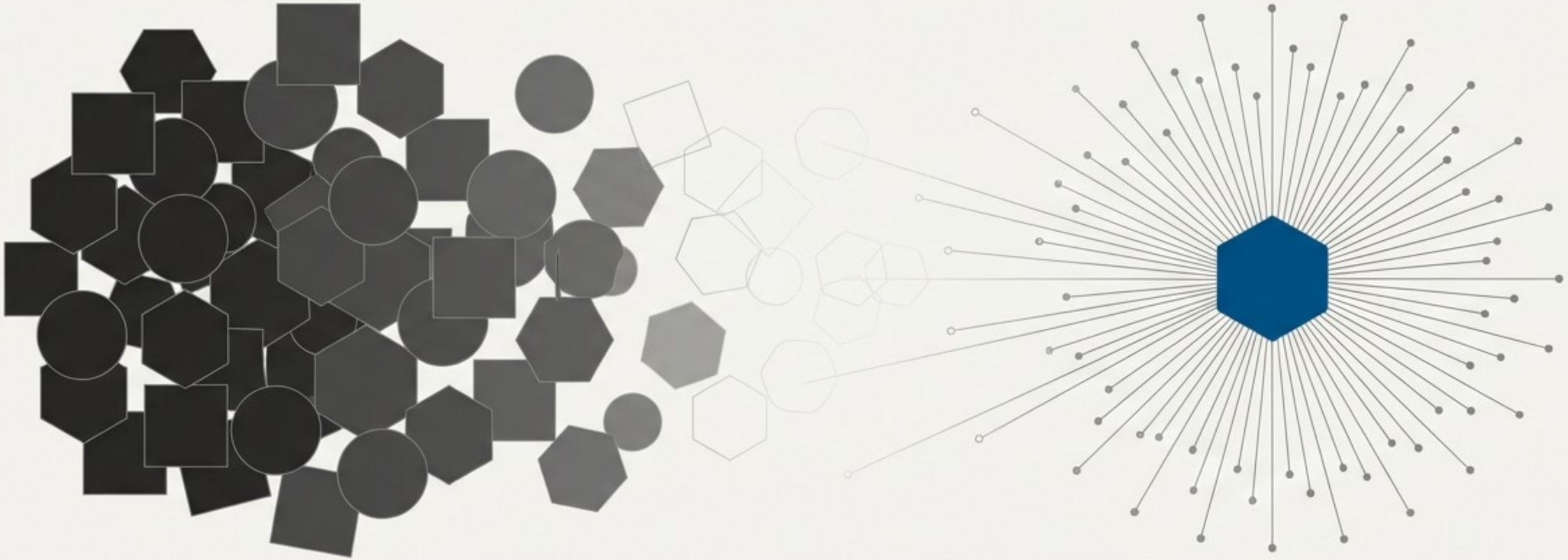
Outsmarting the Silent Memory Thief

A Practical Guide to the Flyweight Design Pattern



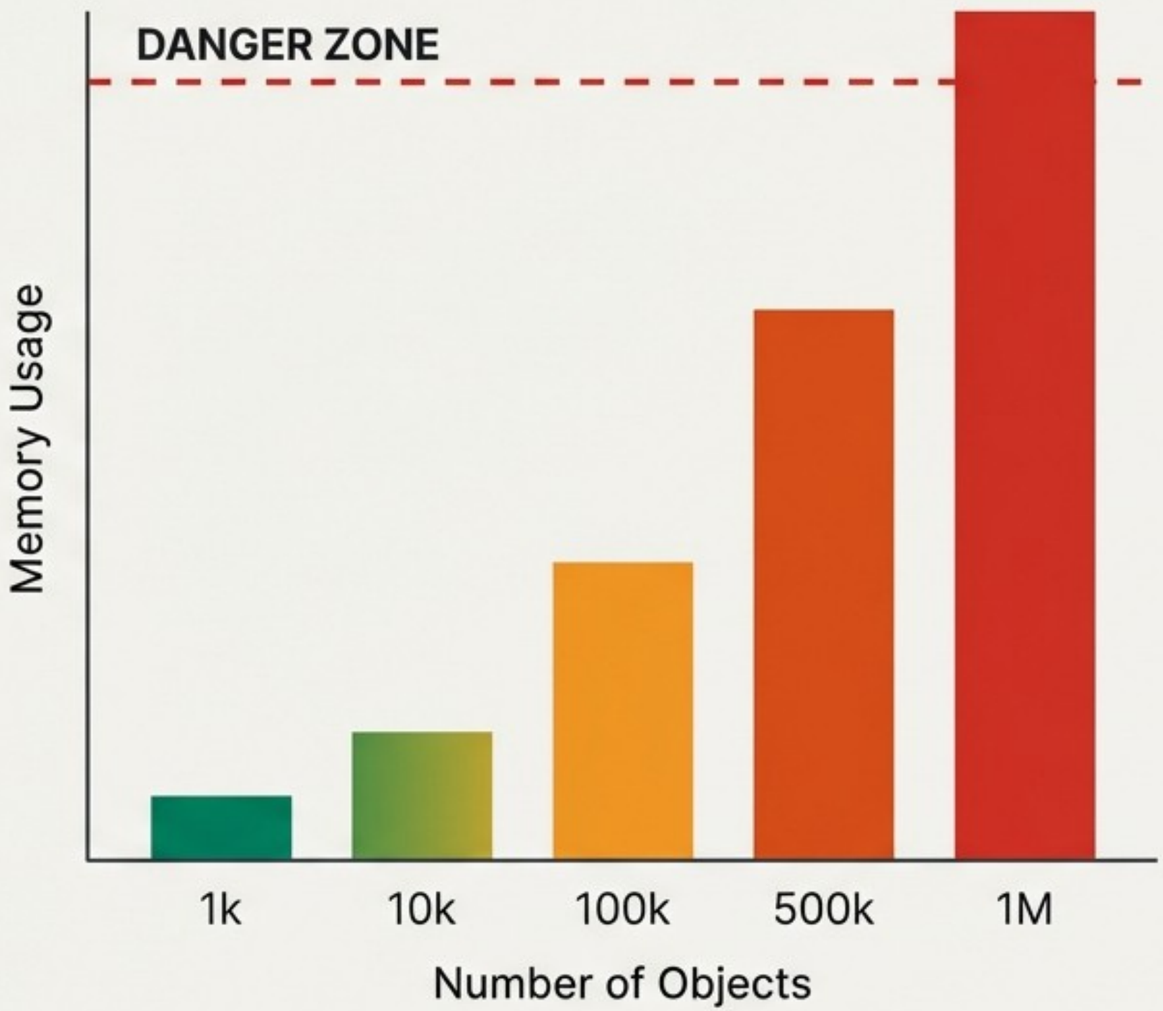
How to manage millions of objects without exploding your memory.

The Problem: When Millions of Similar Objects Attack Your RAM

Your application needs to create a vast number of objects that are almost identical.

Most of these objects share the same core data (e.g., colour, texture, type, style), but each also has a small amount of unique data (e.g., position, current state).

> **Duplicating all this shared data for every single object is a recipe for disaster. Memory explodes** 🚨



Game Trees Map Tiles Text Characters UI Icons Product Attributes

An Everyday Analogy: The Airline Seat Map

The Seat Model

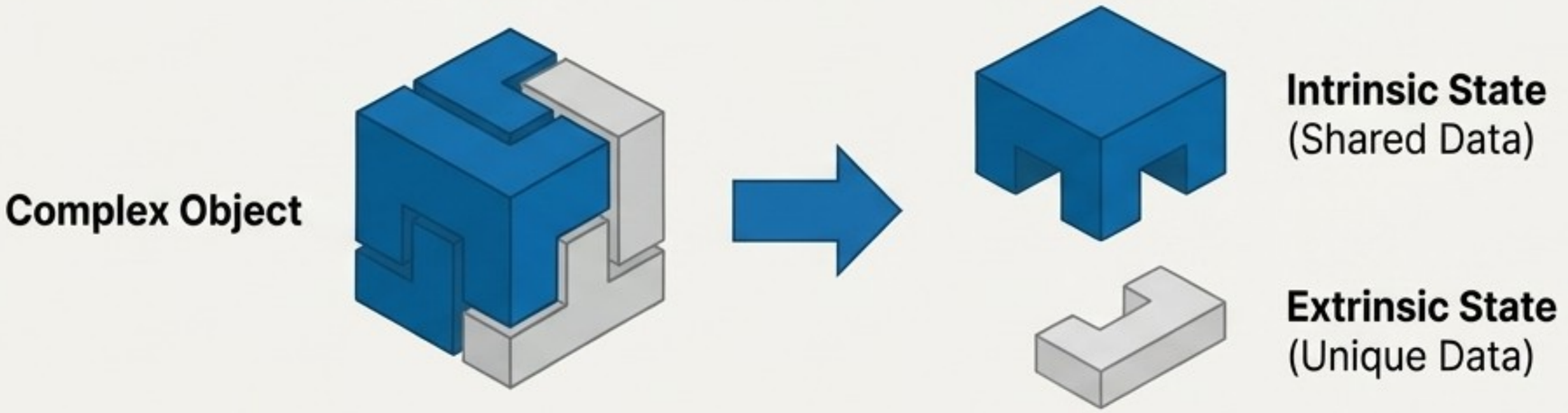
The fundamental design is the same for hundreds of seats. They share the same shape, material, and description. This is manufactured once.

The Seat Assignment

What makes each seat unique is simply its number and location (12A, 12B, 13A). This is lightweight, external information.

Airlines don’t build a custom seat for every passenger. They reuse a standard model and just attach a unique identifier. **This is the core idea of Flyweight.**

Splitting the Data: Intrinsic vs. Extrinsic State



Intrinsic State

The shared, reusable data that is constant across many objects. It lives inside the Flyweight object.

- From our Analogy: The Seat Model (shape, material).
- From a Code Example: Tree Type, Colour, Texture.

Extrinsic State

The unique, external data that varies for each object. It is passed in by the client and lives outside the Flyweight.

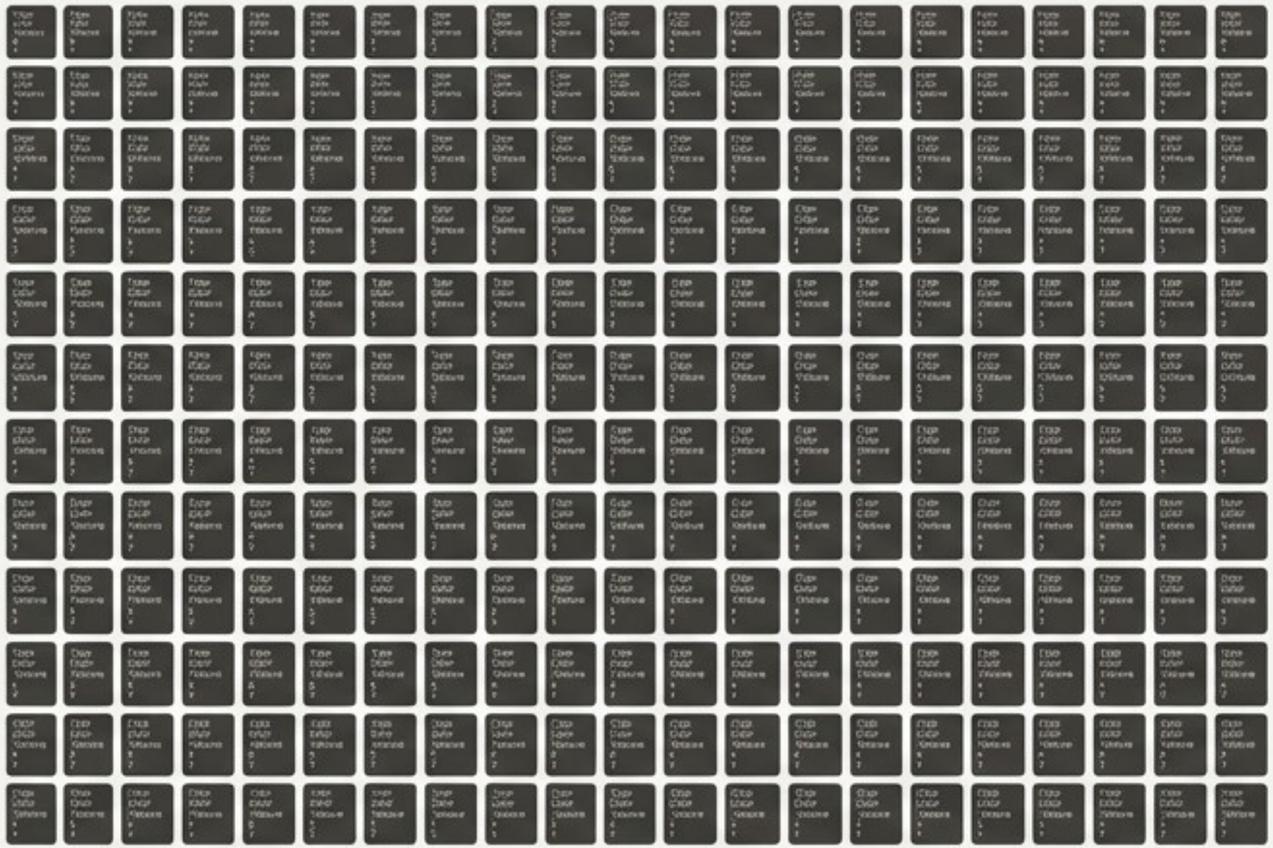
- From our Analogy: The Seat Number (12A, 12B).
- From a Code Example: X/Y Coordinates, current state.

We store the shared (Intrinsic) data **ONCE** and pass the unique (Extrinsic) data as needed.

The Naive Approach: Duplicating Everything

Let's represent thousands of trees in a forest scene.

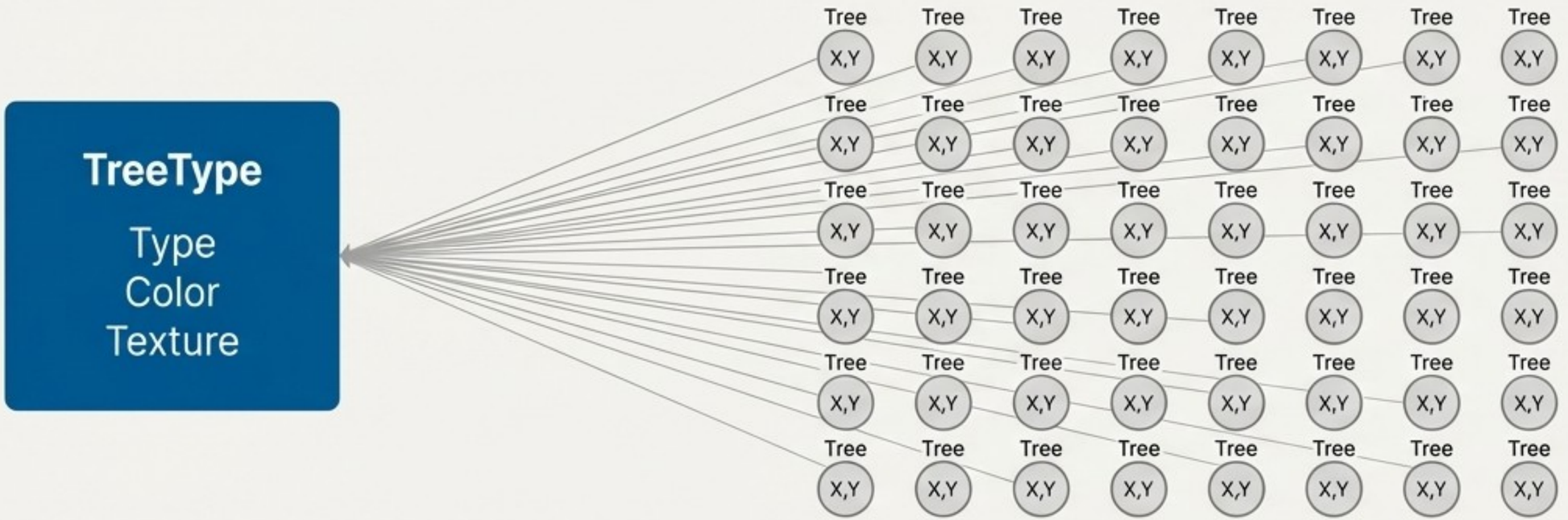
```
public class Tree
{
    // --- Repeated for EVERY tree ---
    public string Type;    // "Oak"
    public string Color;   // "Green"
    public string Texture; // "Rough"
    // -------------------------------

    // --- Unique for every tree ---
    public int X;
    public int Y;
    // -------------------------------
}
```



With 100,000 trees, the same Type, Colour, and Texture data is duplicated 100,000 times. **This is a huge waste of memory.**

The Flyweight Solution: Share, Don’t Duplicate

We split the Tree object into two parts.



Key Component 1: The Flyweight (TreeType)

- Holds the **intrinsic** (shared) state: Type, Color, Texture.
- There will only be one instance of each unique TreeType.

Key Component 2: The Context (Tree)

- Holds the **extrinsic** (unique) state: X, Y coordinates.
- Contains a reference to its shared TreeType object.

Memory saved dramatically. The shared data is stored only once per tree type.

Code Deconstructed — Step 1: The Flyweight Object

This class holds the shared, intrinsic state. It is the reusable component.

```
// The Flyweight: Stores data shared by many trees
public class TreeType
{
    public string Name;
    public string Color;
    public string Texture;

    public TreeType(string name, string color, string texture)
    {
        Name = name;
        Color = color;
        Texture = texture;
    }

    // The operation also receives extrinsic state
    public void Draw(int x, int y)
    {
        Console.WriteLine($"Drawing {Name} at ({x},{y})");
    }
}
```

Intrinsic State: Immutable and shared across all trees of this type.

Extrinsic State (`x`, `y`) is passed in when needed. It is not stored here.

Code Deconstructed — Step 2: The Flyweight Factory

This class ensures that a Flyweight is created only once and then reused.

```
public static class TreeFactory
{
    private static readonly Dictionary<string, TreeType> _types = new();

    public static TreeType GetTreeType(string name, string color, string texture)
    {
        var key = $"{name}-{color}-{texture}";
        if (!_types.ContainsKey(key))
        {
            _types[key] = new TreeType(name, color, texture);
        }
        return _types[key];
    }
}
```

The Cache: A pool of existing `TreeType` objects. This is the key to sharing.

Create-on-demand: A new `TreeType` is created only if it doesn't already exist in our cache.

Code Deconstructed — Step 3: The Lightweight Context

This class holds the unique, extrinsic state and a reference to its shared Flyweight.

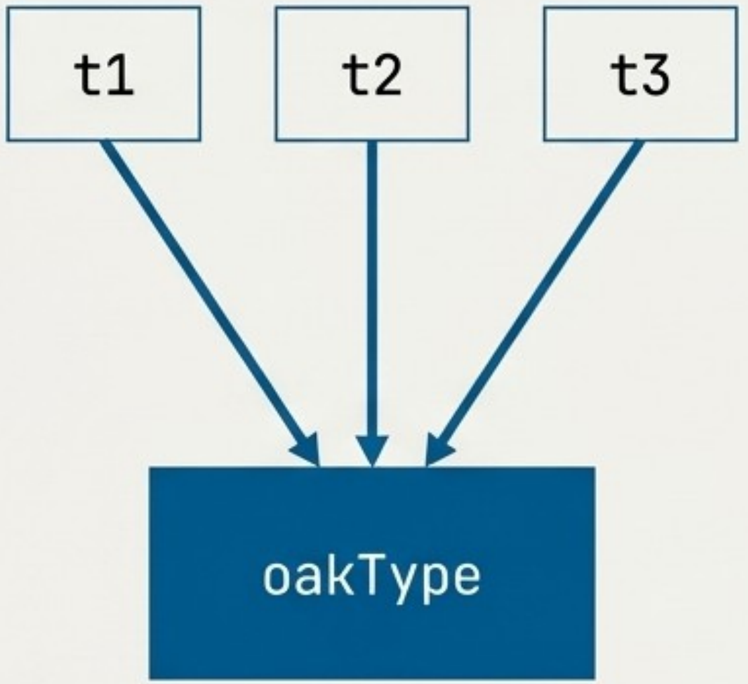
```
// The Context: Stores only the unique state
public class Tree
{
    // --- Extrinsic State ---
    private readonly int _x;
    private readonly int _y;
    // ----------------------

    // Reference to the shared Flyweight
    private readonly TreeType _type;

    public Tree(int x, int y, TreeType type)
    {
        _x = x;
        _y = y;
        _type = type;
    }

    public void Draw()
        => _type.Draw(_x, _y);  // Delegates drawing to the Flyweight
}
```

Unique State: Only the coordinates are stored per `Tree` instance.

Delegation: The `Tree` object passes its unique state to the shared `TreeType` to perform the action.

Putting It All Together: The Client's View

```
// 1. Request a flyweight from the factory
var oakType = TreeFactory.GetTreeType("Oak", "Green", "Rough");

// 2. Create multiple lightweight context objects
//    using the SAME flyweight
var t1 = new Tree(10, 20, oakType);
var t2 = new Tree(15, 25, oakType);
var t3 = new Tree(20, 30, oakType);

// 3. Each tree draws itself, passing its unique state
t1.Draw(); // Drawing Oak at (10,20)
t2.Draw(); // Drawing Oak at (15,25)
t3.Draw(); // Drawing Oak at (20,30)
```



The `TreeType` for "Oak" is stored only once. Each `Tree` object is extremely lightweight, containing only its coordinates and a reference.

The Payoff: Where Flyweight Shines in the Real World

This pattern is a crucial optimization in systems that handle massive scale.

Text Editors

Every character (a, b, c) doesn't store its font, size, and style. It shares a common style object. Only its position and the character itself are unique.

Maps & Games

Millions of tiles, trees, or particles reuse a small set of models, textures, and icons to build vast worlds efficiently.

Product Catalogs

A 'T-Shirt' product model (with description, brand) is shared across all its variants (Small/Blue, Medium/Red), which only store the unique attributes.

UI Frameworks

Common UI elements like buttons or icons share a single style definition (like a CSS class). Only their position and label are unique.

Applicability: When Should You Use the Flyweight Pattern?

Use this checklist to see if Flyweight is the right tool for your problem.

- [x] You have a very large number of similar objects.
  The application must create thousands or millions of instances (e.g., particles, characters, tiles).
- [x] The identity of each object doesn't matter.
  Clients don't care about specific object instances, only their state.
- [x] A significant portion of the object's state can be made intrinsic (shared).
  Most fields are identical across objects and can be extracted.
- [x] Memory consumption is a genuine bottleneck.
  RAM usage grows unacceptably fast with the number of objects.

Caution: When to Avoid the Flyweight Pattern

Flyweight is an optimisation. Don’t apply it prematurely if it’s not needed.

The number of objects is small.

The complexity of setting up the pattern outweighs the minor memory savings.

Objects differ too much to share state.

If there is very little intrinsic state to extract, the pattern has no benefit.

Memory is not a real or projected problem.

“Premature optimization is the root of all evil.”

Object identity is important.

If clients need to distinguish between individual object instances, sharing can cause issues.

The Flyweight Pattern: A Cheat Sheet

Problem	An application with too many similar objects wastes a huge amount of memory by duplicating shared data.
Solution	Share the common data (intrinsic state) between objects and keep only the unique data (extrinsic state) in each object.
Key Idea	Split an object's state into **Intrinsic** (shared, immutable) and **Extrinsic** (unique, context-dependent).
Core Benefits	Drastic memory savings and potential performance improvements due to fewer object allocations.
Used In	Games, maps, text editors, UI frameworks, and large-scale data rendering.
Principles	Single Responsibility Principle (SRP), Open/Closed Principle (OCP), and memory optimisation through sharing.