Factory Method Design Pattern

Presented by Romerico David,
Aimable Mugwaneza and
Kehinde Osunniran

What is the Factory Method Pattern?

- A creational design pattern that provides an interface for object creation
- Allows subclasses to alter the type of objects being created
- Promotes loose coupling between client code and concrete classes
- Follows the "Program to an interface, not an implementation" principle

Key Properties of the Factory Method

01.

No Direct Creation

- Client doesn't use new keyword
- Objects come from factory

02.

Encapsulated Logic

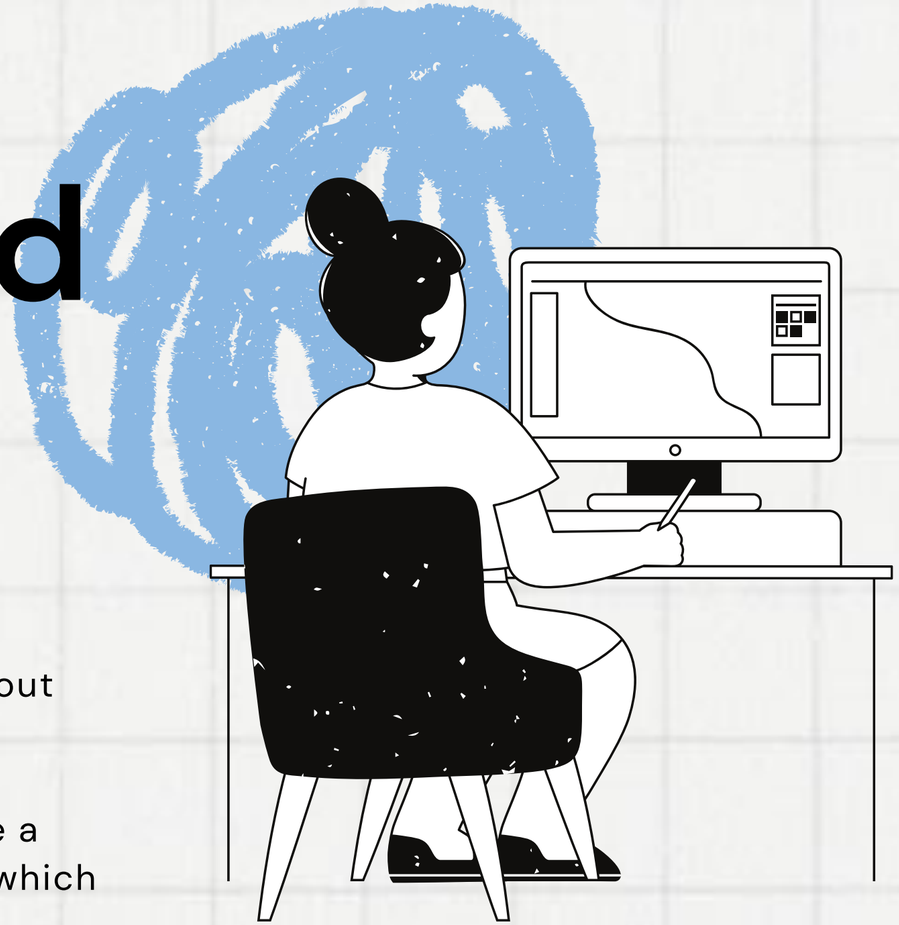
- All creation code in one place
- Easy to maintain and modify

03.

Extensible Design

- Add new types easily
- No changes to existing code

Motivation for Factory Method



Imagine running a logistics company that:

- Handles multiple types of transport
- Ships by land and sea
- Needs to be flexible for future expansion
- Wants clean, maintainable code

Challenge: How do we create different transport types without making our code messy?

The **Factory Method** pattern helps by allowing you to define a general method in a base class that lets subclasses decide which specific type of transport to create.

This way, your code can request a transport object without worrying about whether it’s a truck, ship, or any other type.

Real-World Example: Electricity Billing

The Problem:

- Different customer types
- Different billing rates
- Same calculation process
- Need flexible system

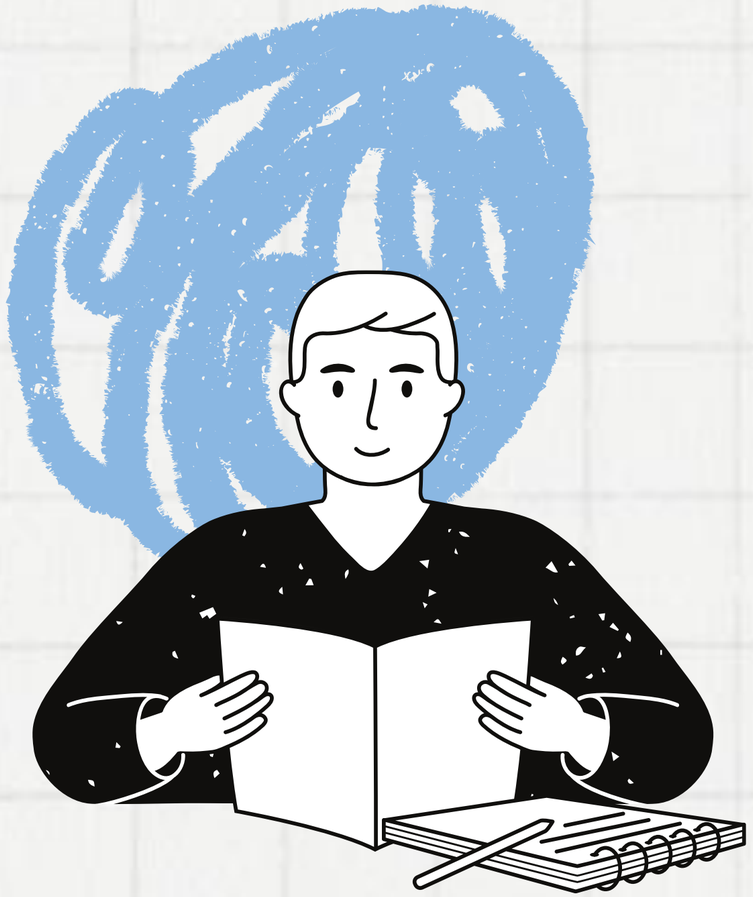
The Solution:

- Factory creates appropriate plan
- Each plan knows its rate
- Uniform calculation method

Concrete Implementations

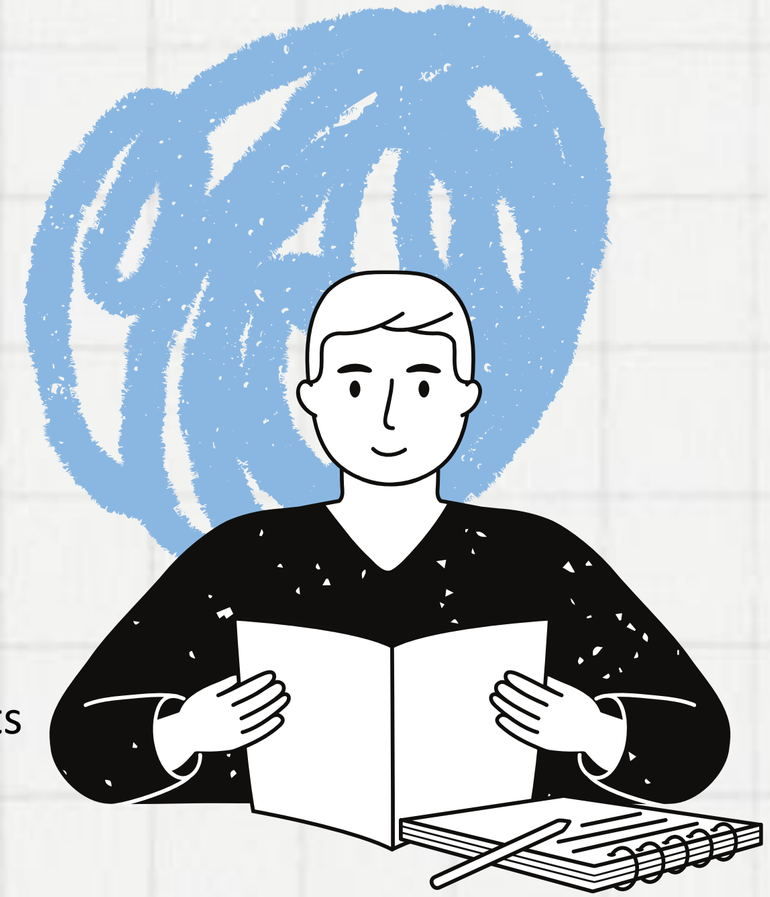
Different Types of Plans:

- Domestic Plan
  - Rate: 3.50 per unit
  - For residential customers
- Commercial Plan
  - Rate: 7.50 per unit
  - For business customers
- Institutional Plan
  - Rate: 5.50 per unit
  - For institutional customers



Key Components

- **Abstract Product (Plan)**
  - Defines the interface for objects created by the factory
  - In our example: Plan abstract class
- **Concrete Products**
  - Specific implementations of the abstract product
  - Examples: DomesticPlan, CommercialPlan, InstitutionalPlan
- **Factory (PlanFactory)**
  - Contains the factory method that creates and returns product objects
  - Centralizes object creation logic



1. Abstract Plan (Base Class)

```java
public abstract class Plan {
    protected double rate;
    public abstract void getRate();

    public void calculateBill(int units) {
        System.out.println("Bill amount: " + (units * rate));
    }
}

```

Plan is the interface that defines the operations that all objects the factory method creates must implement

2. Concrete Plans Implementations

```java
public class CommercialPlan extends Plan {
    @Override
    public void getRate() {
        rate = 7.50;
    }
}
```

```java
public class DomesticPlan extends Plan {
    @Override
    public void getRate() {
        rate = 3.50;
    }
}
```

```java
public class InstitutionalPlan extends Plan {
    @Override
    public void getRate() {
        rate = 5.50;
    }
}
```

These are the actual classes that define the behavior for each type of product the factory can create

The Factory Class

```java
public class PlanFactory {
    public Plan getPlan(String planType) {
        if (planType == null) {
            return null;
        }
        if (planType.equalsIgnoreCase(anotherString:"DOMESTIC")) {
            return new DomesticPlan();
        } else if (planType.equalsIgnoreCase(anotherString:"COMMERCIAL")) {
            return new CommercialPlan();
        } else if (planType.equalsIgnoreCase(anotherString:"INSTITUTIONAL")) {
            return new InstitutionalPlan();
        }
        return null;
    }
}
```

This is the class that defines and implements the factory method which returns an object of type Plan

```java
import java.util.Scanner;

public class GenerateBill {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        PlanFactory planFactory = new PlanFactory();
        try (Scanner scanner = new Scanner(System.in)) {
            System.out.print(s:"Enter the name of plan for which the bill will be generated: ");
            String planName = scanner.nextLine();

            System.out.print(s:"Enter the number of units for bill calculation: ");
            int units = scanner.nextInt();

            Plan plan = planFactory.getPlan(planName);
            if (plan != null) {
                plan.getRate();
                plan.calculateBill(units);
            } else {
                System.out.println(x:"Invalid plan type.");
            }
        }
    }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

TERMINAL

```
(base) romericodavid@Romericos-Air factory-method-presentation %  cd /Users/romericodavid/repos/Object-Orie
sign-and-Programming/factory-method-presentation ; /usr/bin/env /Library/Java/JavaVirtualMachines/temurin-1
ontents/Home/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/romericodavid/Library/Application\
t/Code/User/workspaceStorage/b058a6c001f74e111866805c0e3c1138/redhat.java/jdt_ws/factory-method-presentatio
e3a/bin GenerateBill
Enter the name of plan for which the bill will be generated: DOMESTIC
Enter the number of units for bill calculation: 20
Bill amount: 70.0
(base) romericodavid@Romericos-Air factory-method-presentation %
```

Advantages

01.

Loose Coupling

- Clients are decoupled from concrete classes.
- Easy to modify plans

02.

Open/Closed Principle

- Easy to introduce new plan types.

03.

Reusability

- Common creation logic is centralized reducing duplication

Disadvantages

01.

Complexity

- More classes to manage
- More code to maintain

02.

Subclassing Required

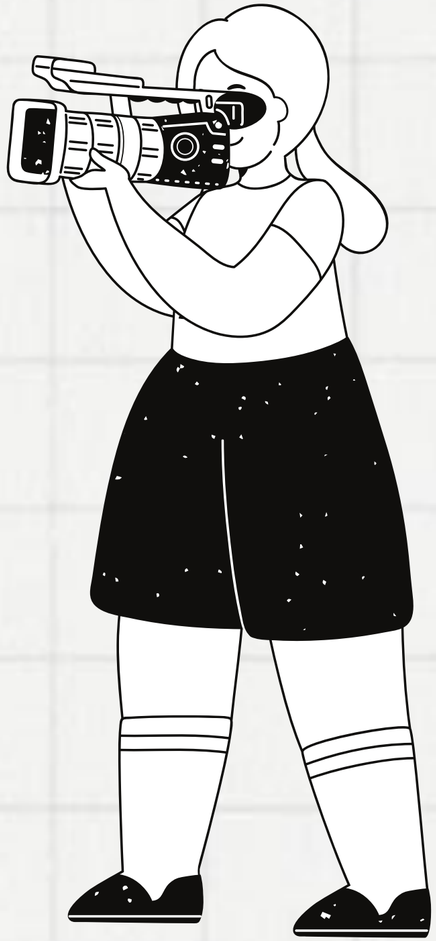
- Must extend base class
- Inheritance hierarchy

When to use:

- Objects need flexible creation
- Common interface, different implementations
- Future types might be added
- Creation logic should be centralized

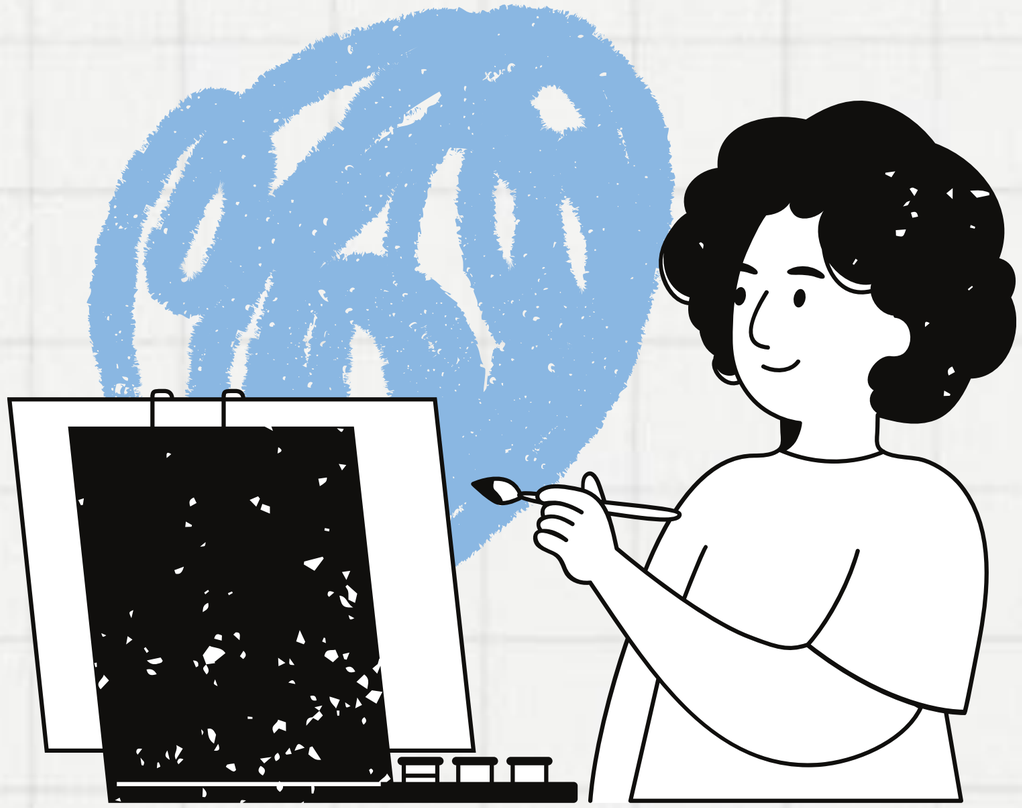
When not to use:

- Simple object creation
- Fixed number of types
- No variation in creation



Summary

- Factory Method creates objects indirectly
- Centralizes object creation logic
- Makes system flexible and maintainable
- Perfect for varying object types
- Used in our billing system successfully



References

- https://www.geeksforgeeks.org/factory-method-for-designing-pattern/
- https://en.wikipedia.org/wiki/Factory_method_pattern
- https://sourcemaking.com/design_patterns/factory_method

Thank you very much!