2-1 : 과제 피드백 및 1주차 내용 토론

들어가기 전에

이번 시간에는...

과제 피드백 & QnA
객체지향언어
자바 컬렉션 활용법

과제 피드백

자바 백준 풀이, 요구사항 개발 및 Git 활용

토론 키워드

객체지향언어

객체지향언어

객체지향언어 배경

“실제 세계는 사물(객체)로 이루어져 있으며, 발생하는 모든 사건들은 사물 간의 상호작용이다.”

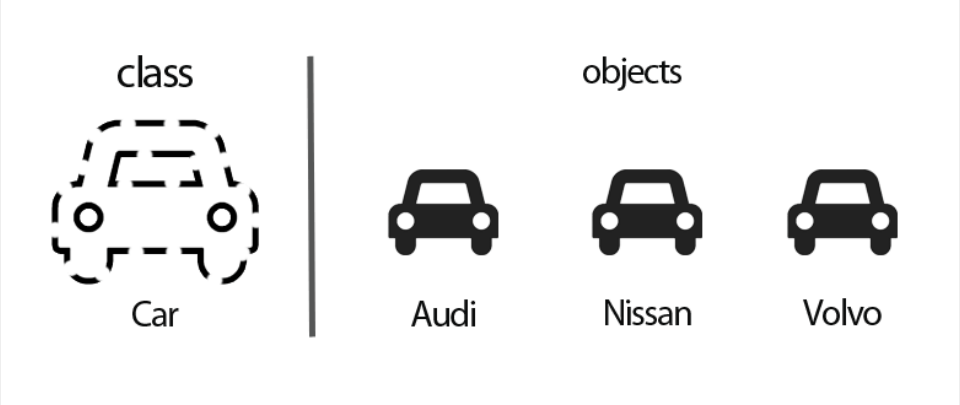
실제 사물의 속성과 기능을 분석하고, 변수와 함수로 정의하여
실제 세계를 컴퓨터 속에 옮겨 놓은 것과 같은 가상 세계를 구현.

객체지향언어

객체지향언어 개념

프로그래밍 기법 중 하나

실제 사물의 속성과 기능을 분석하고, 변수와 함수로 정의하여
실제 세계를 컴퓨터 속에 옮겨 놓은 것과 같은 가상 세계를 구현.



설계도(Class)인 Car를 기반으로 Audi, Nissan, Volvo(objects)들이 만들어진다.

객체지향언어

자동차 공장 예시

우리는 자동차 공장 사장님
자동차를 생산하는 방법이 2가지

개별적으로 직접 개발	설계도 작성 후 참고하여 개발
- 대량 생산 시, 생산 시간이 비교적 많이 듦 - 문제 발생 시, 다른 자동차도 문제가 없는지 모두 검사, 수정 - 각 자동차 마다 품질, 성능 차이가 남	- 대량 생산 시, 생산 시간이 비교적 적게 듦 - 문제 발생 시, 설계도만 검사, 수정 - 각 자동차 마다 품질, 성능 차이가 없음

객체지향언어

객체지향언어 개념, 주요 특징

1. 코드의 재사용성이 높다.
2. 코드의 관리가 용이하다.
3. 신뢰성이 높은 프로그래밍을 가능하게 한다.

개별적으로 직접 개발	설계도 작성 후 참고하여 개발
- 대량 생산 시, 생산 시간이 비교적 많이 듦	- 대량 생산 시, 생산 시간이 비교적 적게 듦
- 문제 발생 시, 다른 자동차도 문제가 없는지 모두 검사, 수정	- 문제 발생 시, 설계도만 검사, 수정
- 각 자동차 마다 품질, 성능 차이가 남	- 각 자동차 마다 품질, 성능 차이가 없음

토론 키워드

캡슐화

캡슐화

인스턴스 접근

인스턴스의 각 변수, 메소드는 .(점)을 통해 접근할 수 있다.

클래스

```
// 클래스 선언
class TV {

  // 필드
  String color;
  boolean power;
  int channel;

  // 생성자
  Tv(String color, boolean power, int channel) {
    this.color = color;
    this.power = power;
    this.channel = channel;
  }

  // 메서드
  void power() { power = !power; }
  void channelUp() { ++channel; }
  void channelDown() { --channel; }
}
```

인스턴스

```
public class TvExam{
    public static void main(String args[]){
        Tv yellowTv = new Tv("yellow", true, 1);

        String color1 = yellowTv.color; // yellow
        yellowTv.channelUp(); // yellowTv의 channel 값이 2로 변경
    }
}
```

캡슐화

캡슐화

인스턴스의 각 변수, 메소드는 .(점)을 통해 접근할 수 있다.
하지만 **외부에서 직접적으로 접근하지 못하게 한다**.

```
// 클래스 선언
class TV {

  // 필드
  String color;
  boolean power;
  int channel;

  // 생성자
  Tv(String color, boolean power, int channel) {
    this.color = color;
    this.power = power;
    this.channel = channel;
  }

  // 메서드
  void power() { power = !power; }
  void channelUp() { ++channel; }
  void channelDown() { --channel; }
}
```

```
// 필드
private String color;
private boolean power;
private int channel;
```

+

```
// getter, setter
public String getColor() {
    return color;
}

public void setColor(String color) {
    this.color = color;
}

public boolean isPower() {
    return power;
}

public void setPower(boolean power) {
    this.power = power;
}

public int getChannel() {
    return channel;
}

public void setChannel(int channel) {
    this.channel = channel;
}
```

캡슐화

캡슐화

.(점) 대신에 set, get 메소드를 통해 접근할 수 있다.

캡슐화 전 인스턴스 사용

```
public class TvExam{
    public static void main(String args[]){
        Tv yellowTv = new Tv("yellow", true, 1);

        String color1 = yellowTv.color; // yellow
        yellowTv.channelUp(); // yellowTv의 channel 값이 2로 변경
    }
}
```

캡슐화 후 인스턴스 사용

```
// 캡슐화 후
public class TvExam{
    public static void main(String args[]){
        Tv yellowTv = new Tv("yellow", true, 1);

        yellowTv.setChannel(10);
        String color1 = yellowTv.getColor(); // yellow
        yellowTv.channelUp(); // yellowTv의 channel 값이 11로 변경
    }
}
```

캡슐화

캡슐화를 사용하는 이유?

1. 객체의 속성 값이 예기치 못하게 변경되는 것을 막을 수 있다.
속성 값의 변화를 외부에서 할 수 있게 원하는 객체에만 setter를 정의해둔다.
외부에서 바꾸면 문제가 되는 값이라면 setter를 구현하지 않는다.

클래스

```
// getter, setter
public String getColor() {
    return color;
}

public void setColor(String color) {
    this.color = color;
}

public boolean isPower() {
    return power;
}

public void setPower(boolean power) {
    this.power = power;
}

public int getChannel() {
    return channel;
}

public void setChannel(int channel) {
    this.channel = channel;
}
```

인스턴스 활용부

```
// 캡슐화 후
public class TvExam{
    public static void main(String args[]){
       Tv yellowTv = new Tv("yellow", true, 1);

      yellowTv.setChannel(10);
      String color1 = yellowTv.getColor(); // yellow
      yellowTv.channelUp(); // yellowTv의 channel 값이 11로 변경
    }
}
```

캡슐화

캡슐화를 사용하는 이유?

2. getter, setter 메소드를 통해 값을 설정하고 받아올 때 로직을 추가할 수 있다.
ex) TV color를 받아올 때, 매 번 “TV 컬러는”이라는 문구를 추가하고 싶다면?

클래스

```
public String getColor() {
    return color;
}
```

```
public String getColor() {
    return "TV 컬러는 " + color;
}
```

인스턴스 활용부

```
yellowTv.setChannel(10);
String color1 = yellowTv.getColor(); // yellow
yellowTv.channelUp(); // yellowTv의 channel 값이 11로 변경
```

```
yellowTv.setChannel(10);
String color1 = yellowTv.getColor(); // TV 컬러는 yellow
yellowTv.channelUp(); // yellowTv의 channel 값이 11로 변경
```

토론 키워드

상속

상속

상속

기존의 클래스를 재사용하여 새로운 클래스를 작성하는 것
각 객체 마다 부모로부터 공통된 특징을 상속하여 자식 객체를 만들면 편리

동물

강아지 고양이

사자 호랑이

가구

침대 소파

책상 의자

자동차

경찰차 소방차

구급차 버스

적은 양의 코드로 새로운 클래스를 작성할 수 있다.
코드를 공통적으로 관리할 수 있다.(관리 용이)
생산성과 유지보수에 크게 기여

상속

상속

부모 클래스(상위 클래스): 상속해주는 클래스

자식 클래스(하위 클래스): 상속을 받는 클래스

```java
// 부모 클래스
public class Car{

  private String modelName;
  private int modelYear;

  Car(String modelName, int modelYear) {
      this.modelName = modelName;
      this.modelYear = modelYear;
  }

  public void drive() {
      System.out.println("출발합니다.");
  }
  public void stop() {
      System.out.println("정지합니다.");
  }

}
```

```java
// 자식 클래스
public class PoliceCar extends Car{

  private String siren;

  public void soundSiren() {
      System.out.println("사이렌을 울립니다.");
  }
}
```

상속

자바 상속 특징

단일 상속

자바에서는 하나의 부모만을 상속받는 단일 상속만을 허용
다중 상속에 비해 불편한 점도 있지만, 클래스 간의 관계가 좀 더 명확해지고 코드를 더욱 신뢰할 수 있다.

Object 클래스

자바의 모든 클래스의 최상위 부모 클래스는 Object 클래스
자바의 모든 클래스는 Object 클래스를 상속받고 있다.
Object 클래스에는 모든 인스턴스가 가져야 할 기본적인 11개의 메소드가 정의되어 있다.
Object 클래스 만큼은 단일 상속 조건에 예외

오버라이딩

오버라이딩

부모 클래스로부터 상속받은 메소드의 내용을 변경하는 것
상속 받은 메소드를 자식 클래스에 맞게 변경해야 하는 경우에 사용

```java
// 부모 클래스
public class Car{

  public void drive() {
      System.out.println("출발합니다.");
  }
  public void stop() {
      System.out.println("정지합니다.");
  }

}
```

```java
// 자식 클래스
public class PoliceCar extends Car{

  public void drive() {
      System.out.println("경찰차 출발합니다.");
  }
  public void stop() {
      System.out.println("경찰차 정지합니다.");
  }
}
```

```java
public class OverRidingExam{
    public static void main(String args[]){
        PoliceCar policeCar = new PoliceCar();
        policeCar.drive();  //PoliceCar drive 메소드가 실행된다.
    }
}
```

오버라이딩

오버라이딩

부모 클래스로부터 상속받은 메소드의 내용을 변경하는 것
상속 받은 메소드를 자식 클래스에 맞게 변경해야 하는 경우에 사용

```java
// 부모 클래스
public class Car{

  public void drive() {
      System.out.println("출발합니다.");
  }
  public void stop() {
      System.out.println("정지합니다.");
  }

}
```

```java
// 자식 클래스
public class PoliceCar extends Car{

  public void drive() {
      System.out.println("경찰차 출발합니다.");
  }
  public void stop() {
      System.out.println("경찰차 정지합니다.");
  }
}
```

오버라이딩의 조건

1. 메소드의 이름이 같아야 한다.
2. 메소드의 매개변수가 같아야 한다.
3. 메소드의 반환 타입이 같아야 한다.

오버로딩 vs 오버라이딩

오버로딩

같은 이름으로 기존에 없는 **새로운 메소드를 정의**하는 것

오버라이딩

같은 이름으로 **상속받은 메소드의 내용을 변경**하는 것

토론 키워드

다형성

다형성(형변환)

다형성(형변환)

부모 타입의 참조 변수로 자식 타입의 객체를 다루는 것
상속 관계에 있을 때, 객체들 끼리 형변환이 가능하다.

```
public class Exam{
    public static void main(String args[]){
        Car car = new PoliceCar(); // 자식타입을 부모타입으로 묵시적 형변환
        car.run();
        //car.soundSiren(); // 컴파일 오류 발생

        PoliceCar policeCar = (PoliceCar)car;  //부모타입을 자식타입으로 명시적 형변환
        policeCar.drive();
        policeCar.soundSiren();
    }
}
```

다형성(형변환)

다형성(형변환)

부모 타입의 참조 변수로 자식 타입의 객체를 다루는 것
상속 관계에 있을 때, 객체들 끼리 형변환이 가능하다.

```
// 부모 클래스
public class Car{
  public void drive() {
      System.out.println("출발합니다.");
  }
}

// 자식 클래스
public class PoliceCar extends Car{
  public void soundSiren() {
      System.out.println("사이렌을 울립니다.");
  }
}
```

자식타입 -> 부모타입: 형변환 생략 가능.
부모타입 -> 자식타입: 형변환 생략 불가능.

```
public class Exam{
    public static void main(String args[]){
        Car car = new PoliceCar(); // 자식타입을 부모타입으로 묵시적 형변환
        car.run();
        //car.soundSiren(); // 컴파일 오류 발생

        PoliceCar policeCar = (PoliceCar)car;  //부모타입을 자식타입으로 명시적 형변환
        policeCar.drive();
        policeCar.soundSiren();
    }
}
```

항상 변수 선언하는 쪽(왼쪽)이 기준

다형성(형변환)

다형성 예제(형변환 사용X)

```java
class Animal{

  public void move() {
    System.out.println("동물이 움직입니다.");
  }

  public void eating() {

  }
}

class Human extends Animal{
  public void move() {
    System.out.println("사람이 두발로 걷습니다.");
  }

  public void readBooks() {
    System.out.println("사람이 책을 읽습니다.");
  }
}

class Tiger extends Animal{

  public void move() {
    System.out.println("호랑이가 네 발로 뜁니다.");
  }

  public void hunting() {
    System.out.println("호랑이가 사냥을 합니다.");
  }
}


class Eagle extends Animal{
  public void move() {
    System.out.println("독수리가 두 발로 걸어갑니다");
  }

  public void flying() {
    System.out.println("독수리가 날개를 쭉 펴고 멀리 날아갑니다");
  }
}
```

```java
public class Main {
    public static void main(String[] args) {

        Human Animal1 = new Human();
        Tiger Animal2 = new Tiger();
        Eagle Animal3 = new Eagle();

        Main main = new Main();
        main.moveHuman(Animal1);
        main.moveTiger(Animal2);
        main.moveEagle(Animal3);
    }

    1 usage
    public void moveHuman(Human human) {
        human.move();
    }
    1 usage
    public void moveTiger(Tiger tiger) {
        tiger.move();
    }
    1 usage
    public void moveEagle(Eagle eagle) {
        eagle.move();
    }
}
```

```
사람이 두 발로 걷습니다.
호랑이가 네 발로 뜁니다.
독수리가 두 발로 걸어갑니다.

Process finished with exit code 0
```

다형성(형변환)

다형성 예제(형변환 사용)

```
class Animal{

  public void move() {
    System.out.println("동물이 움직입니다.");
  }

  public void eating() {

  }
}

class Human extends Animal{
  public void move() {
    System.out.println("사람이 두발로 걷습니다.");
  }

  public void readBooks() {
    System.out.println("사람이 책을 읽습니다.");
  }
}

class Tiger extends Animal{

  public void move() {
    System.out.println("호랑이가 네 발로 뜁니다.");
  }

  public void hunting() {
    System.out.println("호랑이가 사냥을 합니다.");
  }
}


class Eagle extends Animal{
  public void move() {
    System.out.println("독수리가 두 발로 걸어갑니다");
  }

  public void flying() {
    System.out.println("독수리가 날개를 쭉 펴고 멀리 날아갑니다");
  }
}
```

```
public class Main {
    public static void main(String[] args) {

        Animal hAnimal = new Human();
        Animal tAnimal = new Tiger();
        Animal eAnimal = new Eagle();

        Main main = new Main();
        main.moveAnimal(hAnimal);
        main.moveAnimal(tAnimal);
        main.moveAnimal(eAnimal);
    }
    3 usages
    public void moveAnimal(Animal animal) {
        animal.move();

    }
}
```

```
사람이 두 발로 걷습니다.
호랑이가 네 발로 뜁니다.
독수리가 두 발로 걸어갑니다.

Process finished with exit code 0
```

다형성(형변환)

다형성 예제

형변환 사용X

```java
public class Main {
    public static void main(String[] args) {

        Human Animal1 = new Human();
        Tiger Animal2 = new Tiger();
        Eagle Animal3 = new Eagle();

        Main main = new Main();
        main.moveHuman(Animal1);
        main.moveTiger(Animal2);
        main.moveEagle(Animal3);
    }

    1 usage
    public void moveHuman(Human human) {
        human.move();
    }
    1 usage
    public void moveTiger(Tiger tiger) {
        tiger.move();
    }
    1 usage
    public void moveEagle(Eagle eagle) {
        eagle.move();
    }
}
```

형변환 사용

```java
public class Main {
    public static void main(String[] args) {

        Animal hAnimal = new Human();
        Animal tAnimal = new Tiger();
        Animal eAnimal = new Eagle();

        Main main = new Main();
        main.moveAnimal(hAnimal);
        main.moveAnimal(tAnimal);
        main.moveAnimal(eAnimal);
    }
    3 usages
    public void moveAnimal(Animal animal) {
        animal.move();

    }
}
```

자식 클래스를 부모 클래스로 형변환 하여 활용하면 객체를 효율적으로 관리, 활용할 수 있다.

토론 키워드

추상화

추상 클래스

추상 클래스

미완성 메소드(추상 메소드)를 가지고 있는 클래스.
부분완성 클래스(미완성 클래스), 미완성 설계도에 비유할 수 있다

특징

추상 클래스와 메소드에는 **abstract 키워드**가 붙는다.
추상 클래스는 **자식 클래스에 의해서만 완성**될 수 있다.(추상 클래스 자체로 인스턴스 생성 불가)

목적

선언부를 독립시켜 유지보수에 효율(수정 시 컴파일 오류)
추상 클래스를 사용하는 이유는 자식 클래스에서 추상 메소드를 반드시 구현하도록 강제하기 위함.

```
public abstract class Bird{ // 추상 클래스
    public abstract void sing(); // 추상 메서드(미완성)

    public void fly(){ // 메서드
        System.out.println("날다.");
    }
}
```

인터페이스

인터페이스

추상 메소드의 집합으로 구현된 것이 전혀 없는 설계도

오직 **추상 메소드와 상수**만을 가질 수 있다.

인터페이스 구현 시, **implements 키워드**를 사용한다.

하나의 클래스가 **여러 개의 인터페이스를 구현**할 수 있다.

```java
// 인터페이스 선언
public interface Calculator {
    public int plus(int i, int j);
    public int multiple(int i, int j);
}
```

```java
// 인터페이스 구현
public class MyCalculator implements Calculator {
    @Override
    public int plus(int i, int j) {
        return i + j;
    }

    @Override
    public int multiple(int i, int j) {
        return i * j;
    }
  }
}
```

모든 멤버변수는 public, static, final 이어야 하며, 생략할 수 있다.
모든 메소드는 public, abstract 이어야 하며, 생략할 수 있다.
자바 8버전 부터는 default 메소드, static 메소드도 가질 수 있게 변경

인터페이스

인터페이스를 사용하는 이유

인터페이스는 두 객체 간의 연결을 돕는 중간 역할을 한다.

1. 개발 시간을 단축시킬 수 있다.

구현해야 할 기본 변수, 메소드를 미리 정의해 두어 효과적으로 개발 가능

2. 표준화가 가능

미리 정의해둔 변수, 메소드의 틀을 활용하여 개발하기 때문에 표준화가 가능
정형화된 프로그램을 개발할 수 있다.

3. 클래스 관계 매핑

서로 아무 관계가 없는 클래스들에게 공통적인 변수, 메소드를 묶어 관리하기 수월
다형성 특징을 그대로 활용할 수 있다.

4. 독립적인 프로그래밍

역할과 구현을 분리시켜 독립적인 프로그램을 작성할 수 있다.(호출부 수정X)

추상 클래스 vs 인터페이스

추상 클래스 vs 인터페이스

추상 클래스는 **부분 완성** 설계도
인터페이스는 **구현이 전혀 안된** 설계도

추상 클래스

```
public abstract class Bird{ // 추상 클래스
    public abstract void sing(); // 추상 메서드(미완성)

    public void fly(){ // 메서드
        System.out.println("날다.");
    }
}
```

인터페이스

```
// 인터페이스 선언
public interface Calculator {
    public int plus(int i, int j);
    public int multiple(int i, int j);
}
```

토론 키워드

Java 자료구조

Java.util 패키지

Java.util 패키지

날짜와 관련된 클래스인 Date, Calendar 클래스

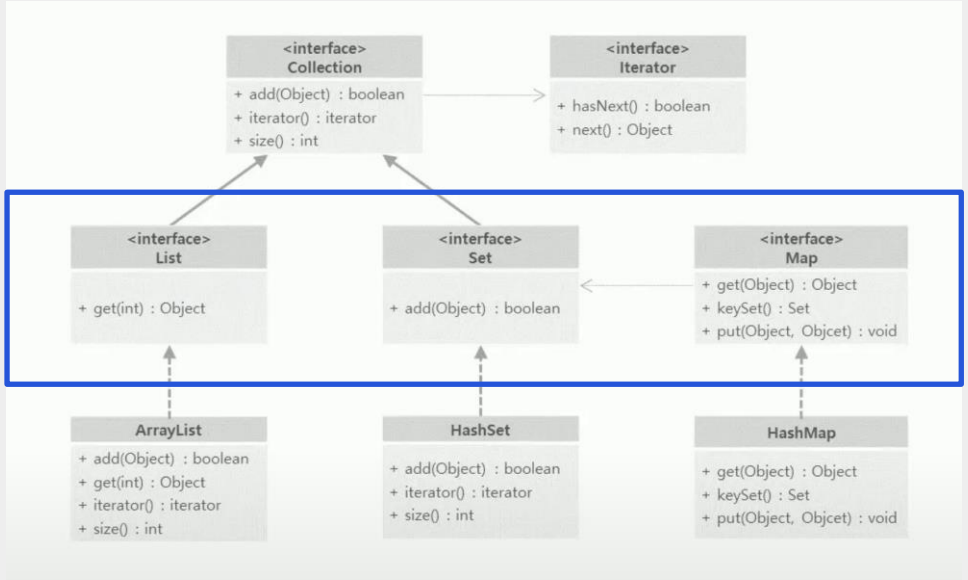
자료구조와 관련된 컬렉션 프레임워크와 관련된 인터페이스와 클래스

Java.util 패키지

컬렉션 프레임워크

데이터 군을 저장하는 **자료구조**와 관련된 클래스들을 표준화한 설계

컬렉션 프레임워크의 모든 컬렉션 클래스들은 **List, Set, Map** 중의 하나를 구현하고 있다.



Java.util 패키지

컬렉션 프레임워크

데이터 군을 저장하는 **자료구조**와 관련된 클래스들을 표준화한 설계
컬렉션 프레임워크의 모든 컬렉션 클래스들은 **List, Set, Map** 중의 하나를 구현하고 있다.

- **List: 순서가 있는 데이터. 중복을 허용한다.**
구현체 예시) ArrayList, LinkedList, Stack, Vector 등

- **Set: 순서가 유지되지 않으며, 중복을 허용하지 않는다.**
구현체 예시) HashSet, TreeSet 등

- **Map: 키와 값의 쌍으로 구성된 자료구조. 순서 유지X, 키 중복X, 값은 중복이 가능하다.**
구현체 예시) HashMap, TreeMap, HashTable, Properties 등

Java.util 패키지

컬렉션 프레임워크 사용 예제

상황 예시

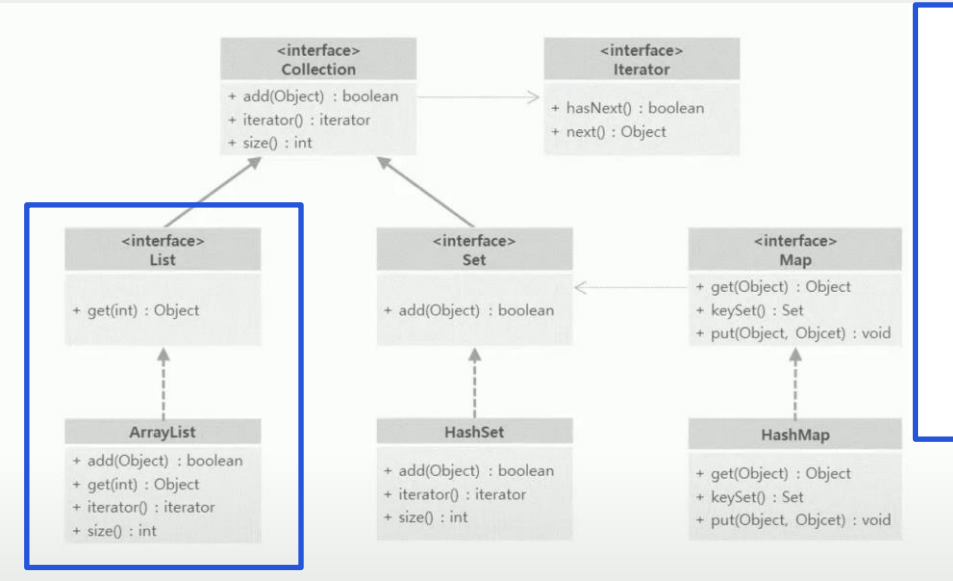
클라이언트가 **다수의** 학생 데이터 **성적** 주면,
받은 학생 성적 데이터 **순서대로** 학점을 계산하여 응답해주는 상황

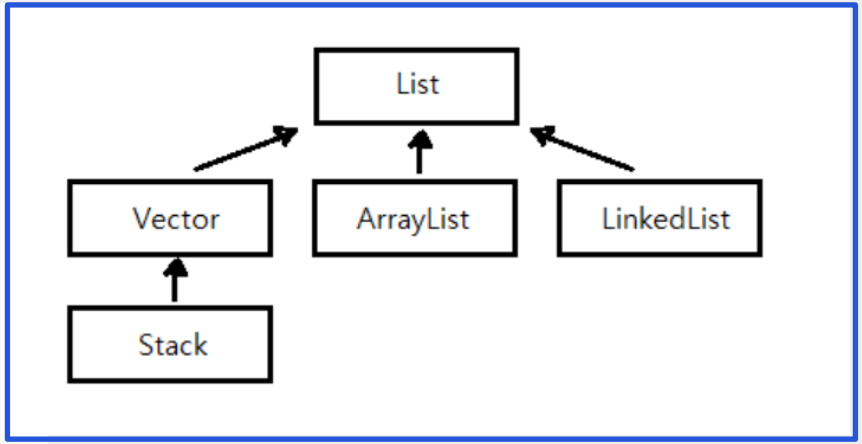
순서대로	자료구조 내에서 저장, 삭제 순서가 보장	Array(배열), List(리스트)
성적	성적은 중복이 될 수 있음	
다수의	여러 개의 데이터를 다룰 수 있어야 함 몇 개의 데이터를 다룰지 정해지지 않음	**List(리스트)**

Java.util 패키지

컬렉션 프레임워크 사용 예제

컬렉션 프레임워크에 List는 다음과 같다.





- Vector: ArrayList로 대체
- Stack: 다른 개념의 자료구조
- ArrayList: 데이터 생성, 삭제에 약함, 조회에 강한 리스트
- LinkedList: 데이터 생성, 삭제에 강함, 조회에 약한 리스트

Java.util 패키지

컬렉션 프레임워크 사용 예제

선택한 자료구조(ArrayList)를 생성해서 사용한다.

```java
public class Main {
    public static void main(String[] args) {

        // 학생 성적 임의 배정
        int grade1 = 95;
        int grade2 = 88;
        int grade3 = 72;

        // ArrayList 생성
        List<Integer> grades = new ArrayList<>();

        grades.add(grade1);
        grades.add(grade2);
        grades.add(grade3);

        // 성적 가공 로직
        for (int i = 0; i < grades.size(); i++) {

            // TODO. 가공 작업

            System.out.println("학생의 성적은 " + grades.get(i) + "입니다.");
        }

    }
}
```

두 차이는 뭘까?

```java
// ArrayList 생성
ArrayList<Integer> grades = new ArrayList<>();
```

```java
// ArrayList 생성
List<Integer> grades = new ArrayList<>();
```

Java.util 패키지

컬렉션 프레임워크 사용 예제

선택한 자료구조(ArrayList)를 생성해서 사용한다.

```java
public class Main {
    public static void main(String[] args) {

        // 학생 성적 임의 배정
        int grade1 = 95;
        int grade2 = 88;
        int grade3 = 72;

        // ArrayList 생성
        List<Integer> grades = new ArrayList<>();

        grades.add(grade1);
        grades.add(grade2);
        grades.add(grade3);

        // 성적 가공 로직
        for (int i = 0; i < grades.size(); i++) {

            // TODO. 가공 작업

            System.out.println("학생의 성적은 " + grades.get(i) + "입니다.");
        }

    }
}
```

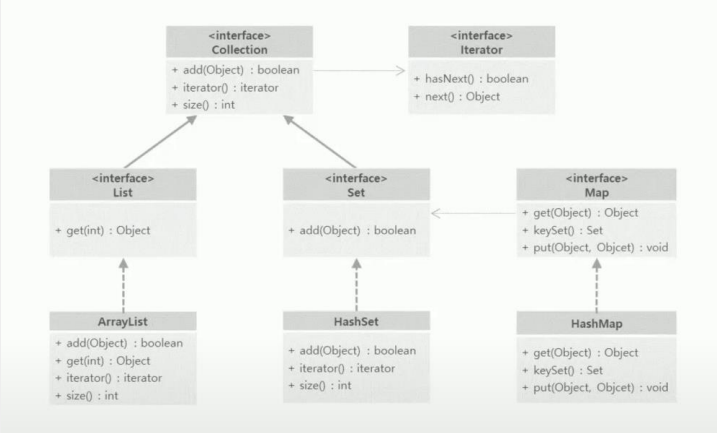
각 자료구조 마다 제공되는 메소드가 있다.
필요 시 마다 구글링을 통해 찾아서 사용하자

여기서는
List가 제공하는 add(), .size(), get() 등이 사용

Java.util 패키지

컬렉션 프레임워크 활용

1. 구현하려는 로직을 파악
2. 해당 로직에 가장 적합한 자료구조를 선택
3. 자바 컬렉션 프레임워크에서 선택한 자료구조를 찾음
4. 구글링을 통해 해당 자료구조의 메소드를 참고하여 개발



Hello World!