**Lesson01 Proto Type Design Pattern**

**Notes:-**

**1-prototype: when it’s easier to copy an existing object to fully initialize new one.**

**When to use it**

**When to use:-**

**1-complicated objects (e.g. cars) aren’t designed from scratch**

**2-an existing (partially or fully constructed) design is a prototype**

**3-we make copy (clone) the prototype and customize it (apply deep copy support)**

**4-make cloning convenient (vis factory)**

**Lesson02 Why ICloneable is bad**

**Notes:-**

**1-by default ICloneable is using only for shallow copy which means that it will refer the two reference variable to the same instance , so its bad to use IClonable in deep copy**

**using System;**

**namespace ICloneablePro.Models{**

**public class Address : ICloneable{**

**public string StreetName { get; set; }**

**public int HouseNumber { get; set; }**

**public Address(string streetName,int houseNumber){**

**if (streetName == null){throw new ArgumentNullException(paramName: nameof(streetName)); ;}**

**StreetName = streetName;**

**HouseNumber = houseNumber;}**

**public override string ToString(){**

**return $"{nameof(StreetName)} : {StreetName} , {nameof(HouseNumber)} : {HouseNumber}";}**

**//we inherit ICloneable in Person and Address class and it will apply shallow copy**

**public object Clone(){return new Address(StreetName, HouseNumber);}}}**

**using System;**

**namespace ICloneablePro.Models{**

**public class Person : ICloneable{**

**public string[] Names { get; set; }**

**public Address Address { get; set; }**

**public Person(string[] names, Address address){**

**if (names == null){throw new ArgumentNullException(paramName: nameof(names)); ;}**

**if (address == null){throw new ArgumentNullException(paramName: nameof(address)); ;}**

**Names = names;**

**Address = address;}**

**public override string ToString(){**

**return $"{nameof(Names)} : {string.Join(",",Names)} , {nameof(Address)} : {Address}";}**

**//we inherit ICloneable in Person and Address class and it will apply shallow copy**

**public object Clone(){return new Person(Names, (Address)Address.Clone());}}}**

**using ICloneablePro.Models;**

**using static System.Console;**

**namespace ICloneablePro{**

**class Program{**

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

**var john = new Person(new[] { "John", "Smith" },new Address("London St",123));**

**WriteLine(john.ToString());**

**//this way is inefficent**

**//var jane = john;**

**//jane.Names[0] = "Jane";**

**//we see that it will make shallow copy which make the two refernces**

**//refer to same instance object**

**var jane2 = (Person)john.Clone();**

**jane2.Address.HouseNumber = 2345;**

**WriteLine(jane2.ToString());**

**ReadLine();}}}**

**Lesson03 Copy Constructor**

**Notes:-**

**1-another approach to apply deep copy is to make overload constructor that accept instance of the target instance we want to clone and make overloaded constructor foreach internal instance property (not recommended)**

**using System;**

**namespace CopyConstPro.Models{**

**public class Address{**

**public string StreetAddress, City, Country;**

**public Address(string streetAddress, string city, string country){**

**StreetAddress = streetAddress ?? throw new ArgumentNullException(paramName: nameof(streetAddress));**

**City = city ?? throw new ArgumentNullException(paramName: nameof(city));**

**Country = country ?? throw new ArgumentNullException(paramName: nameof(country));}**

**//to apply deep copy we pass the instnace and assign value with apply initialize of sub class**

**public Address(Address other){**

**StreetAddress = other.StreetAddress;**

**City = other.City;**

**Country = other.Country;}**

**public override string ToString(){**

**return $"{nameof(StreetAddress)}: {StreetAddress}, {nameof(City)}: {City}, {nameof(Country)}: {Country}";}}}**

**using System;**

**namespace CopyConstPro.Models{**

**public class Employee{**

**public string Name;**

**public Address Address;**

**public Employee(string name, Address address){**

**Name = name ?? throw new ArgumentNullException(paramName: nameof(name));**

**Address = address ?? throw new ArgumentNullException(paramName: nameof(address));}**

**//to apply deep copy we pass the instnace and assign value with apply initialize of sub class**

**public Employee(Employee other){**

**Name = other.Name;**

**Address = new Address(other.Address);}**

**public override string ToString(){**

**return $"{nameof(Name)}: {Name}, {nameof(Address)}: {Address}";}}}**

**using CopyConstPro.Models;**

**using static System.Console;**

**namespace CopyConstPro{**

**class Program{**

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

**var john = new Employee("John", new Address("123 London Road", "London", "UK"));**

**//var chris = john;**

**var chris = new Employee(john);**

**chris.Name = "Chris";**

**WriteLine(john); // oops, john is called chris**

**WriteLine(chris);}}}**

**Lesson04 Explicit Copy Constructor**

**Notes:-**

**1-with explicit copy constructor we define interface that receive generic type and with define method with return generic type and used inside every class and sub class as below (But this way is better that previous ways but its hard call coded).**

**//this interface define method called DeepCopy that implement on each class**

**namespace ExplicitDeepCopyPro.Interfaces{**

**public interface IProtoType<T>{T DeepCopy();}}**

**using ExplicitDeepCopyPro.Interfaces;**

**using System;**

**namespace ExplicitDeepCopyPro.Models{**

**public class Address : IProtoType<Address>{**

**public string StreetAddress, City, Country;**

**public Address(string streetAddress, string city, string country){**

**StreetAddress = streetAddress ?? throw new ArgumentNullException(paramName: nameof(streetAddress));**

**City = city ?? throw new ArgumentNullException(paramName: nameof(city));**

**Country = country ?? throw new ArgumentNullException(paramName: nameof(country));}**

**//to apply deep copy we pass the instnace and assign value with apply initialize of sub class**

**public Address(Address other){**

**StreetAddress = other.StreetAddress;**

**City = other.City;**

**Country = other.Country;}**

**public override string ToString(){**

**return $"{nameof(StreetAddress)}: {StreetAddress}, {nameof(City)}: {City}, {nameof(Country)}: {Country}";}**

**public Address DeepCopy(){**

**return new Address(StreetAddress, City, Country);}}}**

**using ExplicitDeepCopyPro.Interfaces;**

**using System;**

**namespace ExplicitDeepCopyPro.Models{**

**public class Employee : IProtoType<Employee>{**

**public string Name;**

**public Address Address;**

**public Employee(string name, Address address){**

**Name = name ?? throw new ArgumentNullException(paramName: nameof(name));**

**Address = address ?? throw new ArgumentNullException(paramName: nameof(address));}**

**//deep copy we pass the instnace and assign value with apply initialize of sub class**

**public Employee(Employee other){**

**Name = other.Name;**

**Address = new Address(other.Address);}**

**public override string ToString(){**

**return $"{nameof(Name)}: {Name}, {nameof(Address)}: {Address}";}**

**//we see that we call DeppCopy() of sub class inside the base class as below**

**public Employee DeepCopy(){**

**return new Employee(Name, Address.DeepCopy());}}}**

**//on the main entry we apply the explicit copy as below**

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

**var john = new Employee("John", new Address("123 London Road", "London", "UK"));**

**var chris = new Employee(john);**

**chris.Name = "Chris";**

**WriteLine(john); // oops, john is called chris**

**WriteLine(chris);}**

**Lesson05 Prototype Inheritance**

**Notes:-**

**1-we use the prototype inheritance that apply Deep Copy in generic way by provide interface with default implementation with abstract method that implement on each class parent > child (Person, Employee, Address)**

**Steps:-**

**1-on the interface IDeepCopyable<T> only for the default constructor and used interface default implementation to call it from the extension method and call abstract method**

**namespace CopyInheritPro.Interfaces{**

**//in order to apply Depp copy we apply default implementation interface DeepCopy()**

**//with call the abstract CopyTo that call the extension method DeepCopyExtensions > DeepCopy**

**public interface IDeepCopyable<T> where T : new(){**

**void CopyTo(T target);**

**//the default interface implemntation will return new Deep Copy instance**

**public T DeepCopy(){**

**T t = new T();**

**CopyTo(t);**

**return t;}}}**

**2-on the Address Class inherit from the IDeepCopyable and assign Street Name, House Number**

**using CopyInheritPro.Interfaces;**

**namespace CopyInheritPro.Models{**

**//we implement the interface IDeepCopyable**

**public class Address : IDeepCopyable<Address>{**

**public string StreetName;**

**public int HouseNumber;**

**public Address(string streetName, int houseNumber){**

**StreetName = streetName;**

**HouseNumber = houseNumber;}**

**public Address(){}**

**public override string ToString(){**

**return $"{nameof(StreetName)}: {StreetName}, {nameof(HouseNumber)}: {HouseNumber}";}**

**public void CopyTo(Address target){**

**target.StreetName = StreetName;**

**target.HouseNumber = HouseNumber;}}}**

**3-on the Person Class we inherit IDeepCopyable<Person> > DeepCopy that call the extension method DeepCopy() that call the IDeepCopyable<Person> > CopyTo**

**using CopyInheritPro.Extensions;**

**using CopyInheritPro.Interfaces;**

**namespace CopyInheritPro.Models{**

**//we implement the interface IDeepCopyable**

**public class Person : IDeepCopyable<Person>{**

**public string[] Names;**

**public Address Address;**

**public Person(){}**

**public Person(string[] names, Address address){Names = names;Address = address;}**

**public override string ToString(){**

**return $"{nameof(Names)}: {string.Join(",", Names)}, {nameof(Address)}: {Address}";}**

**public virtual void CopyTo(Person target){**

**//call the ICloneable interface**

**target.Names = (string[])Names.Clone();**

**//it will call the Address DeepCopy() of Address**

**target.Address = Address.DeepCopy();}}}**

**4-on the Employee Class that inherit from the Person calss we inherit CopyTo with assign Salary as below**

**using CopyInheritPro.Interfaces;**

**namespace CopyInheritPro.Models{**

**//we inherit class Employee from the Person**

**public class Employee : Person, IDeepCopyable<Employee>{**

**public int Salary;**

**//it will call CopyTo with call the Person class**

**public void CopyTo(Employee target){**

**base.CopyTo(target);**

**target.Salary = Salary;}**

**public override string ToString(){return $"{base.ToString()}, {nameof(Salary)}: {Salary}";}}}**

**using CopyInheritPro.Interfaces;**

**using CopyInheritPro.Models;**

**namespace CopyInheritPro.Extensions{**

**public static class DeepCopyExtensions{**

**//we call DeepCopy inside the overloaded CopyTo > Person class**

**//this is used to DeepCopy Child > Address**

**public static T DeepCopy<T>(this IDeepCopyable<T> item) where T : new()**

**{return item.DeepCopy();}**

**//this extension method call IDeepCopyable > DeepCopy() > CopyTo abstract**

**//this is used to DeepCopy Parent > Person**

**public static T DeepCopy<T>(this T person) where T : Person, new(){**

**return ((IDeepCopyable<T>)person).DeepCopy();}}}**

**//on the main entry we call DeepCopy() that call DeepCopy(this person) from the extension method //and it will internally call CopyTo from the interface twice one the Person and another for the //address property**

**using CopyInheritPro.Extensions;**

**using CopyInheritPro.Models;**

**using static System.Console;**

**namespace CopyInheritPro{**

**class Program{**

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

**var john = new Employee();**

**john.Names = new[] { "John", "Doe" };**

**john.Address = new Address { HouseNumber = 123, StreetName = "London Road" };**

**john.Salary = 321000;**

**var copy = john.DeepCopy();**

**copy.Names[1] = "Smith";**

**copy.Address.HouseNumber++;**

**copy.Salary = 123000;**

**WriteLine(john);**

**WriteLine(copy);**

**ReadLine();}}}**

**Lesson06 Copy through Serialization**

**Notes:-**

**1-instead of implement interface on each class Address, Person, Employee , we can use extension method with apply Memory Stream or Xml Serialize or Json Serialize**

**Steps:-**

**1-on the extension method we declare the three extension methods as below**

**using Newtonsoft.Json;**

**using System.IO;**

**using System.Runtime.Serialization.Formatters.Binary;**

**using System.Text;**

**using System.Xml.Serialization;**

**namespace SerIizationCopyPro.Extensions{**

**public static class ExtensionMethods{**

**public static T DeepCopy<T>(this T self){**

**MemoryStream stream = new MemoryStream();**

**BinaryFormatter formatter = new BinaryFormatter();**

**formatter.Serialize(stream, self);**

**stream.Seek(0, SeekOrigin.Begin);**

**object copy = formatter.Deserialize(stream);**

**stream.Close();**

**return (T)copy;}**

**public static T DeepCopyXml<T>(this T self){**

**using (var ms = new MemoryStream()){**

**XmlSerializer s = new XmlSerializer(typeof(T));**

**s.Serialize(ms, self);**

**ms.Position = 0;**

**return (T)s.Deserialize(ms);}}**

**public static T DeepCopyJSON<T>(this T self){**

**var ms = new MemoryStream();**

**using (var sw = new StreamWriter(stream: ms, encoding: Encoding.UTF8, bufferSize: 4096, leaveOpen: true)) // last parameter is important**

**using (var jsonWriter = new JsonTextWriter(sw)){**

**var s = new JsonSerializer();**

**s.Serialize(jsonWriter, self);**

**sw.Flush();**

**ms.Seek(0, SeekOrigin.Begin);**

**using (var sr = new StreamReader(ms)){return (T)s.Deserialize(sr, typeof(T));}}}}}**

**//on each models we don’t need any more the Copy To only apply [Serializable]**

**[Serializable]**

**public class Person{**

**public string[] Names;**

**public Address Address;**

**public Person() { }**

**public Person(string[] names, Address address){**

**Names = names;**

**Address = address;}**

**public override string ToString(){**

**return $"{nameof(Names)}: {string.Join(",", Names)}, {nameof(Address)}: {Address}";}}**

**[Serializable]**

**public class Employee : Person{**

**public int Salary;**

**public Employee() { }**

**public override string ToString(){return $"{base.ToString()}, {nameof(Salary)}: {Salary}";}}**

**[Serializable]**

**public class Address{**

**public string StreetName;**

**public int HouseNumber;**

**public Address(string streetName, int houseNumber){**

**StreetName = streetName;**

**HouseNumber = houseNumber;}**

**public Address() { }**

**public override string ToString(){**

**return $"{nameof(StreetName)}: {StreetName}, {nameof(HouseNumber)}: {HouseNumber}";}}**