Behavioral Design Patterns

- A behavioral design pattern is concerned with communication between objects and how responsibilities are assigned between objects.
- **Chain of Responsibility**: A method for commands to be delegated to a chain of processing objects.
- **Command**: Encapsulates a command request in an object.
- **Interpreter**: Supports the use of language elements within an application.
- **Iterator**: Supports iterative (sequential) access to collection elements.
- **Mediator**: Articulates simple communication between classes.
- **Memento**: A process to save and restore the internal/original state of an object.
- **Observer**: Defines how to notify objects of changes to other object(s).
- **State**: How to alter the behavior of an object when its stage changes.
- **Strategy**: Encapsulates an algorithm inside a class.
- **Visitor**: Defines a new operation on a class without making changes to the class.
- **Template Method**: Defines the skeleton of an operation while allowing subclasses to refine certain steps.

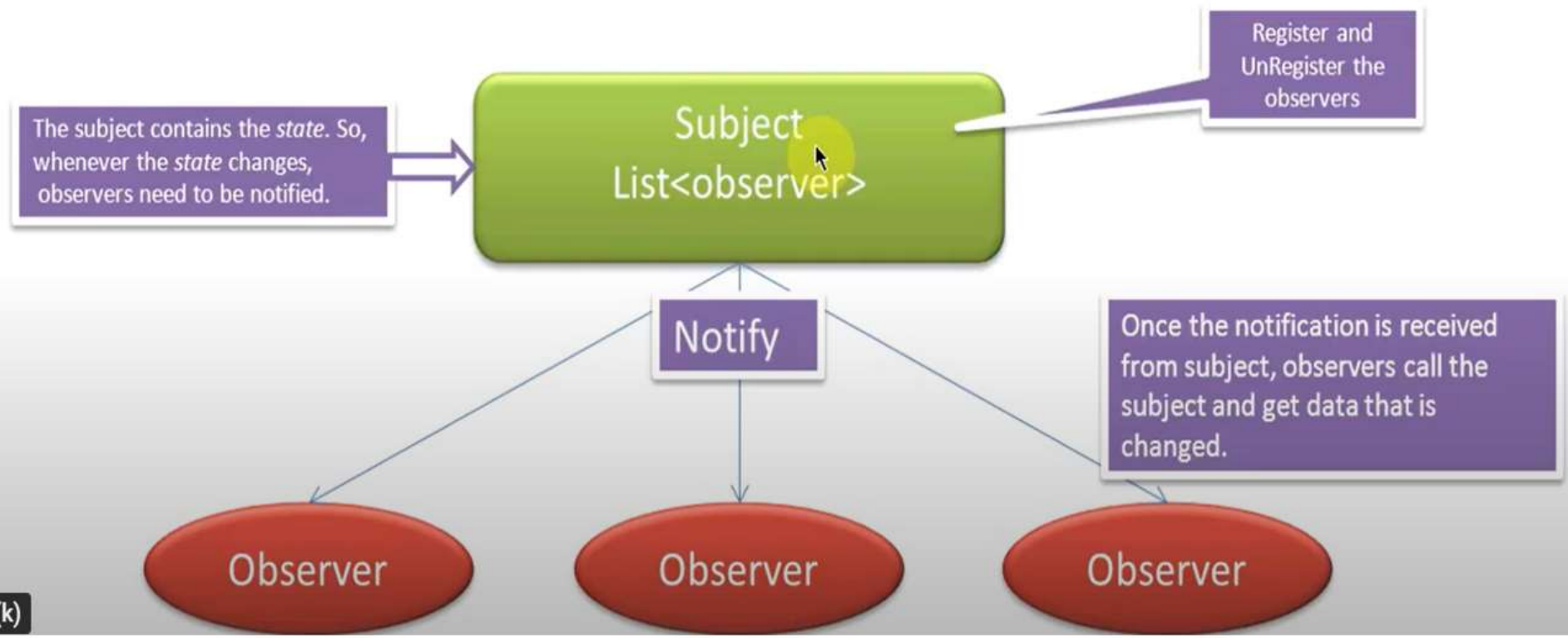
- Definition: The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Purpose: Promotes loose coupling between the subject and observers.

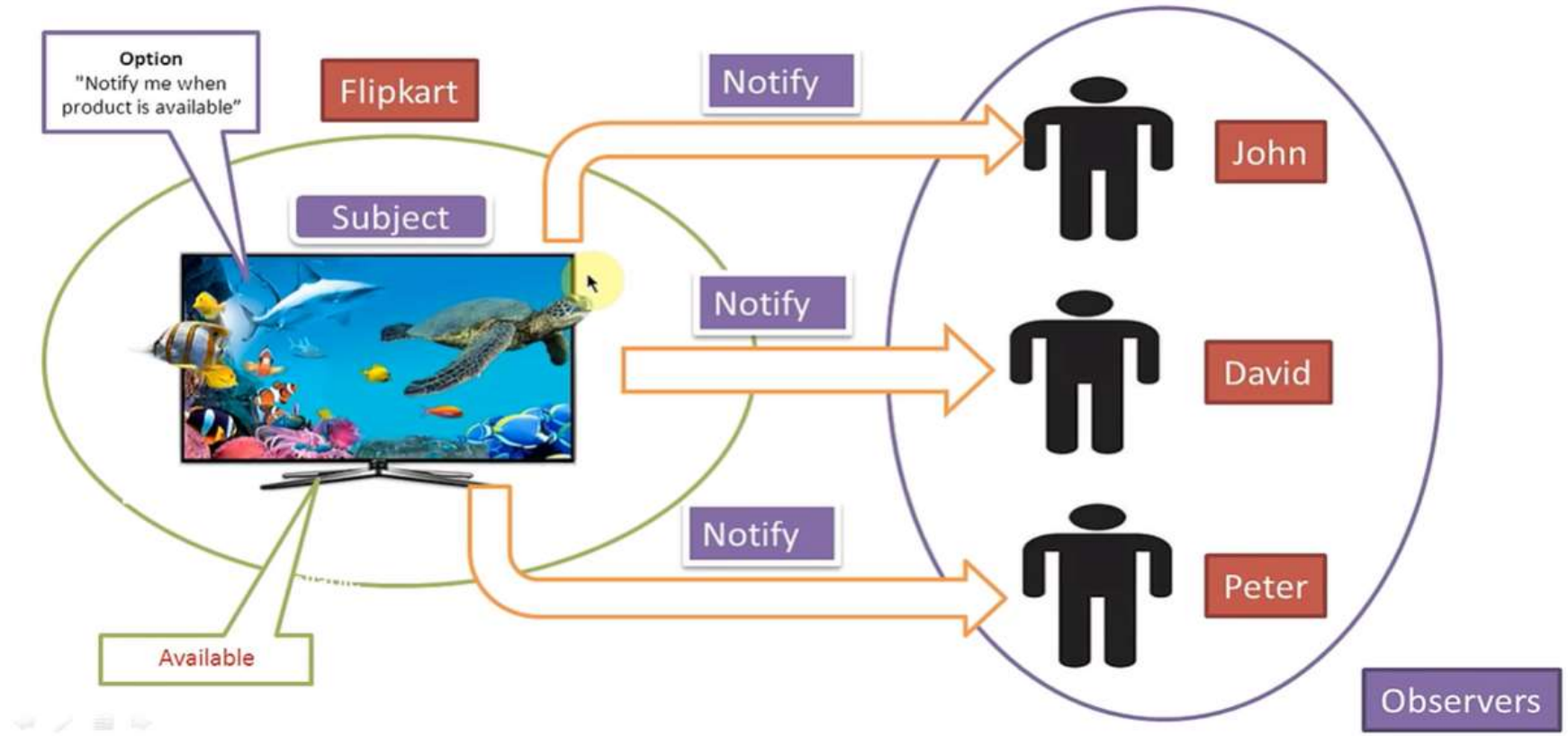
- Real-World Analogy: A news agency (subject) notifies subscribers (observers) when news is published.

- The **Observer Design Pattern** is a **behavioral pattern** that defines a one-to-many dependency between objects so that when one object (the **subject**) changes state, all its dependents (the **observers**) are notified and updated automatically.
- The observer pattern is also known as Dependents or Publish-Subscribe.

- ✓ The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods
- ✓ The *observer* pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically
- ✓ Another name of observer is listener



Play (k)


Option
"Notify me when product is available"
Flipkart
Subject
Notify
Notify
Notify
John
David
Peter
Available
Observers

Key Concepts

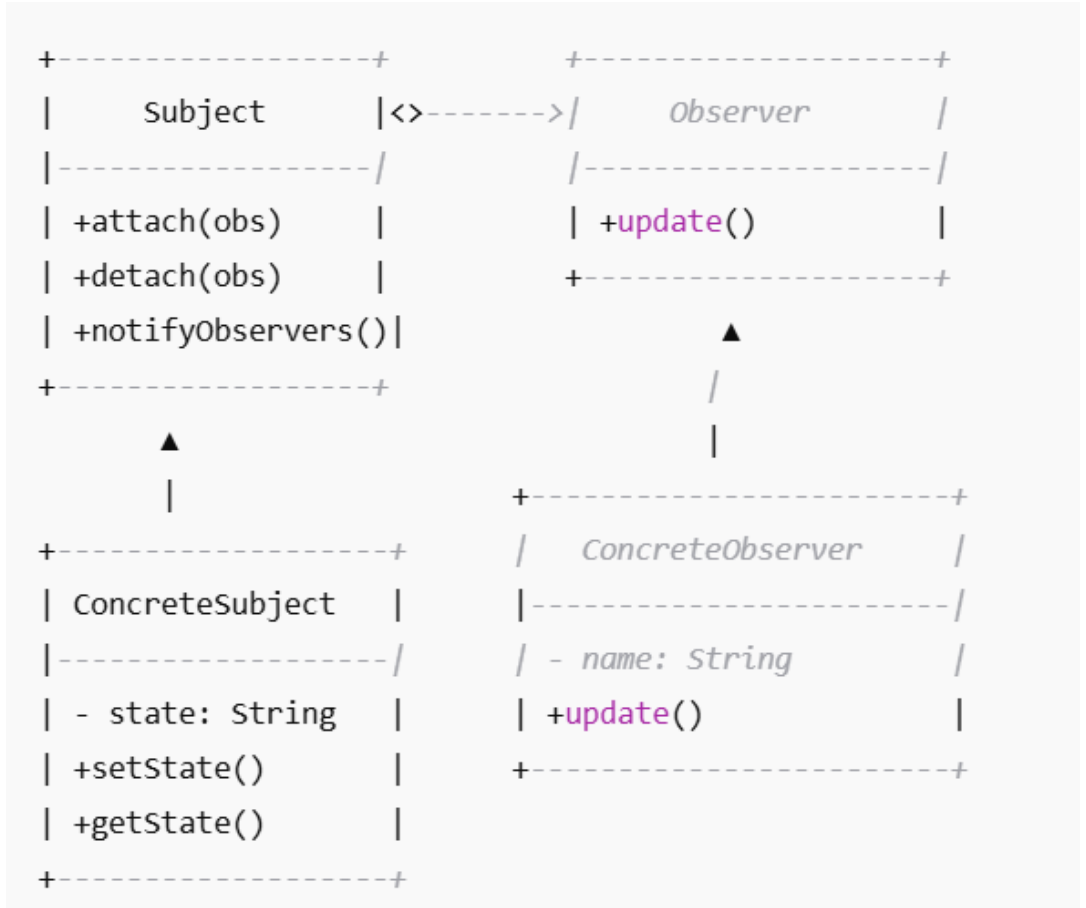
- **Subject**: Maintains a list of observers and notifies them of changes.
- **Observer**: Defines an interface for receiving updates.
- **ConcreteSubject**: Stores the state of interest and notifies observers on state changes.
- **ConcreteObserver**: Implements the Observer interface and responds to updates.

Scenario Example: Weather Station

- A **WeatherStation** (subject) broadcasts temperature updates. Several **displays** (observers) like a **mobile app**, **LED panel**, or **website widget** subscribe to receive updates.

When to Use

- When changes in one object require updates in others.
- When an object should notify other objects without knowing who they are.


Subject
+attach(obs)
+detach(obs)
+notifyObservers()
Observer
+update()
ConcreteSubject
- state: String
+setState()
+getState()
ConcreteObserver
- name: String
+update()

Java Implementation – Interfaces

```
// Observer Interface
interface Observer {
    void update(String message);
}

// Subject Interface
interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notifyObservers();
}
```

Java Implementation – Concrete Classes

```
// Concrete Subject
class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String news;
    public void attach(Observer o) {
        observers.add(o);
    }
    public void detach(Observer o) {
        observers.remove(o);
    }
    public void setNews(String news) {
        this.news = news;
        notifyObservers();
    }
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(news);
        }
    }
}
```

```
// Concrete Observer
class Subscriber implements Observer {
    private String name;

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

    public void update(String news) {
        System.out.println(name + " received update: " + news);
    }
}
```

Java Implementation – Demo

```
public class ObserverPatternDemo {
    public static void main(String[] args) {
        NewsAgency agency = new NewsAgency();

        Subscriber s1 = new Subscriber("Alice");
        Subscriber s2 = new Subscriber("Bob");

        agency.attach(s1);
        agency.attach(s2);

        agency.setNews("Observer Pattern Launched!");
        agency.setNews("More Updates Available!");
    }
}
```

Output:

- Alice received update: Observer Pattern Launched!
- Bob received update: Observer Pattern Launched!
- Alice received update: More Updates Available!
- Bob received update: More Updates Available!

Real-World Applications

- GUI Frameworks: Event listeners in Java Swing.
- Messaging Systems: Publish-subscribe models.
- Distributed Systems: Real-time data feeds.

Benefits

- Loose Coupling: Subjects and observers can vary independently.
- Dynamic Relationships: Observers can be added or removed at runtime.
- Broadcast Communication: Efficiently notify multiple observers.

Scenario:

- Task: Implement a weather monitoring system where multiple displays (observers) receive updates from a weather station (subject) when temperature changes.
- Steps:
- Define Observer and Subject interfaces.
- Implement WeatherStation as the concrete subject.
- Implement CurrentConditionsDisplay and StatisticsDisplay as concrete observers.
- Simulate temperature changes and observe updates.

Weather Monitoring System

Context:

A weather station gathers data such as temperature, humidity, and pressure. Different display elements (like current conditions, statistics, and forecast) need to update automatically when the weather changes.

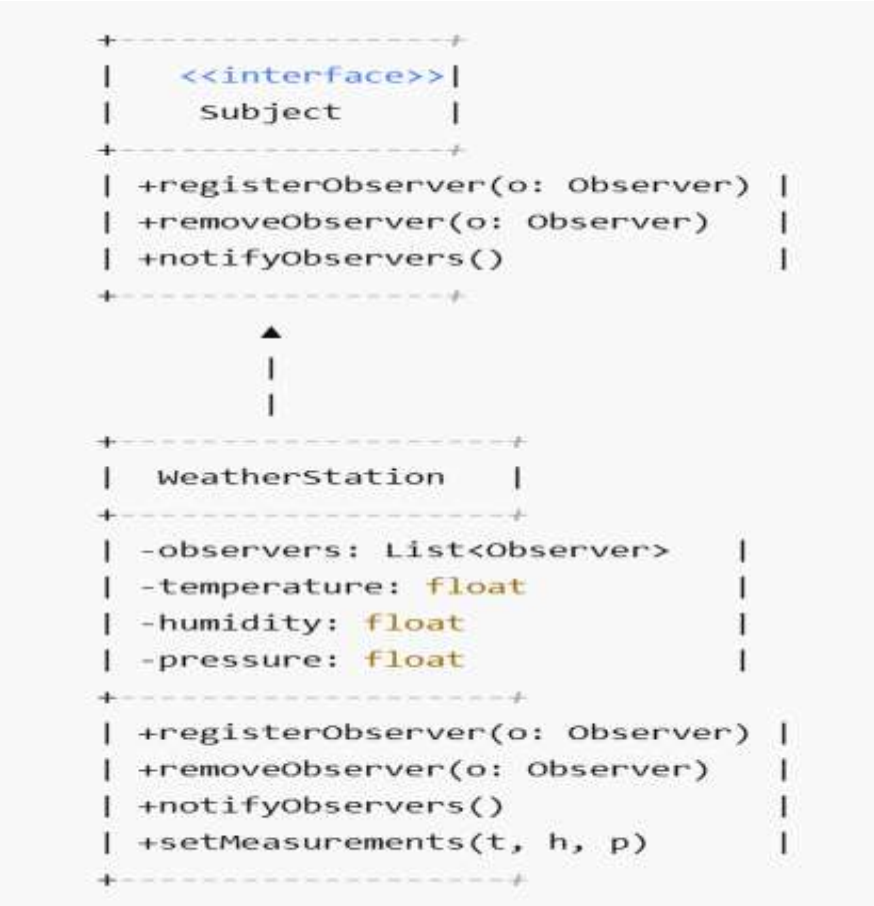
Problem:

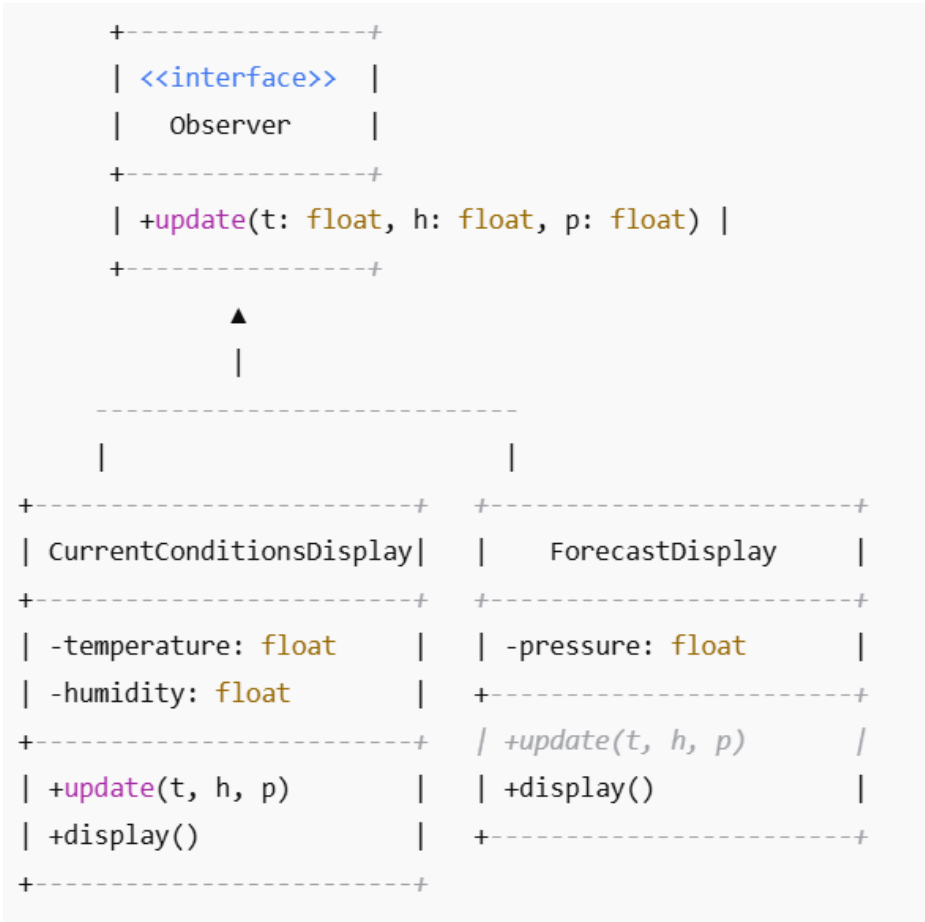
How can the system ensure that all display elements reflect the latest weather data **without tightly coupling them** to the WeatherStation?

Solution using Observer Pattern:

- Use the **Observer Design Pattern**. The WeatherStation is the **Subject**, and all display elements are **Observers**. When weather data changes, the WeatherStation **notifies** all observers automatically.

- Flow:
- CurrentConditionsDisplay and ForecastDisplay register themselves with WeatherStation.
- WeatherStation gets new sensor data.
- It calls notifyObservers().
- Each registered display element’s update() method is called automatically.


<<interface>>
Subject
+registerObserver(o: Observer)
+removeObserver(o: Observer)
+notifyObservers()
WeatherStation
-observers: List<Observer>
-temperature: float
-humidity: float
-pressure: float
+registerObserver(o: Observer)
+removeObserver(o: Observer)
+notifyObservers()
+setMeasurements(t, h, p)


<<interface>>
Observer
+update(t: float, h: float, p: float)
CurrentConditionsDisplay
-temperature: float
-humidity: float
+update(t, h, p)
+display()
ForecastDisplay
-pressure: float
+update(t, h, p)
+display()

- . **Subject (WeatherData)**
- This is the core class that holds the data and notifies observers of any changes.
- Methods:
- registerObserver(Observer o)
- Adds an observer to the list.
- removeObserver(Observer o)
- Removes an observer from the list.
- notifyObservers()
- Updates all registered observers when the state changes.
- measurementsChanged()
- Called whenever new weather data is available. It triggers notifyObservers().
- getTemperature(), getHumidity(), getPressure()
- Accessor methods to get current weather data.

- 2. Observer (Interface)
- Defines the contract for all observers who want to be notified of changes in the Subject.

- Method:
- update(float temp, float humidity, float pressure)
- Called by the subject when weather data changes.

- DisplayElement (Interface)
- Used to display the data on screen or console.

- Method:
- display()
- Called to show the current state of the observer.

- 4. CurrentConditionsDisplay (Concrete Observer)
- This is a concrete observer that implements Observer and DisplayElement.

- Fields:
- temperature, humidity
- Stores the data it receives from the subject.
- subject
- Keeps a reference to the WeatherData object to register/unregister itself.

- Methods:
- update(temp, humidity, pressure)
- Stores new values and calls display().
- display()
- Prints or shows the current conditions.

- Flow Summary:
- WeatherData (Subject) stores weather information.
- CurrentConditionsDisplay (Observer) registers with WeatherData.
- When measurementsChanged() is called, WeatherData invokes notifyObservers().
- All registered observers get updated via the update() method.
- Observers then display the updated info using display().

```
//Observer Interface
public interface Observer {
    void update(float temperature, float humidity, float pressure);
}
```

```
//Subject Interface
public interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}
```

```
//WeatherStation (Concrete Subject)
import java.util.*;
public class WeatherStation implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherStation() {
        observers = new ArrayList<>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }
```

```
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }
public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        notifyObservers();
    }
}
```

```
//CurrentConditionsDisplay (Concrete Observer)
public class CurrentConditionsDisplay implements Observer {
    private float temperature;
    private float humidity;

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature + "°C and " + humidity + "% humidity");
    }
}
```

```
//ForecastDisplay (Concrete Observer)
public class ForecastDisplay implements Observer {
    private float pressure;

    public void update(float temperature, float humidity, float pressure) {
        this.pressure = pressure;
        display();
    }

    public void display() {
        System.out.println("Forecast: Pressure is " + pressure + " hPa");
    }
}
```

```
//Main
public class WeatherApp {
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();

        CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay();
        ForecastDisplay forecastDisplay = new ForecastDisplay();

        weatherStation.registerObserver(currentDisplay);
        weatherStation.registerObserver(forecastDisplay);

        // Simulating new weather data
        weatherStation.setMeasurements(25.5f, 65.0f, 1013.0f);
        weatherStation.setMeasurements(27.0f, 70.0f, 1012.5f);
    }
}
```

Summary

- The Observer Pattern is essential for implementing distributed event-handling systems.
- It promotes a clean separation between the subject and its observers.
- Ideal for scenarios requiring dynamic relationships between objects

Template Design Pattern

- The **Template Method Pattern** defines the **skeleton of an algorithm** in a method, deferring some steps to subclasses.
- It allows subclasses to redefine certain steps of the algorithm **without changing its structure**

- ✓ Template Method Pattern defines a sequence of steps of an algorithm. The subclasses are allowed to override the steps but not allowed to change the sequence.
- ✓ The key to the Template Method pattern is that we put the general logic in the abstract parent class, and let the child class define the specifics.

Foundation

Pillars

Walls

Windows

Template Method [Build a House]

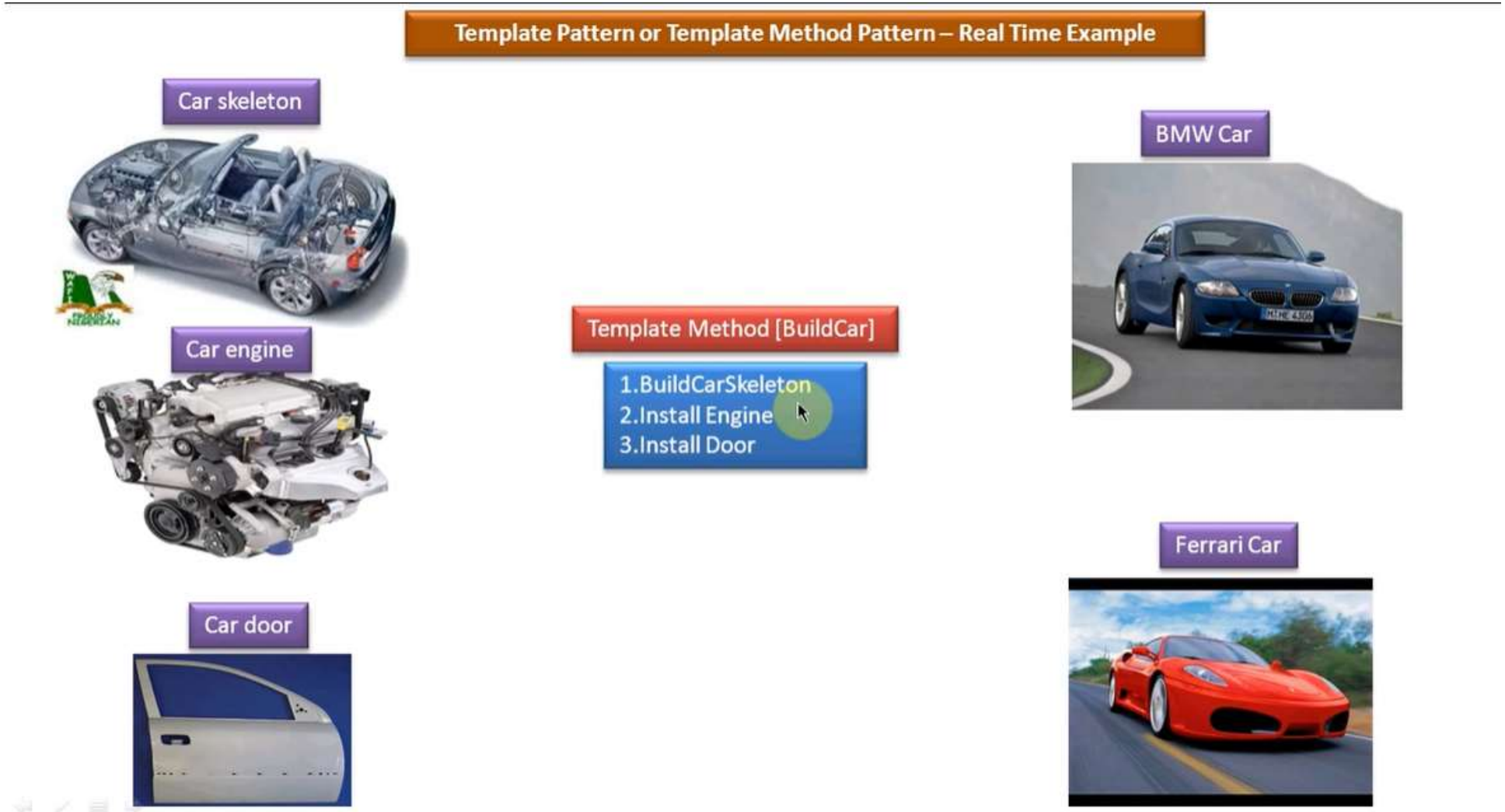
1) Building foundation
2) Building pillars
3) Building walls
4) Building windows

Concrete House

Wooden House

Template Method Design Pattern: Real-Time Example with Coffee Making

Template Method Design Pattern: Real Time Example of Building a Car



- Benefits
- Promotes code reuse by factoring out common code in a base class
- Inversion of control: Subclasses decide the specifics without altering the algorithm
- Ensures consistency in how subclasses implement certain steps

Component	Description
`AbstractClass`	Defines the template method and concrete methods
`TemplateMethod()`	Outlines the algorithm steps, some implemented in base, some abstract
`PrimitiveOperations()`	Abstract methods to be implemented by subclasses
`ConcreteClass`	Implements the specific steps of the algorithm

Template Design Pattern or Template Method Design Pattern Implementation: Building a House

Template Pattern or Template Method Pattern – Implementation

Foundation

Pillars

Walls

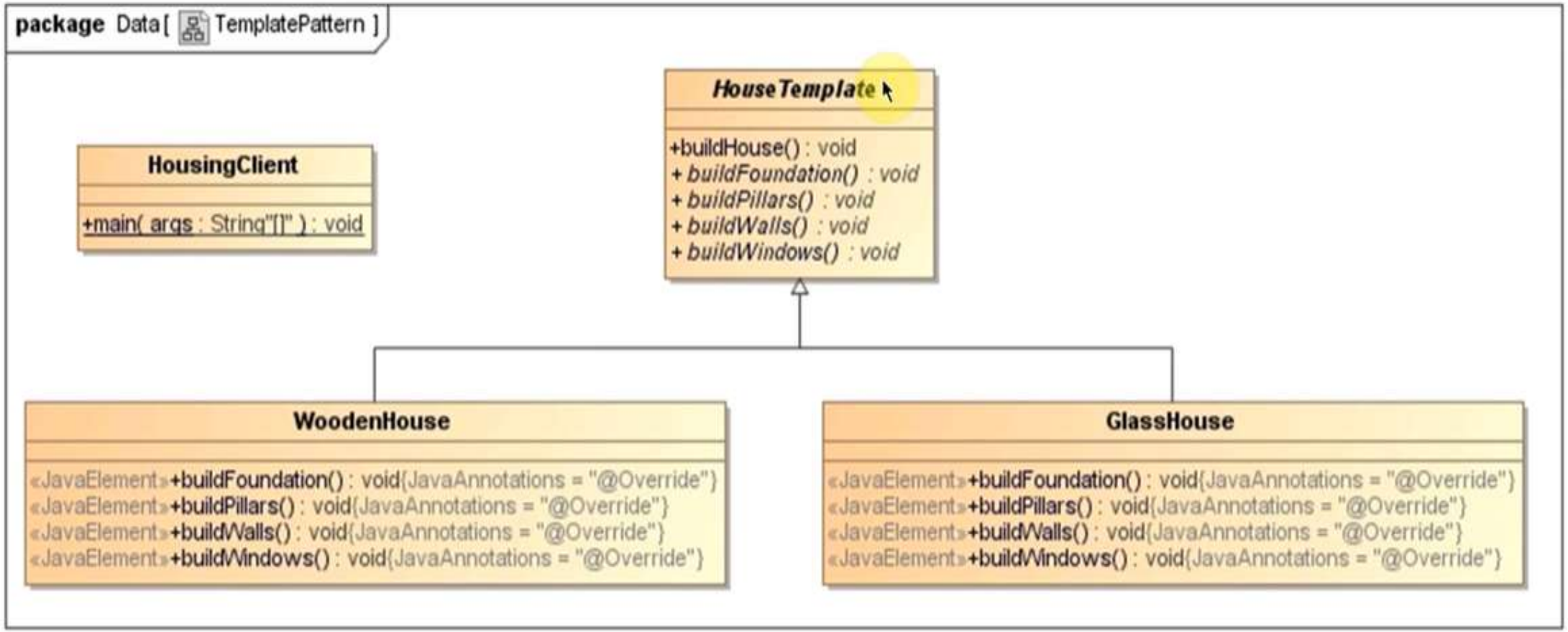
Windows

Template Method [Build a House]

1) Building foundation
2) Building pillars
3) Building walls
4) Building windows

Glass House

Wooden House


package Data [TemplatePattern]
HouseTemplate
+buildHouse() : void
+ buildFoundation() : void
+ buildPillars() : void
+ buildWalls() : void
+ buildWindows() : void
HousingClient
+main(args : String"[]") : void
WoodenHouse
«JavaElement»+buildFoundation() : void{JavaAnnotations = "@Override"}
«JavaElement»+buildPillars() : void{JavaAnnotations = "@Override"}
«JavaElement»+buildWalls() : void{JavaAnnotations = "@Override"}
«JavaElement»+buildWindows() : void{JavaAnnotations = "@Override"}
GlassHouse
«JavaElement»+buildFoundation() : void{JavaAnnotations = "@Override"}
«JavaElement»+buildPillars() : void{JavaAnnotations = "@Override"}
«JavaElement»+buildWalls() : void{JavaAnnotations = "@Override"}
«JavaElement»+buildWindows() : void{JavaAnnotations = "@Override"}

```java
public abstract class HouseTemplate
{

        // Template method is final so subclasses can't override
        public final void buildHouse()
        {
                buildFoundation();
                buildPillars();
                buildWalls();
                buildWindows();
                System.out.println("House is built.");
        }

        // Methods to be implemented by subclasses

        public abstract void buildFoundation();

        public abstract void buildPillars();

        public abstract void buildWalls();

        public abstract void buildWindows();

}
```

```java
public class GlassHouse extends HouseTemplate
{

	@Override
	public void buildFoundation()
	{
		System.out.println("Building foundation with cement,iron rods and sand");
	}

	@Override
	public void buildPillars()
	{
		System.out.println("Building Pillars with glass coating");
	}

	@Override
	public void buildWalls()
	{
		System.out.println("Building Glass Walls");
	}

	@Override
	public void buildWindows()
	{
		System.out.println("Building Glass Windows");

	}

}
```

```java
public class WoodenHouse extends HouseTemplate
{

	@Override
	public void buildFoundation()
	{
		System.out.println("Building foundation with cement,iron rods,sand and wood");
	}

	@Override
	public void buildPillars()
	{
		System.out.println("Building Pillars with Wood coating");
	}

	@Override
	public void buildWalls()
	{
		System.out.println("Building Wooden Walls");
	}

	@Override
	public void buildWindows()
	{
		System.out.println("Building Wooden Windows");

	}

}
```

```
1  public class HousingClient
2  {
3
4          public static void main( String[] args )
5          {
6
7                  System.out.println("Build a WoodenHouse\n");
8                  HouseTemplate houseType = new WoodenHouse();
9
0                  // using template method
1                  houseType.buildHouse();
2                  System.out.println("**************************************************************");
3
4                  System.out.println("Build a GlassHouse\n");
5                  houseType = new GlassHouse();
6
7                  houseType.buildHouse();
8          }
9
0  }
```

- Scenario: Document Generation System
- Goal
- You want to build a system that generates reports in different formats — e.g., PDF, HTML. The structure of the document is the same:
- Header
- Content
- Footer
- The format of these parts varies based on the type of report.

Abstract Class: DocumentGenerator

```
abstract class DocumentGenerator {
    // Template Method
    public final void generateDocument() {
        writeHeader();
        writeContent();
        writeFooter();
    }

    abstract void writeHeader();
    abstract void writeContent();
    abstract void writeFooter();
}
```

Concrete Class: PDFDocument

```
class PDFDocument extends DocumentGenerator {
    void writeHeader() {
        System.out.println("PDF Header: Company Logo");
    }

    void writeContent() {
        System.out.println("PDF Content: Financial Summary in tables");
    }

    void writeFooter() {
        System.out.println("PDF Footer: Confidential - Page 1");
    }
}
```

Concrete Class: HTMLDocument

```
class HTMLDocument extends DocumentGenerator {
    void writeHeader() {
        System.out.println("<h1>HTML Report Title</h1>");
    }

    void writeContent() {
        System.out.println("<p>HTML content with graphs and tables</p>");
    }

    void writeFooter() {
        System.out.println("<footer>Generated by System</footer>");
    }
}
```

Demo Program

```
public class TemplateMethodExample {
    public static void main(String[] args) {
        DocumentGenerator pdf = new PDFDocument();
        DocumentGenerator html = new HTMLDocument();

        System.out.println("Generating PDF Document:");
        pdf.generateDocument();

        System.out.println("\nGenerating HTML Document:");
        html.generateDocument();
    }
}
```

Output

- Generating PDF Document:
- PDF Header: Company Logo
- PDF Content: Financial Summary in tables
- PDF Footer: Confidential - Page 1

- Generating HTML Document:
- <h1>HTML Report Title</h1>
- HTML content with graphs and tables
- <footer>Generated by System</footer>