Source: C# Corner (www.c-sharpcorner.com) PRINT

Article

Prototype Design Pattern With Java

By **Aman Gupta** on **Updated date Dec 30, 2020**

Introduction

In this series of learning different design patterns, we came across the most used and famous pattern, the prototype design pattern. As discussed in the last article, we saw what a design pattern is, and what it provides when implemented with the bridge design pattern.

A design pattern provides a simple solution to common problems that are faced in day-to-day life by software developers. Three basic types of design patterns exist, which are listed below:

- Creational Pattern
- Structural Pattern
- Behavioral Pattern

Thus, the pattern described here is like a blueprint that you can customize to solve a particular design problem and they only differ by complexity and scalability of the code.

In this article, we will learn about the most famous and commonly used design pattern, i.e the prototype design pattern. Let's get started.

Why the Prototype Design Pattern?

To start with, what is a prototype design pattern? One should know why we need this design pattern, what are the drawbacks behind the other patterns to need to introduce a new design pattern, and why we need this new pattern in our code.

When you have two design patterns that are useful for object creation we can use factory or prototype design patterns.

But when to use what? If we want to create an object factory design pattern is the best way, but there are certain scenarios where object creation takes lots of memory, and lots of time.

Let's take an example of an object created with initial values. These values are coming from the database so it will take some time.

Let's suppose we want a second object, then why create a new object from the database? A second object can be created from the first object, something like copying the details of the first object or something like cloning the object that is known as a prototype design pattern.

What is the Prototype Design Pattern?

Prototype Design Pattern is a part of a creational design pattern, as the name suggests it means “the first model” or “the sample/template of any object before the actual object is constructed”.

As per the “Gang of Four” definition, the Prototype Design Pattern is used in scenarios where the application needs to create a large number of instances of a class, which have almost the same state or differ very little.

In the native language, the prototype design pattern says cloning an existing object instead of creating a new object again.

Implementation

The first thing that strikes the mind when we hear a prototype is “sample/template of any object before the actual object creation”.

So to understand what this prototype design pattern is, let’s assume a case study of a Bookshop.

We create a class Book.java with member variables as bid(Book Id) and bname(BookName) and for that, we need getters and setters for the same and also we need to use the toString() method to print the object.

Book.java

```
package PrototypeDesign;
public class Book {
    private int bid;
    private String Bname;
    public int getBid() {
        return bid;
    }
    public void setBid(int bid) {
        this.bid = bid;
    }
    public String getBname() {
        return Bname;
    }
    public void setBname(String bname) {
        Bname = bname;
    }
    @Override
    public String toString() {
        return "Book [bid=" + bid + ", Bname=" + Bname + "]";
    }
}
```

To open a BookShop, we need to create a class BookShop.java with member variables as shopName and the list of books i.e of type <Book> and for that, we need getters and setters with toString() method to print the object.

BookShop.java

```
package PrototypeDesign;
import java.util.ArrayList;
import java.util.List;
public class BookShop //implementsCloneable
{
    private String shopname;
    List < Book > book = new ArrayList < Book > ();
    public String getShopname() {
        return shopname;
    }
    publicvoidsetShopname(String shopname) {
        this.shopname = shopname;
    }
    public List < Book > getBook() {
        return book;
    }
    publicvoidsetBook(List < Book > book) {
        this.book = book;
    }
    @Override
    public String toString() {
        return "BookShop [shopname=" + shopname + ", book=" + book + "]";
    }
}
```

Let’s create an object of BookShop.java and now let's print this object.

BookShopTester.java

```
package PrototypeDesign;
public class BookShopTester {
    public static void main(String[] args) {
        BookShop bs = new BookShop();
        System.out.println(bs);
    }
}
```

Output

```
BookShop [shopname=null, book=[]]
```

So we don’t want a BookShop which doesn’t have a name and a book list. We should assign some data, add a new method in BookShop.java.

```
public void loadData() {
    for (int i = 1; i <= 5; i++) {
        Book b = new Book();
        b.setBid(i);
        b.setBname("Book" + i);
        getBook().add(b);
    }
}
```

This method loadData() will add the data to the list and now we got BookShop with a new updated method and BookShopTester with the desired output.

BookShop.java(Updated)

```
package PrototypeDesign;
import java.util.ArrayList;
import java.util.List;
public class BookShop //implementsCloneable
{
    private String shopname;
    List < Book > book = new ArrayList < Book > ();
    public String getShopname() {
        return shopname;
    }
    public void setShopname(String shopname) {
        this.shopname = shopname;
    }
    public List < Book > getBook() {
        return book;
    }
    public void setBook(List < Book > book) {
        this.book = book;
    }
    public void loadData() {
        for (int i = 1; i <= 10; i++) {
            Book b = new Book();
            b.setBid(i);
            b.setBname("Book" + i);
            getBook().add(b);
        }
    }
    @Override
    public String toString() {
        return "BookShop [shopname=" + shopname + ", book=" + book + "]";
    }
}
```

BookShopTester.java(Updated)

```
package PrototypeDesign;
public class BookShopTester {
    public static void main(String[] args) {
        BookShop bs = new BookShop();
        bs.setShopname("R Lall Book Depot");
        bs.loadData();
        System.out.println(bs);
    }
}
```

Output

```
BookShop [shopname=R Lal Book Depot, book=[Book [bid=1, Bname=Book1], Book [bid=2,
Bname=Book2], Book [bid=3, Bname=Book3], Book [bid=4, Bname=Book4], Book [bid=5,
Bname=Book5]]]
```

Note
If you are thinking that's all a prototype design pattern, then it's "NO".

What if we want a new object of BookShop in BookShopTester, we have to create a new object and load the data as done with the object(bs), then it will take some time to load the data.

So we have the concept of cloning in java that will copy the object from the old object.

To achieve cloning we have to give permission to BookShop.java class to implement an interface Cloneable. Thus, we need to @override the clone method in BookShop.java.
After updating all the methods and interface we have, updated BookShop and BookShopTester classes below.

BookShop.java(Updated)

```
package PrototypeDesign;
import java.util.ArrayList;
import java.util.List;
public class BookShop //implementsCloneable
{
    private String shopname;
    List < Book > book = new ArrayList < Book > ();
    public String getShopname() {
        return shopname;
    }
    public void set Shopname(String shopname) {
        this.shopname = shopname;
    }
    public List < Book > getBook() {
        return book;
    }
    public void setBook(List < Book > book) {
        this.book = book;
    }
    public void loadData() {
        for (int i = 1; i <= 10; i++) {
            Book b = new Book();
            b.setBid(i);
            b.setBname("Book" + i);
            getBook().add(b);
        }
    }
    @Override
    public String toString() {
        return "BookShop [shopname=" + shopname + ", book=" + book + "]";
    }
    @Override
    protected BookShop clone() throws CloneNotSupportedException {
        //code to achieve deep cloning instead of shallow cloning
        BookShop shop = new BookShop();
        for (Book b: this.getBook()) {
            shop.getBook().add(b);
        }
        return shop;
    }
}
```

BookShopTester.java(Updated)

```
package PrototypeDesign;
public class BookShopTester {
    public static void main(String[] args) throws CloneNotSupportedException {
        BookShop bs = new BookShop();
        bs.setShopname("R Lall Book Depot");
        bs.loadData();
        BookShop bs1 = bs.clone();
        bs.getBook().remove(2); //to achieve deep cloning
        bs1.setShopname("Khurana Books");
        System.out.println(bs);
        System.out.println();
        System.out.println(bs1);
    }
}
```

Output

BookShop [shopname=R Lal Book Depot, book=[Book [bid=1, Bname=Book1], Book [bid=2, Bname=Book2], Book [bid=4, Bname=Book4], Book [bid=5, Bname=Book5]]]
BookShop [shopname=Khurana Books, book=[Book [bid=1, Bname=Book1], Book [bid=2, Bname=Book2], Book [bid=3, Bname=Book3], Book [bid=4, Bname=Book4], Book [bid=5, Bname=Book5]]]

Note
That's the power of the prototype method, for any new method we don't need to load data again from the database as it will lead to more time complexity i.e for the first object we are loading data from the database but for the second object, we are just cloning the object.

Here, in the above code bs.getBook().remove(2); will help in checking whether we have one object two references or if we have a copy of the first object; i.e after removing data from the first object the data for the second object remains unchanged and that is deep cloning.

Advantage

- Reduces the need for sub-classing.
- Hides complexities of creating objects.
- Add or remove objects at runtime(dynamic).

Usage

- The cost of object creation is more.
- Classes are instantiated at runtime.
- Need a minimum number of classes.

Summary

Coming towards the end of this article, we learned how to create a prototype design pattern.

What did we learn?

- What is a Design Pattern?
- Why the Prototype Design Pattern?
- What is a Prototype Design Pattern?
- Implementation
- Advantage of Prototype Design Pattern
- Usage of Prototype Design Pattern.

*__NOTE__
Read all notes in between the article carefully for a better understanding of this Design Pattern.

For other articles on design patterns:

- Singleton Design Pattern here.
- Factory Design Pattern here.
- Builder Design Pattern here.
- Bridge Design Pattern here.
- The static keyword here.