FLYWEIGHT DESIGN PATTERN

- The **Flyweight Design Pattern** is a structural pattern that focuses on **efficiently sharing common parts of object** state **across many objects** to reduce memory usage and boost performance.
- You need to create **a large number of similar objects**, but most of their data is **shared or repeated**.
- **Storing all object data individually** would result in **high memory consumption.**
- You want to separate **intrinsic state (shared, reusable data)** from **extrinsic state (context-specific, passed in at runtime)**.
- When building high-volume systems like **text editors** (with thousands of character glyphs), **map applications** (with repeated icons or tiles), or **game engines** (with many similar objects like trees or particles), developers often **instantiate huge numbers of objects** many of which are functionally identical.
- But this can lead to significant **performance issues, excessive memory allocation**, and poor scalability especially when most of these objects differ only by a few small, context-specific values.
- The **Flyweight Pattern** solves this by sharing common state (the **intrinsic part**) across all similar objects and externalizing unique data (the **extrinsic part**). It allows you to create **lightweight objects** by caching and reusing instances instead of duplicating data.

THE PROBLEM: RENDERING CHARACTERS

- Imagine you're building a rich text editor that needs to render characters on screen — much like Google Docs or MS Word.
- Every character (a, b, c, ..., z, punctuation, etc.) must be displayed with formatting information such as:
    - Font family
    - Font size
    - Color
    - Style (bold, italic, etc.)
    - Position (x, y on the screen)

```
class CharacterGlyph {
    private char symbol;        // e.g., 'a', 'b', etc.
    private String fontFamily;    // e.g., "Arial"
    private int fontSize;        // e.g., 12
    private String color;        // e.g., "#000000"
    private int x;              // position X
    private int y;              // position Y

    public CharacterGlyph(char symbol, String fontFamily, int fontSize, String color, int x, int y) {
```

```
        this.symbol = symbol;
        this.fontFamily = fontFamily;
        this.fontSize = fontSize;
        this.color = color;
        this.x = x;
        this.y = y;
    }

    public void draw() {
        System.out.println("Drawing '" + symbol + "' in " + fontFamily +
            ", size " + fontSize + ", color " + color + " at (" + x + "," + y + ")");
    }
}
```

WHY THIS IS A PROBLEM

1. **High Memory Usage**
   Each character glyph holds repeated data (font, size, color) — even though these are shared across thousands of characters. You're wasting memory by storing the same values over and over.
2. **Performance Bottleneck**
   Creating and managing a massive number of objects increases GC pressure, reduces cache performance, and may cause your app to lag on lower-end machines.
3. **Poor Scalability**
   Want to render an entire book or open multiple large documents? Memory usage will balloon out of control, and you'll hit limits quickly.

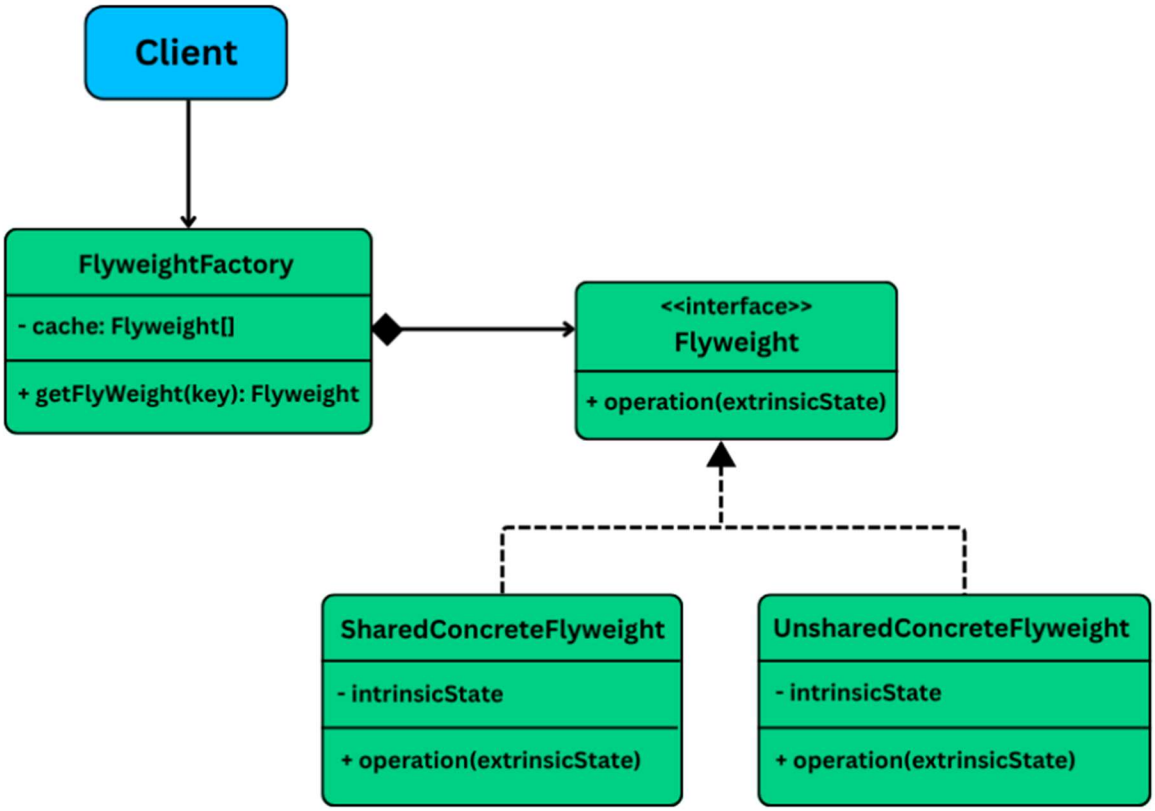
WHAT WE REALLY NEED

- Share the **formatting data** (font, size, color, etc.) among all similar characters
- Only store what's **truly unique** (like position or context) for each character
- **Avoid duplicating redundant** data while still rendering characters accurately

WHAT IS THE FLYWEIGHT PATTERN

- The **Flyweight Pattern minimizes memory usage by sharing as much data** as possible between similar objects.
- Instead of creating a new object for every instance — even when the data is the same — the Flyweight Pattern allows you to **reuse shared objects** (called flyweights) and externalize the state that differs between them.

Class Diagram



Flyweight Interface
Declares a method like draw(x, y) that takes extrinsic state (position)

ConcreteFlyweight
Implements the flyweight and stores intrinsic state like font and symbol

FlyweightFactory
Caches and reuses flyweights to avoid duplication

Client
Maintains extrinsic state and uses shared flyweights to perform operations

IMPLEMENTING FLYWEIGHT

- Let's implement the **Flyweight Pattern** to optimize how we render text in a document editor. Our goal is to share **common formatting properties** (font, size, color) across characters and **store only unique** data (like position) at the instance level.

1. DEFINE THE FLYWEIGHT INTERFACE

- The **flyweight interface** declares a method like **draw(x, y)** that renders a character on screen.
- Each **flyweight object** represents shared formatting (intrinsic state), but it expects position to be passed in when drawn.

```
interface CharacterFlyweight {
   void draw(int x, int y);
}
```

2. IMPLEMENT THE CONCRETE FLYWEIGHT

- This class holds **the intrinsic state** — the shared, repeatable properties like:

```
class CharacterGlyph implements CharacterFlyweight {
   private final char symbol;
   private final String fontFamily;
   private final int fontSize;
   private final String color;

   public CharacterGlyph(char symbol, String fontFamily, int fontSize, String color) {
      this.symbol = symbol;
      this.fontFamily = fontFamily;
      this.fontSize = fontSize;
      this.color = color;
   }

   @Override
   public void draw(int x, int y) {
      System.out.println("Drawing '" + symbol + "' [Font: " + fontFamily +
         ", Size: " + fontSize + ", Color: " + color + "] at (" + x + "," + y + ")");
   }
}
```

3. CREATE THE FLYWEIGHT FACTORY

- The factory ensures **flyweights are shared and reused**. It checks whether a flyweight with a given set of intrinsic values already exists and returns it, or creates a new one if it doesn't.

```
class CharacterFlyweightFactory {
   private final Map<String, CharacterFlyweight> flyweightMap = new HashMap<>();

   public CharacterFlyweight getFlyweight(char symbol, String fontFamily, int fontSize, String
color) {
      String key = symbol + fontFamily + fontSize + color;
```

```
        flyweightMap.putIfAbsent(key, new CharacterGlyph(symbol, fontFamily, fontSize,
color));
        return flyweightMap.get(key);
    }

    public int getFlyweightCount() {
        return flyweightMap.size();
    }
}
```

- This factory is **the heart of memory optimization**. It ensures no duplicate formatting objects are created.

4. CREATE THE CLIENT

- Retrieves flyweight objects from the factory
- Combines each flyweight with position-specific data (extrinsic state)
- Stores RenderedCharacter objects that contain a flyweight and coordinates

```
class TextEditorClient {
    private final CharacterFlyweightFactory factory = new CharacterFlyweightFactory();
    private final List<RenderedCharacter> document = new ArrayList<>();

    public void addCharacter(char c, int x, int y, String font, int size, String color) {
        CharacterFlyweight glyph = factory.getFlyweight(c, font, size, color);
        document.add(new RenderedCharacter(glyph, x, y));
    }

    public void renderDocument() {
        for (RenderedCharacter rc : document) {
            rc.render();
        }
        System.out.println("Total flyweight objects used: " + factory.getFlyweightCount());
    }

    private static class RenderedCharacter {
        private final CharacterFlyweight glyph;
        private final int x, y;

        public RenderedCharacter(CharacterFlyweight glyph, int x, int y) {
            this.glyph = glyph;
            this.x = x;
            this.y = y;
        }
```

```
    public void render() {
      glyph.draw(x, y);
    }
  }
}
```

MAIN METHOD

```
public class FlyweightDemo {
  public static void main(String[] args) {
    TextEditorClient editor = new TextEditorClient();

    // Render "Hello" with same style
    String word = "Hello";
    for (int i = 0; i < word.length(); i++) {
      editor.addCharacter(word.charAt(i), 10 + i * 15, 50, "Arial", 14, "#000000");
    }

    // Render "World" with different font and color
    String word2 = "World";
    for (int i = 0; i < word2.length(); i++) {
      editor.addCharacter(word2.charAt(i), 10 + i * 15, 100, "Times New Roman", 14, "#3333FF");
    }

    editor.renderDocument();
  }
}
```

WHAT WE ACHIEVED

- **Memory efficiency**: Shared formatting data eliminates duplication
- **Improved performance**: Fewer objects = faster rendering and lower GC pressure
- **Separation of concerns**: Formatting logic and position/context are cleanly separated
- **Reusability**: Glyphs for common characters are reused across the document
- **Scalability**: Can handle thousands of characters with minimal memory footprint