***DAY 24***

### ***Task 01: Singleton Design Pattern Demo***

**UML :**

@startuml

title Singleton Pattern - SingletonDemo

class SingletonDemo {

-instance: SingletonDemo

-SingletonDemo()

+getInstance(): SingletonDemo

+doHere()

}

class SingletonDP {

+main(args: String[])

}

SingletonDemo <.. SingletonDP : uses

note top of SingletonDemo

- Private constructor prevents direct instantiation

- getInstance() ensures only one object exists

end note

@enduml

### 

***DAY 25***

### ***Task 01: Facade Design Pattern Demo***

**UML:**  
  
@startuml

title Facade Design Pattern - Bank Services

class FdService {

+ getFdServiceDetails(accountNo : String)

}

class LoanService {

+ getLoanDetails(accountNo : String)

}

class AccountService {

+ getAccountDetails(accountNo : String)

}

class BankFacade {

- fdService : FdService

- loanService : LoanService

- accountService : AccountService

+ BankFacade()

+ getFdDetails(accountNo : String)

+ getLoanDetails(accountNo : String)

+ getAccountDetails(accountNo : String)

}

class FacadePatternDemo {

+ main(args : String[])

}

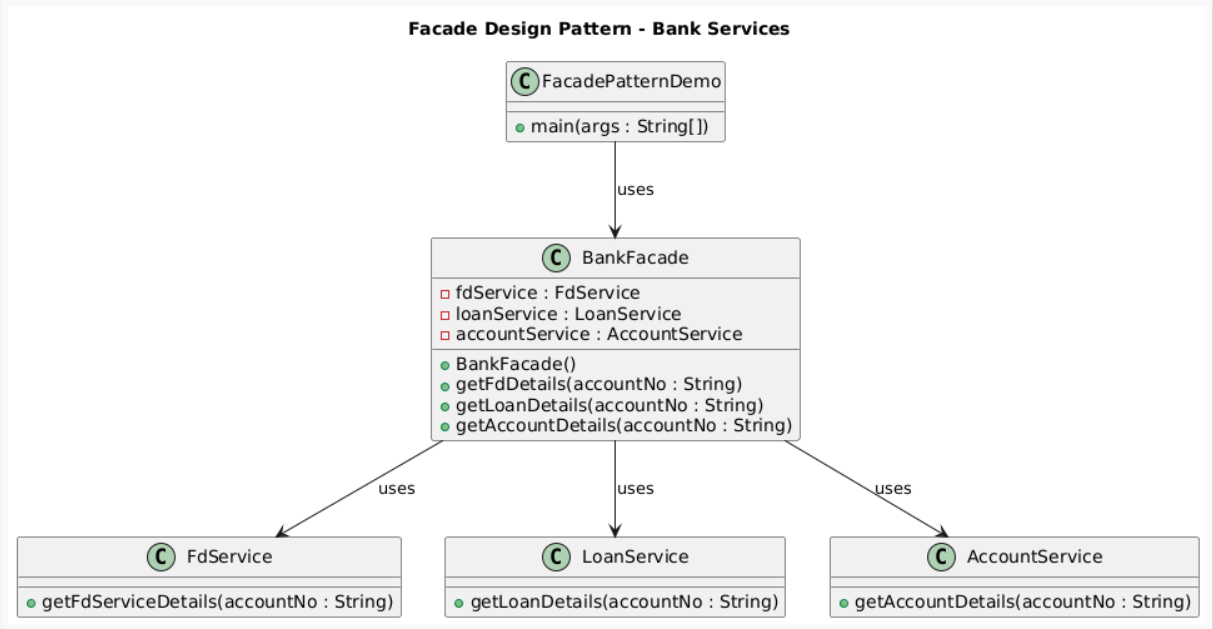
BankFacade --> FdService : uses

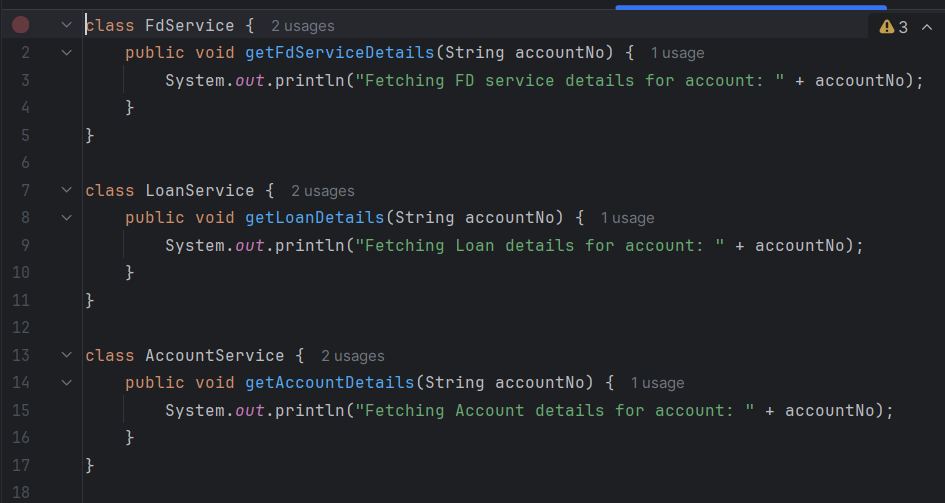
BankFacade --> LoanService : uses

BankFacade --> AccountService : uses

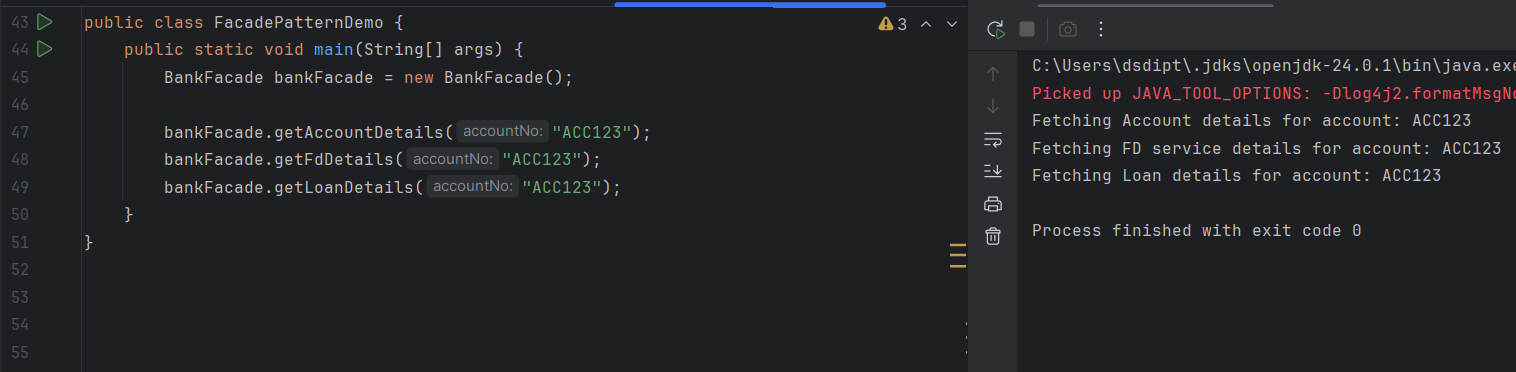
FacadePatternDemo --> BankFacade : uses

@enduml









* The **client** (FacadePatternDemo) talks only to BankFacade.
* BankFacade internally coordinates **multiple services** (FdService, LoanService, AccountService).
* This hides complexity and makes the client’s code super clean.

### ***Task 02: Composite Method Design Pattern Demo***

**UML:**

@startuml

title Composite Pattern - Laptop Brands

interface Component {

+brandName()

}

class Leaf implements Component {

-brand: String

+Leaf(brand: String)

+brandName()

}

class Composite implements Component {

-components: List<Component>

-groupName: String

+Composite(groupName: String)

+add(subComponent: Component)

+remove(subComponent: Component)

+brandName()

}

class CompositePatternDemo {

+main(args: String[])

}

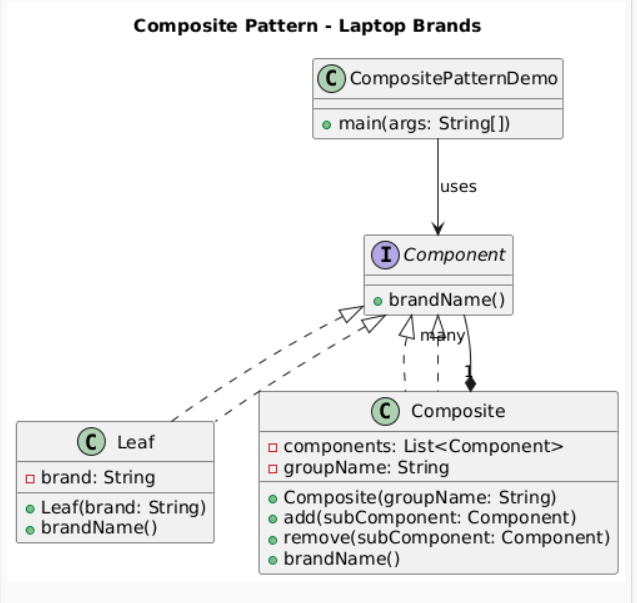
Component <|.. Leaf

Component <|.. Composite

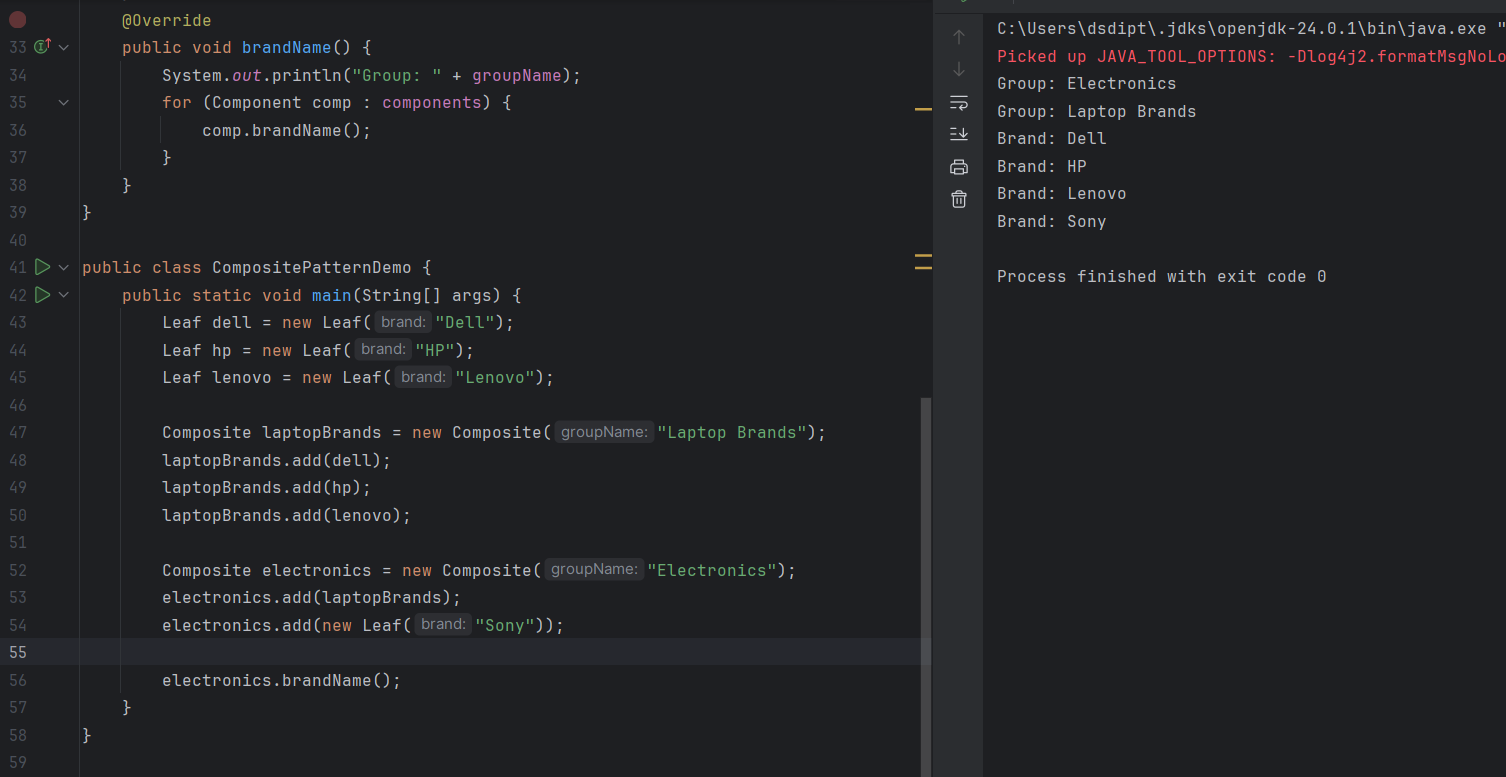
Composite "1" \*-- "many" Component

CompositePatternDemo --> Component : uses

@enduml







### 

### ***Task 03: Proxy Method Design Pattern Demo***

**UML:**

@startuml

title Proxy Pattern - DB Access Control

interface DBExecutor {

+runQuery(type: String)

}

class DB {

+DB()

+runIt(type: String, id: String)

}

class DBProxy implements DBExecutor {

-id: String

-db: DB

+DBProxy(id: String)

+runQuery(type: String)

}

class ClientProxyDP {

+main(args: String[])

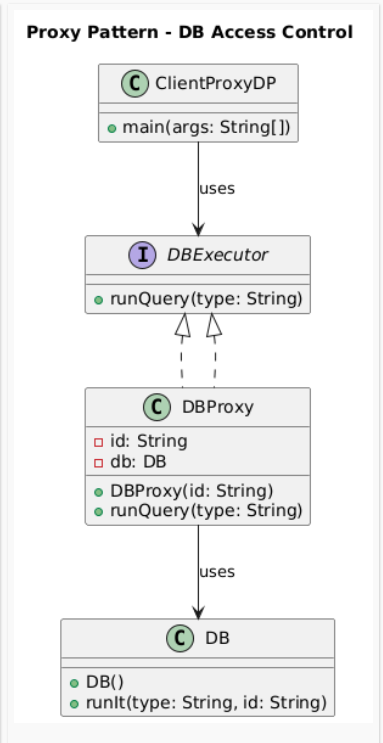
}

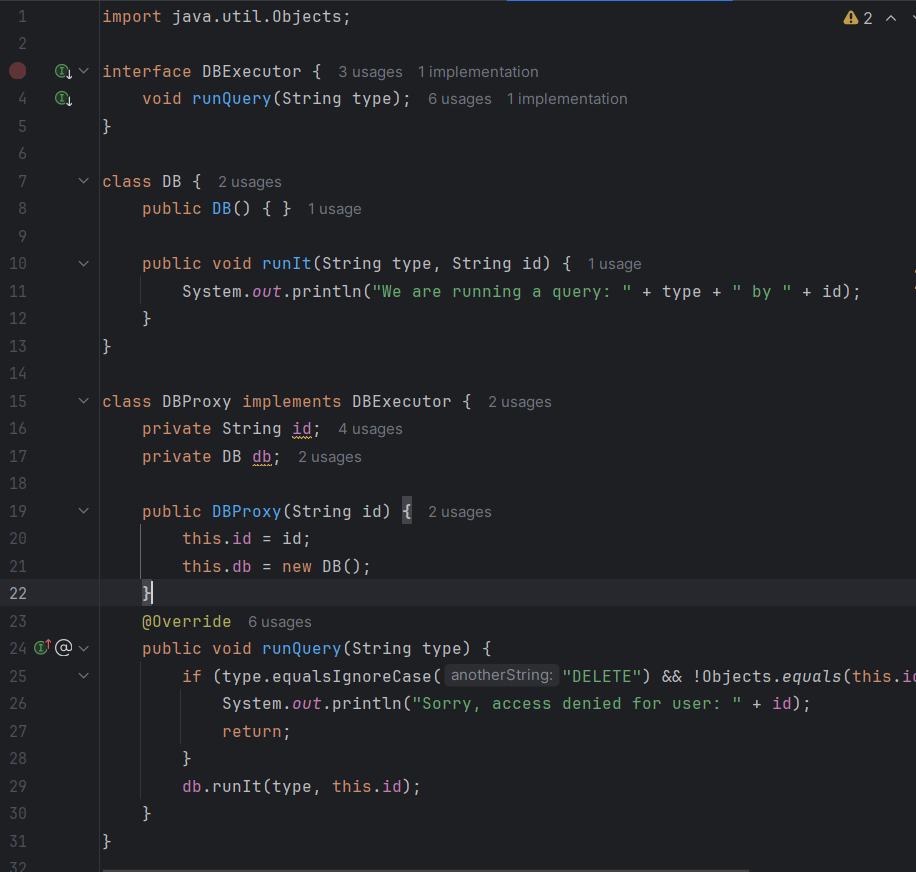
DBExecutor <|.. DBProxy

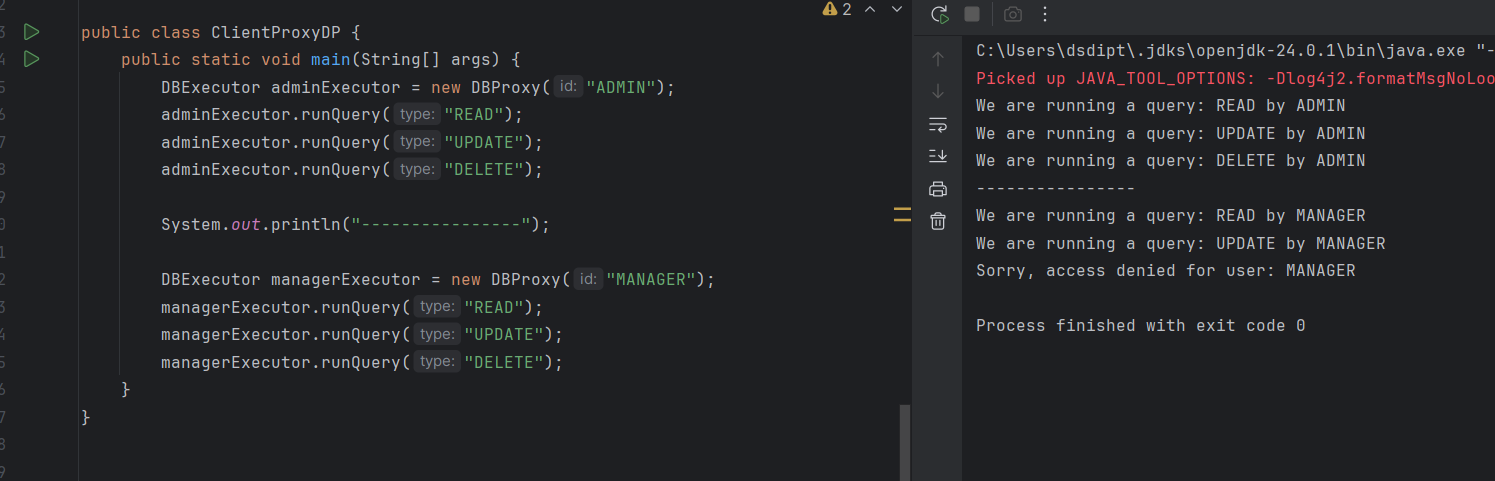
DBProxy --> DB : uses

ClientProxyDP --> DBExecutor : uses

@enduml







### 

### ***Task 04: Flyweight Method Design Pattern Demo***

**UML:**

@startuml

title Flyweight Pattern - Shape Example

interface Shape {

+draw(x: int, y: int)

}

class Circle implements Shape {

-color: String

+Circle(color: String)

+draw(x: int, y: int)

}

class ShapeFactory {

-circleMap: Map<String, Shape>

+getCircle(color: String): Shape

}

class FlyweightDemo {

+main(args: String[])

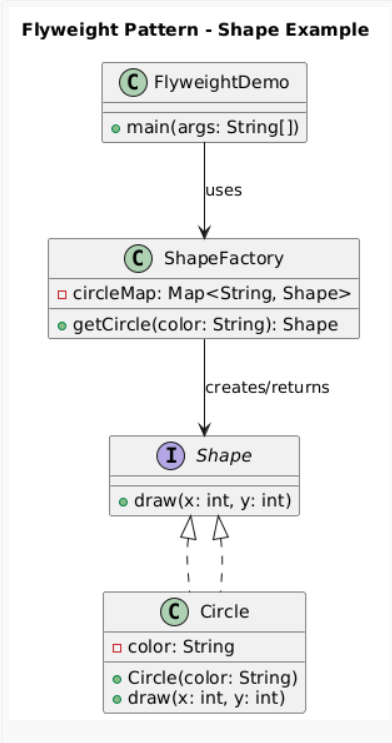
}

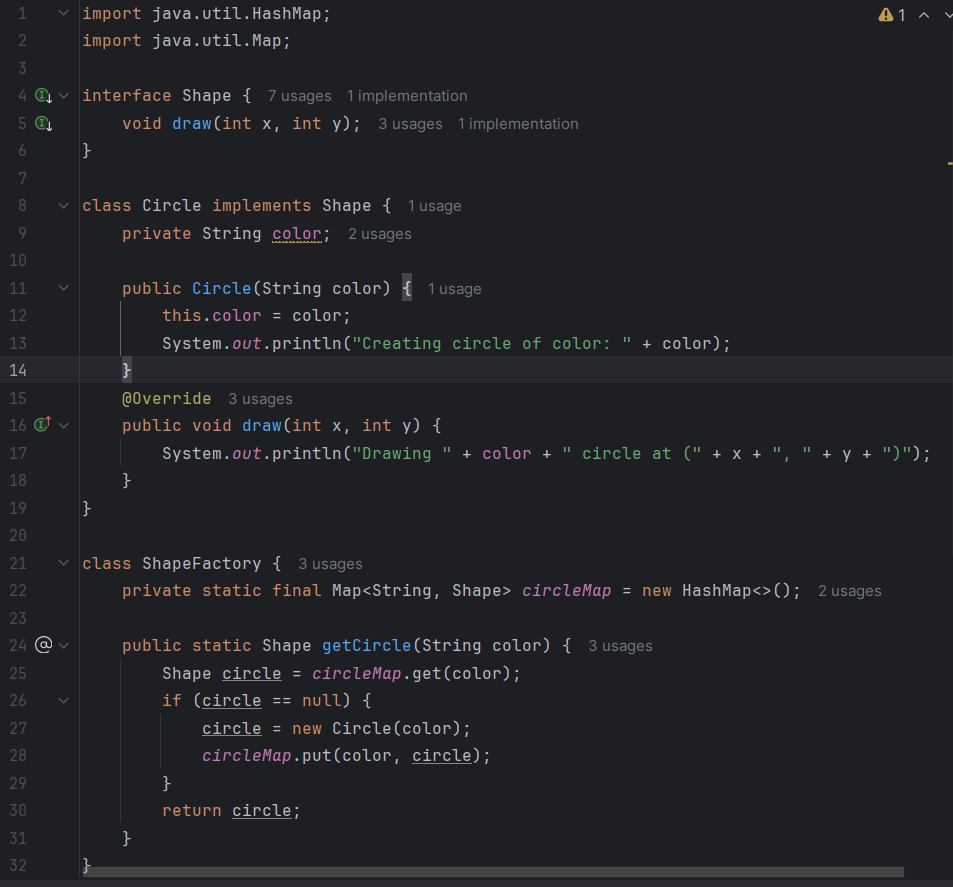
Shape <|.. Circle

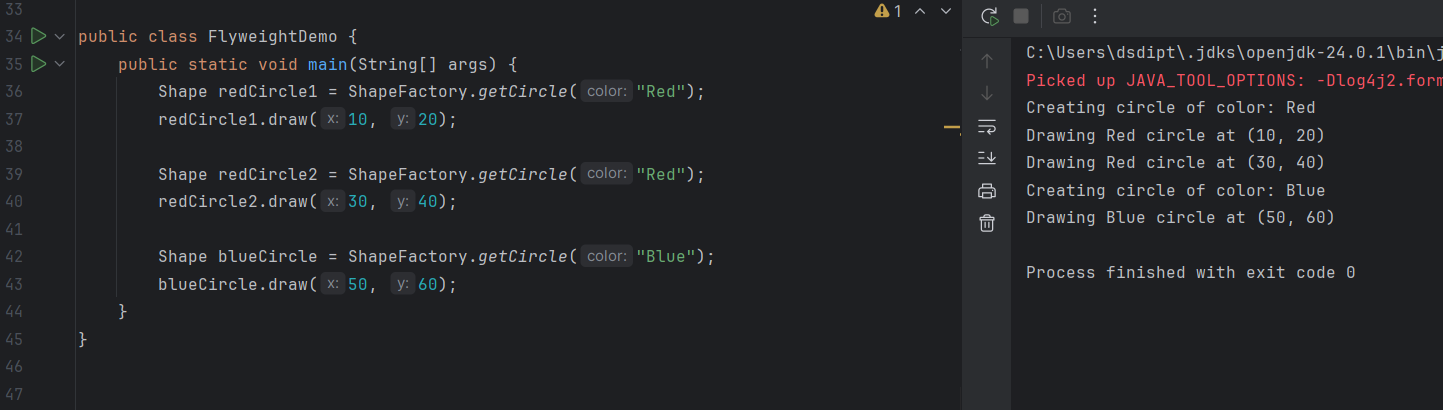
ShapeFactory --> Shape : creates/returns

FlyweightDemo --> ShapeFactory : uses

@enduml







"Red" circle is created **only once**, even though it’s used twice.

That’s the **memory-saving magic** of Flyweight.

### ***Task 05: Bridge Method Design Pattern Demo***

**UML:**  
  
@startuml

title Bridge Pattern - Remote & Devices

interface Device {

+turnOn()

+turnOff()

+setVolume(volume: int)

}

class TV implements Device

class Radio implements Device

abstract class RemoteControl {

#device: Device

+RemoteControl(device: Device)

+turnOn()

+turnOff()

+setVolume(volume: int)

}

class BasicRemote extends RemoteControl

class BridgePatternDemo {

+main(args: String[])

}

Device <|.. TV

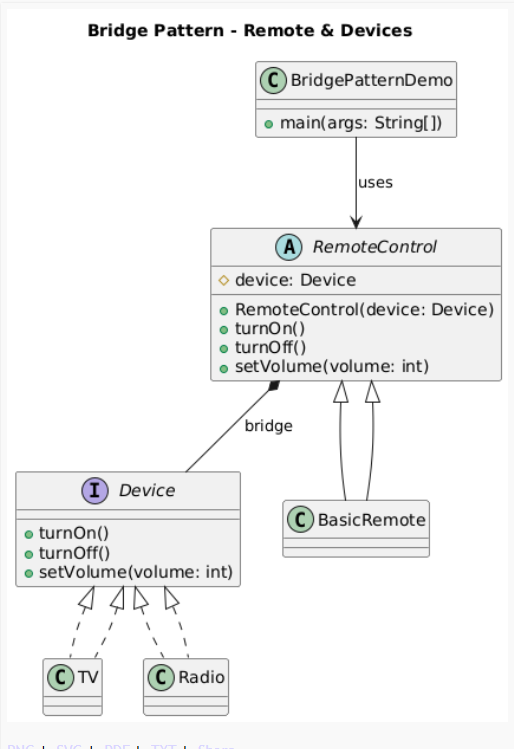
Device <|.. Radio

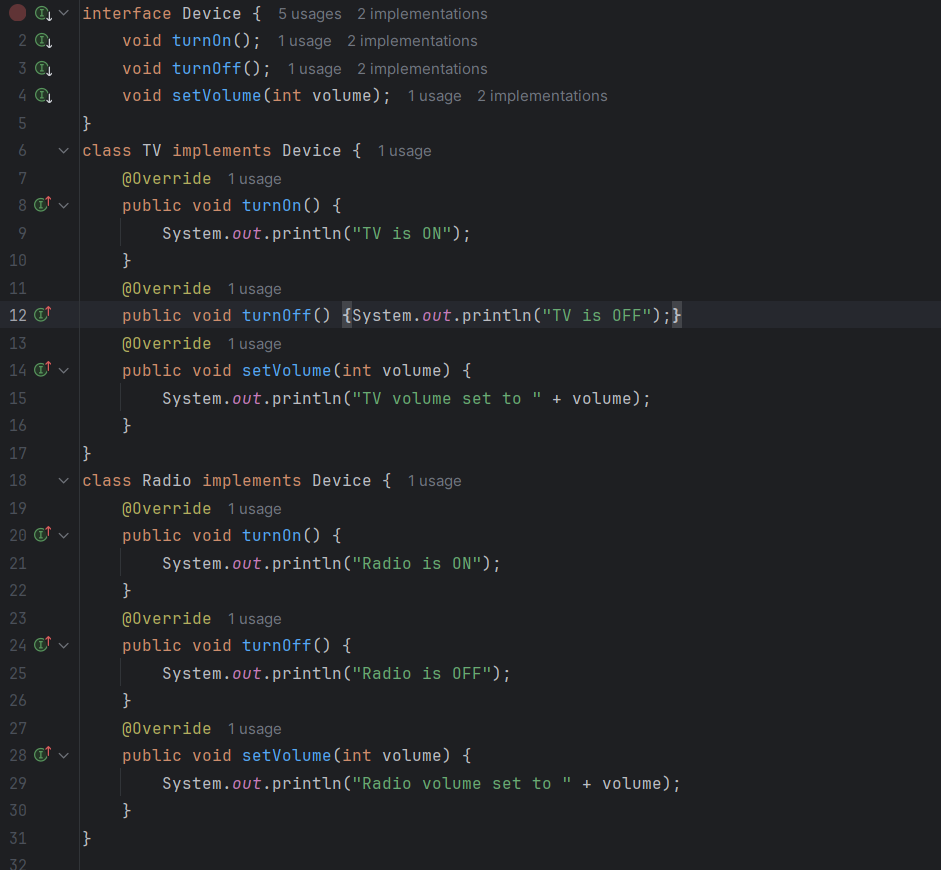
RemoteControl \*-- Device : "bridge"

BasicRemote --|> RemoteControl

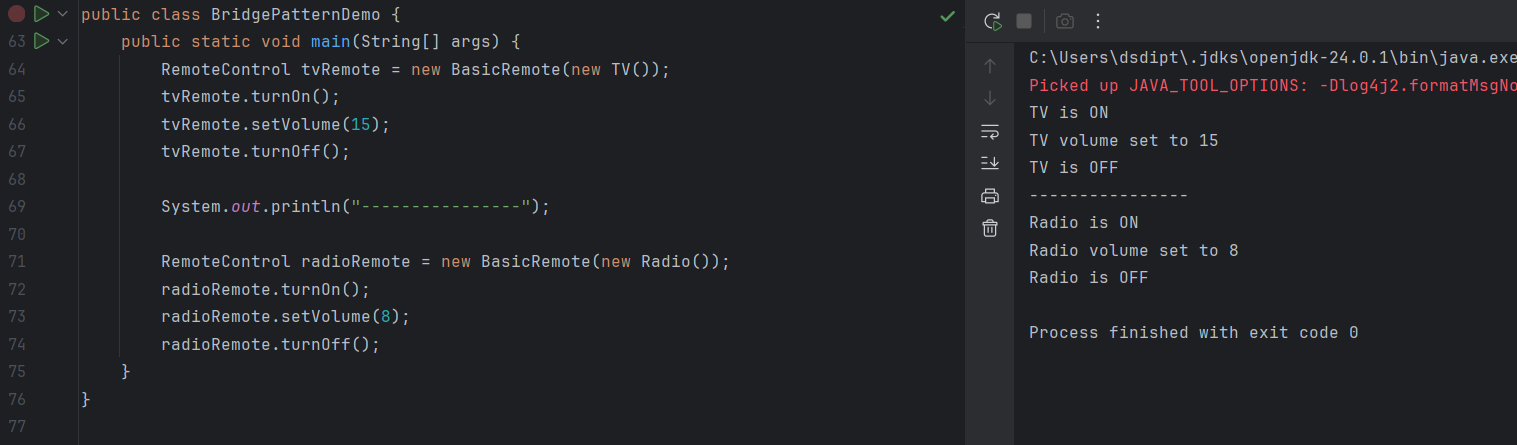
BridgePatternDemo --> RemoteControl : uses

@enduml









* **RemoteControl** (abstraction) holds a reference to a **Device** (implementation).
* We can add **new devices** (like SmartSpeaker) or **new remotes** (like AdvancedRemote) independently.
* Prevents a messy class explosion.

### ***Task 06: Prototype Method Design Pattern Demo***

**UML:**  
  
@startuml

title Prototype Method DP - Plant Example

interface Plant {

+grow(): Plant

+getGrowthType(): String

+setGrowthType(type: String)

+clone(): Plant

}

class Creeper implements Plant {

-growthType: String

+Creeper(growthType: String)

+grow(): Plant

+getGrowthType(): String

+setGrowthType(type: String)

+clone(): Plant

}

class Shrub implements Plant {

-growthType: String

+Shrub(growthType: String)

+grow(): Plant

+getGrowthType(): String

+setGrowthType(type: String)

+clone(): Plant

}

class PrototypePatternDemo {

+main(args: String[])

}

Plant <|.. Creeper

Plant <|.. Shrub

PrototypePatternDemo --> Plant : uses

note right of Plant

Prototype Interface

- Defines cloning method

- Defines plant growth behavior

end note

@enduml

