**LAB04 – Design Pattern**

# Learning Outcomes

1. Encapsulate object creation using a factory.
2. Apply Open/Closed Principle.
3. Use Factory Method to decouple clients from concrete classes.

# Task 1 – Simple Factory: Notifications (INCOMPLETE)

Objective: Replace direct 'new' usage in client code with a single factory. Implement the factory and refactor the client.

What to do:

* Implement NotificationFactory.Create(string type) (TODO).
* Refactor Program\_Main\_Before to a Program\_Main\_After that calls the factory (TODO).
* Ensure no client code outside the factory calls 'new' for concrete notifications.

## Starter Interfaces and Concrete Types

// File: INotification.cs public interface INotification { void Send(string to, string message);

}

// File: EmailNotification.cs public class EmailNotification : INotification { public void Send(string to, string message) {

System.Console.WriteLine($"[EMAIL] to {to}: {message}");

}

}

// File: SmsNotification.cs public class SmsNotification : INotification { public void Send(string to, string message) {

System.Console.WriteLine($"[SMS] to {to}: {message}");

}

}

// File: PushNotification.cs public class PushNotification : INotification { public void Send(string to, string message) {

System.Console.WriteLine($"[PUSH] to {to}: {message}");

}

}

## Factory (INCOMPLETE)

// File: NotificationFactory.cs public class NotificationFactory {

// TODO: Implement this method to return a concrete INotification

// based on 'type' ("email" | "sms" | "push").

// - Throw ArgumentException on unknown type or null/empty. public INotification Create(string type) {

// TODO BEGIN throw new System.NotImplementedException("Create() not implemented");

// TODO END

}

}

## Client BEFORE (to be REFACTORED)

// File: Program\_Main\_Before.cs

// TODO: This is the starting point that uses 'new' directly.

// Your job: create a new Program\_Main\_After that uses NotificationFactory instead.

// Acceptance: No 'new

EmailNotification()/SmsNotification()/PushNotification()' in client code.

class Program\_Before { static void Main() {

// DIRECT INSTANTIATION (to be refactored)

INotification n1 = new EmailNotification();

INotification n2 = new SmsNotification();

INotification n3 = new PushNotification();

n1.Send("alice@example.com", "Welcome!"); n2.Send("+84123456789", "OTP: 123456"); n3.Send("device:abc", "You've got a badge!"); }

}

## Client AFTER (create this) – uses Factory

// File: Program\_Main\_After.cs

// TODO: Implement this as your refactored client. Example shape (fill in and adjust as needed).

// class Program {

// static void Main() {

// var factory = new NotificationFactory();

// var n1 = factory.Create("email");

// var n2 = factory.Create("sms");

// var n3 = factory.Create("push");

//

// n1.Send("alice@example.com", "Welcome!");

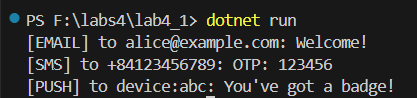
// n2.Send("+84123456789", "OTP: 123456");

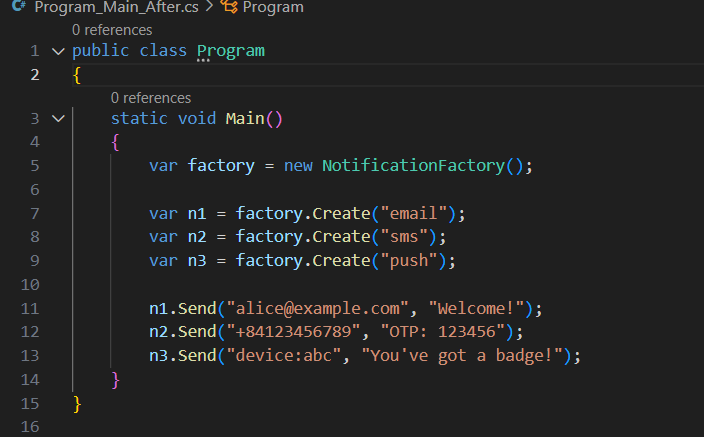
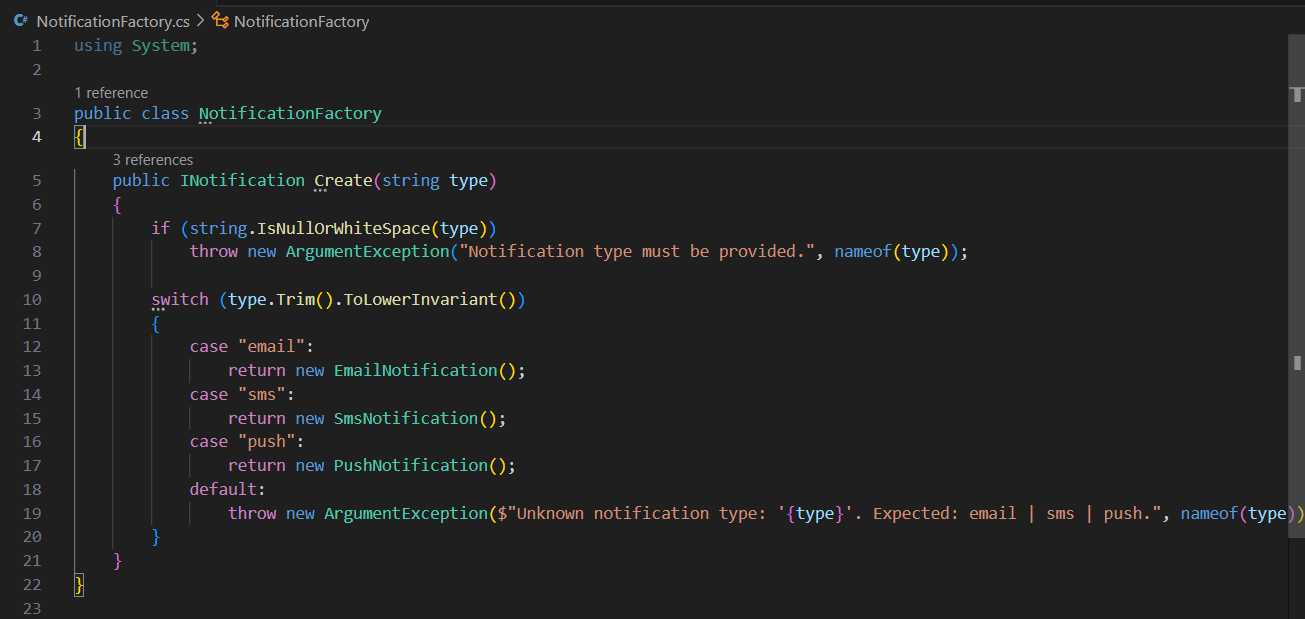
// n3.Send("device:abc", "You've got a badge!"); // }

// }

## Task 1 – Acceptance

* Program\_Main\_After compiles and runs, printing three lines for EMAIL/SMS/PUSH.



* No direct 'new' of concrete notification classes in the client.   
  
* All creation logic lives inside NotificationFactory.Create.   
  

# Task 2 – Factory Method: Report Exporters (INCOMPLETE)

Objective: Implement the Factory Method where subclasses (creators) decide which concrete product to instantiate.

## Starter Contracts (INCOMPLETE PARTS MARKED TODO)

// File: IReportExporter.cs public interface IReportExporter { string Export(string[][] rows);

}

// File: CsvExporter.cs using System.Linq;

public class CsvExporter : IReportExporter { public string Export(string[][] rows) {

// Provided

return string.Join("\n", rows.Select(r => string.Join(",", r)));

}

}

// File: JsonExporter.cs using System.Linq; public class JsonExporter : IReportExporter { public string Export(string[][] rows) { // Provided (simple JSON)

var objects = rows.Select(r => "{" + string.Join(",", r.Select((v, i)

=> $"\"c{i}\":\"{v}\"")) + "}");

return "[" + string.Join(",", objects) + "]";

}

}

// File: ReportExporterCreator.cs

public abstract class ReportExporterCreator { // TODO: Implement Factory Method in subclasses protected abstract IReportExporter CreateExporter();

// Template operation that uses the product public string ExportRows(string[][] rows) { var exporter = CreateExporter(); // call factory method return exporter.Export(rows);

}

}

// File: CsvExporterCreator.cs public class CsvExporterCreator : ReportExporterCreator { protected override IReportExporter CreateExporter() {

// TODO: return CSV concrete product throw new

System.NotImplementedException("CsvExporterCreator.CreateExporter not implemented");

}

}

// File: JsonExporterCreator.cs public class JsonExporterCreator : ReportExporterCreator { protected override IReportExporter CreateExporter() {

// TODO: return JSON concrete product throw new

System.NotImplementedException("JsonExporterCreator.CreateExporter not implemented");

}

}

// File: Program\_FactoryMethod\_Demo.cs public class Program\_FactoryMethod\_Demo { public static void Main() { string[][] rows = new string[][]{ new []{"id","name"}, new []{"1","Alice"}, new []{"2","Bob"}

};

// TODO: Replace nulls with real creators

ReportExporterCreator csvCreator = /\* new CsvExporterCreator() \*/ null;

ReportExporterCreator jsonCreator = /\* new JsonExporterCreator() \*/ null;

// TODO: Uncomment after implementing creators

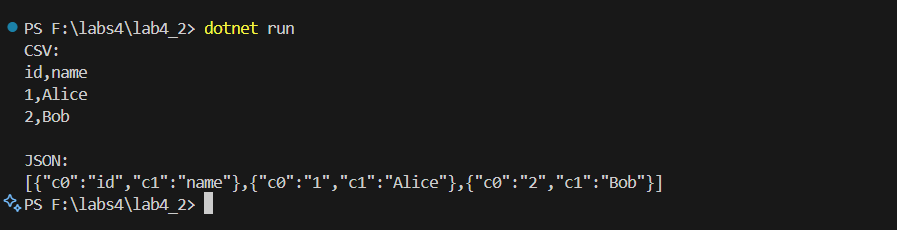
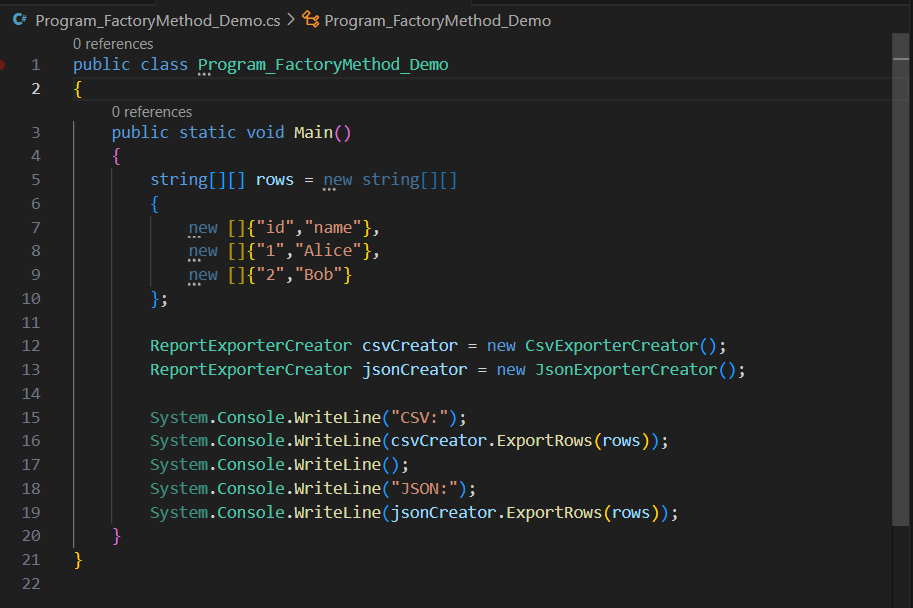
// System.Console.WriteLine("CSV:\n" + csvCreator.ExportRows(rows));

// System.Console.WriteLine("JSON:\n" + jsonCreator.ExportRows(rows));

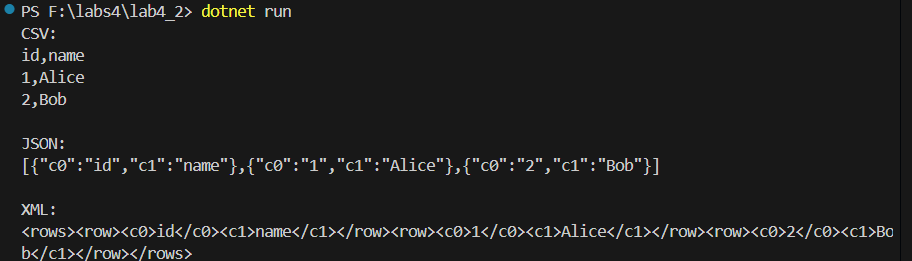
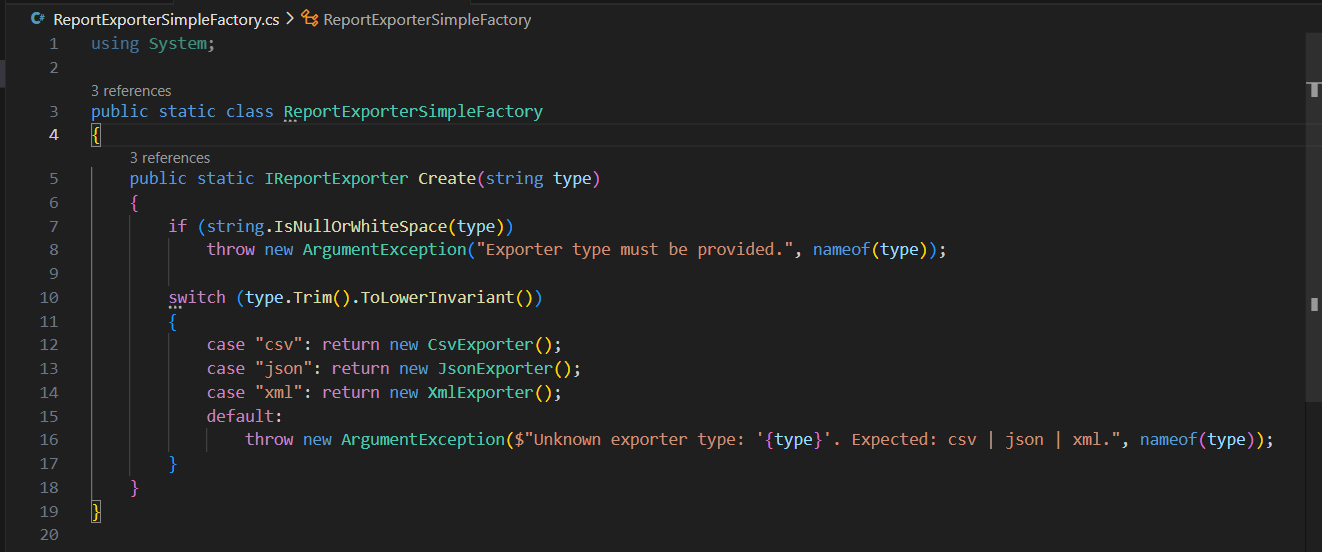
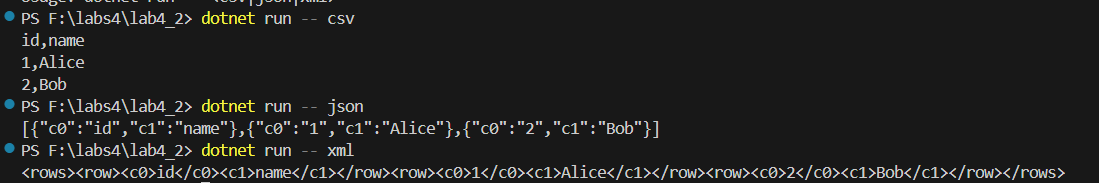
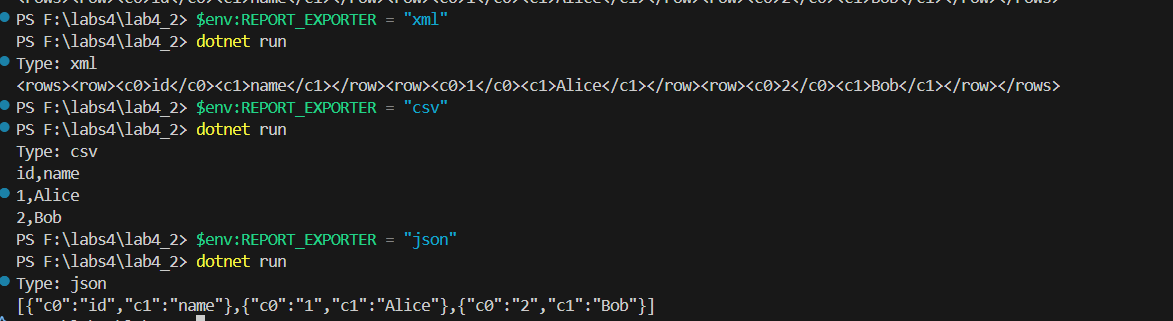
}

}

## Task 2 – Acceptance

* Program\_FactoryMethod\_Demo prints CSV and JSON outputs.   
  
* Client only references creators, not concrete products.   
  
* No 'new CsvExporter()' or 'new JsonExporter()' in client code.

# Task 3 – Extensions

1. Add XmlExporter + XmlExporterCreator and demonstrate exporting XML by adding only a creator selection line in demo.   
   
2. Build SimpleFactory: ReportExporterSimpleFactory.Create(type) returning Csv/Json/Xml. Use it in a small CLI (args).   
    
3. Config-driven type selection (env var or appsettings.json). No client code change, only config change.   
   
4. Robust error handling (unknown type -> exception message). Add a minimal test or console proof.   
   
5. Tiny performance test over 10k rows and print timings.   
   