1. 객체지향언어란?

1.1 객체 지향 프로그래밍

(Object-Oriented Programming)

- 모든 데이터를 객체로 취급하고, 객체가 프로그램밍의 중심이 되는 언어
- 객체란 속성과 기능을 가지는 프로그램의 단위이다.
- 객체를 구체화 하는 형태의 프로그래밍을 객체 지향 프로그래밍이라 함

1.2 객체지향언어의 특징

높은 생산성

- 코드의 재사용성을 높여서 코드를 작성하는 비용을 줄일 수 있음

유지보수가 쉬움

- 각각의 객체를 나눠서 문제 발생 시 그 부분만을 확인하면 됨

보안성이 향상 됨

- 캡슐화를 이용하여 실제 구현되는 부분을 외부에 드러내지 않게 정보 은닉이 가능

신뢰성 있는 프로그램 작성 가능.

- 코드를 재사용 하여 제대로 동작하는 기능이라면 다른 곳에서 안정적으로 사용 가능

2. 클래스와 객체

2.1 클래스와 객체의 정의와 용도

클래스

- 객체를 정의하는 설계도
- 객체를 생성하는데 사용
- ex: TV 설계도

객체

- 실존하는 사물이나 개념
- ex: TV

2.2 객체와 인스턴스

객체 ≒ 인스턴스

객체

- 추상적인 개념
- 차(Car)

인스턴스

- 객체의 구체적인 예시
- 내 차(Car)

2.3 객체의 구성요소 – 속성과 기능

속성

- 객체의 특성을 나타내는 값
- 멤버변수

기능

- 객체가 할 수 있는 동작(행위)를 의미
- 메서드

```
class Car {
    // 속성
    int id;
    int speed;
    // 행위
    void accelerate() {
        speed++;
    }
}
```

2.4 인스턴스의 생성과 사용

인스턴스의 생성방법

```
public static void main(String[] args) {
    Car car;    // 객체를 다룰 참조변수 선언. 아직은 null
    car = new Car();    // 객체 생성. 생성된 객체의 주소를 할당

    Car car2 = new Car();   // 위의 작업을 한 줄로 가능

}
```

2.4 인스턴스의 생성과 사용(2/4)

```
public static void main(String[] args) {
    Car car;    // 객체를 다룰 참조변수 선언. 아직은 null
    car = new Car();    // 객체 생성. 생성된 객체의 주소를 할당

    car.id = 1;     // car의 id 값을 1로 저장
    car.speed = 10; // car의 speed 값을 10으로 저장

    car.accelerate();   // car의 accelerate 메서드 호출

}
```

3. 변수와 메서드

3.1 선언위치에 따른 변수의 종류

변수의 선언 위치에 따라 변수의 종류와 사용 범위가 달라짐

```
class Var {
    int x;    // 인스턴스 변수
    static int y;    // 클래스(스태틱) 변수

    void method() {
        int z;  // 지역변수
    }
}
```

3.1 선언위치에 따른 변수의 종류

1. 인스턴스변수(instance variable)
   - 각각의 인스턴스마다 다른 값을 가짐
   - 인스턴스를 만들 때 생성됨.
   - 참조변수.변수명 으로 접근

2. 클래스변수(class variable)
   - 클래스에서 공통으로 사용하는 변수
   - 클래스명.변수 로 접근

3. 지역변수(local variable)
   - 메서드 내에 선언되어 메서드 안에서만 사용되는 변수

3.2 클래스변수와 인스턴스변수

클래스 변수

- 클래스 영역에서 static 키워드를 갖는 변수
- 해당 클래스를 통해 만들어진 인스턴스들은 모두 공통된 클래스 변수를 가짐

인스턴스 변수

- 클래스 영역에서 static 키워드를 가지지 않는 변수
- 각 인스턴스 마다 다른 값을 가짐

3.3 메서드(method)

메서드

- 어떤 작업을 수행하기 위한 명령문의 집합
- 변수를 입력 받아서 메서드의 리턴 타입의 값을 돌려준다.

(입력이 없을 수도 있으며, void는 리턴 값이 없다.)

메서드 특징

- 코드의 반복을 줄일 수 있다.
- 메서드를 기능 단위로 작성하여 유지보수를 용이하게 한다.(SRP)

3.4 return문

- 현재 실행 중인 메서드를 종료하고 호출한 메서드로 돌아감.
- 이때 반환하는 값의 타입은 리턴 타입과 같아야 한다.

```
long subtract(long a, long b) {
    return a - b;
}
```

- 위의 메서드는 반환 타입이 long 이므로 a – b는 long 타입이어야 함

3.4 return문

- 만약 반환되는 값이 없다면(void) return을 적지 않더라도 블록({})이 끝나면 메서드는 종료 된다.

```
void method() {
    System.out.println("반환값이 없는 경우");
}
```

3.5 메서드의 호출

메서드의 호출방법

- 메서드 이름(인자1, 인자2, ...)으로 호출

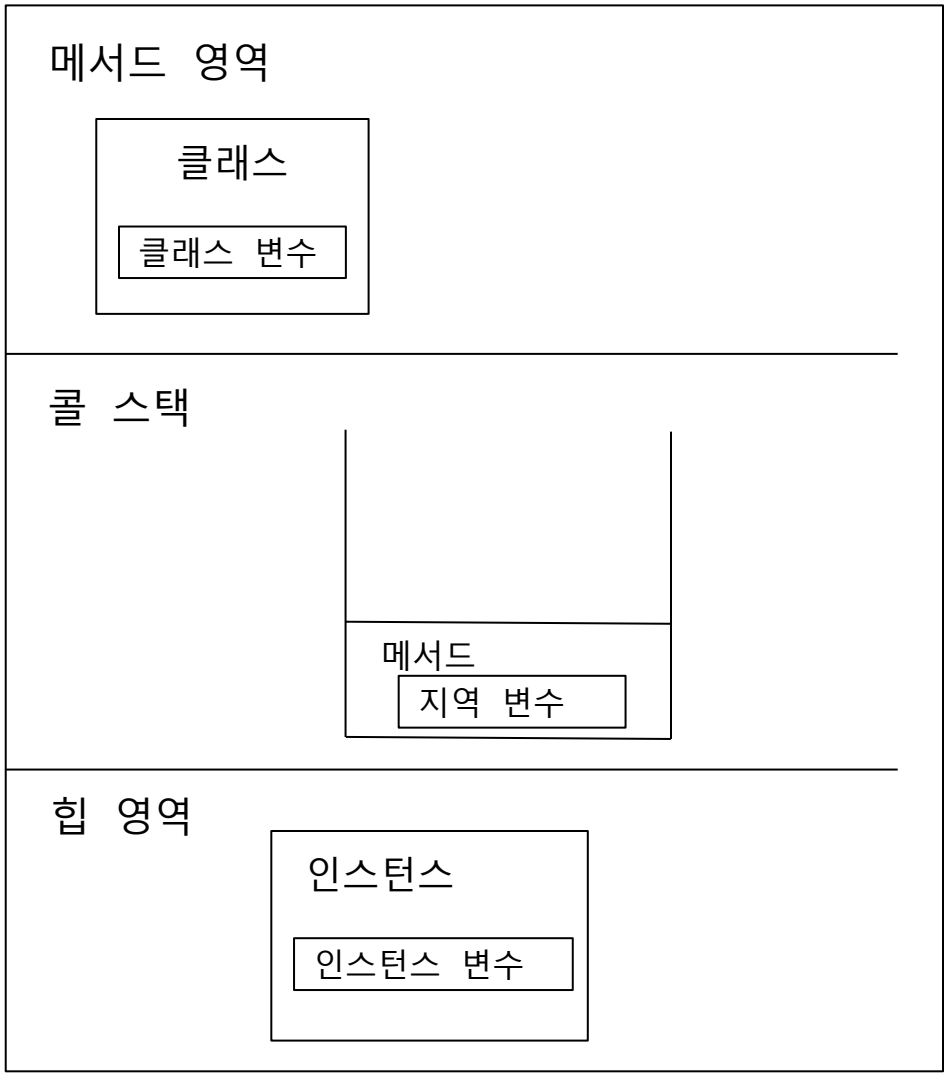
매개변수가 없는 경우

```
method();
```

매개변수가 있는 경우

```
method(x, y);
```

3.6 JVM 메모리 구조



메서드 영역

- 클래스에 대한 정보를 저장
- 클래스 변수도 같이 들어있음

호출(콜, Call) 스택

- 메서드의 작업에 필요한 메모리 공간을 제공
- 지역변수들과 연산의 중간 결과등을 저장

힙 영역

- 생성된 인스턴스가 저장되는 공간
- 인스턴스 변수도 같이 저장 됨

3.6 JVM 메모리 구조 - 호출스택

호출스택의 특징

- 메서드가 호출 되면서 필요한 메모리를 할당 받음
- 메서드가 종료되면 사용한 메모리를 반환하고 스택에서 제거 됨
- 메서드는 스택 구조로 위에 있는 메서드가 현재 실행중인 메서드

3.7 기본형 매개변수와 참조형 매개변수

기본형 매개변수

- 값을 읽기만 가능
- 호출 받은 메서드에서 값을 변경하더라도 호출 한 메서드에서는 값이 변경되지 않음

참조형 매개변수

- 값을 읽고 변경 가능
- 호출 받은 메서드에서 값을 변경하면 호출 한 메서드에서도 값이 변경 되어 있음

3.8 재귀호출(recursive call)

재귀호출이란

- 메서드 안에서 자기 자신을 호출
- 짧게 코드를 작성할 수 있으나 디버깅이 어렵다는 단점을 가짐

```
static int factorial(int n) {
    int res = 0;

    if (n == 1 || n == 0) {     // 탈출 조건. 기저 사례
        return 1;
    }
    res = n * factorial(n - 1);

    return res;
```

3.9 클래스메서드(static메서드)와 인스턴스메서드

인스턴스 메서드

- 참조변수.메서드이름(인자) 로 호출
- 인스턴스 변수 사용 가능

클래스메서드(static메서드)

- 객체 생성 없이 클래스이름.메서드이름(인자) 로 호출
- 인스턴스 변수 사용 불가능

3.10 멤버간의 참조와 호출

- 같은 클래스의 인스턴스들은 static 메서드와 static 인스턴스를 공유
- Static 메서드는 인스턴스 변수와 인스턴스 메서드를 사용 불가능

```
double divide() {
    return a / b;
}

static void add(int x, int y) {
    divide();   // 인스턴스 메서드 호출 불가
}
```

```
int x, y;

static void add() {
    divide(x, y);   // 인스턴스 변수 사용 불가
}
```

4. 메서드 오버로딩

4.1 메서드 오버로딩

하나의 클래스에 같은 메서드 이름으로 여러 개의 메서드를 만드는 것

```
int add(int a, int b) {
    return a+b;
}

long add(int a, long b) {
    return a+b;
}

long add(long a, int b) {
    return a+b;
}
```

4.2 오버로딩의 조건

- 메서드의 이름이 같아야 한다
- 메서드의 매개변수의 개수나 타입이 달라야 한다
  (리턴타입만 다른 경우는 불가능)

5. 생성자

5.1 생성자(constructor)란?

생성자

- 객체를 생성하면서 인스턴스 변수를 원하는 값으로 초기화 시키는 메서드
- 인스턴스를 만들면서 수행할 작업을 추가할 수 있음
- 모든 클래스에는 하나 이상의 생성자가 있어야 함

5.2 생성자의 조건

생성자의 조건

- 해당 클래스의 이름과 동일해야 한다
- 리턴값이 없다.
- 하나의 클래스가 여러 개의 생성자를 가질 수 있음(오버로딩 가능)

```
class Student {
    int id;
    String name;

    public Student() {} // 기본 생성자

    public Student(int i, String n) {   // 매개변수가 있는 생성자
        id = i;
        name = n;
    }
}
```

5.3 기본 생성자(default constructor)

기본 생성자

- 매개변수가 하나도 없으며, 아무런 명령도 가지지 않는 생성자
- 컴파일 시 클래스에 생성자가 하나도 없으면 자동으로 추가 됨

5.4 매개변수가 있는 생성자

매개변수가 있는 생성자

- 호출 시 인자 값으로 초기화 할 변수의 값을 넘겨 초기화
- 매개변수가 있는 생성자를 만들면 자바 컴파일러는 기본생성자를 만들어주지 않는다.

5.5 생성자에서 다른 생성자 호출하기 – this()

this()

- 같은 클래스 내의 다른 생성자를 호출할 때 사용
- 맨 윗줄에만 사용 가능

```
class Student {
    int id;
    String name;

    public Student() {
        this(3, "김자바");
    }

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }


}
```

5.6 참조변수 this

this

- 인스턴스 자기 자신을 참조하는 참조 변수
- 아래의 코드는 this.id(멤버변수 id)에 id(매개변수)를 저장하라는 의미

```
public Student(int id, String name) {
    this.id = id;
    this.name = name;
}
```

6. 변수의 초기화

6.1 변수의 초기화

- 변수를 선언하고 처음 값을 지정하는 것
- 멤버변수와 배열은 기본값을 초기화가 되어 생략할 수 있다.
- 지역변수는 초기화를 하지 않으면 사용할 수 없다.

6.2 멤버변수의 초기화

멤버변수의 초기화 방법

1. 명시적 초기화(explicit initialization)

- 지역변수를 초기화 하는 것과 마찬가지로 생성과 동시에 값을 넣어주는 것을 의미

2. 생성자(constructor)

3. 초기화 블럭(initialization block)

- 클래스 필드의 초기화만을 담당하는 중괄호로 둘러쌓인 블록
- 생성자보다 먼저 호출됨.

6.3 초기화 블록

인스턴스 초기화 블록

- 생성자 보다 먼저 실행됨
- 여러 개의 생성자가 있는 경우 모든 생성자에서 공통으로 수행되어야 하는 로직을 넣어 코드 중복을 줄일 수 있음

클래스 초기화 블록

- 인스턴스 초기화 블록에 static 키워드가 추가 된 형태
- 클래스가 메모리에 처음 올라갈 때 한 번만 실행
- 생성자나 인스턴스 초기화 블록으로 수행할 수 없는 클래스 변수의 초기화를 수행할 때 사용