

TP 06

Salma Bellaou

October 29, 2025

1 Exercise 1: Singleton Pattern

1.1 Objective

Create a database class that allows only one instance to exist.

1.2 Singleton Class Code

```
class Singleton { 7 usages
    String name; 2 usages
    private static Singleton instance; 4 usages
    private Singleton(String name){ 1 usage
        this.name = name;
    }

    public static Singleton getConnection(String name){ 2 usages
        if (instance == null){
            instance = new Singleton (name);
            System.out.println("You are connected to the database "+ instance.name);}

        return instance;
    }
}
```

Figure 1: The private constructor and static instance ensure only one object is created.

1.3 Test Class Code

```
public class TestSingleton{  
    public static void main(String[] args){  
        Singleton s1 = Singleton.getConnection( name: "Java");  
        Singleton s2 = Singleton.getConnection( name: "Python");  
        System.out.println("s1 == s2: " + (s1 == s2));  
    }  
}
```

Figure 2: Two connection attempts return the same instance, proving Singleton works.

1.4 Output

```
You are connected to the database Java  
s1 == s2: true  
  
Process finished with exit code 0
```

Figure 3: Only the first connection message appears, and s1 equals s2.

2 Exercise 2: Factory Pattern

2.1 Part 1: Initial Design Without Factory

2.1.1 Program Interface

```
import java.util.*;
public interface Program{ 3 usages 3 implementations
    void go(); 3 usages 3 implementations
}
```

Figure 4: Simple interface with one method that all programs must implement.

2.1.2 Program Classes

```
public class Program1 implements Program{ 2 usages
    public Program1(){ 1 usage
    }
    public void go (){ 3 usages
        System . out . println ( " Je suis le traitement 1 " ) ;
    }
}
```

Figure 5: program 1.

```
public class Program2 implements Program{ 2 usages
    public Program2(){ 1 usage
    }
    public void go (){ 3 usages
        System . out . println ( " Je suis le traitement 2 " ) ;
    }
}
```

Figure 6: program 2.

```

public class Program3 implements Program{ 2 usages
    public Program3(){ 1 usage
    }
    public void go (){ 3 usages
        System . out . println ( " Je suis le traitement 3 " ) ;
    }
}

```

Figure 7: program 3

2.1.3 Client Without Factory

```

import java.util.Scanner;

public class Client {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of the program (1-3):");
        int n = sc.nextInt();

        switch(n) {
            case 1:
                main1();
                break;
            case 2:
                main2();
                break;
            case 3:
                main3();
                break;
            default:
                System.out.println("Invalid option");
        }

        sc.close();
    }
}

```

Figure 8: Duplicated code in main1, main2, and main3 methods is bad practice.

```
import java.util.Scanner;

public class Client {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of the program (1-3):");
        int n = sc.nextInt();

        switch(n) {
            case 1:
                main1();
                break;
            case 2:
                main2();
                break;
            case 3:
                main3();
                break;
            default:
                System.out.println("Invalid option");
        }

        sc.close();
    }
}
```

Figure 9: Duplicated code in main1, main2, and main3 methods is bad practice.

```

    public static void main1() { 1 usage
        Program1 p1 = new Program1();
        System.out.println("I am main 1");
        p1.go();
    }

    public static void main2() { 1 usage
        Program2 p2 = new Program2();
        System.out.println("I am main 2");
        p2.go();
    }

    public static void main3() { 1 usage
        Program3 p3 = new Program3();
        System.out.println("I am main 3");
        p3.go();
    }
}

```

Figure 10: main functions.

```

Enter the number of the program (1-3):
1
I am main 1
Je suis le traitement 1

Process finished with exit code 0

```

Figure 11: output

2.2 Part 2: Solution With Factory Pattern

2.2.1 ProgramFactory Class

```
public class ProgramFactory{ 2 usages
    public Program ProgramNumber(int n){ 1 usage
        if(n==1){
            return new Program1();
        }
        if(n==2){
            return (new Program2());
        }
        if (n==3){
            return new Program3();
        }
        else{
            return null;
        }
    }
}
```

Figure 12: Factory centralizes object creation logic in one place.

2.2.2 Refactored Client

```
import java.util.Scanner;

public class Client {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of the program (1-3):");
        int n = sc.nextInt();
        ProgramFactory factory = new ProgramFactory();
        Program program = factory.ProgramNumber(n);

        if (program != null) {
            program.go();
        } else {
            System.out.println("Invalid option");
        }
    }
}
```

Figure 13: Client now uses the factory, eliminating duplicate code completely.

2.2.3 Class Diagram

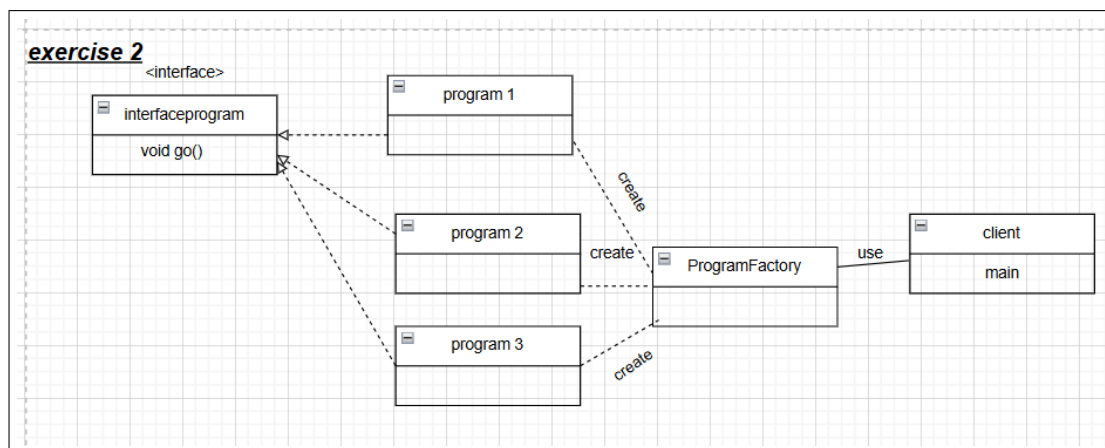


Figure 14: Factory pattern decouples Client from concrete Program implementations.

2.2.4 Outputs

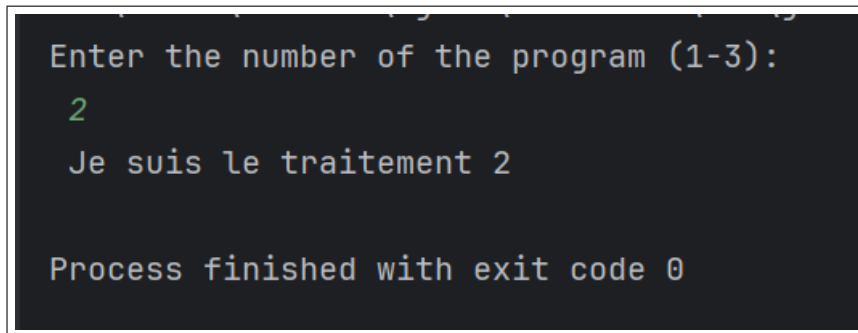
A terminal window with a dark background and light-colored text. The text shows a prompt 'Enter the number of the program (1-3):', followed by the input '2' in green, then the output 'Je suis le traitement 2', and finally 'Process finished with exit code 0'.

Figure 15: Program 2 executes successfully through the factory.

2.3 Part 3: Adding Program4

2.3.1 Program4 Class

```
public class Program4 implements Program{ 1 usage
    public Program4(){ 1 usage
    }
    public void go (){ 1 usage
        System . out . println ( " Je suis le traitement 4 " ) ;
    }
}
```

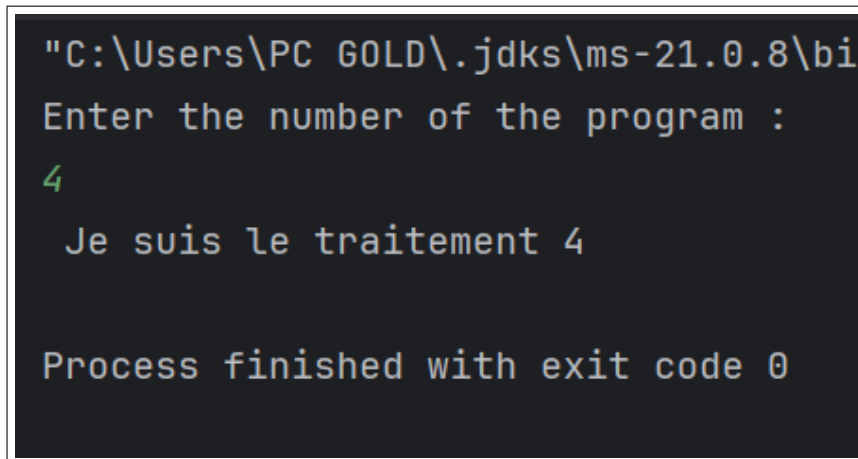
Figure 16: New Program4 class implements the same interface as others.

2.3.2 Updated Factory

```
public class ProgramFactory{ 2 usages
    public Program ProgramNumber(int n){ 1 usage
        if(n==1){
            return new Program1();
        }
        if(n==2){
            return (new Program2());
        }
        if (n==3){
            return new Program3();
        }
        if (n==4){
            return new Program4();
        }
        else{
            return null;
        }
    }
}
```

Figure 17: Only the factory needs modification; Client remains unchanged.

2.3.3 Program4 Output



```
"C:\Users\PC GOLD\.jdk\ms-21.0.8\bin
Enter the number of the program :
4
Je suis le traitement 4
Process finished with exit code 0
```

Figure 18: Program 4 runs without requiring any changes to Client class.

3 Comparison

The factory pattern eliminates code duplication and makes adding new programs easy. Client code decreased from 43 lines to 15 lines with better maintainability. Adding Program4 required zero changes to Client, only two line in Factory.

4 Conclusion

Singleton pattern ensures only one instance exists, useful for database connections. Factory pattern centralizes object creation, reducing coupling and duplication. Both patterns improve code quality and maintainability significantly.