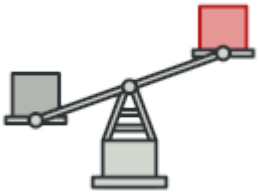


Flyweight in Java

Flyweight is a structural design pattern that allows programs to support vast quantities of objects by keeping their memory consumption low.

The pattern achieves it by sharing parts of object state between multiple objects. In other words, the Flyweight saves RAM by caching the same data used by different objects.

Learn more about Flyweight →

Navigation

Complexity: ★★★

Popularity: ★☆☆

your program doesn't struggle with a shortage of RAM, then you might just ignore this pattern for a while.

Examples of Flyweight in core Java libraries:

`java.lang.Integer#valueOf(int)` (also `Boolean`, `Byte`, `Character`, `Short`, `Long` and `BigDecimal`)

Identification: Flyweight can be recognized by a creation method that returns cached objects instead of creating new.

Rendering a forest

In this example, we're going to render a forest (1.000.000 trees)! Each tree will be represented by its own object that has some state (coordinates, texture and so on). Although the program does its primary job, naturally, it consumes a lot of RAM.

The reason is simple: too many tree objects contain duplicate data (name, texture, color). That's why we can apply the Flyweight pattern and store these values inside separate flyweight objects (the `TreeType` class). Now, instead of storing the same data in thousands of `Tree` objects, we're going to reference one of the flyweight objects with a particular set of values.

The client code isn't going to notice anything since the complexity of reusing flyweight objects is buried inside a flyweight factory.

trees

trees/Tree.java: Contains state unique for each tree

```
package refactoring_guru.flyweight.example.trees;

import java.awt.*;

public class Tree {
    private int x;
    private int y;
    private TreeType type;

    public Tree(int x, int y, TreeType type) {
        this.x = x;
        this.y = y;
```

```
    public void draw(Graphics g) {
        type.draw(g, x, y);
    }
}
```

trees/TreeType.java: Contains state shared by several trees

```
package refactoring_guru.flyweight.example.trees;

import java.awt.*;

public class TreeType {
    private String name;
    private Color color;
    private String otherTreeData;

    public TreeType(String name, Color color, String otherTreeData) {
        this.name = name;
        this.color = color;
        this.otherTreeData = otherTreeData;
    }

    public void draw(Graphics g, int x, int y) {
        g.setColor(Color.BLACK);
        g.fillRect(x - 1, y, 3, 5);
        g.setColor(color);
        g.fillOval(x - 5, y - 10, 10, 10);
    }
}
```

trees/TreeFactory.java: Encapsulates complexity of flyweight creation

```
package refactoring_guru.flyweight.example.trees;

import java.awt.*;
import java.util.HashMap;
import java.util.Map;

public class TreeFactory {
    static Map<String, TreeType> treeTypes = new HashMap<>();

    public static TreeType getTreeType(String name, Color color, String otherTreeData) {
```

```
            result = new TreeType(name, color, otherTreeData);
            treeTypes.put(name, result);
        }
        return result;
    }
}
```

forest

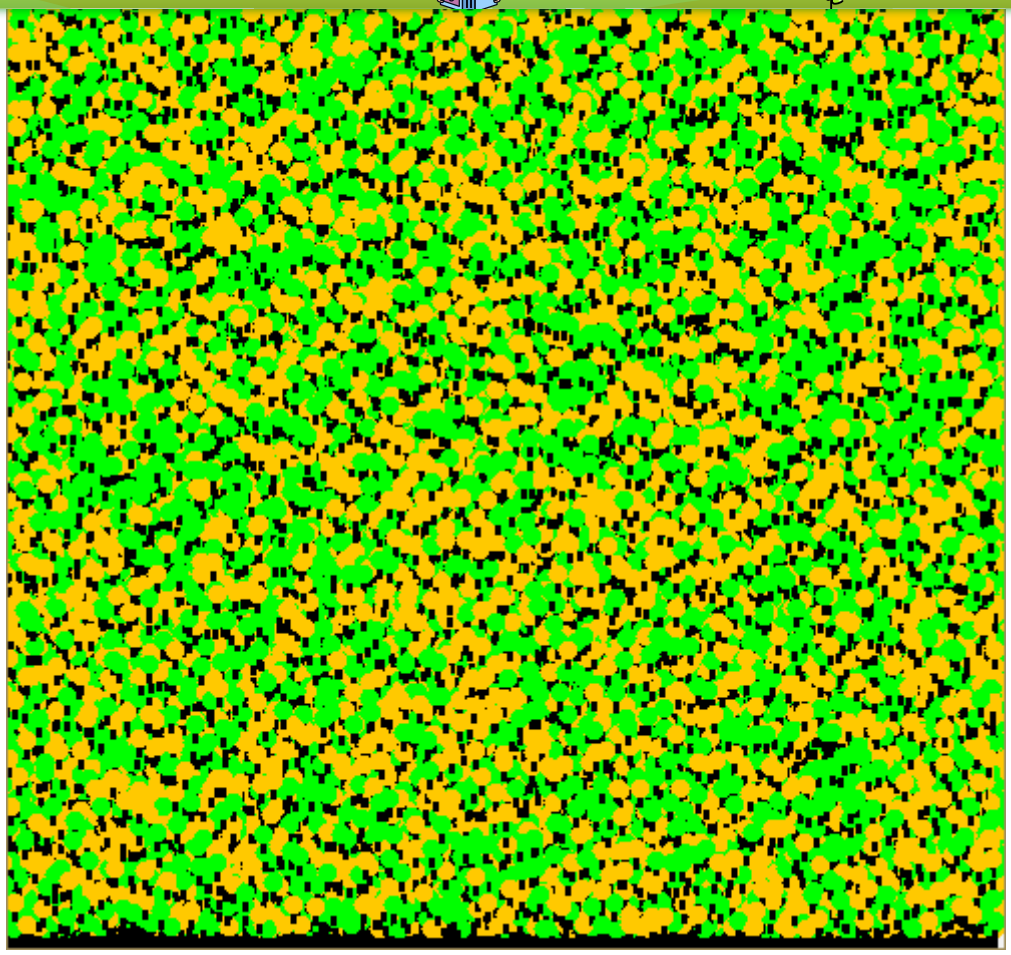
forest/Forest.java: Forest, which we draw

```
package refactoring_guru.flyweight.example.forest;

import refactoring_guru.flyweight.example.trees.Tree;
import refactoring_guru.flyweight.example.trees.TreeFactory;
import refactoring_guru.flyweight.example.trees.TreeType;

import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;

public class Forest extends JFrame {
    private List<Tree> trees = new ArrayList<>();

    public void plantTree(int x, int y, String name, Color color, String otherTreeData) {
        TreeType type = TreeFactory.getTreeType(name, color, otherTreeData);
        Tree tree = new Tree(x, y, type);
        trees.add(tree);
    }

    @Override
    public void paint(Graphics graphics) {
        for (Tree tree : trees) {
            tree.draw(graphics);
        }
    }
}
```

Demo.java: Client code

```
import refactoring_guru.flyweight.example.forest.Forest;

import java.awt.*;

public class Demo {
    static int CANVAS_SIZE = 500;
    static int TREES_TO_DRAW = 1000000;
    static int TREE_TYPES = 2;

    public static void main(String[] args) {
        Forest forest = new Forest();
        for (int i = 0; i < Math.floor(TREES_TO_DRAW / TREE_TYPES); i++) {
            forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                    "Summer Oak", Color.GREEN, "Oak texture stub");
            forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                    "Autumn Oak", Color.ORANGE, "Autumn Oak texture stub");
        }
        forest.setSize(CANVAS_SIZE, CANVAS_SIZE);
        forest.setVisible(true);

        System.out.println(TREES_TO_DRAW + " trees drawn");
        System.out.println("---------------------");
        System.out.println("Memory usage:");
        System.out.println("Tree size (8 bytes) * " + TREES_TO_DRAW);
        System.out.println("+ TreeTypes size (~30 bytes) * " + TREE_TYPES + "");
        System.out.println("---------------------");
        System.out.println("Total: " + ((TREES_TO_DRAW * 8 + TREE_TYPES * 30) / 1024 / 10
                "MB (instead of " + ((TREES_TO_DRAW * 38) / 1024 / 1024) + "MB)");
    }

    private static int random(int min, int max) {
        return min + (int) (Math.random() * ((max - min) + 1));
    }
}
```

OutputDemo.png: Screenshot



OutputDemo.txt: RAM usage stats

```
1000000 trees drawn
---------------------
Memory usage:
Tree size (8 bytes) * 1000000
+ TreeTypes size (~30 bytes) * 2
---------------------
Total: 7MB (instead of 36MB)
```

READ NEXT

Home
Refactoring
Design Patterns
Premium Content