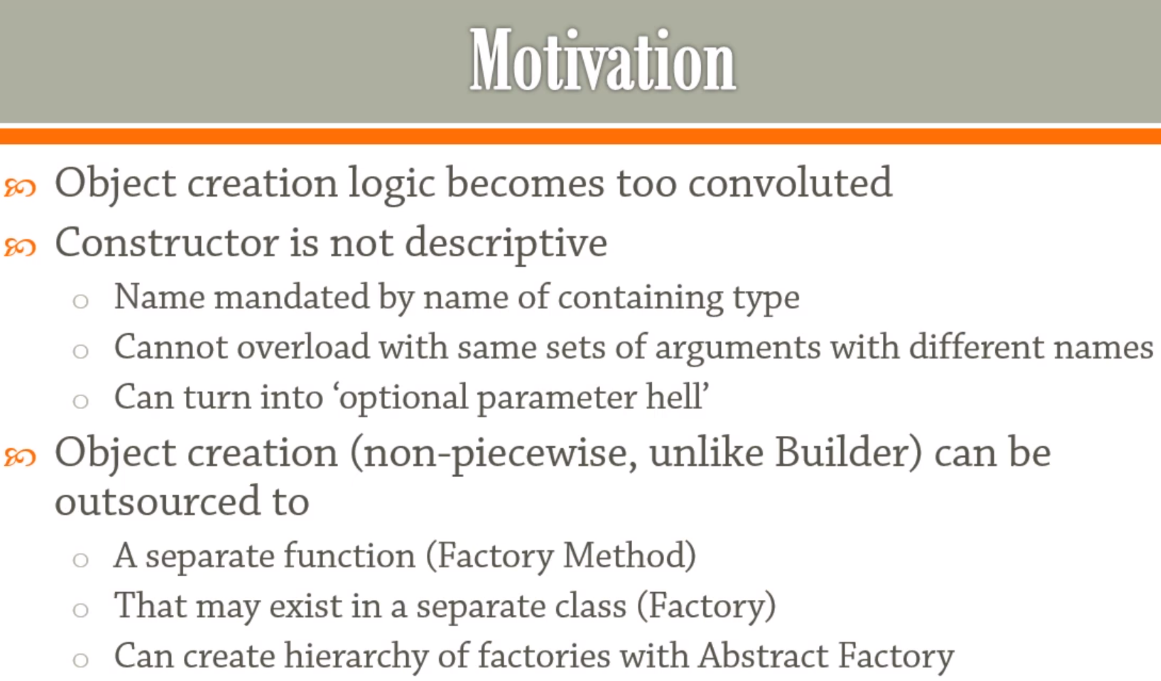
**Lesson01 What is the Factory Design Pattern**

**Notes: -**

**1-we will cover factory method and abstract factory**

**2-with factory design pattern we can implement object creation through factory method or create multiple builder and inherit from abstract factory class**



**3-with factory design pattern it will resolve the issue of the overloaded constructor of the same parameter type and name**

**(like handle point initialization between the Cartesian or Polar system ways)**

**Example of why we using factory pattern**

**public class Point{**

**private double x, y;**

**/// <summary>**

**/// Initialize point from either cartesian or polar**

**/// </summary>**

**/// <param name="a">x if cartesian , rho if polar</param>**

**/// <param name="b">y if cartesian , rho is polar</param>**

**/// <param name="system"></param>**

**public Point(double a, double b, CoordinateSystem system = CoordinateSystem.Cartesian){**

**switch (system){**

**case CoordinateSystem.Cartesian:**

**x = a;**

**y = b;**

**break;**

**case CoordinateSystem.Polar:**

**x = a \* Math.Cos(b);**

**y = a \* Math.Sin(b);**

**break;**

**default:**

**throw new ArgumentOutOfRangeException(nameof(system), system, null);}}}**

**Lesson02 method factory pattern**

**Notes: -**

**1-with factory method we define static method for each system to initialize the properties and use it on the main class as below**

**(it will resolve the issue of multiple system of point instance)**

**public class Point{**

**private double x, y;**

**public Point(double x, double y){**

**this.x = x;**

**this.y = y;}**

**//each method initialize new instance of Point**

**public static Point NewCartesianPoint(double x, double y){**

**return new Point(x, y);}**

**//each method initialize new instance of Point**

**public static Point NewPolarPoint(double rho, double theta){**

**return new Point(rho \* Math.Cos(theta), rho \* Math.Sin(theta));}**

**public override string ToString(){return $"X : {x} , Y : {y}";}}**

**//on Main method we initialize the instance directly through static method as below**

**//this is called factory builder pattern**

**class Program{**

**static void Main(string[] args){**

**var point = Point.NewPolarPoint(1.0, Math.PI / 2);**

**WriteLine(point.ToString());**

**var point2 = Point.NewCartesianPoint(10, 20);**

**WriteLine(point2.ToString());**

**ReadLine();}}**

**Lesson03 async method factory pattern**

**Notes:-**

**1-if you want to apply async operation inside the constructor you cannot , so you have to make async method and call it thorugh the instance itself**

**2-there are another / better way to apply method factory pattern as declare static method with call async method that return the instance itself**

**static async Task Main(string[] args){**

**//it will call the async factory method that it will return the class itself**

**await Foo.CreateAsync();}**

**public class Foo{**

**private Foo(){}**

**private async Task<Foo> InitAsync(){**

**await Task.Delay(1000);**

**return this;}**

**public static Task<Foo> CreateAsync(){**

**var result = new Foo();**

**return result.InitAsync();}}**

**Lesson04 Factory**

**Notes:-**

**1-one of the purpose factory pattern is to separate job by set the creation of the instances on separate class as below**

**2-the disadvantage of the below design is that you can initialize the instance of the point outside from the factory class (it will solve later in Lesson05 Inner factory)**

**public class Point{**

**private double x, y;**

**public Point(double x, double y){**

**this.x = x;**

**this.y = y;}**

**public override string ToString(){return $"X : {x} , Y : {y}";}}**

**//this class handle the creation of class instance**

**public static class PointFactory{**

**public static Point NewCartesianPoint(double x, double y){**

**return new Point(x, y);}**

**public static Point NewPolarPoint(double rho, double theta){**

**return new Point(rho \* Math.Cos(theta), rho \* Math.Sin(theta));}}**

**//the point we face here is that you can initialize the class outside the factory, otherwise //we separate the job on each class the Model and the factory as below**

**static void Main(string[] args){**

**var point = PointFactory.NewPolarPoint(1.0, Math.PI / 2);**

**WriteLine(point.ToString());}**

**Lesson05 Inner Factory**

**Notes:-**

**1-with inner factory it will resolve the issue of create instance of the Point class outside the factory class by set inside the class itself**

**//we set static class factory inside the class Point as below**

**public class Point{**

**private double x, y;**

**public Point(double x, double y){**

**this.x = x;**

**this.y = y;}**

**public override string ToString(){return $"X : {x} , Y : {y}";}**

**public static class PointFactory{**

**public static Point NewCartesianPoint(double x, double y){return new Point(x, y);}**

**public static Point NewPolarPoint(double rho, double theta){**

**return new Point(rho \* Math.Cos(theta), rho \* Math.Sin(theta));}}}**

**class Program{**

**static void Main(string[] args){**

**var point = Point.PointFactory.NewPolarPoint(1.0, Math.PI / 2);**

**WriteLine(point.ToString());}}**

**Lesson06 Factory Pattern**

**Notes:-**

**1-with using abstract factory we apply multiple classes that inherit from the interface and using as parameter on the factory class as below**

**(Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.)**

**public interface IHotDrink{**

**void Consume();**

**IHotDrink Prepare(int amount);}**

**public class Coffee : IHotDrink{**

**public void Consume(){Console.WriteLine("This coffee is delicious!");}**

**public IHotDrink Prepare(int amount){**

**Console.WriteLine($"Grind some beans, boil water, pour {amount} ml, add cream and sugar, enjoy!");**

**return new Coffee();}}**

**public class Tea : IHotDrink{**

**public void Consume(){Console.WriteLine("This tea is nice but I'd prefer it with milk.");}**

**public IHotDrink Prepare(int amount){**

**Console.WriteLine($"Put in tea bag, boil water, pour {amount} ml, add lemon, enjoy!");**

**return new Tea();}}**

**public enum AvailableDrink{Coffee,Tea}**

**public class HotDrinkFactory{**

**public static IHotDrink Create(AvailableDrink name){**

**switch (name){**

**case AvailableDrink.Coffee:**

**return new Coffee();**

**case AvailableDrink.Tea:**

**return new Tea();**

**default:**

**throw new ArgumentNullException($"Provider is not supported");}}}**

**static void Main(string[] args){**

**var machine = HotDrinkFactory.Create(AvailableDrink.Tea);**

**IHotDrink drink = machine.Prepare(100);**

**drink.Consume();**

**ReadLine();}**

**Lesson07 Abstract Factory Pattern**

**Notes:-**

**1-with abstract factory pattern its built by make each builder for each provider as below**

**(with using abstract factory pattern it will provide apply factory pattern over list of factory builder such as I want to make list of hot chocolate of each one of specific requirement)**

**public interface IHotDrink{void Consume();}**

**internal class Coffee : IHotDrink{**

**public void Consume(){Console.WriteLine("This coffee is delicious!");}}**

**internal class Tea : IHotDrink{**

**public void Consume(){Console.WriteLine("This tea is nice but I'd prefer it with milk.");}}**

**public interface IHotDrinkFactory{IHotDrink Prepare(int amount);}**

**internal class TeaFactory : IHotDrinkFactory{**

**public IHotDrink Prepare(int amount){**

**Console.WriteLine($"Put in tea bag, boil water, pour {amount} ml, add lemon, enjoy!");**

**return new Tea();}}**

**internal class CoffeeFactory : IHotDrinkFactory{**

**public IHotDrink Prepare(int amount){**

**Console.WriteLine($"Grind some beans, boil water, pour {amount} ml, add cream and sugar, enjoy!");**

**return new Coffee();}}**

**//in this class it will register and create instance of what we call**

**public class HotDrinkMachine{**

**// violates open-closed{**

**public enum AvailableDrink Coffee, Tea}**

**private List<Tuple<string, IHotDrinkFactory>> namedFactories =**

**new List<Tuple<string, IHotDrinkFactory>>();**

**//we register all factories as below**

**public HotDrinkMachine(){**

**foreach (var t in typeof(HotDrinkMachine).Assembly.GetTypes()){**

**if (typeof(IHotDrinkFactory).IsAssignableFrom(t) && !t.IsInterface){**

**namedFactories.Add(Tuple.Create(**

**t.Name.Replace("Factory", string.Empty), (IHotDrinkFactory)Activator.CreateInstance(t)));}}}**

**//in this method we call the Prepare method to execute the factory class we want**

**public IHotDrink MakeDrink(){**

**Console.WriteLine("Available drinks");**

**for (var index = 0; index < namedFactories.Count; index++){**

**var tuple = namedFactories[index];**

**Console.WriteLine($"{index}: {tuple.Item1}");}**

**while (true){**

**string s;**

**if ((s = Console.ReadLine()) != null && int.TryParse(s, out int i) && i >= 0**

**&& i < namedFactories.Count){**

**Console.Write("Specify amount: ");**

**s = Console.ReadLine();**

**if (s != null && int.TryParse(s, out int amount) && amount > 0){**

**return namedFactories[i].Item2.Prepare(amount);}}**

**Console.WriteLine("Incorrect input, try again.");}}}**

**//we see that it will load only two factory and you specify the feature you want**

**static void Main(string[] args){**

**var machine = new HotDrinkMachine();**

**IHotDrink drink = machine.MakeDrink();**

**drink.Consume();}**