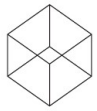
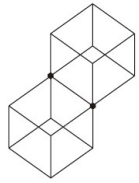
2. 객체지향 원리



JAVA 객체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



학습목표

학습목표

- 추상화 이해하기
- 캡슐화 이해하기
- 일반화(상속) 관계 이해하기
- 다형성 이해하기

2.1 추상화 (p. 60)

❖ 어떤 영역에서 필요로 하는 속성이나 행위를 추출하는 작업

- 관심 있는 부분에 더욱 집중할 수 있다

"추상화는 사물들의 공통된 특징, 즉 추상적 특징을 파악해 인식의 대상으로 삼는 행위다. 추상화가 가능한 개체들은 개체가 소유한 특성의 이름으로 하나의 집합class을 이룬다. 그러므로 추상화한다는 것은 여러 개체들을 집합으로 파악한다는 것과 같다. 추상적 특성은 집합을 구성하는 개체들을 '일반화' 하는 것이므로 집합의 요소들에 보편적인 것이다."

그림 2-1 추상화

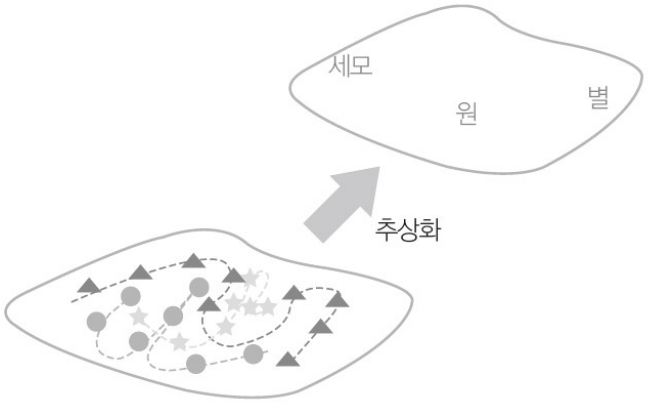
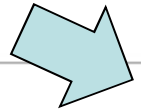



그림 2-2 추상화의 추상화
자동차
추상화
아우디
벤츠

❖ 추상화 개념을 사용하지 않는 경우에는 각각의 개체를 구분

코드 2-1

```
switch(자동차 종류)
  case 아우디:    // 아우디 엔진 오일을 교환하는 과정을 기술
  case 벤츠:      // 벤츠 엔진 오일을 교환하는 과정을 기술
end switch
```

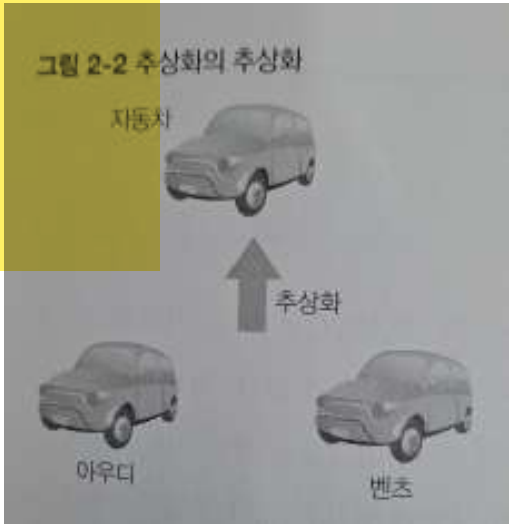


코드 2-2

```
switch(자동차 종류)
  case 아우디:    // 아우디 엔진 오일을 교환하는 과정을 기술
  case 벤츠:      // 벤츠 엔진 오일을 교환하는 과정을 기술
  case BMW:       // BMW 엔진 오일을 교환하는 과정을 기술
end switch
```

❖ 추상화 개념 도입 (코드 2-3)

- 구체적인 자동차 종류와 연관된 부분이 없음
- 새로운 자동차가 추가 되더라도 변경할 필요가 없음
- 폴리모피즘
- 코드를 변경할 필요가 없음(Reuse)
- Car c : Parent Type Car



코드 2-3

```
void changeEngineOil(Car c) {
  c.changeEngineOil();
}
```

2.2 캡슐화(encapsulation) (p. 63)

- 요구사항의 변경
  - 소프트웨어 개발의 골치거리
  - 해결책은 요구사항 변경을 당연한 것으로 받아들이고 이에 대비하는 것.
  - 캡슐화를 통해 높은 응집도와 낮은 결합도를 갖는 설계
- 응집도 (cohesion)
  - 클래스나 모듈 안의 요소들이 얼마나 밀접하게 관련되어 있는가?
- 결합도 (coupling)
  - 어떤 기능을 실행하는데 다른 클래스나 모듈들에 얼마나 의존적인가?
- 요구사항 변경할 때 유연하게 대처
  - 높은 응집도
  - 낮은 결합도
- 캡슐화는 낮은 결합도를 유지하게 해주는 객체지향 원리

❖ 정보은닉 (information hiding)

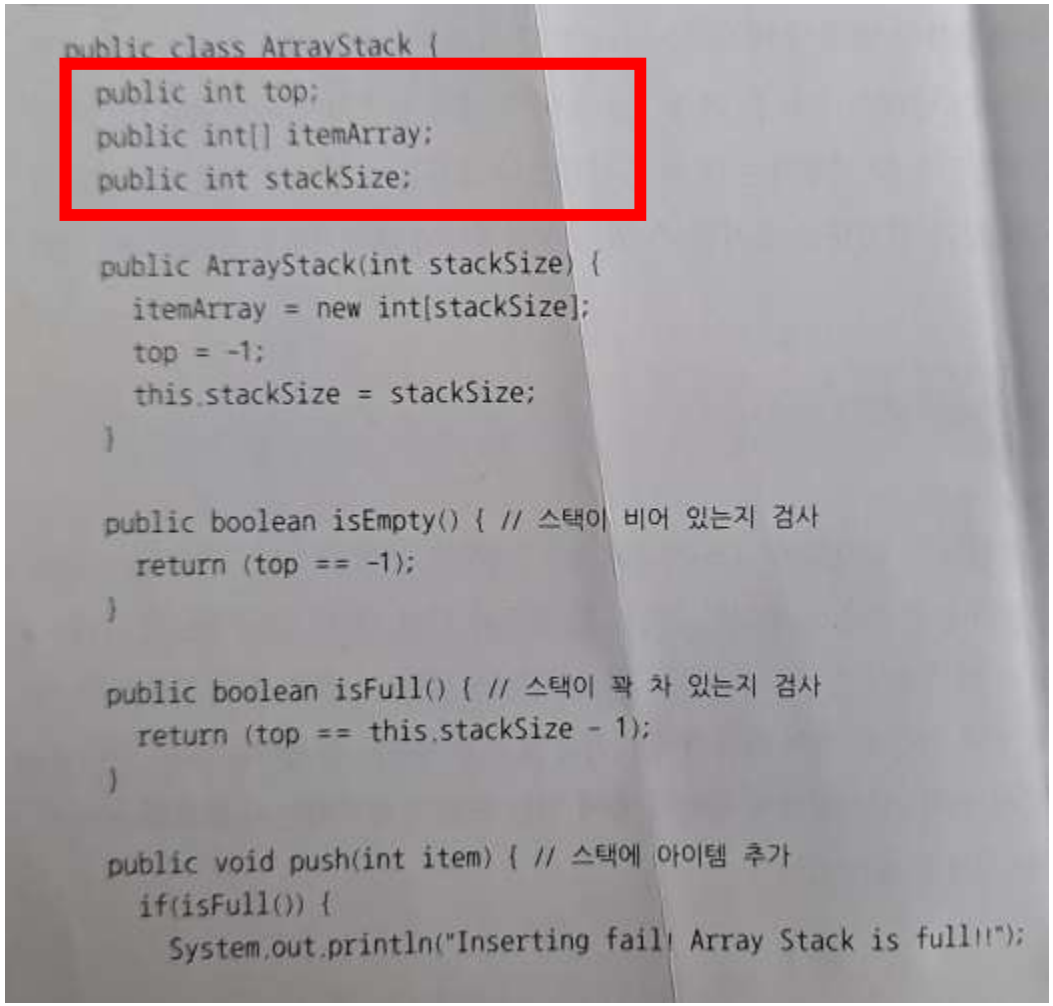
- 높은 응집도와 낮은 결합도를 갖게 함
- 알 필요가 없는 정보는 외부에서 접근하지 못하게 제한하는 것
- 캡슐화
- 예) 가속 페달을 밟았을때 어떤 원리로 가속되는지 알 필요가 없음

❖ 정보은닉의 필요성

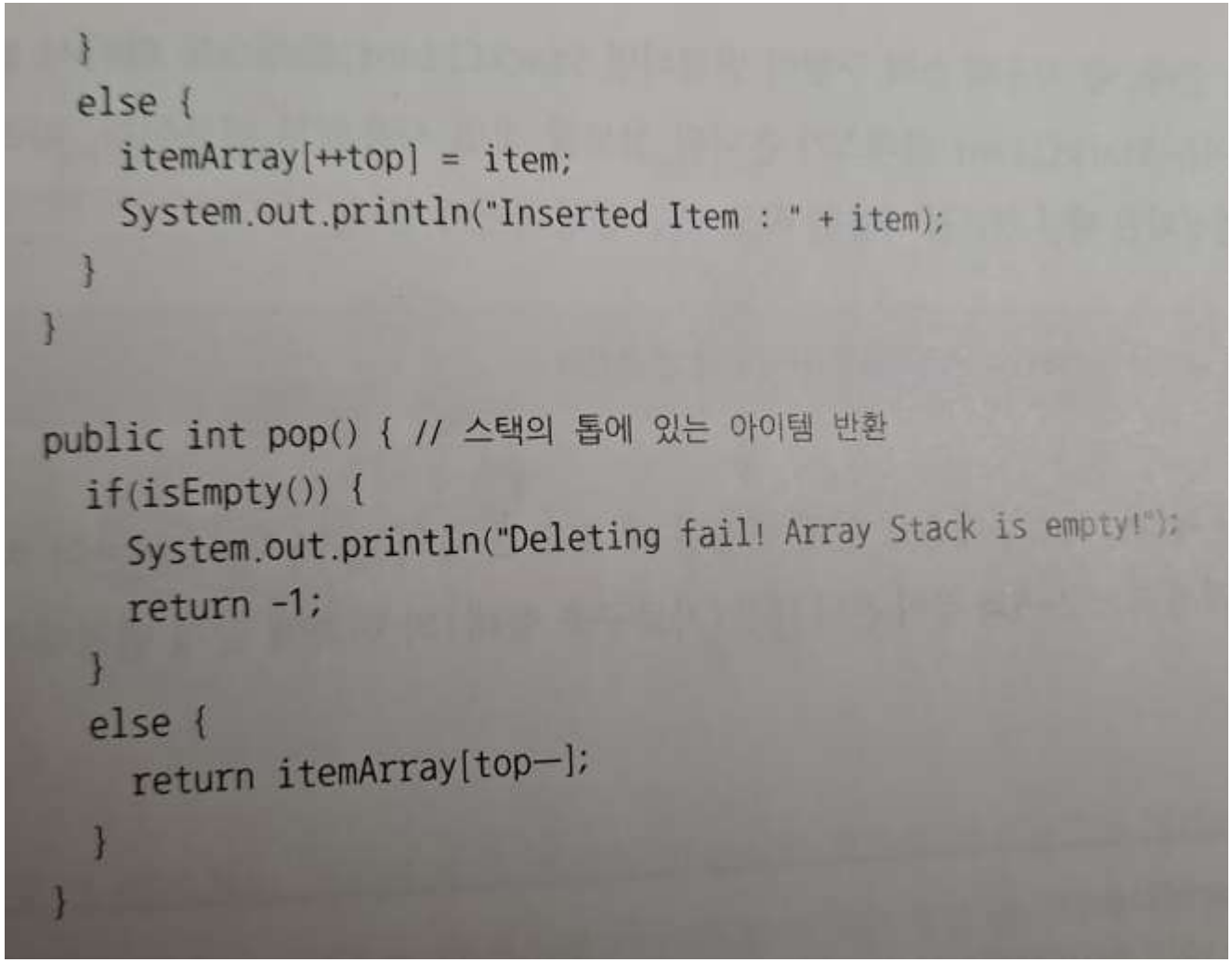
- 결합이 많을수록 문제가 많이 발생
- 한 클래스가 변경되면 변경된 클래스와 연관된 다른 클래스도 변경해야함

그림 2-3 정보 은닉

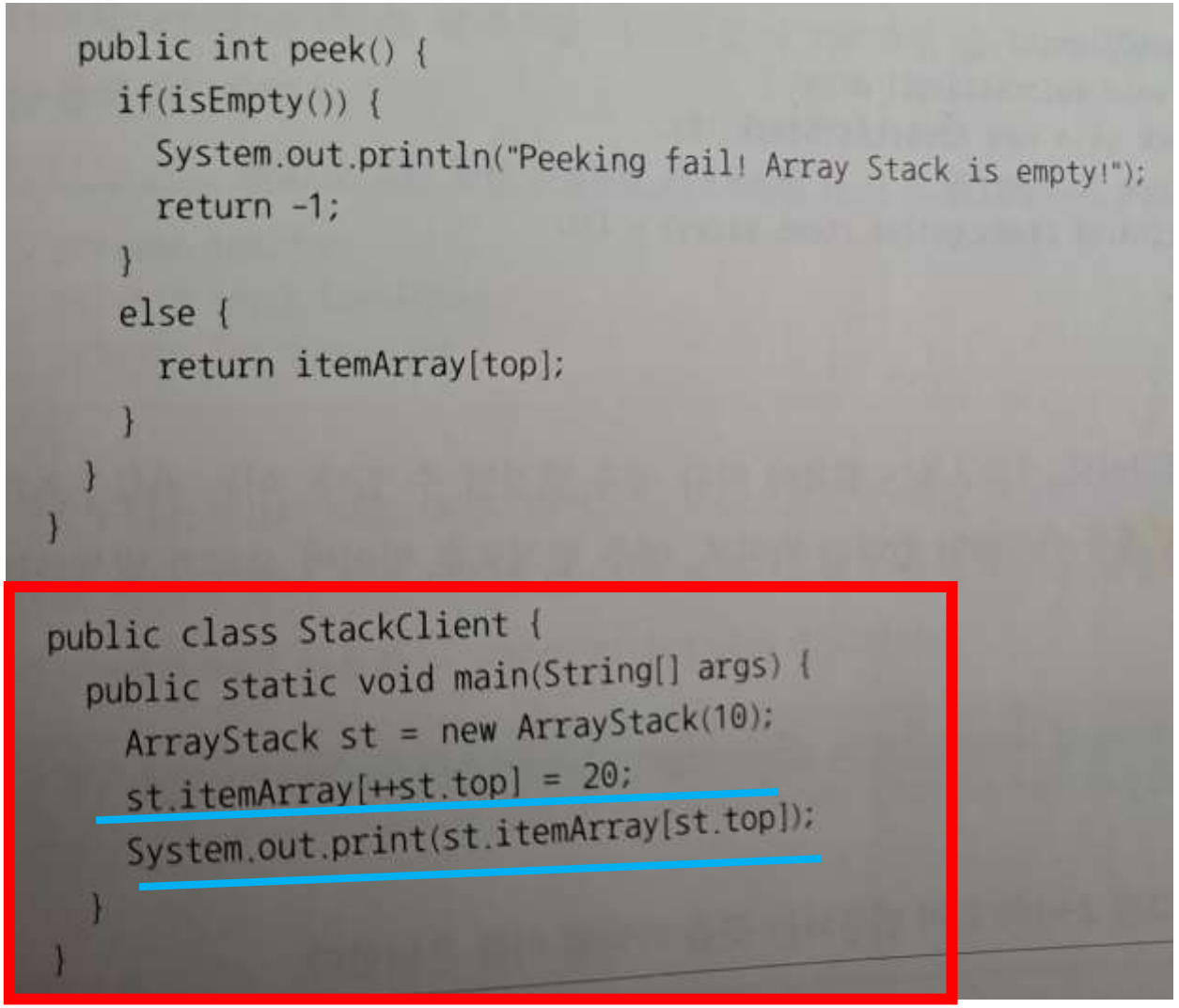
❖ 코드 2-4 리뷰



```
public class ArrayStack {
  public int top;
  public int[] itemArray;
  public int stackSize;

  public ArrayStack(int stackSize) {
    itemArray = new int[stackSize];
    top = -1;
    this.stackSize = stackSize;
  }

  public boolean isEmpty() { // 스택이 비어 있는지 검사
    return (top == -1);
  }

  public boolean isFull() { // 스택이 꽉 차 있는지 검사
    return (top == this.stackSize - 1);
  }

  public void push(int item) { // 스택에 아이템 추가
    if(isFull()) {
      System.out.println("Inserting fail! Array Stack is full!!");
```



```
  }
  else {
    itemArray[++top] = item;
    System.out.println("Inserted Item : " + item);
  }
}

public int pop() { // 스택의 톱에 있는 아이템 반환
  if(isEmpty()) {
    System.out.println("Deleting fail! Array Stack is empty!");
    return -1;
  }
  else {
    return itemArray[top--];
  }
}
```



```
  public int peek() {
    if(isEmpty()) {
      System.out.println("Peeking fail! Array Stack is empty!");
      return -1;
    }
    else {
      return itemArray[top];
    }
  }
}
```

```
public class StackClient {
  public static void main(String[] args) {
    ArrayStack st = new ArrayStack(10);
    st.itemArray[++st.top] = 20;
    System.out.print(st.itemArray[st.top]);
  }
}
```

❖ 코드 2-4. ArrayStack 클래스 평가

- 배열을 사용해 스택을 구현
- 배열이 모두 public 으로 외부에 공개
- Public method인 Push 나 pop 를 사용하지 않고도 직접 배열에 값을 저장
- ArrayStack과 StackClient 는 강한 결합

❖ 체크포인트. 코드 2-4를 ArrayList 클래스를 사용하여 변경하라

- (➔ p. 89)
- 어떻게 달라지는지 확인할 것

```
import java.util.ArrayList;

public class ArrayListStack {
  public int stackSize;
  public ArrayList<Integer> items; // 자료구조가 배열에서 ArrayList로 변경됨

  public ArrayListStack(int stackSize) {
    items = new ArrayList<Integer>(stackSize);
    this.stackSize = stackSize;
  }
```

❖ 코드 2-4에서의 StackClient

```
public class StackClient {
  public static void main(String[] args) {
    ArrayStack st = new ArrayStack(10);
    st.itemArray[++st.top] = 20;
    System.out.print(st.itemArray[st.top]);
  }
}
```

❖ 코드 2-4를 ArrayList를 사용하여 변경한 경우의 StackClient

❖

코드 2-5

```
public class StackClient {
  public static void main(String[] args) {
    ArrayListStack st = new ArrayListStack(10);
    st.items.add(new Integer(20)); // 위 체크포인트 변경점에 따라 바뀔 수 있음
    System.out.print(st.items.get(st.items.size() - 1));
  }
}
```

❖ 코드 2-5 평가

- 자료구조가 달라질때마다 StackClient도 달라져야 함
- 자료구조가 달라질 때마다 코드를 계속 변경해야함
- 오류가 발생하는 원인을 제공

그림 2-4 정보 은닉의 장점

캡슐화

코드 2.4

```
int top;
int[] itemArray;
int stackSize;
```

코드 2-6

```
public class StackClient {
  public static void main(String[] args) {
    ArrayListStack st = new ArrayListStack(10);
    st.push(20);
    System.out.print(st.peek());
  }
}
```

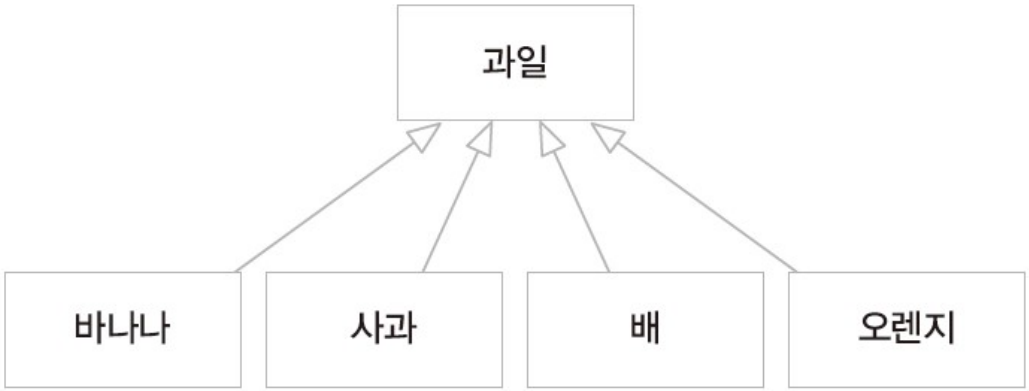
❖ 체크 포인트. 코드 2-4가 코드 2-6과 같이 ArryList 클래스를 이용할 때 main 메소드를 작성하라

- ==> page 90

2.3 일반화 (generalization) 관계 (p. 68)

- 일반화(상속)을 속성이나 기능의 재사용 관점에서만 보는 것은 극히 제한된 관점
- 철학에서 일반화(generalization)는 "여러 개체들이 가진 공통된 특성을 부각시켜 하나의 개념이나 법칙으로 성립시키는 과정"
  - 반대말. 특수화 (specialization)

그림 2-5 일반화 관계



일반화 개념을 사용하지 않았을 때

코드 2-7

```
가격 총합 = 0
while(장바구니에 과일이 있다) {
  switch(과일 종류)
    case 사과:
      가격 총합 = 가격 총합 + 사과 가격
    case 배:
      가격 총합 = 가격 총합 + 배 가격
    case 바나나:
      가격 총합 = 가격 총합 + 바나나 가격
    case 오렌지:
      가격 총합 = 가격 총합 + 오렌지 가격
}
```

장바구니에 있는 과일의 총합을 구하는 함수

키위가 추가된 경우

```
가격 총합 = 0
while(장바구니에 과일이 있다) {
        switch(과일 종류)
        case 사과:
                가격 총합 = 가격 총합 + 사과 가격
        case 배:
                가격 총합 = 가격 총합 + 배 가격
        case 바나나:
                가격 총합 = 가격 총합 + 바나나 가격
        case 오렌지:
                가격 총합 = 가격 총합 + 오렌지 가격
        case 키위:
                가격 총합 = 가격 총합 + 키위 가격
}
```

❖ 새로운 과일의 종류가 추가되더라도 코드를 수정할 필요가 없도록 수정

- calculatePrice()
  - 실제 과일의 객체(curFruit)에 따라 서로 다르게 실행
  - 다형성 (polymorphism)

코드 2-9

```
int computeTotalPrice(LinkedList<Fruit> f) {
  int total = 0;
  Iterator<Fruit> itr = f.iterator();

  while(itr.hasNext()) {
    Fruit curFruit = itr.next();
    total = total + curFruit.calculatePrice();
  }
  return total;
}
```

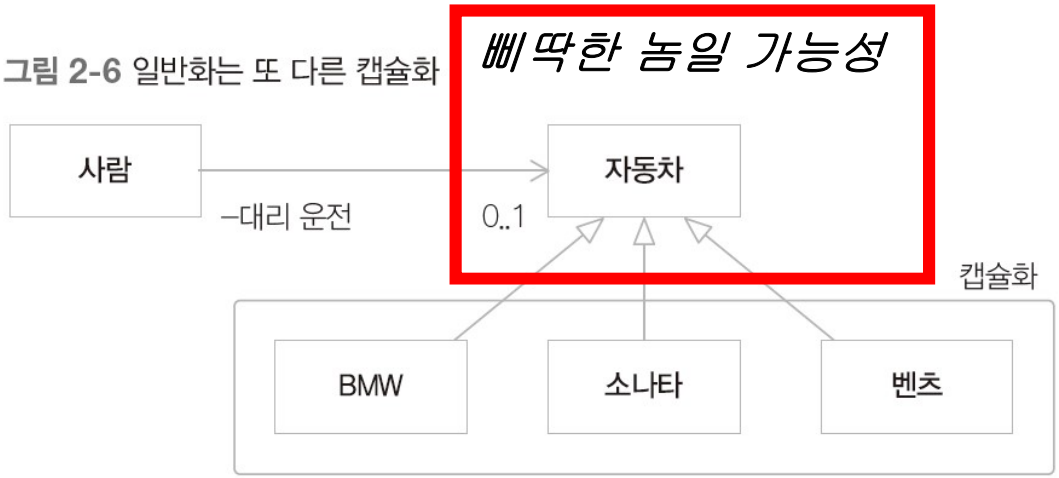
그림 2-5 일반화 관계

- Is a relation **이 적용되는 곳**??

❖ **일반화는 클래스 자체를 캡슐화하여 변경에 대비할 수 있는 설계를 가능하게 한다**

- **새로운 클래스가 추가되더라도 클라이언트는 영향 받지 않음**

그림 2-6 일반화는 또 다른 캡슐화



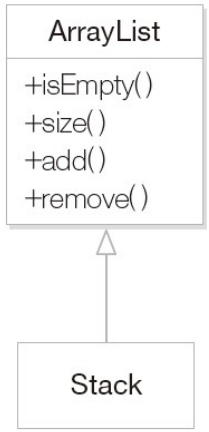
Keypoint_ 일반화 관계는 자식 클래스를 외부로부터 은닉하는 캡슐화의 일종이다.

2.3.2 일반화와 위임 (p. 71)

❖ 그림 2-7의 Stack의 구현 (ArrayList 를 상속받아…)

- ArrayList 클래스에 정의된 메소드들을 그대로 사용하기를 원함
- 그러나 Stack에서 불필요한, ArrayList의 다양한 메소드, 속성들도 같이 상속

그림 2-7 ArrayList를 이용한 Stack의 구현



❖ 코드 2-10

- push나 pop 메소드를 통하지 않고 , 스택의 자료구조에 직접 접근할 수 있음
- 스택의 무결성 조건인 LIFO에 위배

코드 2-10

```
class MyStack<String> extends ArrayList<String> {
  public void push(String element) {
    add(element);
  }

  public String pop() {
    return remove(size() - 1);
  }
}
```

❖ 체크포인트.

- 코드 2-10을 이용해 만든 스택이 스택의 무결성에 위배되도록 main method를 작성하라 (==> p. 92)

코드 2-10을 이용해 만든 스택이 스택의 무결성 조건에 위배되도록 main 메서드를 작성하라.

```
public class Main {
  public static void main(String[] args) {
    MyStack<String> st = new MyStack<String>();

    st.push("insang1");
    st.push("insang2");
    st.set(0, "insang3"); // 허용되어서는 안됨
    System.out.println(st.pop());
    System.out.println(st.pop());
  }
}
```

❖ ‘is a kind of 관계’

- 기본적으로 일반화 관계는 ‘is a kind of 관계’가 성립되어야 함
- ‘is a kind of 관계’를 확인하는 방법 : 다음문장이 참인가?
  - Stack ‘is a kind of ’ ArrayList
  - ➔ False

두 자식 클래스 사이에 ‘is a kind of 관계’가 성립되지 않을 때 상속을 사용하면 불필요한 속성이나 연산(빚이라 해도 될 것이다)도 물려받게 된다.

일반화와 위임

- 어떤 클래스의 일부 기능만을 사용하고 싶을 경우에는 위임을 사용하라
- 일반화를 위임으로 변환하는 프로세스
  1. 자식 클래스에 부모 클래스의 인스턴스를 참조하는 속성을 만든다. 이 속성 필드를 this로 초기화한다.
  2. 서브 클래스에 정의된 각 메서드에 1번에서 만든 위임 속성 필드를 참조하도록 변경한다.
  3. 서브 클래스에서 일반화 관계 선언을 제거하고 위임 속성 필드에 슈퍼 클래스의 객체를 생성해 대입한다.
  4. 서브 클래스에서 사용된 슈퍼 클래스의 메서드에도 위임 메서드를 추가한다.
  5. 컴파일하고 잘 동작하는지 확인한다.

단계 1

❖ 코드 2-11

- MyClass 클래스에 ArrayList 클래스의 인스턴스를 참조하는 속성인 arList 객체를 만든 후, 이속성필드를 this로 초기화

코드 2-10

```
class MyStack<String> extends ArrayList<String> {
  public void push(String element) {
    add(element);
  }

  public String pop() {
    return remove(size() - 1);
  }
}
```

코드 2-11

```
public class MyStack<String> extends ArrayList<String>
  private ArrayList<String> arList = this;

  public void push(String element) {
    add(element);
  }

  public String pop() {
    return remove(size() - 1);
  }
}
```

단계 2

❖ 코드 2-12

- MyStack 클래스의 push와 pop 메소드에 arList 객체를 참조하도록 변경

코드 2-12

```
public class MyStack<String> extends ArrayList<String> {
  private ArrayList<String> arList = this;

  public void push(String element) {
    arList.add(element);
  }

  public String pop() {
    return arList.remove(arList.size() - 1);
  }
}
```

단계 3

❖ 코드 2-13

- ArrayList와 MyStack 클래스 사이의 일반화 관계를 제거하고 arList를 ArrayList 객체로 생성해 초기화

코드 2-13

```
public class MyStack<String> {
  private ArrayList<String> arList = new ArrayList<String>();

  public void push(String element) {
    arList.add(element);
  }

  public String pop() {
    return arList.remove(arList.size() - 1);
  }
}
```

단계 4

❖ 코드 2-14

- MyStack 클래서에서 사용된 arList 객체의 isEmpty와 size 메서드에 위임 메서드를 서브 클래스에 추가

코드 2-14

```
public class MyStackDelegation<String> {
  private ArrayList<String> arList = new ArrayList<String>();

  public void push(String element) {
    arList.add(element);

    }

    public String pop() {
      return arList.remove(arList.size() - 1);
    }

    public boolean isEmpty() {
      return arList.isEmpty();
    }

    public int size() {
      return arList.size();
    }
  }
```

체크포인트

❖ Vector 클래스를 사용(위임)해 Stack 클래스를 구현하라(p. 75).

- p. 92

```
import java.util.Vector;

public class VectorStack {
  private Vector<String> v = new Vector<String>();

  public void push(String element) {
    v.add(element);
  }

  public String pop() {
    return v.remove(v.size() - 1);
  }

  public boolean isEmpty() {
    return v.isEmpty();
  }

  public int size() {
    return v.size();
  }
}
```