Flyweight Design Pattern in Java

APRIL 4, 2018 BY PANKAJ — 21 COMMENTS

Today we will look into Flyweight design pattern.

Table of Contents [hide]

Flyweight Design Pattern

According to GoF, **flyweight design pattern** intent is:

> Use sharing to support large numbers of fine-grained objects efficiently

Flyweight design pattern is a **Structural design pattern** like Facade pattern, Adapter Pattern and Decorator pattern.

Flyweight design pattern is used when we need to create a lot of Objects of a class. Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or

embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects.

Before we apply flyweight design pattern, we need to consider following factors:

- The number of Objects to be created by application should be huge.
- The object creation is heavy on memory and it can be time consuming too.
- The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program.

To apply flyweight pattern, we need to divide Object property into **intrinsic** and **extrinsic** properties. Intrinsic properties make the Object unique whereas extrinsic properties are set by client code and used to perform different operations. For example, an Object Circle can have extrinsic properties such as color and width.

For applying flyweight pattern, we need to create a **Flyweight factory** that returns the shared objects. For our example, lets say we need to create a drawing with lines and Ovals. So we will have an interface `Shape` and its concrete implementations as `Line` and `Oval`. Oval class will have intrinsic property to determine whether to fill the Oval with given color or not whereas Line will not have any intrinsic property.

Flyweight Design Pattern Interface and Concrete Classes

`Shape.java`

```
package com.journaldev.design.flyweight;

import java.awt.Color;
import java.awt.Graphics;

public interface Shape {

	public void draw(Graphics g, int x, int y, int width, int height,
			Color color);
}
```

`Line.java`

```
package com.journaldev.design.flyweight;

import java.awt.Color;
import java.awt.Graphics;

public class Line implements Shape {

	public Line(){
		System.out.println("Creating Line object");
```

```
                //adding time delay
                try {
                        Thread.sleep(2000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
        @Override
        public void draw(Graphics line, int x1, int y1, int x2, int y2,
                        Color color) {
                line.setColor(color);
                line.drawLine(x1, y1, x2, y2);
        }
```

Oval.java

```
package com.journaldev.design.flyweight;

import java.awt.Color;
import java.awt.Graphics;

public class Oval implements Shape {

        //intrinsic property
        private boolean fill;

        public Oval(boolean f){
                this.fill=f;
                System.out.println("Creating Oval object with fill="+f);
                //adding time delay
                try {
                        Thread.sleep(2000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
        @Override
        public void draw(Graphics circle, int x, int y, int width, int height,
```

Notice that I have intensionally introduced delay in creating the Object of concrete classes to make the point that flyweight pattern can be used for Objects that takes a lot of time while instantiated.

Flyweight Factory

The flyweight factory will be used by client programs to instantiate the Object, so we need to keep a map of Objects in the factory that should not be accessible by client application.

Whenever client program makes a call to get an instance of Object, it should be returned from the HashMap, if not found then create a new Object and put in the Map and then return it. We need to make sure that all the intrinsic properties are considered while creating the Object.

Our flyweight factory class looks like below code.

`ShapeFactory.java`

```
package com.journaldev.design.flyweight;

import java.util.HashMap;

public class ShapeFactory {

	private static final HashMap<ShapeType,Shape> shapes = new
HashMap<ShapeType,Shape>();

	public static Shape getShape(ShapeType type) {
		Shape shapeImpl = shapes.get(type);

		if (shapeImpl == null) {
			if (type.equals(ShapeType.OVAL_FILL)) {
				shapeImpl = new Oval(true);
			} else if (type.equals(ShapeType.OVAL_NOFILL)) {
				shapeImpl = new Oval(false);
			} else if (type.equals(ShapeType.LINE)) {
				shapeImpl = new Line();
			}
			shapes.put(type, shapeImpl);
		}
```

Notice the use of Java Enum for type safety, Java Composition (shapes map) and Factory pattern in `getShape` method.

Flyweight Design Pattern Client Example

Below is a sample program that consumes flyweight pattern implementation.

`DrawingClient.java`

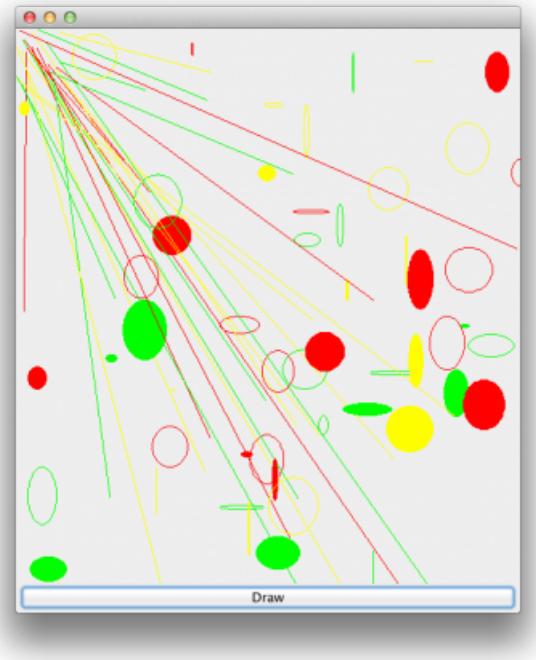
```
package com.journaldev.design.flyweight;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
```

```
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

import com.journaldev.design.flyweight.ShapeFactory.ShapeType;

public class DrawingClient extends JFrame{

        private static final long serialVersionUID = -1350200437285282550L;
        private final int WIDTH;
        private final int HEIGHT;

        private static final ShapeType shapes[] = { ShapeType.LINE
```

I have used random number generation to generate different type of Shapes in our frame.

If you run above client program, you will notice the delay in creating first Line Object and Oval objects with fill as true and false. After that the program executes quickly since its using the shared objects.

After clicking "Draw" button multiple times, the frame looks like below image.



And you will see following output in command line confirming that Objects are shared.

```
Creating Line object
```

```
Creating Oval object with fill=true
Creating Oval object with fill=false
```

Thats all for flyweight pattern, we will look into more design patterns in future posts. If you liked it, please share your thoughts in comments section and share it with others too.

Flyweight Design Pattern Example in JDK

All the wrapper classes `valueOf()` method uses cached objects showing use of Flyweight design pattern. The best example is Java String class String Pool implementation.

Flyweight Design Pattern Important Points

1. In our example, the client code is not forced to create object using Flyweight factory but we can force that to make sure client code uses flyweight pattern implementation but its a complete design decision for particular application.
2. Flyweight pattern introduces complexity and if number of shared objects are huge then there is a trade of between memory and time, so we need to use it judiciously based on our requirements.
3. Flyweight pattern implementation is not useful when the number of intrinsic properties of Object is huge, making implementation of Factory class complex.

That's all for Flyweight design pattern in java.

« PREVIOUS
Facade Design Pattern in Java

NEXT »
Proxy Design Pattern

About Pankaj

If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: DESIGN PATTERNS

Comments

Akshayraj A Kore says
JULY 10, 2018 AT 12:11 PM
Is using a map (HashMap in this example) data structure the only way to implement this pattern?
Reply

Anupama says
APRIL 6, 2017 AT 7:15 PM
Flyweight is used to create objects mainly then why it is called as structural design pattern?
Reply

Philip John says
JANUARY 4, 2017 AT 5:05 AM
Nice tutorial, but I'm confused about the DrawingClient.java example line 23. What does this syntax mean?
private static final ShapeType shapes[] = …
Isn't it suposed to be ShapeType[] shapes = …? Or is it a feature that I'm not aware of? Same applies to the line below.
F. J.
Reply

DEBARATI MAJUMDER says
AUGUST 17, 2017 AT 8:50 PM
Both the syntaxes are correct for array declaration.
Reply

Juan Florez says
NOVEMBER 4, 2016 AT 5:58 PM
For multithreading applications, Should I implement a “double conditional” with synchronised blocks as in Singletons, to check the existence of a specific instance before creating a new one in the HashMap?
Reply

dpk says
SEPTEMBER 13, 2016 AT 9:10 PM
so here we are using same object every time, is not cause any problem ?
Reply

Kuladeep Patil says
AUGUST 2, 2016 AT 7:26 AM
I was browsing google for java design pattern, at many websites I found wrong information .
thanks Pankaj for nice and detailed information about flyweight
Kuladeep Patil
Project Lead, Wipro
Reply

sagar says
SEPTEMBER 24, 2015 AT 4:14 AM
dose not it should be object creation pattern ?
Reply

Rudra says
SEPTEMBER 20, 2015 AT 7:11 PM
I like it to show internal structure of java to show how that work and how much we use them efficiently
Reply

Anil says
AUGUST 25, 2015 AT 9:37 PM

Cool example . Thanks
Reply

You Know says
DECEMBER 22, 2014 AT 6:04 AM
So, raghead, you're not willing to allow criticism, right? Let me tell you something. When something is unique, it means there's no other like it. In that sense, intrinsic properties make objects constant, not unique. Their extrinsic properties are the ones that will make them unique, different from any other. Check your English and stop copying stuff from Joe!
Reply

robothy says
OCTOBER 13, 2016 AT 8:16 PM
I'm also confused with this .
Reply

Valentino says
APRIL 5, 2017 AT 9:46 AM
Who is Joe?
Reply

Enrique says
AUGUST 3, 2014 AT 10:03 PM
thanks
Reply

HIMANSU NAYAK says
FEBRUARY 2, 2014 AT 1:28 AM
1. "Flyweight pattern implementation is not useful when the number of intrinsic properties of Object is huge, making implementation of Factory class complex."
how can object of high intrinsic properties can increase the complexity of the Factory. Since all the properties are getting initialized by the predefined values and are not passed from the factory.
2. "extrinsic properties of an Object should be defined by the client program."
Do you mean if an object has lots of extrinsic properties, lets take around 100, then there is no point using the Flyweight pattern since we cant pass all this properties to flyweight factory.
Reply

Pankaj says
FEBRUARY 2, 2014 AT 4:48 AM
1. We need to make sure that all the intrinsic properties are considered while creating the Object. Notice that Oval class has only one intrinsic property and hence two types of Objects. Now imagine if the number of intrinsic properties are 5, the possible types of Objects will be 2^5=32 that will make our Factory class hard to implement and maintain.
2. Suppose Oval class has 4 additional extrinsic properties, if we will create objects based on them then the Factory class will have so many types of objects and since extrinsic properties can be changed, it's best to let client program use setter methods to set those properties.
Reply

Prakash says
DECEMBER 6, 2017 AT 7:49 AM
Excellent … your comment helped me in better understanding.
Thanking you.
One more question :-
Reference to
https://upload.wikimedia.org/wikipedia/commons/4/4e/W3sDesign_Flyweight_Design_Pattern_UML.jpg
I can see UnsharedFlyweight1 concrete class.
As per my understanding, this class do not have intrinsic properties and so, this class object would not be stores in HashMap to look into when client demand for object.
This class does not facilitates the benefits of Flyweight Design pattern. We need to keep creating new object of this class whenever client demands.
Please correct me if I am wrong.
Thanking you.
Reply

Amit Kumar says
JANUARY 27, 2014 AT 8:35 AM
Very good blogs to understand design pattern concepts
Reply

caleb says
JULY 16, 2013 AT 3:48 PM
This is the proper blog if you wants to find out about this matter. You realize a lot its virtually hard to fight with you (not that I actually would certainly want
Reply

Amit says
JULY 14, 2013 AT 9:57 AM
Nice tutorial pankaj, please post more tutorials on design patterns.
Reply

Smita says
JULY 12, 2013 AT 12:43 PM
very good article pankaj. Keep adding □
Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search for tutorials...

DOWNLOAD ANDROID APP

GET IT ON Google Play

DESIGN PATTERNS TUTORIAL

Java Design Patterns

Creational Design Patterns

- Singleton
- Factory
- Abstract Factory
- Builder
- Prototype

Structural Design Patterns

- Adapter
- Composite
- Proxy
- Flyweight
- Facade
- Bridge
- Decorator

Behavioral Design Patterns

- Template Method
- Mediator
- Chain of Responsibility
- Observer
- Strategy
- Command
- State
- Visitor
- Interpreter

- Iterator
- Memento

Miscellaneous Design Patterns

- Dependency Injection
- Thread Safety in Java Singleton

RECOMMENDED TUTORIALS

Java Tutorials

- Java IO
- Java Regular Expressions
- Multithreading in Java
- Java Logging
- Java Annotations
- Java XML
- Collections in Java
- Java Generics
- Exception Handling in Java
- Java Reflection
- Java Design Patterns
- JDBC Tutorial

Java EE Tutorials

- Servlet JSP Tutorial
- Struts2 Tutorial
- Spring Tutorial
- Hibernate Tutorial
- Primefaces Tutorial
- Apache Axis 2
- JAX-RS
- Memcached Tutorial