Java (클래스 & 객체)

날짜	@2024년 7월 11일
상태	완료

객체지향 프로그래밍

객체지향 프로그래밍

OOP: Object Oriented Programming

- 객체: 사물과 같이 유형적인 것과 개념이나 논리와 같은 무형적인 것들
- 지향: 작정하거나 지정한 방향으로 나아감
- 객체 모델링: 현실 세계의 객체를 SW 객체로 설계하는 것

클래스

객체를 만드는 설계도

인스턴스

클래스를 통해 생성된 객체

객체지향 프로그래밍 특징 (A PIE)

- **A**bstraction (추상화)
- **P**olymorphism (다형성)
- **I**nheritance (상속)
- **E**ncapsulation (캡슐화)

⇒ **모듈화**된 프로그래밍, 높은 **재사용성**, 용이한 **디버깅**, 정보 **보호**

클래스

배열을 이용한 관리

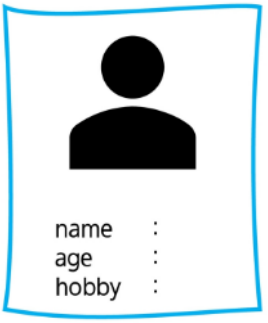

```
String[] names = new String[2];
names[0] = "Yang";
names[1] = "Hong";

int[] ages = new int[2];
age[0] = 45;
age[1] = 25;

String[] hobbies = new String[2]
hobbies[0] = "유튜브"
hobbies[1] = "골프"
```

⇒ 하나로 묶기

```
public class Person {
    String name;
    int age;
    String hobby;
}
```



정보 출력

```
void info(name, age, hobby) {
    System.out.println("나의 이름은 " + name + "입니다.");
    System.out.println("나이는 " + age + "세, 취미는" + hobby +
```

```
}

// 나의 이름은 Yang 입니다.
// 나이는 45세, 취미는 유튜브 입니다.
// 나의 이름은 Hong 입니다.
// 나이는 25세, 취미는 골프 입니다.
```

⇒ 하나로 묶기

```
public class Person {
    String name;
    int age;
    String hobby;

    public void info() {
        System.out.println("나의 이름은 " + name + "입니다.");
        System.out.println("나이는 " + age + "세, 취미는" + hobb
    }
}
```

💡 **클래스**

- 관련 있는 변수와 함수를 묶어서 만든 `사용자 정의 자료형`
- 모든 객체들의 생산처
- `클래스 == 객체` 생성 틀
- 프로그래밍이 쓰이는 목적을 생각하여 어떤 객체를 만들어야 하는지 결정
- 각 객체들이 어떤 특징 (속성과 동작)을 가지고 있을지 결정
- 클래스를 통해 생성된 객체 → `인스턴스`
- 객체들 사이에서 메시지를 주고 받도록 만들어 줌

클래스 구성

- 속성 (`Attribute`) - 필드
- 동작 (`Behavior`) - 메소드

- 생성자 (`Constructor`)
- 중첩 클래스 (클래스 내부의 클래스)

클래스 선언 및 객체 생성

💡 [접근제한자] [활용제한자] class [클래스명] {

속성 정의 (필드)

기능 정의 (메소드)

생성자)

}

- 접근제한자: `public` / `default`
- 활용제한자: `final` / `abstract`

```
public class Person {
    String name;
    int age;

    public void eat() {
    }

    public Person() {
    }
}
```

- `[클래스명] [변수명] = new [클래스명]();`
- `[변수명].[필드명]`
- `[변수명].[메서드명]();`

변수

- **클래스 변수 (`class variable`)**
  - 클래스 영역 선언 (`static` 키워드)
  - 생성 시기: 클래스가 메모리에 올라갔을 때

    - 모든 인스턴스가 공유
- **인스턴스 변수 (`Instance variable`)**
    - 클래스 영역 선언
    - 생성 시기: 인스턴스가 생성되었을 때 (`new`)
    - 인스턴스 별로 생성
- **지역 변수 (`local variable`)**
    - 클래스 영역 이외 (메서드, 생성자 등)
    - 생성 시기: 선언되었을 때

메소드

- 객체가 할 수 있는 **행동** 정의
- 어떤 작업을 수행하는 **명령문의 집합**에 이름을 붙인 것
- 메소드의 이름은 **소문자로 시작**하는 것이 관례

[접근제한자] [활용제한자] [반환값] [메소드명]([매개변수들]) {

행동

...

}

- 접근제한자: `public` / `protected` / `default` / `private`
- 활용제한자: `static` / `final` / `abstract` / `synchronized`

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

메소드 선언

- 선언 시 `{}` 안에 메소드가 해야 할 일 정의

메소드 호출

```
public class Person {
    public void info() {
```

```
        // 메서드 내용 정의
    }
    public static void hello() {
        // 메서드 내용 정의
    }
}
```

- 호출한 메소드가 선언되어 있는 클래스 접근
- `[클래스객체].[메소드명]`으로 호출

```
Person p = new Person();
p.info();
```

- `static`이 메소드에 선언되어 있을 때는 `[클래스명].[메소드명]`으로 호출

```
Person.hello();
```

메서드 구성

- **매개변수** (`Parameter`)
  - 메소드에서 사용하는 것
- **인자** (`Argument`)
  - 호출하는 쪽에서 전달하는 것

```
// Person 클래스 내부
public void study(int time) {
    // int time = ?
    // 매개변수는 해당 위치에 선언한 지역 변수
    System.out.println(time + "시간 공부");
}
```

```
Person p = new Person();
p.study(10);
```

- 매개변수 생략 가능
- 매개변수 전달 시 **묵시적 형 변환**

```
p.study((byte) 10);   // O
p.study((short) 10);   // O
p.study(10);   // O
p.study(10L);   // X
p.study(10.0f);   // X
p.study(10.0);   // X
p.study(10, 10);   // X
```

리턴 타입

- 메소드를 선언할 때 지정, 없으면 `void` (`return문` 생략 가능)
- 리턴 타입을 작성했다면 반드시 해당 타입의 값 리턴 필요
- 리턴 타입은 하나만 적용 가능

```
// Person 클래스 내부
public int getAge() {
    return age;
}
```

```
Person p = new Person();
p.name = "Yang";
p.age = 45;
p.hobby = "유튜브";

int age = p.getAge();
```

메소드 오버로딩

- 이름이 같고 매개변수가 다른 메소드를 여러 개 정의하는 것
- 중복 코드에 대한 효율적 관리 가능
- 매개변수의 개수 또는 순서, 타입이 달라야 함 (이름만 다른 것은 X)
- 리턴 타입이 다른 것은 무의미

📌 **클래스와 객체 정리**

- **클래스**: 관련 있는 변수와 함수를 묶어 만든 사용자 정의 자료형
- **객체**: 하나의 역할을 수행하는 '메소드와 변수(데이터)'의 묶음
- **객체 지향 프로그래밍**: 프로그램을 단순히 데이터와 처리 방법으로 나누는 것이 아니라, 프로그램을 수많은 '객체(object)'라는 기본 단위로 나누고 이들의 상호 작용으로 서술하는 방식

생성자

생성자 메서드

- `new` 키워드와 함께 호출하여 객체 생성
- 클래스명과 동일
- 결과형 리턴값을 갖지 않음 (`void` 작성 x)
- 객체가 생성될 때 **반드시** 하나의 생성자 호출
- 멤버 필드의 초기화에 주로 사용
- 하나의 클래스 내부에 생성자가 하나도 없으면 자동적으로 default 생성자가 있는 것으로 인지
  - `default 생성자`: 매개 변수와 내용이 없는 생성자
- 매개 변수의 개수가 다르거나, 자료형이 다른 여러 개의 생성자가 있을 수 있음 (`생성자 오버로딩`)
- 생성자의 첫 번째 라인으로 `this()` 생성자를 사용하여 또 다른 생성자 호출 가능

```
public class Dog {
    public Dog() {
        System.out.println("기본 생성자!");
        System.out.println("클래스 이름과 동일하고 반환 타입 X");
    }
}
```

기본(디폴트) 생성자

- 클래스 내에 생성자가 하나도 정의되어 있지 않을 경우 `JVM`이 자동으로 제공하는 생성자
- **형태**: 매개 변수가 없는 형태

  `클래스명() {}`

```
class Dog {
    // 생성자가 하나도 없는 상태
    // JVM (Java Virtual Machine)이 자동으로 제공
    // Dog() {}
}

class Main {
    public static void main(String [] a) {
        // 객체 생성
        Dog d = new Dog();
    }
}
```

파라미터가 있는 생성자

- 생성자의 목적 == 필드 초기화
- 생성자 호출 시 값을 넘겨줘야 함
- 해당 생성자를 작성하면 JVM에서 기본 생성자 추가 X

```
class Dog {
    String name;
    int age;
    Dog(String n, int a) {
        name = n;
        age = a;
    }
}

class Main {
    public static void main(String [] a) {
        Dog d1 = new Dog();
```

```
        d1.name = "쫑";
        d1.age = 3;
        Dog d2 = new Dog("메리", 4);
    }
}
```

생성자 오버로딩 지원

- 클래스 내에 메소드 이름이 같고 매개변수의 타입/개수가 다른 것

```
class Dog {
    Dog() {}
    Dog(String name) {}
    Dog(int age) {}
    Dog(String name, int age) {}
}

class Main {
    public static void main(String [] a) {
        Dog d = new Dog();
        Dog d2 = new Dog("쫑");
        Dog d3 = new Dog(3);
        Dog d4 = new Dog("메리", 4);
    }
}
```

this

- 참조 변수로써 객체 자신을 가리킴
- `this`를 이용하여 자신의 멤버 접근 가능
- 지역 변수(매개변수)와 필드의 이름이 동일할 경우, 필드임을 식별할 수 있게 함
- 객체에 대한 참조이므로, `static` 영역에서 `this` 사용 불가

this의 활용

- `this.멤버변수`
- `this([인자값])`: 생성자 호출

- this 생성자 호출 시 제한 사항
  - 생성자 내에서만 호출 가능
  - 생성자 내에서 첫 번째 구문에 위치 필요

```
class Dog {
    String name;
    int age;

    void info() {
        System.out.println(this.name);
        System.out.print(this.age);
    }

    Dog() {
        this("쫑");
    }
    Dog(String name) {
    }
}
```