

OVERVIEW Behavioral Design Patterns: Iterator and Observer

Behavioural patterns distribute behaviour among objects efficiently and safely. Iterator gives clients a standard way (`hasNext()`, `next()`) to walk through a collection without knowing whether it is an array, a list or a tree: the collection hands out an iterator that keeps its own position. Observer sets up a one-to-many dependency: observers attach to a subject, and when the subject's state changes it calls `update()` on every registered observer, so they stay in sync without the subject knowing their concrete classes.

Iterator roles

Iterator interface (`hasNext`, `next`),
ConcreteIterator (keeps the index),
Aggregate/Container (`getIterator`),
ConcreteAggregate.

Iterator benefits

Consistent traversal; collection internals can change without changing clients; several traversals (forward, reverse) over the same data.

Observer roles

Subject (attach, detach, notify, holds state and an observer list), Observer (`update`),
ConcreteObservers.

Observer flow

`setState()` → `notifyAllObservers()` → each
`observer.update()` → observer reads
`subject.getState()`.

HOW THE LECTURE IS ORGANIZED

Slides 1–3 Behavioural patterns

Slides 4–16 Iterator: intent, structure, NameRepository example, steps, benefits, exercise

Slides 17–30 Observer: intent, structure, number-converter example, steps, benefits

Slides 31–32 Social-media exercise and summary

EXAM FOCUS

Final 252 Q7-2 (2.5 marks) asked you to recognise Observer in a weather-station scenario and draw its class diagram. Iterator has not been examined yet; expect it as a “which pattern?” question.

HOW TO READ THIS PAGE

Every original slide is shown unchanged. **Yellow marks** on a slide point to a numbered note: hover or tap a mark to jump to its note, and hover a note to see its marks. Questions tagged **Past exam** appeared on a real paper; **Prediction** questions are likely angles, not confirmed.

Software Design and Architecture

[Behavioral Design Patterns] – Chapter 07, L02

NOTES

- 1 The last design-pattern lecture: **behavioural** patterns, which are about how objects interact and share work. Two are covered: **Iterator** and **Observer**.

SLIDE 2 / 32 Lecture Outlines

Lecture Outlines

➤ **Previously- Structural Design Patterns**

- ✓ Adapter
- ✓ Composite
- ✓ Facade

➤ **Behavioral Design Patterns**

- ✓ **Iterator**
- ✓ **Observer**

NOTES

- 1 Previously: the structural patterns Adapter, Composite and Facade.
- 2 Today: **Iterator** and **Observer**.

SLIDE 3 / 32 Behavioral Design Patterns

Behavioral Design Patterns

- Behavioral patterns are responsible for the **efficient** and **safe distribution** of behaviors among the program's objects.
- Popular behavioral design patterns include:
 - ✓ **Iterator**
 - ✓ **Observer**
 - ✓ **Others not covered:**
 - Command
 - Chain of responsibility
 - Interpreter
 - Mediator
 - Memento
 - State
 - Strategy
 - Template Method
 - Visitor

| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

3

NOTES

- 1 Behavioural patterns are responsible for the **efficient and safe distribution of behaviours** among a program's objects: who does what, and how they communicate.
- 2 This course covers **Iterator** and **Observer**.
- 3 Others exist but are not covered: Command, Chain of Responsibility, Interpreter, Mediator, Memento, State, Strategy, Template Method, Visitor.

1- Iterator Design Pattern

- The Iterator design pattern is an **object behavioral** pattern that provides a standardized way for **accessing** and **traversing** objects in a **collection** data structure.
- The **intent** of the Iterators to:
 - ✓ Provide a way to access the elements of a collection object sequentially without any need to know its **underlying representation**.

NOTES

- 1** **Iterator** is an **object behavioural** pattern that gives a standard way of **accessing and traversing** the objects in a collection.
- 2** Its **intent**: provide a way to **access the elements of a collection sequentially without knowing its underlying representation**. The client asks "is there another one?" and "give me the next one", and never sees whether the data is an array, a linked list or a tree.

EXAM ANGLE

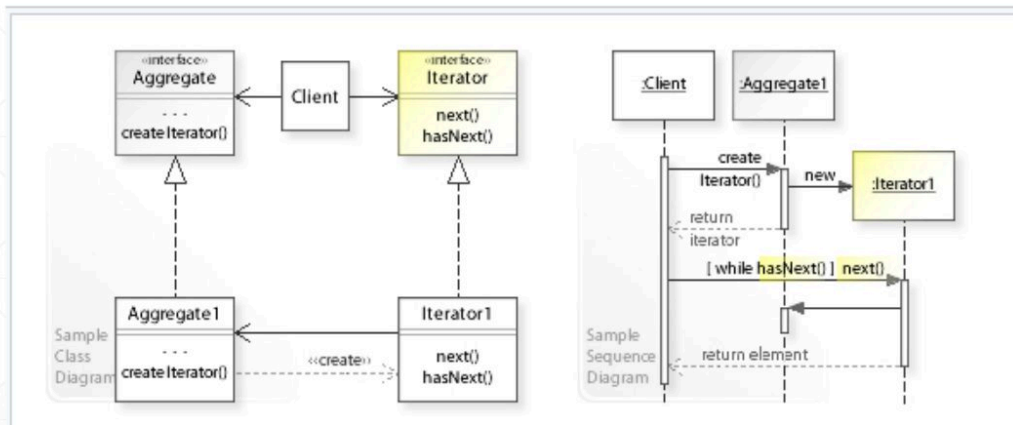
Prediction

Q1

Traversing a collection without exposing whether it is stored as an array, a list or a tree is the:

- A. Iterator pattern
- B. Observer pattern
- C. Composite pattern
- D. Facade pattern

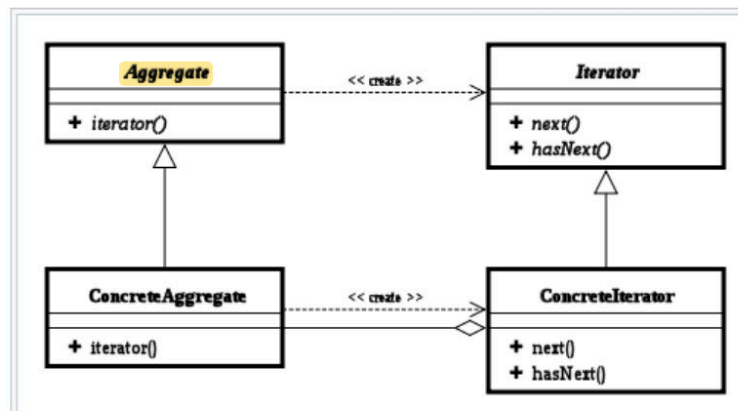
Answer: A. Iterator pattern



In the above UML class diagram, the **Client** class refers (1) to the **Aggregate** interface for creating an **Iterator** object (`createIterator()`) and (2) to the **Iterator** interface for traversing an **Aggregate** object (`next()`, `hasNext()`). The **Iterator1** class implements the **Iterator** interface by accessing the **Aggregate1** class.

NOTES

- 1 In the class diagram (left), the **Client** refers (1) to the **Aggregate** interface to create an iterator with `createIterator()` ...
- 2 ...and (2) to the **Iterator** interface to traverse the aggregate with `next()` and `hasNext()` .
- 3 **Iterator1** implements **Iterator** by accessing **Aggregate1**, which creates it. The sequence diagram (right) shows the run: the client calls `createIterator()` , **Aggregate1** creates an **Iterator1** and returns it, then the client loops `[while hasNext()] next()` and gets one element each time.

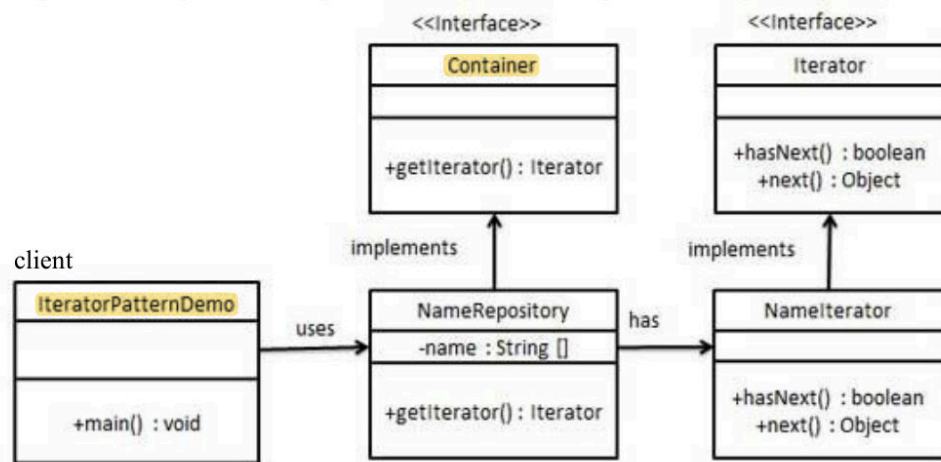


See example on <https://www.geeksforgeeks.org/iterator-pattern/>

NOTES

- 1 The generic structure: an **Aggregate** with *iterator()* and an **Iterator** with *next()* and *hasNext()* (italic = abstract), linked by «create».
- 2 **ConcreteAggregate** implements `iterator()` to create a **ConcreteIterator**, which implements `next()` and `hasNext()` and keeps a reference to its aggregate (the diamond) so it can read the elements.

Iterator Design Pattern – Structure



NOTES

- 1 The example's structure: «Interface» **Container** (`getIterator(): Iterator`) plays the aggregate; «Interface» **Iterator** has `hasNext(): boolean` and `next(): Object`.
- 2 **NameRepository** implements Container and holds `-name : String[]`; it "has" a **NameIterator**, which implements Iterator.
- 3 The client **IteratorPatternDemo** uses NameRepository through the interfaces.

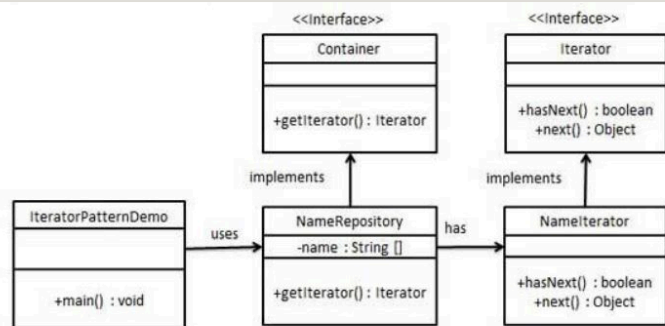
Iterator Design Pattern – Example*

Iterator.java

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
}
```

Container.java

```
public interface Container {
    public Iterator getIterator();
}
```



NOTES

1 The two interfaces:

CODE

```
public interface Iterator { public
boolean hasNext(); public Object
next(); }
```

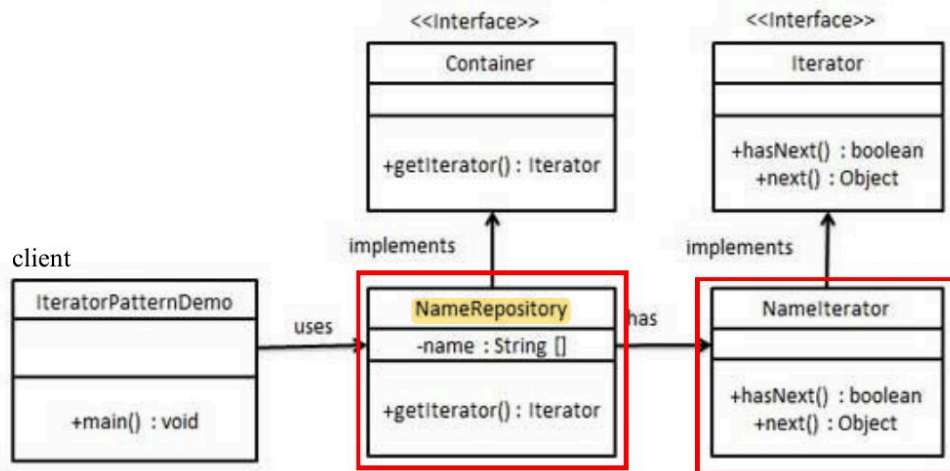
```
public interface Container { public
Iterator getIterator(); }
```

WHAT IT DOES

The traversal contract: is there another element, and give me it.

The aggregate contract: hand out an iterator.

Iterator Design Pattern – Structure



NOTES

- 1 The diagram with the two concrete classes outlined: **NameRepository** (the concrete collection) and **NameIterator** (the concrete iterator). They are written next.

SLIDE 10 / 32 Iterator Example: NameRepository.java

Iterator Design Pattern – Example

- Create concrete class implementing the **Container** interface.
- This class has inner class **NameIterator** implementing the **Iterator** interface.

NameRepository.java

```

public class NameRepository implements Container {
    public String names[] = {"Robert" , "John" , "Julie" , "Lora"};

    @Override
    public Iterator getIterator() {
        return new NameIterator();
    }

    private class NameIterator implements Iterator {

        int index;

        @Override
        public boolean hasNext() {

            if(index < names.length){
                return true;
            }
            return false;
        }

        @Override
        public Object next() {

            if(this.hasNext()){
                return names[index++];
            }
            return null;
        }
    }
}
  
```

NOTES

- 1 Create a concrete class implementing the **Container** interface.
- 2 It has an inner class **NameIterator** implementing **Iterator**:

CODE	WHAT IT DOES
<pre>public class NameRepository implements Container { public String names[] = {"Robert", "John", "Julie", "Lora"};</pre>	The data, stored as an array.
<pre> @Override public Iterator getIterator() { return new NameIterator(); }</pre>	Hands out a fresh iterator each time.
<pre> private class NameIterator implements Iterator { int index;</pre>	An inner class, so it can read <code>names</code> directly; <code>index</code> is its own cursor.
<pre> public boolean hasNext() { if (index < names.length) { return true; } return false; }</pre>	Another element exists while the cursor is inside the array.
<pre> public Object next() { if (this.hasNext()) { return names[index++]; } return null; } }</pre>	Return the current name and advance; null past the end.

SLIDE 11 / 32 Iterator Example: the code together

The slide displays the Iterator pattern implementation with a UML diagram and corresponding Java code.

UML Diagram:

- Container** (Interface): `+getIterator(): Iterator`
- Iterator** (Interface): `+hasNext(): boolean`, `+next(): Object`
- NameRepository** (Class): `-name: String []`, `+getIterator(): Iterator`. It **implements** **Container** and **has** **NameIterator**.
- NameIterator** (Class): `+hasNext(): boolean`, `+next(): Object`. It **implements** **Iterator**.
- IteratorPatternDemo** (Class): `+main(): void`. It **uses** **NameRepository**.

Code Snippets:

Iterator.java

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
}
```

Container.java

```
public interface Container {
    public Iterator getIterator();
}
```

NameRepository.java

```
public class NameRepository implements Container {
    public String names[] = {"Robert", "John", "Julie", "Lora"};

    @Override
    public Iterator getIterator() {
        return new NameIterator();
    }

    private class NameIterator implements Iterator {
        int index;

        @Override
        public boolean hasNext() {
            if(index < names.length){
                return true;
            }
            return false;
        }

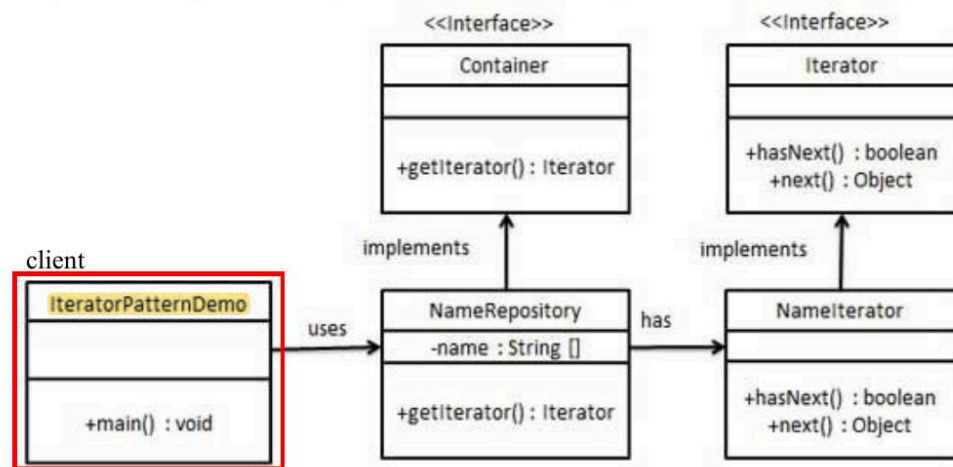
        @Override
        public Object next() {
            if(this.hasNext()){
                return names[index++];
            }
            return null;
        }
    }
}
```

| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

NOTES

- 1 All the code on one slide: the interfaces on the left and **NameRepository** with its inner **NameIterator** on the right. The client only ever sees **Container** and **Iterator**, so the names could move into a list or a database without changing it.

Iterator Design Pattern – Structure



NOTES

- 1 The diagram with the client **IteratorPatternDemo** outlined: next, the code that uses the repository.

SLIDE 13 / 32 Iterator Example: the client and output

Iterator Design Pattern – Example

Use the *NameRepository* to get iterator and print names.

IteratorPatternDemo.java

```

public class IteratorPatternDemo {

    public static void main(String[] args) {
        NameRepository namesRepository = new NameRepository();

        for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
            String name = (String)iter.next();
            System.out.println("Name : " + name);
        }
    }
}
  
```

output

```

Name : Robert
Name : John
Name : Julie
Name : Lora
  
```

NOTES

1 The client:

CODE	WHAT IT DOES
<pre>NameRepository namesRepository = new NameRepository();</pre>	The collection.
<pre>for (Iterator iter = namesRepository.getIterator(); iter.hasNext();) {</pre>	Get an iterator and loop while there is a next element.
<pre> String name = (String) iter.next(); System.out.println("Name : " + name); }</pre>	Read and print each one.

2 Output: "Name : Robert", "Name : John", "Name : Julie", "Name : Lora".

SLIDE 14 / 32 Iterator Design Pattern

Iterator Design Pattern

➤ The step-by-step approach for designing the Iterator design pattern is presented as follows:

1. Identify and design the **Iterator interface**.
2. For each class representing a collection data structure in the software system, **design a concrete Iterator** and associate it with it. Implement the concrete iterator's methods in terms of the collection data structure.
3. Create the **Collection interface**, which includes the interface method to create Iterators.
4. For each class representing a collection data structure, implement the **Aggregate interface** to instantiate and return a concrete Iterator.

NOTES

1 1. Design the **Iterator interface**.

2 2. For each collection class, design a **concrete iterator** and associate it with the collection; implement its methods in terms of that data structure.

3 3. Create the **collection (aggregate) interface** with a method that creates iterators.

4 4. For each collection class, **implement that interface** so it instantiates and returns its concrete iterator.

Iterator Design Pattern

➤ Benefits of the Iterator design pattern include:

- ✓ The Iterator provides a consistent way for clients to **iterate** through the objects in a collection.
- ✓ It **abstracts the internals** of the collection objects so that if they change, clients do not have to change.
- ✓ Allows client code to be extended easily; numerous iterators can be created to support **different traversals** from the same or different collection structure.

NOTES

- 1 A **consistent way** for clients to iterate through any collection.
- 2 It **abstracts the internals** of the collection, so if they change, clients don't.
- 3 Client code is easy to extend: many iterators can support **different traversals** (forward, reverse, filtered) over the same or different collections.

EXAM ANGLE

Prediction

Q1

Give two benefits of the Iterator pattern.

Answer: Clients traverse every collection the same way (hasNext/next); the collection's internal representation is hidden, so it can change without affecting clients; several iterators can provide different traversals (e.g. forward and reverse) of the same collection.

In class exercise

- The Company Directory: This exercise involves creating a corporate directory that can store employee records using two different **internal data structures** (e.g., a simple array and a list) but allows a client to **traverse** them in a **uniform way**. Furthermore, the client needs two specific traversal methods: **standard and reverse**.
- Which pattern to use based on the above scenarios
- Draw class
- Generate code

16

NOTES

- 1 The exercise: a company directory stores employee records in two different **internal data structures** (an array and a list) but lets clients traverse both the **same way**.
- 2 The client also needs two traversals: **standard and reverse**.
- 3 Answer: **Iterator**. A `DirectoryIterator` interface (`hasNext()`, `next()`); `ArrayDirectory` and `ListDirectory` each implement a `Directory` interface with `createIterator()` and `createReverseIterator()`, returning concrete iterators that know their structure. The client loops with `hasNext()` / `next()` whichever directory and direction it uses.

EXAM ANGLE

Prediction

Q1

A company directory stores employees in an array in one version and a linked list in another, and clients need forward and reverse traversal of both. Which pattern, and what classes?

Answer: Iterator. An `EmployeeIterator` interface (`hasNext()`, `next()`); a `Directory` interface with `createIterator()` and `createReverseIterator()`; `ArrayDirectory` and `ListDirectory` implement `Directory` and return `ArrayForwardIterator/ArrayReverseIterator` and `ListForwardIterator/ListReverseIterator`. The client uses only `Directory` and `EmployeeIterator`.

Observer Design Pattern

- The **Observer** design pattern is an **object behavioral** pattern that **standardizes** the operations between objects that interoperate using a **one-to-many relationship**.
- The **intent** of the Observer is to
 - ✓ Define a **one-to-many dependency between objects** so that when one object **changes state**, all its dependents are **notified** and updated automatically.

NOTES

- 1 **Observer** is an **object behavioural** pattern that standardises how objects in a **one-to-many** relationship work together.
- 2 Its **intent**: define a one-to-many dependency so that when one object **changes state**, all its dependents are **notified and updated automatically**. Like a newsletter: subscribe once, and every new issue is pushed to you.

EXAM ANGLE

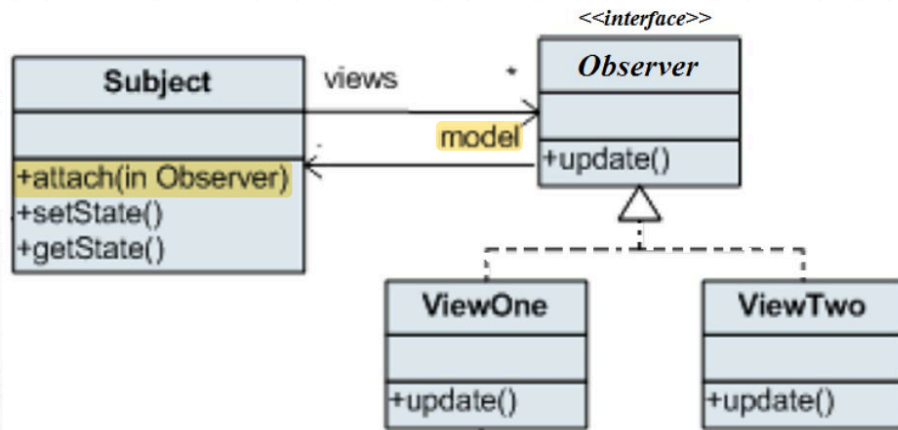
Past exam · Final 252 · Q7-2 · 2.5 marks

Q1

Global Weather Sync: when the primary weather station's temperature changes, all registered city dashboards update instantly, in Celsius and Fahrenheit. Which pattern? Draw its class diagram.

Answer: Pattern: Observer. A one-to-many dependency: when the subject changes, all observers are notified automatically. **Subject** (interface): `attach()`, `detach()`, `notifyObservers()`. **WeatherStation** (concrete subject): holds the temperature and a list of observers; `setTemperature()` stores the value and calls `notifyObservers()`. **Observer** (interface): `update(temp)`. **CelsiusDashboard** and **FahrenheitDashboard** (concrete observers) show the temperature in their own unit (Fahrenheit = $t \times 9/5 + 32$). Diagram: WeatherStation realizes Subject; the dashboards realize Observer; Subject keeps a list of Observers.

Observer Design Pattern - Structure



NOTES

- 1 **Subject:** `attach(in Observer)` to register observers, `setState()` to change, `getState()` to read. It keeps its observers (the "views", multiplicity *).
- 2 `<<interface>>` **Observer:** `update()`. Each observer holds a reference back to its subject (the "model" arrow) so it can read the new state.
- 3 **ViewOne** and **ViewTwo** realise **Observer**, each with its own `update()`. This is the same mechanism MVC uses to refresh views when the model changes.

EXAM ANGLE

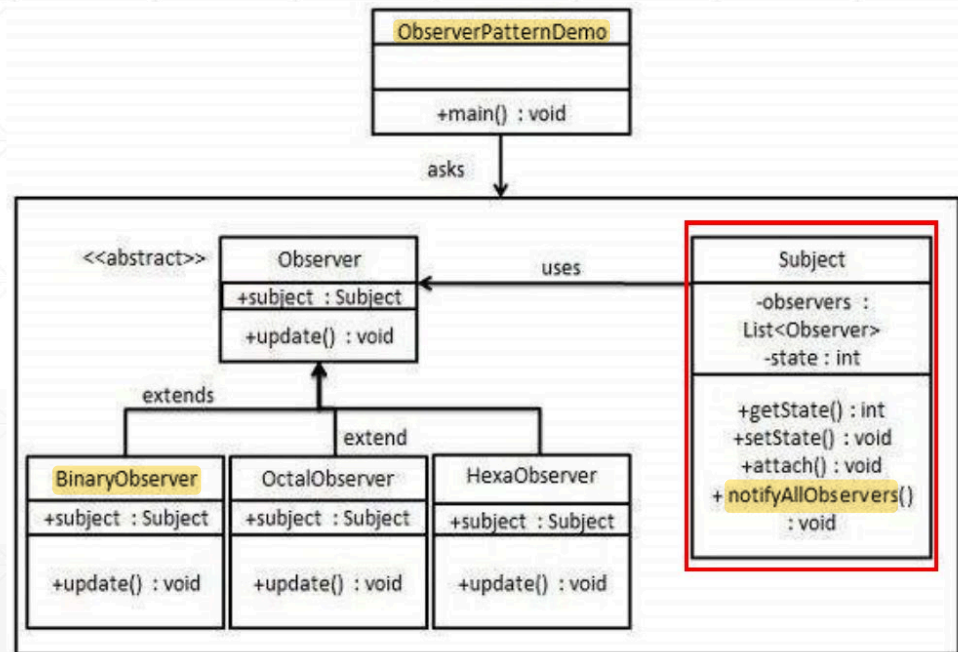
Prediction

Q1

In the Observer pattern, what happens when the subject's state changes?

Answer: The subject calls its notify method, which loops over every registered observer and calls `update()` on each; each observer then reads the new state (e.g. via `subject.getState()`) and refreshes itself.

Observer Design Pattern – Example



| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

*Source: https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

19

NOTES

- 1 The example: a **Subject** with `-observers: List<Observer>` and `-state: int`, and the operations `getState()`, `setState()`, `attach()` and `notifyAllObservers()`.
- 2 «abstract» **Observer** has `+subject: Subject` and `+update()`; **BinaryObserver**, **OctalObserver** and **HexaObserver** extend it.
- 3 **ObserverPatternDemo** is the client. Whenever the number changes, each observer prints it in its own base.

Observer Design Pattern Example

Subject.java

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

public class Subject {

    private List<Observer> observers = new ArrayList<Observer>();
    private int state;

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
        notifyAllObservers();
    }

    public void attach(Observer observer){
        observers.add(observer);
    }

    public void notifyAllObservers(){
        for (Observer observer : observers) {
            observer.update();
        }
    }
}
```

20

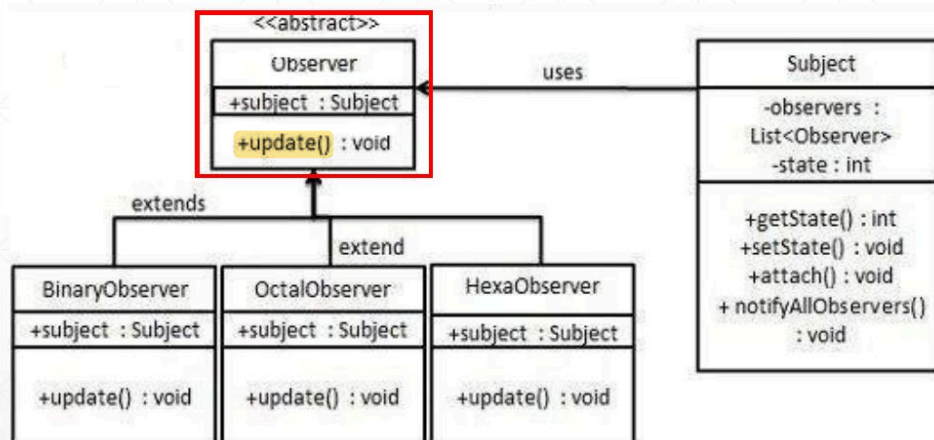
| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

NOTES

1 Subject.java:

CODE	WHAT IT DOES
private List<Observer> observers = new ArrayList<Observer>(); private int state;	The registered observers and the state they care about.
public int getState() { return state; }	Observers read the state through this.
public void setState(int state) { this.state = state; notifyAllObservers(); }	Changing the state triggers notification: the heart of the pattern.
public void attach(Observer observer) { observers.add(observer); }	Registration.
public void notifyAllObservers() { for (Observer observer : observers) { observer.update(); } }	Tell every observer, without knowing their concrete classes.

Observer Design Pattern – Example



NOTES

- 1 The diagram with the abstract **Observer** outlined: it holds `+subject: Subject` and declares `+update()`.

Observer Design Pattern – Example

Observer.java

```

public abstract class Observer {
    protected Subject subject;
    public abstract void update();
}
  
```

NOTES

1 Observer.java:

CODE

```
public abstract class Observer {  
    protected Subject subject;  
  
    public abstract void update(); }  
}
```

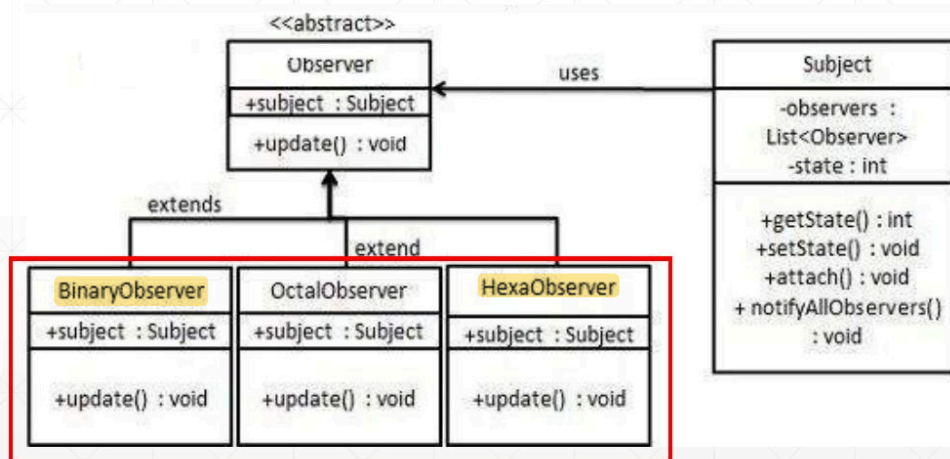
WHAT IT DOES

An abstract class here (an interface works too); every observer knows its subject.

Called by the subject on every change.

SLIDE 23 / 32 Observer Example: concrete observers

Observer Design Pattern – Example



NOTES

- 1 The diagram with the three concrete observers outlined: **BinaryObserver**, **OctalObserver** and **HexaObserver**, each implementing `update()` its own way.

Observer Design Pattern – Example

BinaryObserver.java

```
public class BinaryObserver extends Observer{

    public BinaryObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );
    }
}
```

NOTES

1 BinaryObserver.java:

CODE

```
public BinaryObserver(Subject subject)
{ this.subject = subject;
  this.subject.attach(this); }
```

```
@Override public void update() {
  System.out.println("Binary String: " +
    Integer.toBinaryString(subject.getState())); }
```

WHAT IT DOES

The observer registers itself with the subject when created.

On notification it pulls the new state and prints it in binary.

Observer Design Pattern – Example

OctalObserver.java

```
public class OctalObserver extends Observer{

    public OctalObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );
    }
}
```

NOTES

- 1 **OctalObserver** is identical except that `update()` prints `Integer.toOctalString(subject.getState())`.

Observer Design Pattern – Example

HexaObserver.java

```
public class HexaObserver extends Observer{

    public HexaObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

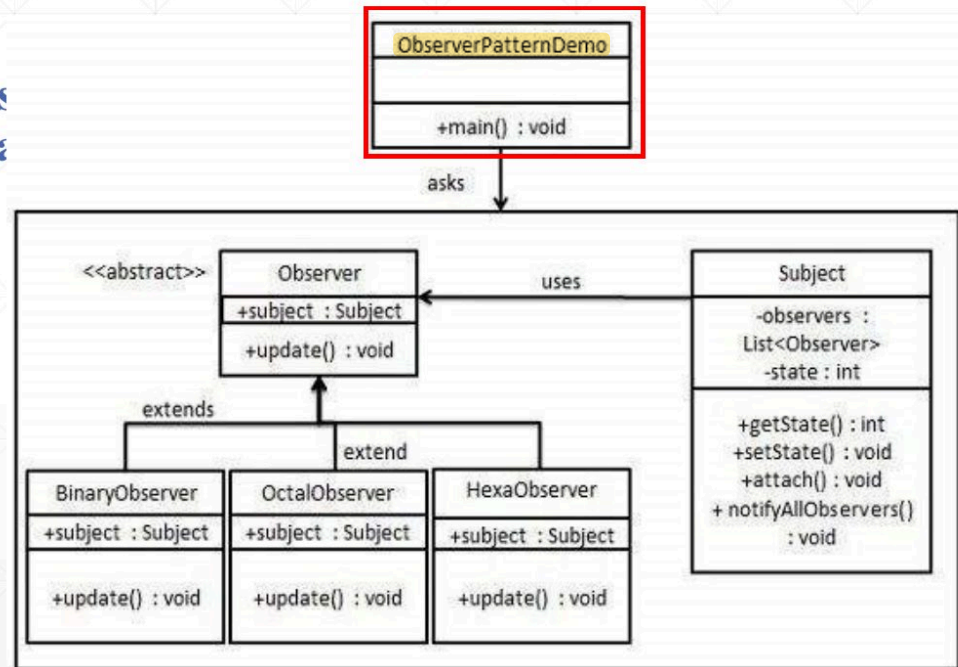
    @Override
    public void update() {
        System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );
    }
}
```

NOTES

- 1 **HexaObserver** prints `Integer.toHexString(subject.getState()).toUpperCase()`. Adding a fourth observer (say, Roman numerals) would need no change to Subject.

SLIDE 27 / 32 Observer Example: the client

Observer Design Pattern – Example



| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

*Source: https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

27

NOTES

- 1 The diagram with the client **ObserverPatternDemo** outlined: it creates the subject and the observers, then changes the state.

Observer Design Pattern – Example

ObserverPatternDemo.java

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

        new HexaObserver(subject);
        new OctalObserver(subject);
        new BinaryObserver(subject);

        System.out.println("First state change: 15");
        subject.setState(15);
        System.out.println("Second state change: 10");
        subject.setState(10);
    }
}
```

output

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

NOTES

1 The client:

CODE	WHAT IT DOES
Subject subject = new Subject();	One subject.
new HexaObserver(subject); new OctalObserver(subject); new BinaryObserver(subject);	Three observers; each attaches itself in its constructor.
System.out.println("First state change: 15"); subject.setState(15);	One state change...
System.out.println("Second state change: 10"); subject.setState(10);	...and another.

2 Output: for 15, "Hex String: F", "Octal String: 17", "Binary String: 1111"; for 10, "A", "12", "1010". Each `setState` call reached all three observers automatically, in the order they attached.

Observer Design Pattern

➤ The steps required to apply the Observer design pattern include:

1. Design the **Subject interface** and implement code for attaching, detaching, and notifying observer objects. The code for keeping track of observers can be done using existing linked-lists data structures.
2. For classes that manage information of interests to observers, **inherit from the** subject class created in step 1.
3. Design the **Observer interface**, which includes the abstract update interface method.
4. For all observers in the system, implement the Observer interface, which requires implementing the **update method**.
5. At run-time, create each observer and **attach them to the subject**. When changes occur, the subject iterates through its list of registered objects, and calls their update method.

| Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

29

NOTES

1. Design the **Subject** with code to attach, detach and notify observers (a linked list can keep track of them).
2. Classes that hold information observers care about **inherit** from that subject class.
3. Design the **Observer** interface with the abstract `update()` method.
4. Every observer implements Observer, i.e. its own `update()`.
5. At run time, create the observers and **attach** them to the subject; on each change the subject iterates through its list and calls `update()` on each.

EXAM ANGLE

Prediction

Q1

True or false: in the Observer pattern the subject must know the concrete classes of its observers.

Answer: False — It knows them only through the Observer interface, which is what keeps them loosely coupled and lets new observers be added freely.

Observer Design Pattern

➤ Benefits of the Observer design pattern:

- ✓ **Flexibility** for adding new services to the system.
- ✓ Since specific services are **compartmentalized**, maintain and modifying existing system services becomes easier.

NOTES

- 1 **Flexibility** to add new services: a new observer is just a new class that attaches itself.
- 2 Specific services are **compartmentalised** (each observer handles its own reaction), so maintaining and modifying them is easier. (Watch for: notification order, slow observers holding up the rest, and observers that are never detached.)

EXAM ANGLE

Prediction

Q1

Give one benefit and one drawback of the Observer pattern.

Answer: Benefit: new observers (services) can be added without changing the subject; each reaction is isolated in its own class. Drawback: the order of notifications isn't controlled, every change notifies every observer (cost), and forgotten observers can leak memory or react unexpectedly.

In class activity

- The Scenario: **Social Media Follower System**: Imagine you are building a simplified version of a social media platform like Instagram or X (Twitter).
- The Components: **The Subject (The Influencer)**: A high-profile user who posts updates. When they post a new "Status Update," they don't want to manually message every single follower.
- **The Observers (The Followers)**: Various users who have "Subscribed" or "Followed" the Influencer. They want to be notified automatically the moment a new post is live.
- The Exercise Task:
 - ✓ **Define the Subject**: Create an Influencer class that maintains a list of followers. It should have methods to follow(), unfollow(), and notifyFollowers().
 - ✓ **Define the Observer**: Create a Follower interface with an update(String post) method.
 - ✓ **Implement Concrete Observers**: Create a MobileAppUser and a WebBrowserUser. When they receive an update, they should print a message like: "Mobile User received notification: [Post Content]".

31

NOTES

- 1 The exercise: a simplified Instagram/X follower system.
- 2 **Subject = the influencer**, who posts status updates and doesn't want to message every follower by hand.
- 3 **Observers = the followers**, who want to be notified the moment a post goes live.
- 4 The tasks, as code:

CODE	WHAT IT DOES
<pre>interface Follower { void update(String post); }</pre>	The observer interface.
<pre>class Influencer { List<Follower> followers = new ArrayList<>(); void follow(Follower f) { followers.add(f); } void unfollow(Follower f) { followers.remove(f); }</pre>	The subject keeps its followers...
<pre>void post(String text) { notifyFollowers(text); } void notifyFollowers(String p) { for (Follower f : followers) f.update(p); } }</pre>	...and notifies all of them on each post.
<pre>class MobileAppUser implements Follower { public void update(String post) { System.out.println("Mobile User received notification: " + post); } }</pre>	Concrete observers (WebBrowserUser is the same with "Web User").

Summary

- In this session, we presented **behavioral design patterns**, including:
 - ✓ **Iterator**
 - ✓ **Observer**
- Next ... we will present **architecture and design evaluation**

NOTES

- 1 Behavioural patterns covered: **Iterator** (hide how a collection is traversed) and **Observer** (broadcast state changes to registered objects).
- 2 Next: **architecture evaluation** (Chapter 8).

Original slides from the SE322 lecture deck, with study notes. Check exact wording against the [original deck](#).
Notes and questions are study aids. Made by Shoug Alomran.