27. Prototype

Prototype is a design pattern that allows you to easily clone existing objects. Instead of copying all the data from one object to another field-by-field, you just have a method to clone the object inside the object itself.

This has two core advantages. Firstly, all of your code for copying an object resides in a single place in your code base, which enforces the *don't repeat yourself* (DRY) principle. Secondly, it allows you to copy private fields into the new object, which you wouldn't have been able to do at all if you just did a field-by-field copying.

Prototype can be summarized as follows:

- A class has a method that returns an exact copy of the instance of the class.
- This method may be called `Clone` or `Copy`.
- This method performs a field-by-field copy of the class, including its private fields.
- But even though the object that this method returns has the same values in all of its fields as the original object, it's a completely different object reference, so modifying it will not change the original object.

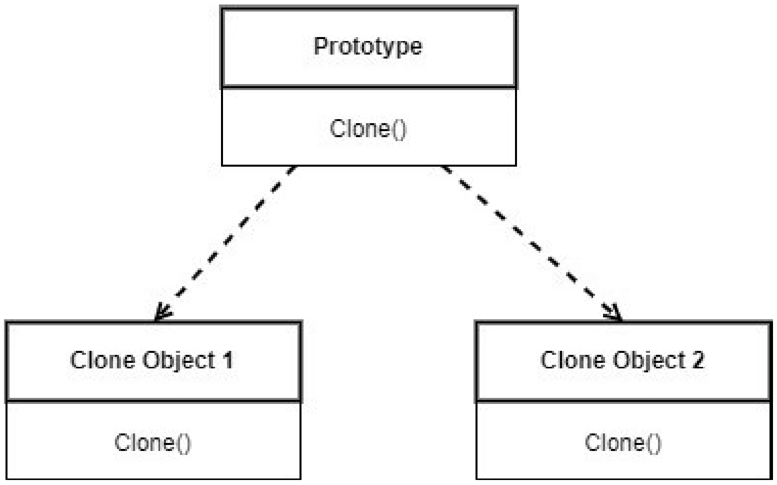


Figure 27.1 - Prototype UML diagram

We will now go through an example implementation of Prototype. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Creational_Patterns/Prototype

Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

Prototype implementation example

To demonstrate Prototype design pattern, we will create a .NET console application project.

When using Prototype, it would be a good practice to create an interface that you would want any cloneable class to implement. This will make all cloneable types behave consistently. So, we will add the `ICloneable.cs` file with the following interface definition:

```
namespace Prototype_Demo;

internal interface ICloneable
{
    ICloneable Clone();
}
```

Then, we will create the actual cloneable object. It will be added as `CloneableObject.cs` file and its code will be as follows:

```
namespace Prototype_Demo;

internal class CloneableObject : ICloneable
{
    private readonly int internalData;
    private readonly string internalTitle;

    public CloneableObject(string title)
    {
        var random = new Random();
        internalData = random.Next();

        internalTitle = title;
    }

    public int Data => internalData;
    public string Title => internalTitle;

    public ICloneable Clone()
    {
        return (CloneableObject)MemberwiseClone();
    }
}
```

So, in our class, we have two public read-only properties: `Data` and `Title`. None of them can be changed once the object has been created. But we have the `Clone` method that returns a copy of our object by calling the `MemberwiseClone` method of the base `object` data type. This method will perform field-by-field copying of our object. And this will include the private fields.

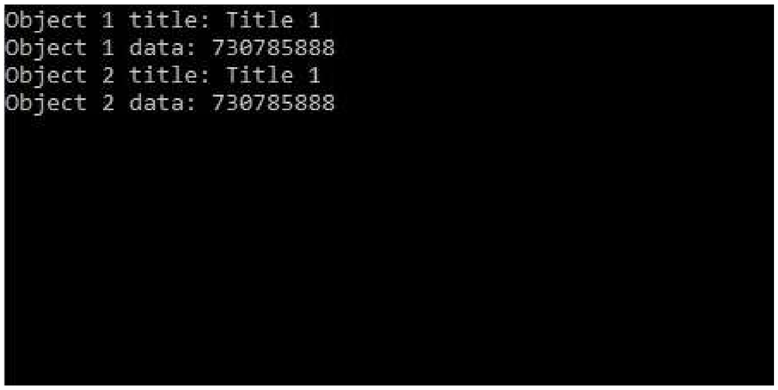
Let's now see if our `Clone` method behaves as we have intended. To do so, we will replace the content of the `Program.cs` file with the following:

```
using Prototype_Demo;

var object1 = new CloneableObject("Title 1");

Console.WriteLine(
    $"Object 1 title: {object1.Title}");
Console.WriteLine(
    $"Object 1 data: {object1.Data}");

var object2 = (CloneableObject)object1.Clone();

Console.WriteLine(
    $"Object 2 title: {object2.Title}");
Console.WriteLine(
    $"Object 2 data: {object2.Data}");
```

```

Console.ReadKey();
```

In here, we create an object and give it a specific title. Its `Data` property will be populated with a random integer when the object is created. We output both the `Title` and `Data` properties into the console. Then we create another instance of this object by cloning our initial object instance. And we, once again, output its `Title` and `Data` properties into the console to verify that they have the same values as in our first object.

And, as this figure demonstrates, the values of the properties are the same. This proves that the cloning process works.



```
Object 1 title: Title 1
Object 1 data: 730785888
Object 2 title: Title 1
Object 2 data: 730785888
```

Figure 27.2 - Proof that our object cloning works

This concludes our overview of the Prototype design pattern. Let's now go through the summary of its main benefits.

Benefits of using Prototype

The main benefits of using the Prototype design pattern can be summarized as follows:

- If you need to copy an object, the logic of copying it will be present in a single place in the entire codebase, which enforces the DRY principle.
- Because cloning of an object is performed inside the object itself, private fields can be copied too.

Let's now have a look at caveats that you need to be familiar with while using the Prototype design pattern (although there aren't many of them).

Caveats of using Prototype

One thing that you need to be aware of while using Prototype is the difference between shallow and deep copying. When you perform a shallow copy, which is exactly what we had a look at in our example, only the values of the fields with basic data types get copied. So, the fields containing `string`, `int`, `double`, `bool`, etc. will be copied, but the fields containing custom data types, such as classes, won't be.

Deep copy is when you copy everything, including any custom data types that your object contains and any custom data type fields that those custom data types contain. Therefore, to ensure that you can perform a deep copy, you need to either make the other data types cloneable or perform manual field-by-field copying inside of your `Clone` method.

Other than that, there are no caveats on using the Prototype design pattern. It is the best way to copy objects.