CptS 487
Software Design and Architecture

Lesson 26

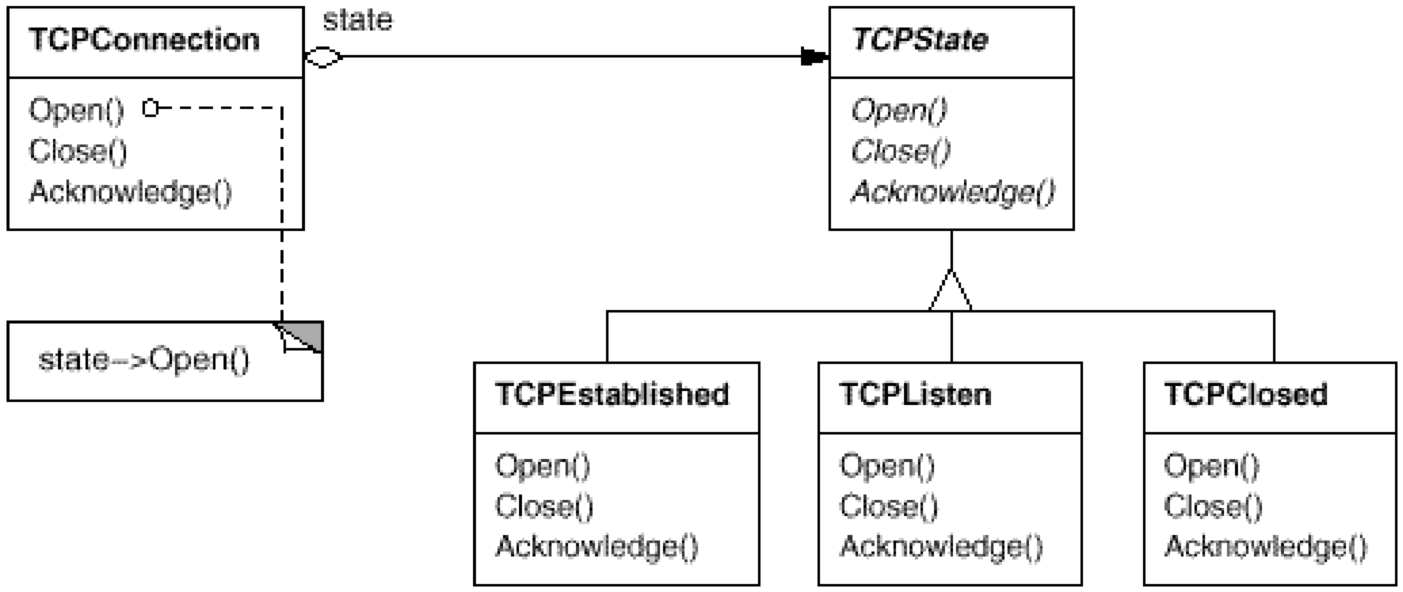
Design Patterns 10:

State & Flyweight

Instructors:

3. State (Object behavioral pattern)

- Intent
  - Allow an object to alter its behavior when its internal state changes.
- Motivation



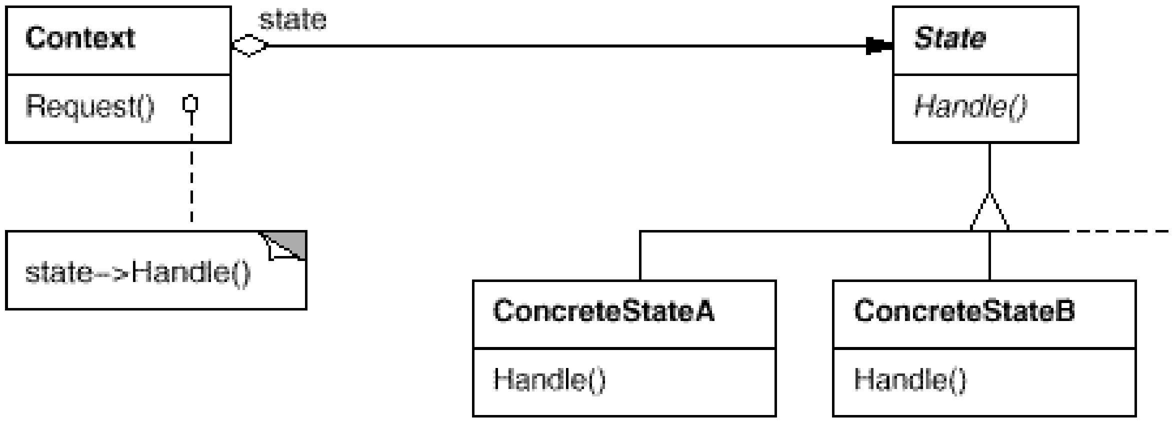
3. State

- Applicability

Use the State pattern whenever:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
- Operations have large, multipart conditional statements that depend on the object's state. The State pattern puts each branch of the conditional in a separate class.

- Structure

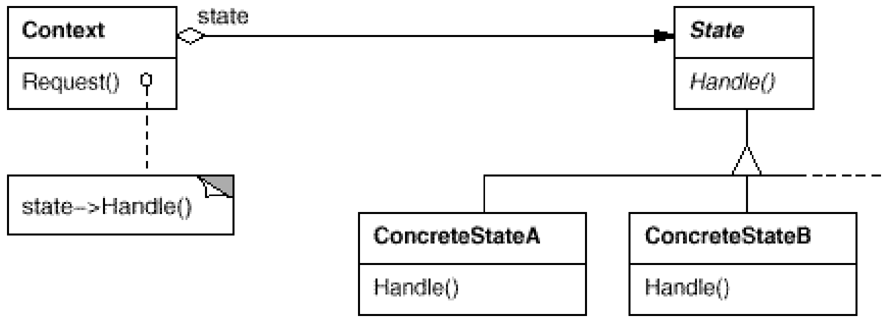


3. State

- Note the similarities between the State and Strategy patterns! The difference is one of intent.
  - A State object encapsulates a state-dependent behavior (and possibly state transitions)
    - Two key points not shown in the structure:
      - 1. State objects are usually aware of the existence of other states! (why?)
      - 2. State objects also usually deal with state transitions! (how?)
  - A Strategy object encapsulates an algorithm
- And they are both examples of Composition with Delegation!

State Pattern Example

```
//Context
public class MP3PlayerContext
{
    private State state;

    private MP3PlayerContext(State state)
   {
     this.state= state;
  }
   public void play()
  {
      state.pressPlay(this);
  }

    public void setState(State state)
   {
      this.state = state;
 }

    public State getState()
 {
      return state;
   }

}
```

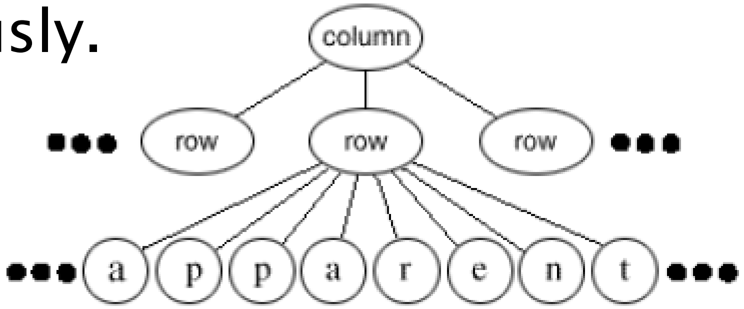


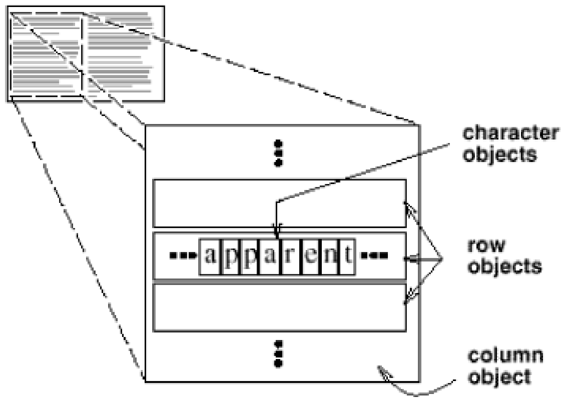
```
private interface State
{
    public void pressPlay(MP3PlayerContext context);
}
```

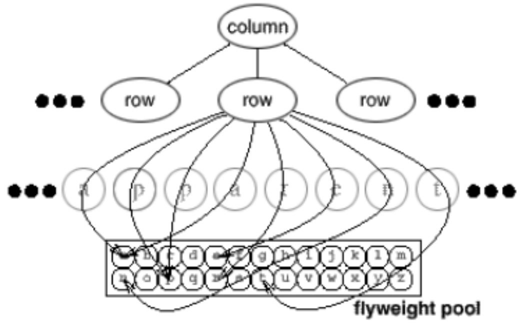
```
public class StandbyState implements State
{
 public void pressPlay(MP3PlayerContext context)
 {
       context.setState(new PlayingState());
   }

}


public class PlayingState implements State
{
    public void pressPlay(MP3PlayerContext context)
 {
       context.setState(new StandbyState());
   }

}
```

7. Flyweight (Object structural pattern)

- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Motivation
  - A *flyweight* is a shared object that can be used in multiple contexts simultaneously.
  - *Intrinsic* and *Extrinsic* state.

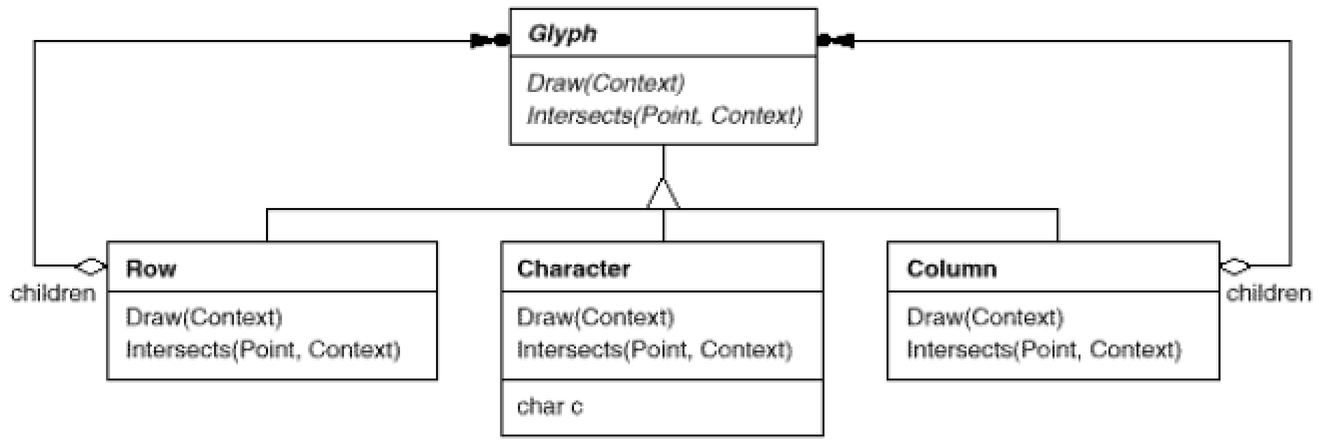






7. Flyweight (Object structural pattern)

- Motivation
  - A *flyweight* is a shared object that can be used in multiple contexts simultaneously.
  - *Intrinsic* and *Extrinsic* state.


Glyph
Draw(Context)
Intersects(Point, Context)
Row
Draw(Context)
Intersects(Point, Context)
children
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children

Flyweight Example - Font

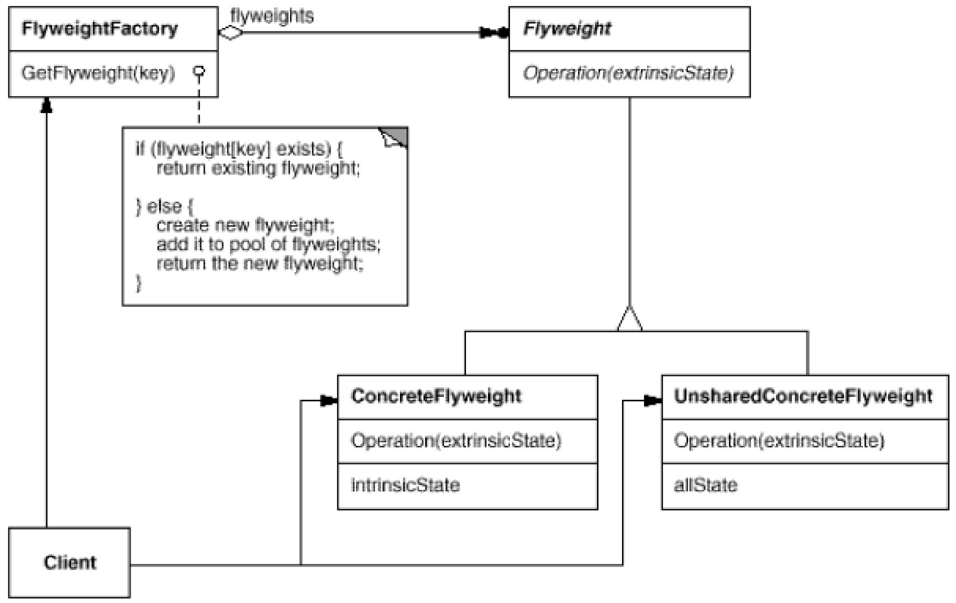
- See the "string": AABBZZ
- Each character contains two information:
  - The letter: 'A' or 'B' or 'Z'
  - The size: (in order) 18, 34, 24, 55, 12, 34
- Consider a class "TextRenderer" that renders the string, say in a Text Editor. It accepts a series of inputs like this: <'A', 18>, <'A', 34>, <'B', 24>, … etc., and then renders them on to the screen.
  - "TextRenderer" needs to create the "Character" objects with "letter" and "size" attributes.
  - "Character" class implements the "draw()" method. The method would draw the shape of the letter on the screen, accordingly to the "letter" and "size" attributes.
    - Think of "letter" as a shape, i.e. pixels to be drawn, and assume that the shape remains the same regardless of size.
- Question:
  - What (information) can be shared among the characters? What can not?
  - How many <u>objects</u> do we need? How can we reduce the number?

Reference Link

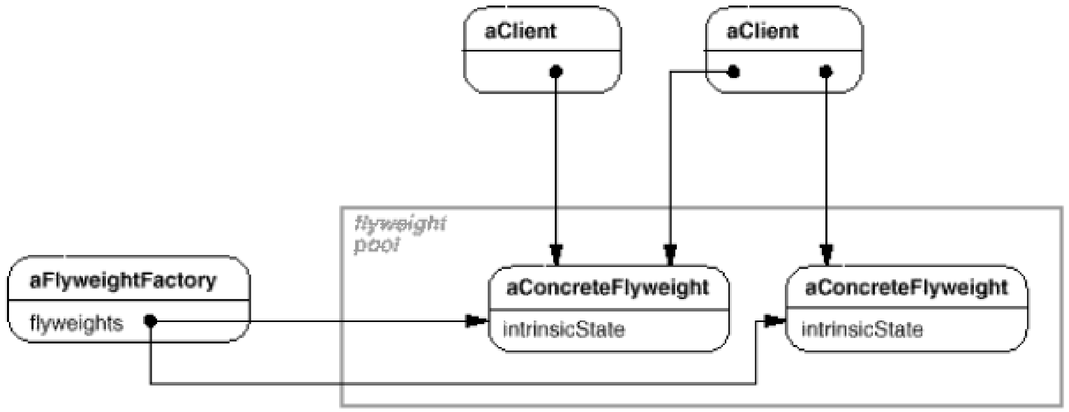
- http://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm

7. Flyweight (Object structural pattern)

- Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

```
if (flyweight[key] exists) {
    return existing flyweight;
} else {
    create new flyweight;
    add it to pool of flyweights;
    return the new flyweight;
}
```



7. Flyweight (Object structural pattern)

- Applicability
  - Apply when *all* of the following are true:
    - *An application uses a large number of objects.*
    - *Storage costs are high because of the sheer quantity of objects.*
    - *Most object state can be made extrinsic.*
    - *Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.*
    - *The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.*

7. Flyweight (Object structural pattern)

- Participants
  - Flyweight
    - Declares an interface through which flyweights can receive and act on extrinsic state.
  - ConcreteFlyweight
    - Implements the Flyweight interface and adds storage for intrinsic state, if any. Must be sharable.
  - UnsharedConcreteFlyweight
  - FlyweightFactory
    - Creates and manages flyweight object.
    - Ensures that flyweights are shared properly.
  - Client
    - Maintains a reference to flyweight(s).
    - Computes or stores the extrinsic state of flyweight(s).

7. Flyweight (Object structural pattern)

- Consequences
  - Run-time costs vs. Storage saving.
  - Costs:
    - Introduced by transferring, finding, and/or computing extrinsic state.
  - Storage saving:
    - The reduction in the total number of instances that comes from sharing.
    - The amount of intrinsic state per object.
    - Whether extrinsic state is computed or stored.
- Flyweight pattern often combined with the Composite pattern to represent a hierarchical structure as a graph with shared leaf nodes.

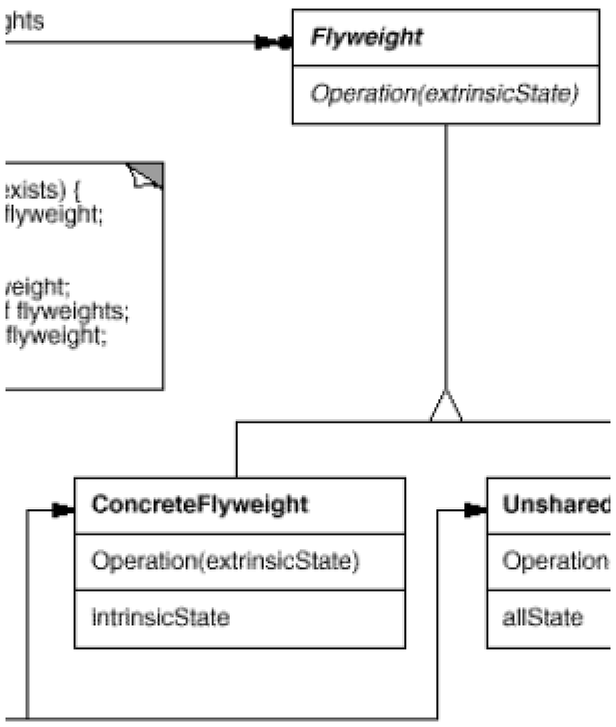
7. Flyweight (Object structural pattern)

- Implementation Issues
  - Removing extrinsic state.
  - Managing shared objects.

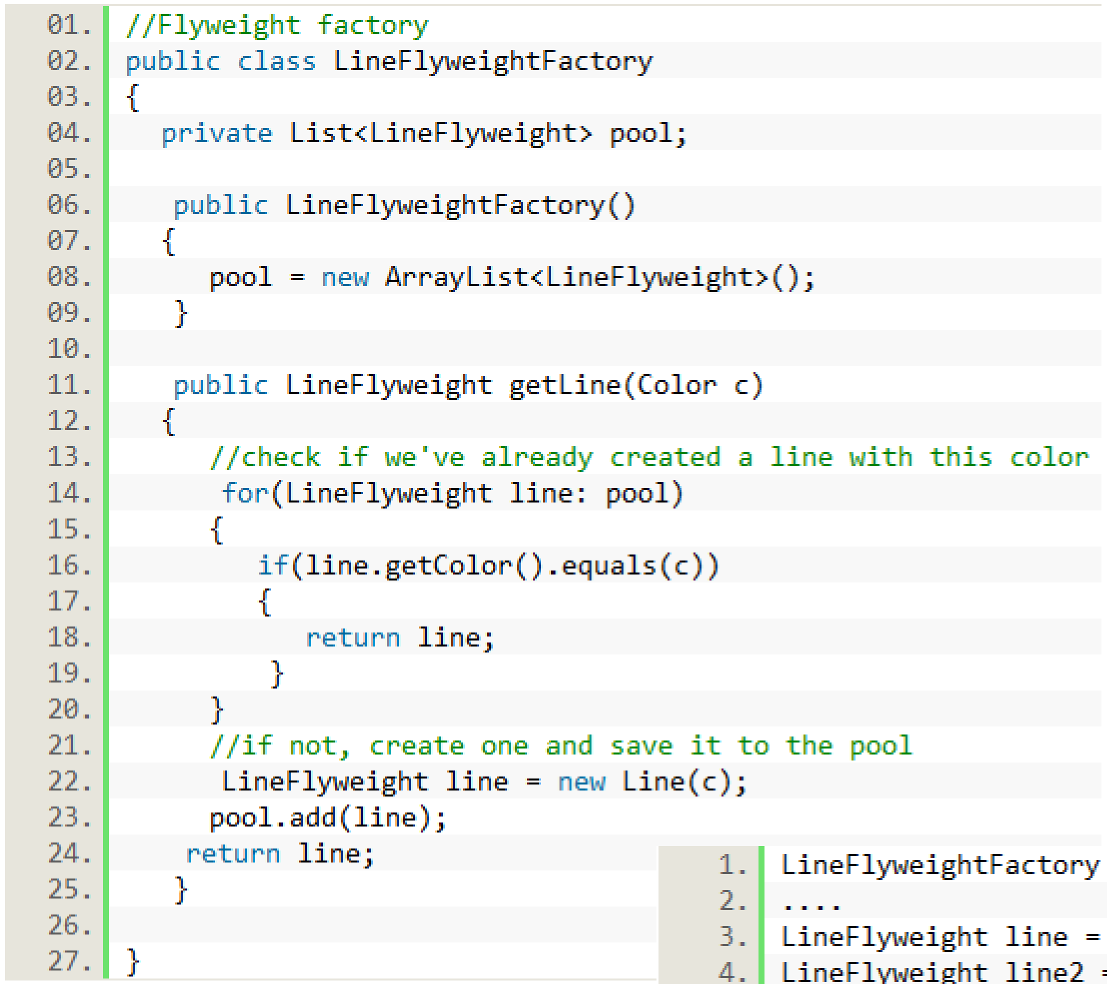
Flyweight Example

```java
//Flyweight
public interface LineFlyweight
{
  public Color getColor();
    public void draw(Point location);
}
```

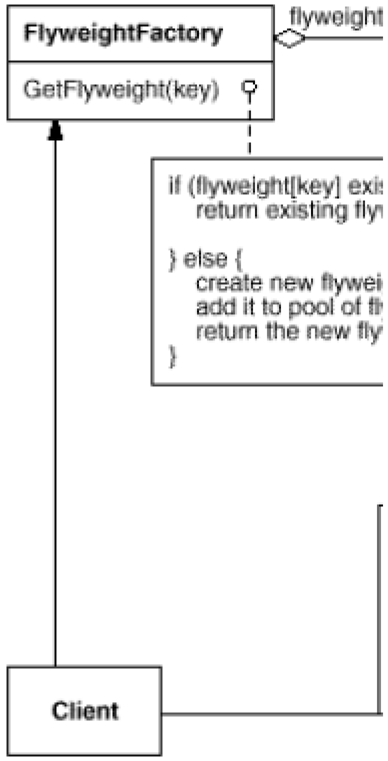
```java
//ConcreteFlyweight
public class Line implements LineFlyweight
{
  private Color color;

    public Line(Color c)
    {
      color = c;
  }

   public Color getColor()
 {
      return color;
   }

    public void draw(Point location)
    {
      //draw the character on screen
  }

}
```

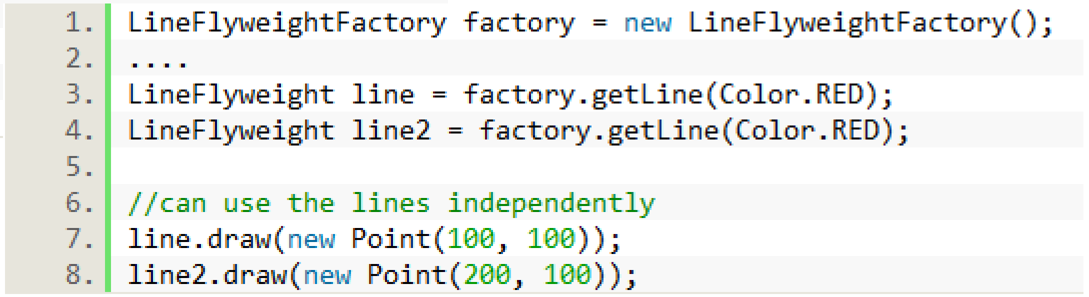


Flyweight Example



```java
//Flyweight factory
public class LineFlyweightFactory
{
   private List<LineFlyweight> pool;

    public LineFlyweightFactory()
   {
       pool = new ArrayList<LineFlyweight>();
    }

    public LineFlyweight getLine(Color c)
   {
       //check if we've already created a line with this color
        for(LineFlyweight line: pool)
       {
           if(line.getColor().equals(c))
           {
               return line;
            }
       }
       //if not, create one and save it to the pool
        LineFlyweight line = new Line(c);
       pool.add(line);
     return line;
    }

}
```





```java
LineFlyweightFactory factory = new LineFlyweightFactory();
....
LineFlyweight line = factory.getLine(Color.RED);
LineFlyweight line2 = factory.getLine(Color.RED);

//can use the lines independently
line.draw(new Point(100, 100));
line2.draw(new Point(200, 100));
```