

Java Study Skill Up Team Review

[다형성]

- 다형성이란?
 - 사용 방법은 동일하지만 실행 결과가 다양하게 나오는 성질
 - 동일한 메서드를 가지고 있지만 실행 동작은 다르다는 말이다.
- 다형성을 구현하기 위해서는 자동 타입 변환과 메서드 재정의가 필요하다.

[필드 다형성]

- 필드 타입은 동일하지만(사용 방법은 동일하지만), 대입되는 객체가 달라져서 실행 결과가 다양하게 나올 수 있는 것을 말한다.

[매개변수 다형성]

- 다형성은 필드보다는 메서드를 호출할 때 많이 발생한다.
- 타타입 변환 후 자식 객체가 재정의하고 있는 메서드가 호출된다.

[객체 타입 확인]

- instanceof 연산자
- Java 12 부터 아래처럼 사용 가능

```
if (parent instanceof Child child) {  
    // child 변수 사용 가능  
}
```

[봉인된 클래스]

- 기본적으로 final 클래스를 제외한 모든 클래스는 부모 클래스가 될 수 있다.
- 그러나 Java 15 부터 무분별한 자식 클래스 생성을 방지하기 위해 봉인된(sealed) 클래스가 도입되었다.
- Person 자식 클래스는 Employee, Manager만 가능하고 그 이외의 자식 클래스가 될 수 없도록 Person을 봉인된 클래스로 선언할 수 있다.

```
public sealed class Person permits Employee, Manager { ...  
}
```

- sealed 키워드를 사용하면 permits 키워드 뒤에 상속 가능한 자식 클래스를 지정해야 한다.
- 봉인된 Person 클래스를 상속하는 Employee와 Manager는 final 또는 non-sealed 키워드로 다음과 같이 선언하거나,
- sealed 키워드를 사용해서 또 다른 봉인 클래스로 선언해야 한다.

```
public final class Employee extends Person { ...
}

public non-sealed class Manager extends Person { ...
}
```

- final은 더 이상 상속할 수 없다는 뜻이고,
- non-sealed는 봉인을 해제한다는 뜻이다.
- 따라서 Employee는 더 이상 자식 클래스를 만들 수 없지만 Manager는 다음과 같이 자식 클래스를 만들 수 있다.

```
public class Director extends Manager { ...
}
```

인터페이스

[인터페이스 역할]

사전적 의미 : 두 장치를 연결하는 접속기 해석 : 서로 다른 객체를 연결하는 역할 가정 : 객체 A는 인터페이스를 통해 객체 B를 사용할 수 있다.

* 객체 A가 인터페이스 Method 호출 시 인터페이스는 객체 B 메서드를 호출하고, 그 결과를 받아 객체 A로 전달해준다.

☹️ 객체 A가 객체 B Method를 직접 호출하지 않고 왜 중간에 인터페이스를 거치도록 하는 걸까?

인터페이스는 다형성 구현에 주된 기술로 이용된다. 상속을 이용해서 다형성을 구현할 수도 있지만, 인터페이스를 이용해서 다형성을 구현하는 경우가 더 많다.

[추상 클래스와 인터페이스 비교]

- 추상 클래스
 - 추상 클래스에서 '과일'은 일반적인 특징(색깔, 맛)을 가지고 있다.
 - '과일'은 '사과', '바나나'와 같이 구체적인 하위 클래스를 통해 인스턴스화될 수 있다.
- 인터페이스
 - 인터페이스 '비타민 함유량'은 '과일'이 가져야 하는 다양한 비타민과 무기질을 정의할 수 있다.
 - 이 인터페이스는 어떠한 비타민이나 무기질이 존재해야 하는지 정의할 수 있지만, 그것들이 어떠한 형태로 존재해야 하는지, 어떻게 얻어져야 하는지에 대한 구체적인 사항은 정의하지 않는다. * 예를 들어 '비타민C 함유량', '비타민A 함유량'등의 메서드를 정의할 수 있다.

따라서 추상 클래스 '과일'은 과일의 공통적인 특성과 일부 기본 구현을 제공한다.

인터페이스 '비타민 함유량'은 과일이 가져야 할 추가적인 특성(비타민, 무기질 등)을 정의한

다.

하나의 클래스는 상속과 동시에 인터페이스를 구현할 수 있다.

[인터페이스와 구현 클래스 선언]

- 인터페이스는 '~.java' 형태의 소스 파일로 작성되고 '~.class' 형태로 컴파일되기 때문에 물리적 형태는 클래스와 동일하다.
- 단, 소스를 작성할 때 선언하는 방법과 구성 멤버가 클래스와 다르다.

[인터페이스 선언]


- 인터페이스 선언은 class 키워드 대신 interface 키워드를 사용한다.
- 접근 제한자로는 클래스와 마찬가지로 같은 패키지 내에서만 사용 가능한 default, 패키지와 상관 없이 사용하는 public을 붙일 수 있다.

```
interface 인터페이스명 { ... } // default 접근
public interface 인터페이스명 { ... } // public 접근 제한

=> public interface 인터페이스명 {
// public 상수 필드
// public 추상 메서드
// public default 메서드
// public 정적 메서드
// private 메서드
// private 정적 메서드
}
```

Jay와 하리보는 개발을 한다.

Jay는 프론트엔드 개발자, Ribo는 백엔드 개발자이다.

 Jay.java

```
package inteface;

public class Jay extends FrontEnd implements JavaScript, React {
    public Jay() {
    }

    @Override
    public void javaScriptDevelopment() {
        System.out.println("제이가 자스 개발");
    }

    @Override
    public void reactDevelopment() {
        System.out.println("리엑트 개발");
    }
}
```

```
}
}
```

🔍 소스 검토

- Jay 클래스는 FrontEnd 클래스를 상속 받았으며 JavaScript, React 인터페이스를 구현하고 있다.
- JavaScript, React 에 정의된 추상 메서드를 오버라이드 하여 구현해 주어야 자격이 생긴다. 자격이 없으면 오류가 발생한다.(이는 자스, 리액트 언어를 사용하는 자격이 주어지지 않는다는 말이다.)

📄 Ribo.java

```
package interface;

public class Ribo extends BackEnd implements Java, Spring, JavaScript {
    public Ribo() {
        super(true);
    }

    @Override
    public void javaDevelopment() {
        System.out.println("자바 개발");
    }

    @Override
    public void springDevelopment() {
        System.out.println("스프링 개발");
    }

    @Override
    public void javascriptDevelopment() {
        System.out.println("리보가 자스 개발");
    }
}
```

🔍 소스 검토

- Ribo 클래스는 BackEnd 클래스를 상속 받았으며 Java, Spring, JavaScript 인터페이스를 구현하고 있다.
- Jay 클래스와 같이 추상 메서드를 오버라이드 해줘야 자격이 생긴다.

📄 Main.java

```
package interface;

public class Main {
    public static void main(String[] args) {
        // 다형성
        Jay jay = new Jay();
        JavaScript js = jay;
        React react = jay;
    }
}
```

```

        Ribo ribo = new Ribo();
// 자바스크립트를 사용하는 클래스들의 배열
        JavaScript[] javaScripts = {
            jay, ribo
        };

// 인터페이스 역시 다형성에 의해 자료형으로 작용 가능
        for (JavaScript javaScript : javaScripts) {
            javaScript.javaScriptDevelopment();
        }
    }
}

```

출력 결과

```

제이가 자스 개발
리보가 자스 개발

```

Tip ! 인터페이스에서 static, default, private 사용 가능한 예제

```

package interface;

public interface Java {
    String java = "자바"; // ✕ 초기화 하지 않을 시 오류

    void javaDevelopment();

    static void study() {
        System.out.println("학습 하기");
    }

    default void googling() {
        System.out.println("구글링 하기");
    }

    private String test() {
        return "테스트";
    }
}

```

대충 인터페이스 자체적으로 실행할 수 있다는 말 + implements 구현 클래스 자체적으로 구상하지 않고 객체로도 실행할 수 있다는 말

```

package interface;

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

```

```
Java.study();
Ribo ribo = new Ribo();
ribo.googling();
    }
}
```

출력 결과

```
학습 하기
구글링 하기
```

☹ 인터페이스에 'default', 'static', private 메서드가 도입된 주된 이유가 뭘까?

- 새로운 메서드를 추가할 때, 구현하는 모든 클래스가 해당 메서드를 정의해야 하는 어려움이 있었다. default 메서드 도입으로 새로운 메서드를 추가하더라도 기본적인 구현을 제공할 수 있게되었다. 이 접근 방식은 이전 버전과의 호환성을 보장하므로 인터페이스를 사용하는 기존 클래스를 건들이지 않게 되었다. (ex. main 에서 객체.추가된기능())
- 정적 메서드를 추가하면 인터페이스에서 유틸리티 기능을 직접 제공할 수 있다. 즉, 인터페이스와 관련된 도우미 기능이나 작업을 인터페이스 내에서 직접 제공할 수 있으므로 별도의 유틸리티 클래스가 필요하지 않다.(ex. MathOperations.add(3, 3))
- default, static 메서드 내에서 중복되는 코드 로직을 private 메서드로 분리함으로써 코드의 재사용성과 가독성을 향상시킬 수 있게 되었다. (ex. 두 개의 분리된 메서드 구현부에 같은 private 메서드를 호출하는 로직)````