**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](https://www.journaldev.com/1557/facade-design-pattern-in-java), [Adapter Pattern](https://www.journaldev.com/1487/adapter-design-pattern-java) and [Decorator pattern](https://www.journaldev.com/1540/decorator-design-pattern-in-java-example).

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,

Color color) {

circle.setColor(color);

circle.drawOval(x, y, width, height);

if(fill){

circle.fillOval(x, y, width, 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);

}

return shapeImpl;

}

public static enum ShapeType{

OVAL\_FILL,OVAL\_NOFILL,LINE;

}

}

Notice the use of [Java Enum](https://www.journaldev.com/716/java-enum) for type safety, [Java Composition](https://www.journaldev.com/1325/composition-in-java-example) (shapes map) and [Factory pattern](https://www.journaldev.com/1392/factory-design-pattern-in-java) 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, ShapeType.OVAL\_FILL,ShapeType.OVAL\_NOFILL };

private static final Color colors[] = { Color.RED, Color.GREEN, Color.YELLOW };

public DrawingClient(int width, int height){

this.WIDTH=width;

this.HEIGHT=height;

Container contentPane = getContentPane();

JButton startButton = new JButton("Draw");

final JPanel panel = new JPanel();

contentPane.add(panel, BorderLayout.CENTER);

contentPane.add(startButton, BorderLayout.SOUTH);

setSize(WIDTH, HEIGHT);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setVisible(true);

startButton.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent event) {

Graphics g = panel.getGraphics();

for (int i = 0; i < 20; ++i) {

Shape shape = ShapeFactory.getShape(getRandomShape());

shape.draw(g, getRandomX(), getRandomY(), getRandomWidth(),

getRandomHeight(), getRandomColor());

}

}

});

}

private ShapeType getRandomShape() {

return shapes[(int) (Math.random() \* shapes.length)];

}

private int getRandomX() {

return (int) (Math.random() \* WIDTH);

}

private int getRandomY() {

return (int) (Math.random() \* HEIGHT);

}

private int getRandomWidth() {

return (int) (Math.random() \* (WIDTH / 10));

}

private int getRandomHeight() {

return (int) (Math.random() \* (HEIGHT / 10));

}

private Color getRandomColor() {

return colors[(int) (Math.random() \* colors.length)];

}

public static void main(String[] args) {

DrawingClient drawing = new DrawingClient(500,600);

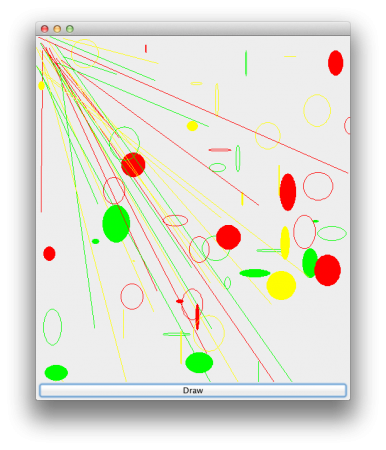
}

}

I have used [random number generation](https://www.journaldev.com/515/random-number-generator-java) 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.

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/flyweight-pattern-example.png)

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](https://www.journaldev.com/1002/wrapper-class-in-java) valueOf() method uses cached objects showing use of Flyweight design pattern. The best example is [Java String](https://www.journaldev.com/1321/java-string-interview-questions-and-answers) class [String Pool](https://www.journaldev.com/797/what-is-java-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.