

search...

Home » [Creational Patterns](#) » [Prototype](#)[Bookmark this on Delicious](#)

## Main Menu

[Home](#)  
[Design Principles](#)  
[Open Close Principle](#)  
[Dependency Inversion Principle](#)  
[Interface Segregation Principle](#)  
[Single Responsibility Principle](#)  
[Liskov's Substitution Principle](#)  
[Creational Patterns](#)  
[Singleton](#)  
[Factory](#)  
[Factory Method](#)  
[Abstract Factory](#)  
[Builder](#)  
[Prototype](#)  
[Object Pool](#)  
[Behavioral Patterns](#)  
[Chain of Responsibility](#)  
[Command](#)  
[Interpreter](#)  
[Iterator](#)  
[Mediator](#)  
[Memento](#)  
[Observer](#)  
[Strategy](#)  
[Template Method](#)  
[Visitor](#)  
[Null Object](#)  
[Structural Patterns](#)  
[Adapter](#)  
[Bridge](#)  
[Composite](#)  
[Decorator](#)  
[Flyweight](#)  
[Proxy](#)  
[Design Pattern Books](#)

## Forum

[What Design Pattern To Choose?](#)  
[Design Principles and Patterns](#)  
[Enterprise Patterns](#)  
[Books](#)

# Prototype Pattern

Like 26



Tweet

## Motivation

Today's programming is all about costs. Saving is a big issue when it comes to using computer resources, so programmers are doing their best to find ways of improving the performance. When we talk about object creation we can find a better way to have new objects: cloning. To this idea one particular design pattern is related: rather than creation it uses cloning. If the cost of creating a new object is large and creation is resource intensive, we clone the object.

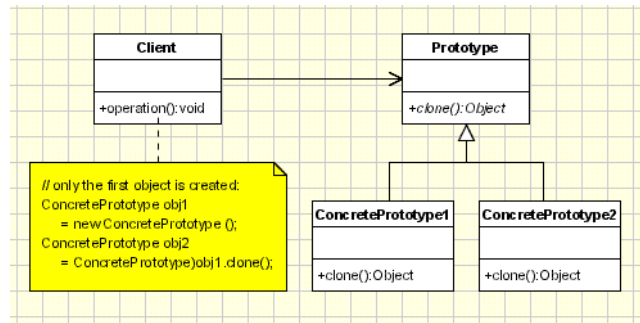
The Prototype design pattern is the one in question. It allows an object to create customized objects without knowing their class or any details of how to create them. Up to this point it sounds a lot like the Factory Method pattern, the difference being the fact that for the Factory the palette of prototypical objects never contains more than one object.

## Intent

specifying the kind of objects to create using a prototypical instance  
 creating new objects by copying this prototype

## Implementation

The pattern uses abstract classes, as we will see below and only three types of classes making its implementation rather easy.



The classes participating to the Prototype Pattern are:

**Client** - creates a new object by asking a prototype to clone itself.  
**Prototype** - declares an interface for cloning itself.  
**ConcretePrototype** - implements the operation for cloning itself.

The process of cloning starts with an initialized and instantiated class. The Client asks for a new object of that type and sends the request to the Prototype class. A ConcretePrototype, depending of the type of object is needed, will handle the cloning through the Clone() method, making a new instance of itself.

Here is a sample code for the Prototype pattern:

```

public interface Prototype {
    public abstract Object clone ( );
}

public class ConcretePrototype implements Prototype {
    public Object clone() {
        return super.clone();
    }
}

public class Client {

    public static void main( String arg[] )
    {
        ConcretePrototype obj1= new ConcretePrototype ();
        ConcretePrototype obj2 = ConcretePrototype.obj1.clone();
    }
}
  
```

This example is rather trivial, but the real use of the pattern comes when we don't know what we're actually cloning. For example if we need the newly created object to be stored in a hashtable we can use it like this:

```

// Violation of Likov's Substitution Principle
class Rectangle
{
    protected int m_width;
    protected int m_height;

    public void setWidth(int width){
        m_width = width;
    }

    public void setHeight(int height){
        m_height = height;
    }
}
  
```

```

        public int getWidth(){
            return m_width;
        }

        public int getHeight(){
            return m_height;
        }

        public int getArea(){
            return m_width * m_height;
        }
    }

    class Square extends Rectangle
    {
        public void setWidth(int width){
            m_width = width;
            m_height = width;
        }

        public void setHeight(int height){
            m_width = height;
            m_height = height;
        }
    }

    class LspTest
    {
        private static Rectangle getNewRectangle()
        {
            // it can be an object returned by some factory ...
            return new Square();
        }

        public static void main (String args[])
        {
            Rectangle r = LspTest.getNewRectangle();

            r.setWidth(5);
            r.setHeight(10);
            // user knows that r it's a rectangle.
            // It assumes that he's able to set the width and height as for the base class

            System.out.println(r.getArea());
            // now he's surprised to see that the area is 100 instead of 50.
        }
    }

```

## Applicability & Examples

Use Prototype Pattern when a system should be independent of how its products are created, composed, and represented, and:

- Classes to be instantiated are specified at run-time
- Avoiding the creation of a factory hierarchy is needed
- It is more convenient to copy an existing instance than to create a new one.

### Example 1

In building stages for a game that uses a maze and different visual objects that the character encounters it is needed a quick method of generating the maze map using the same objects: wall, door, passage, room... The Prototype pattern is useful in this case because instead of hard coding (using new operation) the room, door, passage and wall objects that get instantiated, CreateMaze method will be parameterized by various prototypical room, door, wall and passage objects, so the composition of the map can be easily changed by replacing the prototypical objects with different ones.

The Client is the CreateMaze method and the ConcretePrototype classes will be the ones creating copies for different objects.

### Example 2:

Suppose we are doing a sales analysis on a set of data from a database. Normally, we would copy the information from the database, encapsulate it into an object and do the analysis. But if another analysis is needed on the same set of data, reading the database again and creating a new object is not the best idea. If we are using the Prototype pattern then the object used in the first analysis will be cloned and used for the other analysis.

The Client is here one of the methods that process an object that encapsulates information from the database. The ConcretePrototype classes will be classes that, from the object created after extracting data from the database, will copy it into objects used for analysis.

## Specific problems and implementation

### Using a prototype manager

When the application uses a lot of prototypes that can be created and destroyed dynamically, a registry of available prototypes should be kept. This registry is called the prototype manager and it should implement operations for managing registered prototypes like registering a prototype under a certain key, searching for a prototype with a given key, removing one from the register, etc. The clients will use the interface of the prototype manager to handle prototypes at run-time and will ask for permission before using the Clone() method.

There is not much difference between an implementation of a prototype which uses a prototype manager and a factory method implemented using class registration mechanism. Maybe the only difference consists in the performance.

### Implementing the Clone operation

A small discussion appears when talking about how deep or shallow a clone should be: a deep clone clones the instance variables in the cloning object while a shallow clone shares the instance variables between the clone and the original. Usually, a shallow clone is enough and very simple, but cloning complex prototypes should use deep clones so the clone and the original are independent, a deep clone needing its components to be the clones of the complex object's components.

### Initializing clones

There are cases when the internal states of a clone should be initialized after it is created. This happens because these values cannot be passed to the Clone() method, that uses an interface which would be destroyed if such parameters were used. In this case the initialization should be done by using setting and resetting operations of the prototype class or by using an initializing method that takes as parameters the values at which the clone's internal states should be set.

## Hot points

- Prototype Manager – implemented usually as a hashtable keeping the object to clone. When use it, prototype become a factory method which uses cloning instead of instantiation.

Deep Clones vs. Shallow Clones – when we clone complex objects which contains other objects, we should take care how they are cloned. We can clone contained objects also (deep cloning) or we can the same reference for them, and to share them between cloned container objects.

Initializing Internal States – there are certain situations when objects need to be initialized after they are created.

[\[ Back \]](#)[< Prev](#)[Next >](#)