9. 옵서버 패턴

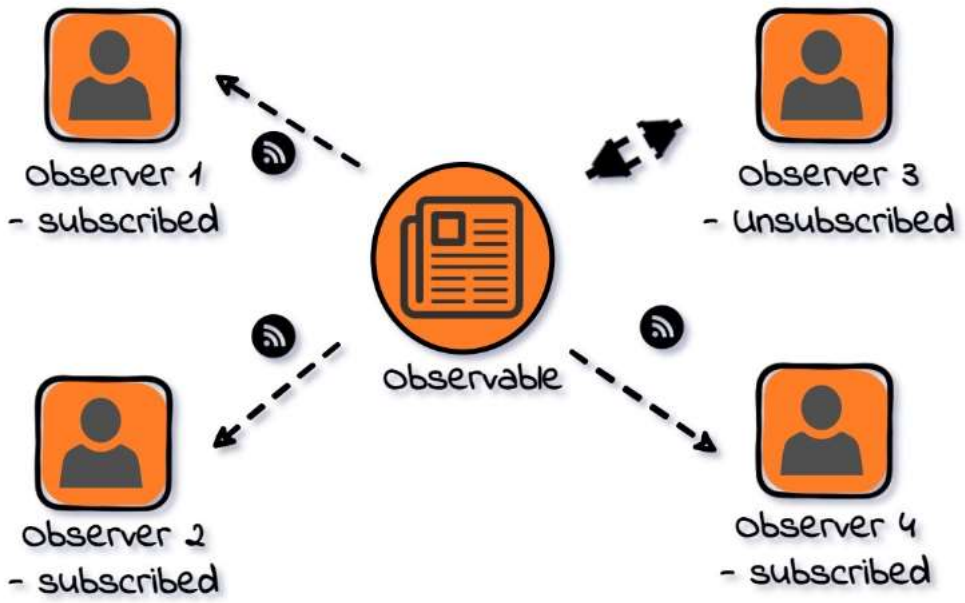
학습목표

학습목표

- 데이터의 변화를 통보하는 방법 이해하기
- 옵서버 패턴을 통한 통보의 캡슐화 방법 이해하기
- 사례 연구를 통한 옵서버 패턴의 핵심 특징 이해하기

Subject(발행자)와 Observer(구독자)의 관계

- 옵서버 패턴은 정확히는 관찰(Observe)이라기 보다 정보의 갱신이 있을 경우 해당 내용을 전달받기를 기다리는 것으로 보는 것이 적절



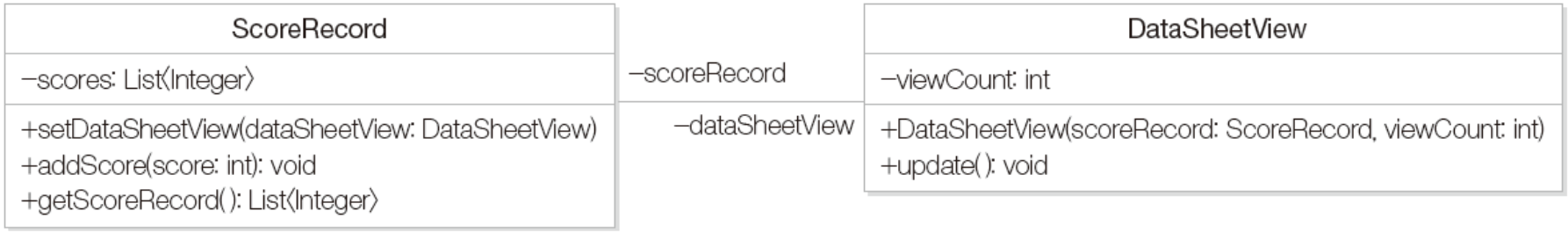
https://stackabuse.com/observer-design-pattern-in-python/

9.1 여러 가지 방식으로 성적 출력하기

❖ 성적 출력 프로그램

- ScoreRecord 클래스: 점수를 저장/관리하는 클래스
- DataSheetView 클래스: 점수를 목록형태로 출력하는 클래스

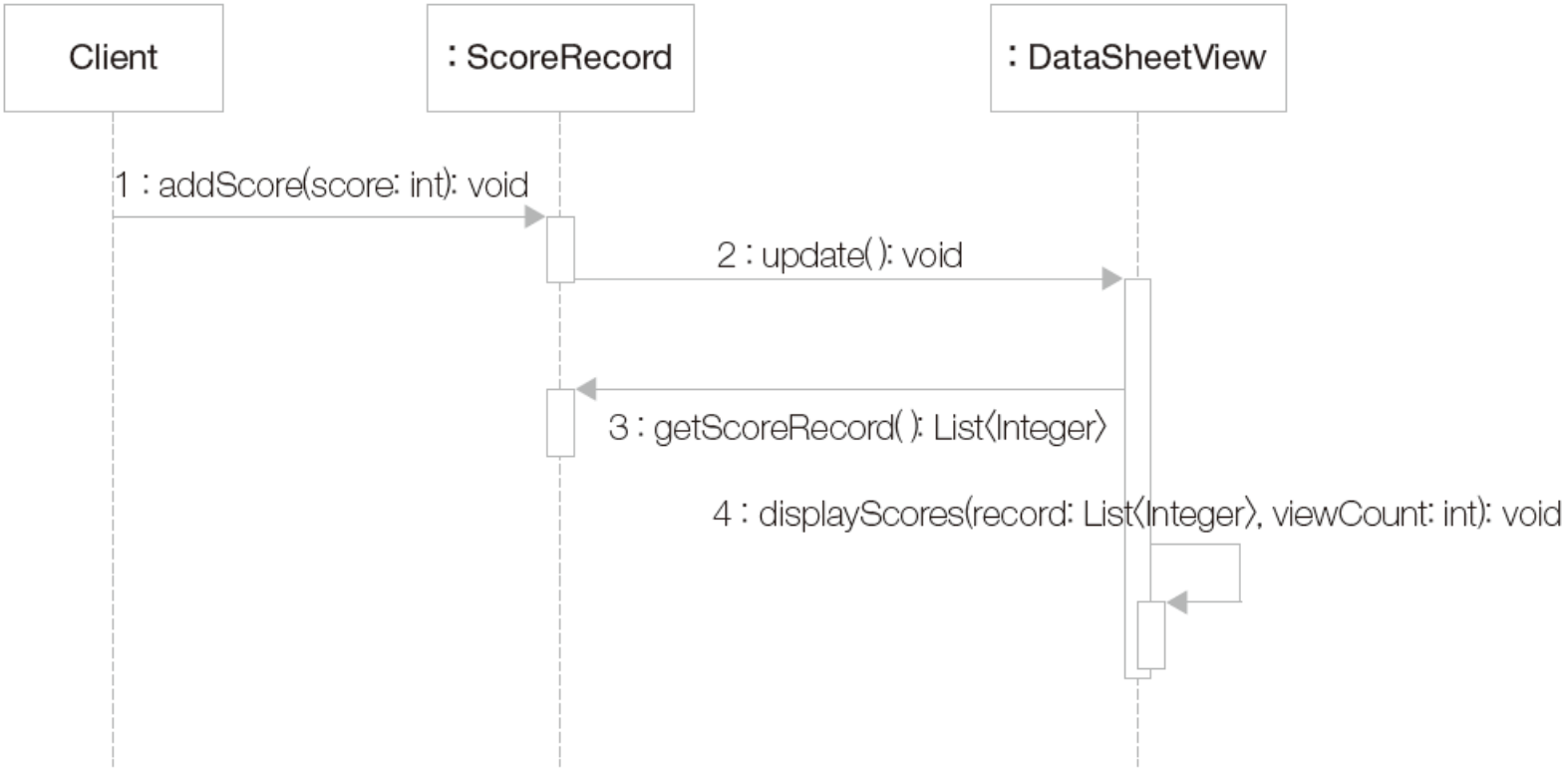
그림 9-1 ScoreRecord 클래스의 값을 출력하는 DataSheetView 클래스의 설계



9.1 여러 가지 방식으로 성적 출력하기

❖ 성적 출력 프로그램: 순차 다이어그램

그림 9-2 ScoreRecord와 DataSheetView 클래스 사이의 상호작용



소스 코드

코드 9-1

```
public class ScoreRecord {
   private List<Integer> scores = new ArrayList<Integer>() ; // 점수를 저장함
   private DataSheetView dataSheetView ; // 목록 형태로 점수를 출력하는 클래스

   public void setDataSheetView(DataSheetView dataSheetView) {
      this.dataSheetView = dataSheetView ;
   }
   public void addScore(int score) { // 새로운 점수를 추가함
      scores.add(score) ; // scores 목록에 주어진 점수를 추가함
      dataSheetView.update() ; // scores가 변경됨을 통보함
   }
   public List<Integer> getScoreRecord() {
      return scores ;
   }
}
```

소스 코드

코드 9-1

```
public class DataSheetView {
   private ScoreRecord scoreRecord ;
   private int viewCount ;

   public DataSheetView(ScoreRecord scoreRecord, int viewCount) {
      this.scoreRecord = scoreRecord ;
      this.viewCount = viewCount ;
   }
   public void update() {  // 점수의 변경을 통보 받음
      List<Integer> record = scoreRecord.getScoreRecord() ; // 점수를 조회함
      displayScores(record, viewCount);  // 조회된 점수를 viewCount만큼 출력함
   }
   private void displayScores(List<Integer> record, int viewCount) {
      System.out.print("List of " + viewCount + " entries: ") ;
      for ( int i = 0 ; i < viewCount && i < record.size() ; i ++ ) {
         System.out.print(record.get(i) + " ") ;
      }
      System.out.println() ;
   }
}
```

소스 코드

코드 9-1

```
public class Client {
   public static void main(String[] args) {
      ScoreRecord scoreRecord = new ScoreRecord() ;
      // 3개까지의 점수만 출력함
      DataSheetView dataSheetView = new DataSheetView(scoreRecord, 3) ;

      scoreRecord.setDataSheetView(dataSheetView) ;

      for (int index = 1  ; index <= 5 ; index ++ ) {
         int score = index * 10 ;
         System.out.println("Adding " + score) ;
         // 10 20 30 40 50을 추가함, 추가할 때마다 최대 3개의 점수만 출력함
         scoreRecord.addScore(score) ;
      }
   }
}
```

9.2 문제점

- 성적을 다른 방식으로 출력하고 싶다면 어떤 변경 작업을 해야 하는가? 예를 들어 성적을 목록으로 출력하지 않고 최소/최대값만을 출력하려면?
- 뿐만 아니라 성적을 동시에 여러 가지 형태로 출력하려면 어떤 변경 작업을 해야 하는가? 예를 들어 성적이 입력되면 최대 3개 목록으로 출력, 최대 5개 목록으로 출력 그리고 동시에 최소/최대값만을 출력하려면?
- 그리고 프로그램이 실행 시에 성적의 출력 대상이 변경되는 것을 지원한다면 어떤 변경 작업을 해야 하는가? 예를 들어 처음에는 목록으로 출력하고 나중에는 최소/최대값을 출력하려면?

9.2.1 성적을 다른 형태로 출력하는 경우

❖ 최소/최대 값만 출력

코드 9-2

```
public class MinMaxView {  // 전체 점수가 아니라 최소/최대값만을 출력하는 클래스
   private ScoreRecord scoreRecord ;

   public MinMaxView(ScoreRecord scoreRecord) {
      this.scoreRecord = scoreRecord ;
   }
   public void update() {
      List<Integer> record = scoreRecord.getScoreRecord() ;
      displayMinMax(record);  // 최소/최대값만을 출력
   }
   private void displayMinMax(List<Integer> record) {
      int min =  Collections.min(record, null) ;
      int max =  Collections.max(record, null) ;
      System.out.println("Min: " + min + " Max: " + max) ;
   }
}
```

9.2.1 성적을 다른 형태로 출력하는 경우

코드 9-2

```
public class ScoreRecord {
   private List<Integer> scores = new ArrayList<Integer>() ;
   private MinMaxView minMaxView ;

   public void setStatisticsView(MinMaxView minMaxView) {  // MinMaxView를 설정함
      this.minMaxView = minMaxView;
   }
   public void addScore(int score) {
      scores.add(score) ;
      minMaxView.update() ;  // MinMaxView에게 점수의 변경을 통보함
   }
   public List<Integer> getScoreRecord() {
      return scores ;
   }
}
```

- MinMaxView를 이용하도록 소스코드가 수정되었음
- 기능 변경을 위해서 기존 소스 코드를 수정하므로 OCP를 위반하는 것임

9.2.1 성적을 다른 형태로 출력하는 경우

코드 9-2

```
public class Client {
   public static void main(String[] args) {
      ScoreRecord scoreRecord = new ScoreRecord() ;
      MinMaxView minMaxView = new MinMaxView(scoreRecord) ;

      scoreRecord.setMinMaxView(minMaxView) ;

      for (int index = 1  ; index <= 5 ; index ++ ) {
         int score = index * 10 ;
         System.out.println("Adding " + score) ;
         // 10 20 30 40 50을 추가함, 추가할 때마다 최소/최대 점수만 출력함
         scoreRecord.addScore(score) ;
      }
   }
}
```

9.2.2 동시에 여러 가지 방식으로 성적을 출력하는 경우

코드 9-3

```
public class ScoreRecord {
   private List<Integer> scores = new ArrayList<Integer>() ;
   private List<DataSheetView> dataSheetViews = new ArrayList<DataSheetView>() ;
   private MinMaxView minMaxView ;
   public void addDataSheetView(DataSheetView dataSheetView) {
      dataSheetViews.add(dataSheetView) ;
   }
   public void setMinMaxView(MinMaxView minMaxView) {
      this.minMaxView = minMaxView ;
   }
   public void addScore(int score) {
      scores.add(score) ;
      for ( DataSheetView dataSheetView: dataSheetViews )
         dataSheetView.update() ;  // 각 DataSheetView에게 점수의 변경을 통보
      minMaxView.update() ; // MinMaxView에게 점수의 변경을 통보
   }
   public List<Integer> getScoreRecord() { return scores ; }
}
// DataSheetView 클래스는 코드 9-1과 동일
// MinMaxView 클래스는 코드 9-2와 동일
```

- 기능 변경을 위해서 기존 소스 코드를 수정하므로 OCP를 위반하는 것임

9.2.2 동시에 여러 가지 방식으로 성적을 출력하는 경우

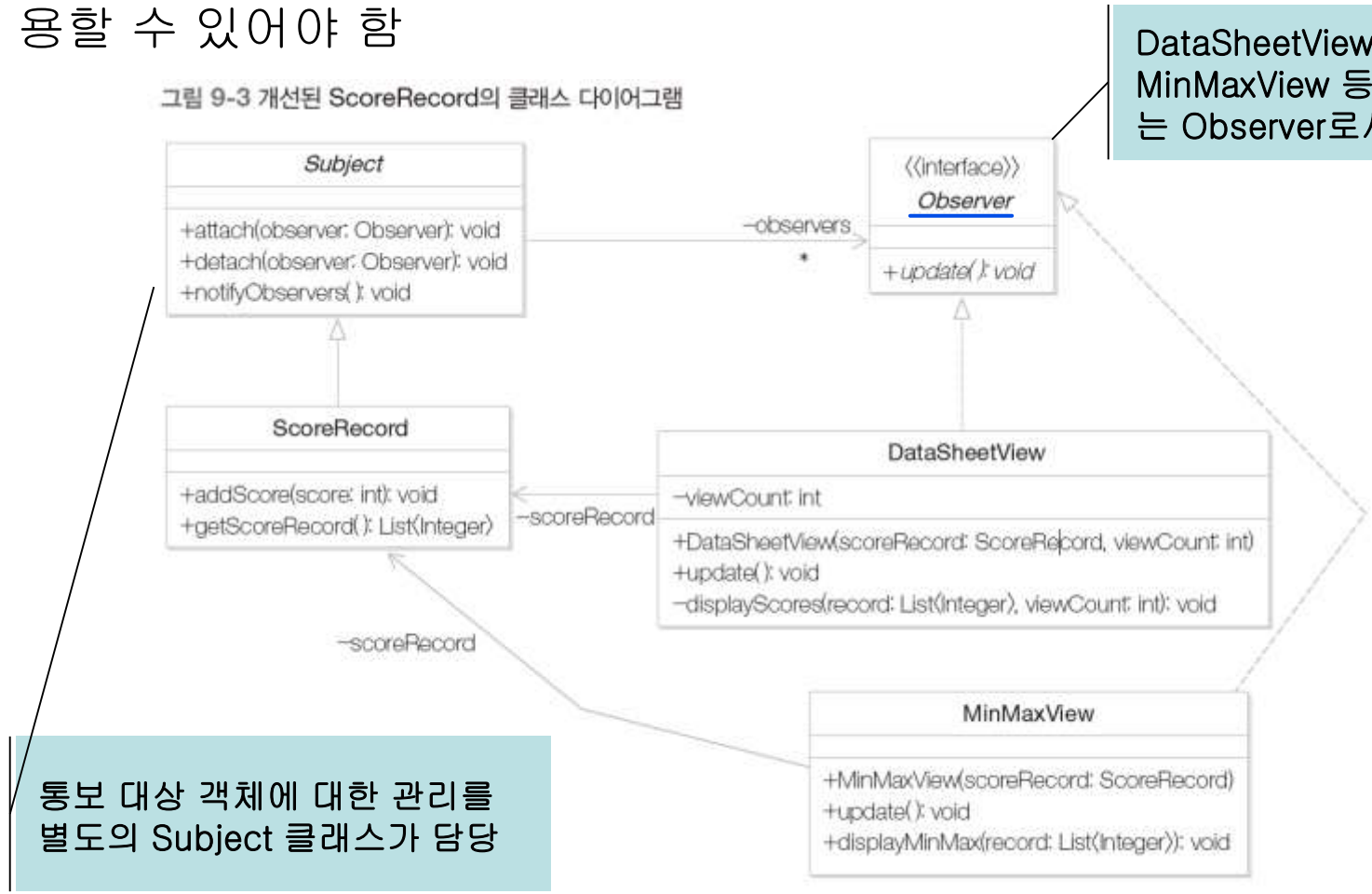
코드 9-3

```
public class Client {
   public static void main(String[] args) {
      ScoreRecord scoreRecord = new ScoreRecord() ;
      // 3개 목록의 DataSheetView 생성
      DataSheetView dataSheetView3 = new DataSheetView(scoreRecord, 3) ;
      // 5개 목록의 DataSheetView 생성
      DataSheetView dataSheetView5 = new DataSheetView(scoreRecord, 5) ;
      MinMaxView minMaxView = new MinMaxView(scoreRecord) ;

      scoreRecord.addDataSheetView(dataSheetView3) ;
      scoreRecord.addDataSheetView(dataSheetView5) ;
      scoreRecord.setMinMaxView(minMaxView) ;

      for (int index = 1  ; index <= 5 ; index ++ ) {
         int score = index * 10 ;
         System.out.println("Adding " + score) ;
         // 10 20 30 40 50을 추가함
         // 추가할 때마다 최대 3개목록, 최대 5개 목록, 그리고 최소/최대 점수가 출력됨
         scoreRecord.addScore(score) ;
      }
   }
}
```

9.3. 해결책

❖ 성적 통보 대상이 변경되더라도 ScoreRecord 클래스를 그대로 재사용할 수 있어야 함

그림 9-3 개선된 ScoreRecord의 클래스 다이어그램



DataSheetView, MinMaxView 등 대상 클래스는 Observer로서 구현

통보 대상 객체에 대한 관리를 별도의 Subject 클래스가 담당

9.3. 해결책: 소스 코드

코드 9-4

```
public interface Observer {  // 추상화된 통보 대상
  void update() ;
}
```

```
public abstract class Subject { // 추상화된 변경 관심 대상 데이터
  private List<Observer> observers = new ArrayList<Observer>() ;
  public void attach(Observer observer) { // 옵서버 즉 통보 대상을 추가함
    observers.add(observer) ;
  }
  public void detach(Observer observer) { // 옵서버 즉 통보 대상을 제거함
    observers.remove(observer) ;
  }
  // 통보 대상 목록, 즉 observers의 각 옵서버에게 변경을 통보함
  public void notifyObservers() {
    for ( Observer o : observers ) o.update() ;
  }
}
```

9.3. 해결책: 소스 코드

코드 9-4

```
public class ScoreRecord extends Subject {  // 구체적인 변경 감시 대상 데이터
  private List<Integer> scores = new ArrayList<Integer>() ;
  public void addScore(int score) {
    scores.add(score) ;
    // 데이터가 변경되면 Subject 클래스의 notifyObservers 메서드를 호출해
    // 각 옵서버(통보 대상 객체)에게 데이터의 변경을 통보함
    notifyObservers() ;
  }
  public List<Integer> getScoreRecord() {
    return scores ;
  }
}
```

```
// DataSheetView는 Observer의 기능 즉 update 메서드를 구현함으로써 통보 대상이 됨
public class DataSheetView implements Observer {
// 코드 9-1과 동일
}
```

```
// MinMaxView는 Observer의 기능 즉 update 메서드를 구현함으로써 통보 대상이 됨
public class MinMaxView implements Observer {
// 코드 9-2과 동일
}
```

9.3. 해결책: 소스 코드

코드 9-4

```
public class Client {
  public static void main(String[] args) {
    ScoreRecord scoreRecord = new ScoreRecord() ;
    DataSheetView dataSheetView3 = new DataSheetView(scoreRecord, 3) ;
    DataSheetView dataSheetView5 = new DataSheetView(scoreRecord, 5) ;
    MinMaxView minMaxView = new MinMaxView(scoreRecord) ;
    // 3개 목록 DataSheetView를 ScoreRecord에 Observer로 추가함
    scoreRecord.attach(dataSheetView3) ;
    // 5개 목록 DataSheetView를 ScoreRecord에 Observer로 추가함
    scoreRecord.attach(dataSheetView5) ;
    // MinMaxView를 ScoreRecord에 Observer로 추가함
    scoreRecord.attach(minMaxView) ;

    for (int index = 1  ; index <= 5 ; index ++ ) {
      int score = index * 10 ;
      System.out.println("Adding " + score) ;
      scoreRecord.addScore(score) ;
    }
  }
}
```

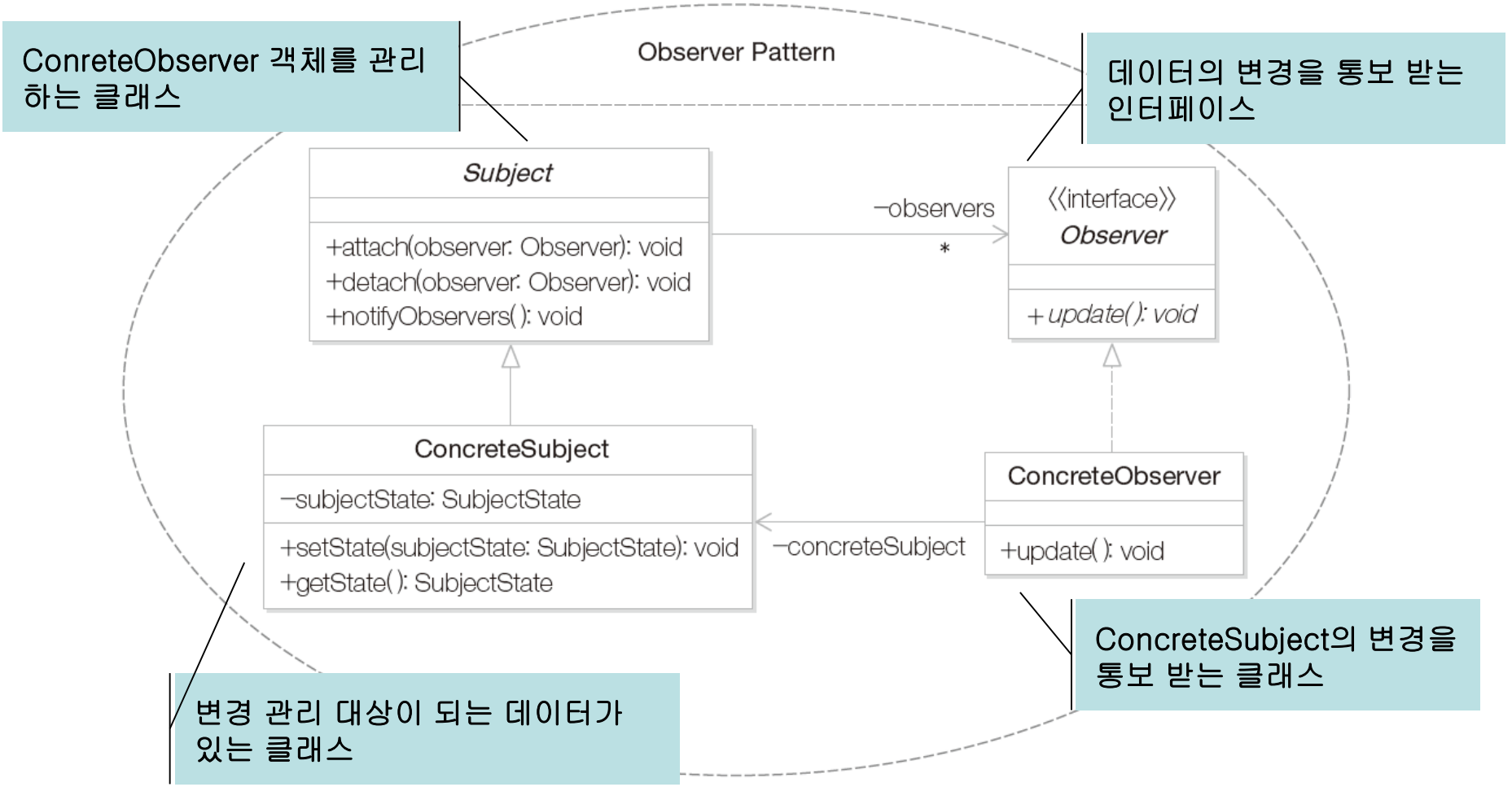
9.4 옵서버 패턴

- 데이터의 변경이 발생하였을 때 상대 클래스 및 객체에 의존하지 않으면서 데이터 변경을 통보하고자 할 때

옵서버 패턴은 통보 대상 객체의 관리를 Subject 클래스와 Observer 인터페이스로 일반화한다. 그러면 데이터 변경을 통보하는 클래스(ConcreteSubject)는 통보 대상 클래스/객체ConcreteObserver)에 대한 의존성을 제거할 수 있다. 결과적으로 옵서버 패턴은 통보 대상 클래스나 대상 객체의 변경에도 ConcreteSubject 클래스를 수정 없이 그대로 사용할 수 있도록 한다.

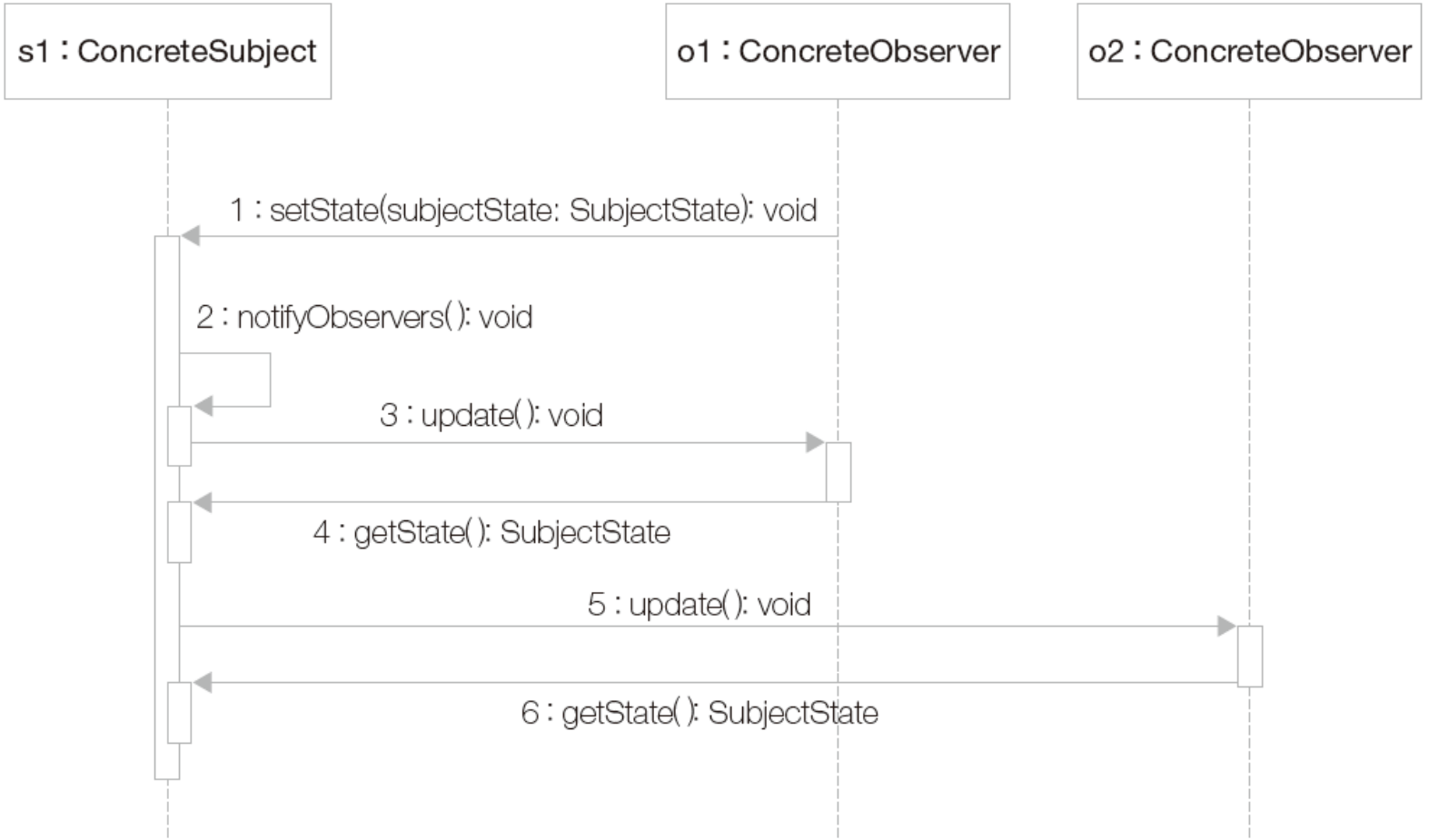
9.4 옵서버 패턴

그림 9-4 옵서버 패턴의 컬레보레이션



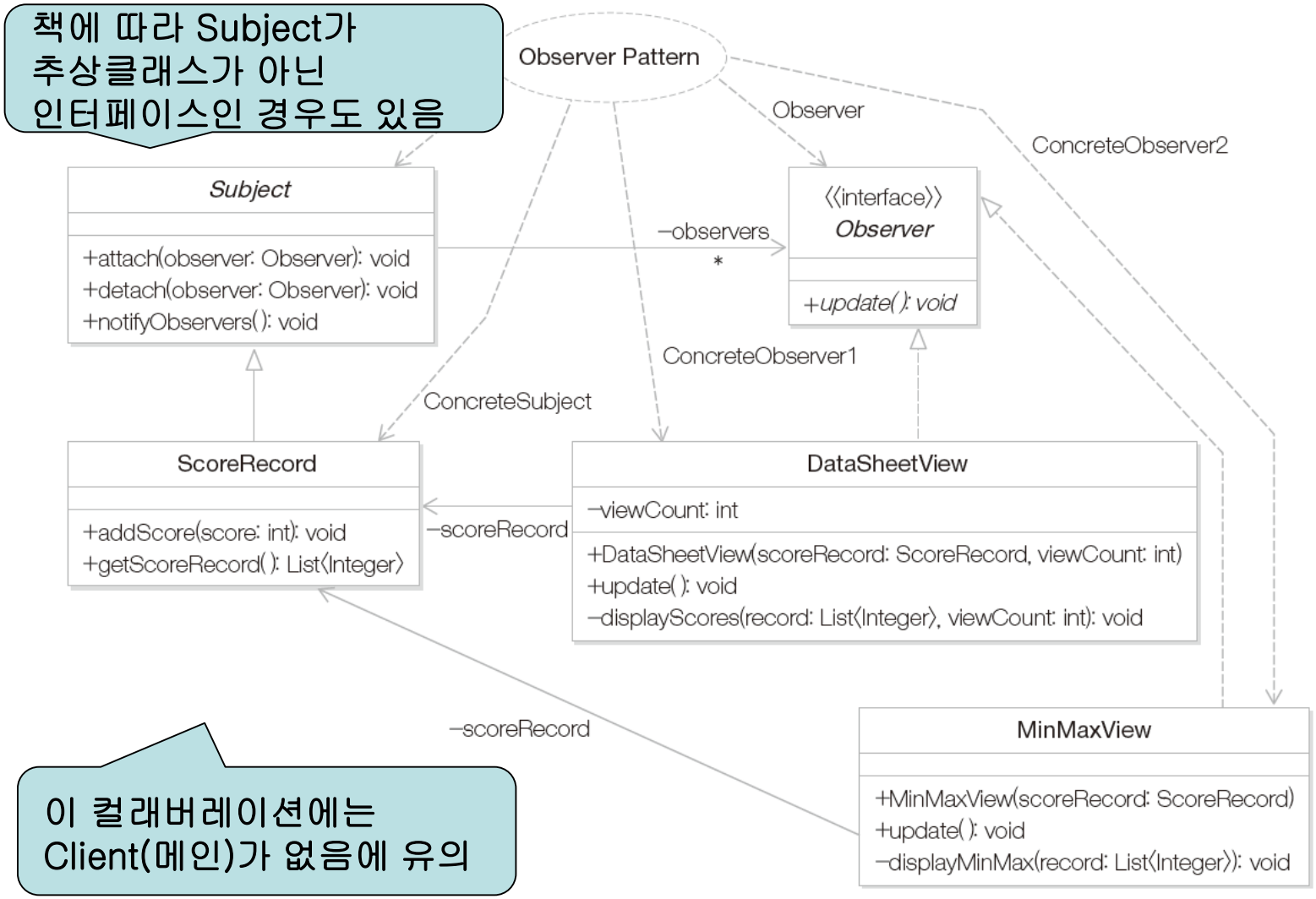
9.4 옵서버 패턴

그림 9-5 옵서버 패턴의 순차 다이어그램

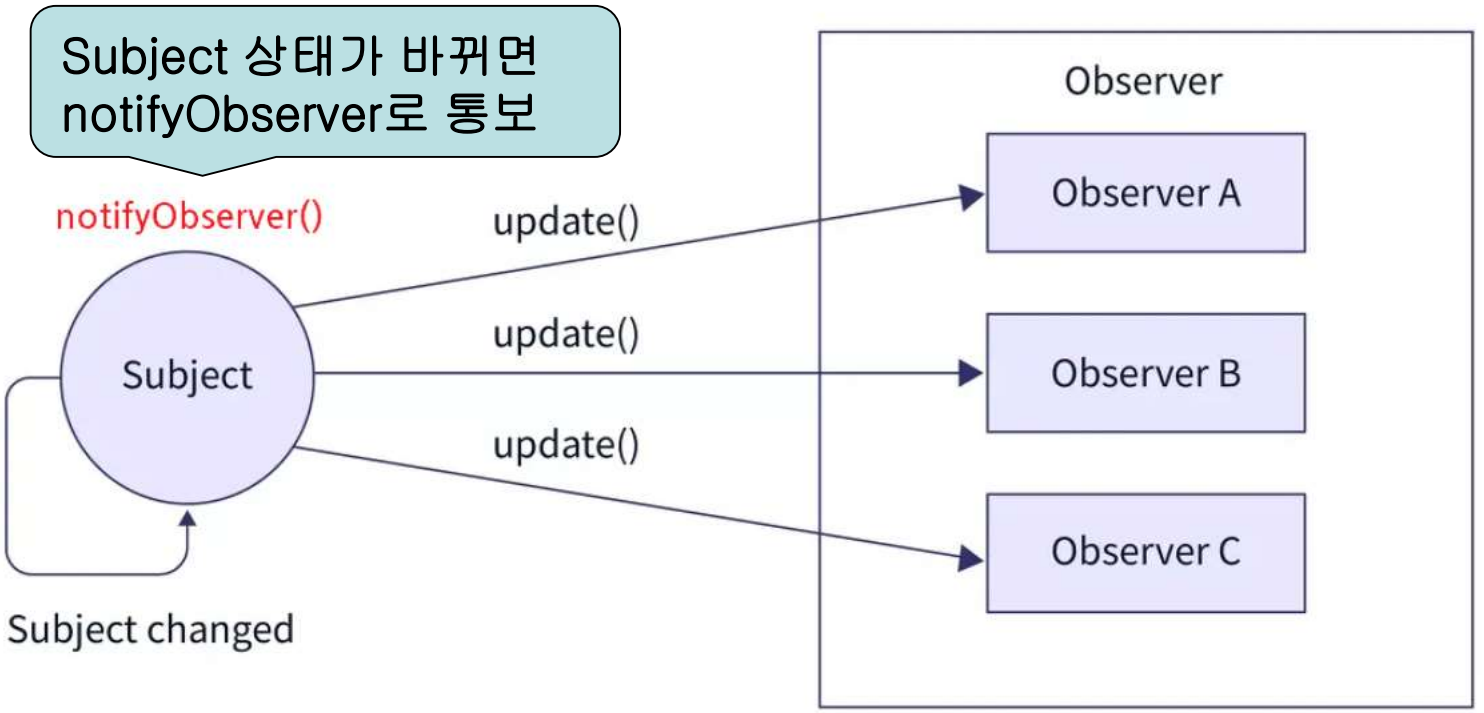


옵서버 패턴의 적용

그림 9-6 옵서버 패턴을 성적 출력하기 예제에 적용한 경우



옵서버 패턴 흐름



https://www.scaler.com/topics/design-patterns/observer-design-pattern/

유튜브 채널 구독/해지 구현

```
interface Subject{
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObserver();
}
```

```
interface Observer{
    void update();
}
```

유튜브 채널 구독/해지 구현

```
class Channel implements Subject{
    List<Observer> observers = new ArrayList<>();

    @Override
    public void registerObserver(Observer o){
        observers.add(o);
        System.out.println(o + " 님이 구독을 시작하셨습니다.");
    }
    @Override
    public void removeObserver(Observer o){
        observers.remove(o);
        System.out.println(o + "님이 구독을 해지하였습니다.");

    }
    @Override
    public void notifyObserver(){
        for(Observer o : observers){
            o.update();
        }
    }
}
```

유튜브 채널 구독/해지 구현

```
class BigFan implements Observer{
    private String name;
    public BigFan(String name){
            this.name=name;
    }
    public void update(){
        System.out.println(name+"님, Channel에 새로운 영상이 업로드 되었습니다.");
    }
}
```

```
public class Client{
    public static void main(String[] args){
        Subject publisher = new Channel();

        Observer o1 = new BigFan("superman");
        Observer o2 = new BigFan("batman");
        publisher.registerObserver(o1);
        publisher.registerObserver(o2);

        publisher.notifyObserver();
        publisher.removeObserver(o2);
        publisher.notifyObserver();
    }
}
```

참고 : 자바의 내장 옵서버 객체

- 옵서버 패턴을 직접 구현할 수도 있지만, 자바에서 제공하는 java.util.Observable(인터페이스)과 java.util.Observer(클래스)로 구현 없이 옵서버 구조 이용
- 하지만 자바는 단일 상속만 지원하기 때문에 Subject가 다른 클래스를 이미 상속받은 상태이면 내장 옵서버 객체 이용 불가능
- Java9부터는 Observable이 deprecated

옵서버 패턴 장점

- 느슨한 결합(Loose Coupling): 옵저버와 주체(Subject) 간의 결합이 느슨해져, 서로 독립적으로 변경이 가능함. 주체는 옵저버에 대한 구체적인 정보를 알 필요가 없고, 옵저버도 주체의 구현에 의존하지 않음
- 자동화된 업데이트: 주체의 상태가 변경될 때 자동으로 모든 옵저버에게 통지되므로, 수동으로 상태를 업데이트할 필요가 없음
- 유연성: 새로운 옵저버를 추가하거나 제거하는 것이 쉬움. 시스템의 확장성이 높아짐
- 다양한 옵저버: 여러 종류의 옵저버를 만들 수 있어, 다양한 방식으로 주체의 상태 변화를 처리할 수 있음

옵서버 패턴 단점

- 복잡성 증가: 옵저버 패턴을 구현하면 시스템의 복잡성이 증가할 수 있음. 특히 많은 옵저버가 있을 경우, 상태 변화에 대한 통지가 복잡해질 수 있음
- 성능 문제 : 옵저버가 많을 경우, 주체의 상태가 변경될 때 모든 옵저버에게 통지하는 과정에서 성능이 저하될 수 있음. 특히 실시간 시스템에서는 문제가 될 수 있음
- 순환 참조 : 주체와 옵저버 간의 관계가 잘못 설정되면 순환 참조가 발생할 수 있음. 이는 메모리 누수로 이어질 수 있음
- 상태 일관성 문제 : 옵저버가 여러 개일 경우, 각 옵저버가 주체의 상태를 다르게 해석할 수 있어 상태 일관성 문제가 발생할 수 있음