Design Pattern: PROTOTYPE

Old way:

- Copy the object of class by assigning ‘=’ operator.

Employee employee1 = new Employee()

Employee employee2 = employee1

- Problem:
  - the ‘=’ copies only the reference, so the objects copied by using this way will have the same memory address.
  - Changes happened to the copies of the object will alter the roots.

PROTOTYPE

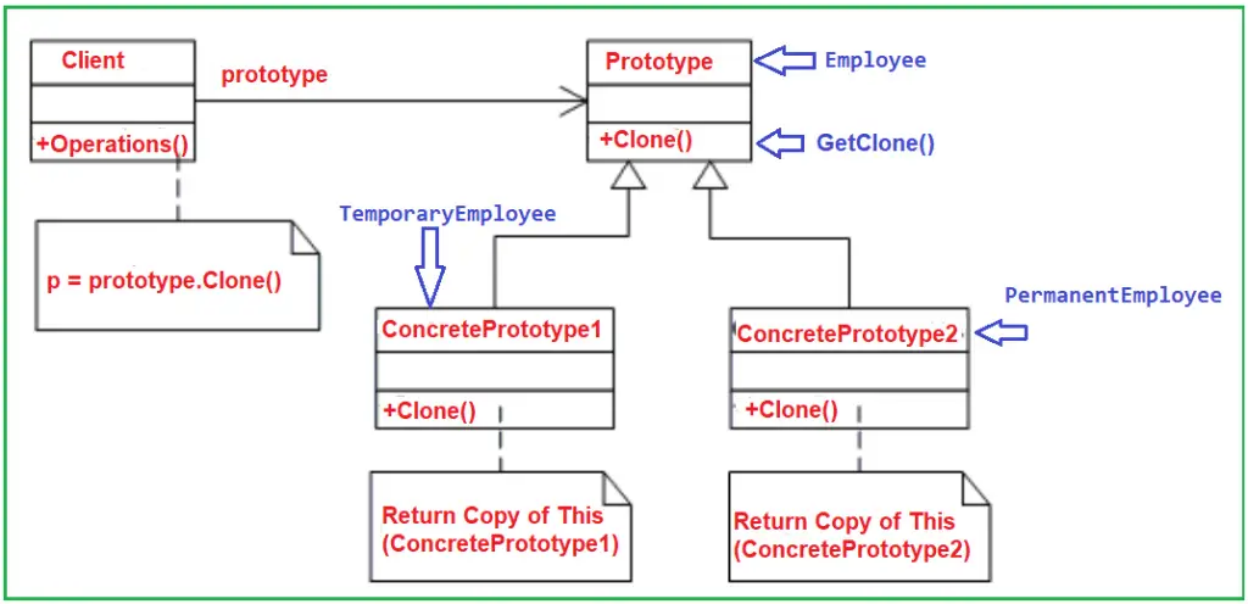
Cloning an object : Call by Value

Copying the object and its data into the new one without depending on its root.

Important components to implement Prototype:

- Prototype, used for types of objects that can be cloned.
- Concrete Prototype, the class that implements Prototype
- Client, the class that creates the object by asking Prototype to clone itself.

Scheme (Class Diagram)



Shallow Copy and Deep Copy

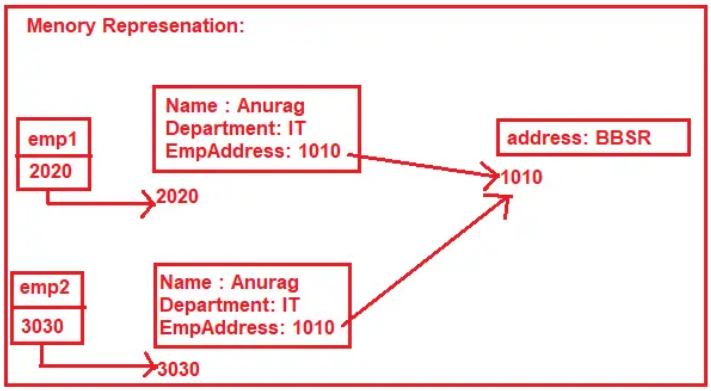
- shallow

The object may contain value type fields and reference type fields. ‘Shallow copy’ will copy the value type fields to the new object, but for the reference type fields, it will only copy the reference. Both the reference and the cloned reference type fields will point to the same memory location.

```
public class Employee
{
    public string Name { get; set; }
    public string Department { get; set; }
    public Address EmpAddress { get; set; }

    public Employee GetClone()
    {
        return (Employee)this.MemberwiseClone();
    }
}

public class Address
{
    public string address { get; set; }
}
```

```
static void Main(string[] args)
{
    Employee emp1 = new Employee();
    emp1.Name = "Anurag";
    emp1.Department = "IT";
    emp1.EmpAddress = new Address() { address = "BBSR" };

    Employee emp2 = emp1.GetClone();
    emp2.Name = "Pranaya";
    emp2.EmpAddress.address = "Mumbai";
}
```

Client Code



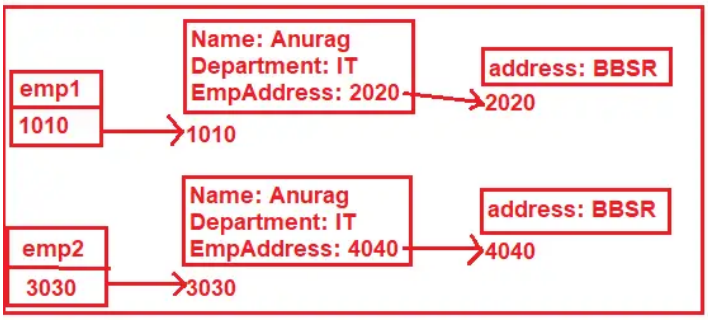
- deep

Copy the whole data: both reference and value type fields.

```
public class Employee
{
    public string Name { get; set; }
    public string Department { get; set; }
    public Address EmpAddress { get; set; }

    public Employee GetClone()
    {

        Employee employee = (Employee)this.MemberwiseClone();
        employee.EmpAddress = EmpAddress.GetClone();
        return employee;
    }
}

public class Address
{
    public string address { get; set; }
    public Address GetClone()
    {
        return (Address)this.MemberwiseClone();
    }
}
```

```
static void Main(string[] args)
{
    Employee emp1 = new Employee();
    emp1.Name = "Anurag";
    emp1.Department = "IT";
    emp1.EmpAddress = new Address()
        {
            address = "BBSR"
        };

    Employee emp2 = emp1.GetClone();
    emp2.Name = "Pranaya";
    emp2.EmpAddress.address = "Mumbai";
}
```

Client Code



Memory Represenatation

PROS

- Convenient
    - Clone objects without coupling to the original classes
    - Avoid repeated initialization code in favor of cloning pre-built prototypes
    - When dealing with complex objects, prototype is the alternative for inheritance

CONS

The more complex the objects are, the more tricky it will be to clone the circular references.