

Deep Copy of Object in C#

Posted by [Jignesh Trivedi](#) in [Articles](#) | [C# Language](#) on July 15, 2013

Tags: [C#](#), [C# Deep Copy of Object](#), [Create deep copy of an object](#), [erialization and Reflection](#)

In this article, We can create deep copy of an object with the help of Serialization and Reflection.

Reader Level: 



Download Files:

- [DeepCopy.zip](#)

Source: <http://www.c-sharpcorner.com/UploadFile/ff2f08/deep-copy-of-object-in-C-Sharp/>

Introduction

System.Object is base class of all classes, structures, enumeration and delegates. We can say it is the root of the type hierarchy. System.Object has a method called MemberwiseClone that helps to create a clone of the current object instance.

Problem Statement

The MemberwiseClone method of System.Object creates a shallow copy of a new object and it copies the non-static fields of the current object instance to a new object. Copy Object is performed property by property, if property is a value type then it copies data bit by bit and if a property is a reference type then it copies the reference of the original object, in other words the clone object refers to the same object. This means that the MemberwiseClone method does not create a deep copy of the object.

Solution

There are numerous ways to implement a deep copy class object. Two of them I have described here with examples.

1. Implement Deep Cloning using Serializing Deserializing objects
2. Implement Deep Cloning using Reflection

1. Implement Deep Cloning using Serializing Deserializing objects

The ICloneable interface enables us to provide customized implementation to create a copy of the existing object using the "Clone" method. Generally the "Object.MemberwiseClone" method helps us to create a copy of an existing object, but it creates a shallow copy of the object.

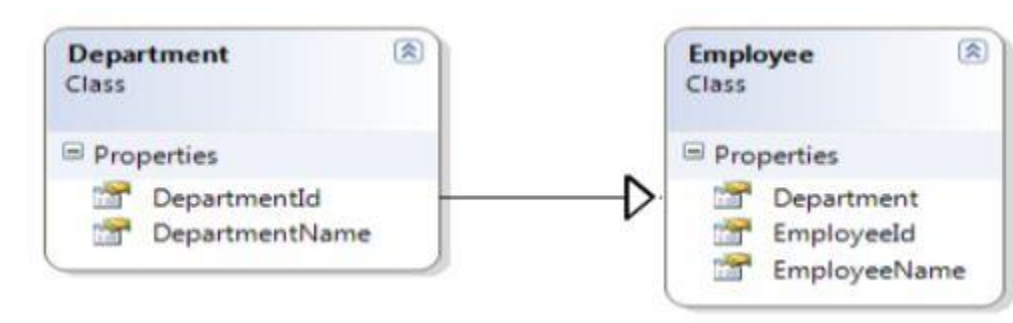
Serialization is the process of storing the state of an object to the stream of bytes. Deserialization is the process of converting from a byte stream to an original object. In .NET there are many ways to perform

serialization and deserialization like Binary serialization, XML serialization, data contract serialization and so on. Binary serialization is much faster than XML serialization and binary serialization uses private and public fields to serialize so binary serialization is a good option to perform serialization and deserialization.

Using serialization and deserialization, we can create a deep copy of any object. Note that to perform serialization and deserialization all types must be marked as "serializable".

Example

Suppose I have an Employee class and it contains the Department class type as a property. Now I implemented the Employee class with the ICloneable interface and I have implemented the "Clone" method of the ICloneable interface. Using a binary formatter, I just serialized the current object and deserialized it into a new object.



```
[Serializable]
public class Department
{
    public int DepartmentId { get; set; }
    public string DepartmentName { get; set; }
}

[Serializable]
public class Employee : ICloneable
{
    public int EmployeeId { get; set; }
    public string EmployeeName { get; set; }
    public Department Department { get; set; }

    public object Clone()
    {
        using (MemoryStream stream = new MemoryStream())
        {
            if (this.GetType().IsSerializable)
            {
                BinaryFormatter formatter = new BinaryFormatter();
                formatter.Serialize(stream, this);
                stream.Position = 0;
            }
        }
    }
}
```

```

        return formatter.Deserialize(stream);
    }
    return null;
}
}
}

```

We can also implement this using an Extension method.

```

public static class ObjectExtension
{
    public static T CopyObject<T>(this object objSource)
    {
        using (MemoryStream stream = new MemoryStream())
        {
            BinaryFormatter formatter = new BinaryFormatter();
            formatter.Serialize(stream, objSource);
            stream.Position = 0;
            return (T)formatter.Deserialize(stream);
        }
    }
}

```

2. Implement Deep Cloning using Reflection

Reflection is used for obtaining Meta information of an object at runtime. The System.Reflection namespace has classes that allow us to obtain object information at runtime as well as we can create an instance of a type of object from the existing object and also accessing its properties and invoke its methods. To do a deep copy of an object, we can use reflection. Consider the following code. Here I have created one static method that accepts any object and it returns the same type of object with a new reference.

```

public class Utility
{
    public static object CloneObject(object objSource)
    {
        //step : 1 Get the type of source object and create a new instance of that type
        Type typeSource = objSource.GetType();
        object objTarget = Activator.CreateInstance(typeSource);

        //Step2 : Get all the properties of source object type
        PropertyInfo[] propertyInfo = typeSource.GetProperties(BindingFlags.Public | BindingFlags.NonPublic
| BindingFlags.Instance);

        //Step : 3 Assign all source property to target object 's properties
        foreach (PropertyInfo property in propertyInfo)
        {
            //Check whether property can be written to

```

```

        if (property.CanWrite)
        {
            //Step : 4 check whether property type is value type, enum or string type
            if (property.PropertyType.IsValueType || property.PropertyType.IsEnum ||
                property.PropertyType.Equals(typeof(System.String)))
            {
                property.SetValue(objTarget, property.GetValue(objSource, null), null);
            }
            //else property type is object/complex types, so need to recursively call this method until the
            end of the tree is reached
            else
            {
                object objPropertyValue = property.GetValue(objSource, null);
                if (objPropertyValue == null)
                {
                    property.SetValue(objTarget, null, null);
                }
                else
                {
                    property.SetValue(objTarget, CloneObject(objPropertyValue), null);
                }
            }
        }
    }
    return objTarget;
}
}

```

This can be also be achieved by an Object extension method that is described below.

Using Object Extension Method

```

public static class ObjectExtension
{
    public static object CloneObject(this object objSource)
    {
        //Get the type of source object and create a new instance of that type
        Type typeSource = objSource.GetType();
        object objTarget = Activator.CreateInstance(typeSource);

        //Get all the properties of source object type
        PropertyInfo[] propertyInfo = typeSource.GetProperties(BindingFlags.Public | BindingFlags.NonPublic
            | BindingFlags.Instance);

        //Assign all source property to taget object 's properties
        foreach (PropertyInfo property in propertyInfo)
        {
            //Check whether property can be written to

```

```

        if (property.CanWrite)
        {
            //check whether property type is value type, enum or string type
            if (property.PropertyType.IsValueType || property.PropertyType.IsEnum ||
property.PropertyType.Equals(typeof(System.String)))
            {
                property.SetValue(objTarget, property.GetValue(objSource, null), null);
            }
            //else property type is object/complex types, so need to recursively call this method until the
            end of the tree is reached
            else
            {
                object objPropertyValue = property.GetValue(objSource, null);
                if (objPropertyValue == null)
                {
                    property.SetValue(objTarget, null, null);
                }
                else
                {
                    property.SetValue(objTarget, objPropertyValue.CloneObject(), null);
                }
            }
        }
    }
    return objTarget;
}
}

```

Sample code and out put

```

class Program
{
    static void Main(string[] args)
    {
        Employee emp = new Employee();
        emp.EmployeeId = 1000;
        emp.EmployeeName = "Jignesh";
        emp.Department = new Department { DepartmentId = 1, DepartmentName = "Examination" };

        Employee empClone = emp.Clone() as Employee;

        Employee empClone1 = Utility.CloneObject(emp) as Employee;

        //now Change Original Object Value
        emp.EmployeeName = "Tejas";
        emp.Department.DepartmentName = "Admin";

        //Print origianl as well as clone object properties value.
    }
}

```

```

Console.WriteLine("Original Employee Name : " + emp.EmployeeName);
Console.WriteLine("Original Department Name : " + emp.Department.DepartmentName);

Console.WriteLine("");

Console.WriteLine("Clone Object Employee Name (Clone Method) : " +
empClone.EmployeeName);
Console.WriteLine("Clone Object Department Name (Clone Method) : " +
empClone.Department.DepartmentName);

Console.WriteLine("");

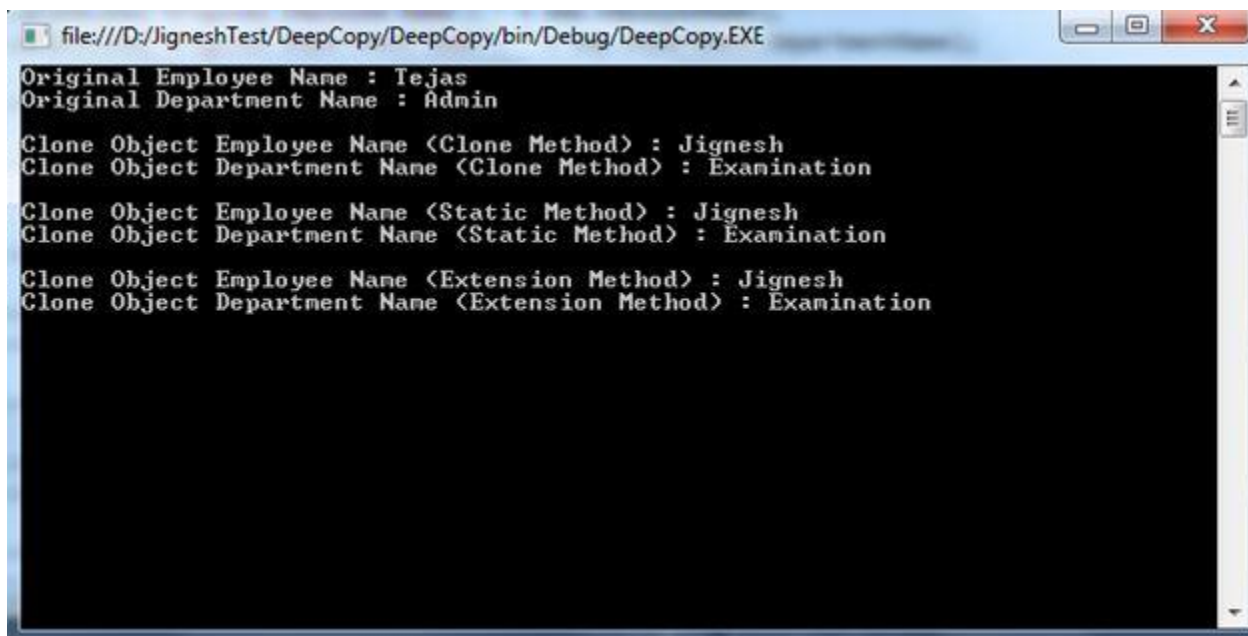
Console.WriteLine("Clone Object Employee Name (Static Method) : " +
empClone1.EmployeeName);
Console.WriteLine("Clone Object Department Name (Static Method) : " +
empClone1.Department.DepartmentName);

Console.WriteLine("");

Console.WriteLine("Clone Object Employee Name (Extension Method) : " +
empClone2.EmployeeName);
Console.WriteLine("Clone Object Department Name (Extension Method) : " +
empClone2.Department.DepartmentName);

Console.ReadKey();
}
}

```



```

file:///D:/JigneshTest/DeepCopy/DeepCopy/bin/Debug/DeepCopy.EXE
Original Employee Name : Tejas
Original Department Name : Admin

Clone Object Employee Name <Clone Method> : Jignesh
Clone Object Department Name <Clone Method> : Examination

Clone Object Employee Name <Static Method> : Jignesh
Clone Object Department Name <Static Method> : Examination

Clone Object Employee Name <Extension Method> : Jignesh
Clone Object Department Name <Extension Method> : Examination

```

Conclusion

Using Serialization and Reflection we can create a deep copy of an object. The only disadvantage of a deep copy using serialization is that we must mark our object as "serializable".