

# **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

## No Direct Creation

01.

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

## Encapsulated Logic

02.

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

## Extensible Design

03.

- 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



# 1. Abstract Plan (Base Class)

```
1 public abstract class Plan {  
2     protected double rate;  
3     public abstract void getRate();  
4  
5     public void calculateBill(int units) {  
6         System.out.println("Bill amount: " + (units * rate));  
7     }  
8 }  
9 |
```

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

## 2. Concrete Plans Implementations

```
1 public class CommercialPlan extends Plan {
2     @Override
3     public void getRate() {
4         rate = 7.50;
5     }
6 }
```

```
1 public class DomesticPlan extends Plan {
2     @Override
3     public void getRate() {
4         rate = 3.50;
5     }
6 }
```

```
1 public class InstitutionalPlan extends Plan {
2     @Override
3     public void getRate() {
4         rate = 5.50;
5     }
6 }
```

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



# The Factory Class

```
1  public class PlanFactory {
2      public Plan getPlan(String planType) {
3          if (planType == null) {
4              return null;
5          }
6          if (planType.equalsIgnoreCase(anotherString:"DOMESTIC")) {
7              return new DomesticPlan();
8          } else if (planType.equalsIgnoreCase(anotherString:"COMMERCIAL")) {
9              return new CommercialPlan();
10         } else if (planType.equalsIgnoreCase(anotherString:"INSTITUTIONAL")) {
11             return new InstitutionalPlan();
12         }
13         return null;
14     }
15 }
16
17
```

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

```

1  import java.util.Scanner;
2
3  public class GenerateBill {
4      Run | Debug | Run main | Debug main
5      public static void main(String[] args) {
6          PlanFactory planFactory = new PlanFactory();
7          try (Scanner scanner = new Scanner(System.in)) {
8              System.out.print(s:"Enter the name of plan for which the bill will be generated: ");
9              String planName = scanner.nextLine();
10
11              System.out.print(s:"Enter the number of units for bill calculation: ");
12              int units = scanner.nextInt();
13
14              Plan plan = planFactory.getPlan(planName);
15              if (plan != null) {
16                  plan.getRate();
17                  plan.calculateBill(units);
18              } else {
19                  System.out.println(x:"Invalid plan type.");
20              }
21          }
22      }
23

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

#### TERMINAL

```

(base) romericodavid@Romericos-Air factory-method-presentation % cd /Users/romericodavid/repos/Object-Oriented-Design-and-Programming/factory-method-presentation ; /usr/bin/env /Library/Java/JavaVirtualMachines/temurin-11.jdk-9/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/romericodavid/Library/Application\Support/Code/User/workspaceStorage/b058a6c001f74e111866805c0e3c1138/redhat.java/jdt_ws/factory-method-presentation/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.

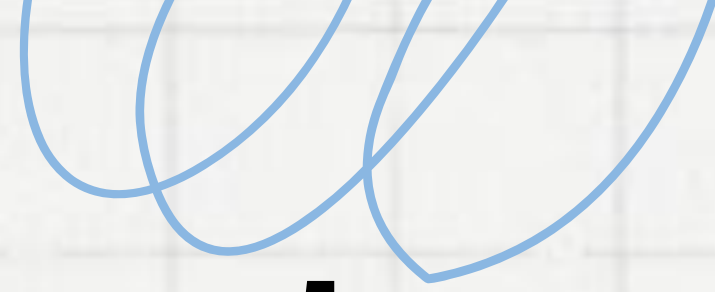
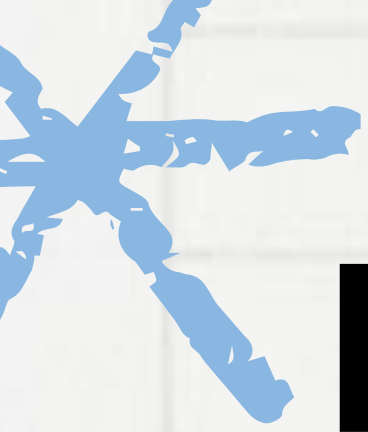
## **Open/Closed Principle**

- 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://en.wikipedia.org/wiki/Factory_method_pattern)
- [https://sourcemaking.com/design\\_patterns/factory\\_method](https://sourcemaking.com/design_patterns/factory_method)

The background is a light blue grid. It is decorated with various hand-drawn blue doodles. In the top left, there are several overlapping circles and loops. In the top center, there is a thick, textured blue scribble. In the top right, there are more overlapping circles and a star-like shape. On the right side, there are horizontal lines and a circular scribble. In the bottom left, there are concentric arcs and a textured scribble. In the bottom center, there is a wavy line and a series of small 'v' marks. In the bottom right, there is a large, flowing loop.

**Thank you  
very much!**