Lecture - 6

1- Composite
2- Proxy
3- Fly weight

Composite Pattern

Definition:

"Compose objects into tree structures to represent part-whole hierarchies. The composite pattern lets clients treat individual objects and compositions of objects uniformly."

Concept

- This pattern can show part-whole hierarchy among objects.
- A client can treat a composite object just like a single object.

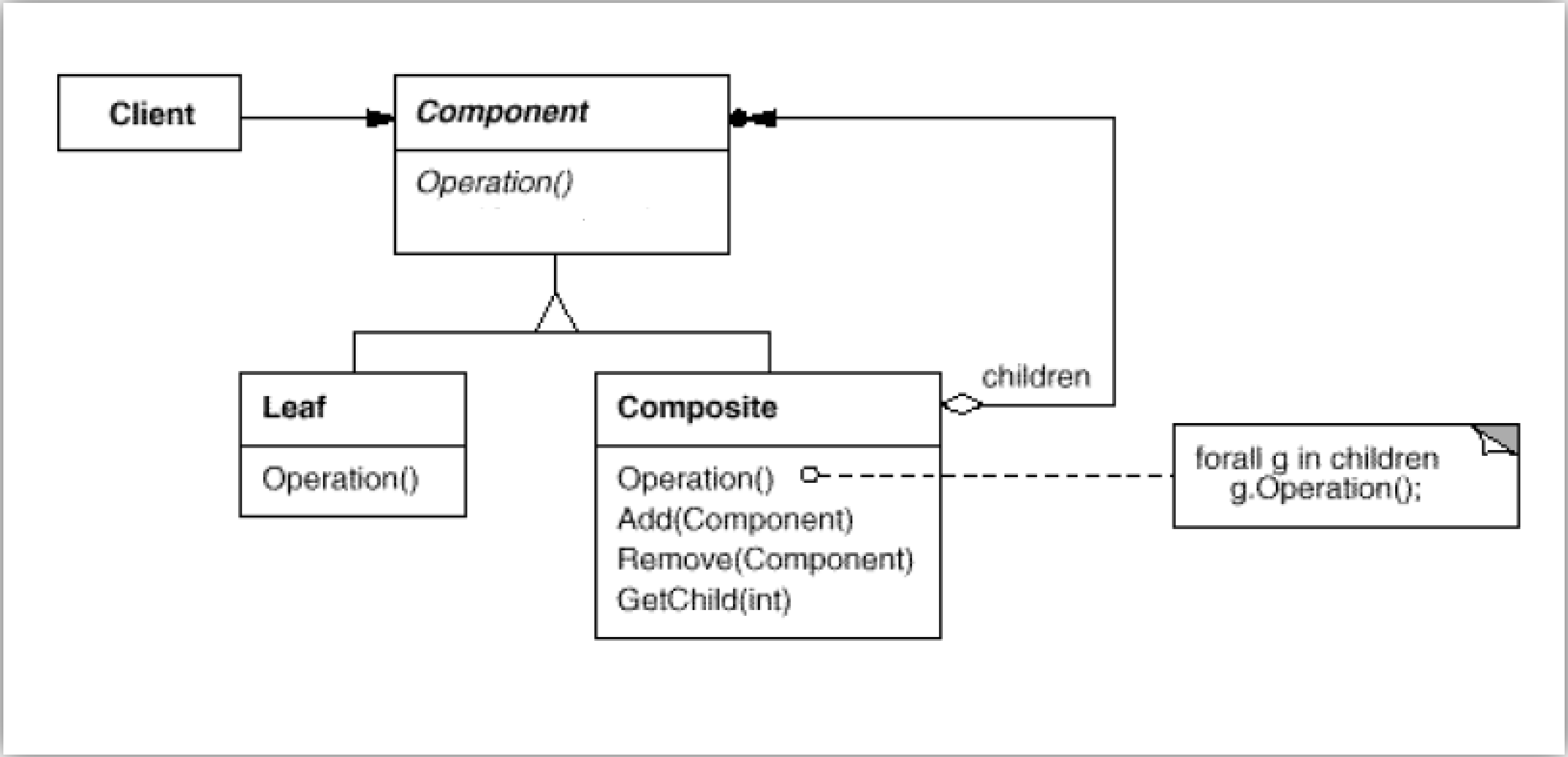
Real-Life Examples

- We can think of any organization that has many departments, and in turn each department has many employees to serve.
- Groupings of employees create a department, and those departments ultimately can be grouped together to build the whole organization.

Where to use

- When you want to represent a part-whole relationship in a tree structure.
- When you want clients to be able to ignore the differences between compositions of objects and individual objects.

Structure


Client
Component
Operation()
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

Participants

- **Component**
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all.
- **Leaf**
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.

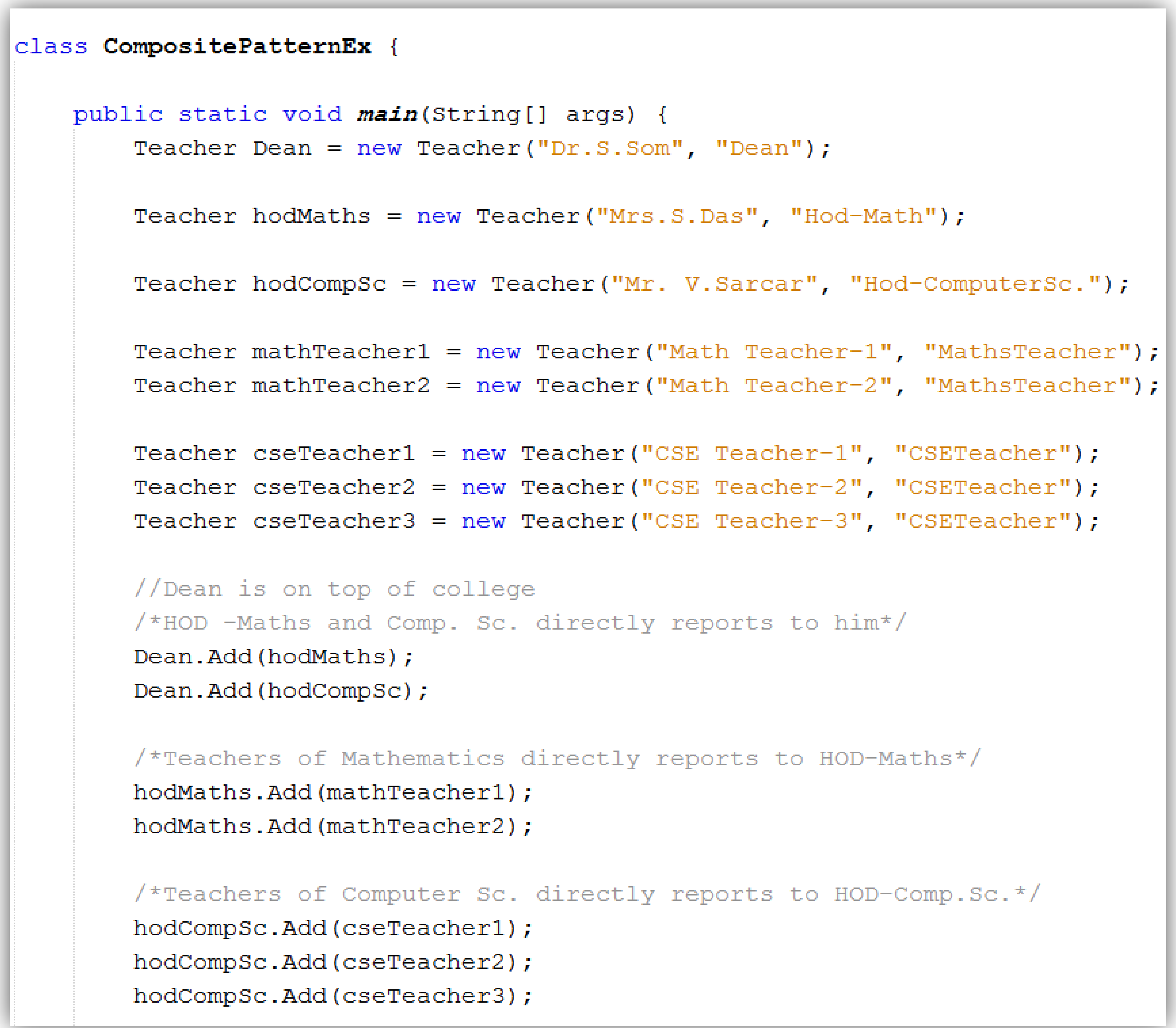
Participants

- **Composite**
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- **Client**
  - manipulates objects in the composition through the Component interface.

Example:

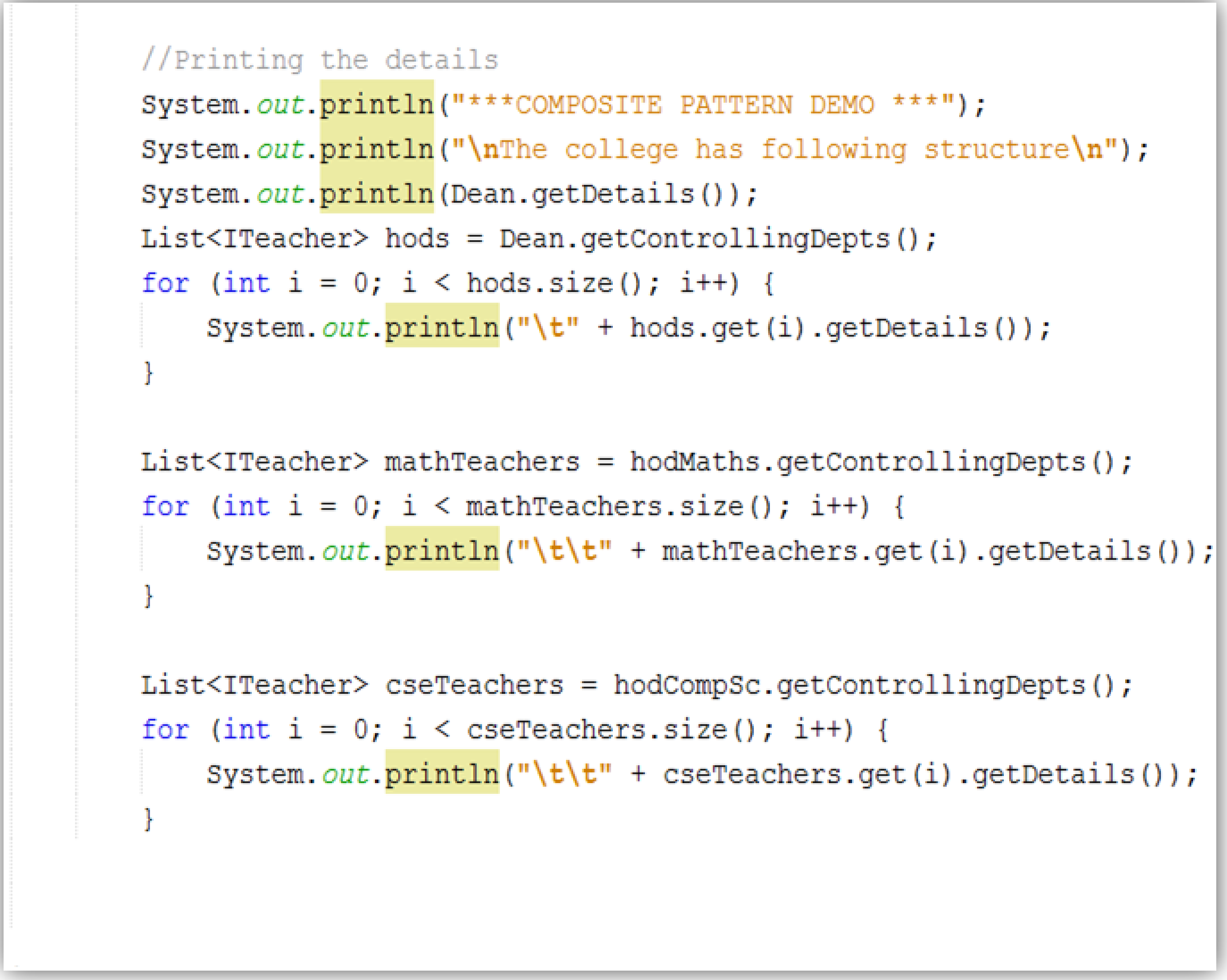
- In this example we are showing a college organization. We have a Dean and two Heads of Departments:
- one for computer science and one for mathematics.
- At present, in the mathematics department, we have two lecturers; in the computer science department we have three lecturers.

Example:

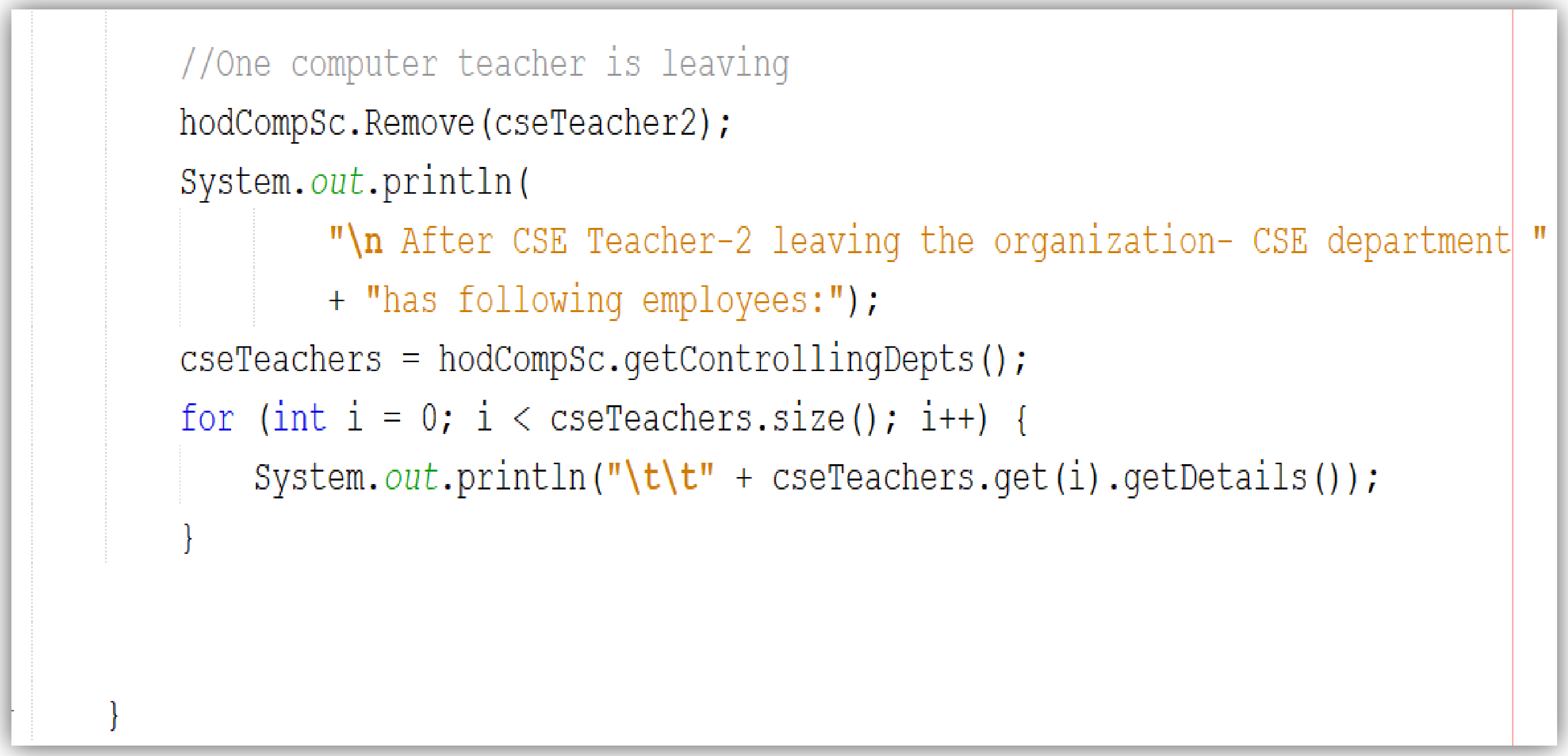


```
class CompositePatternEx {

    public static void main(String[] args) {
        Teacher Dean = new Teacher("Dr.S.Som", "Dean");

        Teacher hodMaths = new Teacher("Mrs.S.Das", "Hod-Math");

        Teacher hodCompSc = new Teacher("Mr. V.Sarcar", "Hod-ComputerSc.");

        Teacher mathTeacher1 = new Teacher("Math Teacher-1", "MathsTeacher");
        Teacher mathTeacher2 = new Teacher("Math Teacher-2", "MathsTeacher");

        Teacher cseTeacher1 = new Teacher("CSE Teacher-1", "CSETeacher");
        Teacher cseTeacher2 = new Teacher("CSE Teacher-2", "CSETeacher");
        Teacher cseTeacher3 = new Teacher("CSE Teacher-3", "CSETeacher");

        //Dean is on top of college
        /*HOD -Maths and Comp. Sc. directly reports to him*/
        Dean.Add(hodMaths);
        Dean.Add(hodCompSc);

        /*Teachers of Mathematics directly reports to HOD-Maths*/
        hodMaths.Add(mathTeacher1);
        hodMaths.Add(mathTeacher2);

        /*Teachers of Computer Sc. directly reports to HOD-Comp.Sc.*/
        hodCompSc.Add(cseTeacher1);
        hodCompSc.Add(cseTeacher2);
        hodCompSc.Add(cseTeacher3);
```

Cont…

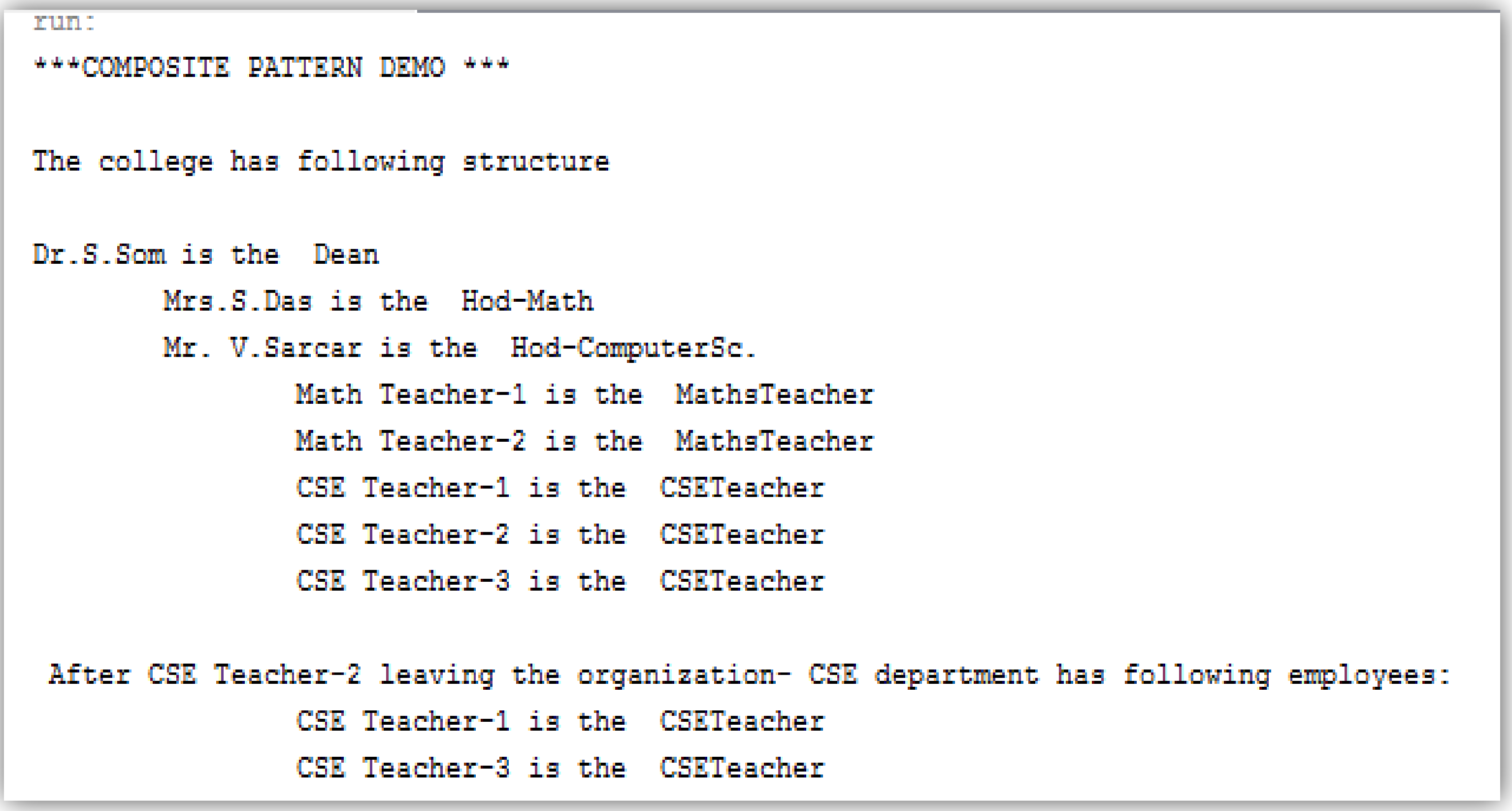


```java
//Printing the details
System.out.println("***COMPOSITE PATTERN DEMO ***");
System.out.println("\nThe college has following structure\n");
System.out.println(Dean.getDetails());
List<ITeacher> hods = Dean.getControllingDepts();
for (int i = 0; i < hods.size(); i++) {
    System.out.println("\t" + hods.get(i).getDetails());
}

List<ITeacher> mathTeachers = hodMaths.getControllingDepts();
for (int i = 0; i < mathTeachers.size(); i++) {
    System.out.println("\t\t" + mathTeachers.get(i).getDetails());
}

List<ITeacher> cseTeachers = hodCompSc.getControllingDepts();
for (int i = 0; i < cseTeachers.size(); i++) {
    System.out.println("\t\t" + cseTeachers.get(i).getDetails());
}
```



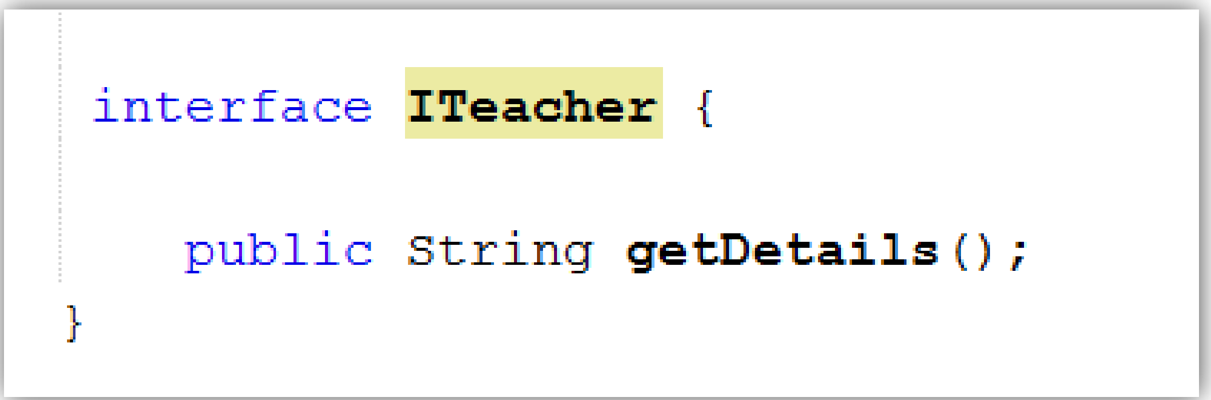
```
        //One computer teacher is leaving
        hodCompSc.Remove(cseTeacher2);
        System.out.println(
                "\n After CSE Teacher-2 leaving the organization- CSE department "
                + "has following employees:");
        cseTeachers = hodCompSc.getControllingDepts();
        for (int i = 0; i < cseTeachers.size(); i++) {
            System.out.println("\t\t" + cseTeachers.get(i).getDetails());
        }


    }
```

Output:



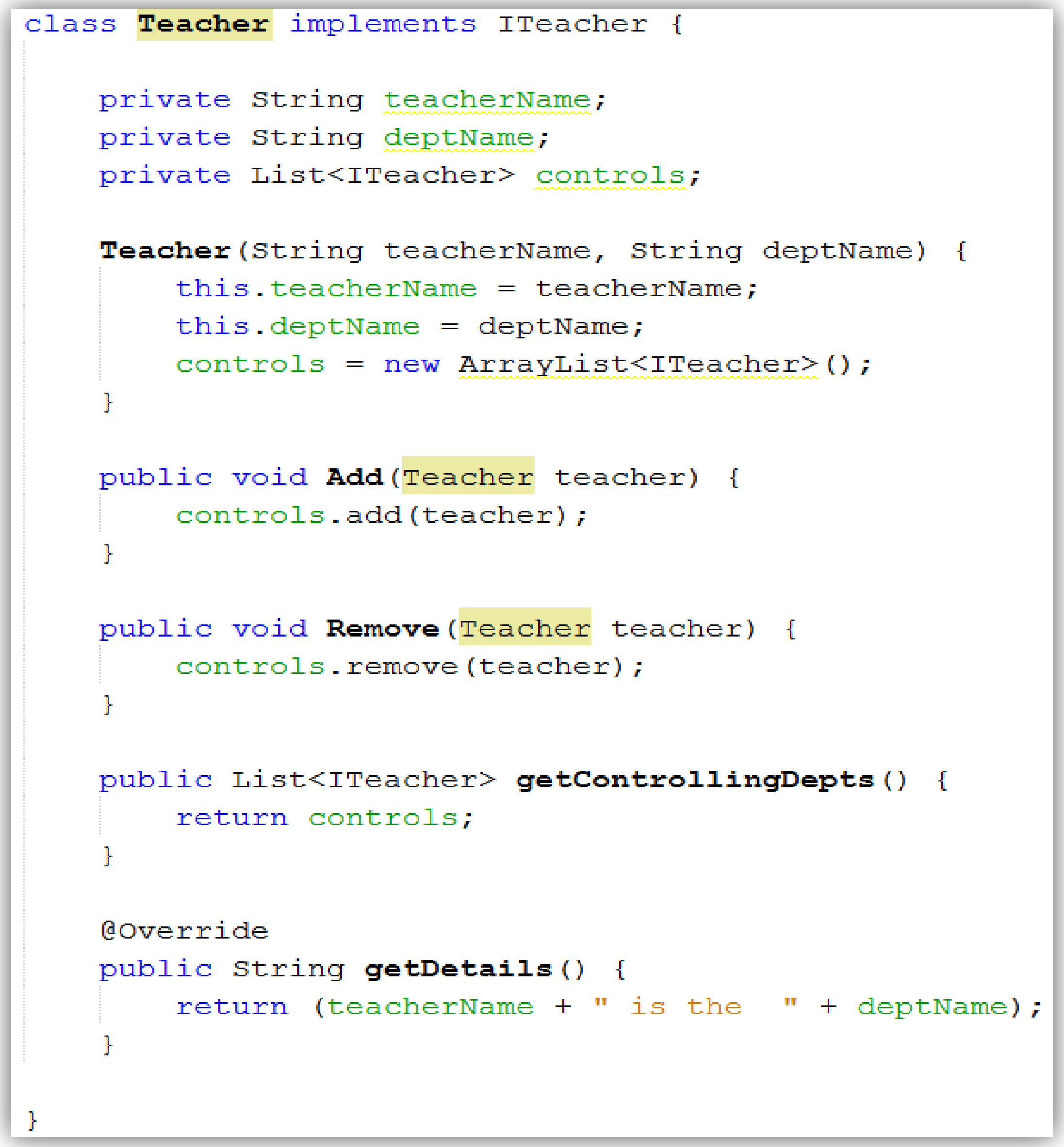
```
run:
***COMPOSITE PATTERN DEMO ***

The college has following structure

Dr.S.Som is the  Dean
        Mrs.S.Das is the  Hod-Math
        Mr. V.Sarcar is the  Hod-ComputerSc.
                Math Teacher-1 is the  MathsTeacher
                Math Teacher-2 is the  MathsTeacher
                CSE Teacher-1 is the  CSETeacher
                CSE Teacher-2 is the  CSETeacher
                CSE Teacher-3 is the  CSETeacher

 After CSE Teacher-2 leaving the organization- CSE department has following employees:
                CSE Teacher-1 is the  CSETeacher
                CSE Teacher-3 is the  CSETeacher
```

Component



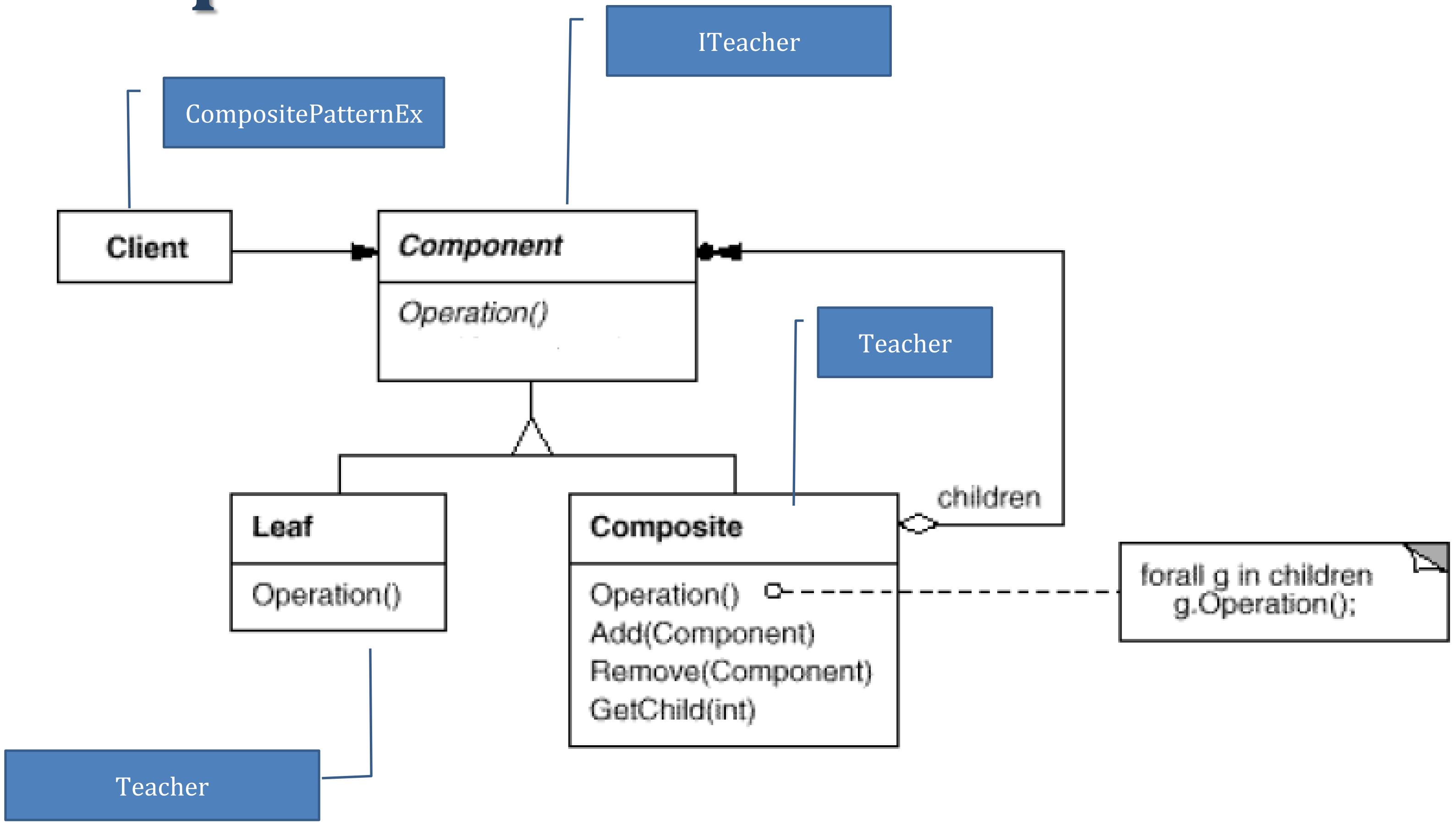
```
interface ITeacher {

    public String getDetails();
}
```

Composite or Leaf



```
class Teacher implements ITeacher {

    private String teacherName;
    private String deptName;
    private List<ITeacher> controls;

    Teacher(String teacherName, String deptName) {
        this.teacherName = teacherName;
        this.deptName = deptName;
        controls = new ArrayList<ITeacher>();
    }

    public void Add(Teacher teacher) {
        controls.add(teacher);
    }

    public void Remove(Teacher teacher) {
        controls.remove(teacher);
    }

    public List<ITeacher> getControllingDepts() {
        return controls;
    }

    @Override
    public String getDetails() {
        return (teacherName + " is the  " + deptName);
    }

}
```

Participants:



Participants

Participant	Class Or Interface
Component	ITeacher interface
Composite	Teacher class
Leaf	Teacher class
Client	CompositePatternEx class

Proxy Pattern

Definition:

“Provide a surrogate or placeholder for another object to control access to it.”

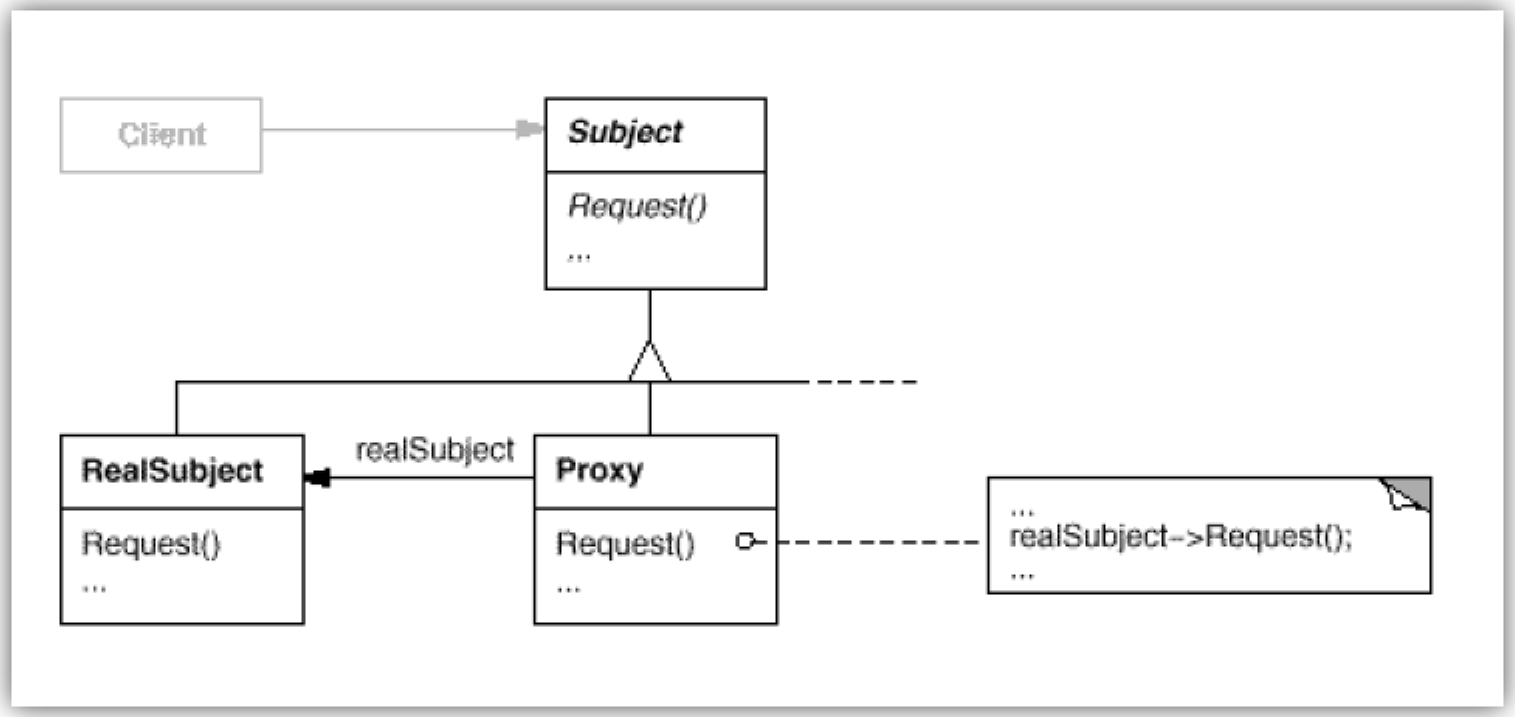
Concept

- We want to use a class which can perform as an interface to something else.

Where to use

- When the creation of one object is relatively expensive it can be a good idea to replace it with a proxy that can make sure that instantiation of the expensive object is kept to a minimum.
- Proxy pattern implementation allows for login and authority checking before one reaches the actual object that's requested.
- Can provide a local representation for an object in a remote location.

Structure


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...

Participants

- **Proxy**
  - maintains a reference that lets the proxy access the real subject
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilities depend on the kind of proxy:
    - **remote proxies** are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    - **protection proxies** check that the caller has the access permissions required to perform a request.

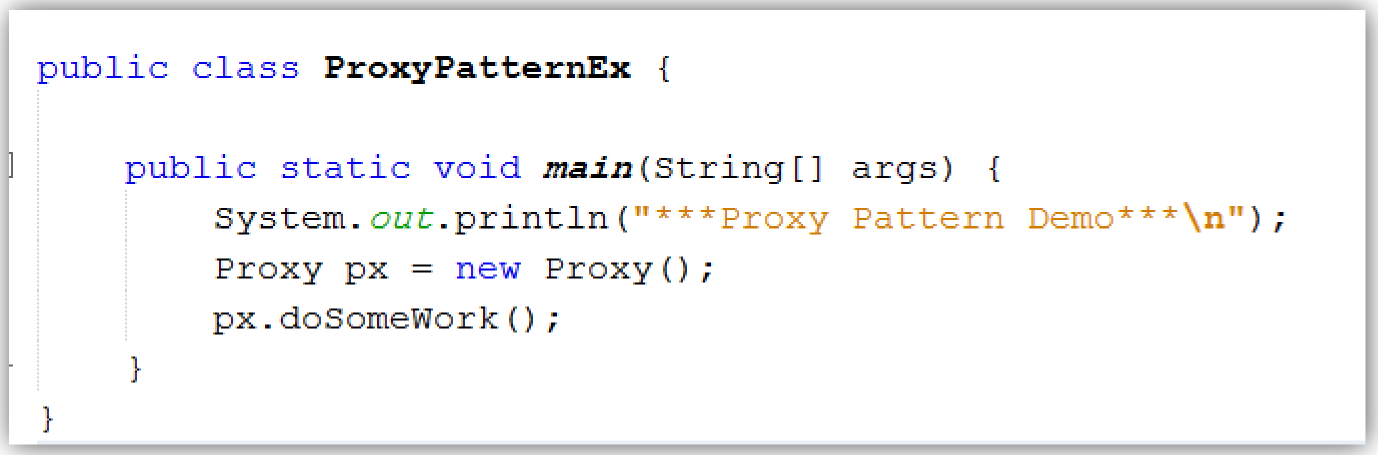
Participants

- **Subject**
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject**
  - defines the real object that the proxy represents.

Example:

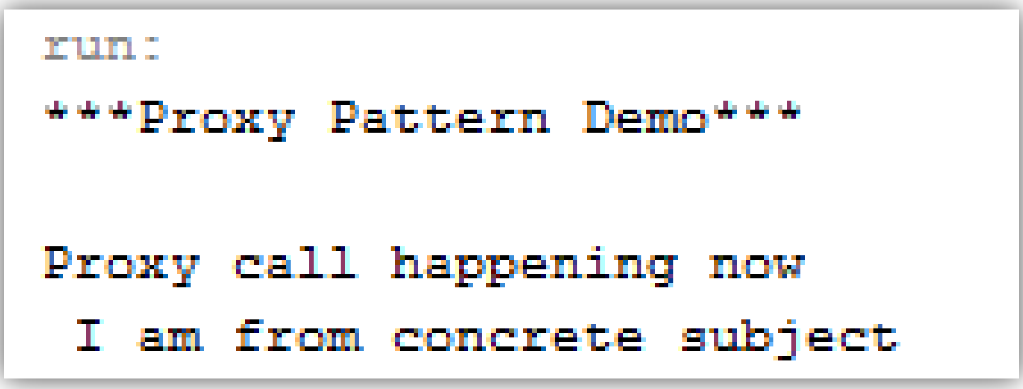
- In the following program, we are calling the doSomework() function of the proxy object, which in turn calls the doSomework() of the concrete object. With the output, we are getting the result directly through the concrete object.

Example:



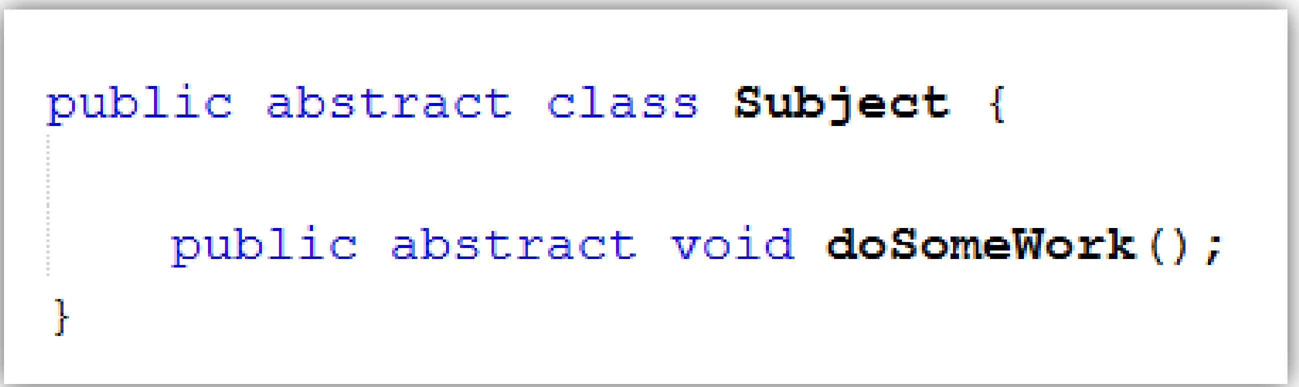
```java
public class ProxyPatternEx {

    public static void main(String[] args) {
        System.out.println("***Proxy Pattern Demo***\n");
        Proxy px = new Proxy();
        px.doSomeWork();
    }
}
```

Output:



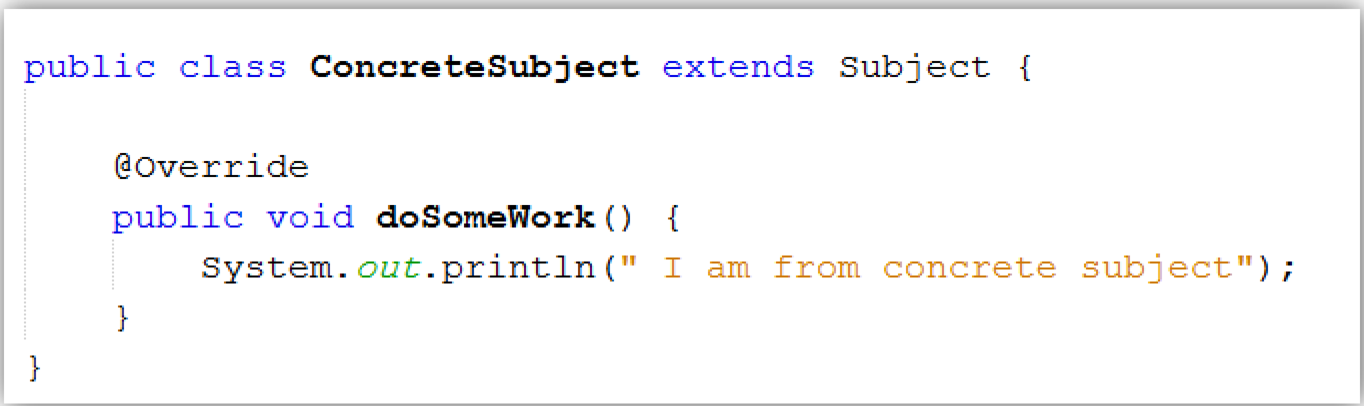
```
run:
***Proxy Pattern Demo***

Proxy call happening now
 I am from concrete subject
```

Subject



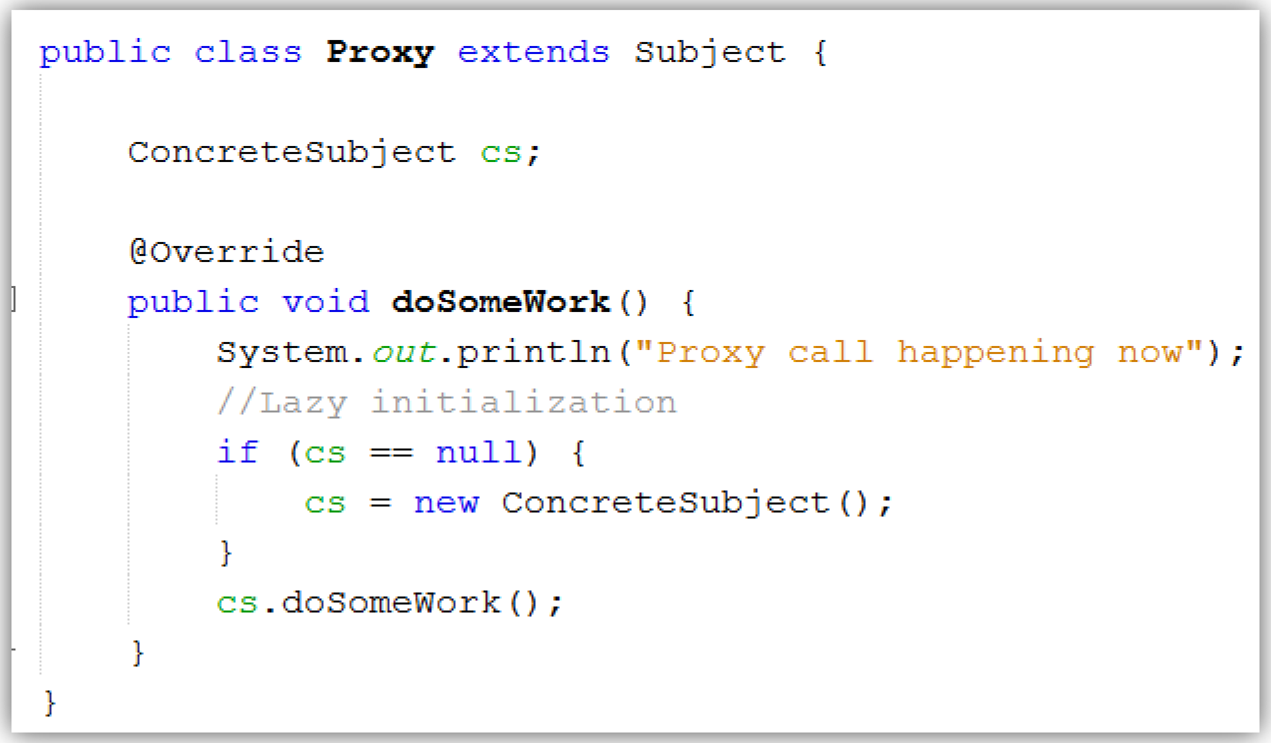
```
public abstract class Subject {

    public abstract void doSomeWork();
}
```

RealSubject



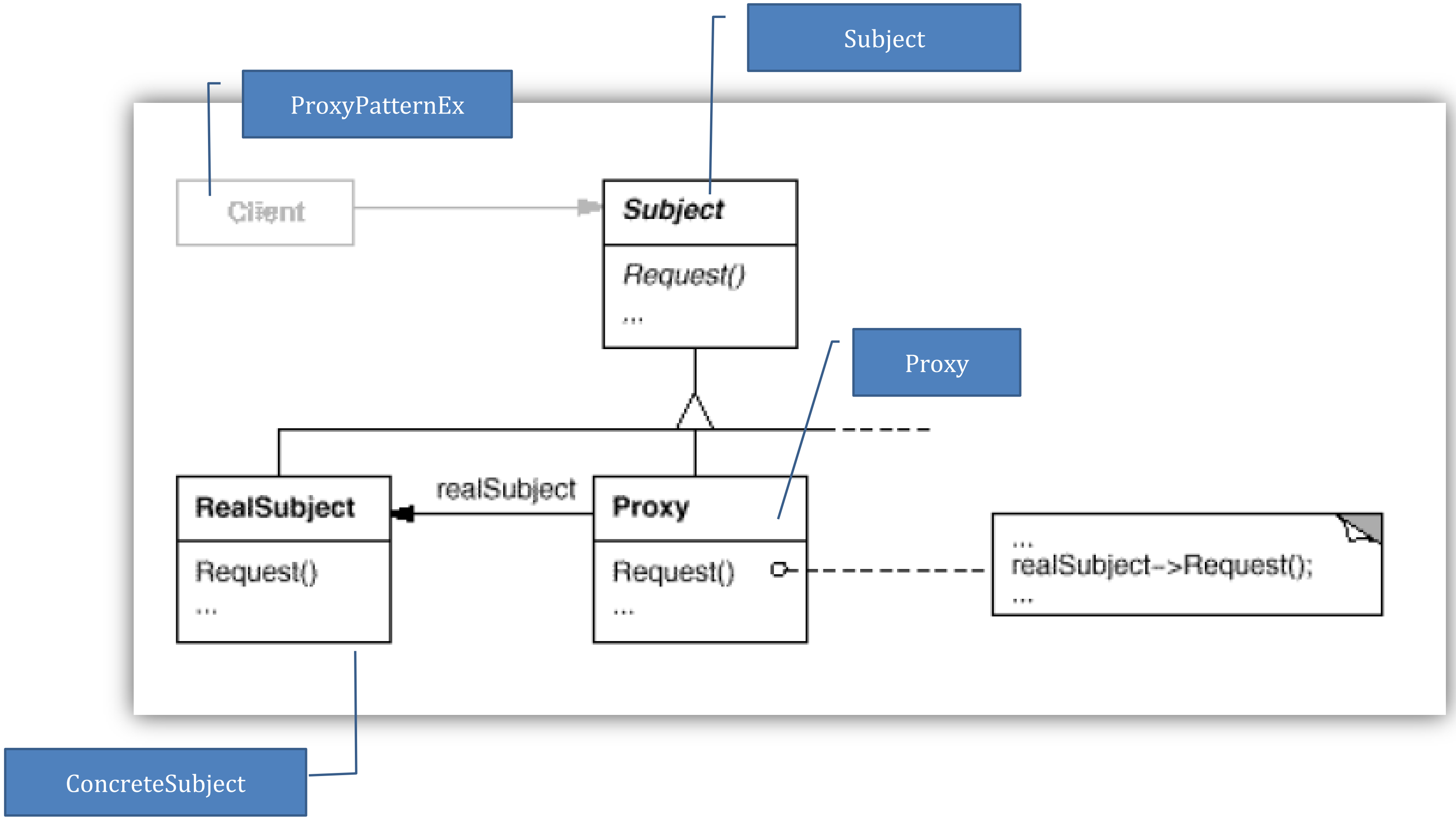
```
public class ConcreteSubject extends Subject {

    @Override
    public void doSomeWork() {
        System.out.println(" I am from concrete subject");
    }
}
```

Proxy



```
public class Proxy extends Subject {

    ConcreteSubject cs;

    @Override
    public void doSomeWork() {
        System.out.println("Proxy call happening now");
        //Lazy initialization
        if (cs == null) {
            cs = new ConcreteSubject();
        }
        cs.doSomeWork();
    }
}
```

Participants:



Participants

Participant	Class Or Interface
Proxy	Proxy class
Subject	Subject class
ConcreteSubject	ConcreteSubject class
Client	ProxyPatternEx class

Flyweight Pattern

Definition:

“Use sharing to support large numbers of fine-grained objects efficiently.”

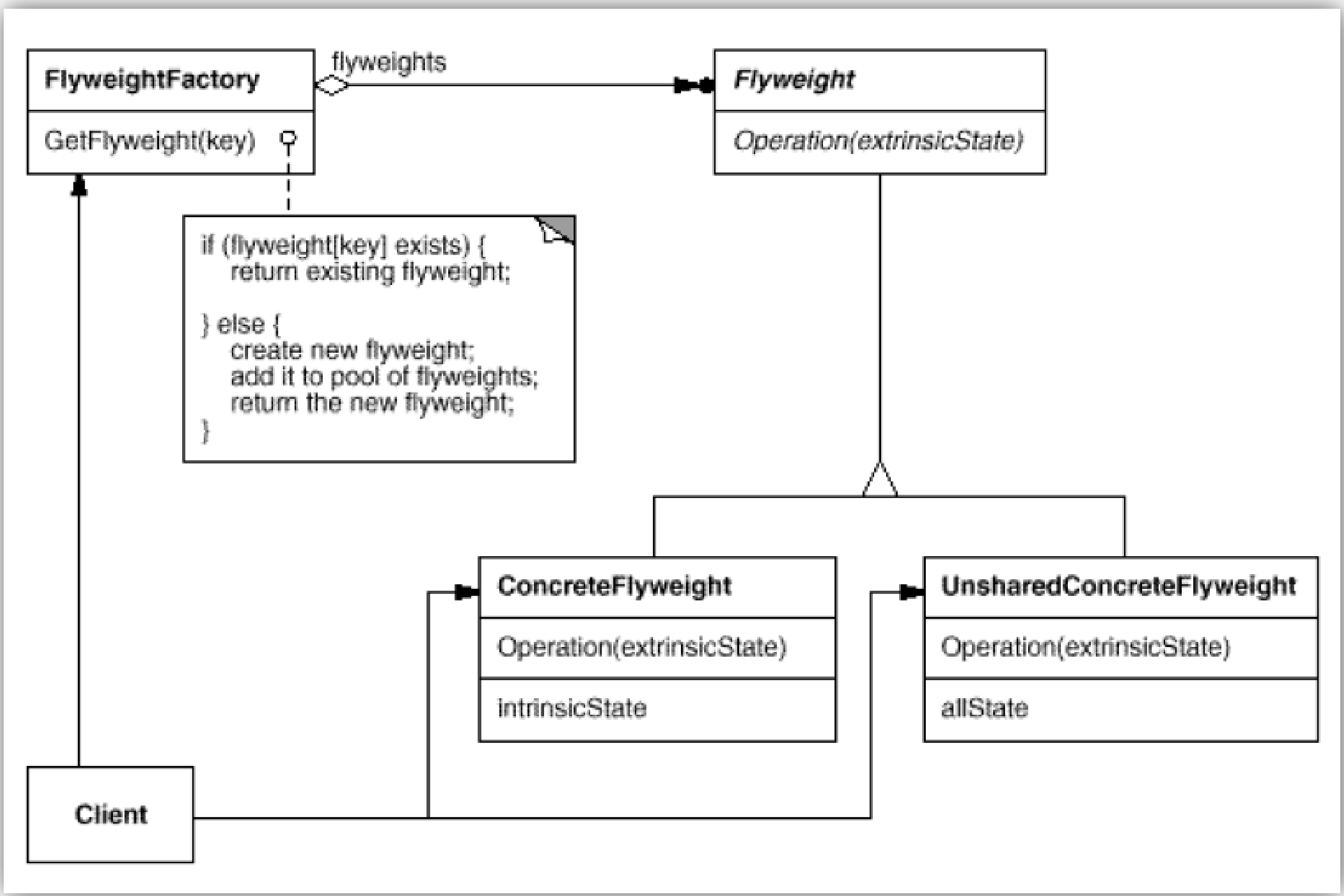
Concept

- 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 and extrinsic state.
- 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.
- Client objects need to pass the extrinsic state to the flyweight.

Where to use

- When there is a very large number of objects that may not fit in memory.
- When most of an objects state can be stored on disk or calculated at runtime.
- When there are groups of objects that share state.

Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

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. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight**
  - not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure.

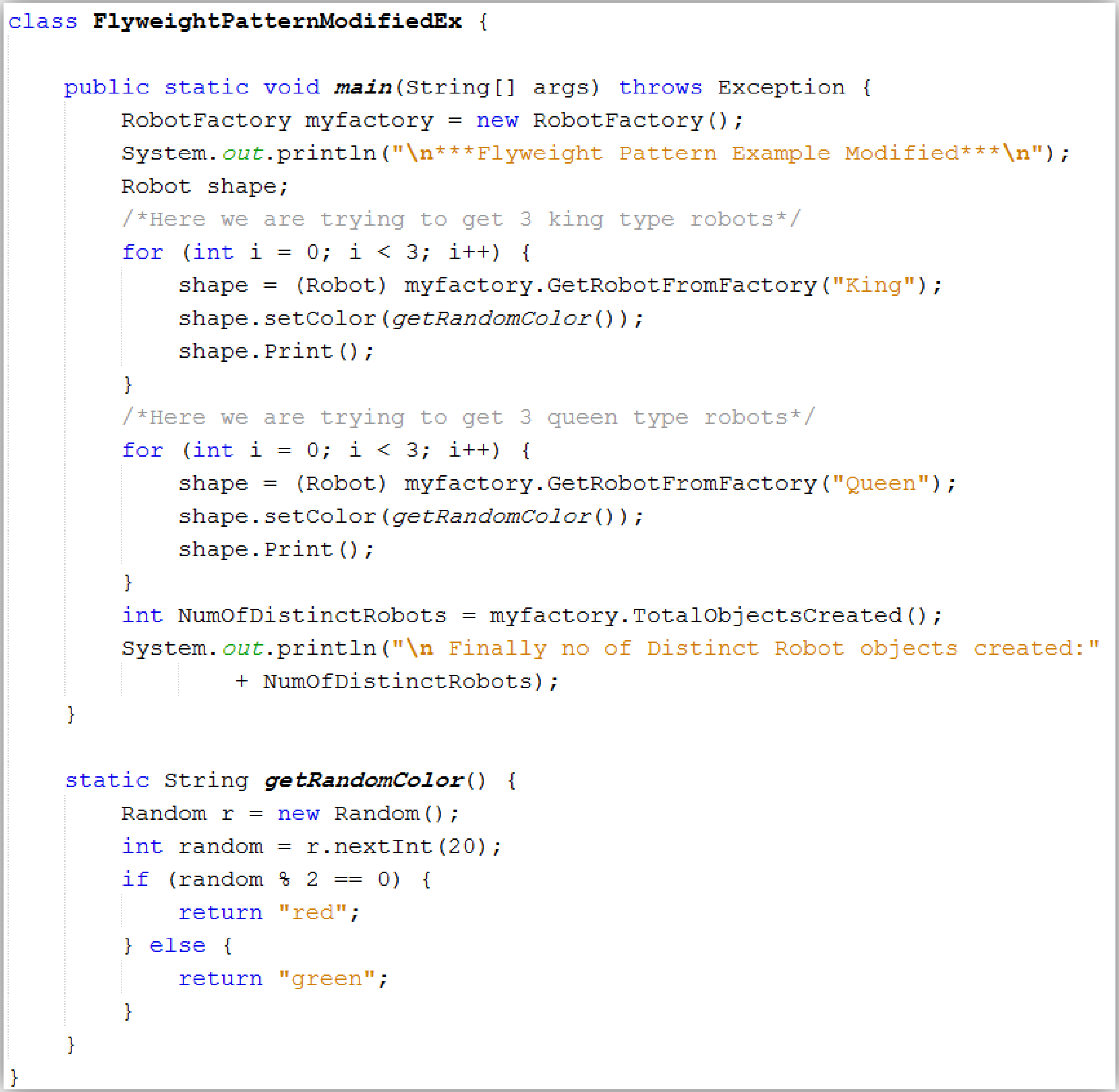
Participants

- **FlyweightFactory**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.
- **Client**
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s). .

Example:

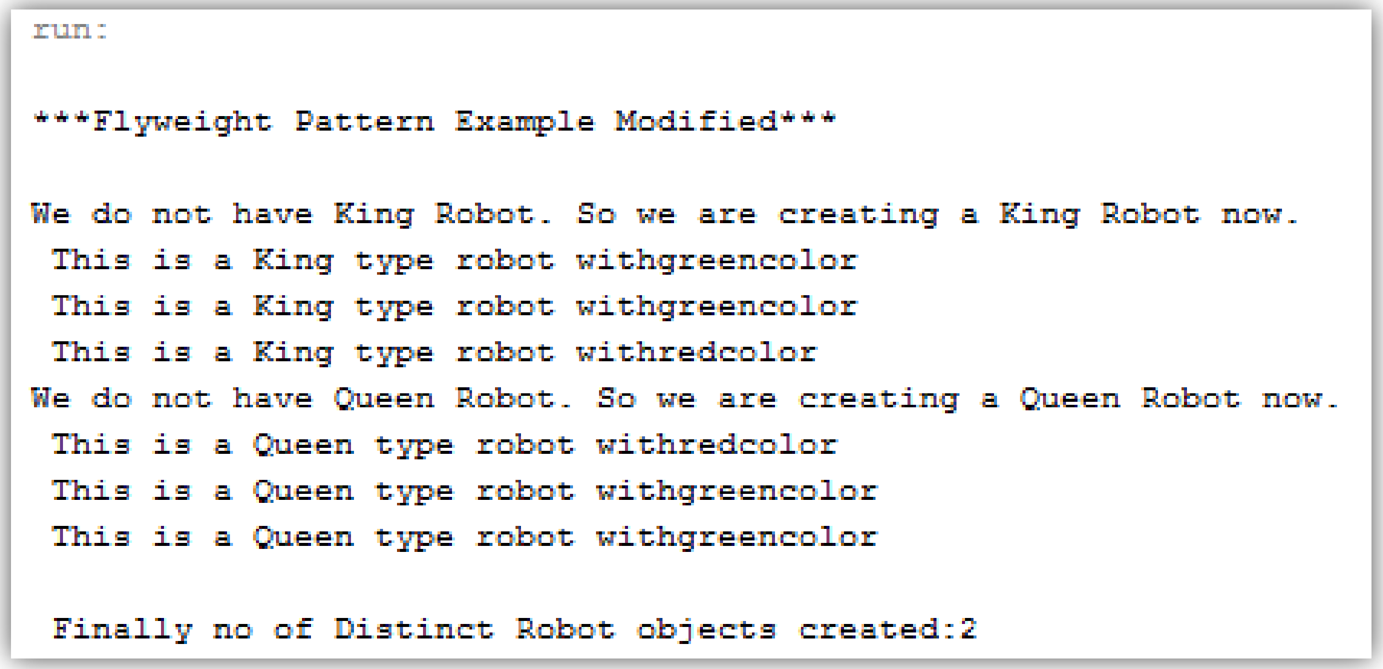
- In this example, we are dealing with robots which can be either king type or queen type.
- Each of these types can be either green or red.
- Before making any robot, we'll consult with our factory. If we already have king or queen types of robots, we'll not create them again. We will collect the basic structure from our factory and after that we'll color them.
- The color is extrinsic data here, but the category of robot (king or queen) is intrinsic.

Example:

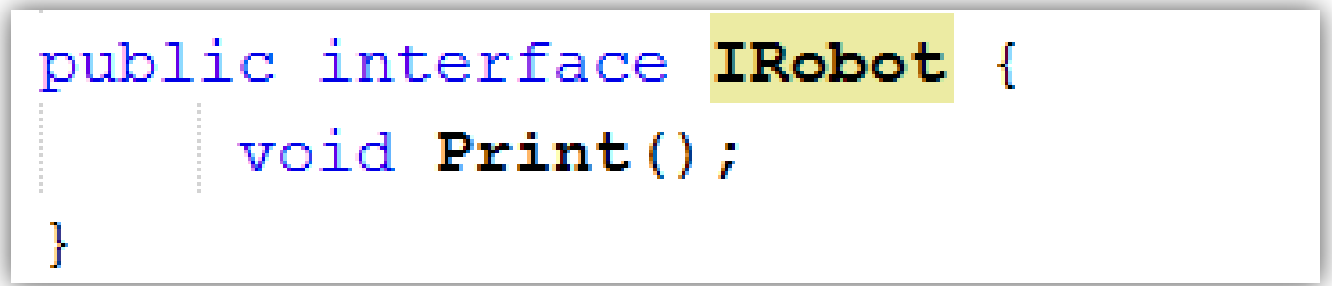


```java
class FlyweightPatternModifiedEx {

    public static void main(String[] args) throws Exception {
        RobotFactory myfactory = new RobotFactory();
        System.out.println("\n***Flyweight Pattern Example Modified***\n");
        Robot shape;
        /*Here we are trying to get 3 king type robots*/
        for (int i = 0; i < 3; i++) {
            shape = (Robot) myfactory.GetRobotFromFactory("King");
            shape.setColor(getRandomColor());
            shape.Print();
        }
        /*Here we are trying to get 3 queen type robots*/
        for (int i = 0; i < 3; i++) {
            shape = (Robot) myfactory.GetRobotFromFactory("Queen");
            shape.setColor(getRandomColor());
            shape.Print();
        }
        int NumOfDistinctRobots = myfactory.TotalObjectsCreated();
        System.out.println("\n Finally no of Distinct Robot objects created:"
                + NumOfDistinctRobots);
    }

    static String getRandomColor() {
        Random r = new Random();
        int random = r.nextInt(20);
        if (random % 2 == 0) {
            return "red";
        } else {
            return "green";
        }
    }
}
```

Output:

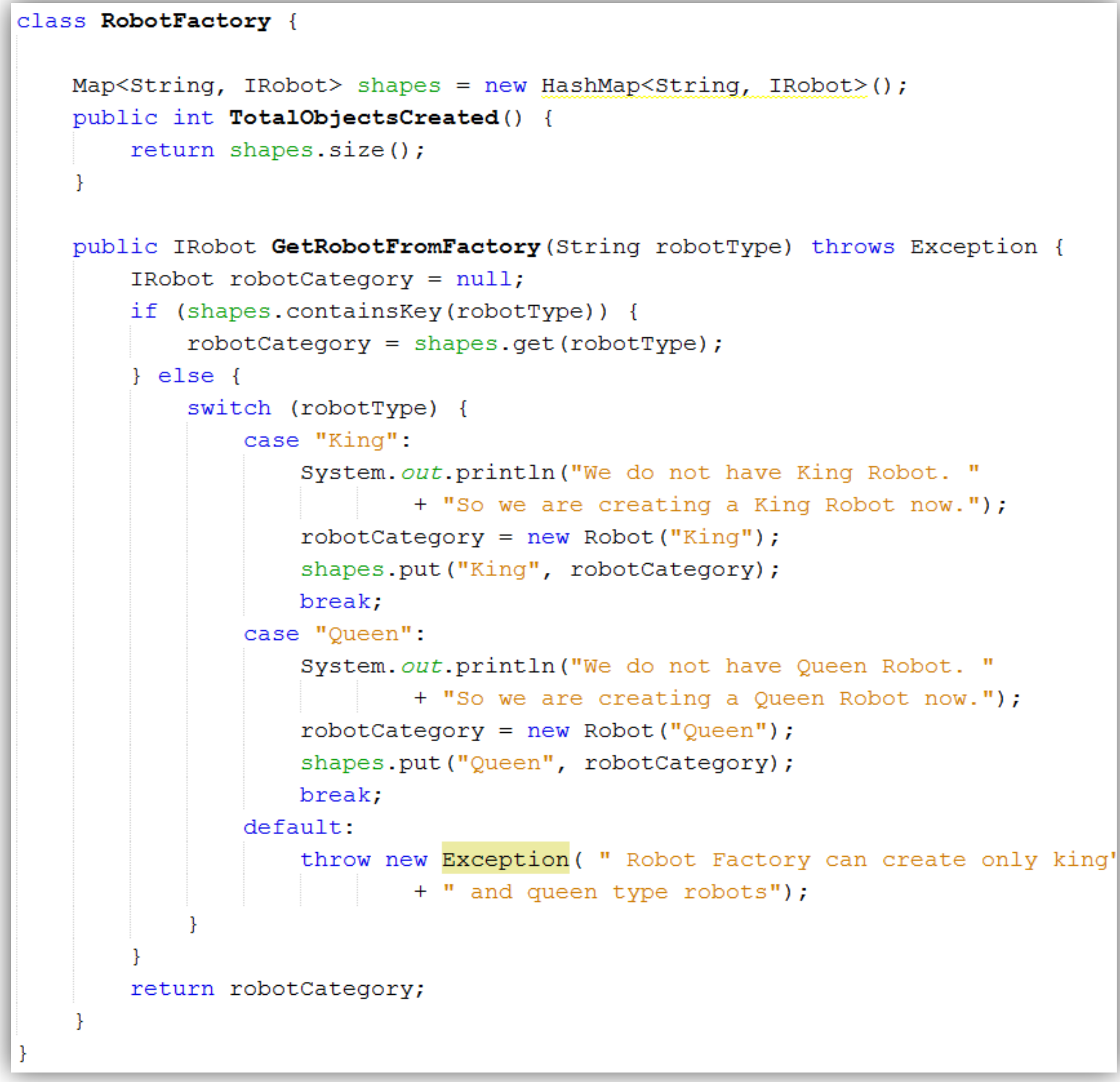

```
run:

***Flyweight Pattern Example Modified***

We do not have King Robot. So we are creating a King Robot now.
 This is a King type robot withgreencolor
 This is a King type robot withgreencolor
 This is a King type robot withredcolor
We do not have Queen Robot. So we are creating a Queen Robot now.
 This is a Queen type robot withredcolor
 This is a Queen type robot withgreencolor
 This is a Queen type robot withgreencolor

 Finally no of Distinct Robot objects created:2
```

IRobot



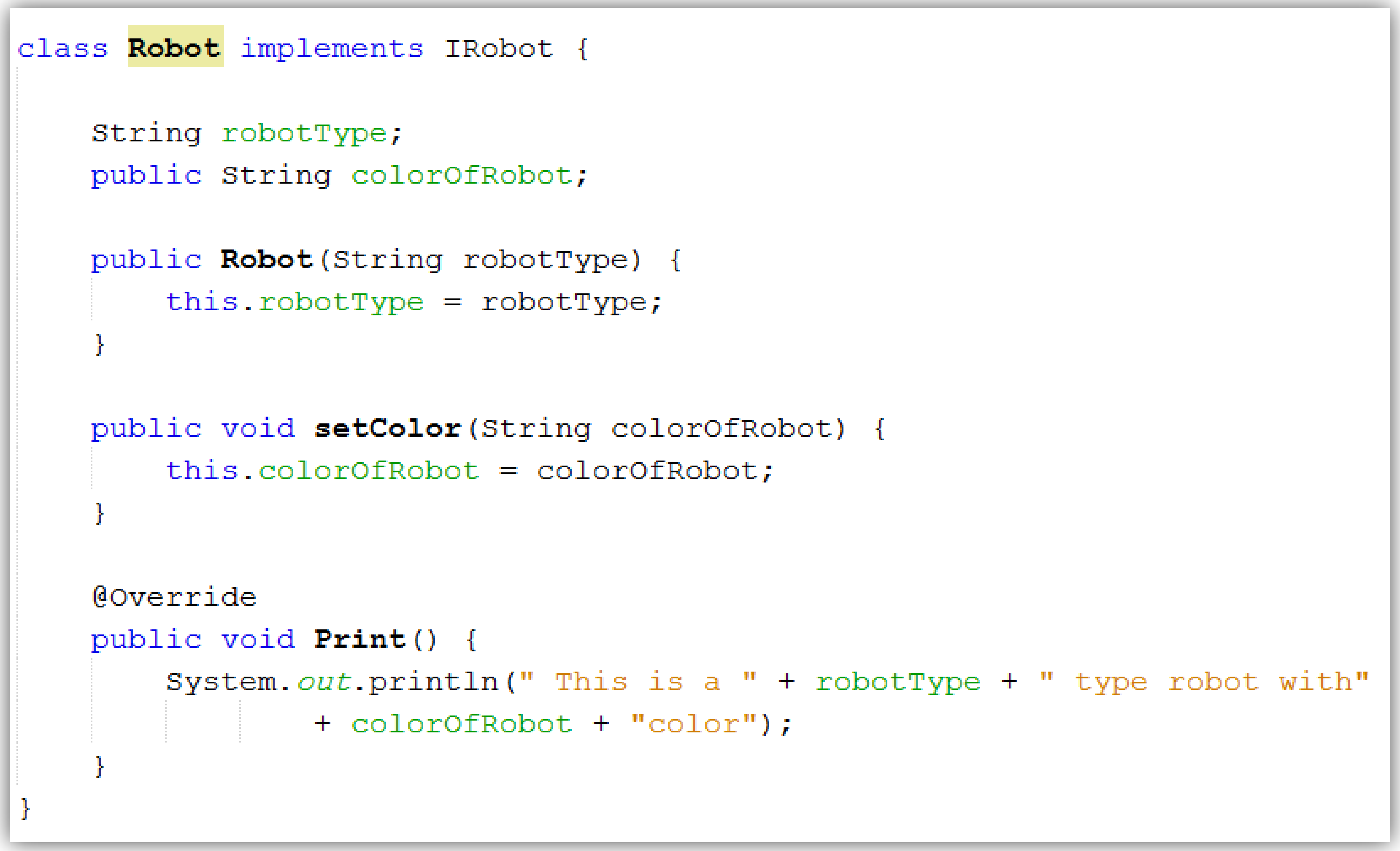
```
public interface IRobot {
	void Print();
}
```

RobotFactory

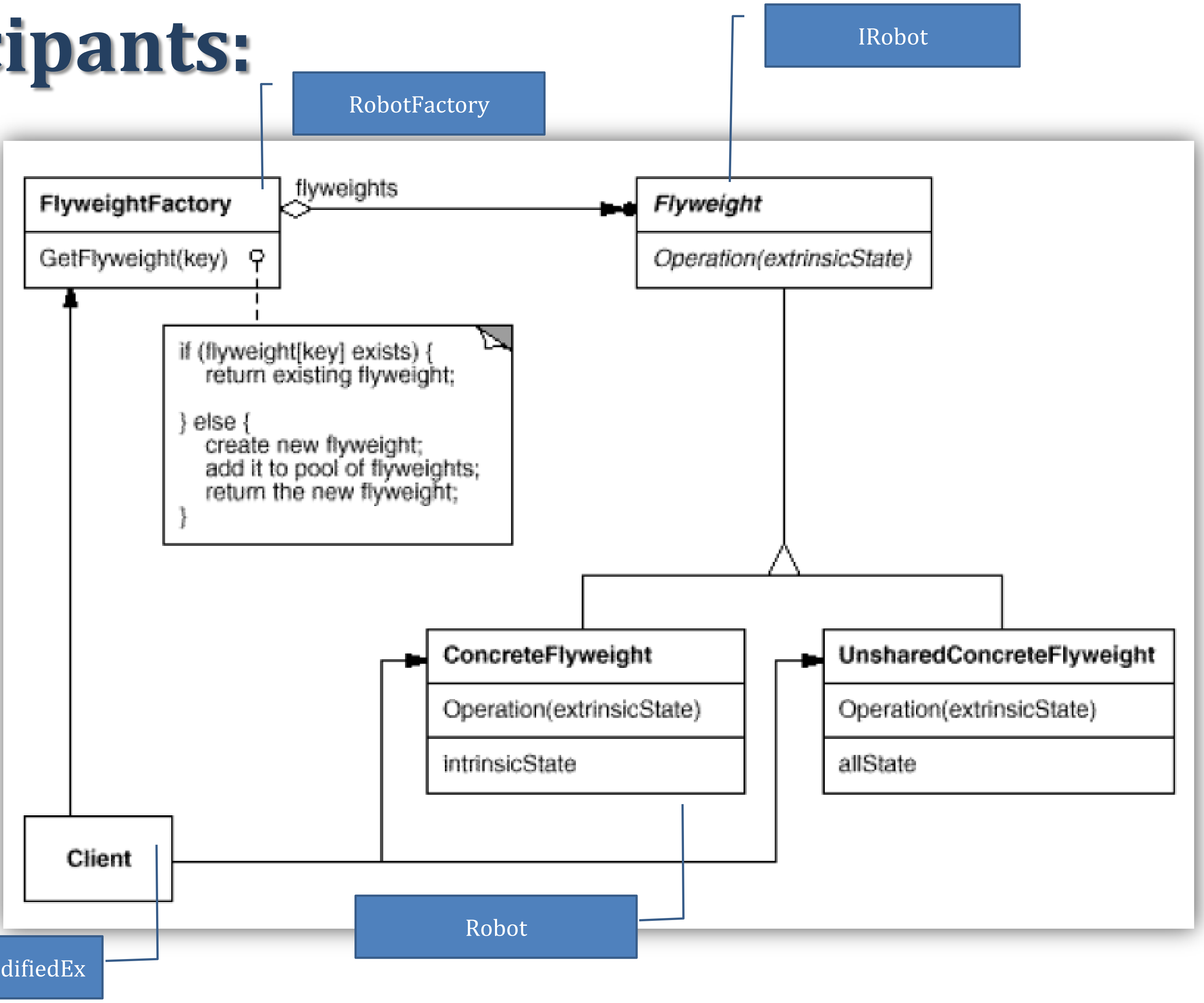


```
class RobotFactory {

    Map<String, IRobot> shapes = new HashMap<String, IRobot>();
    public int TotalObjectsCreated() {
        return shapes.size();
    }

    public IRobot GetRobotFromFactory(String robotType) throws Exception {
        IRobot robotCategory = null;
        if (shapes.containsKey(robotType)) {
            robotCategory = shapes.get(robotType);
        } else {
            switch (robotType) {
                case "King":
                    System.out.println("We do not have King Robot. "
                            + "So we are creating a King Robot now.");
                    robotCategory = new Robot("King");
                    shapes.put("King", robotCategory);
                    break;
                case "Queen":
                    System.out.println("We do not have Queen Robot. "
                            + "So we are creating a Queen Robot now.");
                    robotCategory = new Robot("Queen");
                    shapes.put("Queen", robotCategory);
                    break;
                default:
                    throw new Exception( " Robot Factory can create only king'
                            + " and queen type robots");
            }
        }
        return robotCategory;
    }
}
```

Robot


```
class Robot implements IRobot {

    String robotType;
    public String colorOfRobot;

    public Robot(String robotType) {
        this.robotType = robotType;
    }

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

    @Override
    public void Print() {
        System.out.println(" This is a " + robotType + " type robot with"
                + colorOfRobot + "color");
    }
}
```

Participants:



RobotFactory
IRobot
FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client
Robot
FlyweightPatternModifiedEx

Participants

Participant	Class Or Interface
Flyweight	Irobot interface
ConcreteFlyweight	Robot class
FlyweightFactory	RobotFactory class
Client	FlyweightPatternModifiedEx class

Thanks !