Composite Design Pattern

Design Patterns

Purpose		
Creational	Structural	Behavioral
Factory Abstract Factory Builder Prototype Singleton	Composite Adapter Bridge Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Structural Patterns - Composite

Intent

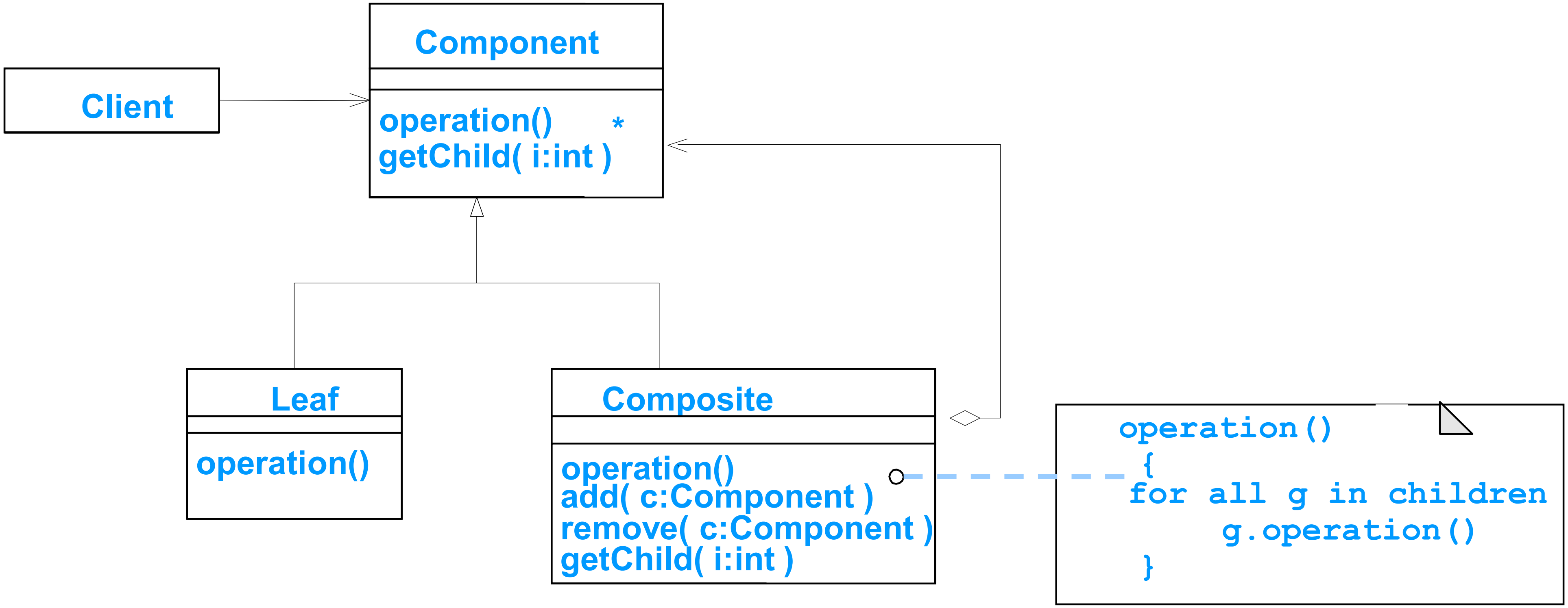
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Composite: Applicability

- Represents part-whole hierarchies of objects.
- Clients ignore the difference between compositions of objects and individual objects.
- Clients treat all objects in the composite structure uniformly.

Structural Patterns – Composite

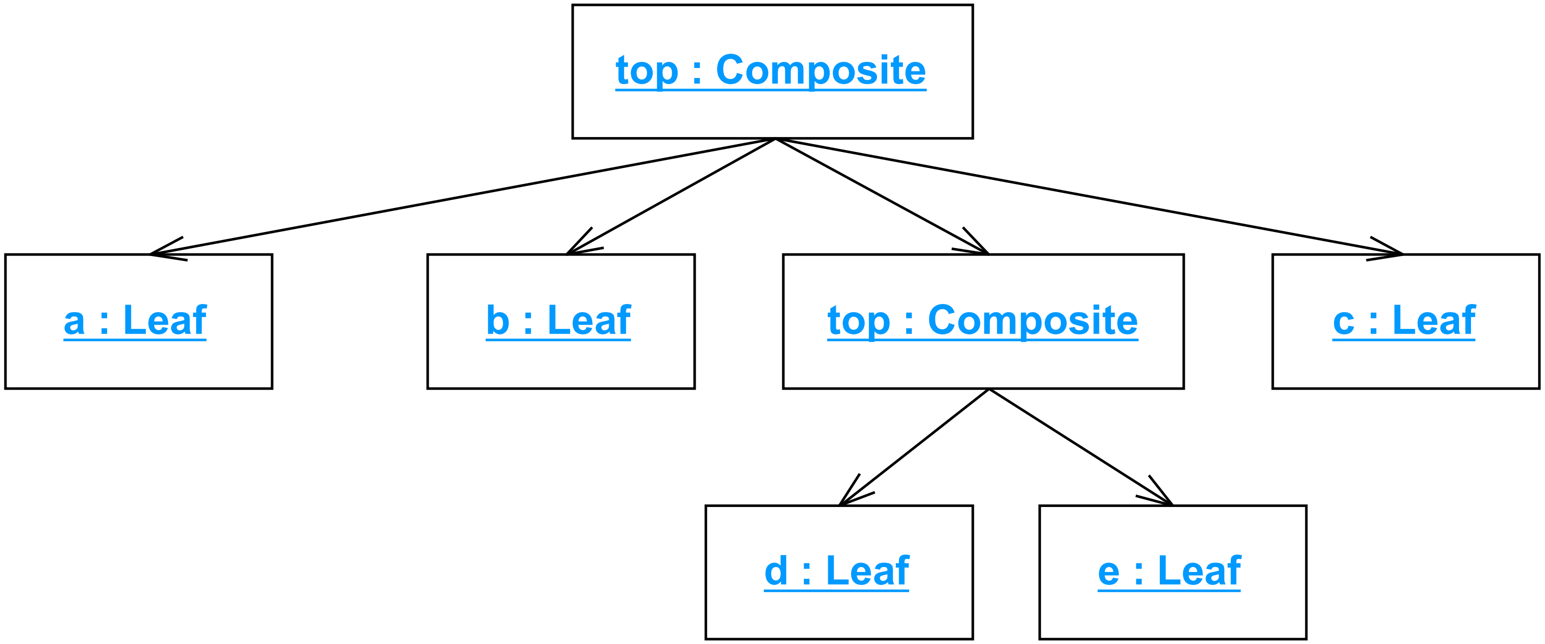
Class Diagram



```
operation()
 {
for all g in children
     g.operation()
 }
```

Structural Patterns - Composite

Object Diagram



Structural Patterns – Composite

Participants

Component

- Declares the interface for objects in the composition.
- Implements default behavior for the interface common to all classes, as appropriate.
- Declares an interface for accessing and managing its child components.
- Optionally defines an interface for accessing a components parent.

Leaf

- Represents leaf objects in the composition.
- Defines behavior for primitive objects in the composition.

Composite

- Defines behavior for components having children.
- Stores child components.
- Implements child-related operations.

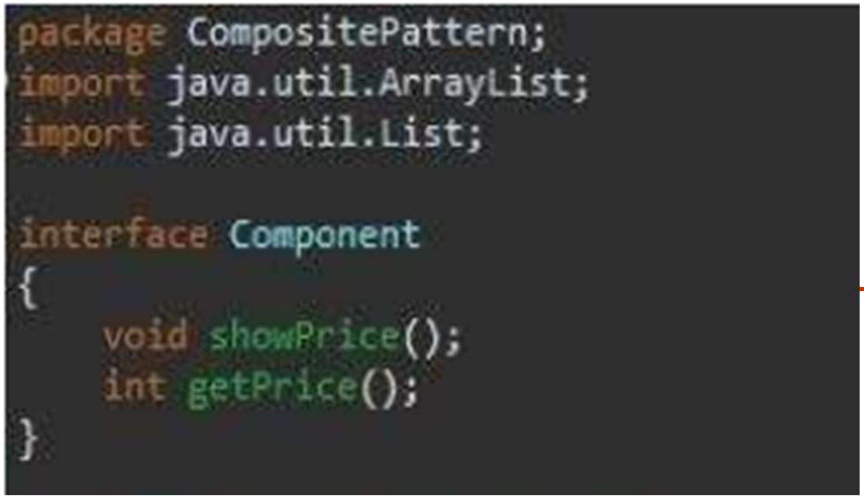
Client

- Manipulates objects in the composition through the Component interface.

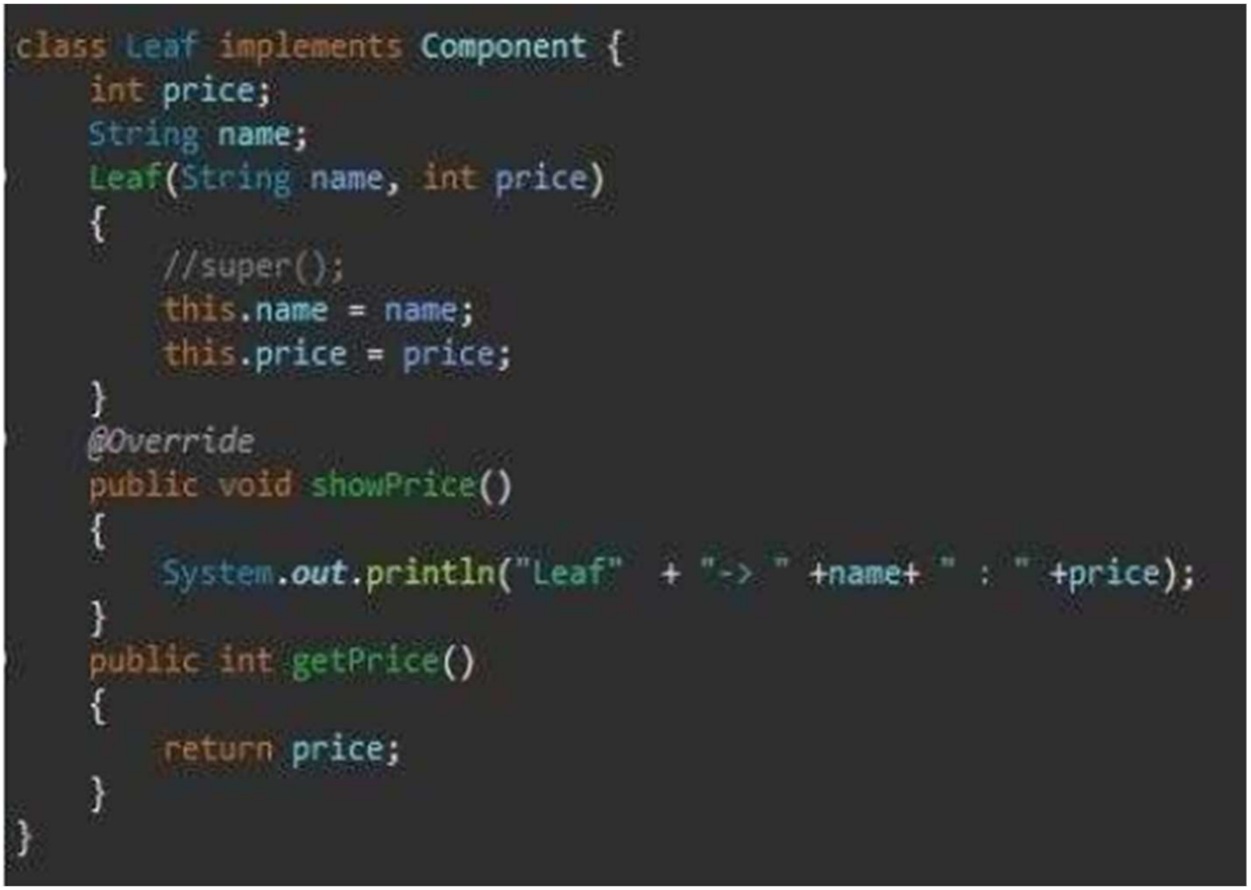
Structural Patterns – Composite

Collaborations

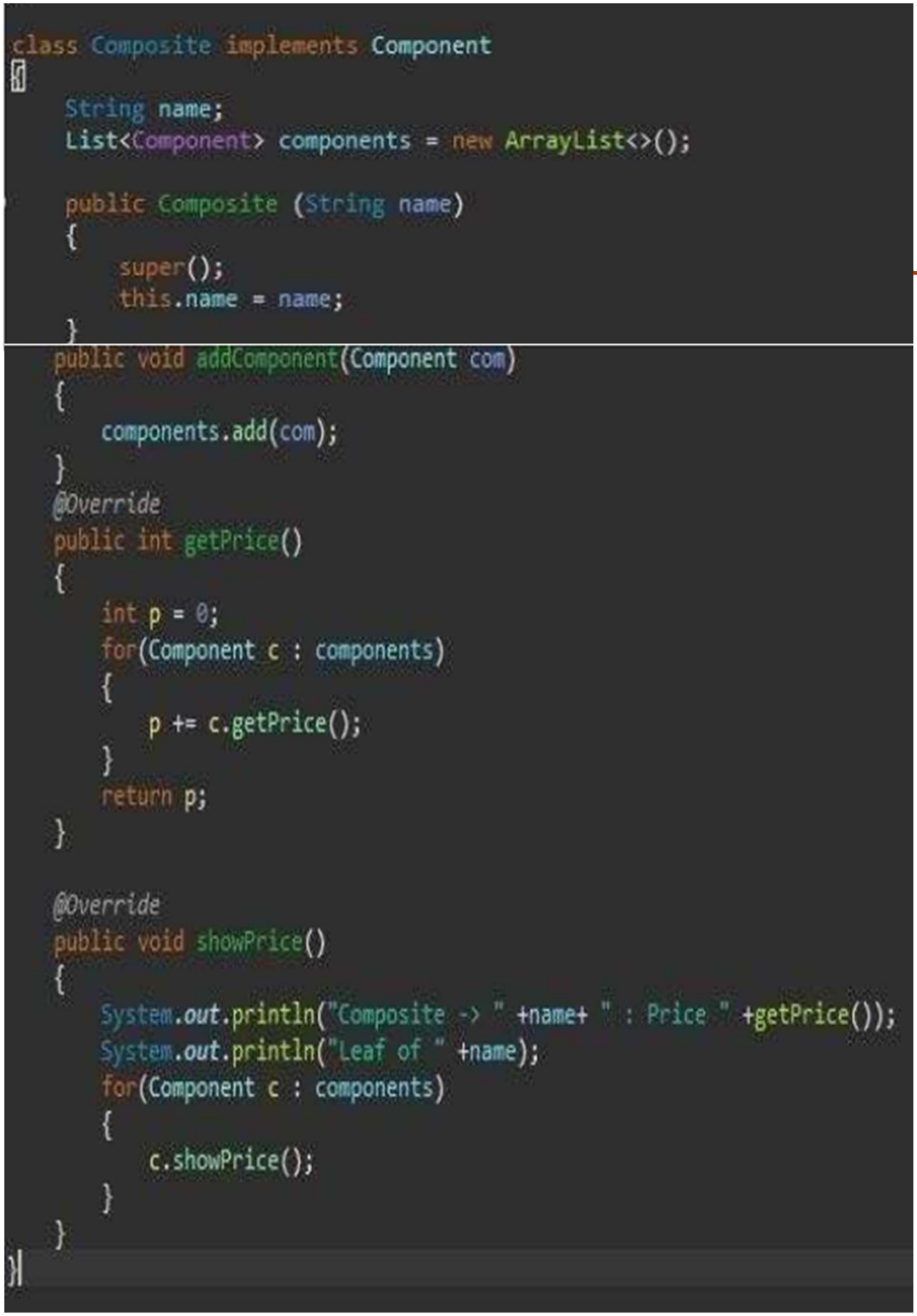
- Clients use the Component class interface to interact with objects in the composite structure.
- If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.



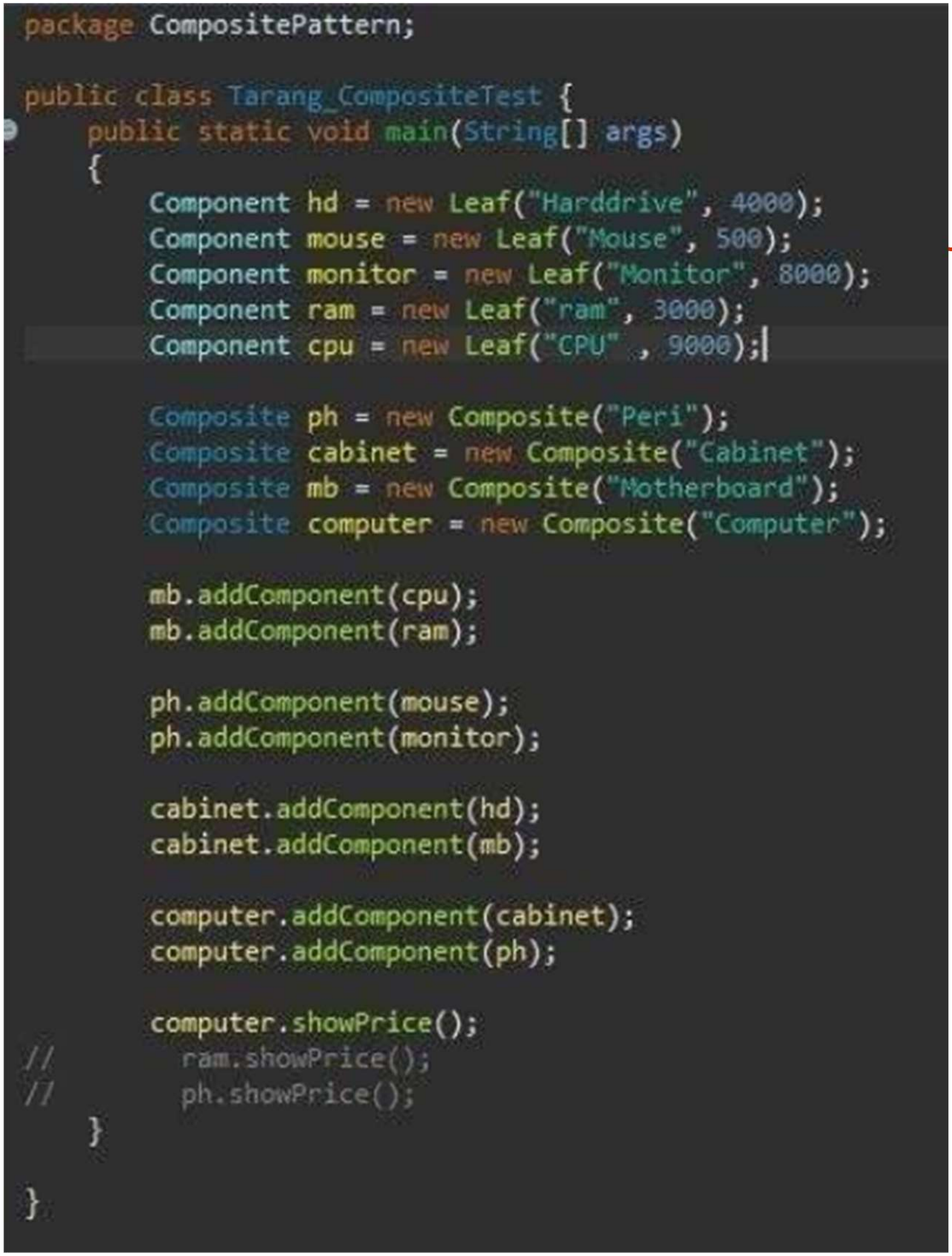
```
package CompositePattern;
import java.util.ArrayList;
import java.util.List;

interface Component
{
    void showPrice();
    int getPrice();
}
```



```
class Leaf implements Component {
    int price;
    String name;
    Leaf(String name, int price)
    {
        //super();
        this.name = name;
        this.price = price;
    }
    @Override
    public void showPrice()
    {
        System.out.println("Leaf"  + "-> " +name+ " : " +price);
    }
    public int getPrice()
    {
        return price;
    }
}
```

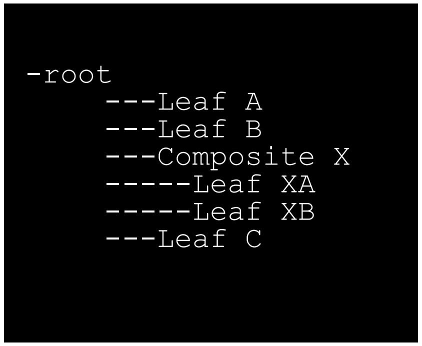


```
class Composite implements Component
{
    String name;
    List<Component> components = new ArrayList<>();

    public Composite (String name)
    {
        super();
        this.name = name;
    }
    public void addComponent(Component com)
    {
        components.add(com);
    }
    @Override
    public int getPrice()
    {
        int p = 0;
        for(Component c : components)
        {
            p += c.getPrice();
        }
        return p;
    }

    @Override
    public void showPrice()
    {
        System.out.println("Composite -> " +name+ " : Price " +getPrice());
        System.out.println("Leaf of " +name);
        for(Component c : components)
        {
            c.showPrice();
        }
    }
}
```



```java
package CompositePattern;

public class Tarang_CompositeTest {
    public static void main(String[] args)
    {
        Component hd = new Leaf("Harddrive", 4000);
        Component mouse = new Leaf("Mouse", 500);
        Component monitor = new Leaf("Monitor", 8000);
        Component ram = new Leaf("ram", 3000);
        Component cpu = new Leaf("CPU" , 9000);

        Composite ph = new Composite("Peri");
        Composite cabinet = new Composite("Cabinet");
        Composite mb = new Composite("Motherboard");
        Composite computer = new Composite("Computer");

        mb.addComponent(cpu);
        mb.addComponent(ram);

        ph.addComponent(mouse);
        ph.addComponent(monitor);

        cabinet.addComponent(hd);
        cabinet.addComponent(mb);

        computer.addComponent(cabinet);
        computer.addComponent(ph);

        computer.showPrice();
//          ram.showPrice();
//          ph.showPrice();
    }

}
```

```
using System;
using System.Collections;

namespace DoFactory.GangOfFour.Composite.Structural
{

 // MainApp test application

 class MainApp
 {
  static void Main()
  {
   // Create a tree structure
   Composite root = new Composite("root");
   root.Add(new Leaf("Leaf A"));
   root.Add(new Leaf("Leaf B"));

   Composite comp = new Composite("Composite X");
   comp.Add(new Leaf("Leaf XA"));
   comp.Add(new Leaf("Leaf XB"));

   root.Add(comp);
   root.Add(new Leaf("Leaf C"));

   // Add and remove a leaf
   Leaf leaf = new Leaf("Leaf D");
   root.Add(leaf);
   root.Remove(leaf);

   // Recursively display tree
   root.Display(1);

   // Wait for user
   Console.Read();
  }
 }
```



```
-root
    ---Leaf A
    ---Leaf B
    ---Composite X
    -----Leaf XA
    -----Leaf XB
    ---Leaf C
```

```
// "Component"
 abstract class Component
 {protected string name;

  // Constructor
  public Component(string name)
  {this.name = name;}

  public abstract void Add(Component c);
  public abstract void Remove(Component c);
  public abstract void Display(int depth);
 }

 // "Composite"
 class Composite : Component
 {private ArrayList children = new ArrayList();

  // Constructor
  public Composite(string name) : base(name) {  }

  public override void Add(Component component)
  {children.Add(component);}

  public override void Remove(Component component)
  {children.Remove(component);}

  public override void Display(int depth)
  {Console.WriteLine(new String('-', depth) + name);

   // Recursively display child nodes
   foreach (Component component in children)
   {component.Display(depth + 2);}
  }
 }
 // "Leaf"
 class Leaf : Component
 {// Constructor
  public Leaf(string name) : base(name) {  }

  public override void Add(Component c)
  {Console.WriteLine("Cannot add to a leaf");}

  public override void Remove(Component c)
  {Console.WriteLine("Cannot remove from a leaf");}

  public override void Display(int depth)
  {Console.WriteLine(new String('-', depth) + name);}
 }
}
```