HowToDoInJava

Flyweight Design Pattern

Last Updated: August 26, 2021 By: Lokesh Gupta Structural Patterns Design Patterns

As per GoF definition, **flyweight design pattern** enables use sharing of objects to support large numbers of fine-grained objects efficiently. A flyweight is a **shared object** that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context.

1. When to use flyweight design pattern

We can use flyweight pattern in following scenarios:

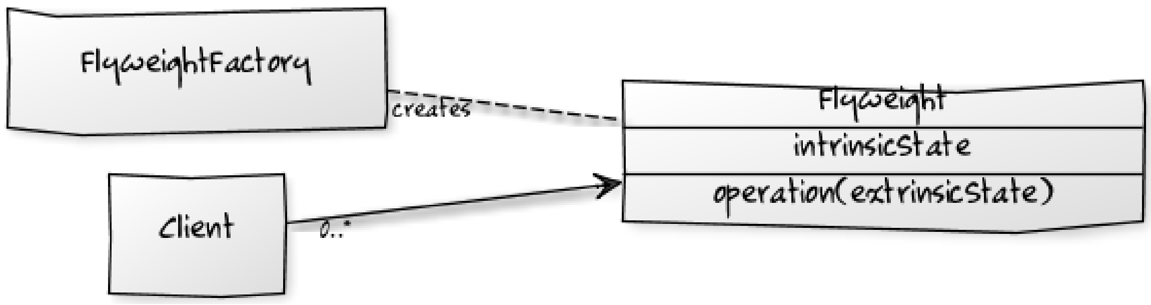
- When we need a large number of similar objects that are unique in terms of only a few parameters and most of the stuffs are common in general.
- We need to control the memory consumption by large number of objects – by creating fewer objects and sharing them across.

2. Extrinsic and intrinsic attributes

A flyweight objects essentially has two kind of attributes – intrinsic and extrinsic.

An **intrinsic** state attribute is stored/shared in the flyweight object, and it is independent of flyweight's context. As the best practice, we should make intrinsic states immutable.

An **extrinsic** state varies with flyweight's context, which is why they cannot be shared. Client objects maintain the extrinsic state, and they need to pass this to a flyweight object during object creation.



Flyweight Pattern

3. Real world example of flyweight pattern

- Suppose we have a **pen** which can exist with/without **refill**. A refill can be of any color thus a pen can be used to create drawings having N number of colors. Here `Pen` can be flyweight object with `refill` as extrinsic attribute. All other attributes such as pen body, pointer etc. can be intrinsic attributes which will be common to all pens. A pen will be distinguished by its refill color only, nothing else.

  All application modules which need to access a red pen – can use the same instance of red pen (shared object). Only when a different color pen is needed, application module will ask for another pen from flyweight factory.

- In programming, we can see **java.lang.String** constants as flyweight objects. All strings are stored in string pool and if we need a string with certain content then runtime return the reference to already existing string constant from the pool – if available.
- In browsers, we can use an image in multiple places in a webpage. Browsers will load the image only one time, and for other times browsers will reuse the image from cache. Now image is same but used in multiple places. It's URL is intrinsic attribute because it's fixed and shareable. Images position coordinates, height and width are extrinsic attributes which vary according to place (context) where they have to be rendered.

4. Flyweight design pattern example

In given example, we are building a Paint Brush application where client can use brushes on three types – THICK, THIN and MEDIUM. All the thick (thin or medium) brush will draw the content in exact similar fashion – only the content color will be different.

```
Pen.java

public interface Pen
{
  public void setColor(String color);
  public void draw(String content);
}
```

```
BrushSize.java

public enum BrushSize {
  THIN, MEDIUM, THICK
}
```

```
ThickPen.java

public class ThickPen implements Pen {

  final BrushSize brushSize = BrushSize.THICK;  //intrinsic state - shareable
  private String color = null;          //extrinsic state - supplied by client

  public void setColor(String color) {
    this.color = color;
  }

  @Override
  public void draw(String content) {
    System.out.println("Drawing THICK content in color : " + color);
  }
}
```

```
ThinPen.java

public class ThinPen implements Pen {

  final BrushSize brushSize = BrushSize.THIN;
  private String color = null;
  public void setColor(String color) {
    this.color = color;
  }

  @Override
  public void draw(String content) {
    System.out.println("Drawing THIN content in color : " + color);
  }
}
```

```
MediumPen.java

public class MediumPen implements Pen {

  final BrushSize brushSize = BrushSize.MEDIUM;
  private String color = null;

  public void setColor(String color) {
    this.color = color;
  }

  @Override
  public void draw(String content) {
    System.out.println("Drawing MEDIUM content in color : " + color);
  }
}
```

Here brush `color` is extrinsic attribute which will be supplied by client, else everything will remain same for the **Pen**. So essentially, we will create a pen of certain size only when the color is different. Once another client or context need that pen size and color, we will reuse it.

```
PenFactory.java

import java.util.HashMap;

public class PenFactory
{
  private static final HashMap<String, Pen> pensMap = new HashMap<>();

  public static Pen getThickPen(String color)
  {
    String key = color + "-THICK";

    Pen pen = pensMap.get(key);
```

```
      if(pen != null) {
        return pen;
      } else {
        pen = new ThickPen();
        pen.setColor(color);
        pensMap.put(key, pen);
      }

      return pen;
    }

    public static Pen getThinPen(String color)
    {
      String key = color + "-THIN";

      Pen pen = pensMap.get(key);

      if(pen != null) {
        return pen;
      } else {
        pen = new ThinPen();
        pen.setColor(color);
        pensMap.put(key, pen);
      }

      return pen;
    }

    public static Pen getMediumPen(String color)
    {
      String key = color + "-MEDIUM";

      Pen pen = pensMap.get(key);

      if(pen != null) {
        return pen;
      } else {
        pen = new MediumPen();
        pen.setColor(color);
        pensMap.put(key, pen);
      }

      return pen;
    }
  }
```

Let's test the flyweight pen objects using a client. The client here creates three THIN pens, but in runtime their is only one pen object of thin type and it's shared with all three invocations.

```
PaintBrushClient.java

public class PaintBrushClient
{
  public static void main(String[] args)
  {
    Pen yellowThinPen1 = PenFactory.getThickPen("YELLOW");  //created new pen
    yellowThinPen1.draw("Hello World !!");

    Pen yellowThinPen2 = PenFactory.getThickPen("YELLOW");  //pen is shared
    yellowThinPen2.draw("Hello World !!");

    Pen blueThinPen = PenFactory.getThickPen("BLUE");   //created new pen
    blueThinPen.draw("Hello World !!");

    System.out.println(yellowThinPen1.hashCode());
    System.out.println(yellowThinPen2.hashCode());

    System.out.println(blueThinPen.hashCode());
  }
}
```

Program output.

```
Console

Drawing THICK content in color : YELLOW
Drawing THICK content in color : YELLOW
Drawing THICK content in color : BLUE

2018699554    //same object
2018699554    //same object
1311053135
```

5. FAQs

5.1. Difference between singleton pattern and flyweight pattern

The singleton pattern helps we maintain only one object in the system. In other words, once the required object is created, we cannot create more. We need to reuse the existing object in all parts of the application.

The flyweight pattern is used when we have to create large number of similar objects which are different based on client provided extrinsic attribute.

5.2. Effect of concurrency on flyweights

Similar to singleton pattern, if we create flyweight objects in concurrent environment, we may end up having multiple instances of same flyweight object which is not desirable.

To fix this, we need to use **double checked locking** as used in singleton pattern while creating flyweights.

5.3. Benefits of flyweight design pattern

Using flyweights, we can –

- reduce memory consumption of heavy objects that can be controlled identically.
- reduce the total number of "complete but similar objects" in the system.
- provide a centralized mechanism to control the states of many "virtual" objects.

5.4. Is intrinsic and extrinsic data shareable?

The intrinsic data is shareable as it is common to all contexts. The extrinsic data is not shared. Client need to pass the information (states) to the flyweights which is unique to it's context.

5.5. Challenges of flyweight pattern

- We need to take the time to configure these flyweights. The design time and skills can be overhead, initially.
- To create flyweights, we extract a common template class from the existing objects. This additional layer of programming can be tricky and sometimes hard to debug and maintain.
- The flyweight pattern is often combined with singleton factory implementation and to guard the singularity, additional cost is required.

Drop me your questions related to **flyweight pattern** in comments.

Happy Learning !!

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. Decorator Design Pattern in Java
2. Adapter Design Pattern in Java
3. Bridge Design Pattern
4. Composite Design Pattern
5. Facade Design Pattern
6. Proxy Design Pattern
7. Visitor Design Pattern Example
8. Template Method Design Pattern
9. Command Design Pattern
0. Iterator Design Pattern

4 thoughts on "Flyweight Design Pattern"

kiran

March 13, 2020 at 8:12 am

Hii lokesh we can make extrinsic attribute immutable. In multi threading environment every thread will be sure same extrinsic object they are working with no other threads modifies it.

Private final String color;

MediumPen (String color){
this.color=color;
}

Using above approach instead of setter metter for setting extrinsic attribute.

Reply

Michele Arpaia

December 7, 2019 at 3:25 am

Hi there,
The issue I am facing is that you need to know a compile-time the name of all the flyweights objects – there are cases that that decision is taken at runtime. What would you suggest here? generating and compiling classes at runtime?

Reply

Abdul

March 18, 2019 at 11:58 am

hi Lokesh,

Thanks for your time to create this post.

Suppose I have created a system using the exact logic you provided.

Suppose its a multi threaded app,

Thread1: T1 requested YELLOW pen object and factory returns it successfuly.

Thread2: T2 requested YELLOW pen object(but T1 has not completed its work) and factory returned the object which is being used by T1.

My Solution: If I synchronize the below method that should solve my concurrency problems right?

```
@Override
public void draw(String content) {
System.out.println("Drawing THICK content in color : " + color);
}
```

Reply

Lokesh Gupta

March 18, 2019 at 12:52 pm

That makes sense.

Reply

Leave a Comment

Name *

Email *

Website

☐ Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...

servicenow.

When DVSA boosts self-service use by 170%, the world works.

Read Case Study

HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial