📖 Java Study Skill Up Team Review

객체 지향 프로그래밍

[객체]

- 정의는 '실세계에 존재하거나 생각할 수 있는, 인간이 생각하고 표현할 수 있는 모든 것'이다.
- '컴퓨터, 책상, 휴대폰 같은 사물은 물론이고, 강의나 수강 신청 같은 개념 존재하는 것 또한 객체'이다.
- 관점에 따른 개념
  - 모델링 : 객체는 명확한 의미를 담고 있는 대상 또는 개념이다.
  - 프로그래머 관점 : 객체는 `class`에서 생성된 변수이다.
  - 소프트웨어 개발 관점 : 메서드(함수)를 모아놓은 데이터 + 메서드 형태의 소프트웨어 모듈이다.
  - 객체지향 프로그래밍 관점 : 객체는 데이터와 함수를 속성(필드)과 동작(메서드) 용어로 구현한다.

> 쉽게 말해, 데이터(변수)와 데이터(변수)에 관련된 동작(함수)의 절차, 방법, 기능을 모두 포함한 개념이다.
> => 내가 서점에서 책을 사는 경우 : 속성인 '나(손님)'와 동작인 '책 구매'는 하나의 객체인 것이다.
> => 속성인 '서점 주인'과 동작인 '책 판매'도 하나의 객체라고 할 수 있다.

[객체 상호작용]

```
package com.chapter.ch06.example.ex01;

public class ObjectInteraction {
  public static void main(String[] args) {
    Shop shop = new Shop();
    Person person = new Person(shop);
    String IWalk = person.walk("1km");

    System.out.println(IWalk);
  }
}

class Person {
  private Shop shop;

  Person(Shop shop) {
    this.shop = shop;
  }

  String walk(String distance) {
    return shop.rewardForWalking(distance);
  }
}

class Shop {
```

```
  String rewardForWalking(String distance) {
    return distance.equals("1km") ? "60kcal" : "";
  }
}
```

✍ 나만의 Scenario

- '나'는 Person 객체로, '상점'은 Shop 객체로 표현된다.
- '나'는 walk 메서드를 사용하여 '상점'에게 1km 만큼 걸었다고 알린다.
- '상점'은 이 정보를 기반으로 rewardForWalking 메서드를 통해 60kcal 의 보상을 제공한다.
- 이렇게 객체들은 서로 상호작용하면서 동작하는 것을 확인할 수 있다.

[객체 간 관계]

- 웹사이트를 예를 들면, 페이지당 다양한 기능과 영역을 나타내는 여러 객체로 구성되어 있다.
- 이러한 객체와의 상호 작용은 해당 필드와 메서드에 접근하거나 호출함으로써 이루어진다.
- 또 상속을 통해 객체 간의 기능을 확장하거나 재사용 할 수도 있다.

[객체 지향]

- 부품과 같은 객체들을 만들고 이 객체들을 조합해서 프로그램을 완성해 나가는 개발 기법이다.
- 즉, `class`로 생성된 객체들을 조합해서 프로그램을 완성해 나가는 개발 기법이 객체 지향이다.

[객체 지향 프로그래밍]

- 사람이 컴퓨터를 이용해서 객체들을 조합하여 프로그램을 완성해 나가는 개발 기법이다.

💎 객체 지향 프로그래밍 한 줄 설명은 정리했지만, 객체 지향 프로그래밍 핵심은 특징에 있다.

[객체 지향 프로그래밍 특징]

1. 캡슐화 : 필드와 메서드를 하나로 묶고 실제 구현 내용을 외부에 감추는 것

- 캡슐화는 내가 생성한 필드나 메서드에 해당하는 객체의 세부 구현을 숨기고 필요한 기능만을 외부에 제공함으로써 안정성과 유연성을 높이기 위한 기법이다.
  - 핵심 : **안전성**, **유연성**

```
✍ 나만의 비유 Scenario

내가 약사라고 가정하고 캡슐 알약을 제조한다고 하겠다.
캡슐 안에는 가루약이 존재하는데 이 가루약이 두통, 감기, 항암 등등 목적에 맞게 각각 제조
되어 치료에 도움을 준다.
이 캡슐 알약의 제조 과정은 외부에 노출시키지 않고 필요한 환자들에게 캡슐 모양으로 제공한
다.
환자는 필요한 알약만 약사에게서 제공받을 수 있어서 유연성이 높다고 할 수 있다.
또 알약이 제조되는 과정에는 환자가 관여하지 못하기 때문에 안정성이 높다고 할 수 있다.
```

2. 상속

- 상속은 하위 객체가 상위 객체를 상속 받아 필드와 메서드를 사용할 수 있는 기술이다.

- 핵심 : **재사용성**, **유지 보수**

```
쉽게 이해하기

1. 재사용성
상위 객체의 기능을 잘 만들어 놓으면 상속 받을 하위 객체는 중복 코드 없이 이를 사용하거나 활용할 수 있다.
이를 통해 중복적인 코드가 필요 없어 코드의 재사용성이 향상된다.

2. 유지 보수
잘 구성된 상위 객체의 기능을 필요에 의해 상속 받은 하위 객체들이 많이 있을 수 있다.
이때 수정 작업을 하게 되어도 잘 만들어진 상위 객체의 소스만 수정하면 된다.
그러면 당연히 작업 시간이 단축됨과 동시에 잠재적인 오류가 줄어들어서 유지 보수에 용이하다고 할 수 있다.
```

3. 다형성

- 사용 방법은 동일하지만 실행 결과가 다양하게 나오는 성질을 말한다.
  - 핵심 : **유연성**, **확장성**

```
쉽게 이해하기

"입력하다"라는 메서드를 가진 입력장치라는 기본 클래스(또는 인터페이스)를 상속받아 키보드와 휴대폰자판 클래스를 만들 수 있다.
이 두 클래스는 입력하다 메서드를 오버라이드하여 각각의 특성에 맞게 구현할 수 있다.

이처럼 다형성은 한 가지 형태의 인터페이스나 메서드를 제공하면서 다양한 구현을 할 수 있게 한다.
이를 통해 코드의 재사용성과 유지 보수성이 향상되며, 시스템의 확장성을 증가시키는 데 큰 도움을 준다.
```

객체와 클래스

[클래스]

- 객체를 생성하려면 클래스가 필요하다.
- 클래스로부터 생성된 객체를 해당 클래스의 인스턴스라고 한다.
- 그리고 이 클래스로부터 객체를 만드는 과정을 인스턴스화라고 한다.

[클래스 선언]

- 어떻게 객체를 생성(생성자)하고, 객체가 가져야 할 데이터(필드)가 무엇이고, 객체의 동작(메서드)은 무엇인지를 정의하는 내용이 포함된다.
- 클래스명은 첫 문자를 대문자로 하고 캐멀 스타일로 작성한다.

객체 생성과 클래스 변수

[new]

- 클래스로부터 객체를 생성하려면 객체 생성 생저아니 new 연산자가 필요하다.
- new 연산자 뒤에는 생성자 호출 코드가 오는데, 클래스() 형태를 가진다.
- new 연산자는 객체를 생성시킨 후 객체의 주소를 리턴하기 때문에 클래스 변수에 다음과 같이 대입할 수 있다.

```
Loopy loopy = new Loopy();
=> loopy 변수가 Loopy 객체를 참조한다.

Loopy littleLoppy = new Loopy();
=> littleLoppy 변수가 또 다른 Loopy 객체를 참조한다.

즉 Stack 영역에 loopy, littleLoopy가 변수로 존재하며, Heap 메모리에 위치한 각각의
Loopy 객체들을 참조한다.
```

[클래스의 두 가지 용도]

- 라이브러리 클래스 : 실행할 수 없으며 다른 클래스에서 이용하는 클래스
- 실행 클래스 : main() 메서드를 가지고 있는 실행 가능한 클래스
- 일반적으로 자바 프로그램은 하나의 실행 클리스와 여러 개의 라이브러리 클래스로 구성된다.

```
즉 Java 애플리케이션에서 일반적으로 Main.java 클래스는 하나만 존재하고,
모든 다른 클래스는 의존성으로서 호출되어 동작하는 라이브러리 클래스로 작동한다
```

[클래스 구성 멤버]

- 필드 : 객체의 데이터를 저장하는 역할을 한다.
  - String field;
- 생성자 : new 연산자로 객체를 생성할 때 객체의 초기화 역할을 담당한다. 리턴 타입이 없고 이름은 클래스 이름과 동일하다.
  - ClassName() {}
- 메서드 : 객체가 수행할 동작이다. 객체와 객체간의 상호 작용을 위해 호출된다.
  - void methodName() {}

[필드 선언과 사용]

- 변수를 선언하는 방법과 동일하다.
- 타입 필드명 [= 초기값];

[필드와 로컬 변수의 차이점]

- 필드 : 클래스 블록에서 선언되며, 객체 내부에서 존재하고 객체 내/외부 모두 사용 가능하다.
- 로컬 변수 : 생성자와 메서드 블록에서 선언되며 생성자와 메서드 호출 시에만 생성되고 사용된다.

[필드 사용]

- 외부 객체에서에서 사용할 경우 new 연산자로 객체를 생성한 후에 `객체.필드 = "hi"` 도트 연산자로 접근할 수 있다.

- 객체 내부에서는 생성자와 메서드는 객체가 생성된 후 호출되므로 내부에서 필드를 상요할 수 있다. 필드 = "hi"

[생성자 선언과 호출]

- new 연산자는 객체를 생성한 후 연이어 생성자를 호출해서 객체를 초기화하는 역할을 한다.
- 여기서 초기화는 필드 초기화를 하거나 메서드를 호출해서 객체를 사용할 준비를 하는 것을 뜻한다.

1. Constructor Class 가 있다고 가정

```
package com.chapter.ch06.example.ex07;

public class Constructor {
    String constructorField = "hi";
}
```

2. new 연산자를 통해 생성자를 호출해서 객체를 생성한다.

```
package com.chapter.ch06.example.ex07;

public class New {
    void newMethod () {
        Constructor constructor = new Constructor();
        constructor.constructorField = "bye";
    }
}
```

- 이처럼 생성자가 성공적으로 실행이 끝나면 new 연산자는 객체의 주소를 리턴한다.
- 리턴된 주소는 클래스 변수에 대입되어 객체의 필드나 메서드에 접근할 때 이용되는 것이다.

[오버로딩]

```
// 코카콜라도 같은 제품형이지만 Size 별로 여러 가지가 있다.
// => 같은 이름의 메서드에 버전이 여러 가지 있는 거라고 생각하면 된다.

// 매개변수의 개수가 다른 경우
static int add(int a, int b) {
    return a + b;
}
static int add (int a, int b, int c) {
    return a + b + c;
}

// 매개변수의 자료형이 다른 경우
static double add (double a, double b) {
        return a + b;
}
```

```
// 매개변수의 자료형 순서가 다른 경우
static String add(String a, char b) {
    return a + b;
}
static String add(char a, String b) {
    return a + b;
}
```

- 이처럼 같은 메서드명으로 add 가 5개 존재하는데 컴파일 오류가 나지 않는다.
- 왜냐하면 매개변수가 어떻게 되어있는가에 따라서 다른 메서드로 취급하기 때문이다.
- 이 상태에서 완전히 똑같은 메서드를 복사해서 생성하면 컴파일 오류가 발생한다.
- 즉 이걸 메서드 오버로딩이라고 한다.(오버라이딩이랑 다름)
- 반환 자료형이 다른 것은 오버로딩되지 않는다.
- static double add(int a, int b) {return a + b;}

final class, final method

- final field 는 기본적으로 초기값 설정 이후에 값을 변경할 수 없다고 배웠다.

🤔 그럼 final class, final method 는 어떻게 제어가 될까? 이는 상속과 관련이 있었다.

[final class]

- 먼저 class 에 final 키워드를 선언하면 말 그대로 최종적인 클래스로 상속이 불가능한 클래스가 된다.
- 그 말은 final 키워드로 선언된 클래스는 부모 클래스가 될 수 없다는 말이다.

✍ FinalParent.java

```
package com.chapter.ch07.example.ex03;

final class FinalParent {
}
```

✍ Children.java

```
package com.chapter.ch07.example.ex03;

public class Children extends FinalParent {
}
```

🤯 오류 발생

- Cannot inherit from final 'com.chapter.ch07.example.ex03.FinalParent'

오류 내용

- 부모 클래스가 final 로 선언되어 있어서 해당 클래스를 상속할 수 없다.

[final method]

그럼 method 가 final 이면 호출이 불가능할까?

✕ 그건 아니다.

- method 에 final 키워드를 선언하면 최종적인 메서드가 되는 것은 맞지만,
- 호출은 가능하되 오버라이딩을 할 수 없는 메서드가 된다.

ParentFinalMethod.java

```java
package com.chapter.ch07.example.ex03;

public class ParentFinalMethod {
    final void finalMethod() {
        System.out.println("final method");
    }
}
```

ParentFinalMethod.java

```java
package com.chapter.ch07.example.ex03;

public class Children extends ParentFinalMethod {

}
```

Main.java

```java
package com.chapter.ch07.example.ex03;

public class Main {
    public static void main(String[] args) {
        Children children = new Children();
        children.finalMethod();
    }
}
```

- 즉 이처럼 상속 받고 ParentFinalMethod 의 finalMethod 를 호출하면 "final method" 가 잘 출력이 된다.
- 하지만

Children.java

```
package com.chapter.ch07.example.ex03;

public class Children extends ParentFinalMethod {
    @Override
    void finalMethod() {    // ❌ 오류 발생
        System.out.println("overriding");
    }
}
```

- 이처럼 오버라이딩을 시도하면 오류가 발생한다.

🐶 오류 발생

- 'finalMethod()' cannot override 'finalMethod()' in 'com.chapter.ch07.example.ex03.ParentFinalMethod'; overridden method is final

🐶 오류 내용

- final 로 선언되었기 때문에 자식 클래스에서 변경하거나 재정의(override)할 수 없다.

[Protected 접근 제한자]

- protected 는 상속과 관련이 있다.
- 같은 패키지의 경우 default 처럼 접근이 가능하다.
- 다른 패키지의 경우 자식 클래스만 접근이 가능하다.

✍ ProtectedClass.java

```
package com.chapter.ch07.example.ex04;

public class ProtectedClass {
    void protectedMethod() {
        System.out.println("protected method");
    }
}
```

✍ Children.java (같은 패키지)

```
package com.chapter.ch07.example.ex04;

public class Children {
    // 같은 패키지 ⭕ 접근 가능
    void childrenMethod() {
        ProtectedClass protectedClass = new ProtectedClass();
        protectedClass.protectedMethod();
    }
}
```

- 같은 패키지는 protected 키워드로 선언한 method 에 접근이 가능하다.
- 하지만

✍ Children.java (다른 패키지)

```
package com.chapter.ch07.example.ex03;

import com.chapter.ch07.example.ex04.ProtectedClass;

public class Children {
    // 다른 패키지 ❌ 접근 불가능
    void method () {
        ProtectedClass protectedClass = new ProtectedClass();
        protectedClass.protectedMethod();
    }
}
```

- 다른 패키지의 경우 오류가 발생한다.

🤔 오류 발생

- 'protectedMethod()' is not public in 'com.chapter.ch07.example.ex04.ProtectedClass'. Cannot be accessed from outside package

🤔 오류 내용

- 'protected' 접근 제어자는 같은 패키지 내에서는 접근할 수 있으나, 패키지 외부에서는 접근할 수 없도록 제한된 접근 제어자이다.

🤔 해결 방안

- 생성자에서 super() 로 호출하여 상속을 통해 사용이 가능하다.

✍ Children.java (다른 패키지)

```
package com.chapter.ch07.example.ex03;

import com.chapter.ch07.example.ex04.ProtectedClass;

public class Children extends ProtectedClass{
    // 다른 패키지
    public Children() {
        super();    // 자식 생성자에서 super() 호출
        this.protectedMethod(); // ⭕ 접근 가능
        method();
    }

    public void method () {
        this.protectedMethod(); // ⭕ 접근 가능
    }
}
```

- 하지만 X x = new X(); 처럼 직접 객체를 생성해서 사용하는 것은 불가능 하다.

[Getter 와 Setter]

- 자바 언어의 기능이라기 보다는 객체 지향 기능을 갖춘 프로그래밍 언어들에서 많이 사용하는 코딩 스타일(기법)이다.
- 인텔리로 이용해 보기!

```
package six;

public class Product {
    private static double discount;
    private static double increaseLimit;

    private String name;
    private int price;
}
```

- 우클릭 => Generate(Alt + Insert) => Getter and Setter => 모두 선택 => Enter

```
package six;
public class Product {
  private static double discount = 0.2;
  private static double increaseLimit = 1.2;
  private String name;
  private int price;

  public Product(String name, int price) {
    this.name = name;
    this.price = price;
  }

  // 외부에서 값을 가져가려면 getDiscount() 사용
  public static double getDiscount() {
    return discount; // discount 필드의 값을 return
  }

  // 외부에서 discount 값을 변경하려면 setDiscount() 사용
  public static void setDiscount(double discount) {
    Product.discount = discount;
  }

  public static double getIncreaseLimit() {
    return increaseLimit;
  }

  public static void setIncreaseLimit(double increaseLimit) {
    Product.increaseLimit = increaseLimit;
```

```
  }

  public String getName() {
    return name;
  }

  // 조건적으로 주기
  public void setName(String name) {
    if(name.isBlank()) return; // 만약 name이 공백이면 return => 적용 x
    this.name = name;
  }

  // 가격 측정도 getPrice() 에서 할인된 가격으로 return 한다.
  public int getPrice() {
    return (int) (price * (1 - discount));
  }

  // 현재 가격 보다 새가격이 가격 인상 최대치를 적용한 값보다 작다면 그걸 적용하고 그렇
지 않다면 그것보다 크지 않게 적용한다.
  public void setPrice(int price) {
    // this 사용 주의!
    int max = (int) (this.price * increaseLimit);
    this.price = price < max ? price : max;
  }
}
```

[Getter and Setter 예시]

```
package six;
public class Main_3 {
  public static void main(String[] args) {
    Product ballPen = new Product("볼펜", 1000);
    ballPen.setName("삼색 볼펜");
    ballPen.setName("");

    int ballPenPrice = ballPen.getPrice();
    ballPen.setPrice(1500);
  }
}
// 출력 값
// ballPen = {Product@707}
// name = "삼색 볼펜"
// price = 1200
```

[정적(static) 필드와 메서드]

- 언어마다 클래스의 필드와 클래스의 메서드
- 또는 정적 필드와 정적 메서드라고도 한다.
- 정적(static) 요소는 메모리 중 한 곳만 차지한다.(brand, contact)

- 인스턴스 요소들은 각각이 메모리에 자리를 차지한다.(name, no, intro)

[BBQChicken.java]

```
public class Chicken {
    // 정적(클래스) 필드와 메서드들: 본사의 정보와 기능
    // 인스턴스마다 따로 갖고 있을 필요가 없는 것들에 사용한다.
    static String brand = "BBQ치킨";   // 모두 같은 브랜드

    // 본사에 연락할 일이 있는 경우: 대응은 본사에서 하는 본사의 메서드이기 때문에
static
    static String contact() {
        // 정적 메서드에서는 인스턴스 프로퍼티 사용이 불가능하다.
        // System.out.println(name);

        return "%s입니다. 무엇을 도와드릴까요?".formatted(brand);
    }

    // 브랜드는 같지만 매장마다 다르다.
    int no;
    String name;

    BBQChicken(int no, String name) {
        this.no = no;
        this.name = name;
    }

    String intro() {
        // 인스턴스(매장) 메서드에서는 정적 프로퍼티 사용이 가능하다.
        return "안녕하세요, %s %d호 %s점입니다.".formatted(brand, no, name);
    }
}
```

[Main.java]

```
public class Main {
    public static void main(String[] args) {
        // no, name, intro 같은 매장의 정보를 사용하려면 인스턴스(매장)를 생성한 후
        BBQChicken store1 = new BBQChicken(3, "판교");
        String st1Intro = store1.intro();   // intro 메서드 등에 접근해서 사용한다.

        // 하지만
        // 정적(클래스) 필드와 메서드는 인스턴스(매장)를 생성하지 않고
        // 본사 그 자체에서 바로 사용한다.
        String ycBrand = BBQChicken.brand;
        String ycContact = BBQChicken.contact();

        // 인스턴스 메서드는 사용이 불가능하다.
        // String ycName = BBQChicken.name;  // 본사에 가서 여기 무슨점이에요?와 같
음.
```

```
        // String ycIntro = BBQChicken.intro();

        // 인스턴스에서는 클래스의 필드와 메서드 사용은 가능하지만,
        // 편의상 기능일 뿐, 권장하지는 않는다. (혼란 초래. IDE 에서 자동완성 안 됨 주
목)
        String st1Brand = store1.brand; // 마우스 오버 시 static 을 왜 인스턴스로 접
근해?하고 알려줌
        String st1Contact = store1.contact();
    }
}
```

[싱글톤]

- 프로그램 상에서 특정 인스턴스가 딱 하나만 있어야 할 때
- 인스턴스 : new 키워드를 사용해서 클래스의 객체를 생성하면 해당 클래스의 인스턴스(객체)가 된다.