
- 🌡️ **All thermostats** → 18°C (sleep)

⚡ Energy Saver Mode Rule

Triggers:

- **Inactivity:** 30 minutes (configurable)
- **Manual activation anytime**
- **Activity tracking per room**

- **Actions:**
- 💡 **All lights** → OFF
- 📺 **All TVs** → OFF
- 🌡️ **Thermostats** → Eco mode (-3°C)

Automation Rule Logic (WHEN–THEN Flow)

WHEN (checkCondition):

Option 1: manuallyActivated = true

Option 2: Current time = exactly 6:00 AM

THEN (executeAction):

```
For EACH room in house:
  ├──── If BEDROOM:    Lights → 80% brightness
                       Message: "gradually brightened"
  ├──── If BATHROOM:  Lights → 100% brightness
                       Message: "turned on (100%)"
  ├──── If KITCHEN or LIVING ROOM:  Lights → 60% brightness
                                    Message: "turned on (60%)"
  └──── If THERMOSTAT: Temperature → 22°C
                         Message: "set to 22°C"
```

Exceptions Handling

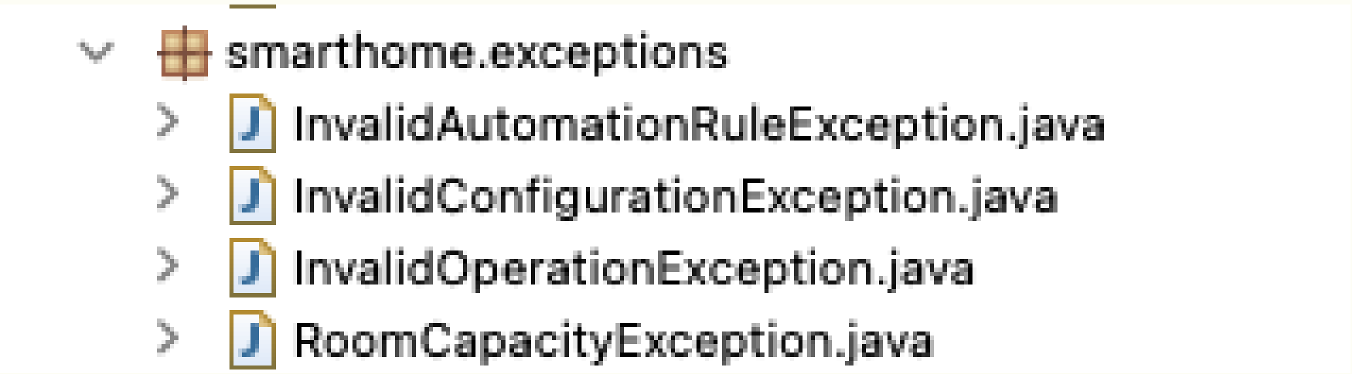
1. Invalid Configuration Exception

Purpose:

Validates configuration parameters across the entire system

Check :

- Invalid brightness values (must be 0-100)
- Temperature out of bounds (10°C - 35°C)
- Invalid time format in schedules



```
smarthome.exceptions
  > InvalidAutomationRuleException.java
  > InvalidConfigurationException.java
  > InvalidOperationException.java
  > RoomCapacityException.java
```

Exceptions Handling

2. Invalid Operation Exception

Purpose:

Prevents invalid operations on devices and system components

Check :

- Device offline - cannot turn on
- TV recording in progress - cannot turn off
- Invalid state transitions

Exceptions Handling

3. Room Capacity Exception

Purpose:

Enforces room capacity constraints for safety and efficiency

Capacity Checks:

- **Device Count Limit Maximum:**
  15 devices per room
- **Density CheckFormula:**
  1 device per $2m^2$
- **Energy LimitMaximum:**
  2000W per room

Exceptions Handling

4. Invalid Automation Rule Exception

Purpose:

Validates automation rules before creation and execution

Capacity Checks:

- Null rule name provided
- Invalid time format for triggers
- Missing or invalid parameters

Implementation of OOP Concepts

	Where it Appears	How We Used It	Benefit
Abstraction	In our base *Device* class and interface definitions	We hid complex device communication logic behind simple methods like *.turnOn()* and *.getStatus().*	allowed us to work with high-level commands without worrying about low-level protocol details, simplifying the entire codebase.

Implementation of OOP Concepts

	Where it Appears	How We Used It	Benefit
Encapsulation	In every class, especially device classes (e.g, *SmartLight*, *Thermostat*).	We kept device state private like *brightness* and provided public getter/setter methods with validation.	protected the internal data from corruption, ensured data integrity, and made debugging easier by localizing state management.

Implementation of OOP Concepts

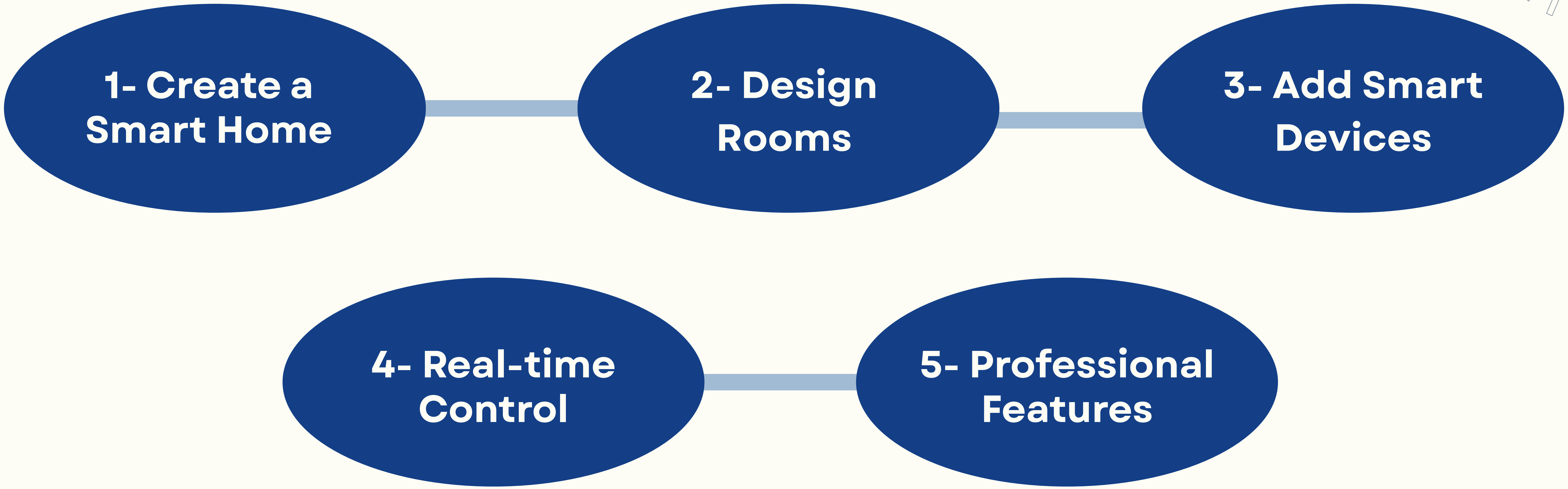
	Where it Appears	How We Used It	Benefit
Composition	In the system structure (*Home → Room → Device*).	We built our architecture using "has-a" relationships instead of inheritance.	composition creates independent, reusable parts that can be assembled and rearranged naturally—just like in a real home !

Implementation of OOP Concepts

	Where it Appears	How We Used It	Benefit
Polymorphism	In the central *DeviceController* and the UI rendering loop.	We treated all devices as generic *Device* objects, but calling .turnOn() executed the specific implementation for a light, plug, or thermostat.	let us write one block of control logic that works for any device, making the system incredibly flexible and extensible.

MAIN APPLICATION IN ACTION

What You'll See in 60 Seconds



1- Create a Smart Home

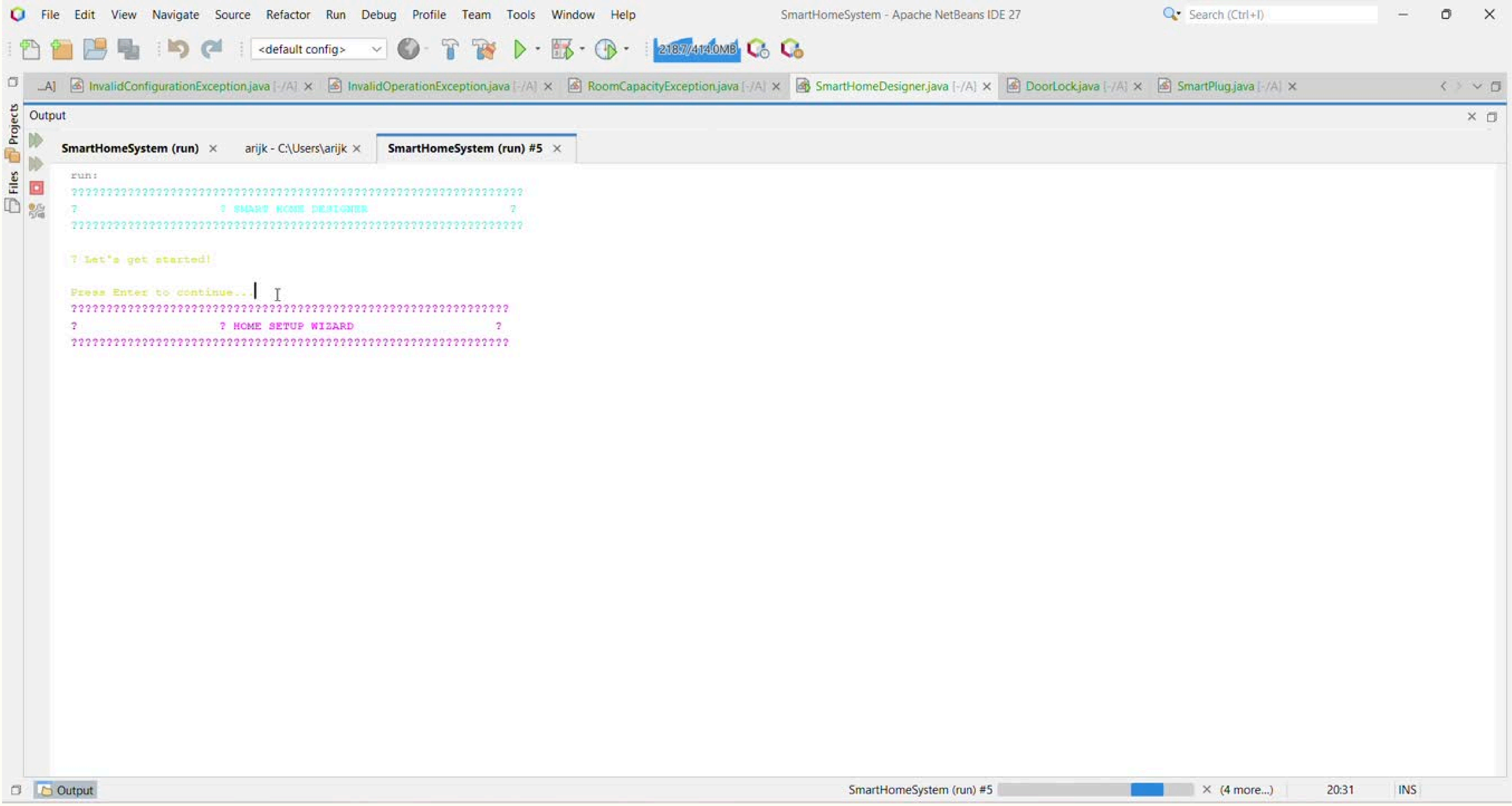
2- Design Rooms

3- Add Smart Devices

4- Real-time Control

5- Professional Features

MAIN APPLICATION IN ACTION



```
run:
?????????????????????????????????????????????????????????????????
?                    ? SMART HOME DESIGNER                      ?
?????????????????????????????????????????????????????????????????

? Let's get started!

Press Enter to continue...
???????????????????????????????????????????????????????????????
?                    ? HOME SETUP WIZARD                      ?
???????????????????????????????????????????????????????????????
```

THANK YOU!