JAVA STUDY

클래스의 이해와 객체지향 프로그래밍

객체 지향 프로그래밍이란?

객체지향 프로그래밍(Object-Oriented Programming), 줄여서 OOP, 프로그램을 어떻게 설계해야 하는지에 대한 일종의 개념이자 방법론.

> **절차지향 프로그래밍 (Procedural Programming)**
> 초기 프로그래밍 방식은 `절차적 프로그래밍` 방식. 물이 위에서 아래로 흐르는 것처럼 **순차적인 처리**가 중요시 되며 프로그램 전체가 유기적으로 연결되도록 만드는 프로그래밍
>
> 대표적인 절차지향 언어에는 **C**언어

- 절차적 프로그래밍의 문제점
  조금만 복잡해져도 순서도로 나타내는 것이 거의 불가능할 정도로 꼬여서 유지보수가 어려울뿐만 아니라 다른사람이 들이 보고 이해하는 것이 불가능.
- 이 문제를 해결하기 위해 `구조적 프로그래밍`을 제안됨
  프로그램을 함수단위로 나누고 함수끼리 호출하는 구조적인 방식을 제안 하면서 이러한 위기를 벗어나게 된다. 즉, 큰 문제가 있다고 가정하면 작은 문제들로 나누워서 해결하는 `하향식(Top-Down)` 방식이라고도 한다.
- 하지만 이러한 구조적 프로그래밍에도 문제가 있었다.
  함수는 데이터의 처리 방법을 구조화 했을뿐, 데이터 자체는 구조화 하지 못했다.
- 이를 극복하기 위해 `객체 지향 프로그래밍`이 등장 하였다.
- **객체 지향프로그래밍** 등장
  큰 문제를 작은것으로 나누는 것이 아니라, 작은 문제들을 해결할 수 있는 객체들을 만든 뒤, 객체들을 조합하여 큰 문제를 해결하는 `상향식(Bottom-Up)` 해결법을 도입한 것.

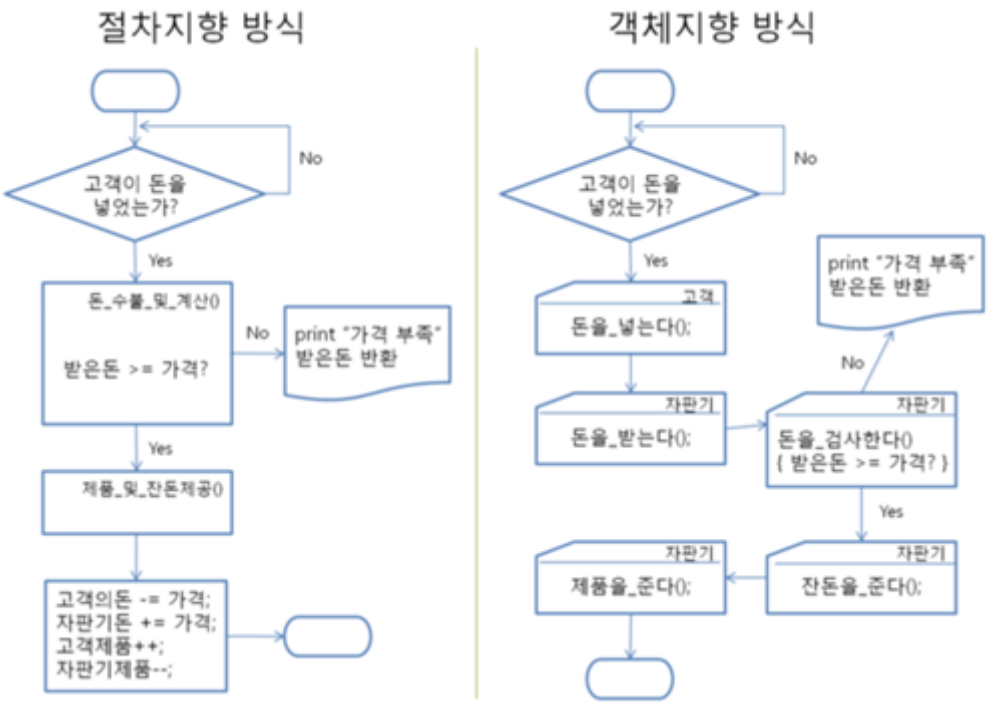
> **객체지향 프로그래밍(Object Oriented Programming)**
>
> 객체지향이란 **실제 세계를 모델링**하여 소프트웨어를 개발하는 방법
>
> 객체지향 프로그래밍에서는 데이터와 절차를 하나의 덩어리로 묶어서 생각하게 된다.
>
> 마치 컴퓨터 부품을 하나씩 사다가 컴퓨터를 조립하는 것과 같은 방법

객체 지향의 반대는 절차 지향?

객체지향의 반대는 절차지향이 아니고 절차지향의 반대는 객체지향이 아니다. 위에서 설명한 것처럼 절차지향은 순차적으로 실행에 초점이 되어 있고 객체지향은 객체간의 관계/조직에 초점을 두고 있다. 객체지향은 절차적으로 실행되지 않냐? 라는 의문이 드는데 객체지향 역시 절차지향과 동일한 순서로 실행된다.

- 절차지향은 **데이터**를 중심으로 함수를 구현

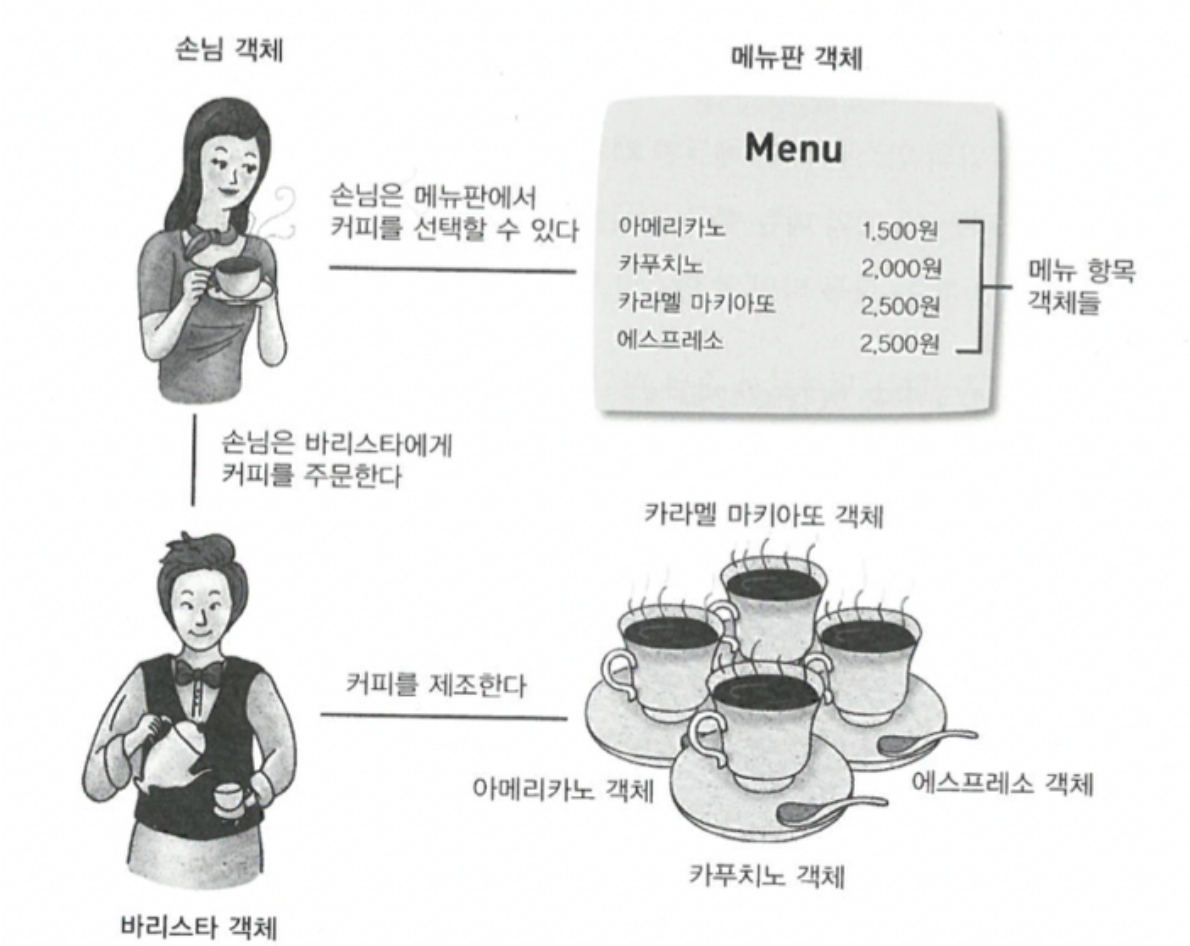
- 이에 반해 객체지향은 **기능**을 중심으로 메서드를 구현



- 즉, **절차지향 프로그래밍**은 프로그램의 순서와 흐름을 먼저 세우고 필요한 자료구조와 함수들을 설계하는 방식이고, **객체지향 프로그래밍**은 반대로 자료구조와 이를 중심으로 한 모듈들을 먼저 설계한 다음에 이들의 실행순서와 흐름을 짜는 방식이다

객체

객체를 지향한다는 프로그래밍이라는 뜻에 **객체**란 무엇일까?
쉽게, 말 그대로 대상을 나타내는 단어라고 생각하자.

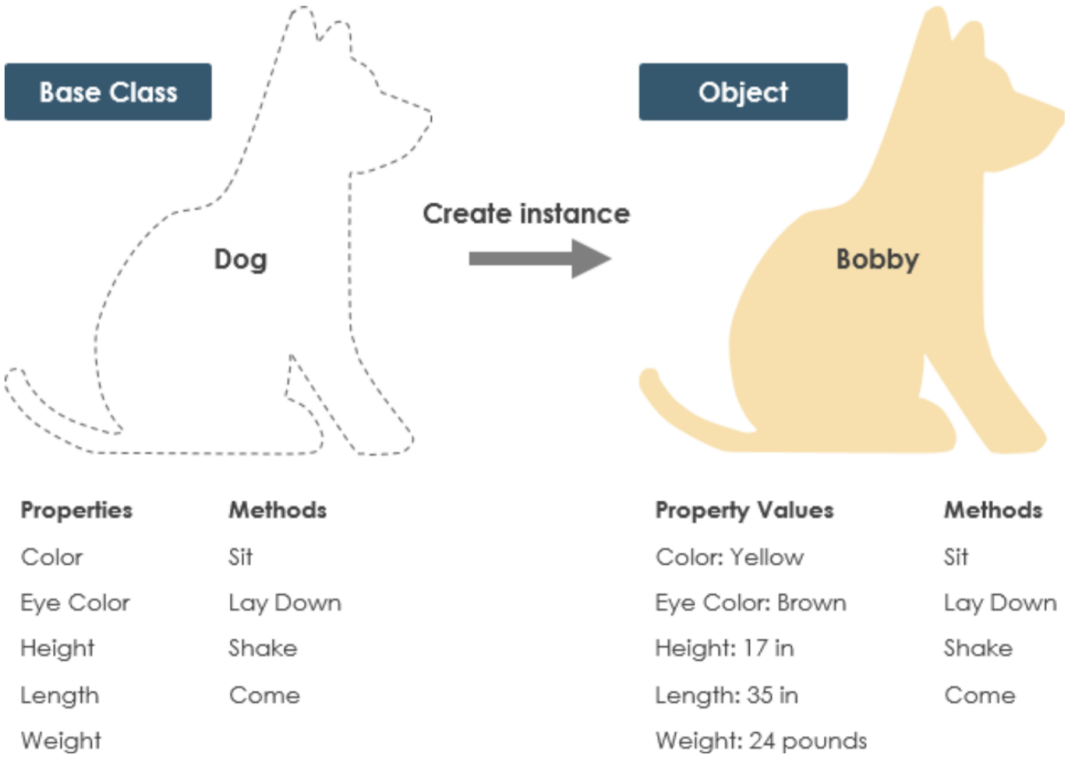


그림과 같이 메뉴판, 손님, 바리스타, 커피들처럼 존재하거나 생각할 수 있는 것들을 객체라고 본다.
이 객체들을 통하여 각 객체와그 객체들간의 관계를 설계하는 것이 객체 지향 프로그래밍이다.

그런데 **객체들 안에는 데이터(상태)와 메서드(행위)라는 것이 들어가있다.**
예를 들어 손님이고 생각해볼때 커피를 마시고 싶은 상태이면, 커피를 주문한다라는 행위가 들어가있다.

클래스

클래스는 일종의 `틀`이며, **객체는 클래스를 실체화 한 것**이라고도 표현할 수 있다.



OOP의 특징

추상화, 캡슐화, 상속성, 다형성이라는 4가지의 개념이 있다.

캡슐화	외부에서는 생성한 객체가 어떤 메서드와 필드로 일을 수행하는지 몰라도 된다.	예) getter/setter
상속	자식클래스는 부모클래스를 물려받으며 확장 가능하다. (재사용)	예) extends 키워드
추상화	공통된 속성/기능을 묶어 이름을 붙인다. (단순화 및 객체모델링)	예) 클래스, 객체
다형성	부모클래스와 자식클래스가 동일한 요청을 다르게 처리할 수 있다. (사용 편의성)	예) 오버라이딩, 오버로딩

캡슐화(Encapsulation)

객체 상태를 나타내는 속성 정보를 **private하게 관리 하는 것**.
직접적으로 속성 정보를 변경하는것 대신에, 메세지를 같은 요청을 보내어 그 응답에 의해 객체의 상태를 변경하게 된다.

캡슐화의 장점

1. 사용자가 불필요한 부분을 접근하지 못하게 하여, 객체의 오용을 방지 (개발자와 사용자가 다를 수 있음)

2. 객체의 내부가 바뀌어도 그 객체의 사용방법이 바뀌지 않음 (사용자는 객체 내부가 바뀌어도 같은 결과를 기대 할 수 있음.)
3. 객체 내부에서 사용되는 데이터가 바뀌어도 다른 객체에게 영향을 주지 않음 (각 객체들의 독립성이 유지)
4. 객체간의 결합도가 낮아지게 됨 (객체간의 결합도를 낮추는 일은 객체 설계의 기본 원칙)

```
class Person {
  private var location = "서울"

  private func doSomething1() {
    print("잠시 휴게소에 들려서 딴짓")
  }
  private func doSomething2() {
    print("한숨 자다 가기")
  }

  func currentLocation() {
    print("====")
    print("현재 위치 :", location)
    print("====")
  }

  func goToBusan() {
    print("서울을 출발합니다.")
    doSomething1()
    doSomething2()
    print("부산에 도착했습니다.")
    location = "부산"
  }
}

// location 속성
// goToBusan()을 메세지 보냈을때 내부적으로 어떤 추가 작업을 실행
// location 서울에서 부산으로 변경
// currentLocation과 goToBusan 메서드 외에는 외부로 노출되면 안됨.
// 즉, 요청자는 무엇을 할지만 알면 되고 어떻게 될지는 몰라도 됨

let A = Person()
A.currentLocation()
A.goToBusan()
A.currentLocation()
```

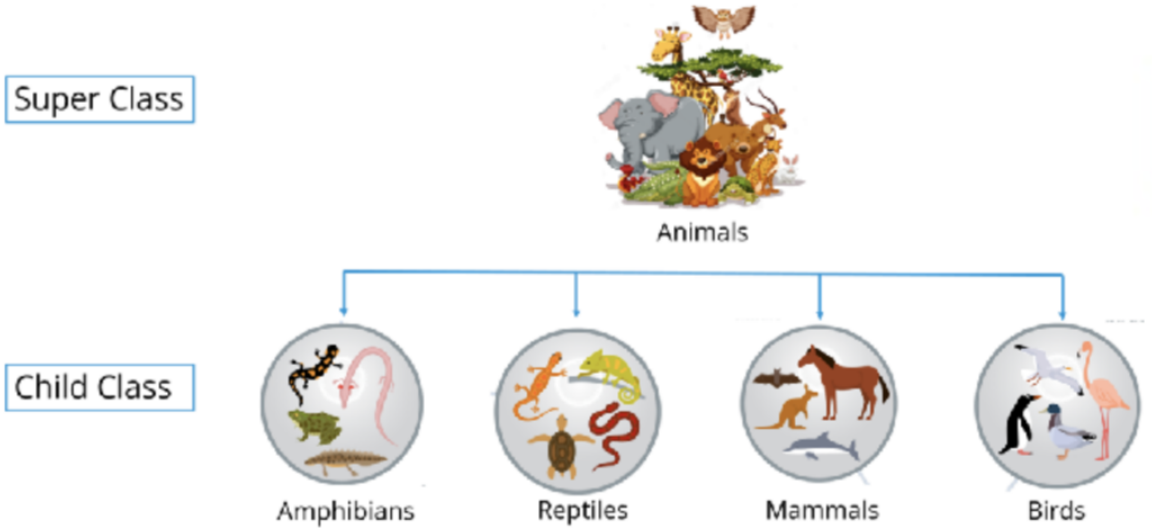
- 변하기 쉬운 것과 변하기 어려운 것
  - private
    - 변하기 쉬운 것은 감춘다!
    - 외부에서 변해도 영향을 받지 않는다.
    - Ex) 멤버 변수, 자료구조
  - public
    - 변하기 어려운 것은 드러낸다!
    - 변하기 어려우므로 외부에서 사용하는데 변경될 일이 적다.
      Ex) Stack의 관련 메서드들 (push, pop의 기능)
- getter/setter

- 은닉화된 변수에 접근 가능하게 해주는 것

```
class Student {
    //은닉된 멤버변수 --> 현재 블록안에서만 접근 가능함
    private String name;
    private int age;

    //은닉된 멤버 변수에 값을 넣는 방법 --> 메소드를 사용
    public void setName(String name){    //set 함수 setter
        this.name = name;
    }
    public void setAge(int age){
        this.age = age;
    }
    //은닉된 멤버변수의 값을 읽는 방법
    public String getName(){      //get 함수 getter
        return name;
    }

    public int getAge(){
        return age;
    }
}
public class Exam3_getter_setter {
    public static void main(String[] args){
        Student s = new Student();
        s.setName("감성공대생");
        s.setAge(20);

        String name= s.getName();
        System.out.println("이름 : "+name);
        int age = s.getAge();
        System.out.println("나이 : " + age);
    }

}
```

상속성(Inheritance)

하나의 클래스의 특징(부모 클래스)을 다른 클래스가 물려받아 그 속성과 기능을 동일하게 사용하는 것
재사용과 확장에 의미가 있다 (상속은 수직확장, Extension은 수평 확장)



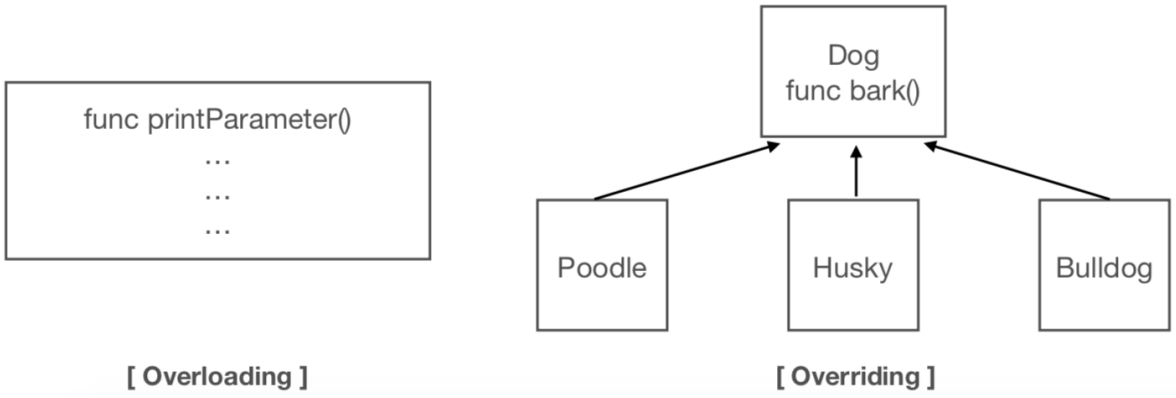
추상화(Abstraction)

대상의 불필요한 부분을 무시하며 복잡성을 줄이고 목적에 집중할 수 있도록 단순화 시키는 것이다.**즉, 사물들 간의 공통점만 취하고 차이점을 버리는 일반화를 통한 단순화라고 볼 수 있다**

> 관심 영역 = 도메인 = 컨텍스트
> 추상화 = 모델링 = 설계

다형성(Polymorphism)

다형성이란 쉽게, 다양한 형태로 나타날 수 있는 형태라고 볼 수 있다. 동일한 요청에 대해 각각 다른 방식으로 응답할 수 있도록 만드는 것. 자바에서는 이와 같은 다형성의 방식으로 오버라이딩(Overriding), 오버로딩(Overloading)을 지원한다.



상속성(Inheritance)

- 객체지향에서 상속이라 함은 상위클래스의 모든 것이 하위클래스에게 전달되는 것을 뜻함

- 그러나 상위클래스의 멤버변수와 멤버함수 중, private로 접근 제한이 된 경우에는 하위클래스로 전달 x

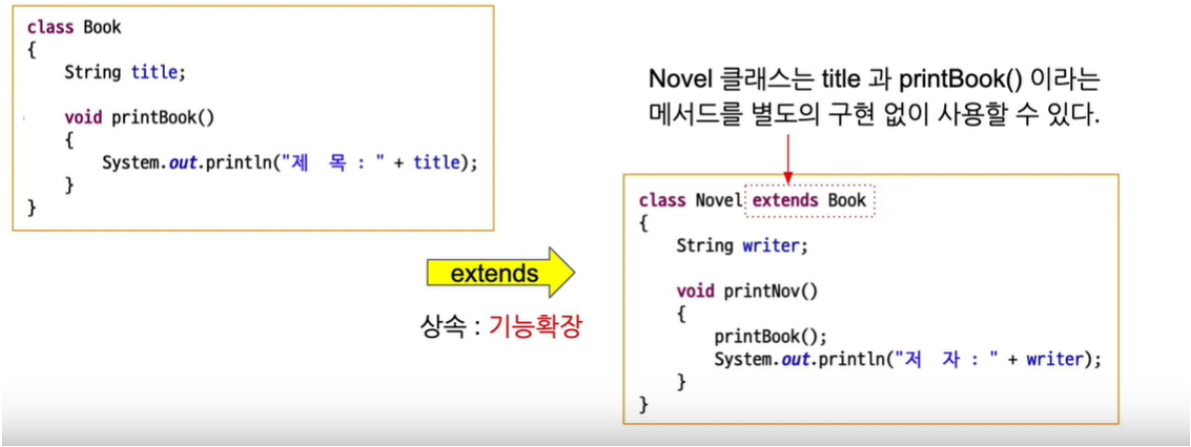
상속의 장점

1. 클래스 간의 체계화된 전체 계층 구조 파악 쉬움
2. **재사용성 증대** : 기존 클래스에 있는 것 재사용 가능
3. **확장 용이** : 새로운 클래스, 데이터, 메서드 추가 쉬움
4. 유지보수 용이 : 데이터와 메서드를 변경할 때 상위에 있는 것만 수정하여 전체적으로 일관성 유지 가능

상속의 구현

서브 클래스를 만들기 위해서는 **extends** 사용

- 자바에서 여러 개의 클래스를 동시에 상속하는 **다중 상속은 허용되지 않음**

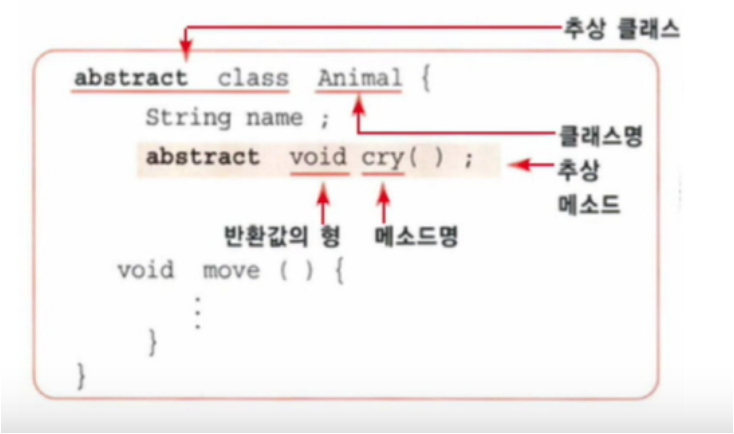



```
class Book {
    String title;

    void printBook() {
        System.out.println("제 목 :" +title);
    }
}

class Novel extends Book {
    String writer;

    void printNov() {
        printBook();
        System.out.println("저 자 :" + writer);
    }
}


class Magazine extends Book {
    int day;

    void printMag() {
        printBook();
```

```
        System.out.println("발매일 :" + day + "일");
    }
}

public class Bookshelf {

    public static void main(String[] args) {
        Novel nov = new Novel();
        nov.title = "홍길동전";
        nov.writer = "허균";

        Magazine mag = new Magazine();
        mag.title = "월간 자바";
        mag.day = 20;

        nov.printNov();
        System.out.println();
        mag.printMag();
    }

}
>>  제 목 :홍길동전
    저 자 :허균

    제 목 :월간 자바
    발매일 :20일
```

추상화(Abstraction)

- 처리 내용을 기술하지 않고, 호출하는 방법만을 정의한 메서드를 **추상 메서드**라고 함
- 추상 메서드를 한 개라도 가진 클래스를 **추상 클래스**라 한다.
- 추상 메서드와 추상 클래스는 **abstract**라는 수식자를 사용하여 다음과 같이 정의한다.



추상 클래스와 오버라이딩

추상클래스를 상속하는 클래스는 추상메서드를 반드시 **오버라이딩**해서 구현해야 함

- 기능이 없으면 동작을 할 수 없기 때문에 반드시 오버라이딩 해야 함
- **오버라이딩**이란?
  - 오버라이딩이란 상속된 메서드와 동일한 이름, 동일한 인수를 가지는 메서드를 정의하여 메서드를 덮어쓰는 것이다. 반환값의 형도 같아야 한다.
  - 오버라이드는 하위클래스에서 상속 받은 메서드를 단순히 재사용하지 않고 재정의하여 다른 연산을 수행하고 싶을 때 사용. 즉, 하위클래스에서 상위클래스의 특정 메서드를 다시 정의하는 것
    - 기능의 변경
    - 기능의 추가
  - 오버라이드는 많은 객체지향 디자인 패턴에 매우 자주 사용
  - 추상 클래스와 합쳐져서 객체지향 방법론의 장점으로 많이 거론되는 '확장성'의 실현을 가능케 함.
  - 오버라이딩과 오버로딩 구분

    오버라이딩

    - 상속의 관계에서 발생
    - 위(부모)에서 아래(자식)으로 연결된다는 점에서 '오버**라**이딩'의 '**ㅏ**'로 위에서 아래로 긋는다는 이미지의 느낌으로 외운다.

    오버로딩

    - 한 클래스 내에서 동일한 이름의 메서드가 여러 개 존재
    - 모든 메서드는 수평적인 관계이므로 '오버**로**딩'의 '**ㅗ**'로 옆으로 긋는다의 이미지 느낌으로 외우기
  - 기능의 추가

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

    void printPet() {
        System.out.println("이름 : " + name);
        System.out.println("나이 : "+ age);
    }
}

class Dog extends Animal {
    String variety;

    // 함수의 오버라이딩
    void printPet() {
        super.printPet();
        System.out.println("종류:" + variety);
    }
}
public class Pet {

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.name = "진돌이";
```

```
        dog.age = 5;
        dog.variety = "진돗개";
        dog.printPet();

    }

}

>> 이름 : 진돌이
   나이 : 5
   종류:진돗개
```

- 기능의 변경

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

    void printPet() {
        System.out.println("이름 : " + name);
        System.out.println("나이 : "+ age);
    }
}

class Dog extends Animal {
    String variety;

    // 함수의 오버라이딩
    void printPet() {
        // super.printPet();
        System.out.println("종류:" + variety);
    }
}
public class Pet {

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.variety = "진돗개";
        dog.printPet();

    }

}

>> 종류:진돗개
```

- 추상 클래스와 오버라이딩

```java
abstract class Animal {
    int age;
    abstract void cry();
}

class Dog extends Animal {
    void cry() {
        System.out.println("멍멍");
    }
}

class Cat extends Animal {
    void cry() {
        System.out.println("야옹");
    }
}
```

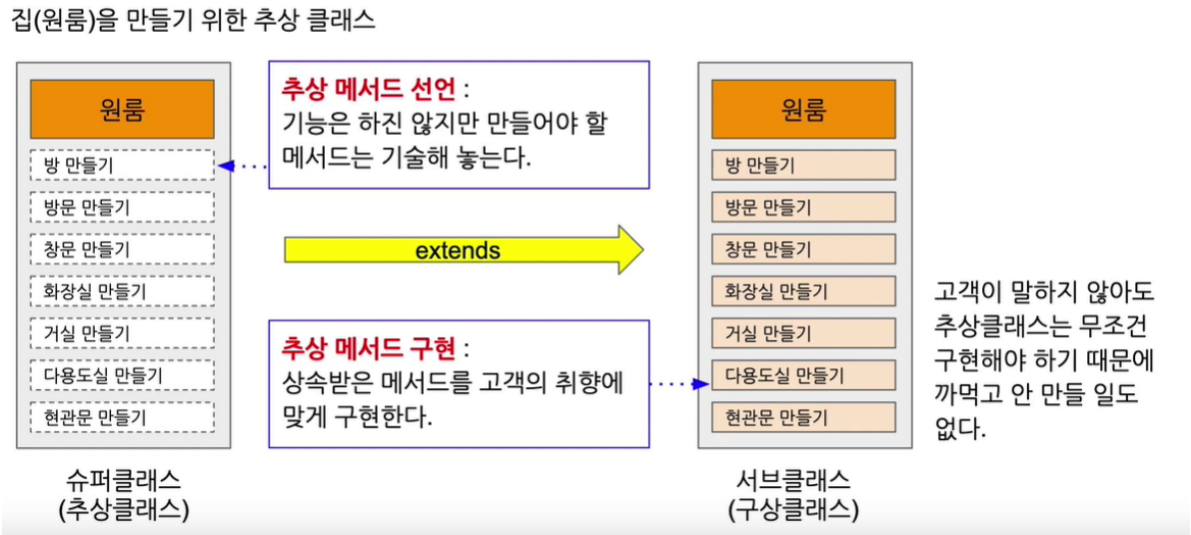
추상 메서드 선언

추상 메서드를 자기 상황에 맞게 구체화

추상 메서드를 자기 상황에 맞게 구체화

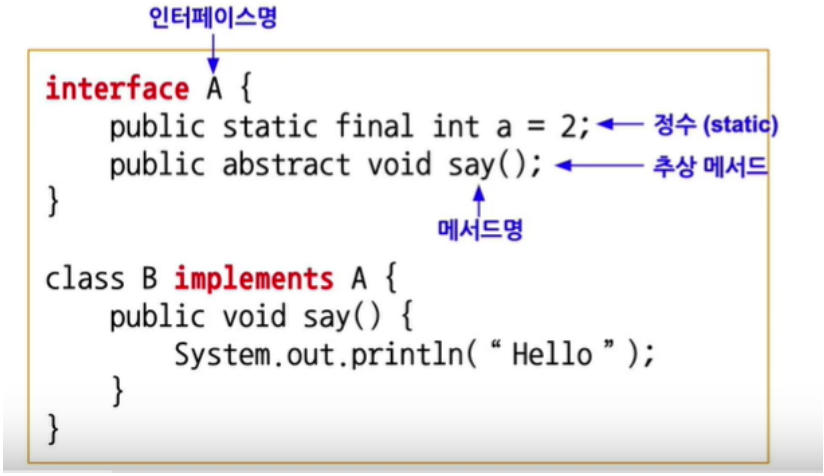
추상 클래스의 이해와 사용

추상클래스는 무조건 구현해야 하기 때문에 까먹고 안 만들 일을 방지



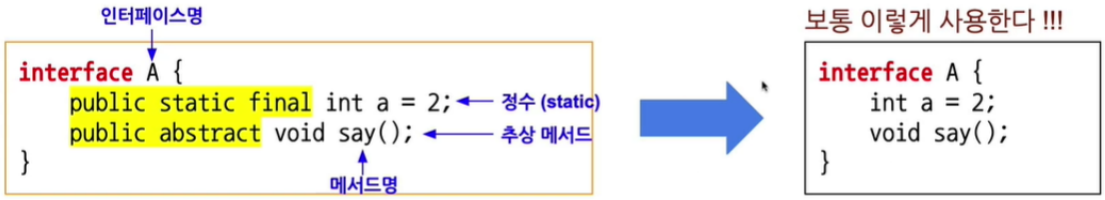
ㄴ 인터페이스

- 인터페이스란 상속 관계가 아닌 클래스에 (추상클래스처럼) 기능을 제공하는 구조
  - 자바는 다중상속 불가능
- 클래스와 비슷한 구조이지만, 정의와 추상 메서드만이 멤버가 될수 있다는 점이 다르다
- 클래스에서 인터페이스를 이용하도록 하게 하는 것을 '**인터페이스의 구현**'이라고 한다.
- 인터페이스를 구현하기 위해서는 **implements**를 사용



ㄴ

- 인터페이스의 정의는 다음과 같이 수식자를 생략할 수 있음



또한 인터페이스는 몇 개라도 구현할 수 있다.

인터페이스의 상속

- 인터페이스도 클래스처럼 상속할 수 있다.
- 인터페이스도 클래스처럼 상속할 수 있다.

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

interface B extends A {
    void goodbye();
}
```

복수의 인터페이스를 상속하여 새로운 인터페이스를 만들 수 있다.

- extends와 implements
  - 다른 클래스를 상속하고 또한 인터페이스를 구현하는 경우에 다음과 같이 정의한다.
  - extends를 먼저 기술
- 실습 코드

```java
package step1;
//인터페이스는 상속 관계가 아닌 클래스에
//기능을 제공하는 구조.
//클래스와 비슷한 구조이지만,
//정의와 추상 메서드만이 멤버가 될 수 있다.

interface Greet {
    void greet();
}

interface Bye {
    void bye();
}

class Morning implements Greet, Bye {

    public void bye() {
        System.out.println("안녕히 계세요.");
    }

    public void greet() {
        System.out.println("안녕하세요.");
    }
}


public class Meet {

    public static void main(String[] args) {
        Morning morning = new Morning();
        morning.greet();
        morning.bye();
    }

}
```

```
>> 안녕하세요.
   안녕히 계세요.
```

다형성 (Polymorphism)

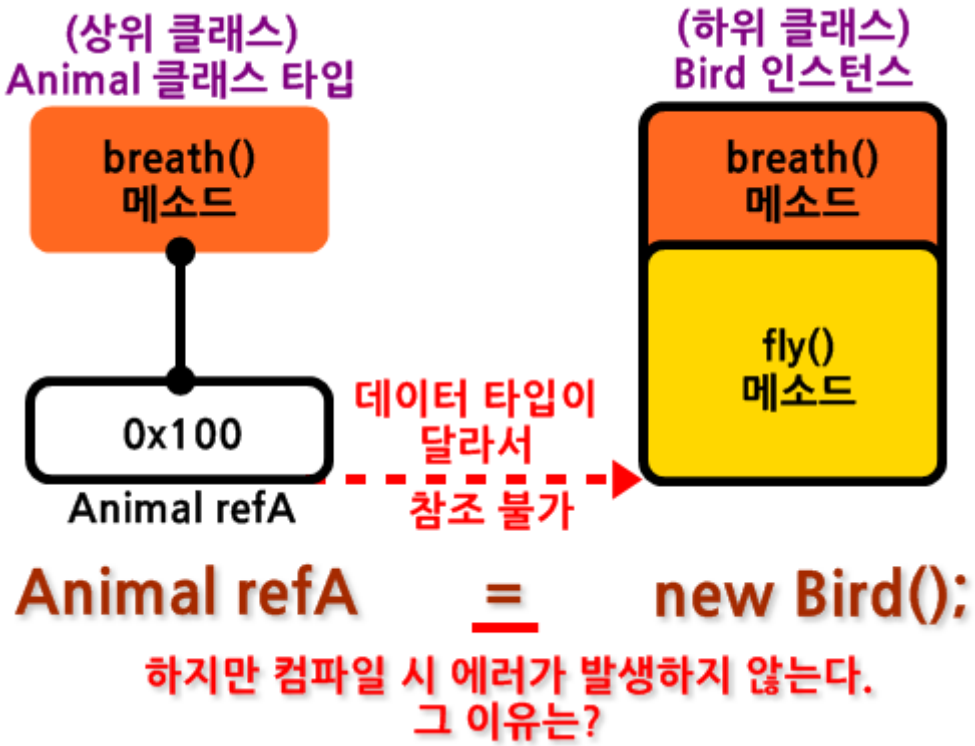
- **다형성(polymorphism)이란**

> 1. 여러가지 형태를 가질 수 있는 것.
> 2. 하나의 메소드나 클래스가 있을 때 이것들이 다양한 방법으로 동작하는 것. 대표적으로 오버로딩,오버라이딩이 있다.
> 3. 하나의 참조변수로 여러 타입의 객체를 참조할 수 있는 것.

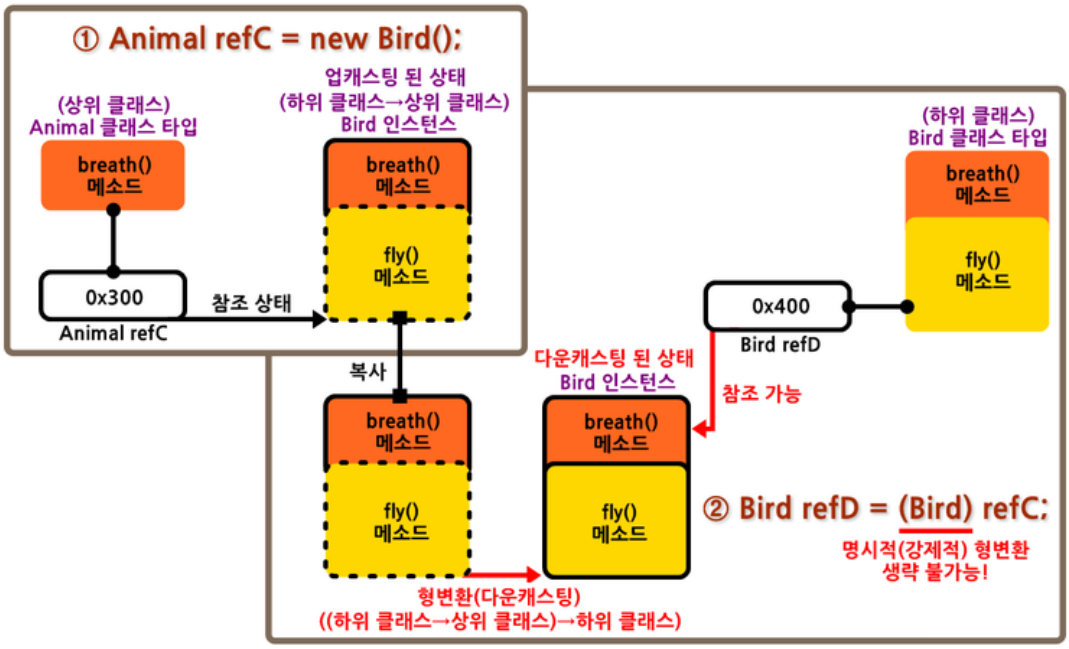
- **객체지향 언어**의 **다형성**의 시작은 **상속**에서 시작
- **한 타입의 참조 변수는 여러 타입의 객체를 참조할 수 있다.**
    - 즉, 조상타입의 참조변수로 자손타입(상속관계)의 객체를 다룰 수 있는 것
- 지금까지의 인스턴스 생성은

```
Animal a = new Animal();

Bird b = new Bird();
```

위와 같은 방식으로 진행했었다. 이 경우는 오류가 발생하지 않는다. 하지만 상위 클래스 타입으로 하위 클래스의 인스턴스를 생성하면 어떻게 될까?



    - 상위 클래스인 Animal 클래스는 상속을 통해 하위 클래스인 Bird 클래스와 관계가 맺어졌다. 그래서 Bird 클래스의 인스턴스인 breath() 메소드에 대한 접근 권한을 가지고 있다.
    - 그렇기 때문에 상위 클래스 타입의 참조 변수에 하위 클래스 인스턴스의 참조값이 대입연산이 가능
    - 다만 이 경우 문제가 있다. 대입연산은 양쪽의 데이터 타입이 같아야 하기 때문에 자바 컴파일러는 컴파일 시 『(Animal)』이라는 코드를 삽입해 **형변환**을 발생시킨다.
        - 기본 변수의 형변환에서 범위가 큰 자료형을 보다 범위가 작은 자료형으로 형변환 시에는 문제가 발생했다.
        - 하지만 참조 변수의 형변환은 기본 변수와는 개념이 다르다.
        - 이유는 참조 변수에는 데이터가 저장되는 것이 아닌 참조값(주소값)이 저장되는 것이기 때문이다. 기본 변수의 형변환과는 다르게 참조 변수의 형변환은 인스턴스 자체를 변화시키는 것이 아닌 하위 클래스에만 속한 메소드에 대한 접근 권한을 제한하는 것
        - 그래서 이러한 경우 묵시적인 형변환이 발생하도록 규칙이 정해진 것이다. 이 묵시적 형변환은 하위 클래스가 상위 클래스 타입으로 변화된다고 하여 업캐스팅이라고 한다
- 다운캐스팅
    - 업캐스팅 된 클래스를 다시 본래의 상태로 변화시키는 다운캐스팅



- 위의 이미지에서 보듯 refC는 업캐스팅되어 Animal 클래스 타입으로 변화시켰기 때문에 Bird 클래스 타입인 refD와는 클래스 타입이 다르기 때문에 대입연산이 이루어질 수 없다.
- 이 경우는 업캐스팅 된 상태의 Bird 인스턴스를 다시 하위 클래스 타입으로 복구시키는 다운캐스팅 기법을 이용해 대입연산을 가능하게 만들 수 있다.
- 다운캐스팅은 업캐스팅처럼 묵시적 형변환이 아닌 명시적 형변환이다. 다시말해 생략이 불가능한 강제적 형변환이라는 뜻이다.
- 또한 다운캐스팅의 목적은 업캐스팅 된 클래스를 원래의 데이터 타입으로 되돌리는데 있기 때문에 업캐스팅된 클래스만을 다운캐스팅의 대상으로 허용한다.

- 코드 실습

```
abstract class Calc {
    int a = 5;
    int b = 6;

    abstract void plus();
}

class MyCalc extends Calc {
    void plus()  { System.out.println(a + b); }
    void minus() { System.out.println(a - b); }
}

public class Polymorphism1 {

    public static void main(String[] args) {
        MyCalc myCalc1 = new MyCalc();
        myCalc1.plus();
        myCalc1.minus();

        // 하위클래스 객체를 상위 클래스 객체에 대입
        Calc myCalc2 = new MyCalc();
        myCalc2.plus();
        // 다음 메서드는 설계도에 없다. 사용할 수 없다.
        //myCalc2.minus();
    }
```

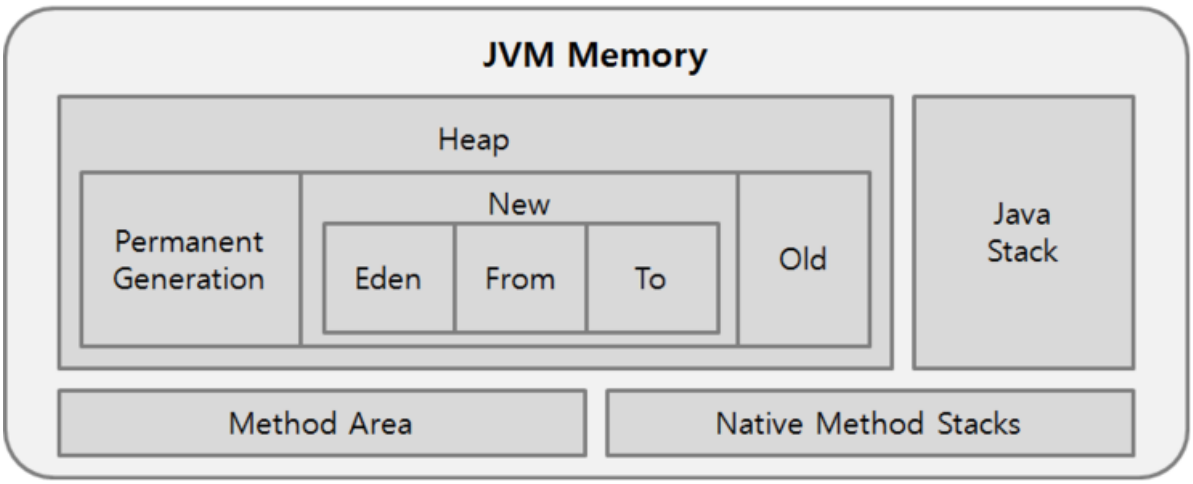
```
}

>> 11
   -1
   11
```

메모리 모델

JVM은 운영체제로부터 할당받은 메모리 공간을 이용해 자기 자신도 실행하고, 자바 프로그램도 실행한다.

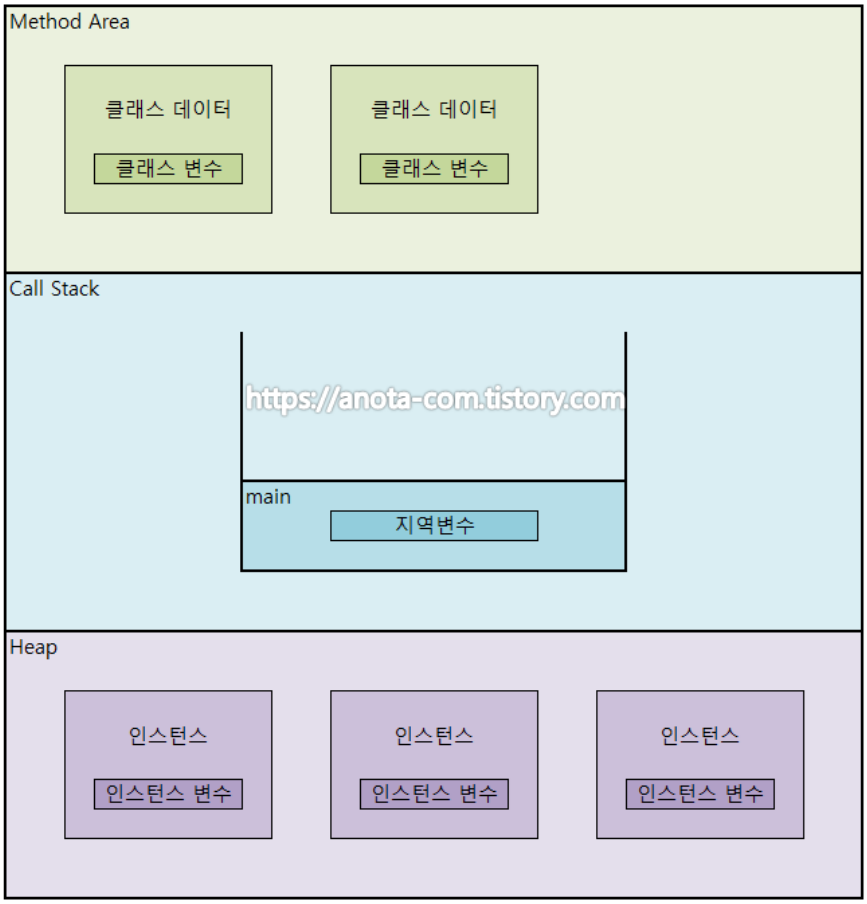
** JVM이란?

자바가상머신으로 자바 바이트 코드를 실행할 수 있는 주체이다. 운영체제 위에서 동작하는 프로세스로 자바 코드를 컴파일해서 얻은 바이트 코드를 해당 운영체제가 이해할 수 있는 기계어로 바꿔 실행시켜주는 역할



- 바이트 코드(Bytecode) - **자바 가상머신에 의해서 실행되는 코드**
    - 고급 언어로 작성된 소스 코드를 가상머신이 이해할 수 있는 중간 코드로 컴파일한 것 / 가상머신은 이 바이트코드를 각각의 하드웨어 아키텍처에 맞는 기계어로 다시 컴파일

JVM 주요 영역 3가지



- 메서드 영역
  - 메서드의 자바 바이트코드는 JVM이 구분하는 메모리 공간 중에서 메서드 영역에 저장
  - 어떤 클래스가 사용되었을 때, 해당 클래스의 클래스파일(*.class)을 읽어서 분석하여 클래스에 대한 정보(클래스 데이터)를 저장
  - 클래스의 클래스 변수(class variable)도 함께 생성
  - static으로 선언된 클래스 변수도 메서드 영역에 저장

    ** static 참고: https://mangkyu.tistory.com/47
  - ***이 영역에 저장된 내용은 프로그램 시작 전에 로드되고 프로그램 종료 시 소멸***

- 스택 영역
  - 매개변수, 지역변수가 할당되는 메모리 공간
  - 메서드 호출시 스택에 호출된 메서드를 위한 메모리가 할당되며, 메서드가 작업을 수행하는 동안 지역변수(매개변수 포함) 들과 연산의 중간결과등을 저장하는데 사용됨
  - 프로그램 실행 도중 임시로 할당됐다가 바로 이어서 소멸되는 특징이 있는 변수 할당
  - ***이 영역에 저장된 변수는 해당 변수가 선언된 메서드 종료 시 소멸***

- 힙 영역
  - 인스턴스(객체)가 생성되는 메모리 공간
  - JVM에 의한 메모리 공간의 정리 이루어지는 공간
  - 할당은 프로그래머 / 소멸은 JVM이
  - 참조변수에 의한 참조 안 이뤄진 인스턴스가 소멸 대상 - Garbage collection

스택 흐름도

```
class CallStackTest {
    public static void main(String[] args) {
        firstMethod(); // static 메서드는 객체 생성 없이 호출됨
    }

    static void firstMethod() {
        secondMethod();
    }

    static void secondMethod() {
        System.out.println("WOW"); // 실행결과 WOW
    }
}
```