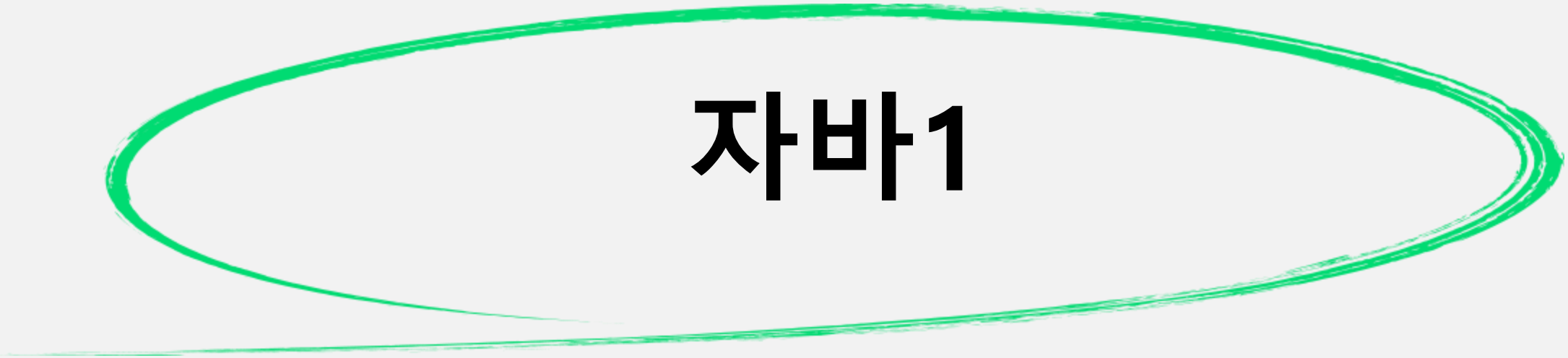

자바1

(클래스와 객체3)

Chapter 09
클래스와 객체3

Chapter09의 학습목표

- 추상 클래스와 인터페이스에 대해 알아본다
- 다형성에 대해 이해한다

다형성

다형성 (Polymorphism)

- 여러 형태를 가지는 성질이란 뜻으로, 한 가지 타입이 여러 가지 형태의 인스턴스를 가질 수 있다는 의미
- 부모 타입 변수에는 모든 자식 인스턴스들이 대입될 수 있음

1) 지금까지의 객체 생성 방식

```
A obj = new A();
```

2) 부모 클래스 타입의 참조변수로 자식 클래스 타입의 객체를 참조할 때

```
A obj = new B();
(클래스 B가 A를 상속할 때)
```

➡ 참조변수 obj 하나로 A타입의 인스턴스를 참조할 수도, B타입의 인스턴스를 참조할 수도 있음

다형성

참조변수와 인스턴스 간의 관계

- 부모 클래스 타입의 참조변수로 자식 인스턴스 참조는 가능, 그 반대는 에러

Child i = new Parents();

Child 클래스가 부모의 멤버 A,B,C를 상속

D라는 참조변수를 호출할 수 없음!

Child 타입 참조변수 i

0 x 100

A, B, C, D

Child 참조변수가 사용할 수 있는 멤버

Parents 타입 인스턴스

A
B
C

Parents p = new Child();

Child 클래스 내부에 D라는 멤버를 선언

Parents 타입 참조변수 p

0 x 200

A, B, C

Parents 참조변수가 사용할 수 있는 멤버

Child 타입 인스턴스

A
B
C
D

다형성

다형성 예제

```
class A{
    void methodA() {
        System.out.println("methodA");
    }
}

class B extends A{
    void methodB() {
        System.out.println("methodB");
    }
}

public class Test{
    public static void main(String[] args) {
        A obj = new B();
        obj.methodA();
        obj.methodB(); //에러!
    }
}
```

```
class A{
    void methodA() {
        System.out.println("methodA");
    }
}

class B extends A{
    void methodB() {
        System.out.println("methodB");
    }
    void methodA() {
        System.out.println("methodA-2");
    }
}

public class Test{
    public static void main(String[] args) {
        A obj = new B();
        obj.methodA();

        B obj2 = new B();
        obj2.methodA();
    }
}
```

다형성

다형성의 활용

동물 객체와 사육사 객체, 동물객체를 상속한 사자 객체를 생성하여
동물원을 가정한 프로그램 구현

```
class Animal{
    void breath() {
        System.out.println("숨쉬기");
    }
}
//동물 클래스를 상속한 사자 클래스 생성
class Lion extends Animal{
    public String toString() {
        return "사자";
    }
}
//사육사 클래스 생성
class ZooKeeper{
    void feed(Lion lion) {
        System.out.println(lion.toString() +"에게 고기 주기");
    }
}
```

```
public class Test{
    public static void main(String[] args) {
        Lion lion1 = new Lion(); //Lion 객체 생성
        ZooKeeper james = new ZooKeeper();
        //james가 lion1에게 먹이를 줌
        james.feed(lion1);
    }
}
```

다형성

다형성의 활용

토끼, 원숭이를 동물원에 추가하려고 했을 경우

```
//동물 클래스를 상속한 사자 클래스 생성
class Lion extends Animal{
    public String toString() {
        return "사자";
    }
}
//동물 클래스를 상속한 토끼 클래스 생성
class Rabbit extends Animal{
    public String toString() {
        return "토끼";
    }
}
//동물 클래스를 상속한 사자 클래스 생성
class Monkey extends Animal{
    public String toString() {
        return "원숭이";
    }
}
```

```
//사육사 클래스 생성
class ZooKeeper{
    void feed(Lion lion) {
        System.out.println(lion.toString() +"에게 고기 주기");
    }
    void feed(Rabbit rabbit) {
        System.out.println(rabbit.toString() +"에게 고기 주기");
    }
    void feed(Monkey money) {
        System.out.println(money.toString() +"에게 고기 주기");
    }
}
```

만약 100개의 동물을 추가하려면..?
메소드 100가지를 추가해야 함

다형성

다형성의 활용

다형성을 이용하여 토끼, 원숭이를 동물원에 추가할 경우

```
//사육사 클래스 생성
class ZooKeeper{
    void feed(Animal animal) {
        System.out.println(animal.toString() +"에게 고기 주기");
    }
}
```

```
public class Test{
    public static void main(String[] args) {
        Animal lion1 = new Lion(); //Lion 객체 생성
        Animal monkey1 = new Monkey();
        Animal rabbit1 = new Rabbit();

        ZooKeeper james = new ZooKeeper();
        //james가 각 동물들에게 먹이를 줌
        james.feed(lion1);
        james.feed(monkey1);
        james.feed(rabbit1);
    }
}
```

- *모든 인스턴스들의 참조변수 타입이 Animal이므로 메소드를 오버라이딩할 필요가 없음*
- *새로운 동물의 종류를 추가하더라도 메소드를 추가할 필요가 없음*

추상 메소드와 추상 클래스

추상 메소드

- 선언부만 정의하고 구체적인 내용은 비워 놓은 메소드
- 추상 메소드를 하위 클래스에서 상속받아 구현
- '추상적인'의 뜻인 'abstract'를 메소드 앞에 붙여 사용

```
abstract 반환값 메소드명();
```

추상 클래스

- 추상 메소드를 멤버로 가지는 클래스
- 추상 메소드를 하나라도 포함하는 클래스를 의미
- 추상 클래스를 상속받은 자식 클래스는 반드시 추상 메소드를 구현해야 함
- 추상 메소드와 같이 abstract를 붙여 선언

```
abstract class 클래스명{
    abstract void 추상메소드명();
    ....
}
```

추상 메소드와 추상 클래스

추상 클래스의 사용

- 추상 클래스를 상속받은 클래스에게 문법적인 제한을 줌으로서 추상 메소드를 자신의 클래스에 맞게 구현하라는 강제성을 줌

```
abstract class Cellphone{
    abstract void methodA();

}

class MyPhone extends Cellphone{

}
```

```
class Cellphone{
    abstract void methodA();

}

class MyPhone extends Cellphone{
    void methodA() {

    }
}
```

추상 메소드와 추상 클래스

추상 클래스 예제

추상클래스 정의

```
abstract class Pokemon{
    String name;

    abstract void attack();
    abstract void sound();

    public String getName() {
        return this.name;
    }
}
```

추상클래스를 상속받는 Pikachu클래스

```
class Pikachu extends Pokemon{
    Pikachu() {
        this.name = "피카츄";
    }

    @Override
    void attack() {
        System.out.println("100만 볼트");
    }

    @Override
    void sound() {
        System.out.println("피카 피카!");
    }
}
```

추상클래스를 상속받는 Squirtle클래스

```
class Squirtle extends Pokemon{
    Squirtle(){
        this.name = "꼬부기";
    }
    @Override
    void attack() {
        System.out.println("물 대포");
    }

    @Override
    void sound() {
        System.out.println("꼬북 꼬북!");
    }
}
```

추상 메소드와 추상 클래스

추상 클래스 예제

```
public class Test{
    public static void main(String[] args) {
        Pikachu pika = new Pikachu();
        System.out.println("이 포켓몬은 " + pika.getName());
        pika.attack();
        pika.sound();

        Squirtle squirtle = new Squirtle();
        System.out.println("이 포켓몬은 " + squirtle.getName());
        squirtle.attack();
        squirtle.sound();
    }
}
```

```
이 포켓몬은 피카츄
100만 볼트
피카 피카!
이 포켓몬은 꼬부기
물 대포
꼬북 꼬북!
```

인터페이스

인터페이스 (Interface)

- inter(사이의) + face(마주하다)의 합성어로, 물체들 사이에서 상호작용을 하기 위한 매개 역할을 하는 것을 의미
- 인터페이스를 바탕으로 클래스를 작성 (클래스 : 붕어빵 기계 , 인터페이스 : 붕어빵을 만드는 재료나 제작법을 적어 놓은 종이)
- public static final, public abstract를 작성하지 않아도 컴파일러가 자동으로 추가
- '상속'이 아닌 '구현'이라고 표현
- 자식 클래스는 implements를 사용하여 인터페이스를 구현함

```
interface 인터페이스명{
    public static final 자료형 = 값;
    public abstract 반환타입 메소드명(매개변수);

}
```

```
interface A{
    int a = 4;
    void methodA();
    void methodB();
}

//A를 구현하는 클래스 B
class B implements A{
    public void methodA() {
        //methodA 구현
    }
    public void methodB() {
        //methodB 구현
    }
}
```

인터페이스

인터페이스의 구현과 상속

- 인터페이스를 구현하는 자식 클래스는 상속과 구현이 동시에 가능함
- 인터페이스 간에도 상속이 가능 => 다수의 인터페이스 구현 가능 => 다중 상속이 가능

```
//C를 상속받는 동시에 A를 구현하는 클래스 B
class B extends C implements A{
    /*
     * interface A의 멤버가 존재
     * 부모 클래스 C의 멤버가 존재
     */
}
```

```
interface A{
    void methodA();
}

interface B{
    void methodB();
}

interface C extends A, B{
    //A와 B를 합친 종합적인 기능을 다루는 인터페이스
}
```

인터페이스 예제

1. 여행 가이드 객체를 만들고 그 인스턴스는 레저, 관광, 음식 투어를 진행할 수 있게 만든 프로그램

- 인터페이스는 객체 간의 상호작용을 위한 일종의 규약

```
interface Providable{
    void leisureSports();
    void sightseeing();
    void food();
}

class KoreaTour implements Providable{
    @Override
    public void leisureSports() {
        System.out.println("한강에서 수상스키 투어");
    }

    @Override
    public void sightseeing() {
        System.out.println("경복궁 관람 투어");
    }

    @Override
    public void food() {
        System.out.println("전주 비빔밥 투어");
    }
}
```

```
class TourGuide{
    private Providable tour = new KoreaTour();
    //인터페이스로 타입선언
    public void leisureSports() {
        tour.leisureSports();
    }
    public void sightseeing() {
        tour.sightseeing();
    }
    public void food() {
        tour.food();
    }
}
```

```
public class Test{
    public static void main(String[] args) {
        TourGuide guide = new TourGuide();
        guide.leisureSports();
        guide.sightseeing();
        guide.food();
    }
}
```

인터페이스 예제

2. 가이드가 일본에서 근무하게 되어 코드를 변경해야 할 경우 Providable 인터페이스를 구현하여 JapanTour 클래스를 작성

```
class JapanTour implements Providable{
    @Override
    public void leisureSports() {
        System.out.println("도쿄타워 번지점프 투어");
    }

    @Override
    public void sightseeing() {
        System.out.println("오사카 관람 투어");
    }

    @Override
    public void food() {
        System.out.println("초밥 투어");
    }
}
```

```
class TourGuide{
    private Providable koreaTour = new KoreaTour();
    private Providable japanTour = new JapanTour();
    //인터페이스로 타입선언
    public void leisureSports() {
        japanTour.leisureSports();
    }
    public void sightseeing() {
        japanTour.sightseeing();
    }
    public void food() {
        japanTour.food();
    }
}
```

인터페이스

인터페이스와 다형성

- 특정 인터페이스를 구현한 인스턴스는 해당 인터페이스 타입의 참조변수로 참조가 가능
- 하나의 객체를 여러가지 타입으로 참조 가능

인터페이스명 참조변수명 = new 클래스명();

```
interface Camera{
    void photo();
}

interface Call{
    void calling();
}

interface Memo{
    void write();
}

interface Clock{
    void clock();
}
```

```
class MyCellPhone implements Camera, Call, Memo, Clock{
    @Override
    public void clock() {}
    @Override
    public void write() {}
    @Override
    public void calling() {}
    @Override
    public void photo() {}
}

class PhoneUser{
    void call(Call c) {
        System.out.println("전화를 걸었습니다.");
    }
}
```

인터페이스

인터페이스와 다형성

```
public class Test{
    public static void main(String[] args) {
        MyCellPhone phone1 = new MyCellPhone();
        Camera  phone2 = new MyCellPhone();
        Call    phone3 = new MyCellPhone();
        Memo    phone4 = new MyCellPhone();
        Clock   phone5 = new MyCellPhone();

        PhoneUser   user1 = new PhoneUser();
        user1.call(phone3);
        user1.call(phone1);
    }
}
```

MyCellPhone 클래스가 Camera, Call, Memo, Clock 4개의 인터페이스를 구현

MyCellPhone의 인스턴스는 자신 이외에 추가로 4개의 인터페이스 타입을 참조

정리

추상 클래스를 사용하는 경우

- 관련된 클래스 사이에서 코드를 공유하고 싶을 때
- 공통적인 필드나 메소드가 많은 경우, 또는 public 이외의 접근 지정자를 사용해야 하는 경우
- 정적이 아닌 필드나 상수가 아닌 필드를 선언하기를 원할 때

일반적인 필드도 선언할 수 있으며, 일반적인 메소드도 정의 가능

인터페이스를 사용하는 경우

- 관련 없는 클래스들이 인터페이스를 구현하기를 원할 때
  Ex) Comparable클래스를 구현할 때
- 특정한 자료형의 동작을 지정하고 싶지만 누가 구현하든지 신경 쓸 필요가 없을 때
- 다중 상속이 필요할 때

모든 메소드는 public, abstract가 되며, 여러 개의 인터페이스를 상속받아 동시에 구현이 가능

수고하셨습니다.