# Learning Outcomes

1. Explain the purpose and structure of the **Abstract Factory Pattern**.
2. Implement families of related products (e.g., toy brands) without using concrete class names.
3. Apply **Open/Closed** and **Dependency Inversion** principles.
4. Extend a product family without changing client code.
5. Relate **Abstract Factory** to **Factory Method**.

# Story

You are designing a **Toy Factory Management System**.

There are two brands of toys: **FunKid** and **PlayTime**.

Each brand can make a **Car** and a **Doll**.

Your job is to implement a factory system where the client can switch between brands without changing its code.

## Program.cs

using System;

class Program

{ static void Main()

{

Console.Write("Enter toy brand (funkid/playtime): "); string brand = Console.ReadLine()?.Trim().ToLower();

IToyFactory factory = brand switch

{

"playtime" => new PlayTimeFactory(),

\_ => new FunKidFactory()

};

var store = new ToyStore(factory); store.ShowToys();

}

}

# Interfaces

## IToyFactory.cs

public interface IToyFactory

{

ICar CreateCar();

IDoll CreateDoll();

}

## ICar.cs

public interface ICar

{ void Play();

}

## IDoll.cs

public interface IDoll

{ void Play();

}

# Concrete Products

## FunKidCar.cs

using System; public class FunKidCar : ICar

{

public void Play() => Console.WriteLine("FunKid Car: Zoom zoom!"); }

## FunKidDoll.cs

using System; public class FunKidDoll : IDoll

{

public void Play() => Console.WriteLine("FunKid Doll: Hello! Let's play dress up!");

}

## PlayTimeCar.cs

using System; public class PlayTimeCar : ICar

{

public void Play() => Console.WriteLine("PlayTime Car: Vroom vroom!"); }

## PlayTimeDoll.cs

using System; public class PlayTimeDoll : IDoll

{

public void Play() => Console.WriteLine("PlayTime Doll: Hi there! Let's go shopping!");

}

# Concrete Factories

## FunKidFactory.cs

public class FunKidFactory : IToyFactory

{ public ICar CreateCar() => new FunKidCar(); public IDoll CreateDoll() => new FunKidDoll();

}

## PlayTimeFactory.cs

public class PlayTimeFactory : IToyFactory

{ public ICar CreateCar() => new PlayTimeCar(); public IDoll CreateDoll() => new PlayTimeDoll(); }

# Client

## ToyStore.cs

public class ToyStore

{ private readonly ICar car; private readonly IDoll doll;

public ToyStore(IToyFactory factory)

{

car = factory.CreateCar(); doll = factory.CreateDoll();

} public void ShowToys()

{ car.Play(); doll.Play();

}

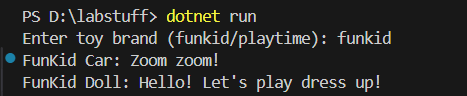
}

# Expected Output

**If user enters funkid:**

FunKid Car: Zoom zoom!

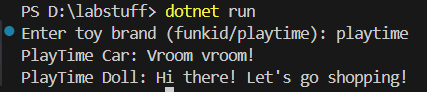
FunKid Doll: Hello! Let's play dress up!



**If user enters playtime:**

PlayTime Car: Vroom vroom!

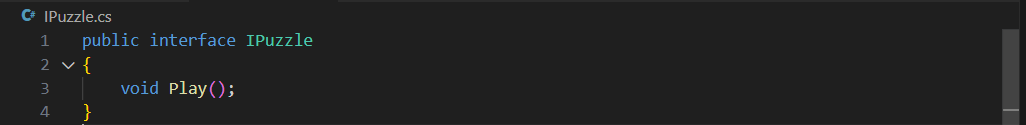
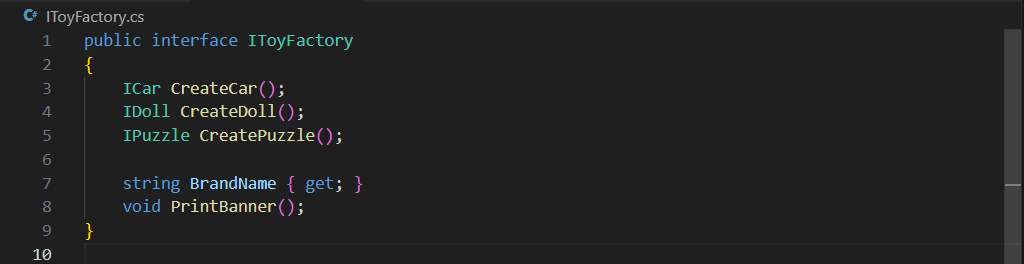
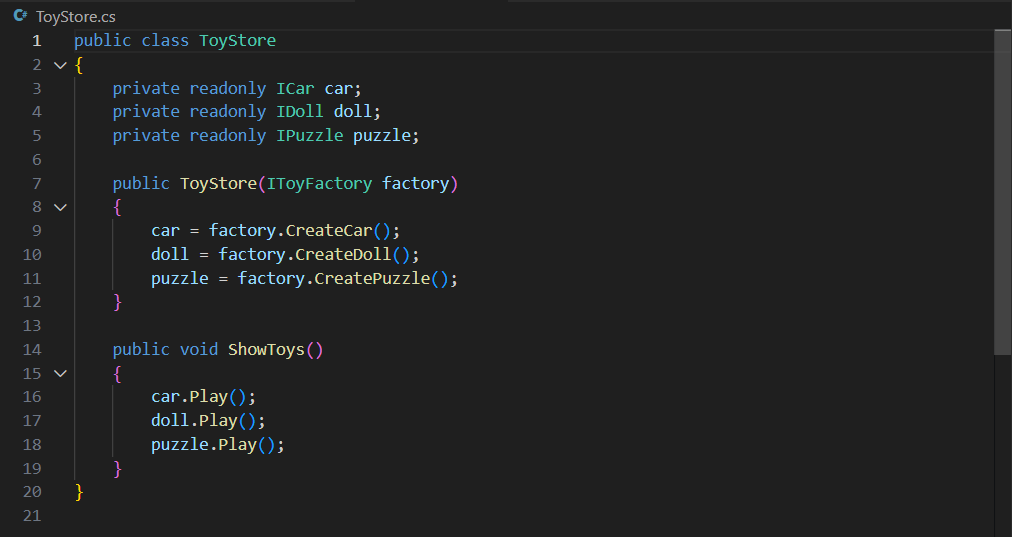
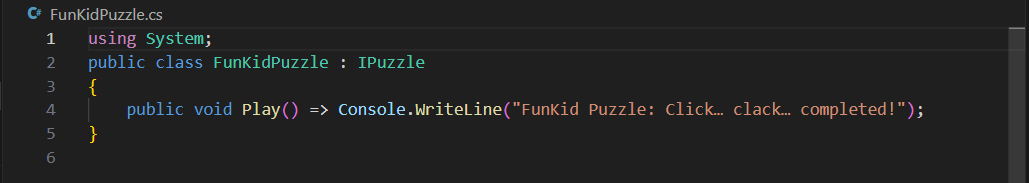
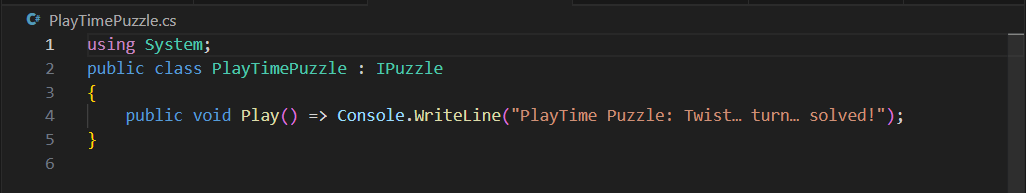
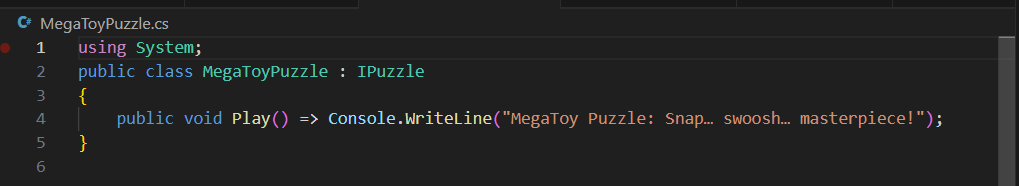
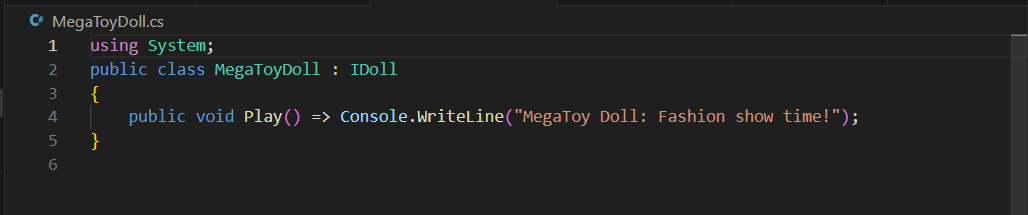
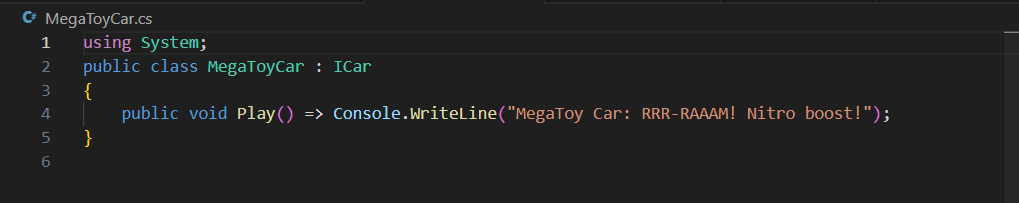
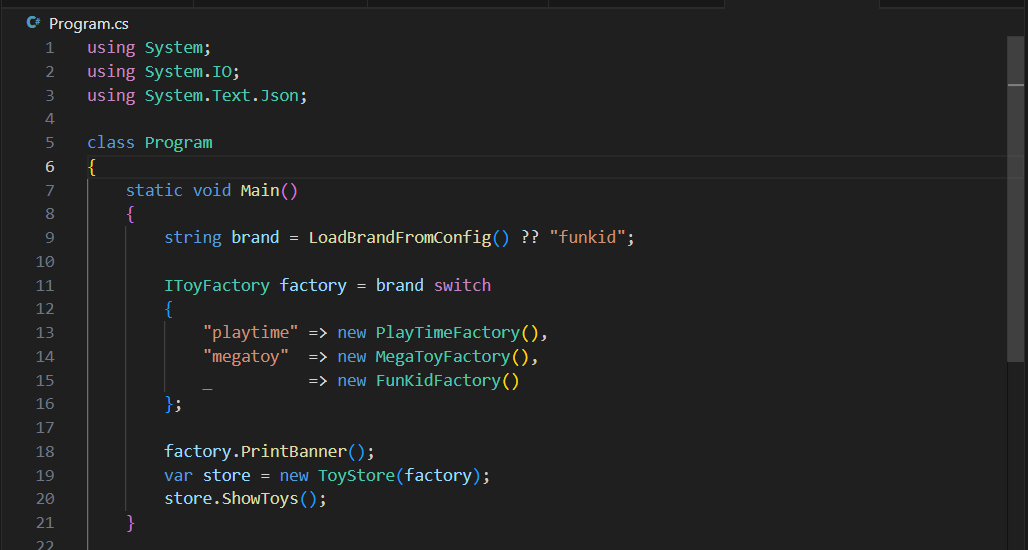
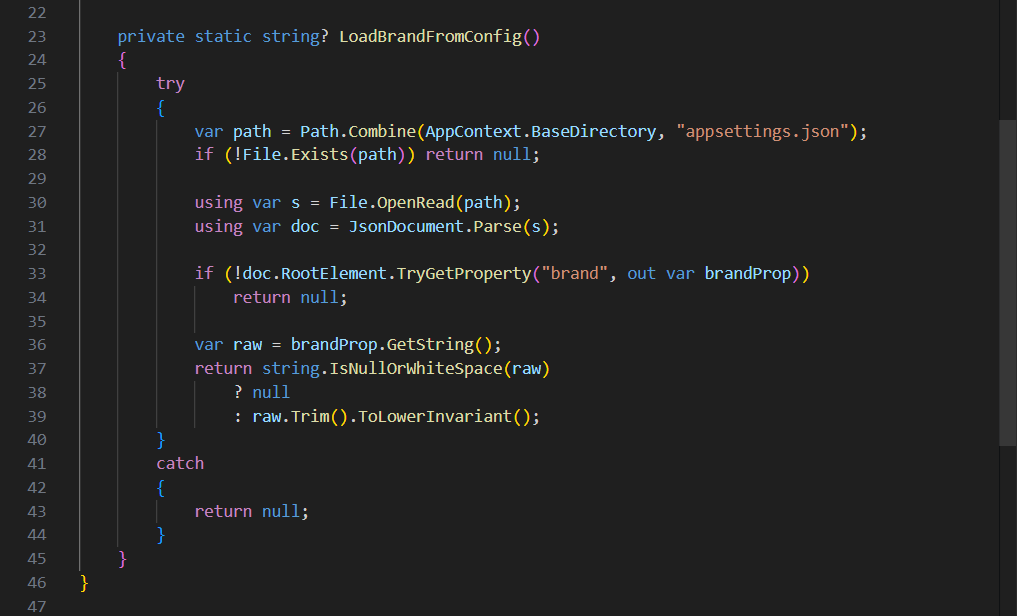
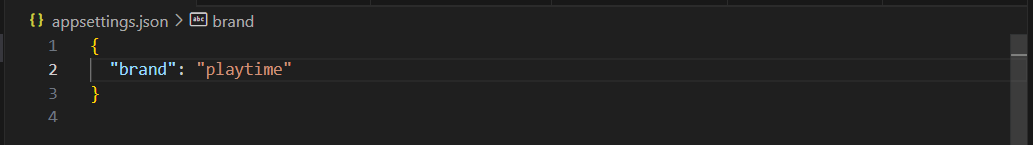
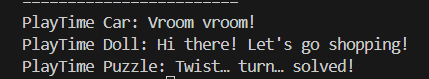
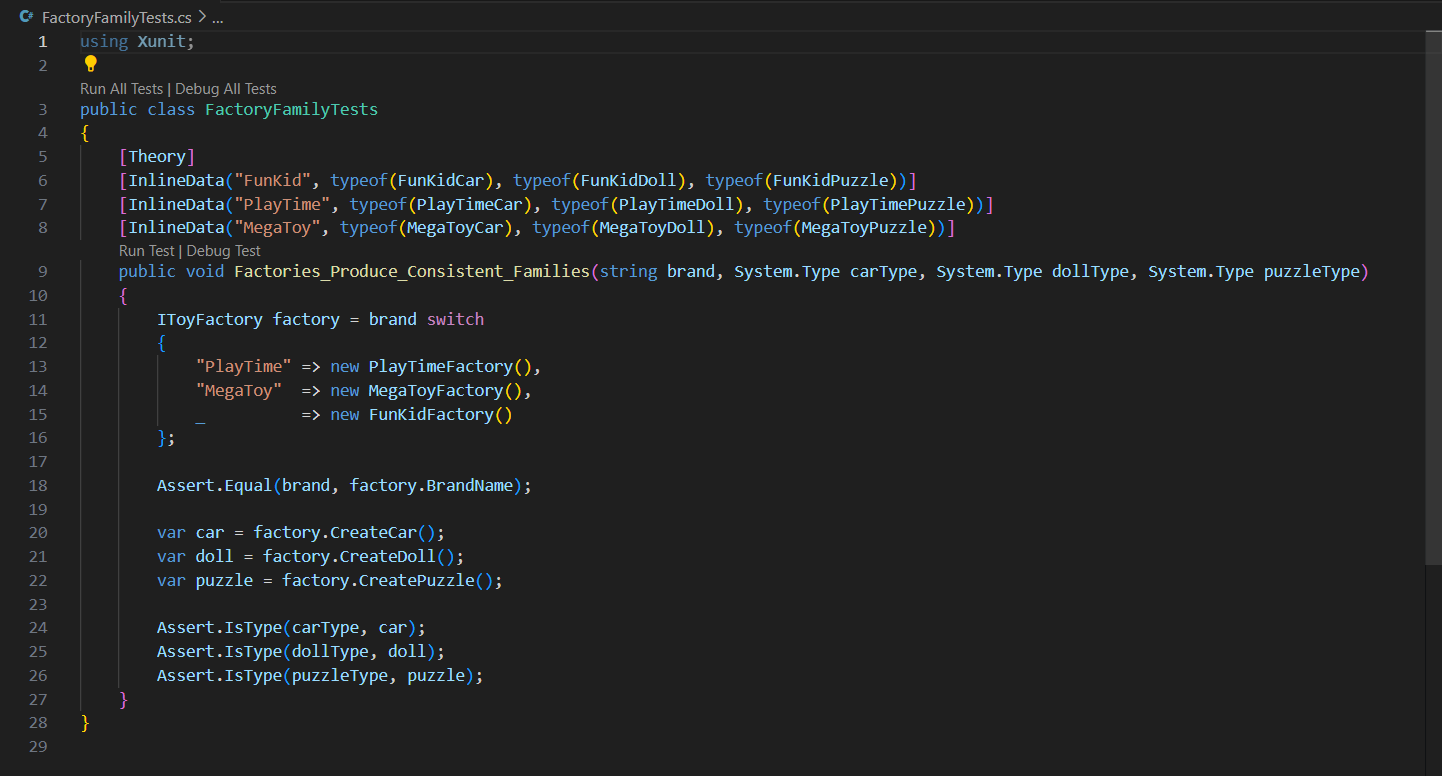
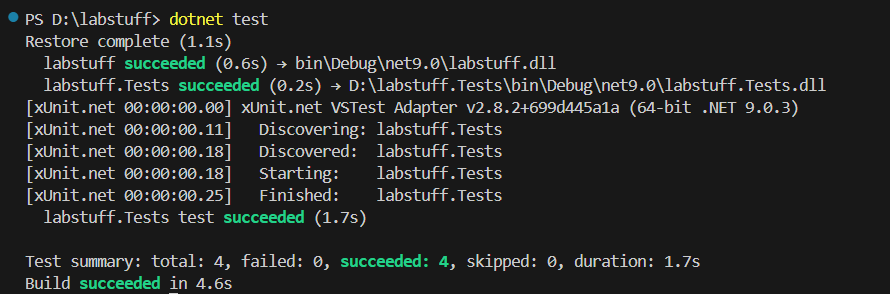
PlayTime Doll: Hi there! Let's go shopping!



# Extended Questions

1. How does the Abstract Factory promote the **Open/Closed Principle**?
2. Why does ToyStore not depend on specific toy classes?
3. What happens if a new product type (e.g., Puzzle) is added?
4. How is Abstract Factory different from Factory Method?
5. What would change if you add a new brand (e.g., MegaToyFactory)?

# Coding Extensions

1. Add a new product type IPuzzle, and extend all factories and brands to support it.   
       
2. Add a new brand MegaToyFactory.   
      
3. Load brand choice from a config file instead of console input.   
      
4. Write a unit test verifying each factory produces consistent families.   
     
   
5. Display ASCII art banners for each brand.   
   