الجامعة اللبنانية
Lebanese University

Graphical Interface and Application(I3305)

Chapter 2: Structural Patterns

Lebanese University
Faculty of Science 1 - Department of Computer Science
Dr. Abed EL Safadi

Outline

- Introduction
- Creational patterns
- Structural patterns
  - Flyweight
  - Bridge
  - Composite
- Behavioral patterns

What is the Flyweight Design Patterns?

- Used when you need to create a large number of similar objects

- To reduce memory usage you share Objects that are similar in some way rather than creating new ones

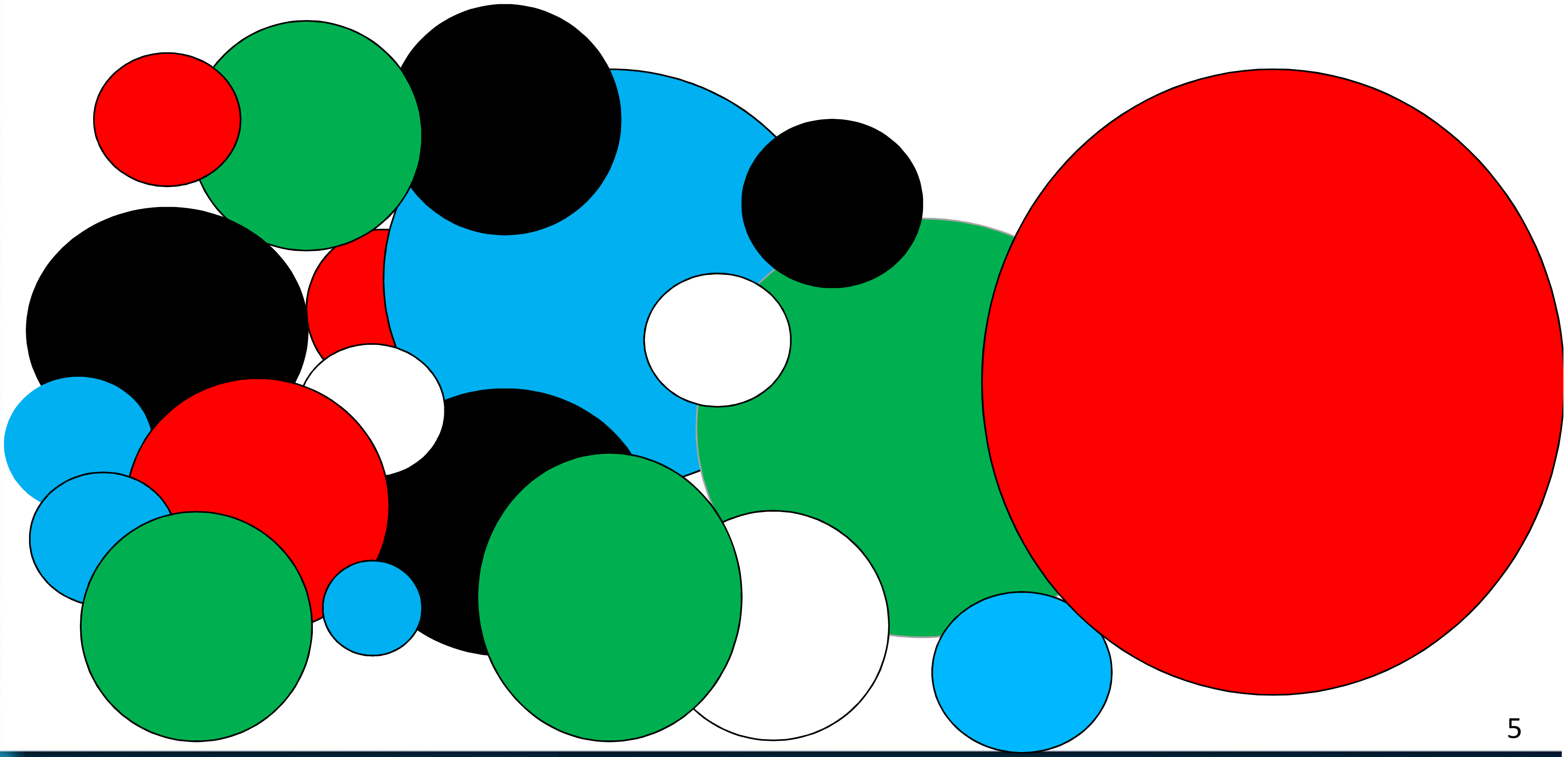
What is the Flyweight Design Patterns?

Real-Life Example

- In all real-world business applications, we want to avoid storing similar objects. The concept of this pattern is applicable in those places.
- Story : a few years ago, two friends were each searching for an apartment to stay nearby their office. However, neither of them was satisfied with the available options. Then, one day, they found a place with all kind of facilities that they both desired. But there were two constraints—first of all, there was only one apartment, and second, the rent was high. So, to avail themselves of all those facilities, they decided to stay together and share the rent. This can be considered a real-life example of the flyweight pattern.

What is the Flyweight Design Patterns?

- **Example before Flyweight pattern** : Creating 20 circles of different locations



What is the Flyweight Design Patterns?

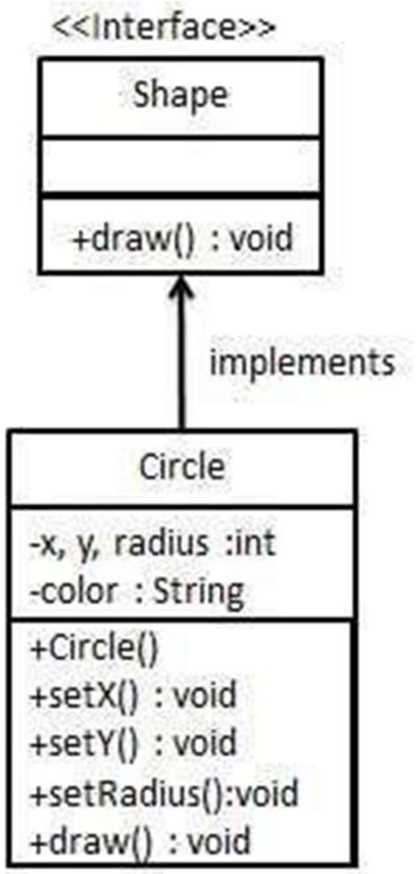
- Implementation : We are going to create a *Shape* interface and concrete class *Circle* implementing the *Shape* interface.

Shape.java

```
public interface Shape {
  void draw();
 }
```

Circle.java

```
public class Circle implements Shape {
  private String color;
  private int x;
  private int y;
  private int radius;

  public Circle(String color){
    this.color = color;
  }

  public void setX(int x) {
    this.x = x;
  }
  public void setY(int y) {
    this.y = y;
   }
  public void setRadius(int radius) {
    this.radius = radius;
   }
  @Override
  public void draw() {
    System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y :" + y + ", radius :" + radius);
   }
 }
```

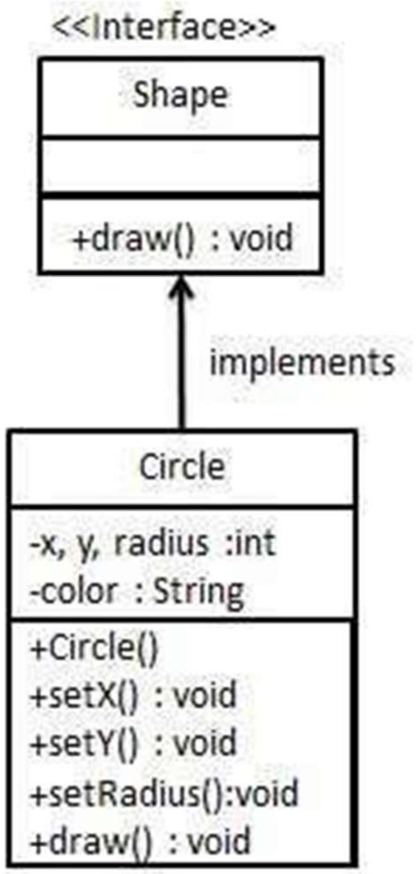


What is the Flyweight Design Patterns?

- Implementation : We are going to create a *Shape* interface and concrete class *Circle* implementing the *Shape* interface.

```
public class Demo {

   private static final String colors[] = { "Red", "Green", "Blue", “Black”,
     ”White”, };
   public static void main(String[] args) {
     for(int i=0; i < 20; ++i) {
        Circle circle = new Circle(color [(int)(Math.random()*colors.length)]);
          circle.setX(Math.random()*100);
          circle.setY(Math.random()*100);
          circle.setRadius(Math.random()*100);
          circle.draw();
      }
    }
}
```

Demo.java



Before Flyweight Design Patterns **we must create 20 objects** in order to draw 20 circles .

What is the Flyweight Design Patterns?

Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found.

A flyweight is an object through which we try to minimize memory usage by sharing data as much as possible.

Two common terms are used here:
—*intrinsic* state : color
—*extrinsic* state: size.

The first category (intrinsic) can be stored in the flyweight and is shareable. The other one depends on the flyweight’s context and is non-shareable.

What is the Flyweight Design Patterns?

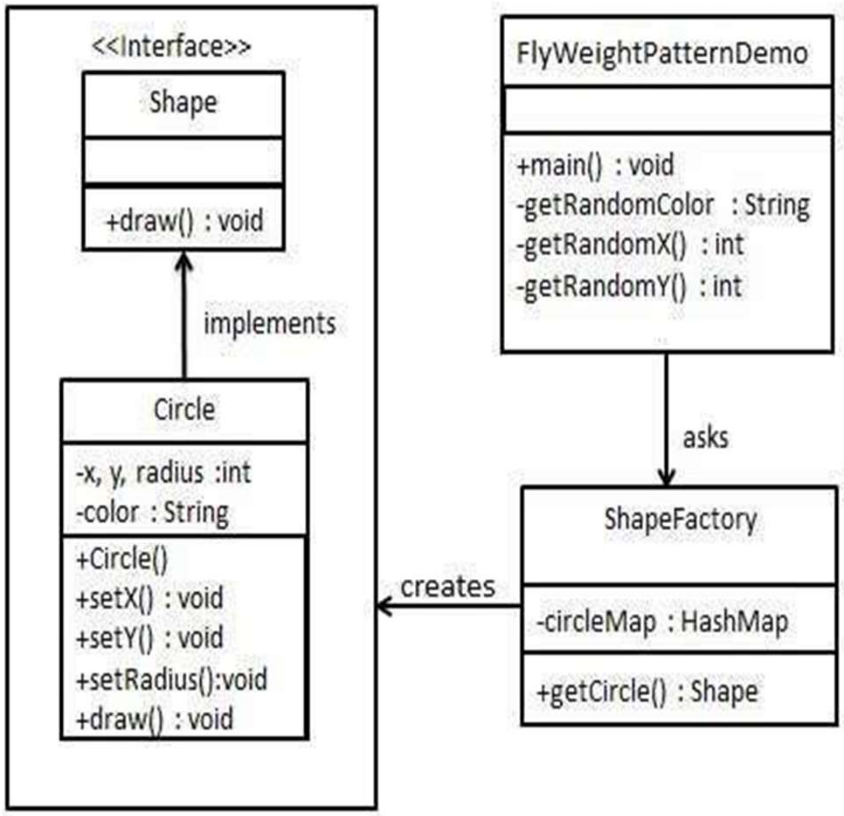
We will demonstrate this pattern by drawing 20 circles of different locations but we will **create only 5 objects**.

Only 5 colors are available so **color property (intrinsic state)** is used to **check** already existing *Circle* objects.

What is the Flyweight Design Patterns?

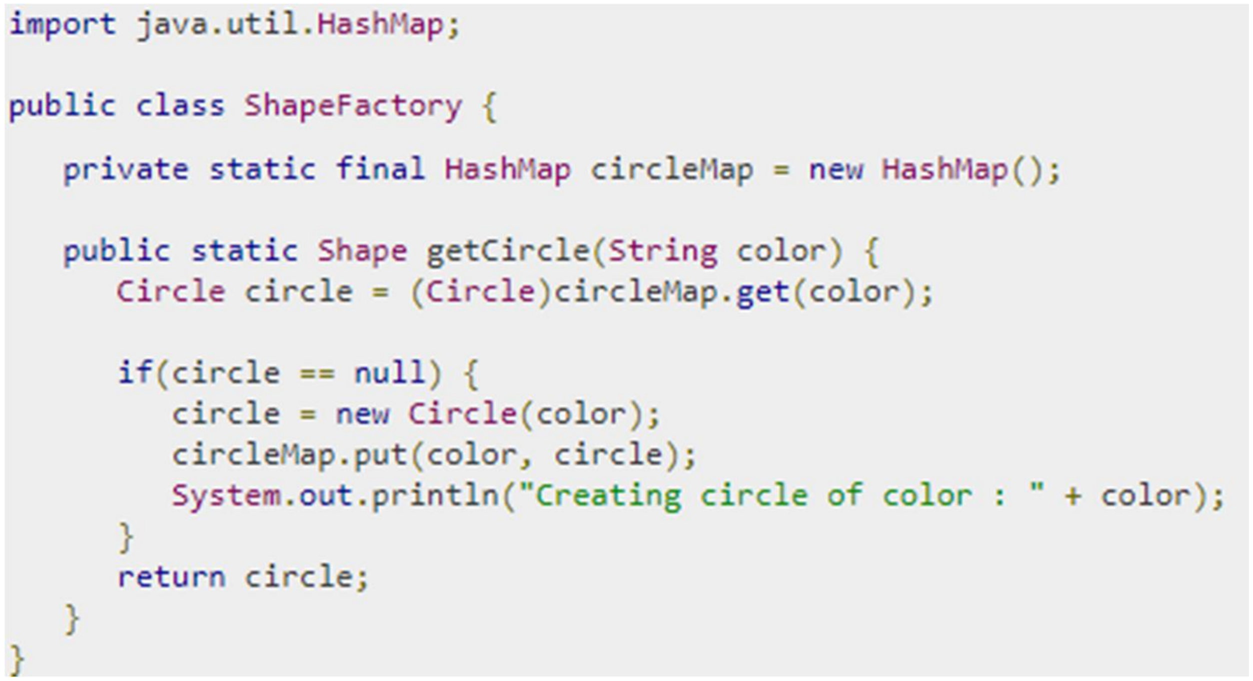
ShapeFactory has a *HashMap* of *Circle* having key as color (intrinsic state) of the *Circle* object. Whenever a request comes to create a circle of particular color to *ShapeFactory*, it checks the circle object in its *HashMap*, if object of *Circle* found, that object is returned otherwise a new object is created, stored in hashmap for future use, and returned to client.

FlyWeightPatternDemo, our demo class, will use *ShapeFactory* to get a *Shape*object. It will pass information (*red / green / blue/ white/ black/*) to *ShapeFactory* to get the circle of desired color it needs.



What is the Flyweight Design Patterns?

Create a factory to generate object of concrete class based on given information.

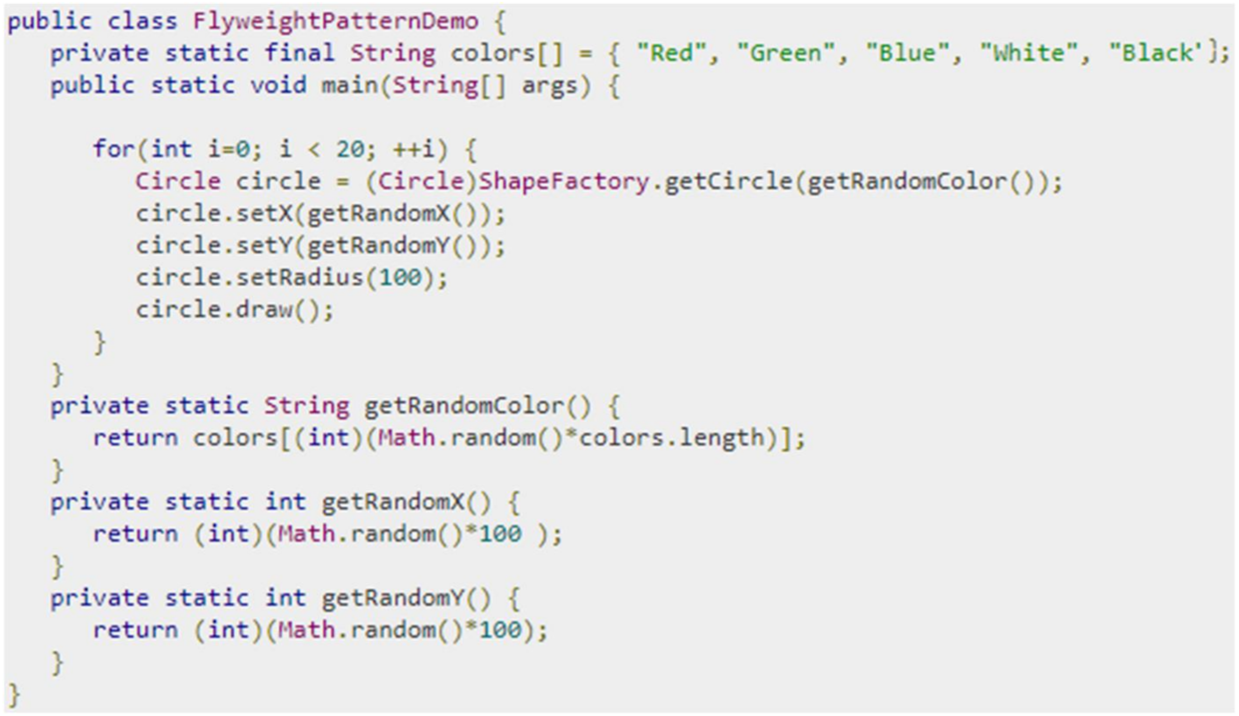


```java
import java.util.HashMap;

public class ShapeFactory {

   private static final HashMap circleMap = new HashMap();

   public static Shape getCircle(String color) {
      Circle circle = (Circle)circleMap.get(color);

      if(circle == null) {
         circle = new Circle(color);
         circleMap.put(color, circle);
         System.out.println("Creating circle of color : " + color);
      }
      return circle;
   }
}
```

ShapeFactory.java

What is the Flyweight Design Patterns?

Use the factory to get object of concrete class by passing an information such as color.

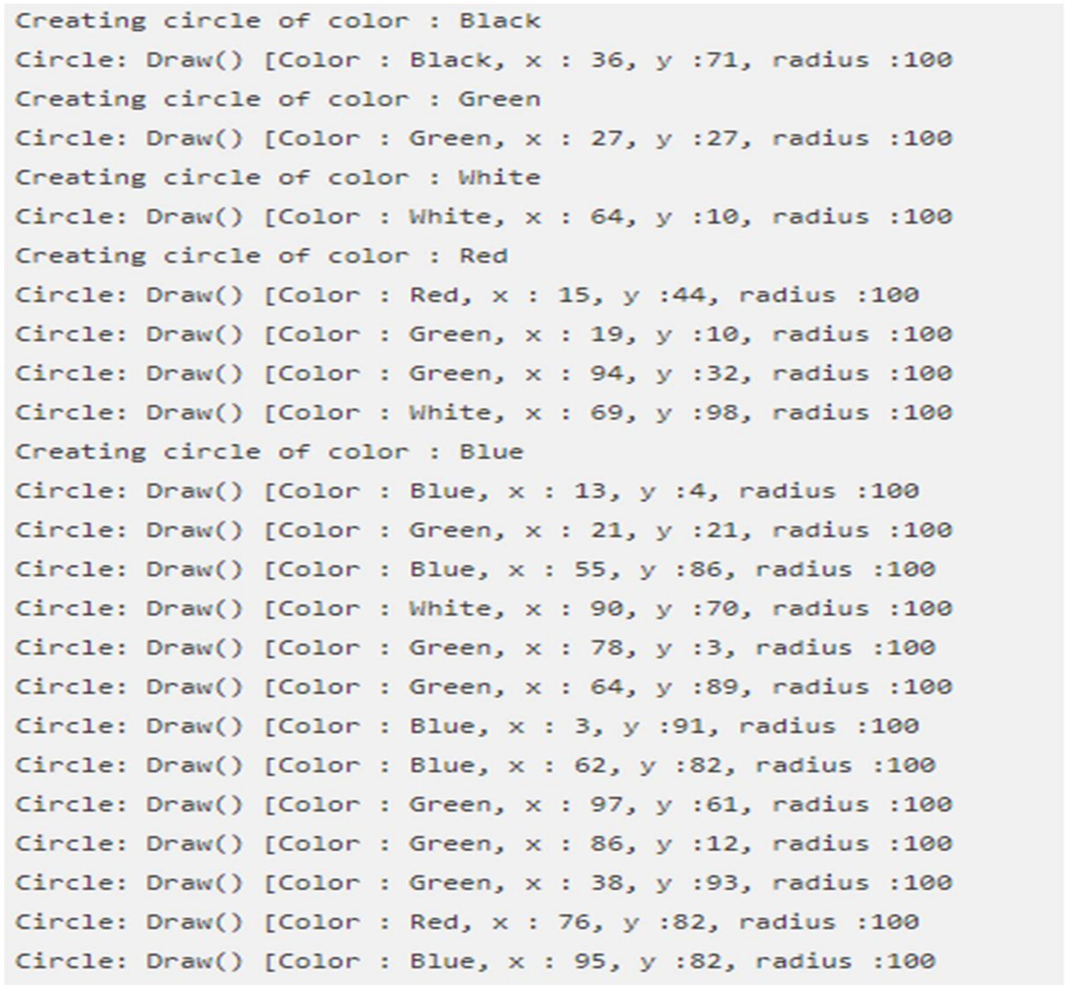


```
public class FlyweightPatternDemo {
   private static final String colors[] = { "Red", "Green", "Blue", "White", "Black'};
   public static void main(String[] args) {

      for(int i=0; i < 20; ++i) {
         Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());
         circle.setX(getRandomX());
         circle.setY(getRandomY());
         circle.setRadius(100);
         circle.draw();
      }
   }
   private static String getRandomColor() {
      return colors[(int)(Math.random()*colors.length)];
   }
   private static int getRandomX() {
      return (int)(Math.random()*100 );
   }
   private static int getRandomY() {
      return (int)(Math.random()*100);
   }
}
```

FlyweightPatternDemo.java

What is the Flyweight Design Patterns?

Verify the output



```
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Circle: Draw() [Color : Blue, x : 95, y :82, radius :100
```

Outline

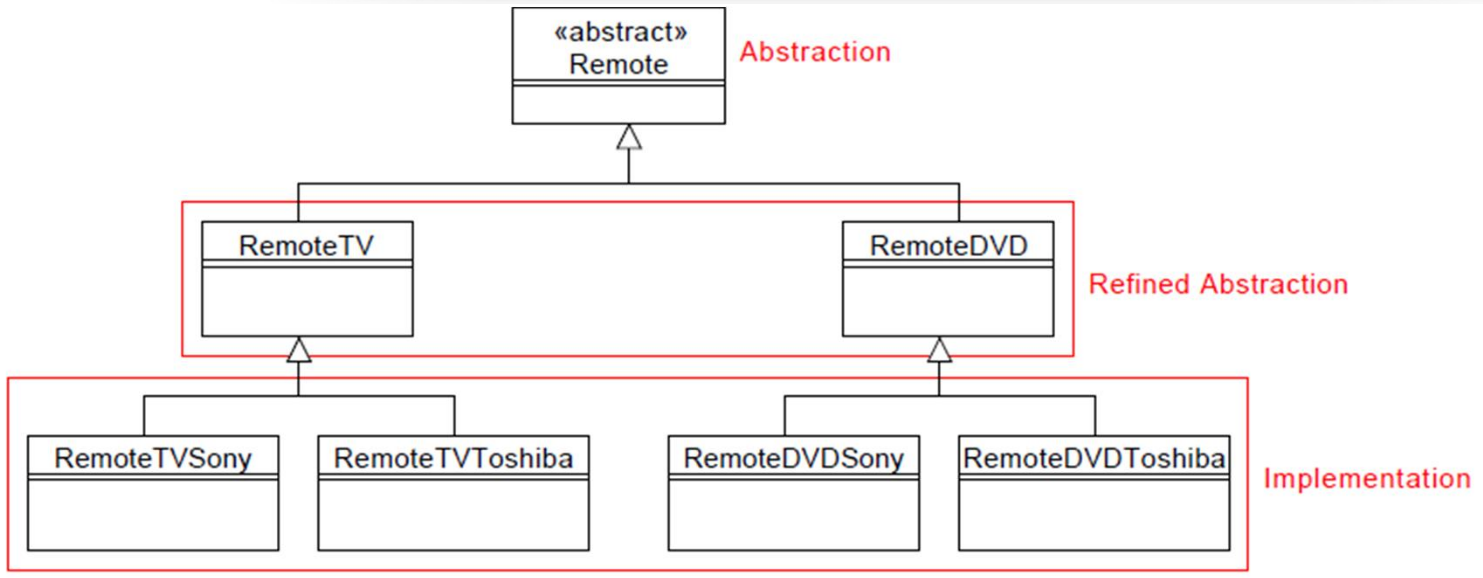
- Introduction
- Creational patterns
- Structural patterns
  - Flyweight
  - Bridge
  - Composite
- Behavioral patterns

Bridge Pattern

Decouple an abstraction from its implementation so that the two can vary independently

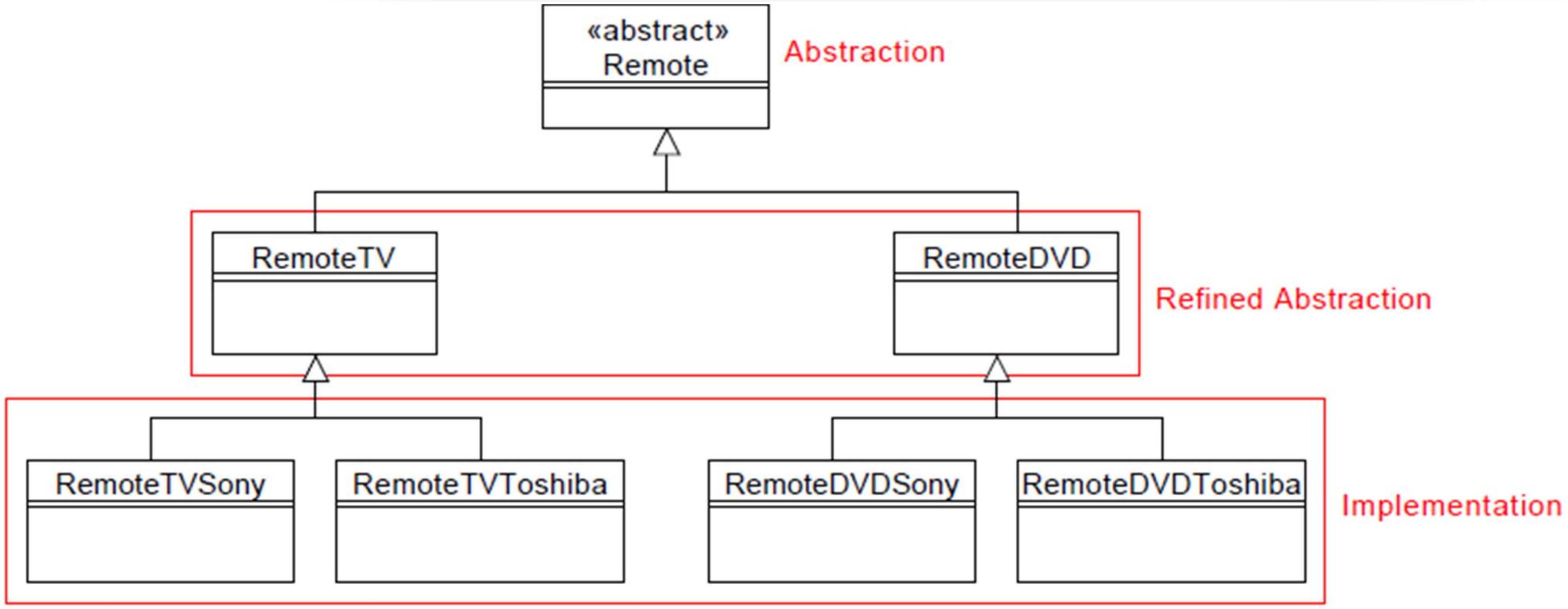
In this pattern, the abstract class is separated from the implementation class and we provide a bridge interface between them. This interface helps us to make concrete class functionalities independent from the interface implementer class. We can modify these different kind of classes structurally without affecting each other.

Example : Before Bridge Design Pattern



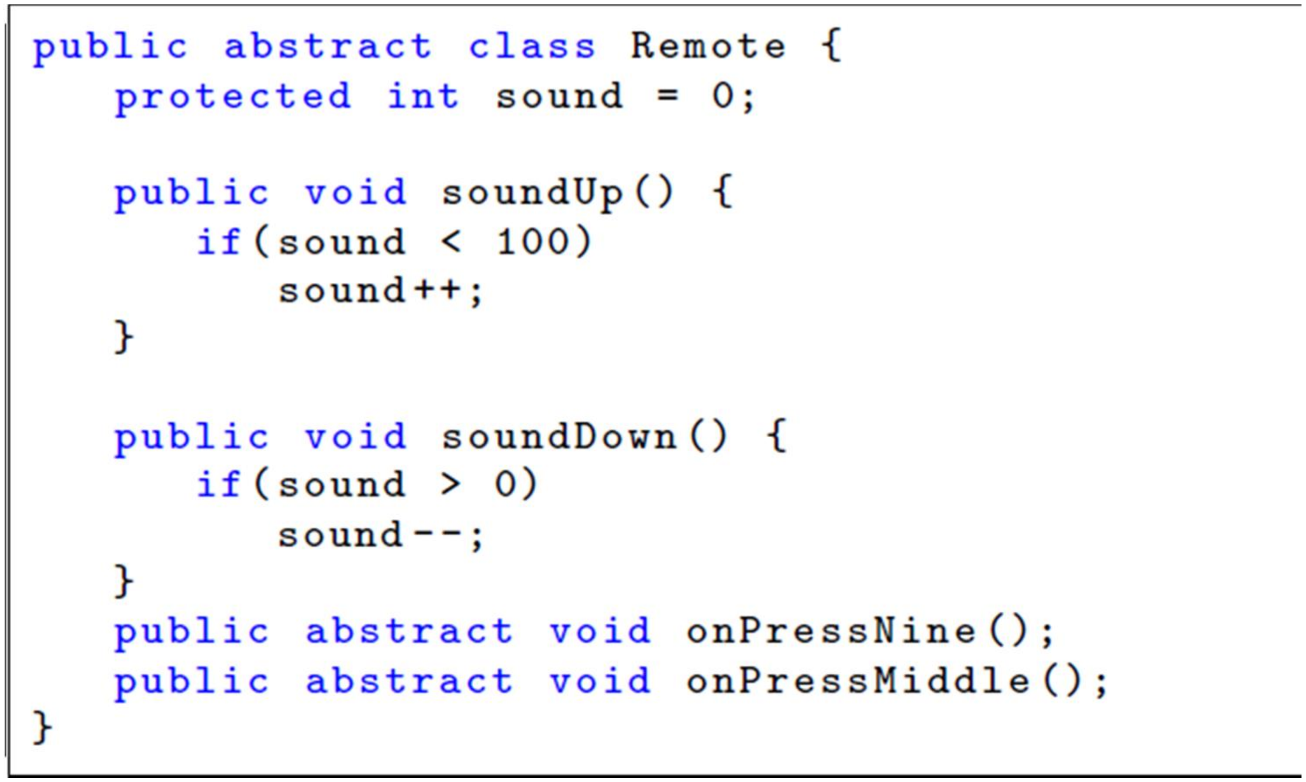
- Remote is an abstract class. It contains:
  - an instance variable sound
  - two methods soundUp and soundDown
  - two abstract methods onPressMiddle and onPressNine
- If it is remote for TV: onPressNine ➔ go to the channel nine.
- If it is remote for DVD: onPressNine ➔pause.
- If it is Sony remote: onPressMiddle ➔open menu.
- If it is Toshiba remote: onPressMiddle ➔turn off.

Example : Before Bridge Design Pattern



- Problems
  - Create a remote for LCD creation of three new data types (RemoteLCD, RemoteLCDSony, RemoteLCDToshiba).
  - Code redundancy (duplication of methods)!

Example : Before Bridge Design Pattern



```java
public abstract class Remote {
    protected int sound = 0;

    public void soundUp() {
        if(sound < 100)
            sound++;
    }

    public void soundDown() {
        if(sound > 0)
            sound--;
    }
    public abstract void onPressNine();
    public abstract void onPressMiddle();
}
```

Example : Before Bridge Design Pattern

```java
public abstract class RemoteDVD extends Remote {
    public void onPressNine() {
        System.out.println("Pause");
    }
}
```

```java
public class RemoteDVDSony extends RemoteDVD {
    public void onPressMiddle() {
        System.out.println("Open Menu");
    }
}
```

```java
public class RemoteDVDToshiba extends RemoteDVD {
    public void onPressMiddle() {
        System.out.println("Turn Off");
    }
}
```

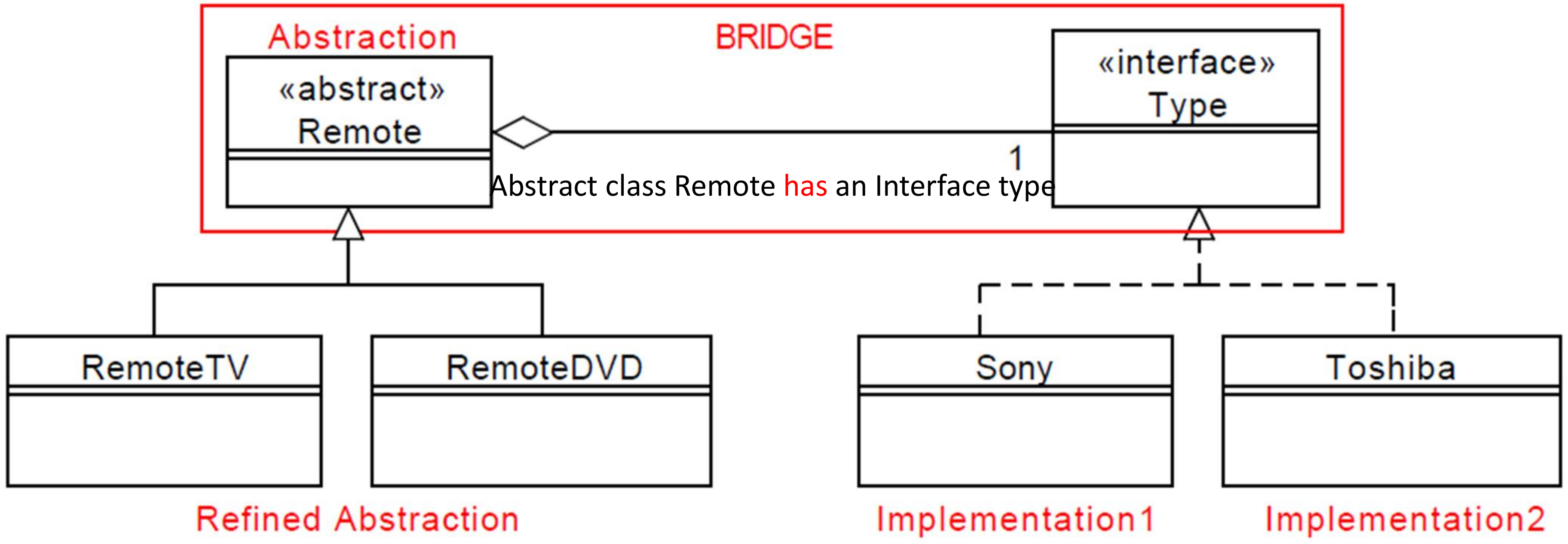
Example : Before Bridge Design Pattern

```
public abstract class RemoteTV extends Remote {
   public void onPressNine() {
      System.out.println("Go To Channel Nine");
   }
}
```

```
public class RemoteTVSony extends RemoteTV {
   public void onPressMiddle() {
      System.out.println("Open Menu");
   }
}
```

```
public class RemoteTVToshiba extends RemoteTV {
   public void onPressMiddle() {
      System.out.println("Turn Off");
   }
}
```

Example : After Bridge Design Pattern



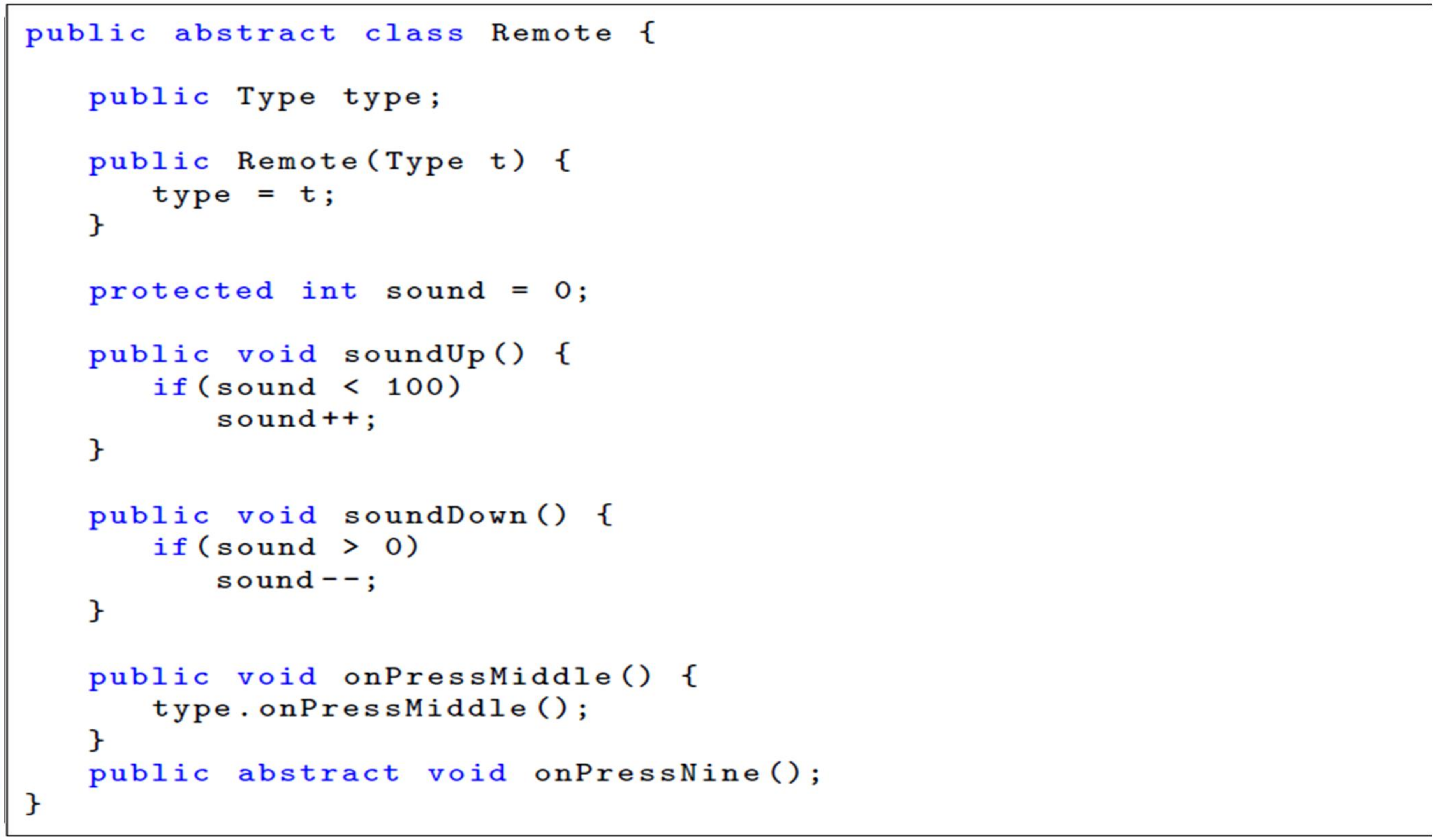
Example : After Bridge Design Pattern

```java
public interface Type {
    public void onPressMiddle();
}
```

```java
public class Toshiba implements Type {
    @Override
    public void onPressMiddle() {
        System.out.println("Turn Off");
    }
}
```

```java
public class Sony implements Type {
    @Override
    public void onPressMiddle() {
        System.out.println("Open Menu");
    }
}
```

Example : After Bridge Design Pattern



```
public abstract class Remote {

   public Type type;

   public Remote(Type t) {
      type = t;
   }

   protected int sound = 0;

   public void soundUp() {
      if(sound < 100)
         sound++;
   }

   public void soundDown() {
      if(sound > 0)
         sound--;
   }

   public void onPressMiddle() {
      type.onPressMiddle();
   }
   public abstract void onPressNine();
}
```

Example : After Bridge Design Pattern

```java
public class RemoteDVD extends Remote {
    public RemoteDVD(Type t) {
        super(t);
    }
    public void onPressNine() {
        System.out.println("Pause");
    }
}
```

```java
public class RemoteTV extends Remote {
    public RemoteTV(Type t) {
        super(t);
    }
    public void onPressNine() {
        System.out.println("Go To Channel Nine");
    }
}
```

Example : After Bridge Design Pattern

```java
public class Test {
   public static void main(String[] args) {
      Remote r1 = new RemoteDVD(new Toshiba());
      Remote r2 = new RemoteTV(new Sony());

      r1.onPressMiddle();
      r2.onPressNine();
   }
}
```

Outline

- Introduction
- Creational patterns
- Structural patterns
  - Flyweight
  - Bridge
  - Composite
- Behavioral patterns

Composite Pattern

- Compose objects into tree structures to represent part-whole hierarchies.
- In other words, when we need to create a structure in a way that the objects in the structure has to be treated the same way, we can apply composite design pattern
- In object-oriented programming, we make a composite object when we have many objects with common functionalities. This relationship is also termed a “has-a” relationship among objects.

Real-Life Example-Composite Pattern

We can think of any organization that has many departments, and in turn each department has many employees to serve.

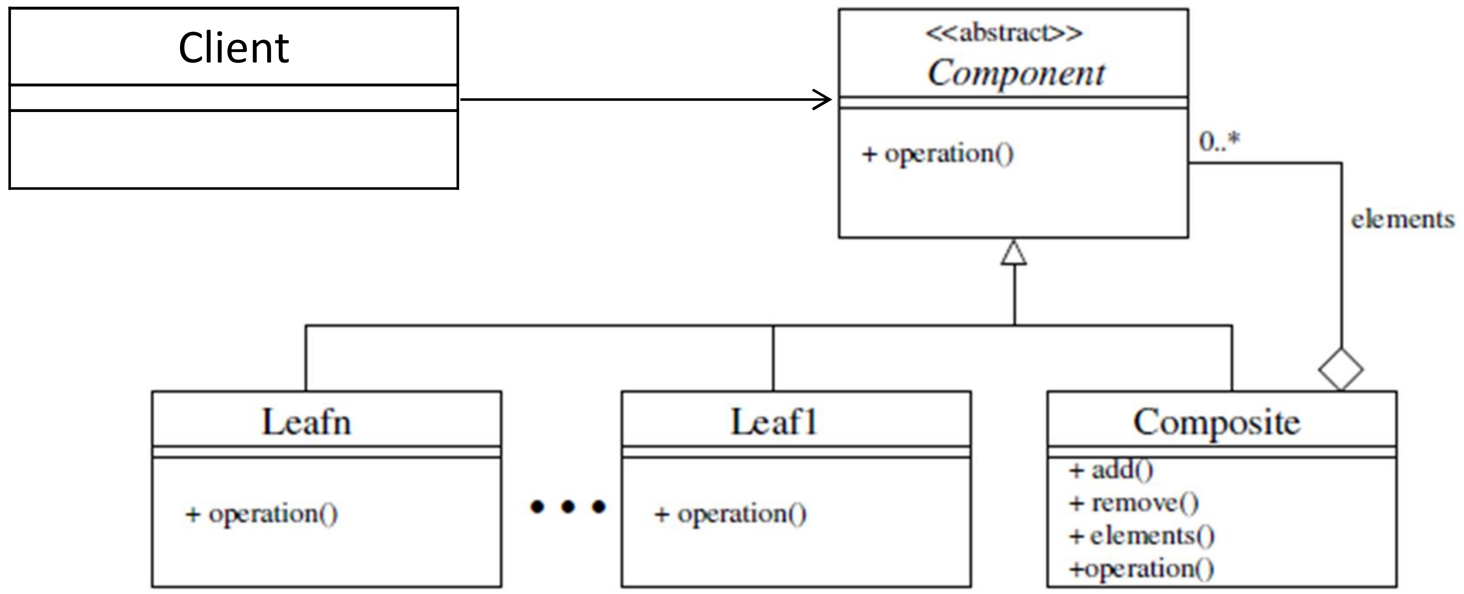
Please note that actually all are employees of the organization.

Groupings of employees create a department, and those departments ultimately can be grouped together to build the whole organization.

Example (Employees)

- You have a set of employees
- An employee could be a manager (of a set of employees), a developer, or a designer

Composite Pattern

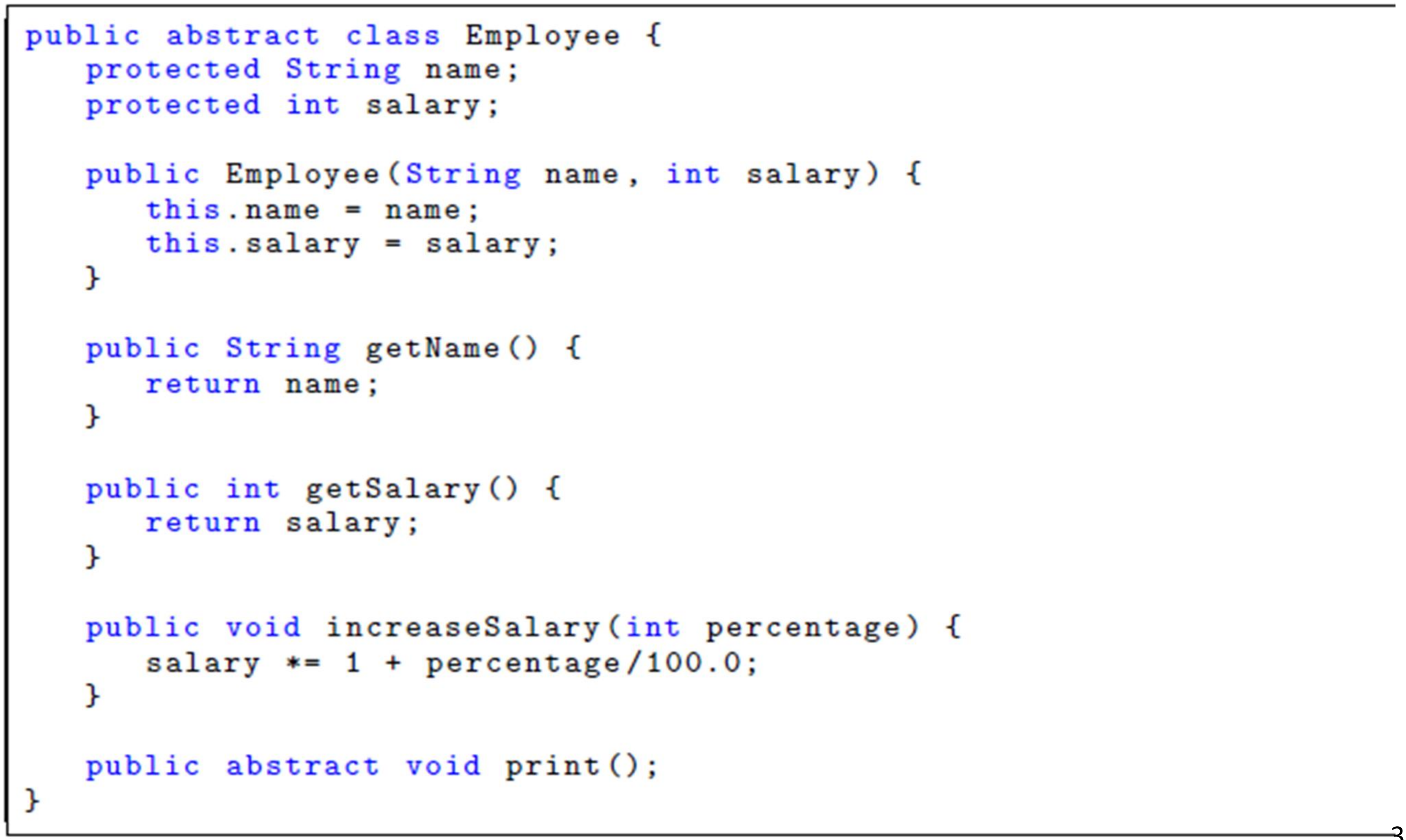


1-Base Component – Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition. It can be an interface or an **abstract class** with some methods common to all the objects.

2-Leaf – Defines the behavior for the elements in the composition. It is the building block for the composition and implements base component. It doesn't have references to other Components.

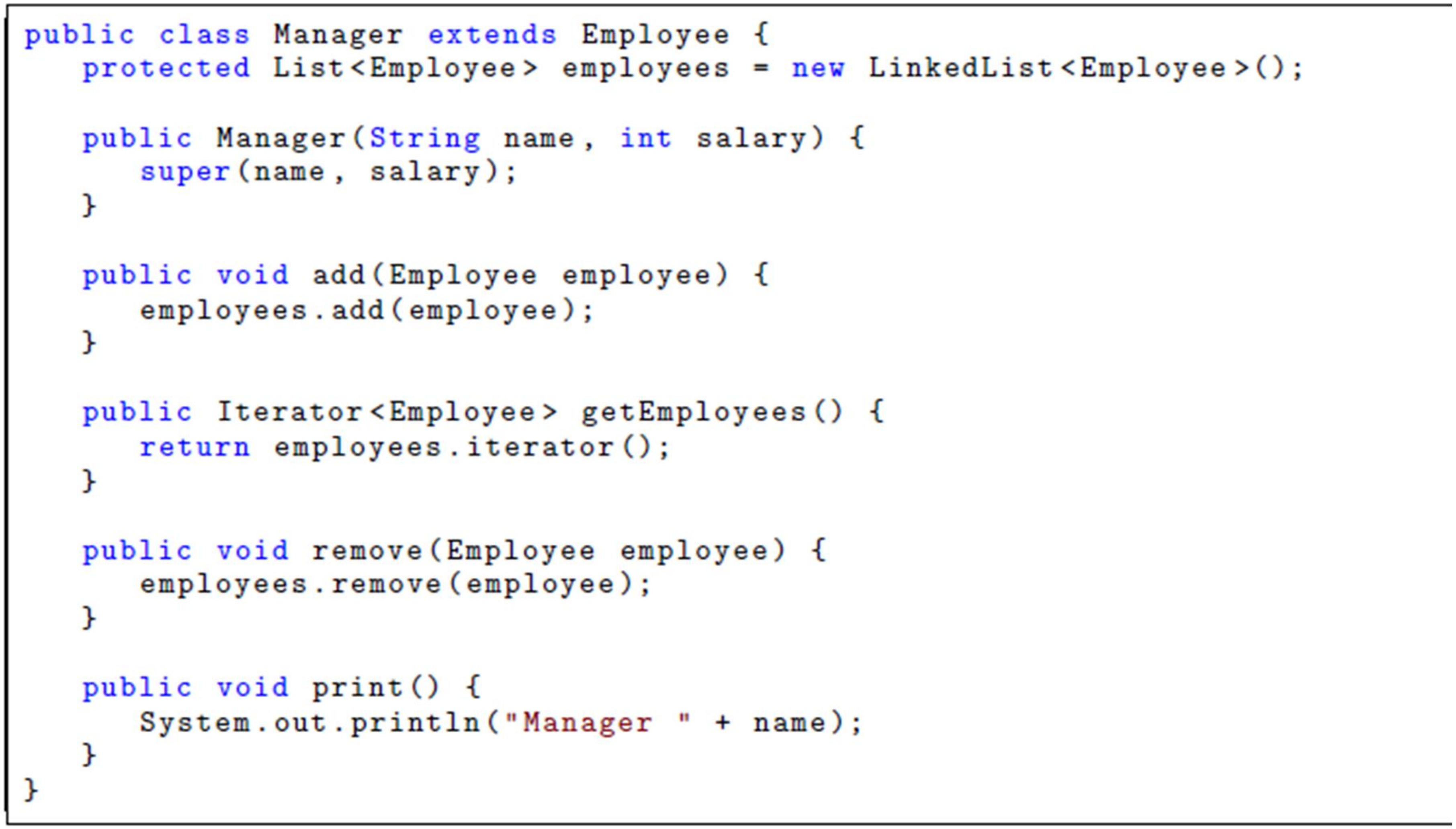
3-Composite – It consists of leaf elements and implements the operations in base component.

Composite Pattern - Employee Example



```
public abstract class Employee {
   protected String name;
   protected int salary;

   public Employee(String name, int salary) {
      this.name = name;
      this.salary = salary;
   }

   public String getName() {
      return name;
   }

   public int getSalary() {
      return salary;
   }

   public void increaseSalary(int percentage) {
      salary *= 1 + percentage/100.0;
   }

   public abstract void print();
}
```

Composite Pattern - Employee Example



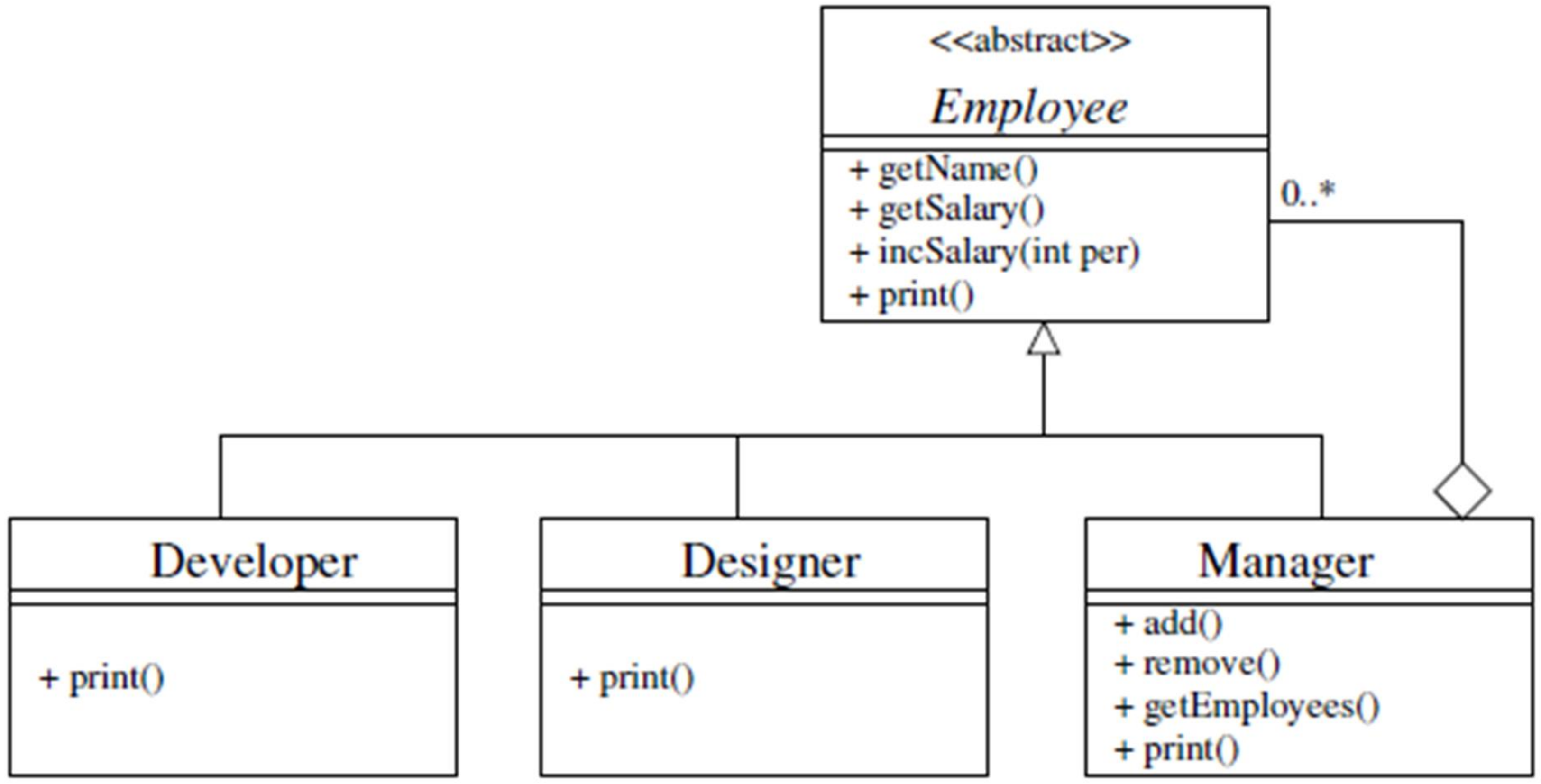
```java
public class Manager extends Employee {
   protected List<Employee> employees = new LinkedList<Employee>();

   public Manager(String name, int salary) {
      super(name, salary);
   }

   public void add(Employee employee) {
      employees.add(employee);
   }

   public Iterator<Employee> getEmployees() {
      return employees.iterator();
   }

   public void remove(Employee employee) {
      employees.remove(employee);
   }

   public void print() {
      System.out.println("Manager " + name);
   }
}
```

Composite Pattern - Employee Example

```java
public class Developer extends Employee {
   public Developer(String name, int salary) {
      super(name, salary);
   }

   public void print() {
      System.out.println("Developer " + name);
   }
}
```

```java
public class Designer extends Employee {
   public Designer(String name, int salary) {
      super(name, salary);
   }

   public void print() {
      System.out.println("Designer " + name);
   }
}
```

Composite Pattern – Employee Example



End of the chapter…