Software Design and Architecture

[Behavioral Design Patterns] – Chapter 07, L02

Lecture Outlines

- **Previously- Structural Design Patterns**
  - ✓Adapter
  - ✓Composite
  - ✓Facade
- **Behavioral Design Patterns**
  - ✓Iterator
  - ✓Observer

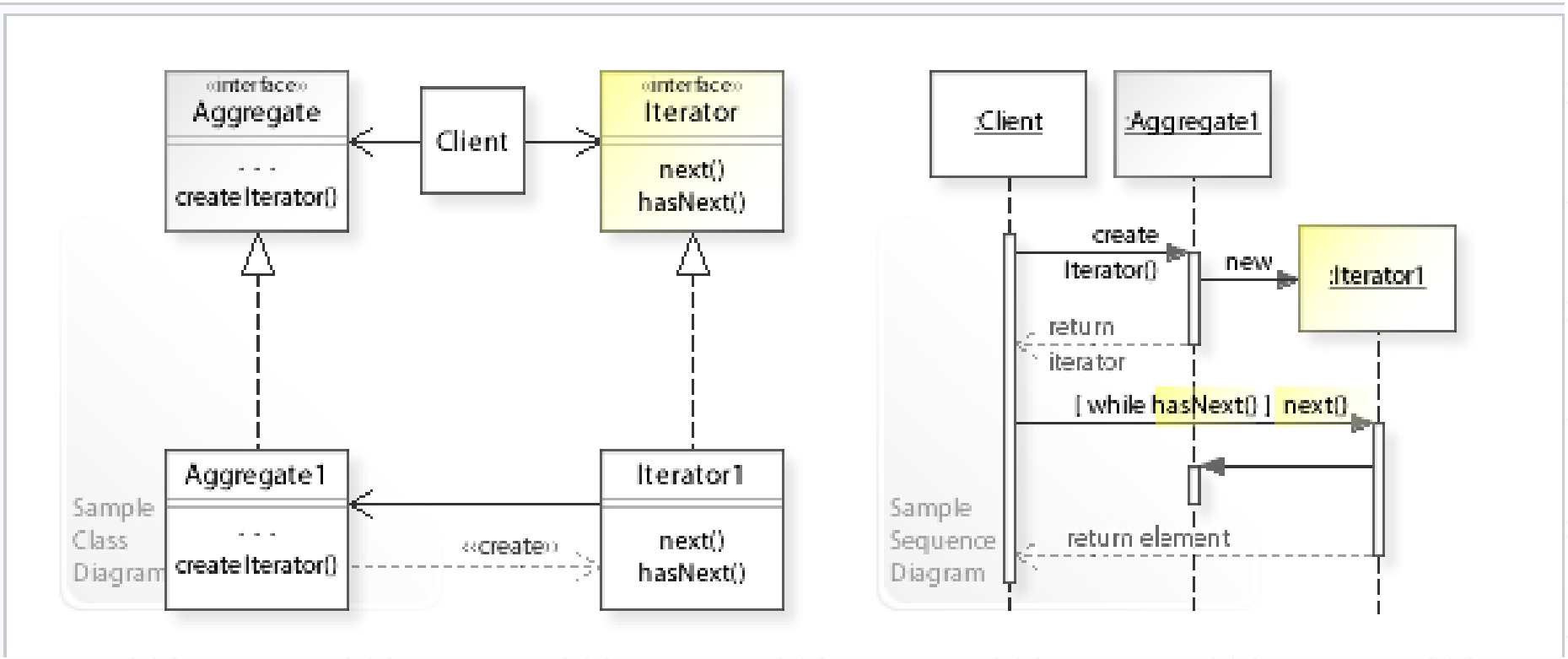
Behavioral Design Patterns

- 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

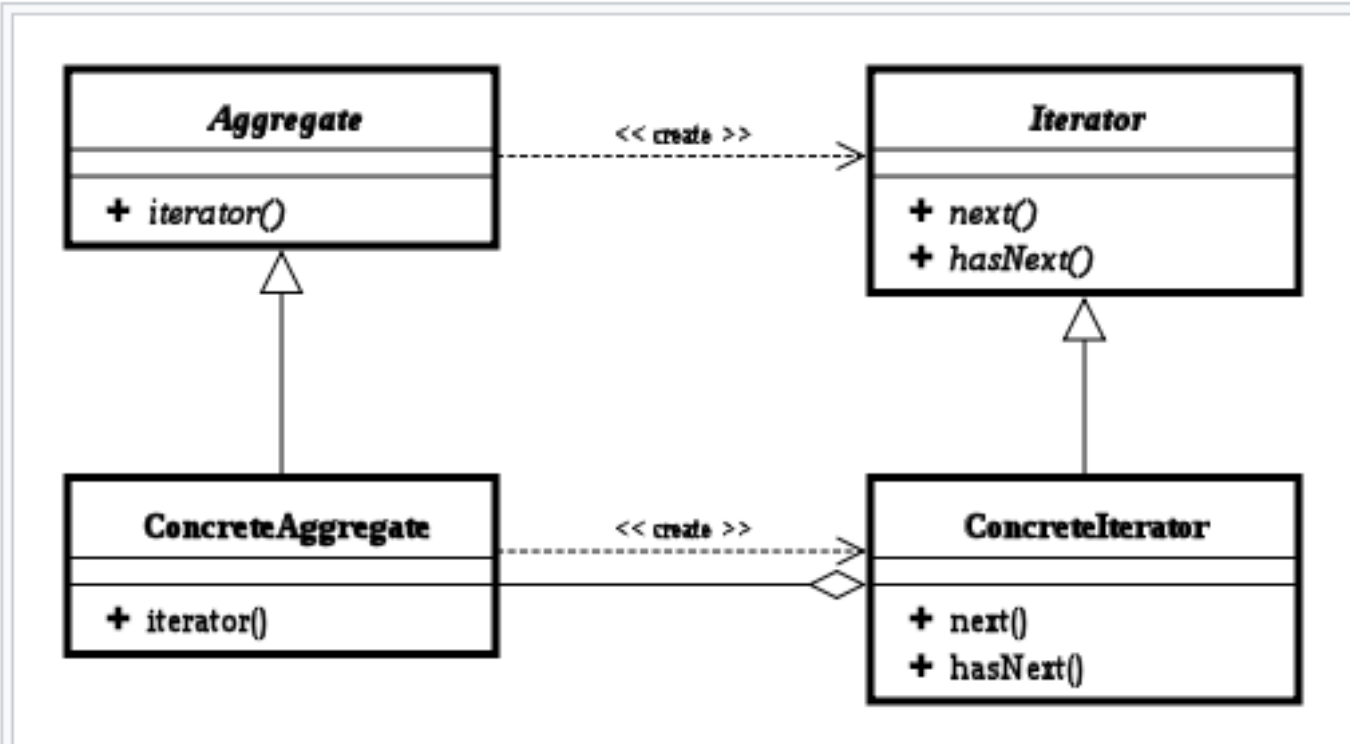
1- Iterator Design Pattern

- 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.

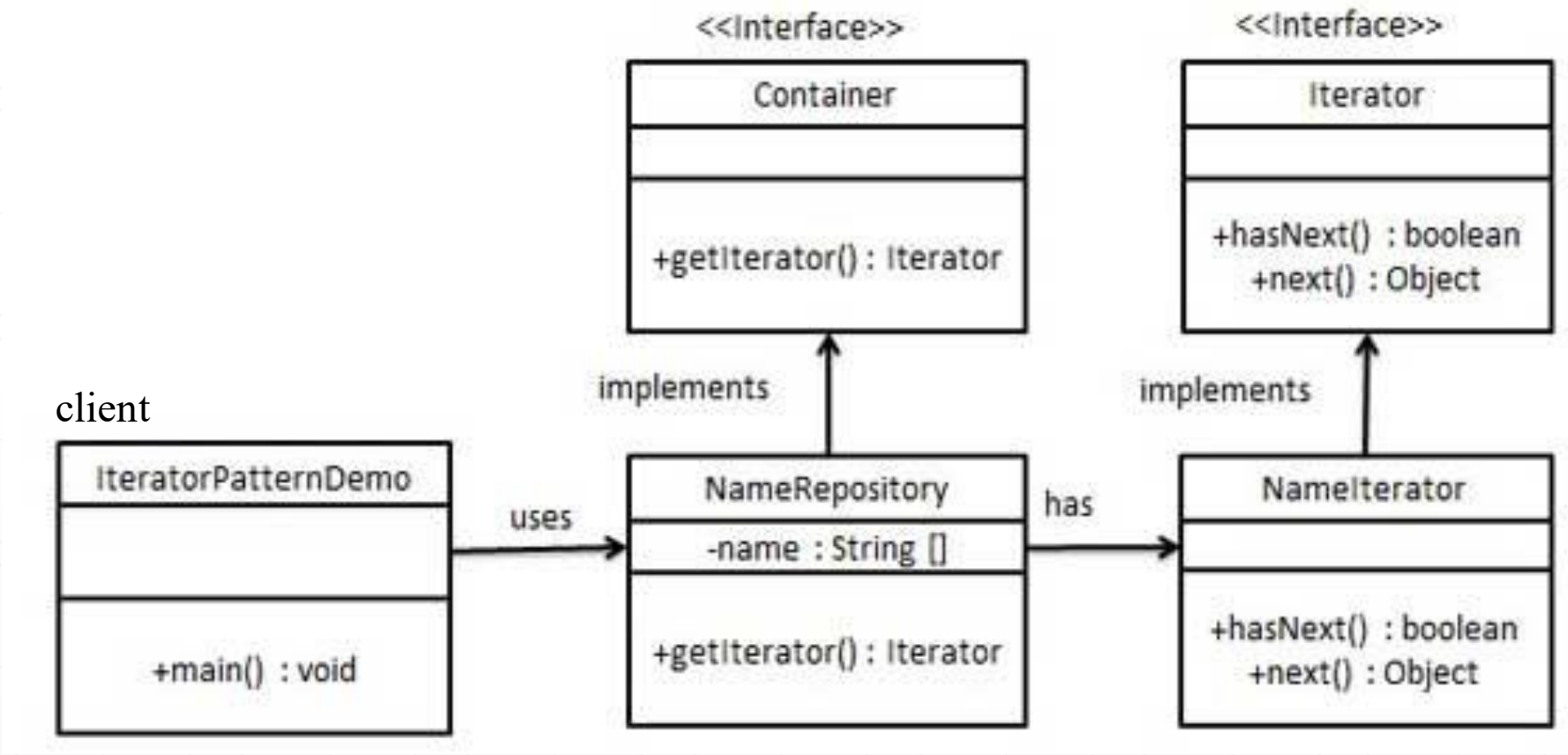


In the above UML class diagram, the Client class refers (1) to the Aggregate interface for creating an Iterator object (createIterator()) and (2) to the Iterator interface for traversing an Aggregate object (next(),hasNext()). The Iterator1 class implements the Iterator interface by accessing the Aggregate1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

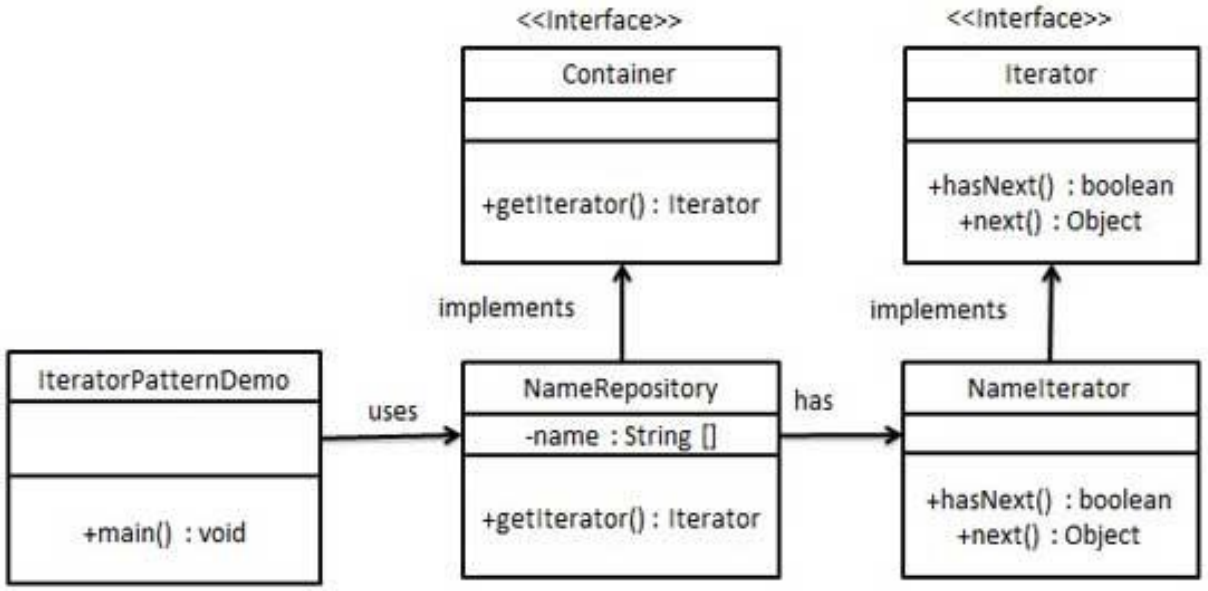


See example on https://www.geeksforgeeks.org/iterator-pattern/

Iterator Design Pattern – Structure



Iterator Design Pattern – Example*



Iterator.java

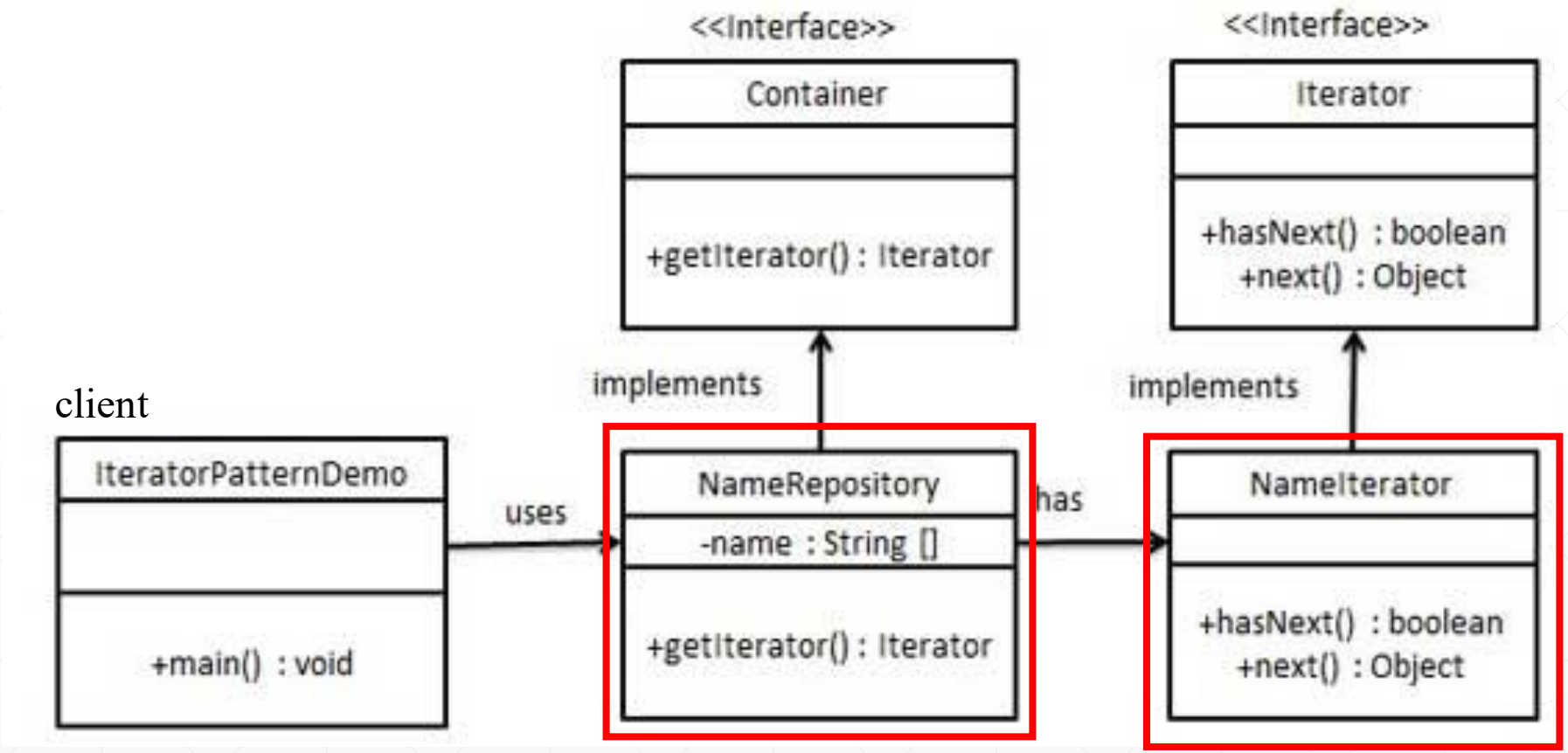
```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Container.java

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


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

Iterator Design Pattern – Structure

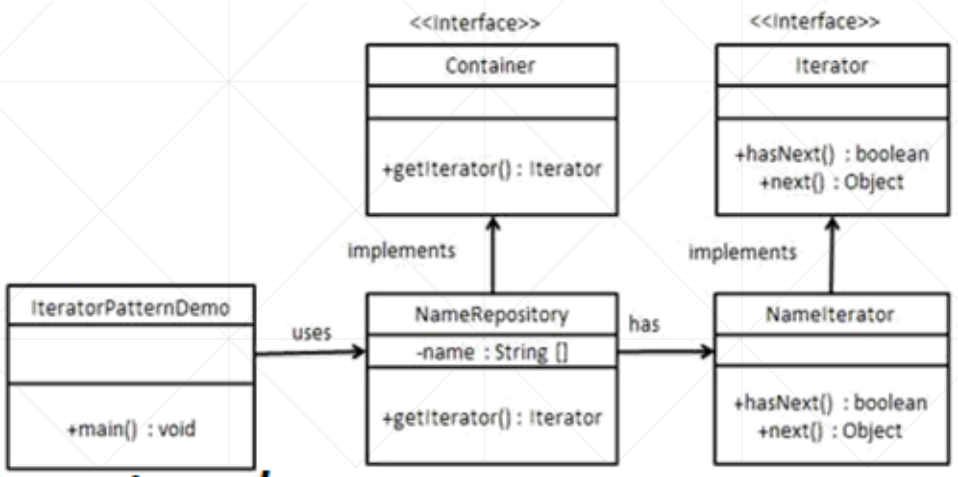


Iterator Design Pattern – Example

- 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;
      }
   }
}
```



Iterator.java

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

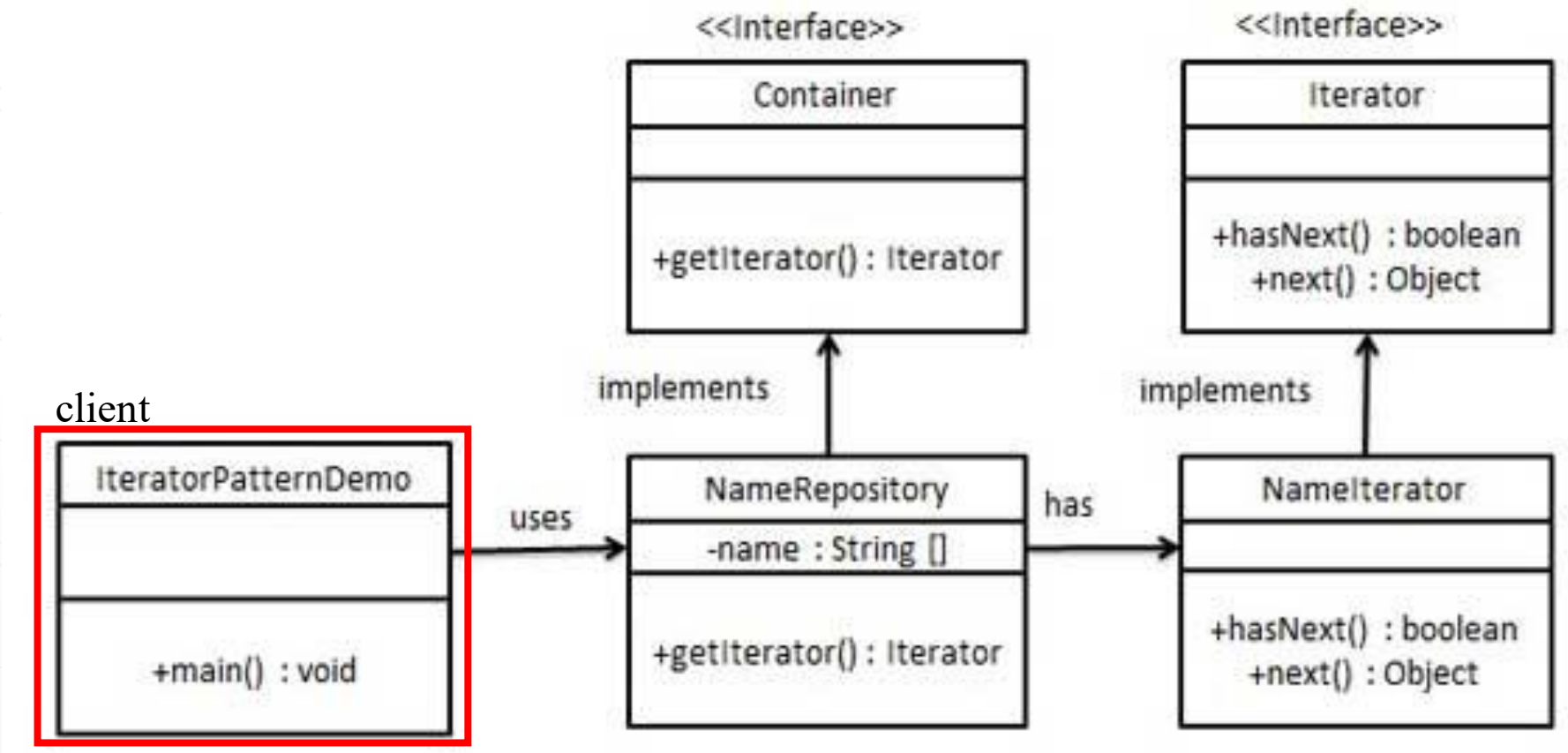
Container.java

```
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;
      }
   }
}
```

Iterator Design Pattern – Structure



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
```

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.

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.

In class exercise

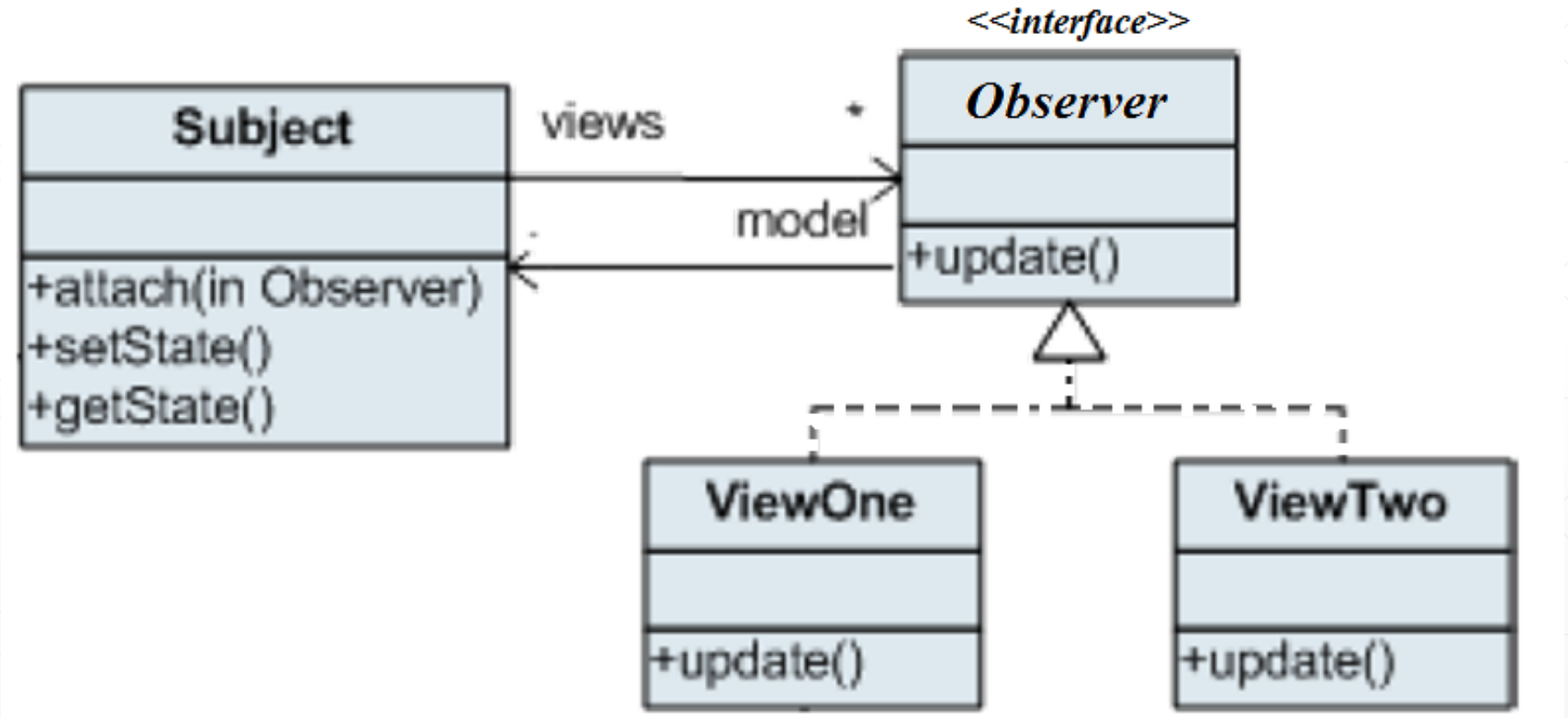
- The Company Directory: This exercise involves creating a corporate directory that can store employee records using two different internal data structures (e.g., a simple array and a list) but allows a client to traverse them in a uniform way. Furthermore, the client needs two specific traversal methods: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

Observer Design Pattern

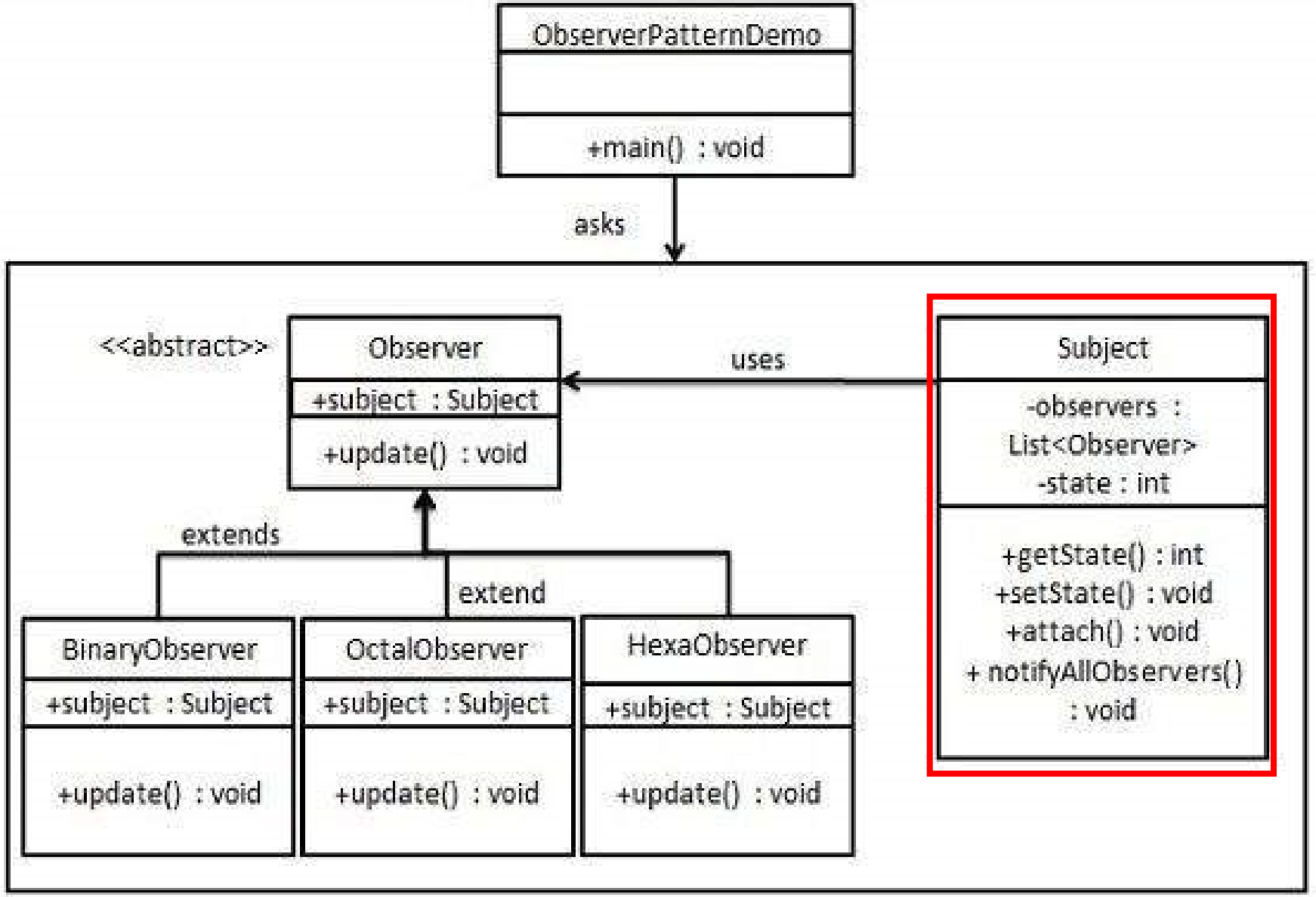
- The **Observer** design pattern is an **object behavioral** pattern that **standardizes** the operations between objects that interoperate using a one-to-many relationship.

- 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.

Observer Design Pattern - Structure



Observer Design Pattern – Example



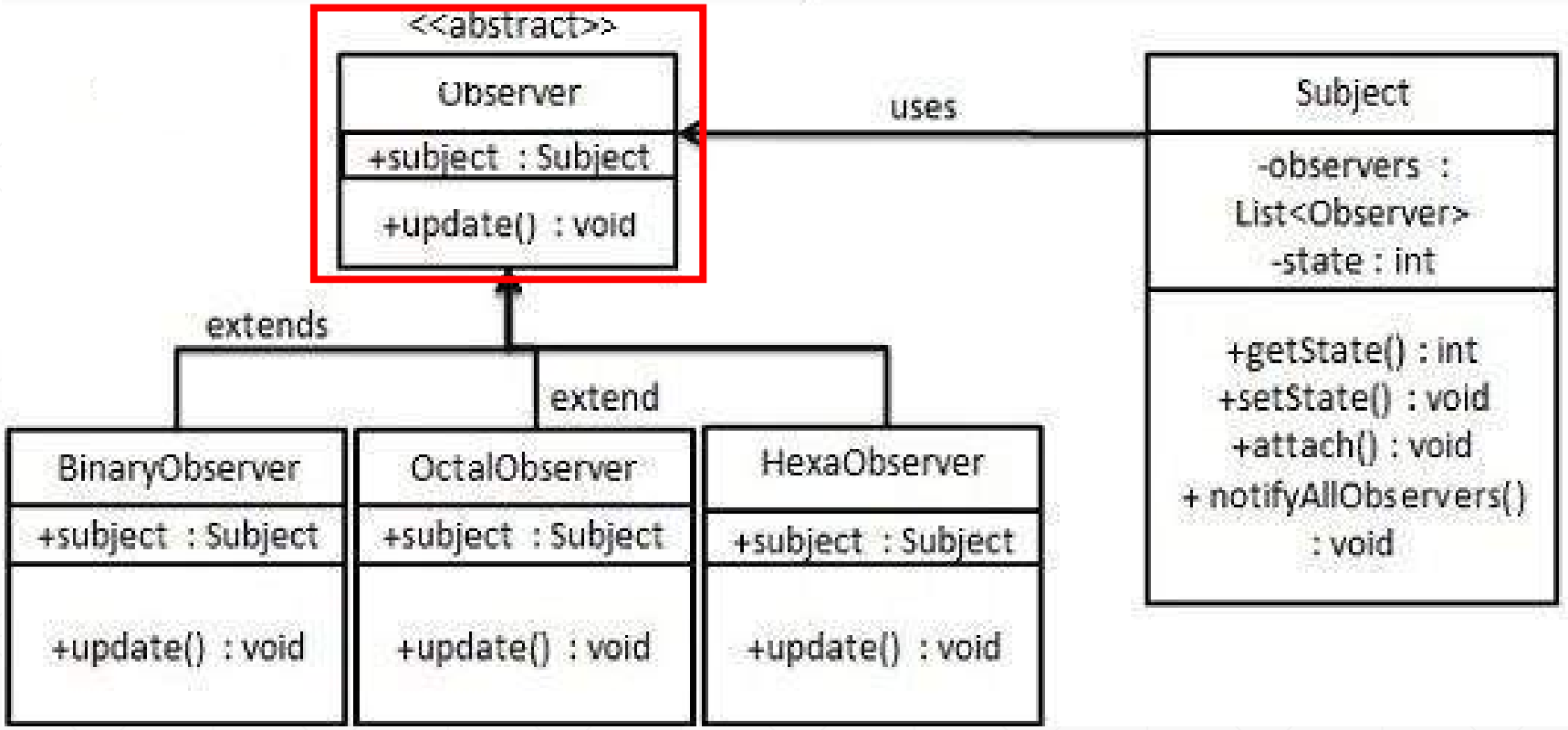
*Source: https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

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();
      }
   }
}
```

Observer Design Pattern – Example



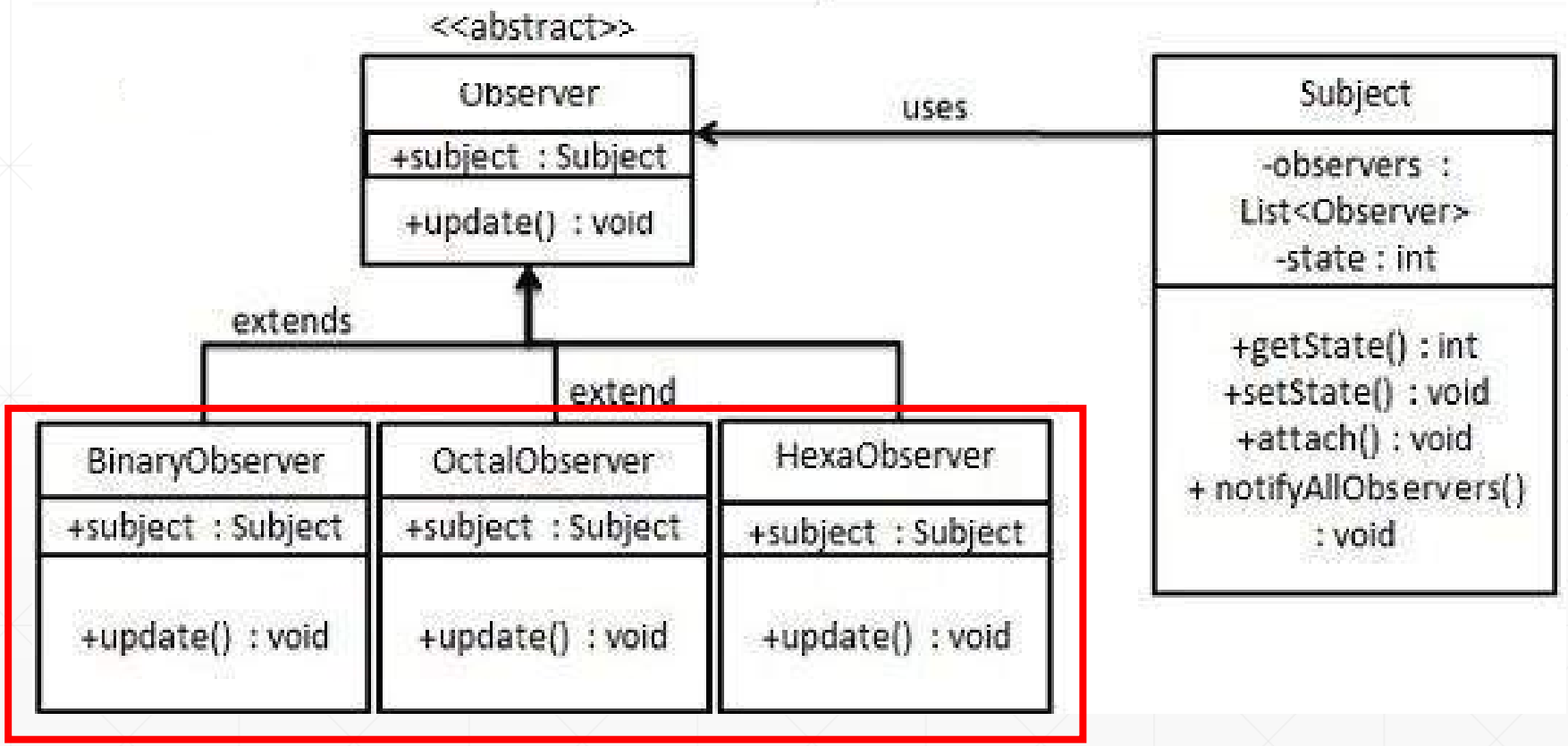
*Source: https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

Observer Design Pattern – Example

Observer.java

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

Observer Design Pattern – Example



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

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() ) );
   }
}
```

Observer Design Pattern – Example

OctalObserver.java

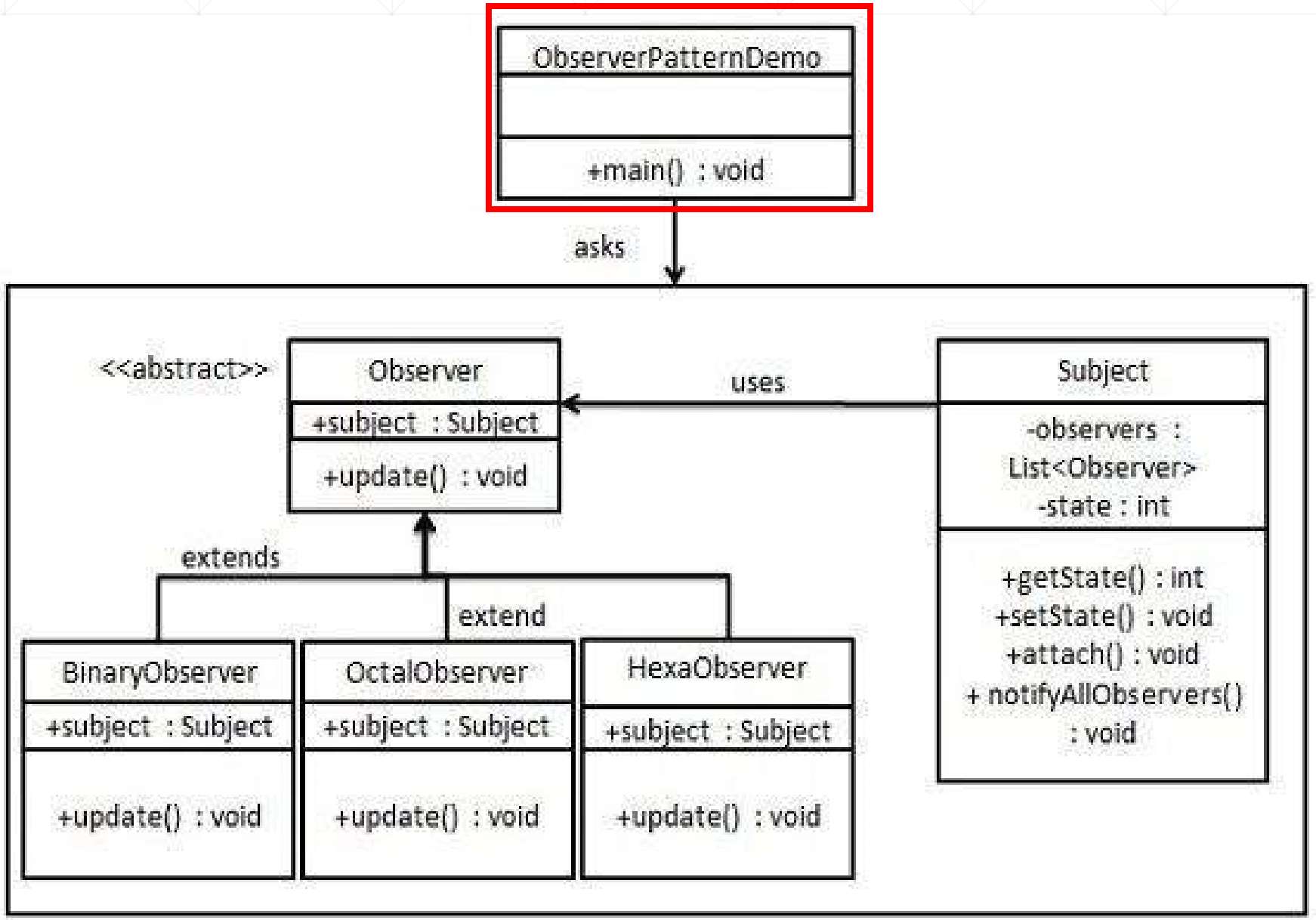
```
public class OctalObserver extends Observer{

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

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

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() );
   }
}
```

Observer Des
Pattern – Exa



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

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
```

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.

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.

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]".

Summary

- In this session, we presented **behavioral design patterns**, including:
  - ✓Iterator
  - ✓Observer

- Next … we will present architecture and design **evaluation**