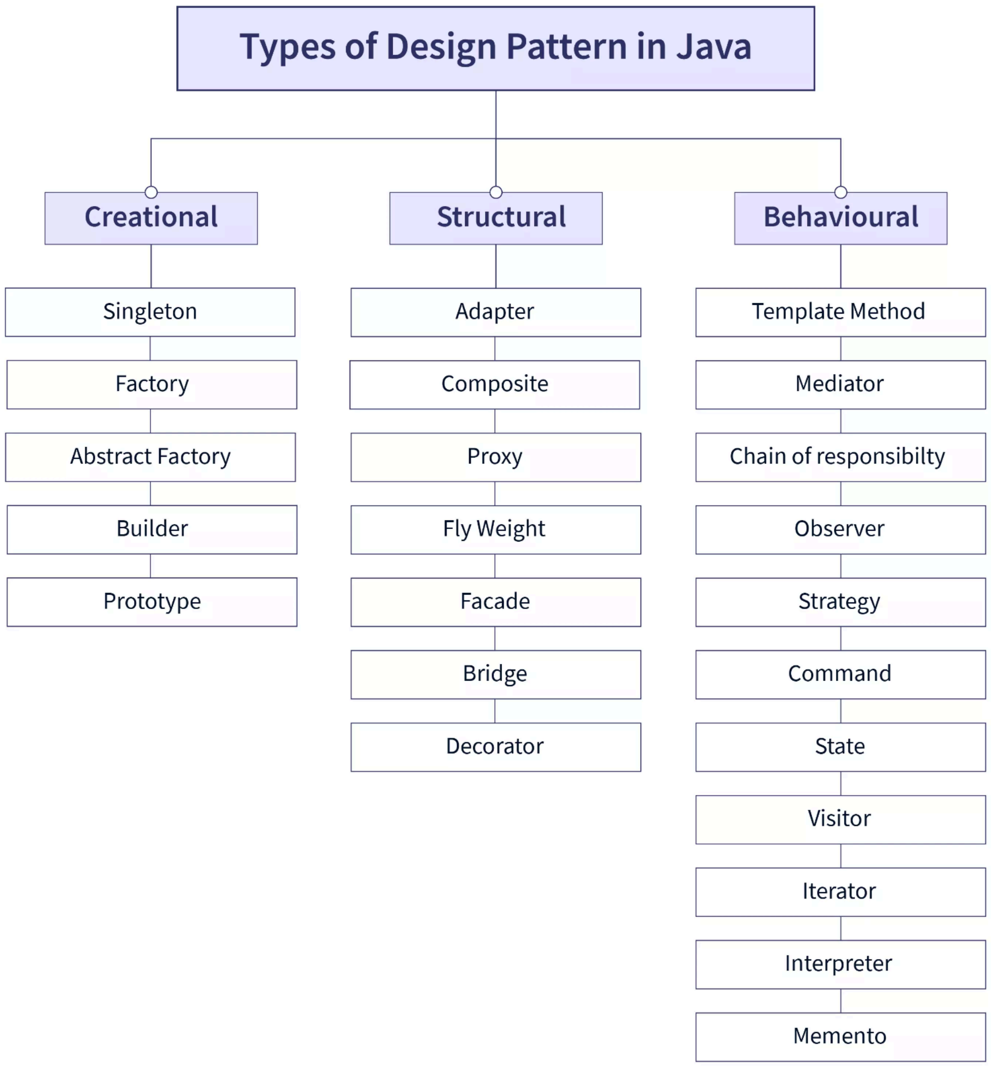
What Are Design Patterns?

Design patterns are like **recipes** for solving common problems in coding. Instead of inventing a new solution every time, you follow a proven approach that others have used successfully.

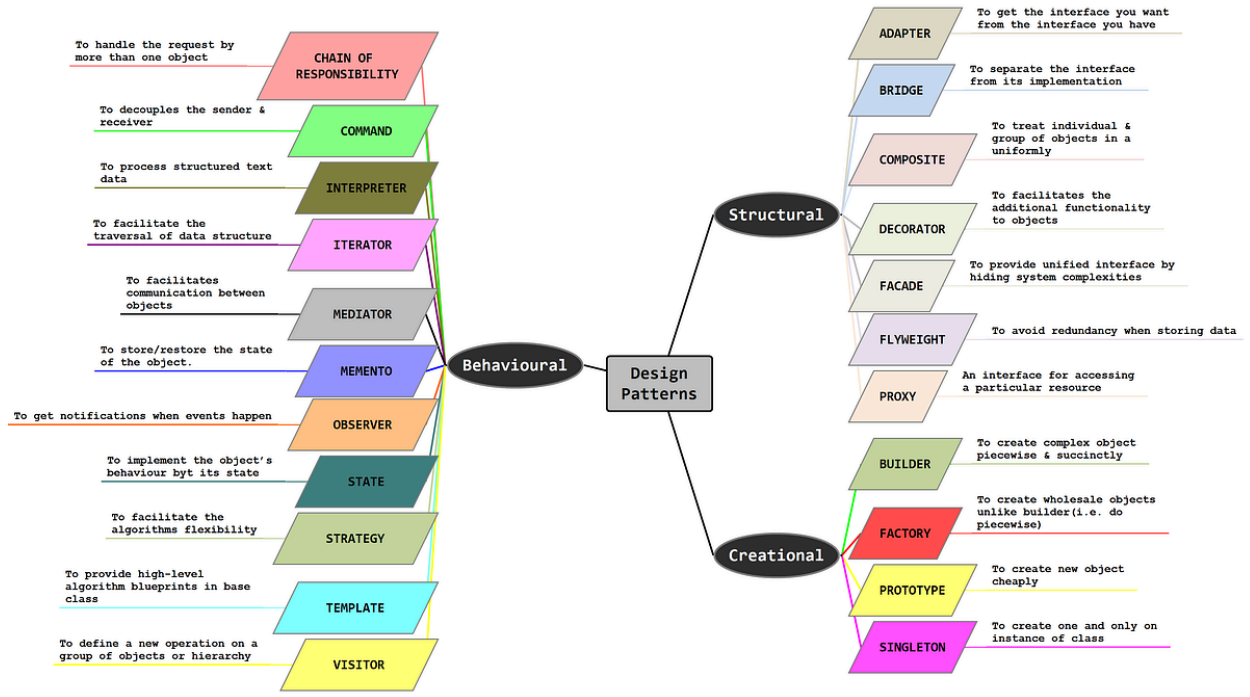
Think of them as **building blocks** or **templates** in software.



Why Use Design Patterns in Scala?

Scala is a **powerful hybrid language** (both object-oriented and functional). Design patterns in Scala help you:

- **Write reusable code**
- **Avoid repeating mistakes**
- **Keep your code clean and scalable**


Design Patterns
Behavioural
To handle the request by more than one object
CHAIN OF RESPONSIBILITY
To decouples the sender & receiver
COMMAND
To process structured text data
INTERPRETER
To facilitate the traversal of data structure
ITERATOR
To facilitates communication between objects
MEDIATOR
To store/restore the state of the object.
MEMENTO
To get notifications when events happen
OBSERVER
To implement the object's behaviour byt its state
STATE
To facilitate the algorithms flexibility
STRATEGY
To provide high-level algorithm blueprints in base class
TEMPLATE
To define a new operation on a group of objects or hierarchy
VISITOR
Structural
ADAPTER
To get the interface you want from the interface you have
BRIDGE
To separate the interface from its implementation
COMPOSITE
To treat individual & group of objects in a uniformly
DECORATOR
To facilitates the additional functionality to objects
FACADE
To provide unified interface by hiding system complexities
FLYWEIGHT
To avoid redundancy when storing data
PROXY
An interface for accessing a particular resource
Creational
BUILDER
To create complex object piecewise & succinctly
FACTORY
To create wholesale objects unlike builder(i.e. do piecewise)
PROTOTYPE
To create new object cheaply
SINGLETON
To create one and only on instance of class

Common Scala Design Patterns (ELI10 Style)

1. Singleton Pattern 🧍

What it means: Only one object exists for something.

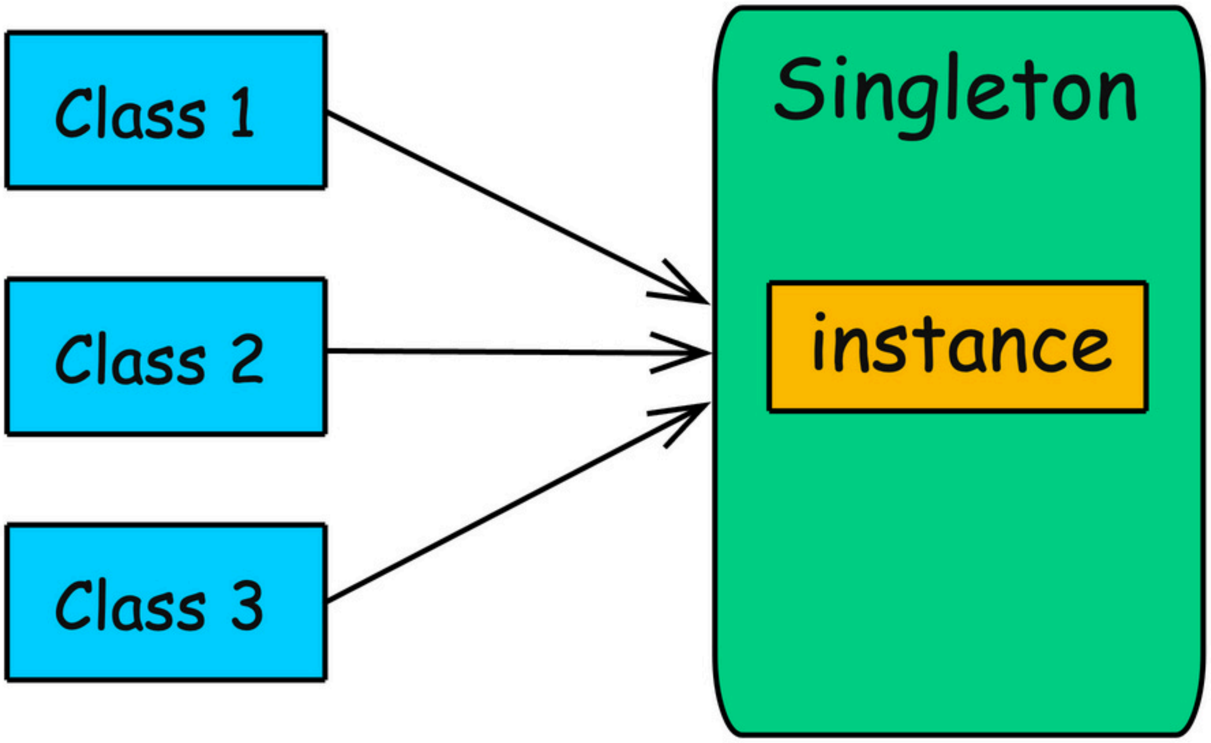
Real life: There's only **one principal** in your school.

Scala Example:

```
object School {
```

```scala
  def sayHello(): Unit = println("Hello from the only school!")
}

School.sayHello()
```

✅ **Use when:** You want a **single shared object**, like a config or logger.



---

2. Factory Pattern 🏭

What it means: A way to make objects **without telling the outside** how they're made.

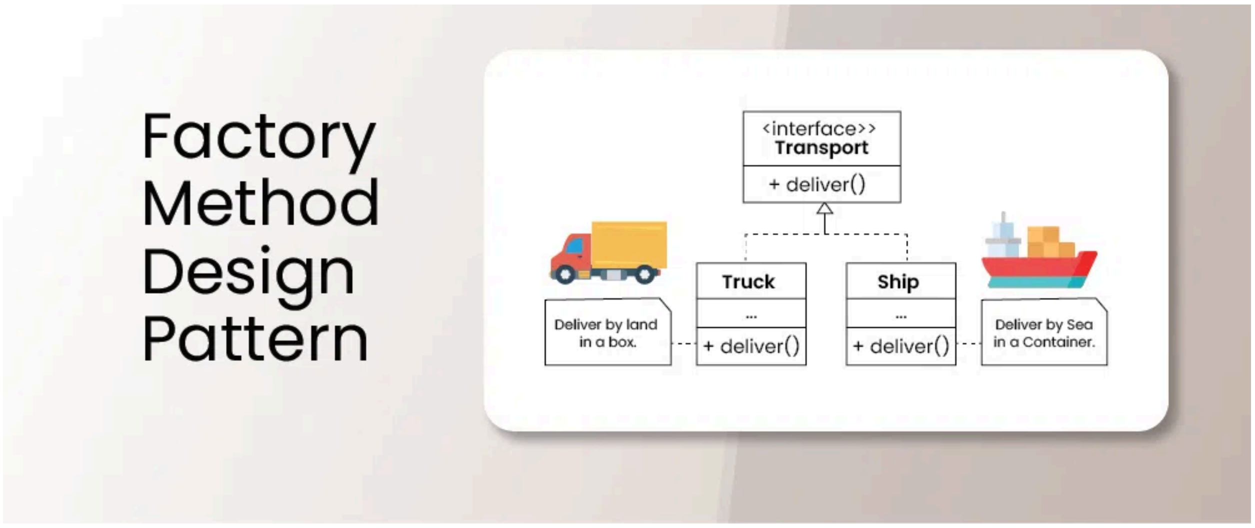
Real life: You order a toy online, and the factory sends it to you — you don't need to know how it was made.

Scala Example:

```scala
trait Animal {
  def speak(): String
}
```

```
class Dog extends Animal {
  def speak(): String = "Woof"
}

class Cat extends Animal {
  def speak(): String = "Meow"
}

object AnimalFactory {
  def getAnimal(animalType: String): Animal = {
    if (animalType == "dog") new Dog
    else new Cat
  }
}

val pet = AnimalFactory.getAnimal("dog")
println(pet.speak())
```

✅ **Use when:** You need to create objects based on logic.


Factory Method Design Pattern
<<interface>>
Transport
+ deliver()
Truck
...
+ deliver()
Ship
...
+ deliver()
Deliver by land in a box.
Deliver by Sea in a Container.

---

3. Builder Pattern

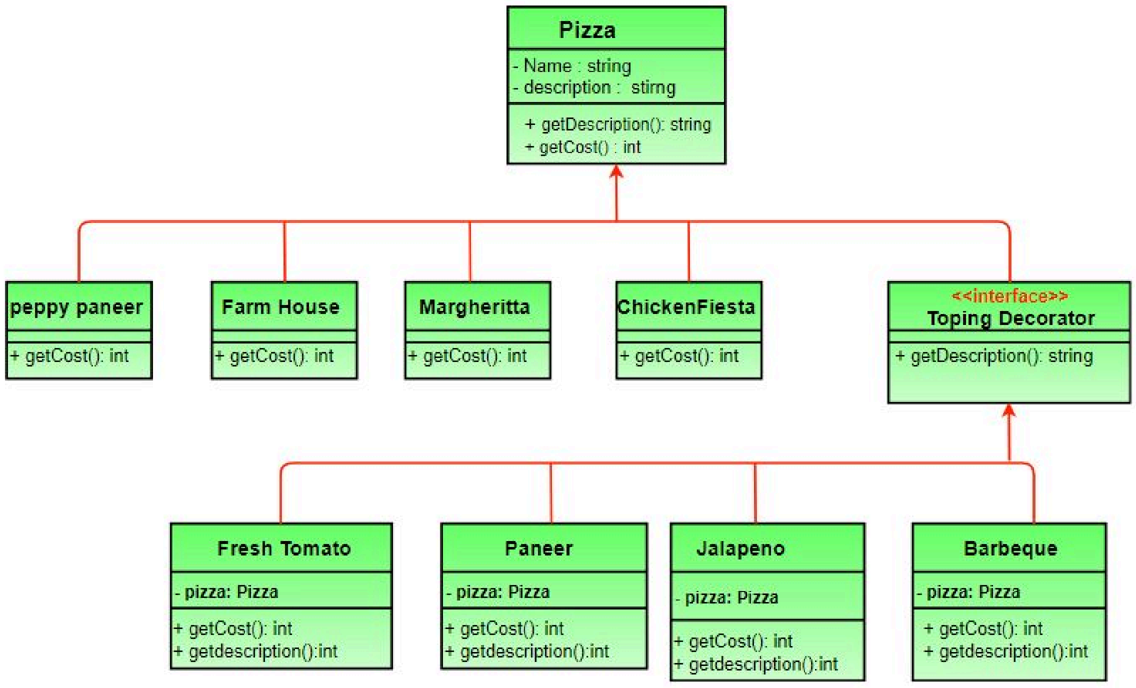
What it means: Build something **step-by-step** instead of all at once.

Real life: You build a Lego house piece by piece.

Scala Example:

```
case class Pizza(crust: String, cheese: Boolean, toppings:
List[String])

class PizzaBuilder {
  var crust = "Thin"
  var cheese = true
  var toppings = List[String]()

  def setCrust(c: String): this.type = { crust = c; this }
  def addTopping(t: String): this.type = { toppings = t :: toppings;
this }
  def noCheese(): this.type = { cheese = false; this }
  def build(): Pizza = Pizza(crust, cheese, toppings.reverse)
}

val pizza = new PizzaBuilder()
  .setCrust("Pan")
  .addTopping("Olives")
  .addTopping("Mushrooms")
  .noCheese()
  .build()

println(pizza)
```

✅ **Use when:** You want to build complex objects **one piece at a time**.



---

4. Strategy Pattern

What it means: Choose **how** to do something at runtime.

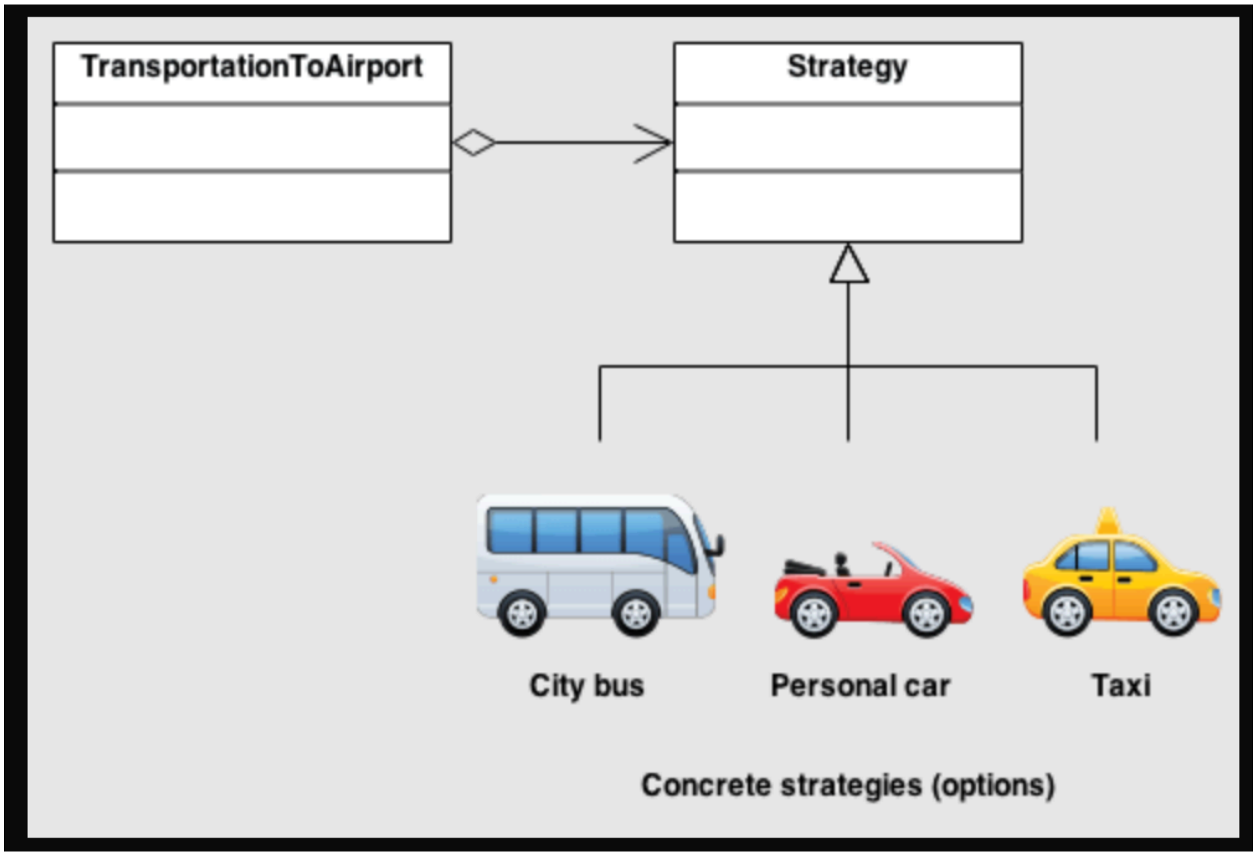
Real life: You can take a **bus**, **bike**, or **walk** to school — different **strategies** to get there.

Scala Example:

```
trait TravelStrategy {
  def travel(): String
}

class Bus extends TravelStrategy {
  def travel(): String = "Taking the bus"
}

class Bike extends TravelStrategy {
  def travel(): String = "Riding a bike"
```

```scala
}

class Traveler(strategy: TravelStrategy) {
  def startJourney(): Unit = println(strategy.travel())
}

val student = new Traveler(new Bike)
student.startJourney()
```

✅ **Use when:** You need to **change the behavior** without changing the whole class.



---

5. Decorator Pattern

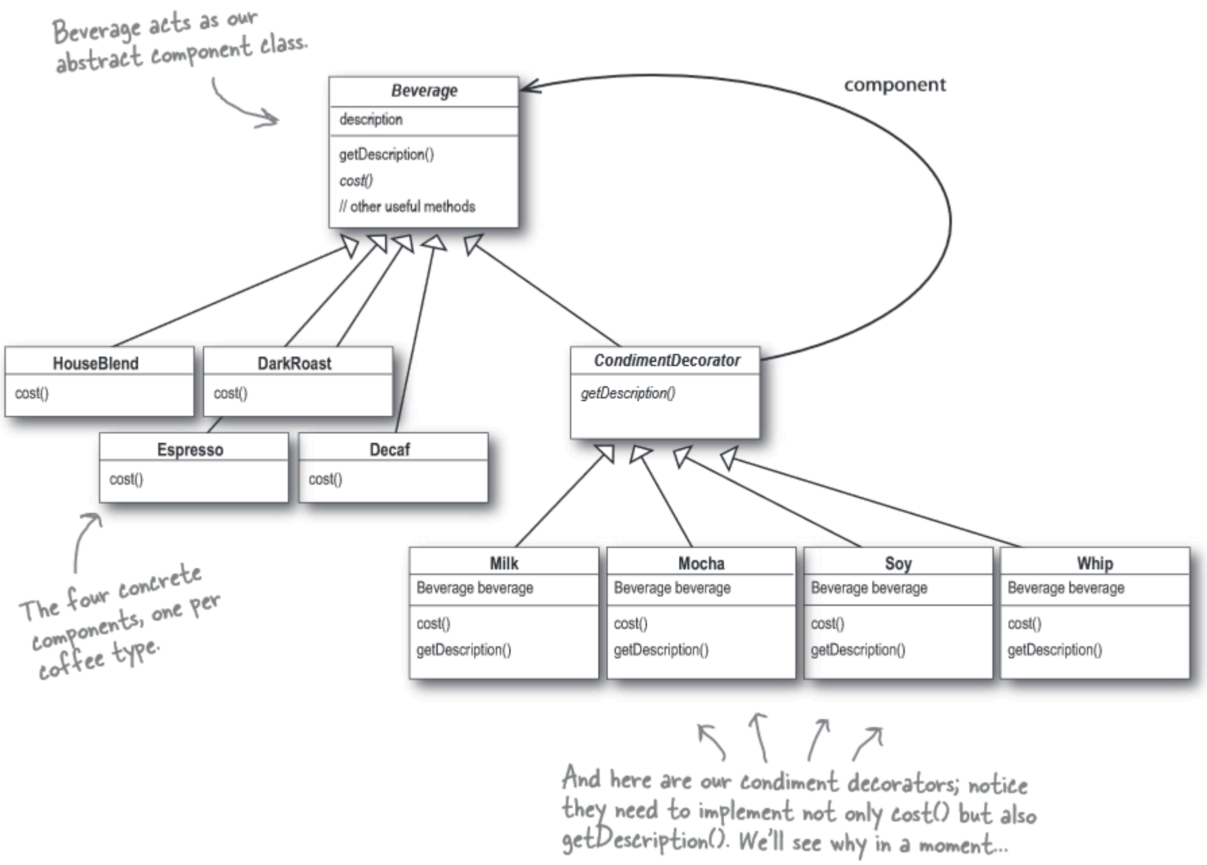
What it means: Add stuff to an object **without changing its original structure**.

Real life: You wear **decorations** (hats, glasses) but you're still the same person.

Scala Example:

```scala
trait Coffee {
  def cost: Double
}

class BasicCoffee extends Coffee {
  def cost: Double = 2.0
}

class MilkDecorator(coffee: Coffee) extends Coffee {
  def cost: Double = coffee.cost + 0.5
}

class SugarDecorator(coffee: Coffee) extends Coffee {
  def cost: Double = coffee.cost + 0.2
}

val coffee = new SugarDecorator(new MilkDecorator(new BasicCoffee()))
println(coffee.cost) // Output: 2.7
```

✅ **Use when:** You want to **extend behavior** without touching original code.



Summary Table

Pattern	Like...	Use it for...
Singleton	1 principal	Only one instance needed
Factory	Online toy order	Object creation logic hidden
Builder	Lego set	Build object in steps
Strategy	Travel choices	Choose behavior at runtime
Decorator	Add toppings	Add features to objects dynamically