Software Design and Architecture

[Structural Design Patterns] – Chapter 07, L01

Lecture Outlines

- **Previously (Creational Design Patterns)**
- **Structural Patterns in Detailed Design**
  - ✓Adaptor
  - ✓Composite
  - ✓Facade
- **What's next…**

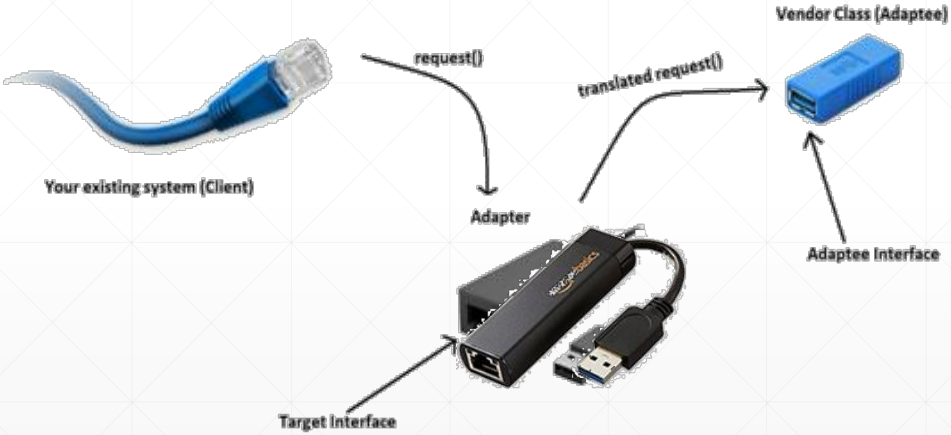
Structural Patterns in Detailed Design

- Structural design patterns are patterns that deal with designing the **structure** of **existing classes** or **objects** at run .
- Popular structural design patterns include:
  - ✓Adaptor
  - ✓Facade
  - ✓Composite

Adapter Design Pattern

- The **Adapter** design pattern is a class/object **structural** design pattern used to adapt an exiting interface that is expected in a software system.
- The **intent** of the Adapter is to:
  - ✓ Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Adapter Design Pattern-



A real-life example for the use of adapters

The adapter design pattern solves problems like:[3]

- How can a class be reused that does not have an interface that a client requires?
- How can classes that have incompatible interfaces work together?
- How can an alternative interface be provided for a class?

Often an (already existing) class can't be reused only because its interface doesn't conform to the interface clients require.

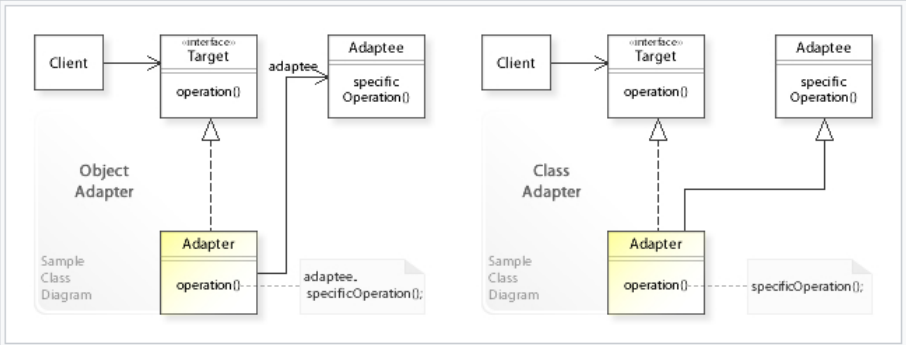
The adapter design pattern describes how to solve such problems:

- Define a separate `adapter` class that converts the (incompatible) interface of a class (`adaptee`) into another interface (`target`) clients require.
- Work through an `adapter` to work with (reuse) classes that do not have the required interface.

The key idea in this pattern is to work through a separate `adapter` that adapts the interface of an (already existing) class without changing it.

Clients don't know whether they work with a `target` class directly or through an `adapter` with a class that does not have the `target` interface.

See also the UML class diagram below.



A sample UML class diagram for the adapter design pattern.[5]

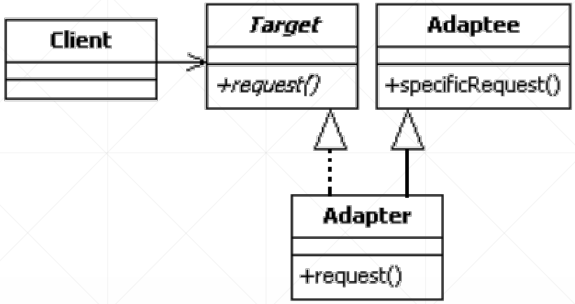
- **In the above UML class diagram, the Client class that requires (depends on) a Target Interface cannot reuse the Adaptee class directly because its interface doesn't conform to the Target Interface. Instead, the Client works through an Adapter class that implements the Target Interface in terms of Adaptee:**

(during the running time)

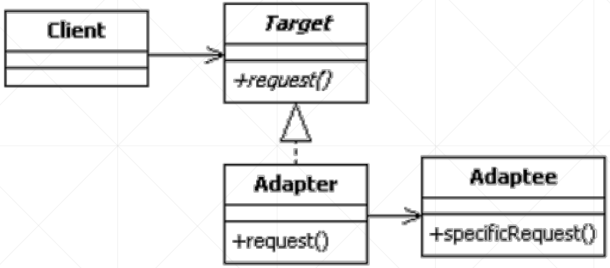
The Object Adapter implements the Target Interface by delegating to an Adaptee object at run-time (adaptee.specificOperation()).

The Class Adapter implements the Target Interface by inheriting from an Adaptee class at compile-time (specificOperation()). some target interface

Adapter Design Pattern - Structure



Class Structural version of Adapter Pattern



Object Structural version of Adapter Pattern

To use an adapter:

1.The client makes a request to the adapter by calling a method on it using the target interface.

2.The adapter translates that request on the adaptee using the adaptee interface.

3.Client receive the results of the call

In Online CLASS we will see Several Examples Final Way

Adapter Design Pattern – Example*

Example:

- Suppose you have a **Bird** class with *fly()* , and *makeSound()* methods.
- Also a **ToyDuck** class with *squeak()* method.
- Let's assume that you are short on ToyDuck objects and you would like to use Bird objects in their place.
- Birds have some similar functionality but implement a different interface, so we can't use them directly.
- So we will use **adapter pattern**. Here our **target** would be **ToyDuck** and **adaptee** would be **Bird**.

* Source: http://www.geeksforgeeks.org/adapter-pattern/

Adapter Design Pattern – Example

```
interface ToyDuck
{
    // target interface
    // toyducks dont fly they just make
    // squeaking sound
    public void squeak();
}
```

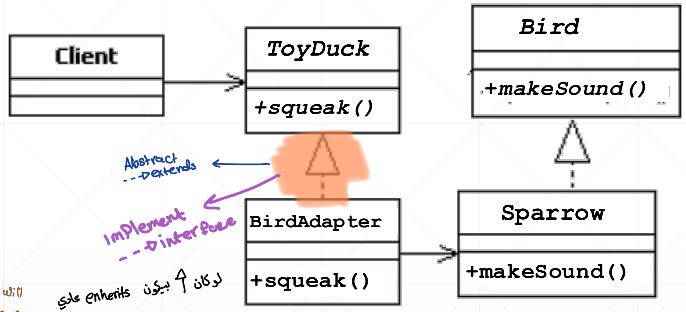
```
class PlasticToyDuck implements ToyDuck
{
    public void squeak()
    {
        System.out.println("Squeak");
    }
}
```

```
interface Bird
{
    // birds implement Bird interface that allows
    // them to fly and make sounds adaptee interface
    public void fly();
    public void makeSound();
}
```

```
class Sparrow implements Bird
{
    // a concrete implementation of bird
    public void fly()
    {
        System.out.println("Flying");
    }
    public void makeSound()
    {
        System.out.println("Chirp Chirp");
    }
}
```

Adapter Design Pattern – Example

```java
class BirdAdapter implements ToyDuck
{
    // You need to implement the interface your
    // client expects to use.
    Bird bird;
    public BirdAdapter(Bird bird)
    {
        // we need reference to the object we
        // are adapting
        this.bird = bird;
    }

    public void squeak()
    {
        // translate the methods appropriately
        bird.makeSound();
    }
}
```



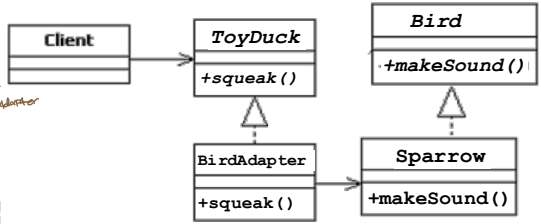
Object Structural version of Adapter Pattern

Adapter Design Pattern – Example

```
class Main
{
    public static void main(String args[])
    {
        Sparrow sparrow = new Sparrow();
        PlasticToyDuck toyDuck = new PlasticToyDuck();

        // Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();

        System.out.println("ToyDuck...");
        toyDuck.squeak();

        // toy duck behaving like a bird
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}
```

Creat TestClass (Clint) inside the main body we will Creat Object from the Adaptee + target + Adapter

Name of the object from target interface → new Object from ? called (Sparrow)



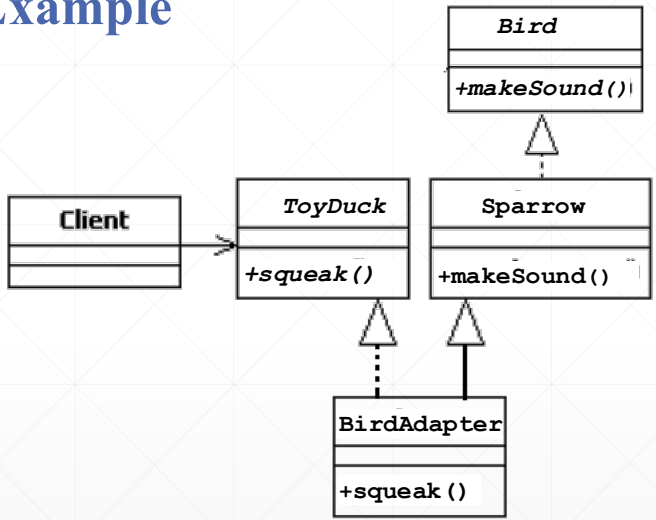
Object Structural version of Adapter Pattern

Output:

Adapter Design Pattern – Example

➢Object Adapter Vs Class Adapter

- ✓The adapter pattern we have implemented above is called **Object Adapter Pattern** because the adapter holds an instance of adaptee. There is also another type called **Class Adapter Pattern** which use inheritance instead of composition but you require multiple inheritance to implement it.

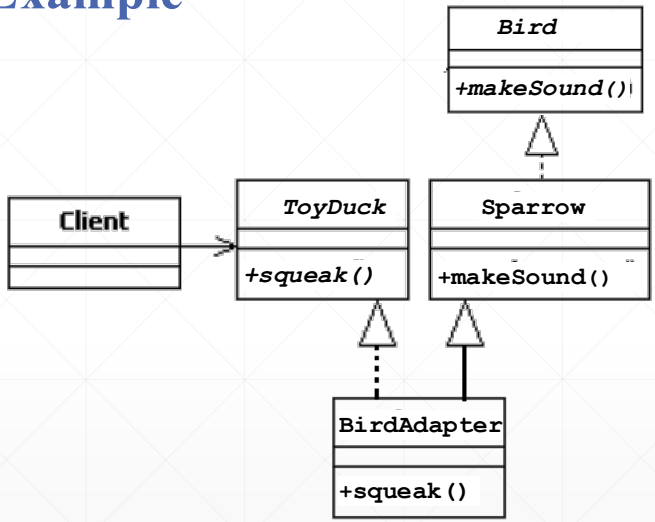


Class Structural version of Adapter Pattern

Adapter Design Pattern – Example

➢Object Adapter Vs Class Adapter

- ✓Here instead of having an adaptee object inside adapter (composition) to make use of its functionality the adapter (BirdAdapter) class **inherits** the adaptee (Sparrow) class.
- ✓Since **multiple inheritance** is not supported by many languages and is associated with many problems object adapter pattern is preferred



Class Structural version of Adapter Pattern

Step 4: Client Code

```java
public class Client {
    public static void main(String[] args) {
        // Create a Sparrow
        Sparrow sparrow = new Sparrow();

        // Wrap it inside an Adapter
        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        // Client expects a ToyDuck, but actually uses a Bird
        birdAdapter.squeak(); // Output: "Chirp chirp"
    }
}
```

- **Explanation**
- **ToyDuck** is the expected interface (Target).
- **Bird/Sparrow** is the legacy interface (Adaptee).
- **BirdAdapter** converts squeak() into makeSound().
- The **Client** works with ToyDuck but internally a Bird is used.

Adapter Design Pattern

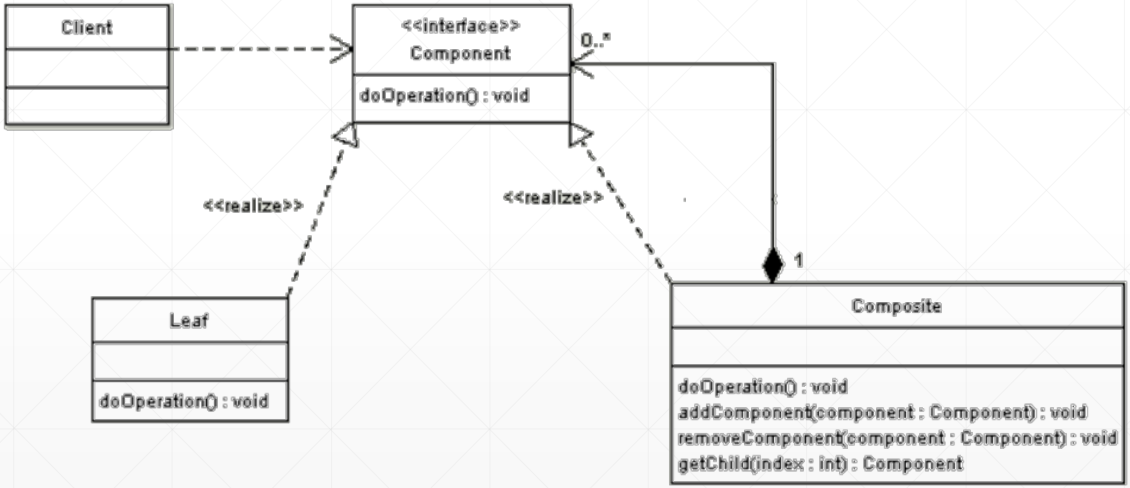
- **Advantages**:
  - ✓ Helps achieve reusability and flexibility.
  - ✓ Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.
- **Disadvantages**:
  - ✓ All requests are forwarded, so there is a slight increase in the overhead.
  - ✓ Sometimes many adaptations are required along an adapter chain to reach the type which is required.

The Composite Design Pattern

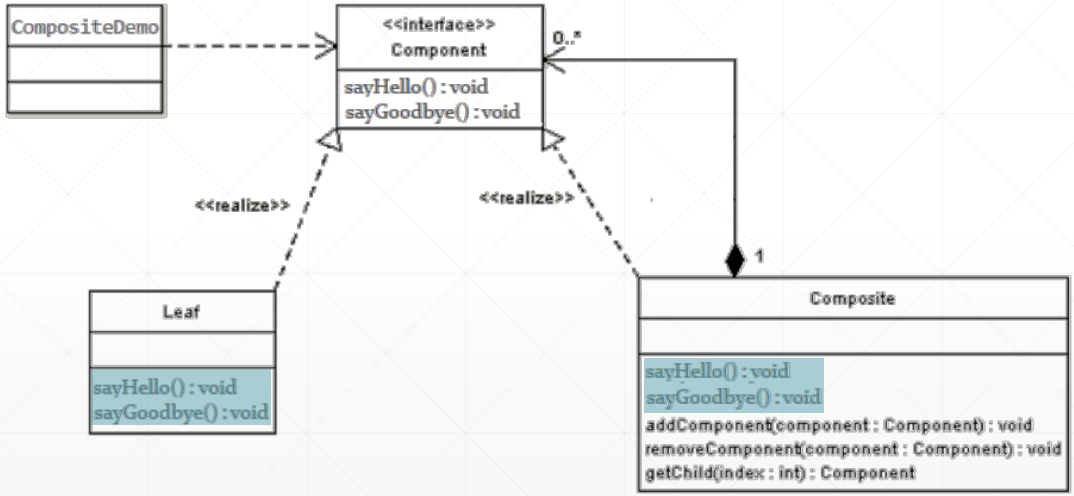
- The Composite design pattern is an **object structural pattern** that allows designers to compose tree-like design by strategically **structuring objects** that share a **whole-part relationship**.

- The **intent** of the Composite is to:
  - ✓ Compose objects into tree structures to represent **whole-part hierarchies**.
  - ✓ Composite pattern lets clients **treat** individual **objects** and **composites** of objects **uniformly**.

The Composite Design Pattern – Structure



The Composite Design Pattern – Example



The Composite Design Pattern – Example*

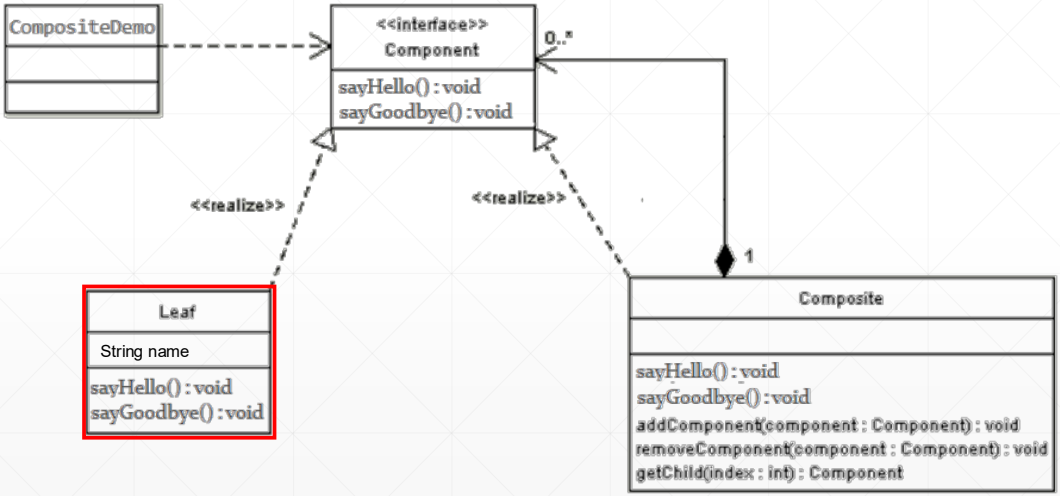
- The following slides have the Java code the represents the implementation of the classes in the Composite Design Pattern

File: Component.java

```
public interface Component {
   public void sayHello();
   public void sayGoodbye();
}
```


*source: https://www.avajava.com/tutorials/lessons/composite-pattern.html?page=1

The Composite Design Pattern – Example

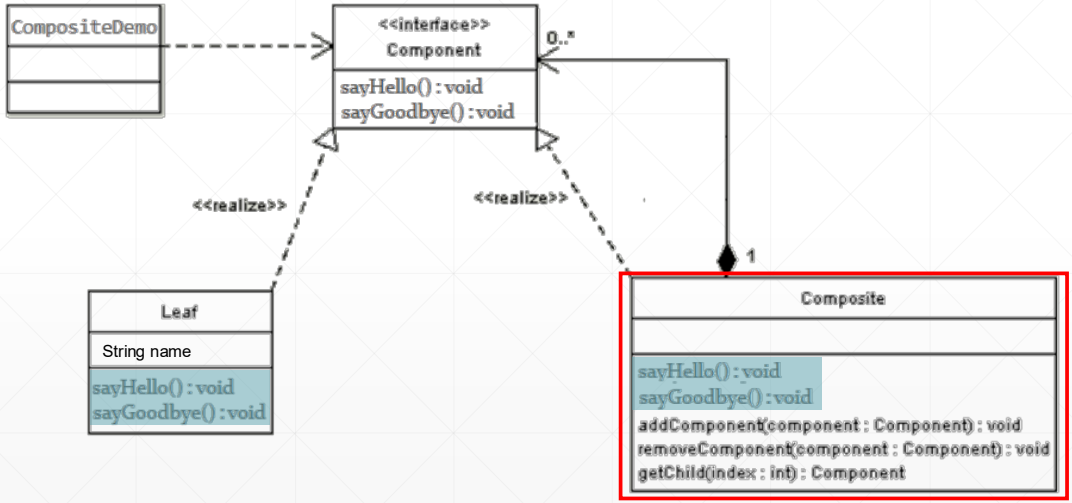


The Composite Design Pattern Example

File: Leaf.java

```
public class Leaf implements Component {

    String name;

    public Leaf(String name) {
            this.name = name;
    }

    @Override
    public void sayHello() {
            System.out.println(name + " leaf says hello");
    }

    @Override
    public void sayGoodbye() {
            System.out.println(name + " leaf says goodbye");
    }
}
```

The Composite Design Pattern – Example

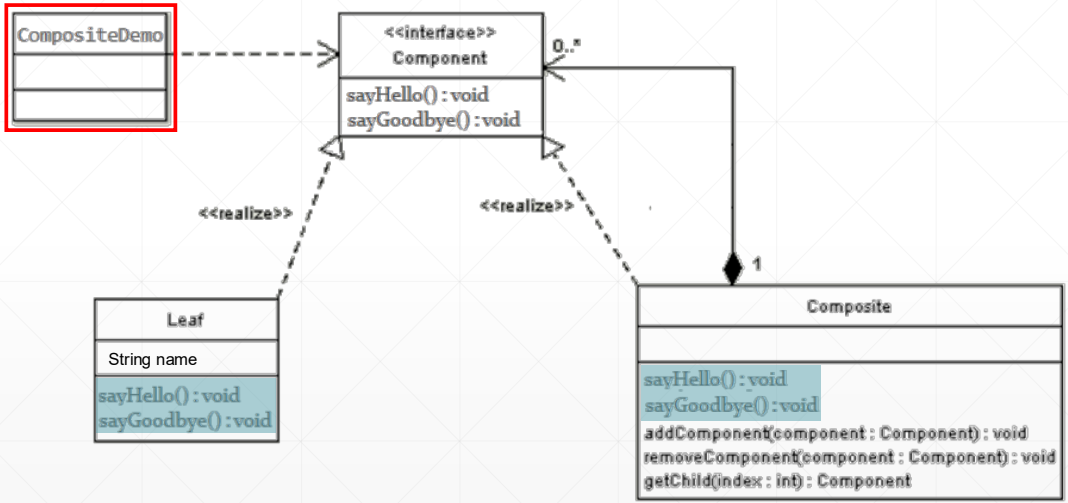


The Composite Design Pattern Example

File: Composite.java

```
import java.util.ArrayList;
import java.util.List;
public class Composite implements Component {
    List<Component> components = new ArrayList<Component>();

    @Override
    public void sayHello() {
            for (Component component : components) {
                    component.sayHello();
            }
    }

    @Override
    public void sayGoodbye() {
            for (Component component : components) {
                    component.sayGoodbye();
            }
    }

    public void add(Component component) {
            components.add(component);
    }

    public void remove(Component component) {
            components.remove(component);
    }

    public List<Component> getComponents() {
            return components;
    }

    public Component getComponent(int index) {
            return components.get(index);
    }
}
```

The Composite Design Pattern – Example



The Composite Design Pattern – Example

File: CompositeDemo.java

```
package com.cakes;

public class CompositeDemo {

        public static void main(String[] args) {

            Leaf leaf1 = new Leaf("Bob");
            Leaf leaf2 = new Leaf("Fred");
            Leaf leaf3 = new Leaf("Sue");
            Leaf leaf4 = new Leaf("Ellen");
            Leaf leaf5 = new Leaf("Joe");

            Composite composite1 = new Composite();
            composite1.add(leaf1);
            composite1.add(leaf2);

            Composite composite2 = new Composite();
            composite2.add(leaf3);
            composite2.add(leaf4);

            Composite composite3 = new Composite();
            composite3.add(composite1);
            composite3.add(composite2);
            composite3.add(leaf5);

            System.out.println("Calling 'sayHello' on leaf1");
            leaf1.sayHello();

            System.out.println("\nCalling 'sayHello' on composite1");
            composite1.sayHello();

            System.out.println("\nCalling 'sayHello' on composite2");
            composite2.sayHello();

            System.out.println("\nCalling 'sayGoodbye' on composite3");
            composite3.sayGoodbye();
        }
}
```

The Composite Design Pattern – Example

Output

```
Calling 'sayHello' on leaf1
Bob leaf says hello

Calling 'sayHello' on composite1
Bob leaf says hello
Fred leaf says hello

Calling 'sayHello' on composite2
Sue leaf says hello
Ellen leaf says hello

Calling 'sayGoodbye' on composite3
Bob leaf says goodbye
Fred leaf says goodbye
Sue leaf says goodbye
Ellen leaf says goodbye
Joe leaf says goodbye
```

The Composite Design Pattern

➢ **The steps required to apply the composite design pattern include:**

1. Identify, understand, and plan the tree-like structure required for the system.
2. Identify and design the **Component** base class:
   - ✓ Includes overridable methods common to both Leaf and Composite objects.
3. Identify and design the **Composite** class, which will:
   - ✓ Overrides common methods in the component interface.
   - ✓ Requires an internal data structure to store objects (nodes) added to the hierarchy.
   - ✓ Includes methods specific for **Composite** objects, which provide capability for adding and removing objects to the internal data structure (i.e. add objects to hierarchy).
4. Identify and design the **Leaf** classes, which overrides methods in the component class
5. Identify and design the **client** that uses both composite and leaf objects.

The Composite Design Pattern

- **Benefits of the Composite design pattern:**
  - ✓Provides a design structure that supports both composite and primitive objects.
  - ✓Minimizes complexity on clients by shielding them from knowing the operational differences between primitive and composite objects.
    - –Clients that expect a primitive object will also work with a composite object, since operations are called uniformly on both primitive and composite objects.
  - ✓Easy to create and add new components objects to applications.

https://www.digitalocean.com/community/tutorials/composite-design-pattern-in-java

Extra:

- **Explanation of the Composite Design Pattern**
- The **Composite design pattern** is used when we want to treat individual objects (primitives) and groups of objects (composites) in the same way.
- It provides a **tree-like structure**, where leaf nodes are simple objects and composite nodes are collections of objects.
- Clients (the user or program that uses the objects) don't need to know whether they're dealing with a single object or a group — they can call the same operations uniformly.
- This reduces complexity because the same interface is used for both simple and complex objects.
- It also makes it easier to **add new objects** (either simple or composite) without changing the client code.

Extra:

Real-Life Example

File System (Folders and Files):

- A **file** is a primitive object — it doesn't contain other objects.
- A **folder** is a composite object — it can contain both files and other folders.
- Whether you open a file or a folder, you're using the same operation (`open`). If it's a file, the content shows directly. If it's a folder, it expands to show its contents.

👉 The file explorer treats both **files** (primitives) and **folders** (composites) the same way.

Another Simple Example for Students

Think about a **university course structure**:

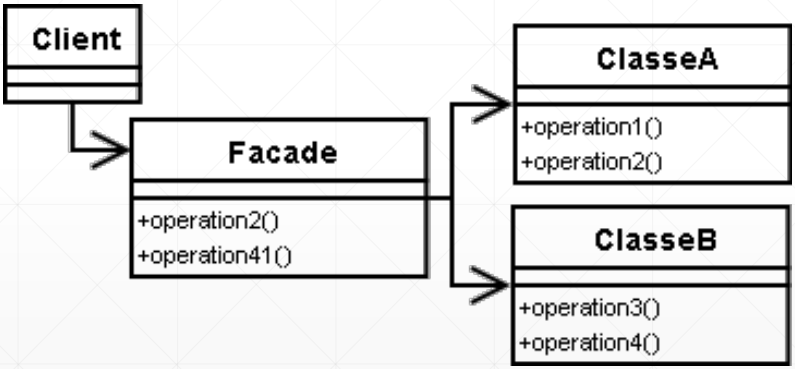
- A **single course** (like "Math 101") is a primitive.
- A **program** (like "Computer Science degree") is a composite — it contains many courses.
- Whether you check the details of one course or the whole program, you interact with them in a similar way.

Facade Design Pattern

- The **Facade** design pattern is an object structural pattern that provides a simplified interface to complex subsystems.

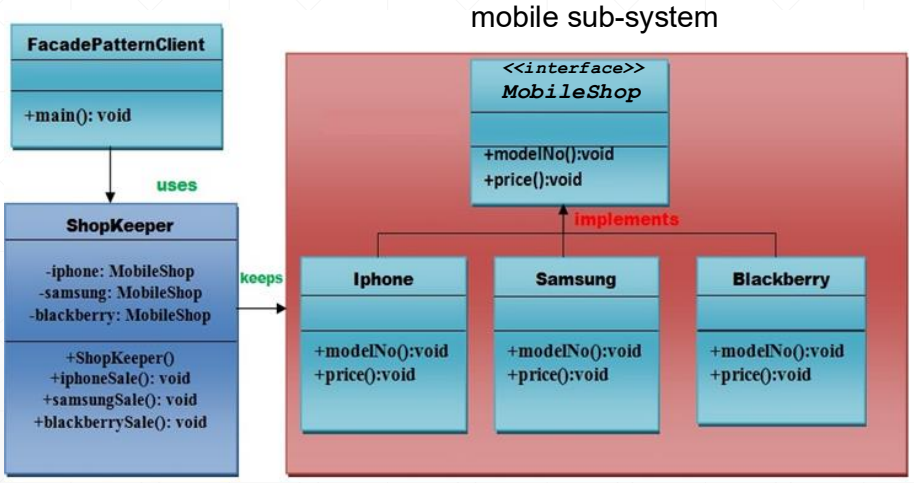
- The **intent** of the Facade is to:
  - ✓ Provide a unified interface to a set of interfaces in a subsystem.
  - ✓ Facade defines a higher-level interface that makes the subsystem easier to use.

Facade Design Pattern - Structure



Facade Design Pattern – Example*

- Consider the following UML example. The client class requires to use the mobile sub-system without the need to deal with details of the sub-systems.
- This can be achieved by using a Facade Pattern class (ShopKeeper) to allow to client class to use the mobile sub-system easily.



*source: https://www.javatpoint.com/facade-pattern

Facade Design Pattern – Example (implementation)

- The following Java code represent the implementation of the classes in the mobile sub-system:

File: MobileShop.java

```
public interface MobileShop {
    public void modelNo();
    public void price();
}
```

File: Iphone.java

```
public class Iphone implements MobileShop {
    @Override
    public void modelNo() {
        System.out.println(" Iphone 6 ");
    }
    @Override
    public void price() {
    System.out.println(" Rs 65000.00 ");
    }
}
```

Facade Design Pattern – Example (implementation)

File: Samsung.java

```
public class Samsung implements MobileShop {
    @Override
    public void modelNo() {
    System.out.println(" Samsung galaxy tab 3 ");
    }
    @Override
    public void price() {
        System.out.println(" Rs 45000.00 ");
    }
}
```

File: Blackberry.java

```
public class Blackberry implements MobileShop {
    @Override
    public void modelNo() {
    System.out.println(" Blackberry Z10 ");
    }
    @Override
    public void price() {
        System.out.println(" Rs 55000.00 ");
    }
}
```

Facade Design Pattern – Example (implementation)

File: ShopKeeper.java

```
public class ShopKeeper {
    private MobileShop iphone;
    private MobileShop samsung;
    private MobileShop blackberry;

    public ShopKeeper(){
        iphone= new Iphone();
        samsung=new Samsung();
        blackberry=new Blackberry();
    }
    public void iphoneSale(){
        iphone.modelNo();
        iphone.price();
    }
    public void samsungSale(){
        samsung.modelNo();
        samsung.price();
    }

    public void blackberrySale(){
        blackberry.modelNo();
        blackberry.price();
    }
}
```

Facade Design Pattern – Example (implementation)

Output

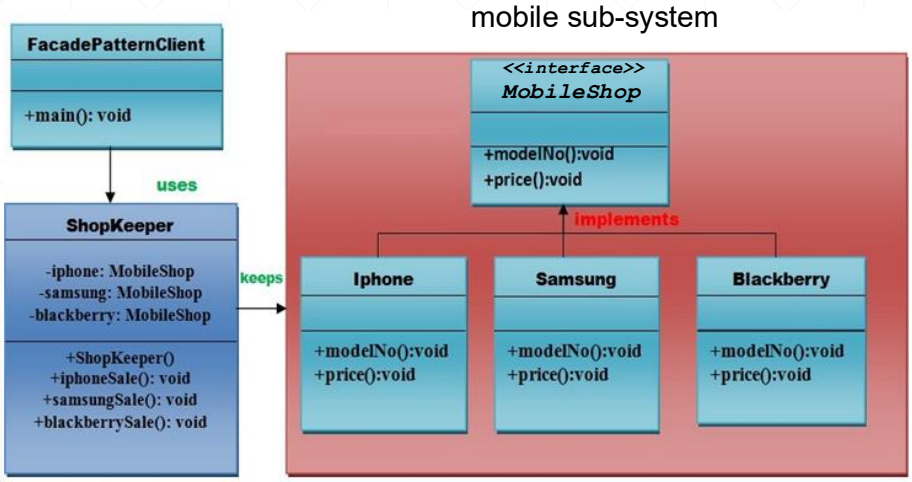
```
========== Mobile Shop ============
        1. IPHONE.
        2. SAMSUNG.
        3. BLACKBERRY.
        4. Exit.
Enter your choice: 1
 Iphone 6
 Rs 65000.00
```

File: FacadePatternClient.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class FacadePatternClient {
   private static int  choice;
   public static void main(String args[]) throws NumberFormatException, IOException{
      do{
         System.out.print("========== Mobile Shop ============ \n");
         System.out.print("          1. IPHONE.            \n");
         System.out.print("          2. SAMSUNG.             \n");
         System.out.print("          3. BLACKBERRY.          \n");
         System.out.print("          4. Exit.              \n");
         System.out.print("Enter your choice: ");

         BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
         choice=Integer.parseInt(br.readLine());
         ShopKeeper sk=new ShopKeeper();

         switch (choice) {
             case 1:
                 sk.iphoneSale();
                 break;
             case 2:
                 sk.samsungSale();
                 break;
             case 3:
                 sk.blackberrySale();
                 break;
             default:
                 System.out.println("Nothing You purchased");
             return;
         }
      }while(choice!=4);
}
```

Facade Design Pattern – Example*

- As noted from the example, the Façade Pattern provided us with a **simplified interface** to a complex subsystems.
- The Facade Pattern class (ShopKeeper) to allow client class to use the mobile sub-system without knowing anything about the sub-system.



*source: https://www.javatpoint.com/facade-pattern

Facade Design Pattern

- **The step-by-step procedure for applying the Facade design pattern include:**

1. Identify all components involved in carrying out a subsystem operation
2. Create a list of the operations required to execute the subsystem operation.
3. Design a Facade class that includes an interface method to carry out the subsystem operation.
4. Implement the Facade interface method by calling operations on one or more subsystem components, in the list identified in step 2.
5. Allow one ore more clients to access the objects of the Facade type so that they can gain access to the subsystem operation.

Facade Design Pattern

- **Benefits of the Facade design pattern include**
  - ✓ Shields clients from knowing the internals of complex subsystem, therefore minimizing complexity in clients.
  - ✓ The facade provides a stable interface that hides changes to internal subsystems; therefore, client code is more stable.
  - ✓ Promotes weak coupling on clients; with facade, clients depend only on one interface instead of multiple interfaces

Summary

- In this session, we presented **structural design patterns**, including:
  - ✓Adapter
  - ✓Facade
  - ✓Composite

- Next … we will present **behavioral** design patterns in detailed design.