Flyweight

Resources

- Video → 32. All Structural Design Patterns | Decorator, Proxy, Composite, Adapter, Bridge, Facade, FlyWeight
- Video → 30. Design Word Processor using Flyweight Design Pattern | Low Level System Design FlyWeight Pattern

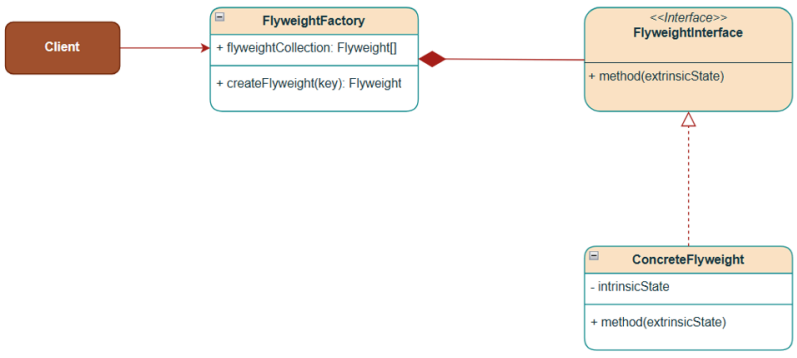
Definition

> ***The Flyweight Design Pattern is a structural pattern that helps reduce memory usage by efficiently sharing data that is common to multiple similar objects. This pattern is widely used in applications where it is required to generate a large number of similar objects.***

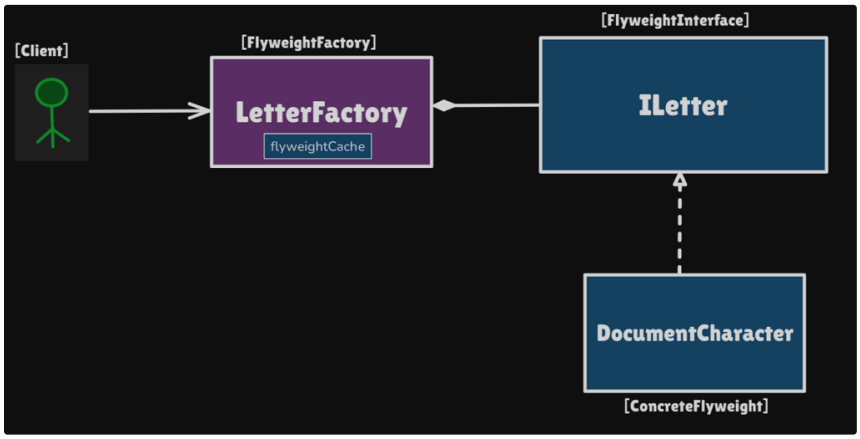
When do we use this?

- When Memory is Limited.
- When Objects share data.
  - **Intrinsic data:** Data that is shared among objects and remains the same once it is set.
  - **Extrinsic data:** This data changes based on client input and differs from one object to another.
- Creation of an Object is expensive.

Class Diagram



Structure of the Flyweight Pattern



Refer to the section below for code:

Flyweight | Word Processor: Flyweight Implementation as a Solution

- **Flyweight Interface(`ILetter`):** Defines methods that use extrinsic state.
- **ConcreteFlyweight(`DocumentCharacter`):** Stores intrinsic (shared) state.
- **FlyweightFactory(`LetterFactory`):** Manages & reuses flyweights.
- **Client(`WordProcessorSimulation`):** Supplies extrinsic state when using flyweights.

Example 1: Robotic Game

Robotic Game: Issue

Let's look at the naive approach in creating a game where we usually tend to create a lot of similar objects.

```
// Sprites class is a heavy-weight object
public class Sprites {
    // In computer graphics, a sprite is a two-dimensional bitmap image used to
    // represent a character, object, or other element in a video game or other
    // computer-generated image.
}
```

```
// Robot class - Used to create Humanoid and Robotic Dog robots
public class Robot {
    int coordinateX;
    int coordinateY;
    String type;
    Sprites body; // heavy-weight object - 2D bitmap image

    Robot(int coordinateX, int coordinateY, String type, Sprites body) {
        this.coordinateX = coordinateX;
        this.coordinateY = coordinateY;
        this.type = type;
        this.body = body;
    }
    // getters and setters
}
```

```
// Client Code - Robotic game creating robots
public class Main {
    public static void main(String[] args) {
        int x = 0;
        int y = 0;
```

```
        // Create 5L Humanoid robots
        for (int i = 0; i < 500000; i++) {
            Sprites humanoidSprite = new Sprites();
            Robot humanoidRobotObject = new Robot(x + i, y + i,
"HUMANOID", humanoidSprite);
        }
        // Create 5L Robotic Dog robots
        for (int i = 0; i < 500000; i++) {
            Sprites roboticDogSprite = new Sprites();
            Robot roboticDogObject = new Robot(x + i, y + i,
"ROBOTIC_DOGS", roboticDogSprite);
        }
        // A total of 10L robots created will result in 10L Sprite
objects created
        // which will consume a lot of memory.
    }
}
```

A total of 10 lakh robots created will result in 10 lakh Sprite objects created, which will consume a lot of memory(let's say each robot is 40 KB, `40 kilobytes * 10 lakh = 40 gigabytes`), and if it exceeds the system's capacity(32 GB RAM), the application might become unresponsive or most probably crash leading to bad user experience and unavailability.

Robotic Game: Flyweight Implementation as a Solution

Let's see how we can solve the issue:

- From the above `Robot` object, remove all the extrinsic data and keep the intrinsic data; it will result in a *flyweight object.*
- This Flyweight Class can be immutable. Provide getter methods only.
- Extrinsic Data can be passed to the Flyweight class as a method parameter.
- Once the Flyweight Object is created, it is **cached** and **reused** whenever required.

```
// Sprites class is a heavyweight object
public class Sprites {
    // In computer graphics, a sprite is a two-dimensional bitmap
image used to
    // represent a character, object, or other element in a video game
or other
    // computer-generated image.
}
// Flyweight (Interface) - for the flyweight object - defines methods
that use extrinsic state.
public interface IRobot {
    // CoordinateX and CoordinateY are extrinsic data - unique to each
object
    void display(int x, int y);
}
```

```
// Concrete Flyweight (Class) - implements the Flyweight interface and
stores intrinsic state.
public class HumanoidRobot implements IRobot {
    // intrinsic data - shared data - common to all objects
    private final String type; // humanoid or robotic dog
    private final Sprites body; //small 2d bitmap (graphic element)

    HumanoidRobot(String type, Sprites body) {
        this.type = type;
        this.body = body;
    }

    public String getType() {
        return type;
    }
```

```

    public Sprites getBody() {
        return body;
    }

    @Override
    public void display(int x, int y) {
        // use the humanoid sprites object
        // and X and Y coordinate to render the image.
        System.out.println("Displaying " + type + " at " + x + ", " + y);
    }
}
```

```
// Concrete Flyweight (Class) - implements the Flyweight interface and stores intrinsic state.
public class RoboticDog implements IRobot {
    // intrinsic data - shared data - common to all objects
    private final String type; // humanoid or robotic dog
    private final Sprites body; // small 2d bitmap (graphic element)

    RoboticDog(String type, Sprites body) {
        this.type = type;
        this.body = body;
    }

    public String getType() {
        return type;
    }

    public Sprites getBody() {
        return body;
    }

    @Override
    public void display(int x, int y) {
        //use the Robotic Dog sprites object
        // and X and Y coordinate to render the image.
        System.out.println("Displaying " + type + " at " + x + ", " + y);
    }
}
```

```
// Flyweight Factory (Class) - creates and manages flyweight objects.
public class RoboticFactory {

    private static final Map<String, IRobot> roboticObjectCache = new HashMap<>();

    public static IRobot createRobot(String robotType) {
        if (roboticObjectCache.containsKey(robotType)) {
            // if exists, return the cached object.
            return roboticObjectCache.get(robotType);
        } else {
            // if not exists, create the object and cache it.
            if (robotType.equals("HUMANOID")) {
                Sprites humanoidSprite = new Sprites();
                IRobot humanoidObject = new HumanoidRobot(robotType, humanoidSprite);
                // add to cache
                roboticObjectCache.put(robotType, humanoidObject);
                return humanoidObject;
            } else if (robotType.equals("ROBOTIC_DOG")) {
                Sprites roboticDogSprite = new Sprites();
                IRobot roboticDogObject = new RoboticDog(robotType, roboticDogSprite);
                // add to cache
                roboticObjectCache.put(robotType, roboticDogObject);
                return roboticDogObject;
            }
        }
```

```
        throw new IllegalArgumentException("Invalid robot type: " +
robotType);
    }

    public static int getTotalRobots() {
        return roboticObjectCache.size();
    }
}
```

```
// Client - supplies extrinsic state when using flyweights.
public class RoboticGameSimulation {
    public static void main(String[] args) {
        System.out.println("====== Flyweight Design Pattern ======");
        // Factory pattern is used to create objects
        // Flyweight pattern is used to reuse objects

        // Create 2 Humanoid robots and provide display
coordinates(extrinsic state) at runtime
        IRobot humanoidRobot1 =
RoboticFactory.createRobot("HUMANOID");
        humanoidRobot1.display(1, 2);
        IRobot humanoidRobot2 =
RoboticFactory.createRobot("HUMANOID");
        humanoidRobot2.display(10, 30);

        // Create 2 Robotic Dog robots and provide display
coordinates(extrinsic state) at runtime
        IRobot roboDog1 = RoboticFactory.createRobot("ROBOTIC_DOG");
        roboDog1.display(2, 9);
        IRobot roboDog2 = RoboticFactory.createRobot("ROBOTIC_DOG");
        roboDog2.display(11, 19);

        // Total robots created: 2 - because we are reusing the same
object
        System.out.println("Total robots created: " +
RoboticFactory.getTotalRobots());
    }
}
```

Here, we created a total of 4 objects(2 humanoids and 2 robotic dogs), but with the flyweight pattern approach, we are consuming 50% less memory(only 2 objects created) by reusing the objects.

Example 2: Word Processor

Word Processor: Issue

```
public class Character {
    char character;
    String fontType;
    int size;
    int row;
    int column;

    public Character(char character, String fontType, int size, int
row, int column) {
        this.character = character;
        this.fontType = fontType;
        this.size = size;
        this.row = row;
        this.column = column;
    }
}
```

```
public class Demo {
    public static void main(String[] args) {
        System.out.println("Word Processor: Issue Demo");
        // Data: "Hello World"
        // Total 11 characters
```

```
        // h = 1 time
        // e = 1 time
        // l = 3 times
        // o = 2 times
        // w = 1 time
        // r = 1 time
        // d = 1 time
        // ' ' = 1 time

        // Create 11 character objects
        Character object1 = new Character('H', "Arial", 10, 0, 0);
        Character object2 = new Character('e', "Arial", 10, 0, 1);
        Character object3 = new Character('l', "Arial", 10, 0, 2);
        Character object4 = new Character('l', "Arial", 10, 0, 3);
        Character object5 = new Character('o', "Arial", 10, 0, 4);
        Character object6 = new Character(' ', "Arial", 10, 0, 5);
        Character object7 = new Character('W', "Arial", 10, 0, 6);
        Character object8 = new Character('o', "Arial", 10, 0, 7);
        Character object9 = new Character('r', "Arial", 10, 0, 8);
        Character object10 = new Character('l', "Arial", 10, 0, 9);
        Character object11 = new Character('d', "Arial", 10, 0, 10);
    }
}
```

A real document may have millions of characters. Implementing a word processor using a naive approach, i.e., without flyweight → we would store "A" object 5000 times → memory heavy. This would result in excessive memory usage, causing our application to crash as discussed above.

Word Processor: Flyweight Implementation as a Solution

```
// Flyweight (Interface) - for the flyweight object - defines methods that use extrinsic state.
public interface ILetter {
    // The position(row,column) is extrinsic data - unique to each object
    void display(int row, int column);
}
```

```
// Concrete Flyweight (Class) - implements the Flyweight interface and stores intrinsic state
public class DocumentCharacter implements ILetter {
    // intrinsic data - shared data - common to all objects
    private final char character;
    private final String fontType;
    private final int size;

    DocumentCharacter(char character, String fontType, int size) {
        this.character = character;
        this.fontType = fontType;
        this.size = size;
    }

    // getter methods only

    @Override
    public void display(int row, int column) {
        //display the character of particular font and size at given location
        System.out.println("Displaying " + character + " at row " + row + " and column " + column);
    }

}
```

```
// Flyweight Factory (Class) - creates and manages flyweight objects.
public class LetterFactory {

    private static final Map<Character, ILetter> characterCache = new HashMap<>();
```

```
    public static ILetter crateLetter(char characterValue) {
        if (characterCache.containsKey(characterValue)) {
            // if exists, return the cached character object.
            return characterCache.get(characterValue);
        } else {
            // if not exists, create the character object and cache
it.
            DocumentCharacter characterObj = new
DocumentCharacter(characterValue, "Arial", 10);
            // add to cache
            characterCache.put(characterValue, characterObj);
            return characterObj;
        }
    }

    public static int getTotalCharacters() {
        return characterCache.size();
    }
}
```

```
// Client - supplies extrinsic state when using flyweights
public class WordProcessorSimulation {
    public static void main(String[] args) {
        // Data: "Hello World"
        // Total 11 characters (including space)
        // h = 1 time
        // e = 1 time
        // l = 3 times (reused)
        // o = 2 times (reused)
        // w = 1 time
        // r = 1 time
        // d = 1 time
        // ' ' = 1 time

        // Create 11 character objects and provide display
position(extrinsic state) at runtime
        ILetter object1 = LetterFactory.crateLetter('H');
        object1.display(0, 0);

        ILetter object2 = LetterFactory.crateLetter('e');
        object2.display(0, 1);

        ILetter object3 = LetterFactory.crateLetter('l');
        object3.display(0, 2);

        ILetter object4 = LetterFactory.crateLetter('l');
        object4.display(0, 3);

        ILetter object5 = LetterFactory.crateLetter('o');
        object5.display(0, 4);

        ILetter object6 = LetterFactory.crateLetter(' ');
        object6.display(0, 5);

        ILetter object7 = LetterFactory.crateLetter('W');
        object7.display(0, 6);

        ILetter object8 = LetterFactory.crateLetter('o');
        object8.display(0, 7);

        ILetter object9 = LetterFactory.crateLetter('r');
        object9.display(0, 8);

        ILetter object10 = LetterFactory.crateLetter('l');
        object10.display(0, 9);

        ILetter object11 = LetterFactory.crateLetter('d');
        object11.display(0, 10);

```

```
        // Total characters created: 8 - because we are reusing the same object
        System.out.println("Total characters created: " + LetterFactory.getTotalCharacters());
    }
}
```

A real document may have millions of characters. Using the flyweight pattern, we store `"A"` just once and reuse it with extrinsic state (position or formatting). This approach proves to be a memory saver, reuses shared data, and passes varying context(extrinsic state) separately.