**Exercise 2: Implementing the Factory Method Pattern**

import java.util.\*;

class Factory{

public static void main(String args[]){

Scanner sc = new Scanner(System.in);

System.out.print("Enter the type of OS you want: ");

String str = sc.next();

FactoryOS fos = new FactoryOS();

OS obj = fos.getInstance(str);

obj.spec();

}

}

public interface OS{

void spec();

}

public class Android implements OS{

public void spec(){

System.out.println("Android : Most powerfull OS.");

}

}

public class IOS implements OS{

public void spec(){

System.out.println("IOS : Most secured OS.");

}

}

public class Windows implements OS{

public void spec(){

System.out.println("Windows : Default OS.");

}

}

public class FactoryOS{

public OS getInstance(String str){

if(str.equals("Open")){

return new Android();

}else if(str.equals("Closed")){

return new IOS();

}else{

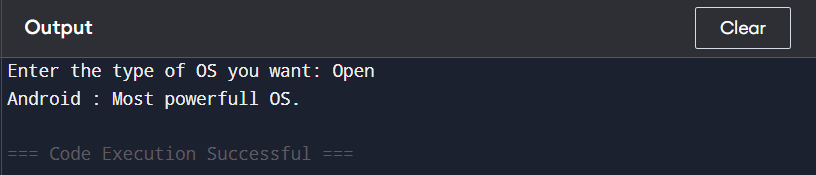
return new Windows();

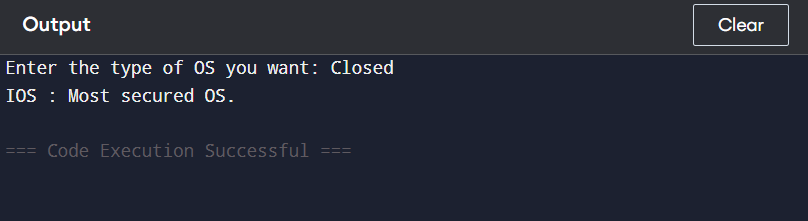
}

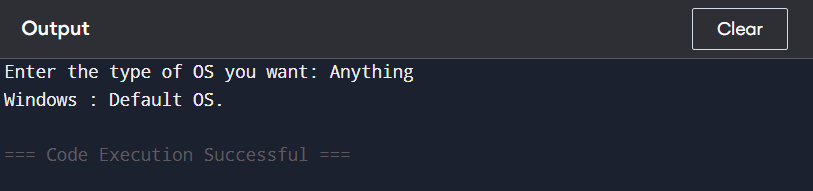
}

}

**OUTPUT:**







* The **client (main) in Factory class asks for an OS type** by user input.
* The FactoryOS class contains a method getInstance(String str) that:

Returns an object of Android if input is "Open"

Returns an object of IOS if input is "Closed"

Otherwise, returns a default object of Windows

* All OS types implement the same **OS interface** with the method spec().
* The **actual object creation logic is hidden inside FactoryOS**, keeping the main class **clean and decoupled** from the concrete implementations.

If you introduce a **new OS class (e.g., Linux) in the future**, you **only modify the FactoryOS class**, **not the client (main) code**.  
This follows the **Open/Closed Principle** (open for extension, closed for modification) — a core OOP design principle.