

0. 목차

#1.인강/0.자바/2.자바-기본

- 인프런 강의
- 저자: 김영한

전체 목차

- 1. 클래스와 데이터
- 2. 기본형과 참조형
- 3. 객체 지향 프로그래밍
- 4. 생성자
- 5. 패키지
- 6. 접근 제어자
- 7. 자바 메모리 구조와 static
- 8. final
- 9. 상속
- 10. 다형성1
- 11. 다형성2
- 12. 다형성과 설계
- 13. 다음으로

버전 수정 이력

2023-12-07

- int → Data 코드 오타 수정 (minsubrother님 도움)

2023-11-28

- 릴리즈

1. 클래스와 데이터

#1.인강/0.자바/2.자바-기본

- /프로젝트 환경 구성
- /클래스가 필요한 이유
- /클래스 도입
- /객체 사용
- /클래스, 객체, 인스턴스 정리
- /배열 도입 - 시작
- /배열 도입 - 리펙토링
- /문제와 풀이
- /정리

프로젝트 환경 구성

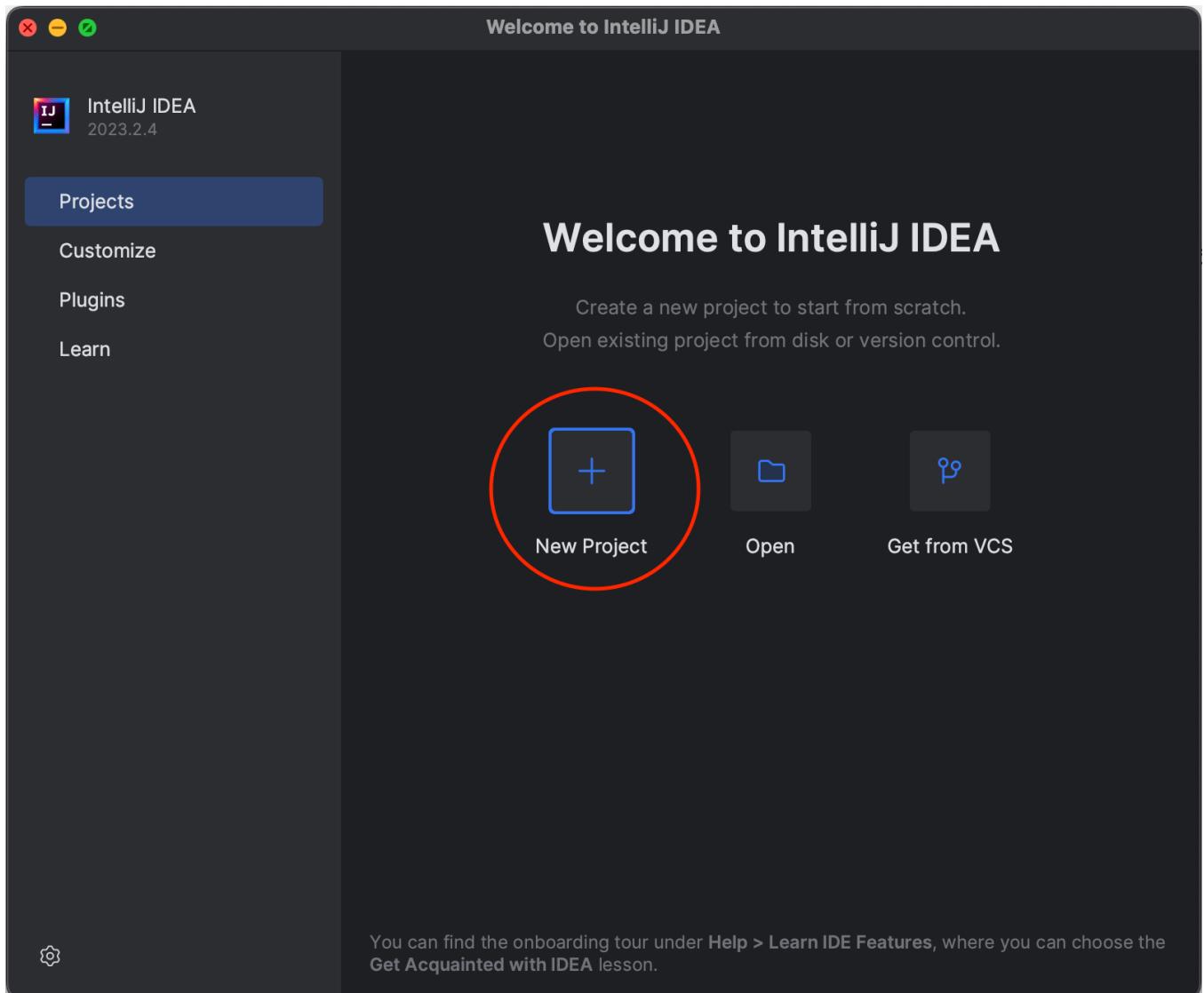
자바 입문편에서 인텔리제이 설치, 선택 이유 설명

프로젝트 환경 구성에 대한 자세한 내용은 자바 입문편 참고

여기서는 입문편을 들었다는 가정하에 자바 기본편 설정 진행

인텔리제이 실행하기

New Project



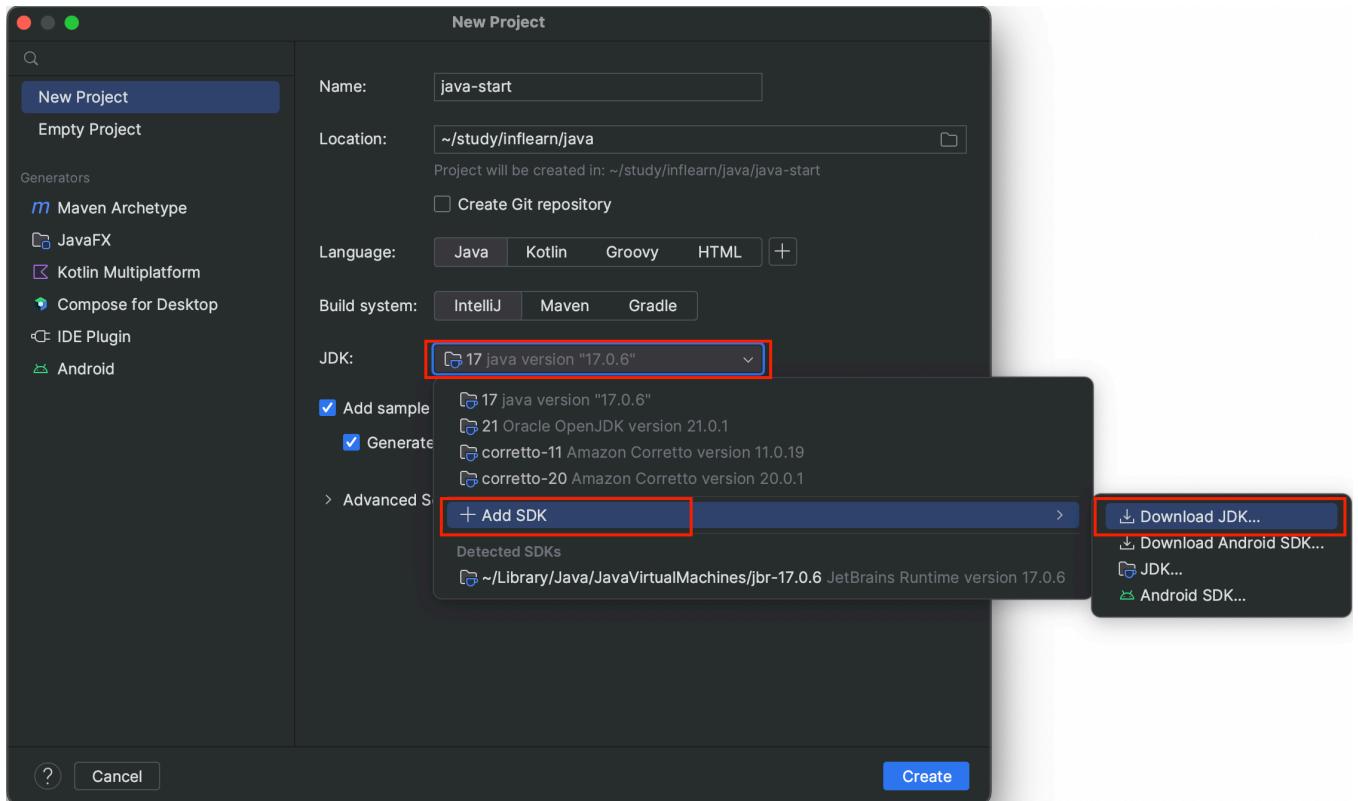
- New Project를 선택해서 새로운 프로젝트를 만들자

New Project 화면

- Name:
 - 자바 입문편 강의: java-start
 - 자바 기본편 강의: **java-basic**
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ
- JDK: 자바 버전 17 이상
- Add sample code 선택

JDK 다운로드 화면 이동 방법

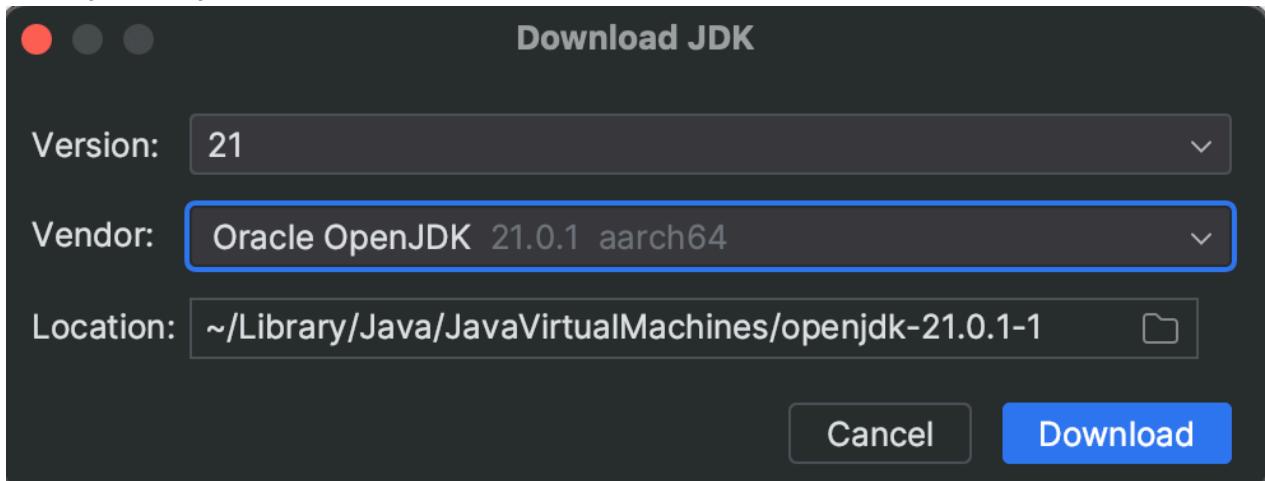
자바로 개발하기 위해서는 JDK가 필요하다. JDK는 자바 프로그래머를 위한 도구 + 자바 실행 프로그램의 묶음이다.



- **Name:**

- 자바 입문편 강의: java-start
- 자바 기본편 강의: **java-basic**

JDK 다운로드 화면



- Version: 21을 선택하자.
- Vendor: Oracle OpenJDK를 선택하자. 다른 것을 선택해도 된다.
 - aarch64: 애플 M1, M2, M3 CPU 사용시 선택, 나머지는 뒤에 이런 코드가 불지 않은 JDK를 선택하면 된다.
- Location: JDK 설치 위치, 기본값을 사용하자.

Download 버튼을 통해서 다운로드 JDK를 다운로드 받는다.

다운로드가 완료 되고 이전 화면으로 돌아가면 **Create** 버튼 선택하자. 그러면 다음 IntelliJ 메인 화면으로 넘어간다.

IntelliJ 메인 화면

The screenshot shows the IntelliJ IDEA interface with the following annotations:

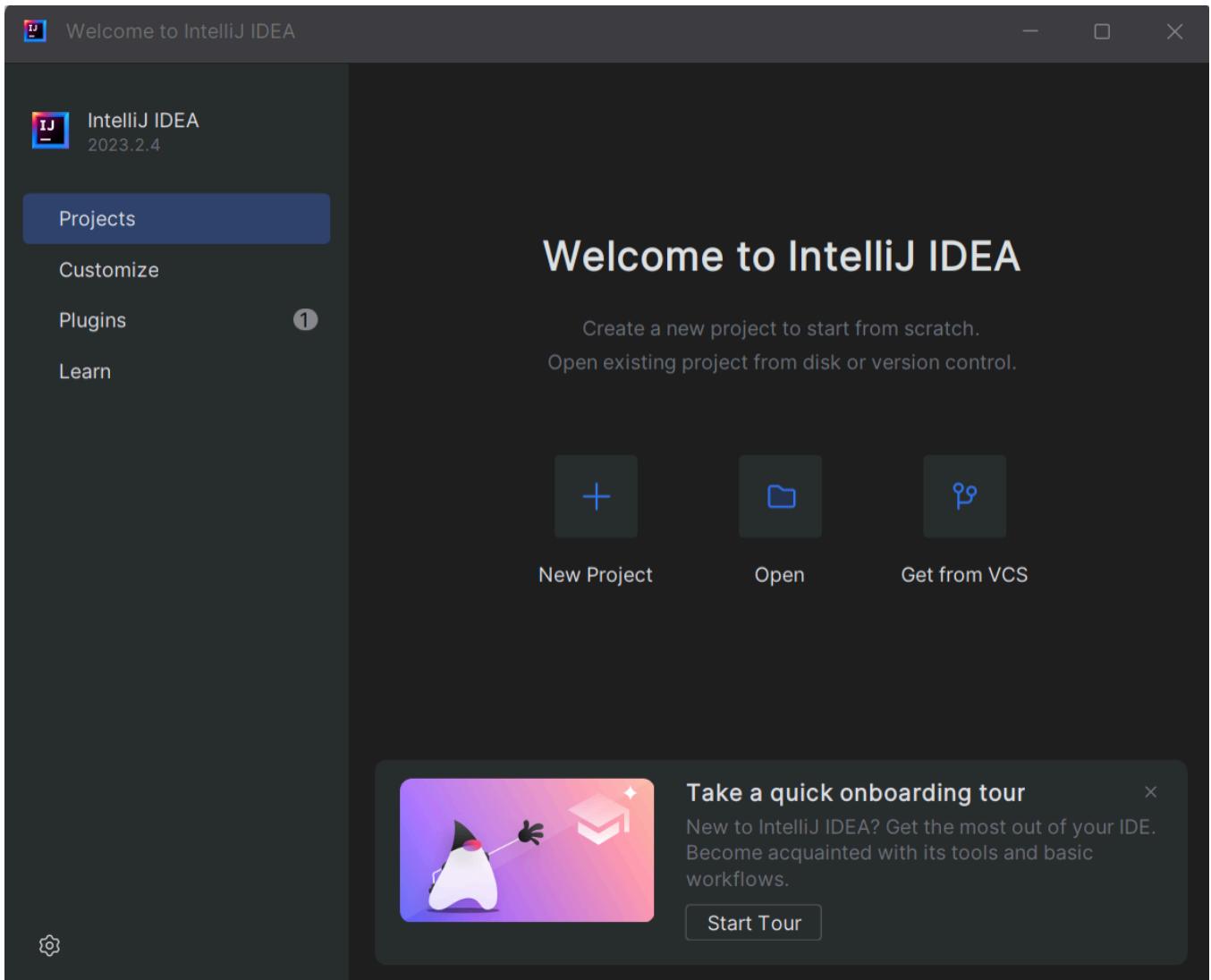
- A red box highlights the "Run" button (green arrow icon) in the gutter of the code editor.
- A green box highlights the "Run Main.main()" button (green arrow icon) in the toolbar.
- A red circle highlights the line of code `System.out.println("i = " + i);`.

```
// Press Shift twice to open the Search Everywhere dialog and type ⌘S
// then press Enter. You can now see whitespace characters in your code
public class Main {
    public static void main(String[] args) {
        // Press Opt+Enter with your caret at the highlighted text to
        // IntelliJ IDEA suggests fixing it.
        System.out.printf("Hello and welcome!");
    }
    // Press Ctrl+R or click the green arrow button in the gutter to
    // for you, but you can always add more by pressing Cmd+F8.
    for (int i = 1; i <= 5; i++) {
        System.out.println("i = " + i);
    }
}
```

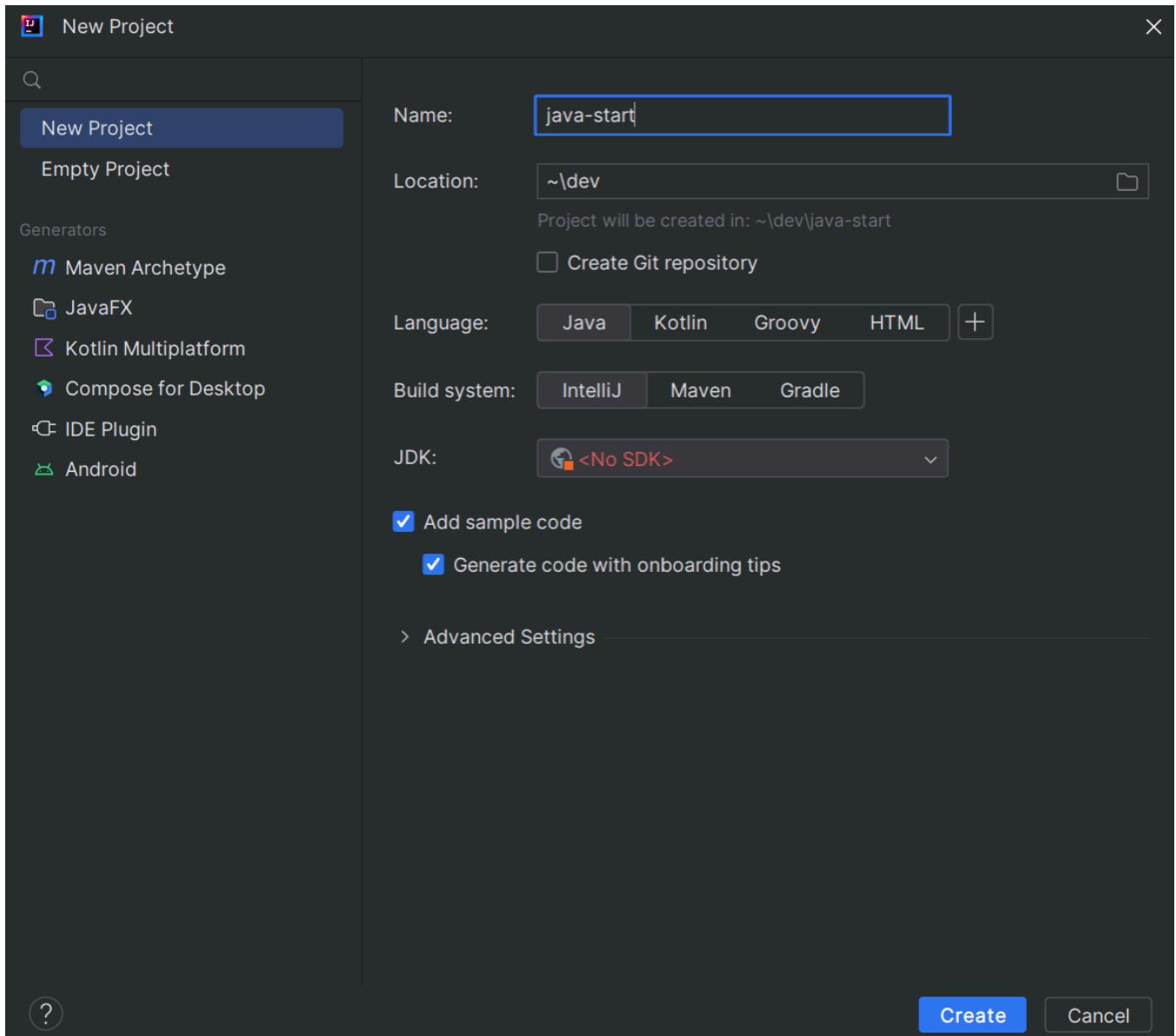
- 앞서 **Add sample code** 선택해서 샘플 코드가 만들어져 있다.
- 위쪽에 빨간색으로 강조한 초록색 화살표 버튼을 선택하고 **Run 'Main.main()'** 버튼을 선택하면 프로그램이 실행된다.

윈도우 사용자 추가 설명서

윈도우 사용자도 Mac용 IntelliJ와 대부분 같은 화면이다. 일부 다른 화면 위주로 설명하겠다.

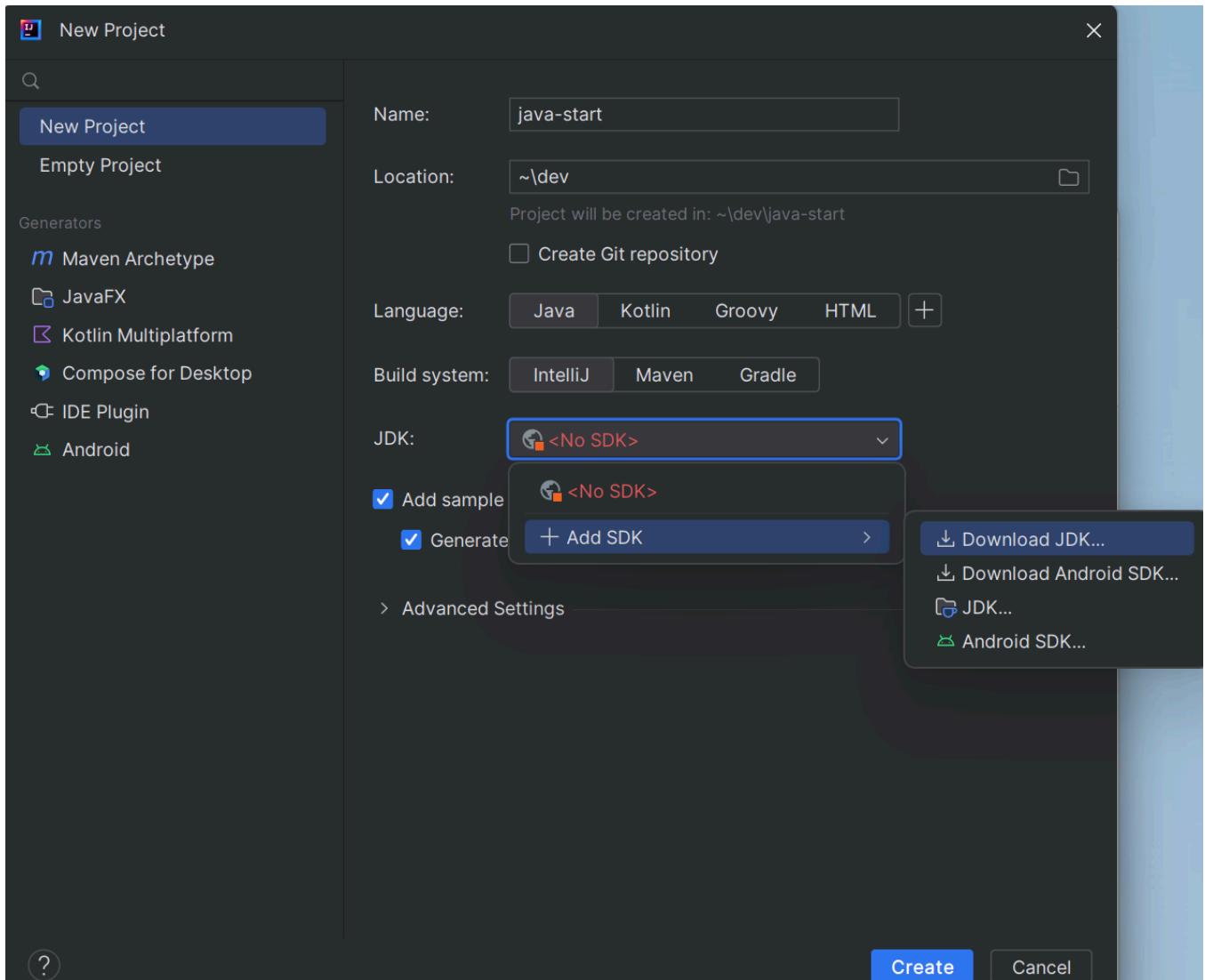


- 프로그램 시작 화면
- New Project 선택

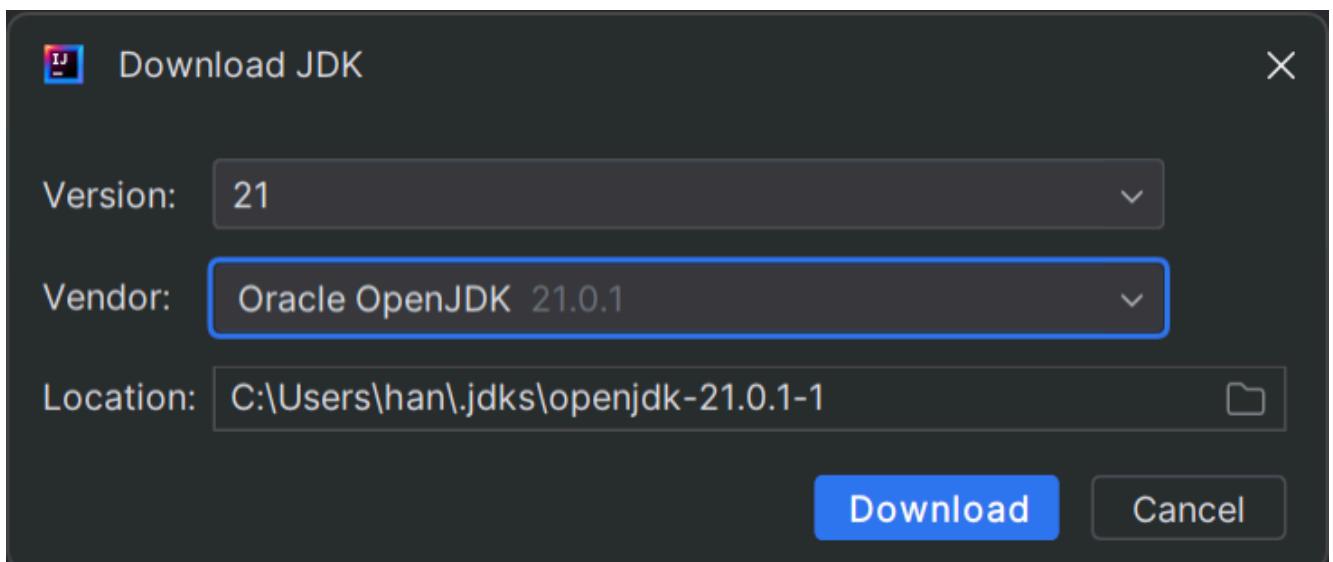


New Project 화면

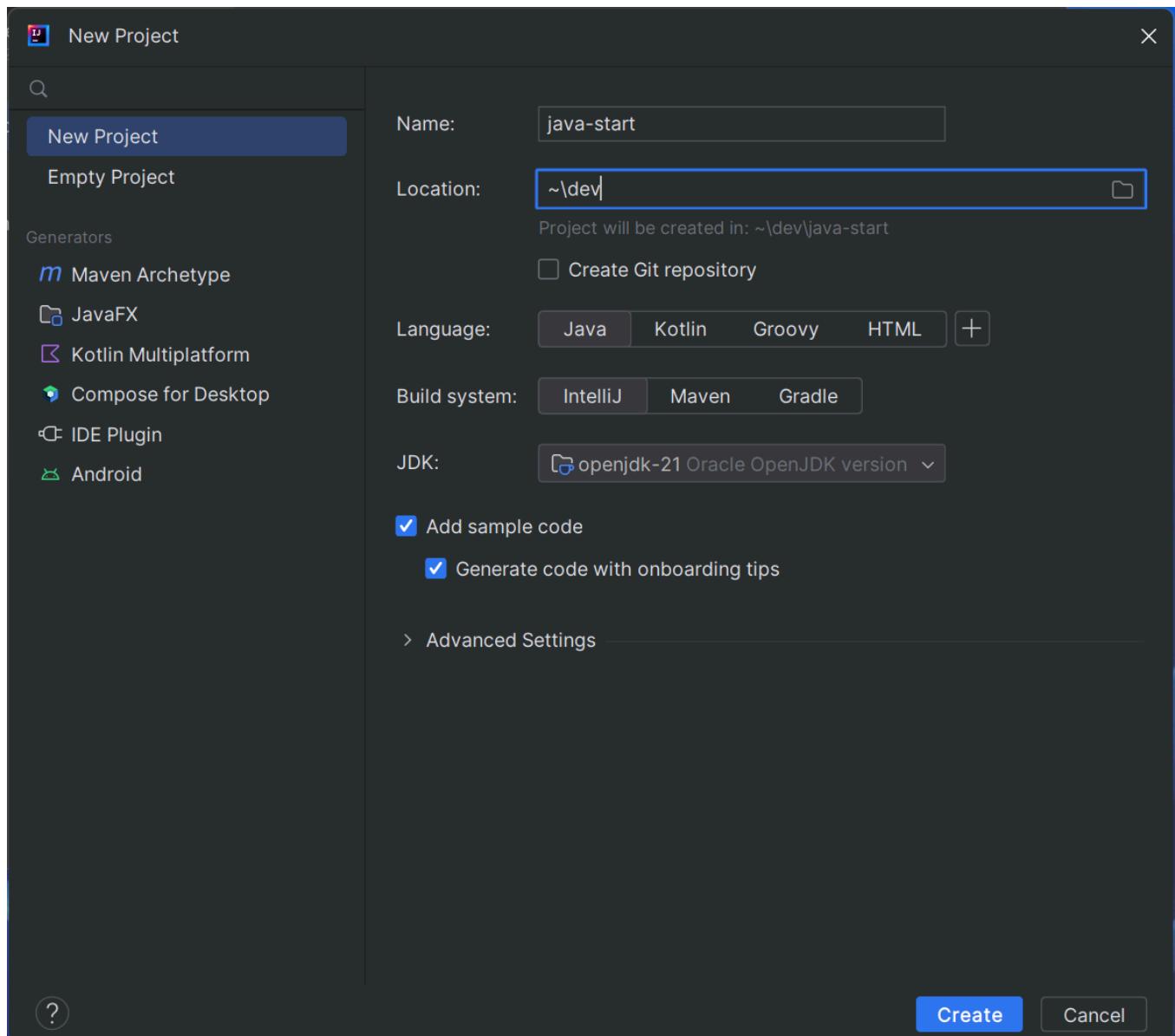
- **Name:**
 - 자바 입문편 강의: java-start
 - 자바 기본편 강의: **java-basic**
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ
- JDK: 자바 버전 17 이상
- Add sample code 선택



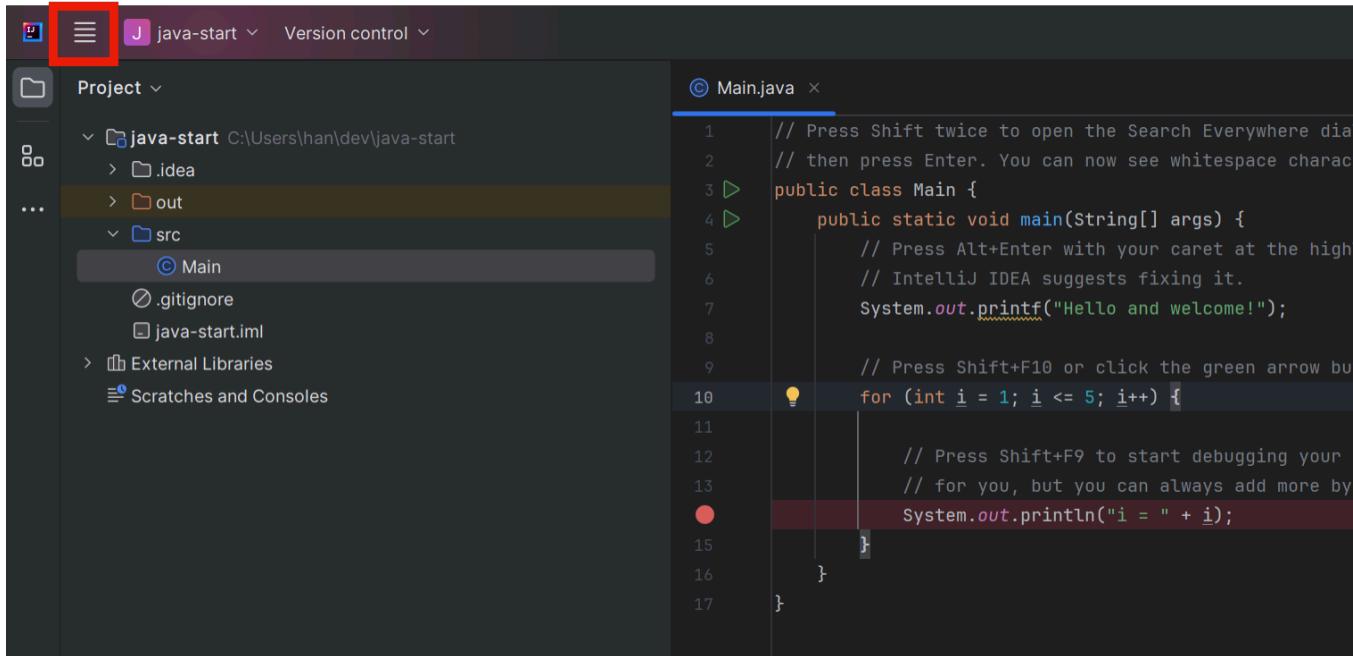
JDK 설치는 Mac과 동일하다.



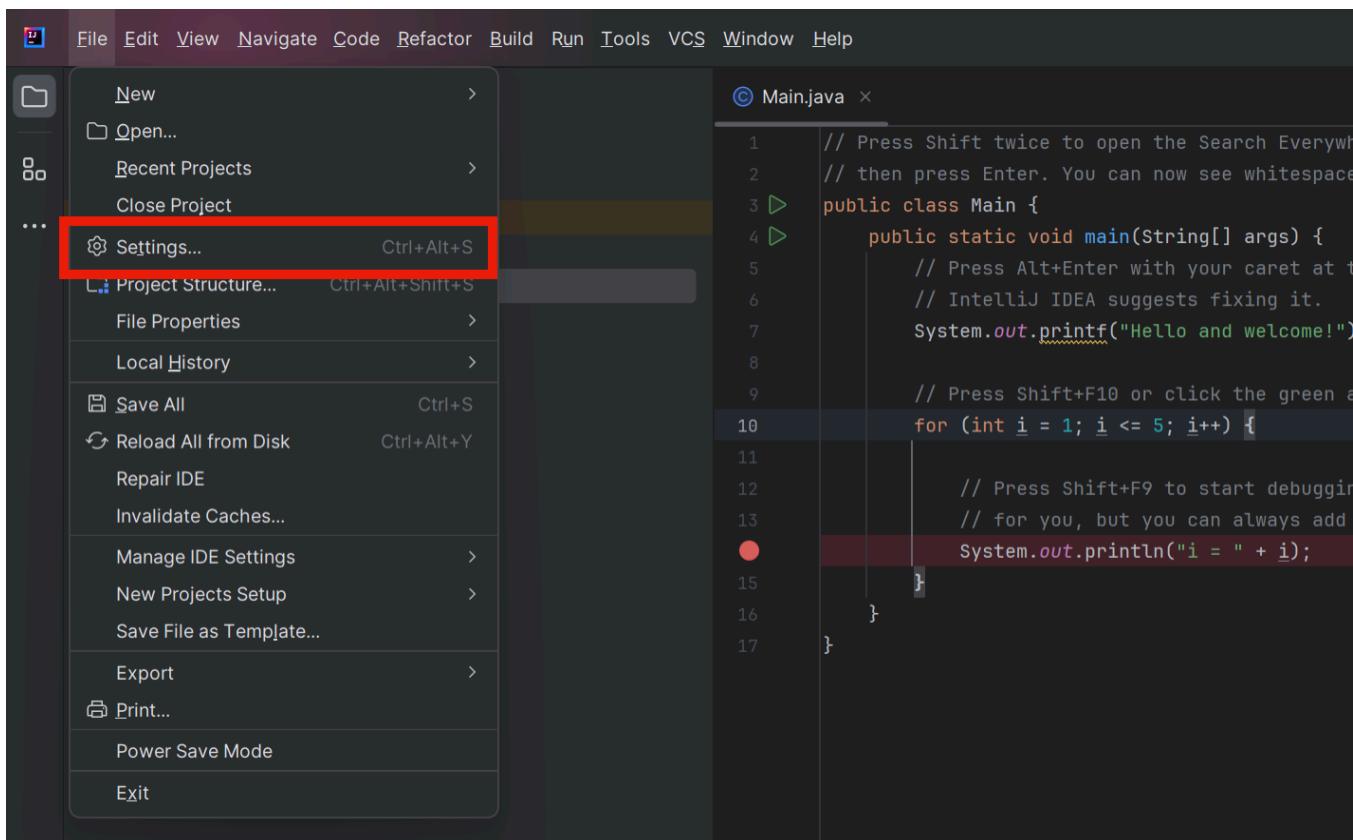
- Version: 21
- Vendor: Oracle OpenJDK
- Location은 가급적 변경하지 말자.



- New Project 완료 화면



- 원도우는 메뉴를 확인하려면 왼쪽 위의 빨간색 박스 부분을 선택해야 한다.



- Mac과 다르게 **Settings...** 메뉴가 **File**에 있다. 이 부분이 Mac과 다르므로 유의하자.

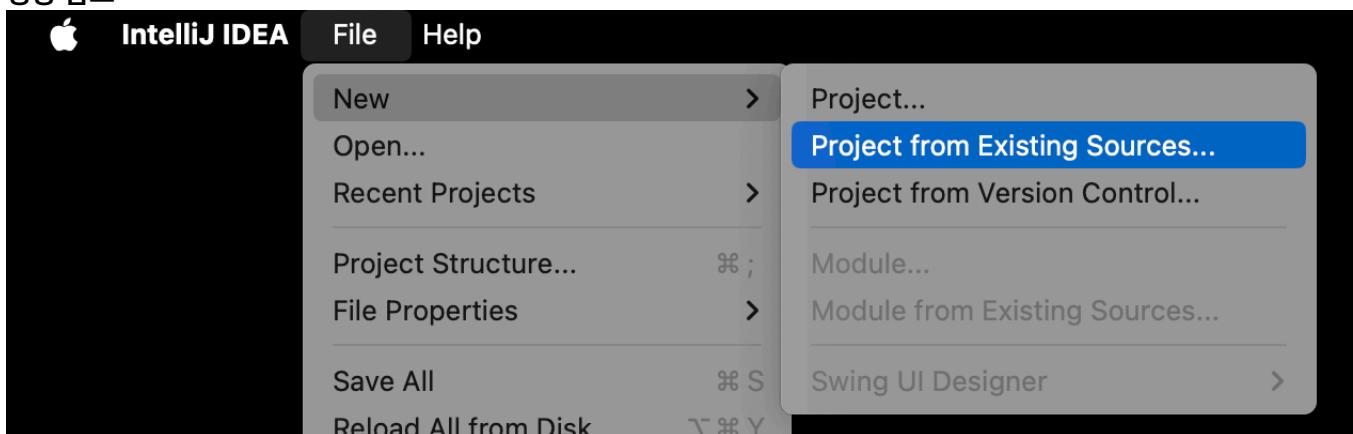
한글 언어팩 → 영어로 변경

- IntelliJ는 가급적 한글 버전 대신, 영문 버전을 사용하자. 개발하면서 필요한 기능들을 검색하게 되는데, 영문으로 된 자료가 많다. 이번 강의도 영문을 기준으로 진행한다.

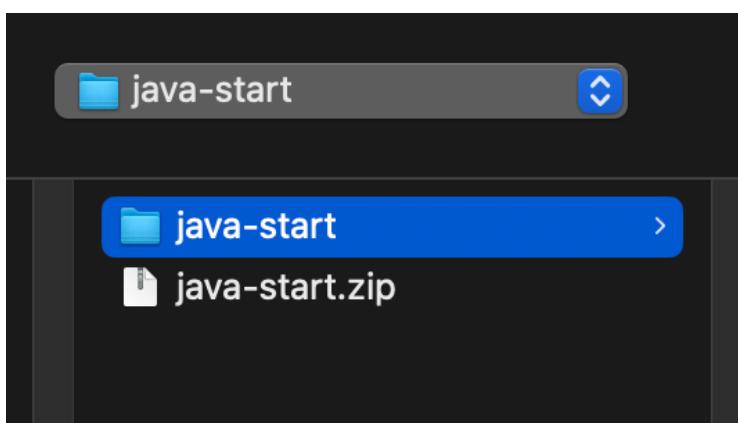
- 만약 한글로 나온다면 다음과 같이 영문으로 변경하자.
- Mac:** IntelliJ IDEA(메뉴) → Settings... → Plugins → Installed
- 윈도우:** File → Settings... → Plugins → Installed
 - Korean Language Pack 체크 해제
 - OK 선택후 IntelliJ 다시 시작

다운로드 소스 코드 실행 방법

영상 참고

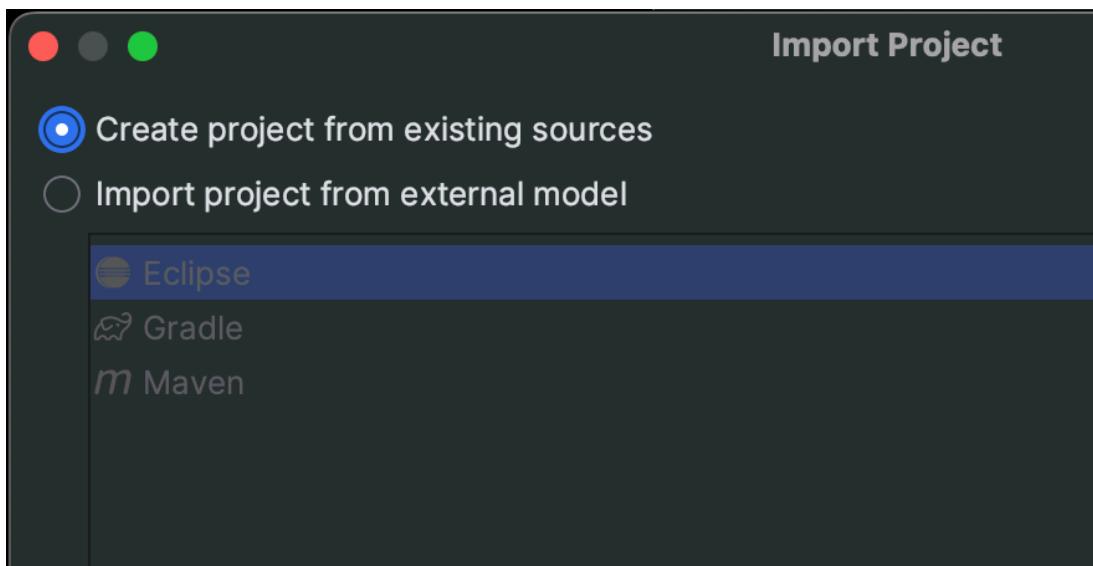


File -> New -> Project from Existing Sources... 선택



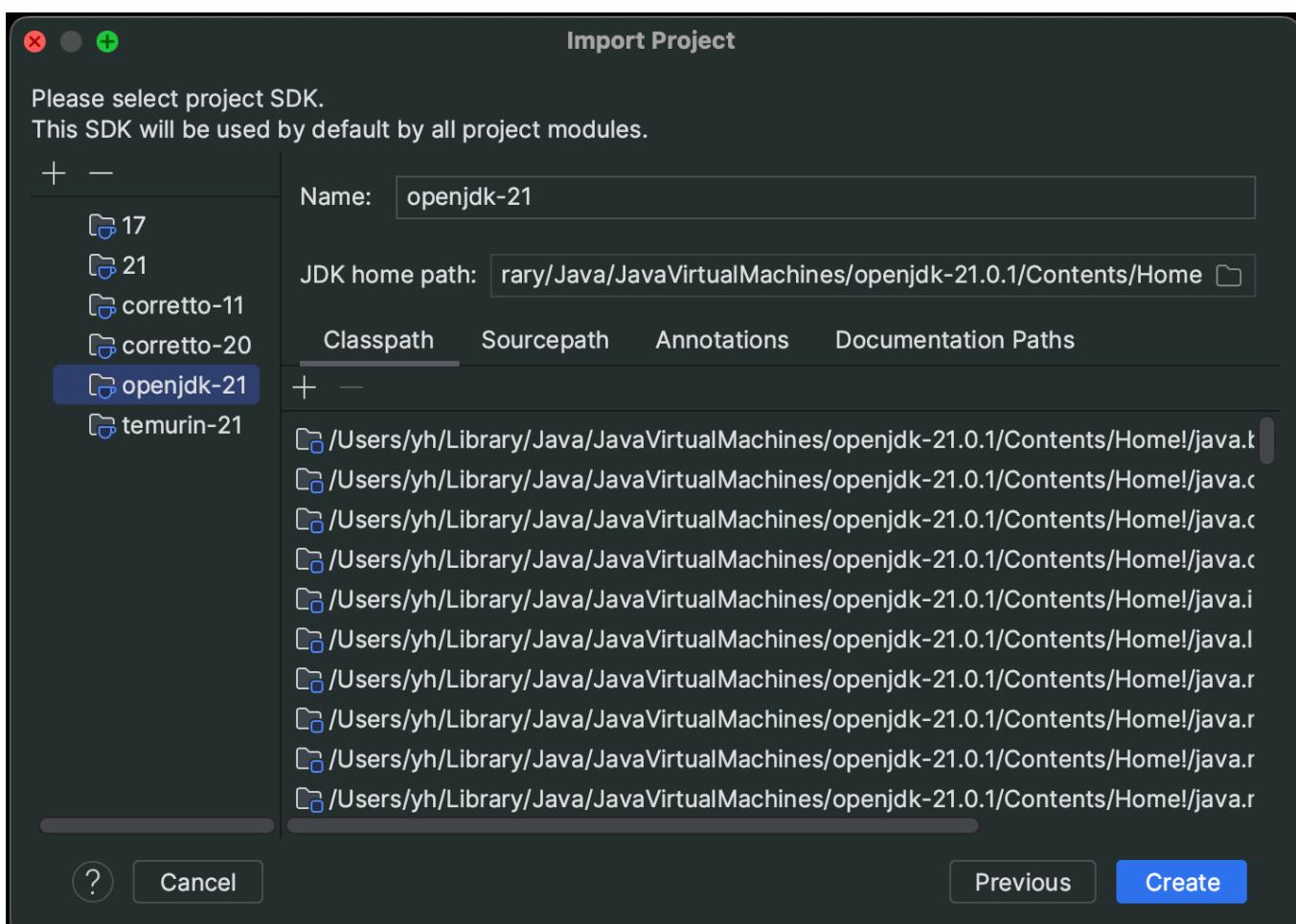
압축을 푼 프로젝트 폴더 선택

- 자바 입문 강의 폴더: java-start
- 자바 기본 강의 폴더: **java-basic**

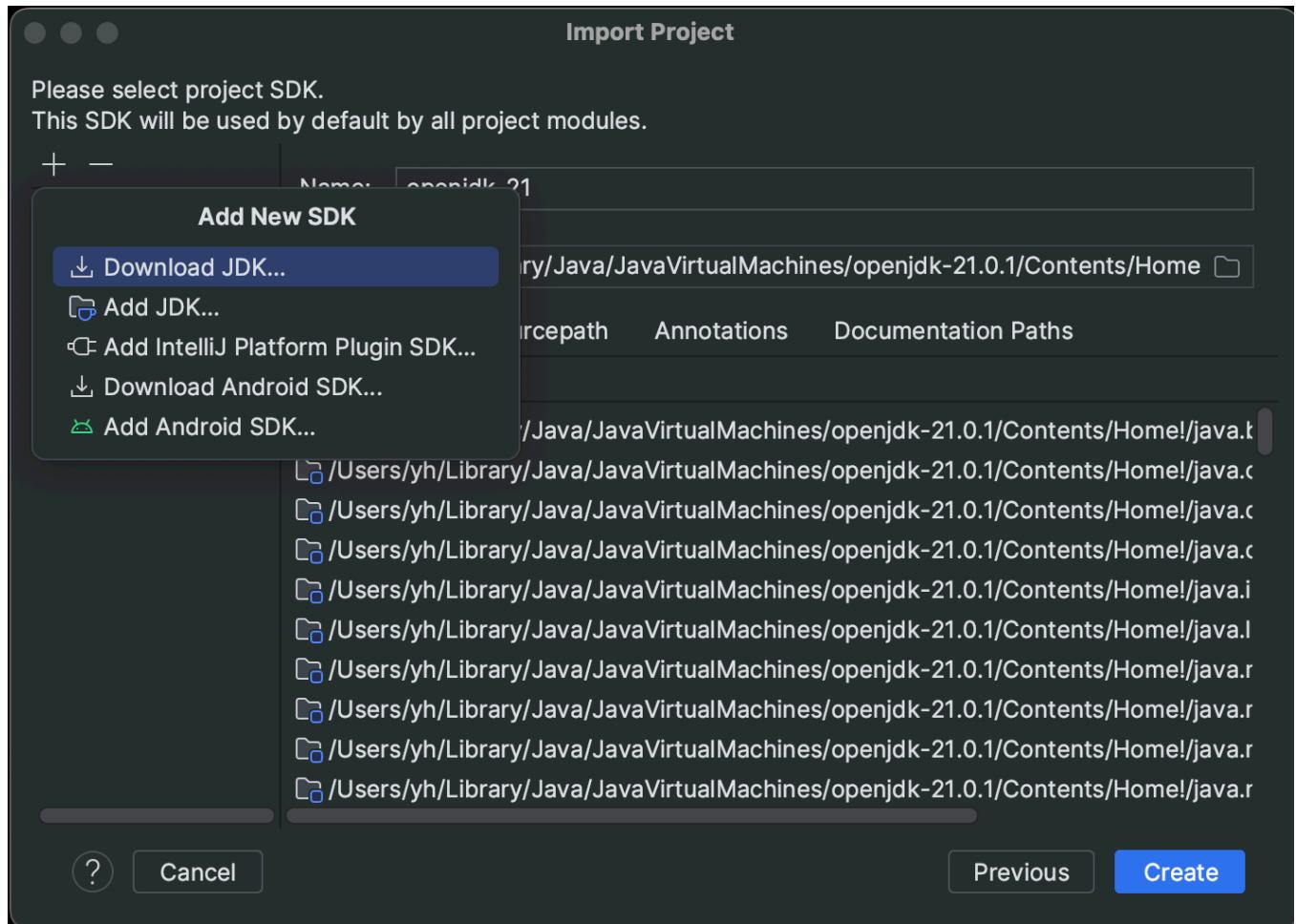


Create project from existing sources 선택

이후 계속 Next 선택



openjdk-21 선택



만약 JDK가 없다면 왼쪽 상단의 + 버튼을 눌러서 openjdk 21 다운로드 후 선택

이후 Create 버튼 선택

클래스가 필요한 이유

자바 세상은 클래스와 객체로 이루어져 있다. 그만큼 클래스와 객체라는 개념은 중요하다. 그런데 클래스와 객체는 너무 많은 내용을 포함하고 있어서 한번에 이해하기 쉽지 않다. 여기서는 클래스와 객체라는 개념이 왜 필요한지부터 시작해서, 클래스가 어떤 방식으로 발전하면서 만들어졌는지 점진적으로 알아보겠다.

먼저 클래스가 왜 필요한지 이해하기 위해 다음 문제를 풀어보자.

지금까지 학습한 내용을 기반으로 문제를 풀어보면 된다.

문제: 학생 정보 출력 프로그램 만들기

당신은 두 명의 학생 정보를 출력하는 프로그램을 작성해야 한다. 각 학생은 이름, 나이, 성적을 가지고 있다.

요구 사항:

1. 첫 번째 학생의 이름은 "학생1", 나이는 15, 성적은 90입니다.
2. 두 번째 학생의 이름은 "학생2", 나이는 16, 성적은 80입니다.
3. 각 학생의 정보를 다음과 같은 형식으로 출력해야 합니다: "이름: [이름] 나이: [나이] 성적: [성적]"
4. 변수를 사용해서 학생 정보를 저장하고 변수를 사용해서 학생 정보를 출력해야 합니다.

예시 출력:

```
이름: 학생1 나이: 15 성적: 90
```

```
이름: 학생2 나이: 16 성적: 80
```

변수를 사용해서 이 문제를 풀어보면 다음과 같다.

ClassStart1 - 변수 사용

```
package class1;

public class ClassStart1 {
    public static void main(String[] args) {
        String student1Name = "학생1";
        int student1Age = 15;
        int student1Grade = 90;

        String student2Name = "학생2";
        int student2Age = 16;
        int student2Grade = 80;

        System.out.println("이름:" + student1Name + " 나이:" + student1Age + " 성
적:" + student1Grade);
        System.out.println("이름:" + student2Name + " 나이:" + student2Age + " 성
적:" + student2Grade);
    }
}
```

학생 2명을 다루어야 하기 때문에 각각 다른 변수를 사용했다. 이 코드의 문제는 학생이 늘어날 때마다 변수를 추가로 선언해야 하고, 또 출력하는 코드도 추가해야 한다.

이런 문제를 어떻게 해결할 수 있을까? 앞서 배운 배열을 사용하면 문제를 해결할 수 있다.

문제: 이전 문제에 배열을 사용하자

이전 문제에 배열을 적용하자. 그래서 학생이 추가되어도 코드 변경을 최소화 해보자.

ClassStart2

```
package class1;

public class ClassStart2 {
    public static void main(String[] args) {
        String[] studentNames = {"학생1", "학생2"};
        int[] studentAges = {15, 16};
        int[] studentGrades = {90, 80};

        for (int i = 0; i < studentNames.length; i++) {
            System.out.println("이름:" + studentNames[i] + " 나이:" +
studentAges[i] + " 성적:" + studentGrades[i]);
        }
    }
}
```

배열을 사용한 덕분에 학생이 추가되어도 배열에 학생의 데이터만 추가하면 된다. 이제 변수를 더 추가하지 않아도 되고, 출력 부분의 코드도 그대로 유지할 수 있다.

학생 추가 전

```
String[] studentNames = {"학생1", "학생2"};
int[] studentAges = {15, 16};
int[] studentGrades = {90, 80};
```

학생 추가 후

```
String[] studentNames = {"학생1", "학생2", "학생3", "학생4", "학생5"};
int[] studentAges = {15, 16, 17, 10, 16};
int[] studentGrades = {90, 80, 100, 80, 50};
```

배열 사용의 한계

배열을 사용해서 코드 변경을 최소화하는데는 성공했지만, 한 학생의 데이터가 `studentNames[]`, `studentAges[]`, `studentGrades[]`라는 3개의 배열에 나누어져 있다. 따라서 데이터를 변경할 때 매우 조심해서 작업해야 한다. 예를 들어서 학생 2의 데이터를 제거하려면 각각의 배열마다 학생2의 요소를 정확하게 찾아서 제거해주어야 한다.

학생2 제거

```
String[] studentNames = {"학생1", "학생3", "학생4", "학생5"};
int[] studentAges = {15, 17, 10, 16};
int[] studentGrades = {90, 100, 80, 50};
```

한 학생의 데이터가 3개의 배열에 나누어져 있기 때문에 3개의 배열을 각각 변경해야 한다. 그리고 한 학생의 데이터를 관리하기 위해 3개 배열의 인덱스 순서를 항상 정확하게 맞추어야 한다. 이렇게 하면 특정 학생의 데이터를 변경할 때 실수할 가능성이 매우 높다.

이 코드는 컴퓨터가 볼 때는 아무 문제가 없지만, 사람이 관리하기에는 좋은 코드가 아니다.

정리

지금처럼 이름, 나이, 성적을 각각 따로 나누어서 관리하는 것은 사람이 관리하기 좋은 방식이 아니다.
사람이 관리하기 좋은 방식은 학생이라는 개념을 하나로 묶는 것이다. 그리고 각각의 학생 별로 본인의 이름, 나이, 성적을 관리하는 것이다.

클래스 도입

앞서 이야기한 문제를 클래스를 도입해서 해결해보자.

클래스를 사용해서 학생이라는 개념을 만들고, 각각의 학생 별로 본인의 이름, 나이, 성적을 관리하는 것이다.

우선 코드를 작성하고 실행해보자.

Student 클래스

```
package class1;

public class Student {
    String name;
    int age;
    int grade;
}
```

`class` 키워드를 사용해서 학생 클래스(`Student`)를 정의한다. 학생 클래스는 내부에 이름(`name`), 나이(`age`), 성적(`grade`) 변수를 가진다.

이렇게 클래스에 정의한 변수들을 멤버 변수, 또는 필드라 한다.

- **멤버 변수(Member Variable)**: 이 변수들은 특정 클래스에 소속된 멤버이기 때문에 이렇게 부른다.

- **필드(Field)**: 데이터 항목을 가리키는 전통적인 용어이다. 데이터베이스, 엑셀 등에서 데이터 각각의 항목을 필드라 한다.
- 자바에서 멤버 변수, 필드는 같은 뜻이다. 클래스에 소속된 변수를 뜻한다.

클래스는 관례상 대문자로 시작하고 낙타 표기법을 사용한다.

예): Student, User, MemberService

이제 학생 클래스를 사용하는 코드를 작성해보자.

ClassStart3

```
package class1;

public class ClassStart3 {
    public static void main(String[] args) {
        Student student1;
        student1 = new Student();
        student1.name = "학생1";
        student1.age = 15;
        student1.grade = 90;

        Student student2 = new Student();
        student2.name = "학생2";
        student2.age = 16;
        student2.grade = 80;

        System.out.println("이름:" + student1.name + " 나이:" + student1.age + " 성
적:" + student1.grade);
        System.out.println("이름:" + student2.name + " 나이:" + student2.age + " 성
적:" + student2.grade);
    }
}
```

실행 결과

```
이름:학생1 나이:15 성적:90
이름:학생2 나이:16 성적:80
```

클래스와 사용자 정의 타입

- 타입은 데이터의 종류나 형태를 나타낸다.
- int라고 하면 정수 타입, String이라고 하면 문자 타입이다.
- 학생(Student)이라는 타입을 만들면 되지 않을까?

- 클래스를 사용하면 `int`, `String`과 같은 타입을 직접 만들 수 있다.
- 사용자가 직접 정의하는 사용자 정의 타입을 만들려면 설계도가 필요하다. 이 설계도가 바로 클래스이다.
- 설계도인 클래스를 사용해서 실제 메모리에 만들어진 실체를 객체 또는 인스턴스라 한다.
- 클래스를 통해서 사용자가 원하는 종류의 데이터 타입을 마음껏 정의할 수 있다.

용어: 클래스, 객체, 인스턴스

클래스는 설계도이고, 이 설계도를 기반으로 실제 메모리에 만들어진 실체를 객체 또는 인스턴스라 한다. 둘다 같은 의미로 사용된다.

여기서는 학생(`Student`) 클래스를 기반으로 학생1(`student1`), 학생2(`student2`) 객체 또는 인스턴스를 만들었다.

이제 코드를 하나씩 분석해보자.

1. 변수 선언

Student student1 //Student 변수 선언



Student **student1**

- `Student student1`
 - `Student` 타입을 받을 수 있는 변수를 선언한다.
 - `int`는 정수를, `String`은 문자를 담을 수 있듯이 `Student`는 `Student` 타입의 객체(인스턴스)를 받을 수 있다.

2. 객체 생성

`student1 = new Student()` //Student 인스턴스 생성



x001

`String name`

`int age`

`int grade`

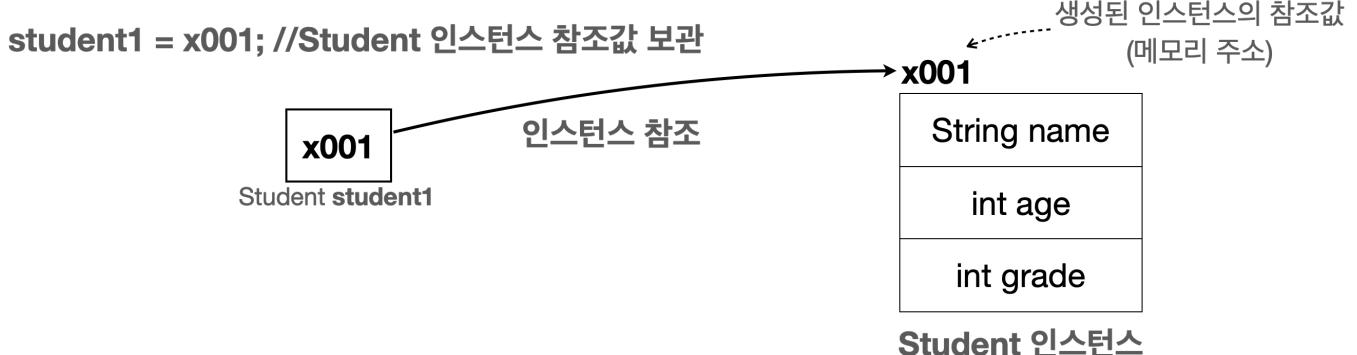
Student 인스턴스

`student1 = new Student()` 코드를 나누어 분석해보자.

- 객체를 사용하려면 먼저 설계도인 클래스를 기반으로 객체(인스턴스)를 생성해야 한다.

- `new Student()`: `new`는 새로 생성한다는 뜻이다. `new Student()`는 `Student` 클래스 정보를 기반으로 새로운 객체를 생성하라는 뜻이다. 이렇게 하면 메모리에 실제 `Student` 객체(인스턴스)를 생성한다.
- 객체를 생성할 때는 `new` 클래스명()을 사용하면 된다. 마지막에 ()도 추가해야 한다.
- `Student` 클래스는 `String name`, `int age`, `int grade` 멤버 변수를 가지고 있다. 이 변수를 사용하는 데 필요한 메모리 공간도 함께 확보한다.

3. 참조값 보관



- 객체를 생성하면 자바는 메모리 어딘가에 있는 이 객체에 접근할 수 있는 참조값(주소)(`x001`)을 반환한다.
 - 여기서 `x001`이라고 표현한 것이 참조값이다. (실제로 `x001`처럼 표현되는 것은 아니고 이해를 돋기 위한 예시이다.)
- `new` 키워드를 통해 객체가 생성되고 나면 참조값을 반환한다. 앞서 선언한 변수인 `Student student1`에 생성된 객체의 참조값(`x001`)을 보관한다.
- `Student student1` 변수는 이제 메모리에 존재하는 실제 `Student` 객체(인스턴스)의 참조값을 가지고 있다.
 - `student1` 변수는 방금 만든 객체에 접근할 수 있는 참조값을 가지고 있다. 따라서 이 변수를 통해서 객체를 접근(참조)할 수 있다. 쉽게 이야기해서 `student1` 변수를 통해 메모리에 있는 실제 객체를 접근하고 사용할 수 있다.

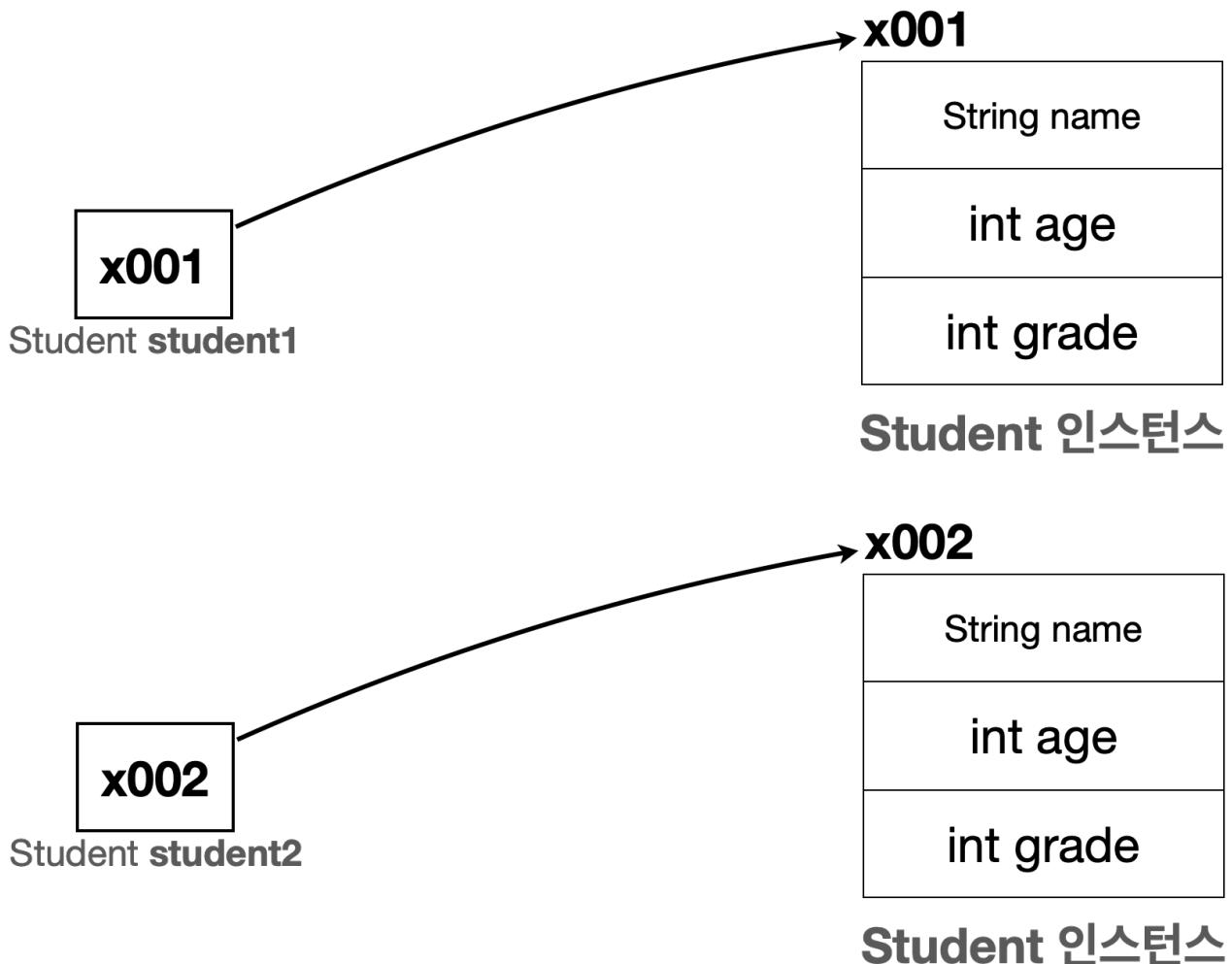
참조값을 변수에 보관해야 하는 이유

객체를 생성하는 `new Student()` 코드 자체에는 아무런 이름이 없다. 이 코드는 단순히 `Student` 클래스를 기반으로 메모리에 실제 객체를 만드는 것이다. 따라서 생성한 객체에 접근할 수 있는 방법이 필요하다. 이런 이유로 객체를 생성할 때 반환되는 참조값을 어딘가에 보관해두어야 한다. 앞서 `Student student1` 변수에 참조값(`x001`)을 저장해두었으므로 저장한 참조값(`x001`)을 통해서 실제 메모리에 존재하는 객체에 접근할 수 있다.

지금까지 설명한 내용을 간단히 풀어보면 다음과 같다.

```
Student student1 = new Student(); //1. Student 객체 생성
Student student1 = x001; //2. new Student()의 결과로 x001 참조값 반환
student1 = x001; //3. 최종 결과
```

이후에 학생2(student2)까지 생성하면 다음과 같이 Student 객체(인스턴스)가 메모리에 2개 생성된다. 각각 참조값이 다르므로 서로 구분할 수 있다.



참조값을 확인하고 싶다면 다음과 같이 객체를 담고 있는 변수를 출력해보면 된다.

```
System.out.println(student1);  
System.out.println(student2);
```

출력 결과

```
class1.Student@7a81197d  
class1.Student@2f2c9b19
```

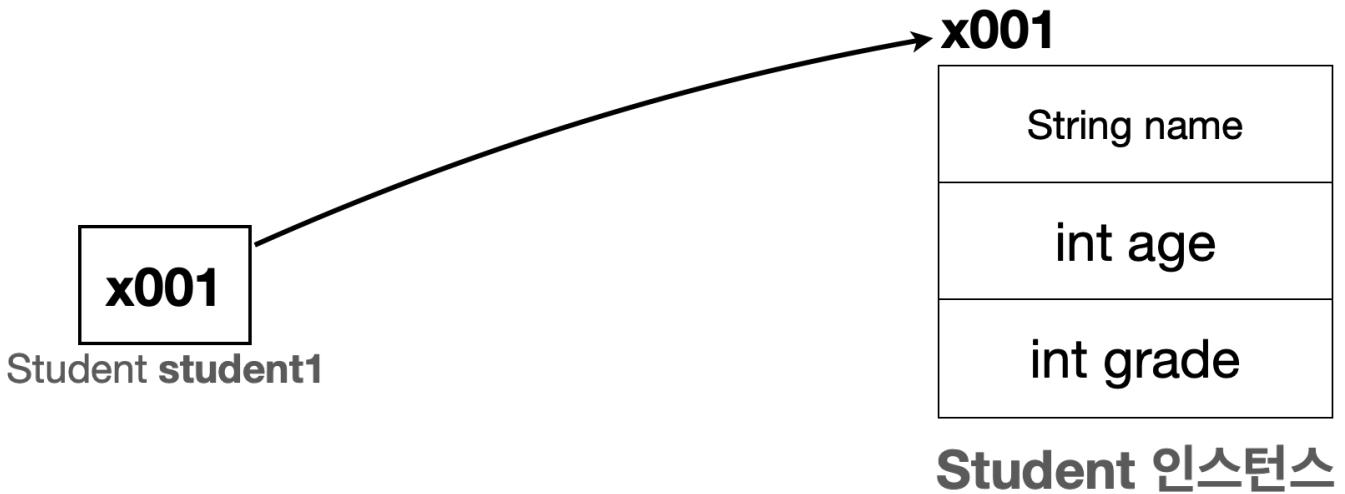
@앞은 패키지 + 클래스 정보를 뜻한다. @뒤에 16진수는 참조값을 뜻한다.

객체 사용

클래스를 통해 생성한 객체를 사용하려면 먼저 메모리에 존재하는 객체에 접근해야 한다. 객체에 접근하려면 .(점, dot)을 사용하면 된다.

```
//객체 값 대입  
student1.name = "학생1";  
student1.age = 15;  
student1.grade = 90;  
  
//객체 값 사용  
System.out.println("이름:" + student1.name + " 나이:" + student1.age + " 성적:" +  
student1.grade);
```

객체 참조 그림



객체에 값 대입

객체가 가지고 있는 멤버 변수(name, age, grade)에 값을 대입하려면 먼저 객체에 접근해야 한다.

객체에 접근하려면 .(점, dot) 키워드를 사용하면 된다. 이 키워드는 변수(student1)에 들어있는 참조값(x001)을 읽어서 메모리에 존재하는 객체에 접근한다.

순서를 간단히 풀어보면 다음과 같다.

```
student1.name="학생1" //1. student1 객체의 name 멤버 변수에 값 대입  
x001.name="학생1" //2. 변수에 있는 참조값을 통해 실제 객체에 접근, 해당 객체의 name 멤버 변수에 값  
대입
```

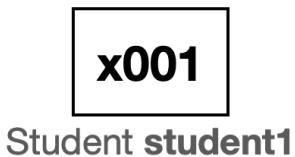
student1.(dot)이라고 하면 student1 변수가 가지고 있는 참조값을 통해 실제 객체에 접근한다.

student1은 x001이라는 참조값을 가지고 있으므로 x001 위치에 있는 Student 객체에 접근한다.

그림으로 자세히 알아보자.

student1.name = "학생1" 코드 실행 전

student1.name = "학생1"



x001

String name
int age
int grade

Student 인스턴스

student1.name = "학생1" 코드 실행 후

x001.name = "학생1"

x001
Student student1

1. x001 접근

2. name 접근

3. name 변수에 대입

String name="학생1"
int age
int grade

Student 인스턴스

- student1.name 코드를 통해 .(dot) 키워드가 사용되었다. student1 변수가 가지고 있는 참조값을 통해 실제 객체에 접근한다.
- x001.name = "학생1" : x001 객체가 있는 곳의 name 멤버 변수에 "학생1" 데이터가 저장된다.

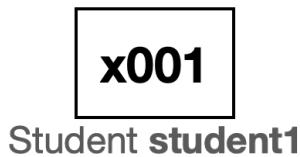
객체 값 읽기

객체의 값을 읽는 것도 앞서 설명한 내용과 같다. .(점, dot) 키워드를 통해 참조값을 사용해서 객체에 접근한 다음에 원하는 작업을 하면 된다. 다음 예제를 보자.

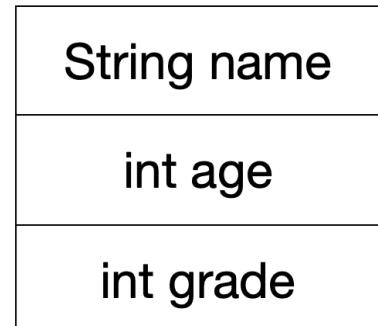
```
//1. 객체 값 읽기
System.out.println("이름:" + student1.name);
//2. 변수에 있는 참조값을 통해 실제 객체에 접근하고, name 멤버 변수에 접근한다.
System.out.println("이름:" + x001.name);
//3. 객체의 멤버 변수의 값을 읽어옴
System.out.println("이름:" + "학생1");
```

객체 값 읽기 - 그림1

`println(student1.name)`



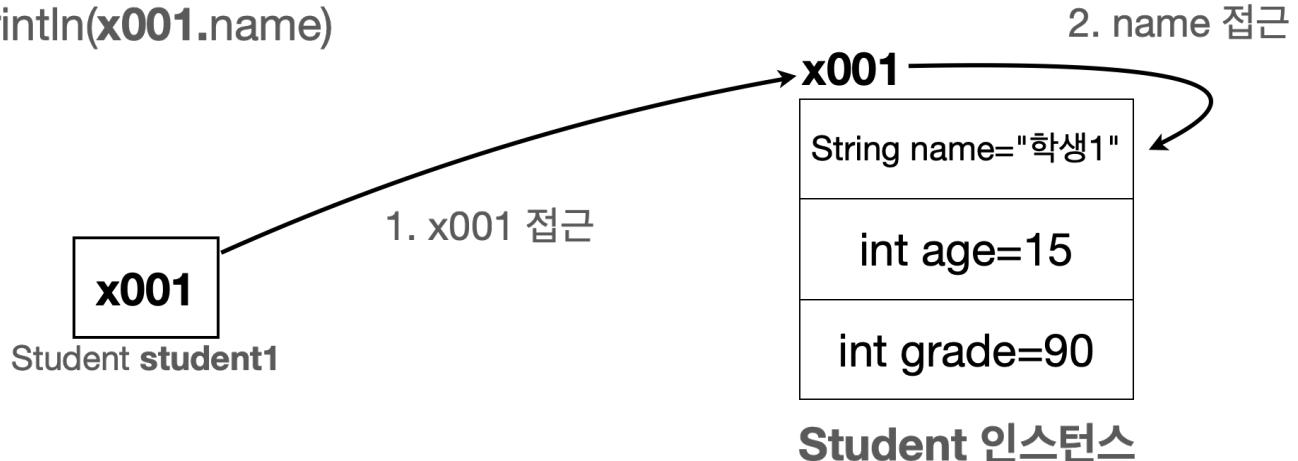
x001



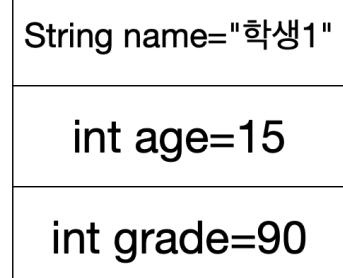
Student 인스턴스

객체 값 읽기 - 그림2

`println(x001.name)`



x001



Student 인스턴스

- x001에 있는 Student 인스턴스의 name 멤버 변수는 "학생1"이라는 값을 가지고 있다. 이 값을 읽어서 사용 한다.

클래스, 객체, 인스턴스 정리

클래스 - Class

클래스는 객체를 생성하기 위한 '틀' 또는 '설계도'이다. 클래스는 객체가 가져야 할 속성(변수)과 기능(메서드)을 정의한다. 예를 들어 학생이라는 클래스는 속성으로 name, age, grade를 가진다. 참고로 기능(메서드)은 뒤에서 설명한다. 지금은 속성(변수)에 집중하자

- 틀: 붕어빵 틀을 생각해보자. 붕어빵 틀은 붕어빵이 아니다! 이렇게 생긴 붕어빵이 나왔으면 좋겠다고 만드는 틀일 뿐이다. 실제 먹을 수 있는 것이 아니다. 실제 먹을 수 있는 팥 붕어빵을 객체 또는 인스턴스라 한다.
- 설계도: 자동차 설계도를 생각해보자. 자동차 설계도는 자동차가 아니다! 설계도는 실제 존재하는 것이 아니라 개

념으로만 있는 것이다. 설계도를 통해 생산한 실제 존재하는 흰색 테슬라 모델 Y 자동차를 객체 또는 인스턴스라 한다.

객체 - Object

객체는 클래스에서 정의한 속성과 기능을 가진 실체이다. 객체는 서로 독립적인 상태를 가진다.

예를 들어 위 코드에서 `student1`은 학생1의 속성을 가지는 객체이고, `student2`는 학생2의 속성을 가지는 객체이다. `student1`과 `student2`는 같은 클래스에서 만들어졌지만, 서로 다른 객체이다.

인스턴스 - Instance

인스턴스는 특정 클래스로부터 생성된 객체를 의미한다. 그래서 객체와 인스턴스라는 용어는 자주 혼용된다. 인스턴스는 주로 객체가 어떤 클래스에 속해 있는지 강조할 때 사용한다. 예를 들어서 `student1` 객체는 `Student` 클래스의 인스턴스다. 라고 표현한다.

객체 vs 인스턴스

둘다 클래스에서 나온 실체라는 의미에서 비슷하게 사용되지만, 용어상 인스턴스는 객체보다 좀 더 관계에 초점을 맞춘 단어이다. 보통 `student1`은 `Student`의 객체이다. 라고 말하는 대신 `student1`은 `Student`의 인스턴스이다. 라고 특정 클래스와의 관계를 명확히 할 때 인스턴스라는 용어를 주로 사용한다.

좀 더 쉽게 풀어보자면, 모든 인스턴스는 객체이지만, 우리가 인스턴스라고 부르는 순간은 특정 클래스로부터 그 객체가 생성되었음을 강조하고 싶을 때이다. 예를 들어 `student1`은 객체이지만, 이 객체가 `Student` 클래스로부터 생성되었다는 점을 명확히 하기 위해 `student1`을 `Student`의 인스턴스라고 부른다.

하지만 둘다 클래스에서 나온 실체라는 핵심 의미는 같기 때문에 보통 둘을 구분하지 않고 사용한다.

배열 도입 - 시작

클래스와 객체 덕분에 학생 데이터를 구조적으로 이해하기 쉽게 변경할 수 있었다.

마치 실제 학생이 있고, 그 안에 각 학생의 정보가 있는 것 같다. 따라서 사람이 이해하기도 편리하다.

이제 각각의 학생 별로 객체를 생성하고, 해당 객체에 학생의 데이터를 관리하면 된다.

하지만 코드를 보면 아쉬운 부분이 있는데, 바로 학생을 출력하는 부분이다.

```
System.out.println("이름:" + student1.name + " 나이:" + student1.age + ...);  
System.out.println("이름:" + student2.name + " 나이:" + student2.age + ...);
```

새로운 학생이 추가될 때마다 출력하는 부분도 함께 추가해야 한다.

배열을 사용하면 특정 타입을 연속한 데이터 구조로 묶어서 편리하게 관리할 수 있다.

Student 클래스를 사용한 변수들도 Student 타입이기 때문에 학생도 배열을 사용해서 하나의 데이터 구조로 묶어서 관리할 수 있다.

Student 타입을 사용하는 배열을 도입해보자.

ClassStart4

```
package class1;

public class ClassStart4 {
    public static void main(String[] args) {
        Student student1 = new Student();
        student1.name = "학생1";
        student1.age = 15;
        student1.grade = 90;

        Student student2 = new Student();
        student2.name = "학생2";
        student2.age = 16;
        student2.grade = 80;

        Student[] students = new Student[2];
        students[0] = student1;
        students[1] = student2;

        System.out.println("이름:" + students[0].name + " 나이:" + students[0].age
+ " 성적:" + students[0].grade);
        System.out.println("이름:" + students[1].name + " 나이:" + students[1].age
+ " 성적:" + students[1].grade);

    }
}
```

코드를 분석해보자.

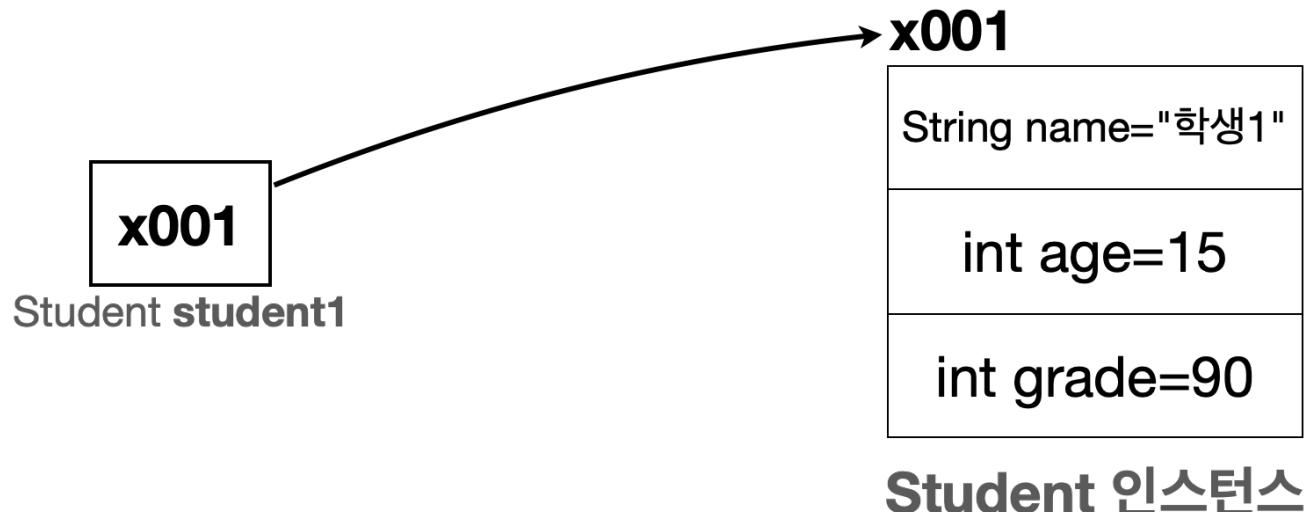
```
Student student1 = new Student();
student1.name = "학생1";
student1.age = 15;
student1.grade = 90;

Student student2 = new Student();
student2.name = "학생2";
```

```
student2.age = 16;  
student2.grade = 80;
```

Student 클래스를 기반으로 student1, student2 인스턴스를 생성한다. 그리고 필요한 값을 채워둔다.

인스턴스 생성 그림



Student `student1`

Student 인스턴스

Student `student2`

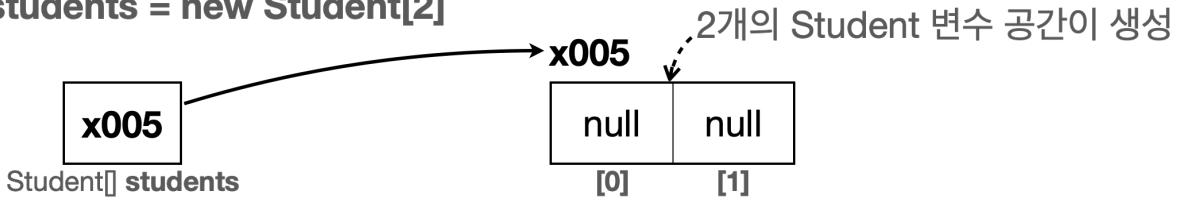
Student 인스턴스

배열에 참조값 대입

이번에는 `Student` 를 담을 수 있는 배열을 생성하고, 해당 배열에 `student1`, `student2` 인스턴스를 보관하자.

```
Student[] students = new Student[2];
```

Student students = new Student[2]



- `Student` 변수를 2개 보관할 수 있는 사이즈 2의 배열을 만든다.
- `Student` 타입의 변수는 `Student` 인스턴스의 참조값을 보관한다. `Student` 배열의 각각의 항목도 `Student` 타입의 변수일 뿐이다. 따라서 `Student` 타입의 참조값을 보관한다.
 - `student1`, `student2` 변수를 생각해보면 `Student` 타입의 참조값을 보관한다.
- 배열에는 아직 참조값을 대입하지 않았기 때문에 참조값이 없다는 의미의 `null` 값으로 초기화 된다.

이제 배열에 객체를 보관하자.

```
students[0] = student1;  
students[1] = student2;
```

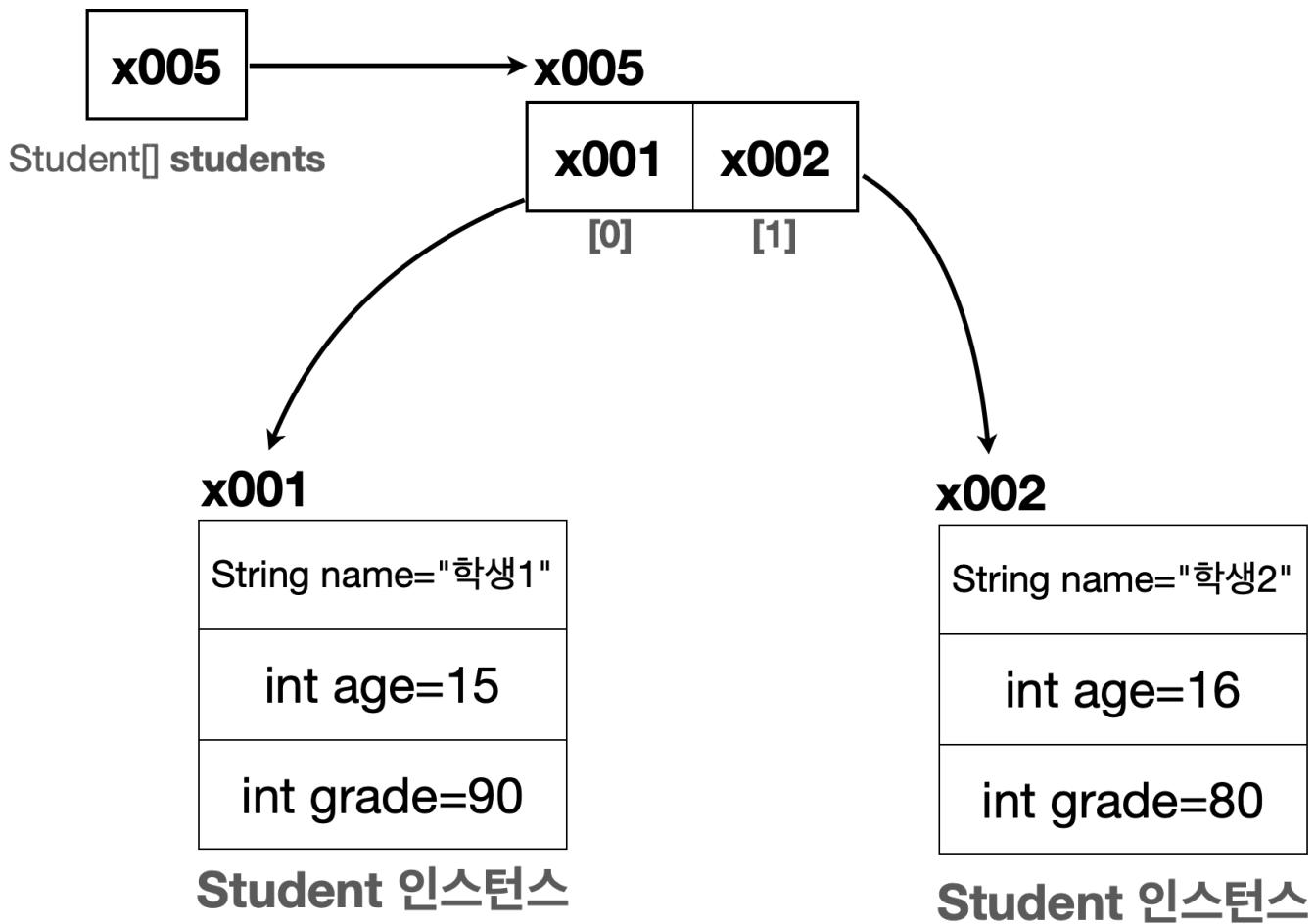
// 자바에서 대입은 항상 변수에 들어 있는 값을 복사한다.

```
students[0] = x001;  
students[1] = x002;
```

자바에서 대입은 항상 변수에 들어 있는 값을 복사한다.

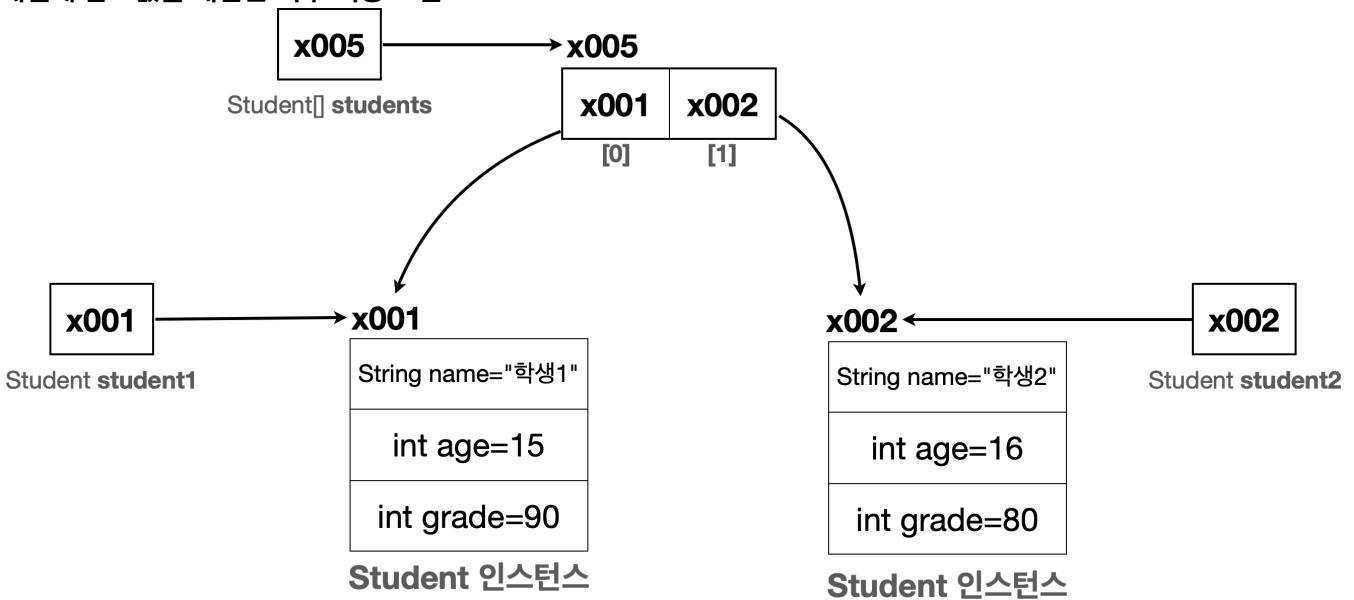
`student1`, `student2`에는 참조값이 보관되어 있다. 따라서 이 참조값이 배열에 저장된다. 또는 `student1`, `student2`에 보관된 참조값을 읽고 복사해서 배열에 대입한다고 표현한다.

배열에 참조값을 대입한 이후 배열 그림



이제 배열은 `x001`, `x002`의 참조값을 가진다. 참조값을 가지고 있기 때문에 `x001` (학생1), `x002` (학생2) `Student` 인스턴스에 모두 접근할 수 있다.

배열에 참조값을 대입한 이후 최종 그림



자바에서 대입은 항상 변수에 들어 있는 값을 복사해서 전달한다.

```
students[0] = student1;
students[1] = student2;
```

```
//자바에서 대입은 항상 변수에 들어 있는 값을 복사한다.  
students[0] = x001;  
students[1] = x002;
```

자바에서 변수의 대입(=)은 모두 변수에 들어있는 값을 복사해서 전달하는 것이다. 이 경우 오른쪽 변수인 student1, student2에는 참조값이 들어있다. 그래서 이 값을 복사해서 왼쪽에 있는 배열에 전달한다. 따라서 기존 student1, student2에 들어있던 참조값은 당연히 그대로 유지된다.

주의!

변수에는 인스턴스 자체가 들어있는 것이 아니다! 인스턴스의 위치를 가리키는 참조값이 들어있을 뿐이다! 따라서 대입 (=)시에 인스턴스가 복사되는 것이 아니라 참조값만 복사된다.

배열에 들어있는 객체 사용

배열에 들어있는 객체를 사용하려면 먼저 배열에 접근하고, 그 다음에 객체에 접근하면 된다. 이전에 설명한 그림과 코드를 함께 보면 쉽게 이해가 될 것이다.

학생1 예제

```
System.out.println(students[0].name); //배열 접근 시작  
System.out.println(x005[0].name); // [0]를 사용해서 x005 배열의 0번 요소에 접근  
System.out.println(x001.name); // .(dot)을 사용해서 참조값으로 객체에 접근  
System.out.println("학생1");
```

학생2 예제

```
System.out.println(students[1].name); //배열 접근 시작  
System.out.println(x005[1].name); // [1]를 사용해서 x005 배열의 1번 요소에 접근  
System.out.println(x002.name); // .(dot)을 사용해서 참조값으로 객체에 접근  
System.out.println("학생2");
```

배열 도입 - 리펙토링

배열을 사용한 덕분에 출력에서 다음과 같이 for문을 도입할 수 있게 되었다.

ClassStart5

```
package class1;
```

```

public class ClassStart5 {
    public static void main(String[] args) {
        Student student1 = new Student();
        student1.name = "학생1";
        student1.age = 15;
        student1.grade = 90;

        Student student2 = new Student();
        student2.name = "학생2";
        student2.age = 16;
        student2.grade = 80;

        //배열 선언
        Student[] students = new Student[]{student1, student2};

        //for문 적용
        for (int i = 0; i < students.length; i++) {
            System.out.println("이름:" + students[i].name + " 나이:" +
students[i].age + " 성적:" + students[i].grade);
        }
    }
}

```

배열 선언 최적화

우리가 직접 정의한 `Student` 타입도 일반적인 변수와 동일하게 배열을 생성할 때 포함할 수 있다.

```
Student[] students = new Student[]{student1, student2};
```

생성과 선언을 동시에 하는 경우 다음과 같이 더 최적화 할 수 있다.

```
Student[] students = {student1, student2};
```

for문 최적화

배열을 사용한 덕분에 `for`문을 사용해서 반복 작업을 깔끔하게 처리할 수 있다.

for문 도입

```
for (int i = 0; i < students.length; i++) {
    System.out.println("이름:" + students[i].name + " 나이:" + students[i].age
+ ...);
```

```
}
```

for문 - 반복 요소를 변수에 담아서 처리하기

```
for (int i = 0; i < students.length; i++) {  
    Student s = students[i];  
    System.out.println("이름:" + s.name + " 나이:" + s.age + ...);  
}
```

`students[i].name`, `students[i].age`처럼 `students[i]`를 자주 접근하는 것이 번거롭다면 반복해서 사용하는 객체를 `Student s`와 같은 변수에 담아두고 사용해도 된다.

물론 이런 경우에는 다음과 같이 향상된 for문을 사용하는 것이 가장 깔끔하다.

향상된 for문(Enhanced For Loop)

```
for (Student s : students) {  
    System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" + s.grade);  
}
```

문제와 풀이

문제를 스스로 풀어보세요.

백문이 불여일타!

프로그래밍은 수영이나 자전거 타기와 비슷하다. 아무리 책을 보고 강의 영상을 봐도 직접 물속에 들어가거나 자전거를 타봐야 이해할 수 있다. 프로그래밍도 마찬가지이다. 이해하고 외우는 것도 중요하지만 무엇보다 직접 코딩을 해보는 것이 더욱 중요하다. 읽어서 이해가 잘 안되더라도 직접 코딩을 해보면 자연스럽게 이해가 되는 경우도 많다. 백번 읽는 것 보다 한번 직접 코딩해서 결과를 보는 것이 좋은 개발자로 빠르게 성장할 수 있는 지름길이다.

문제: 영화 리뷰 관리하기 1

문제 설명

당신은 영화 리뷰 정보를 관리하려고 한다. 먼저, 영화 리뷰 정보를 담을 수 있는 `MovieReview` 클래스를 만들어보자.

요구 사항

1. MovieReview 클래스는 다음과 같은 멤버 변수를 포함해야 한다.
 - 영화 제목 (title)
 - 리뷰 내용 (review)
2. MovieReviewMain 클래스 안에 main() 메서드를 포함하여, 영화 리뷰 정보를 선언하고 출력하자.

예시 코드 구조

```
public class MovieReview {  
    String title;  
    String review;  
}
```

```
public class MovieReviewMain {  
    public static void main(String[] args) {  
        // 영화 리뷰 정보 선언  
        // 영화 리뷰 정보 출력  
    }  
}
```

출력 예시

```
영화 제목: "인셉션", 리뷰: "인생은 무한 루프"  
영화 제목: "어바웃 타임", 리뷰: "인생 시간 영화!"
```

답안 - 영화 리뷰 관리하기1

```
package class1.ex;  
  
public class MovieReview {  
    String title;  
    String review;  
}
```

```
package class1.ex;  
  
public class MovieReviewMain1 {  
    public static void main(String[] args) {  
        MovieReview inception = new MovieReview();  
        inception.title = "인셉션";  
        inception.review = "인생은 무한 루프";  
  
        MovieReview aboutTime = new MovieReview();  
        aboutTime.title = "어바웃 타임";
```

```

        aboutTime.review = "인생 시간 영화!";

        System.out.println("영화 제목: " + inception.title + ", 리뷰: " +
inception.review);
        System.out.println("영화 제목: " + aboutTime.title + ", 리뷰: " +
aboutTime.review);
    }
}

```

문제: 영화 리뷰 관리하기2

기존 문제에 배열을 도입하고, 영화 리뷰를 배열에 관리하자.

리뷰를 출력할 때 배열과 for 문을 사용해서 System.out.println을 한번만 사용하자.

답안 - 영화 리뷰 관리하기2

```

package class1.ex;

public class MovieReviewMain2 {
    public static void main(String[] args) {
        MovieReview[] reviews = new MovieReview[2];

        MovieReview inception = new MovieReview();
        inception.title = "인셉션";
        inception.review = "인생은 무한 루프";
        reviews[0] = inception;

        MovieReview aboutTime = new MovieReview();
        aboutTime.title = "어바웃 타임";
        aboutTime.review = "인생 시간 영화!";
        reviews[1] = aboutTime;

        for (MovieReview review : reviews) {
            System.out.println("영화 제목: " + review.title + ", 리뷰: " +
review.review);
        }
    }
}

```

문제: 상품 주문 시스템 개발

문제 설명

당신은 온라인 상점의 주문 관리 시스템을 만들려고 한다.

먼저, 상품 주문 정보를 담을 수 있는 `ProductOrder` 클래스를 만들어보자.

요구 사항

1. `ProductOrder` 클래스는 다음과 같은 멤버 변수를 포함해야 한다.
 - 상품명 (`productName`)
 - 가격 (`price`)
 - 주문 수량 (`quantity`)
2. `ProductOrderMain` 클래스 안에 `main()` 메서드를 포함하여, 여러 상품의 주문 정보를 배열로 관리하고, 그 정보들을 출력하고, 최종 결제 금액을 계산하여 출력하자.
3. 출력 예시와 같도록 출력하면 된다.

예시 코드 구조

```
public class ProductOrder {  
    String productName;  
    int price;  
    int quantity;  
}
```

```
public class ProductOrderMain {  
    public static void main(String[] args) {  
        // 여러 상품의 주문 정보를 담는 배열 생성  
        // 상품 주문 정보를 `ProductOrder` 타입의 변수로 받아 저장  
        // 상품 주문 정보와 최종 금액 출력  
    }  
}
```

출력 예시

```
상품명: 두부, 가격: 2000, 수량: 2  
상품명: 김치, 가격: 5000, 수량: 1  
상품명: 콜라, 가격: 1500, 수량: 2  
총 결제 금액: 12000
```

답안 - 상품 주문 시스템 개발

```
package class1.ex;
```

```
public class ProductOrder {  
    String productName;  
    int price;  
    int quantity;  
}
```

```
package class1.ex;  
  
public class ProductOrderMain {  
  
    public static void main(String[] args) {  
        ProductOrder[] orders = new ProductOrder[3];  
  
        // 첫 번째 상품 주문 정보 입력  
        ProductOrder order1 = new ProductOrder();  
        order1.productName = "두부";  
        order1.price = 2000;  
        order1.quantity = 2;  
        orders[0] = order1;  
  
        // 두 번째 상품 주문 정보 입력  
        ProductOrder order2 = new ProductOrder();  
        order2.productName = "김치";  
        order2.price = 5000;  
        order2.quantity = 1;  
        orders[1] = order2;  
  
        // 세 번째 상품 주문 정보 입력  
        ProductOrder order3 = new ProductOrder();  
        order3.productName = "콜라";  
        order3.price = 1500;  
        order3.quantity = 2;  
        orders[2] = order3;  
  
        int totalAmount = 0;  
        for (ProductOrder order : orders) {  
            System.out.println("상품명: " + order.productName + ", 가격: " +  
order.price + ", 수량: " + order.quantity);  
            totalAmount += order.price * order.quantity;  
        }  
  
        System.out.println("총 결제 금액: " + totalAmount);  
    }  
}
```

}

정리

2. 기본형과 참조형

#1.인강/0.자바/2.자바-기본

- /기본형 vs 참조형1 - 시작
- /기본형 vs 참조형2 - 변수 대입
- /기본형 vs 참조형3 - 메서드 호출
- /참조형과 메서드 호출 - 활용
- /변수와 초기화
- /null
- /NullPointerException
- /문제와 풀이
- /정리

기본형 vs 참조형1 - 시작

자바에서 참조형을 제대로 이해하는 것은 정말 중요하다.

지금까지 기본형과 참조형에 대해서 조금씩 보았다. 이번에는 기본형과 참조형에 대해서 더 깊이있게 알아보고 확실하게 정리해보자.

변수의 데이터 타입을 가장 크게 보면 기본형과 참조형으로 분류할 수 있다. 사용하는 값을 변수에 직접 넣을 수 있는 기본형, 그리고 이전에 본 `Student student1`과 같이 객체가 저장된 메모리의 위치를 가리키는 참조값을 넣을 수 있는 참조형으로 분류할 수 있다.

- 기본형(Primitive Type): `int`, `long`, `double`, `boolean`처럼 변수에 사용할 값을 직접 넣을 수 있는 데이터 타입을 기본형이라 한다.
- 참조형(Reference Type): `Student student1`, `int[] students`와 같이 데이터에 접근하기 위한 참조(주소)를 저장하는 데이터 타입을 참조형이라 한다. 참조형은 객체 또는 배열에 사용된다.

쉽게 이야기해서 기본형 변수에는 직접 사용할 수 있는 값이 들어있지만 참조형 변수에는 위치(참조값)가 들어가 있다. 참조형 변수를 통해서 뭔가 하려면 결국 참조값을 통해 해당 위치로 이동해야 한다.

기본형 vs 참조형 - 기본

- 기본형은 숫자 `10`, `20`과 같이 실제 사용하는 값을 변수에 담을 수 있다. 그래서 해당 값을 바로 사용할 수 있다.
- 참조형은 실제 사용하는 값을 변수에 담는 것이 아니다. 이를 그대로 실제 객체의 위치(참조, 주소)를 저장한다. 참조형에는 객체와 배열이 있다.
 - 객체는 `.`(dot)을 통해서 메모리 상에 생성된 객체를 찾아가야 사용할 수 있다.

- 배열은 [] 를 통해서 메모리 상에 생성된 배열을 찾아가야 사용할 수 있다.

기본형 vs 참조형 - 계산

- 기본형은 들어있는 값을 그대로 계산에 사용할 수 있다.
 - 예) 더하고 빼고, 사용하고 등등, (숫자 같은 것들은 바로 계산할 수 있음)
- 참조형은 들어있는 참조값을 그대로 사용할 수 없다. 주소지만 가지고는 할 수 있는게 없다. 주소지에 가야 실체가 있다!
 - 예) 더하고 빼고 사용하고 못함! 참조값만 가지고는 계산 할 수 있는 것이 없음!

기본형은 연산이 가능하지만 참조형은 연산이 불가능하다.

```
int a = 10, b = 20;
int sum = a + b;
```

기본형은 변수에 실제 사용하는 값이 담겨있다. 따라서 +, - 와 같은 연산이 가능하다.

```
Student s1 = new Student();
Student s2 = new Student();
s1 + s2 //오류 발생
```

참조형은 변수에 객체의 위치인 참조값이 들어있다. 참조값은 계산에 사용할 수 없다. 따라서 오류가 발생한다.

물론 다음과 같이 . 을 통해 객체의 기본형 멤버 변수에 접근한 경우에는 연산을 할 수 있다.

```
Student s1 = new Student();
s1.grade = 100;
Student s2 = new Student();
s2.grade = 90;
int sum = s1.grade + s2.grade; //연산 가능
```

쉽게 이해하는 팁

기본형을 제외한 나머지는 모두 참조형이다.

- 기본형은 소문자로 시작한다. int, long, double, boolean 모두 소문자로 시작한다.
 - 기본형은 자바가 기본으로 제공하는 데이터 타입이다. 이러한 기본형은 개발자가 새로 정의할 수 없다. 개발자는 참조형인 클래스만 직접 정의할 수 있다.
- 클래스는 대문자로 시작한다. Student
 - 클래스는 모두 참조형이다.

참고 - String

자바에서 String 은 특별하다. String 은 사실은 클래스다. 따라서 참조형이다. 그런데 기본형처럼 문자 값을 바로 대입할 수 있다. 문자는 매우 자주 다루기 때문에 자바에서 특별하게 편의 기능을 제공한다. String 에 대한 자세한 내

용은 뒤에서 설명한다.

기본형 vs 참조형2 - 변수 대입

대원칙: 자바는 항상 변수의 값을 복사해서 대입한다.

자바에서 변수에 값을 대입하는 것은 변수에 들어 있는 값을 복사해서 대입하는 것이다.

기본형, 참조형 모두 항상 변수에 있는 값을 복사해서 대입한다. 기본형이면 변수에 들어 있는 실제 사용하는 값을 복사해서 대입하고, 참조형이면 변수에 들어 있는 참조값을 복사해서 대입한다.

이 대원칙을 이해하면 복잡한 상황에도 코드를 단순하게 이해할 수 있다.

기본형 대입

```
int a = 10;  
int b = a;
```

참조형 대입

```
Student s1 = new Student();  
Student s2 = s1;
```

기본형은 변수에 값을 대입하더라도 실제 사용하는 값이 변수에 바로 들어있기 때문에 해당 값만 복사해서 대입한다고 생각하면 쉽게 이해할 수 있다. 그런데 참조형의 경우 실제 사용하는 객체가 아니라 객체의 위치를 가리키는 참조값만 복사된다. 쉽게 이야기해서 실제 건물이 복사가 되는 것이 아니라 건물의 위치인 주소만 복사되는 것이다. 따라서 같은 건물을 찾아갈 수 있는 방법이 하나 늘어날 뿐이다.

구체적인 예시를 통해서 변수 대입시 기본형과 참조형의 차이를 알아보자.

기본형과 변수 대입

다음 코드를 보고 어떤 결과가 나올지 먼저 생각해보자. 너무 쉽고 당연한 내용이지만, 이후에 나올 참조형과 비교를 위해서 다시 한번 정리해보자.

VarChange1

```
package ref;
```

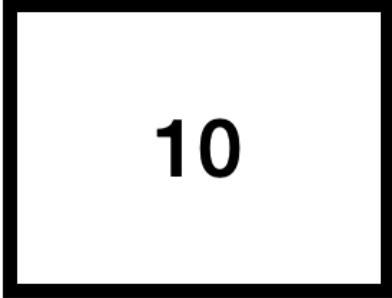
```
public class VarChange1 {  
  
    public static void main(String[] args) {  
        int a = 10;  
        int b = a;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
  
        //a 변경  
        a = 20;  
        System.out.println("변경 a = 20");  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
  
        //b 변경  
        b = 30;  
        System.out.println("변경 b = 30");  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
    }  
}
```

실행 결과

```
a = 10  
b = 10  
변경 a = 20  
a = 20  
b = 10  
변경 b = 30  
a = 20  
b = 30
```

그림을 통해 자세히 알아보자.

int a = 10



a

```
int b = a
```

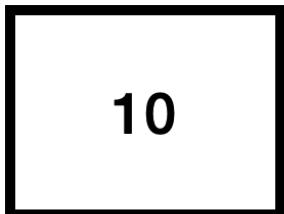


실행 결과

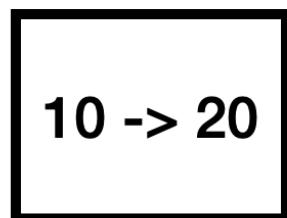
```
a = 10  
b = 10
```

변수의 대입은 변수에 들어있는 값을 복사해서 대입한다. 여기서는 변수 a에 들어있는 값 10을 복사해서 변수 b에 대입한다. 변수 a 자체를 b에 대입하는 것이 아니다!

a = 20



b



a

실행 결과

```
a = 10  
b = 20
```

변수 a에 값 20을 대입했다. 따라서 변수 a의 값이 10에서 20으로 변경되었다. 당연한 이야기지만 변수 b에는 아무런 영향을 주지 않는다.

`b = 30`



실행 결과

```
a = 20  
b = 30
```

변수 `b`에 값 `30`을 대입했다. 변수 `b`의 값이 `10`에서 `30`으로 변경되었다. 당연한 이야기지만 변수 `a`에는 아무런 영향을 주지 않는다.

최종 결과



여기서 핵심은 `int b = a`라고 했을 때 변수에 들어있는 값을 복사해서 전달한다는 점이다. 따라서 `a=20, b=30`이라고 했을 때 각각 본인의 값만 변경되는 것을 확인할 수 있다.

너무 당연한 이야기를 왜 이렇게 장황하게 풀어서 하지? 라고 생각한다면 이제 진짜 문제를 만나보자.

참조형과 변수 대입

참조형 예시를 위해 `Data` 클래스를 하나 만들자. 이 클래스는 단순히 `int value`라는 멤버 변수를 하나 가진다.

Data

```
package ref;
```

```
public class Data {  
    int value;  
}
```

다음 코드를 보고 어떤 결과가 나올지 먼저 생각해보자. 꼭 먼저 생각해보고 이후에 실행해서 정답을 맞춰보자.

VarChange2

```
package ref;  
  
public class VarChange2 {  
  
    public static void main(String[] args) {  
        Data dataA = new Data();  
        dataA.value = 10;  
        Data dataB = dataA;  
  
        System.out.println("dataA 참조값= " + dataA);  
        System.out.println("dataB 참조값= " + dataB);  
        System.out.println("dataA.value = " + dataA.value);  
        System.out.println("dataB.value = " + dataB.value);  
  
        //dataA 변경  
        dataA.value = 20;  
        System.out.println("변경 dataA.value = 20");  
        System.out.println("dataA.value = " + dataA.value);  
        System.out.println("dataB.value = " + dataB.value);  
  
        //dataB 변경  
        dataB.value = 30;  
        System.out.println("변경 dataB.value = 30");  
        System.out.println("dataA.value = " + dataA.value);  
        System.out.println("dataB.value = " + dataB.value);  
    }  
}
```

실행 결과

```
dataA 참조값=ref.Data@x001  
dataB 참조값=ref.Data@x001  
dataA.value = 10  
dataB.value = 10  
변경 dataA.value = 20  
dataA.value = 20
```

```

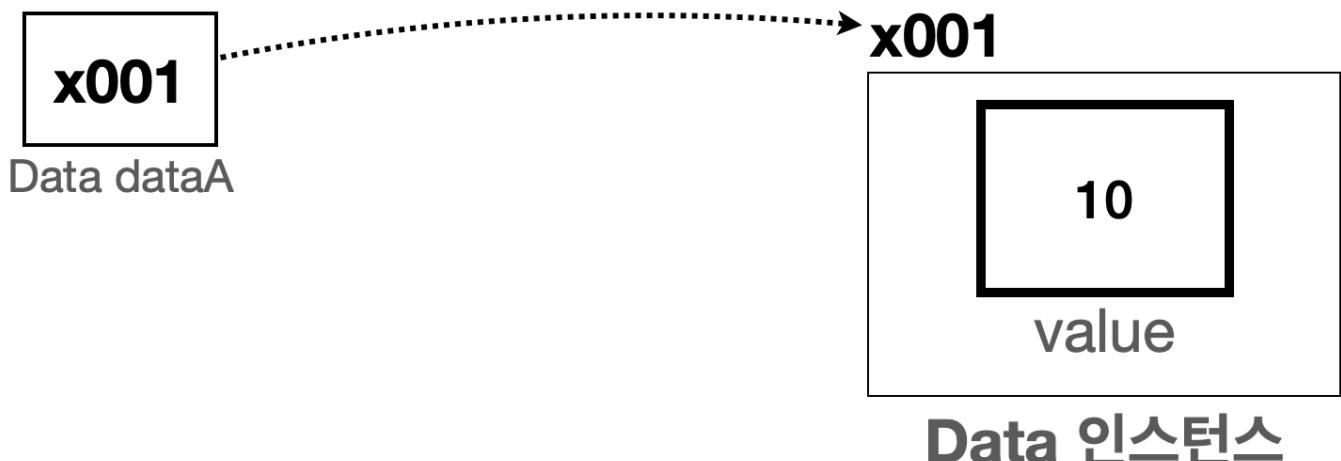
dataB.value = 20
변경 dataB.value = 30
dataA.value = 30
dataB.value = 30

```

그림을 통해 자세히 알아보자.

```
Data dataA = new Data()
```

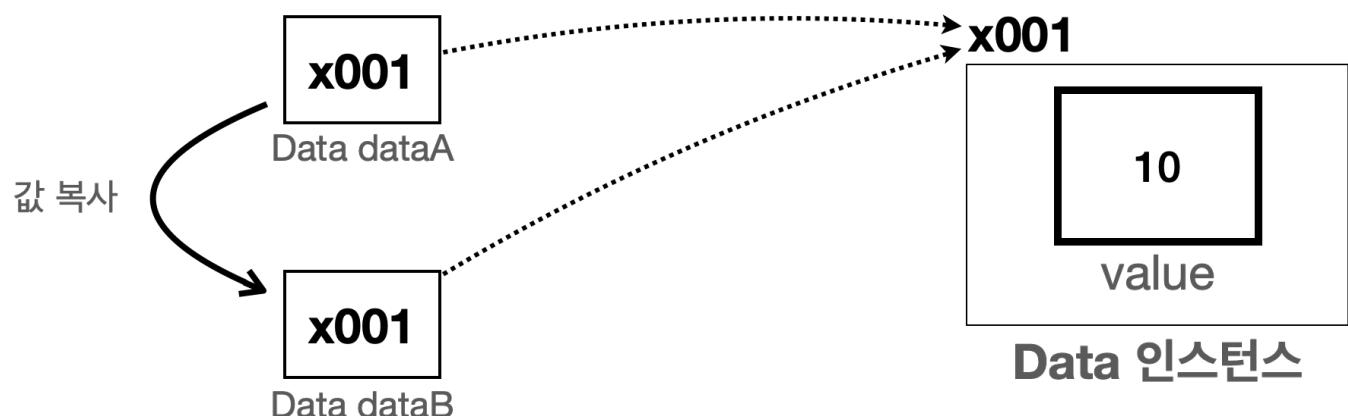
```
dataA.value = 10
```



dataA 변수는 **Data** 클래스를 통해서 만들었기 때문에 참조형이다. 이 변수는 **Data** 형 객체의 참조값을 저장한다.

Data 객체를 생성하고, 참조값을 **dataA**에 저장한다. 그리고 객체의 **value** 변수에 값 **10**을 저장했다.

```
Data dataB = dataA
```



실행 코드

```

Data dataA = new Data();
dataA.value = 10;
Data dataB = dataA;

System.out.println("dataA 참조값=" + dataA);

```

```
System.out.println("dataB 참조값= " + dataB);
System.out.println("dataA.value = " + dataA.value);
System.out.println("dataB.value = " + dataB.value);
```

출력 결과

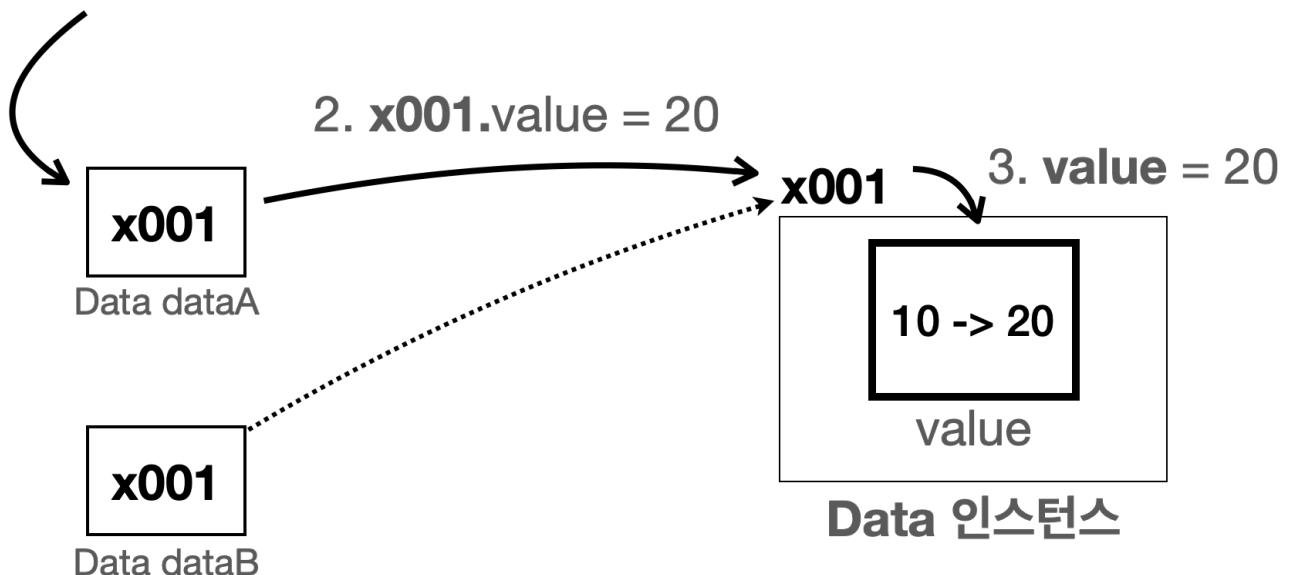
```
dataA = ref.Data@x001
dataB = ref.Data@x001
dataA.value = 10
dataB.value = 10
```

변수의 대입은 변수에 들어있는 값을 복사해서 대입한다. 변수 `dataA`에는 참조값 `x001`이 들어있다. 여기서는 변수 `dataA`에 들어있는 참조값 `x001`을 복사해서 변수 `dataB`에 대입한다. 참고로 변수 `dataA`가 가리키는 인스턴스를 복사하는 것이 아니다! 변수에 들어있는 참조값만 복사해서 전달한다.

이제 `dataA`와 `dataB`에 들어있는 참조값은 같다. 따라서 둘다 같은 `x001 Data` 인스턴스를 가리킨다.

`dataA.value = 20`

1. `dataA.value = 20`



실행 코드

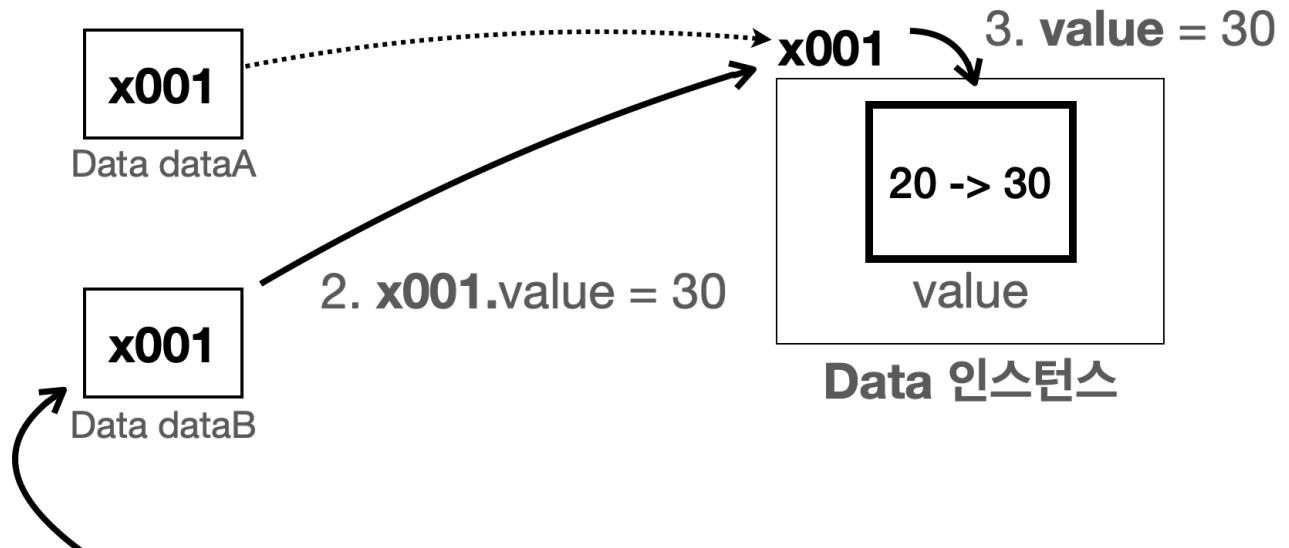
```
dataA.value = 20;
System.out.println("dataA = " + dataA.value);
System.out.println("dataB = " + dataB.value);
```

출력 결과

```
dataA.value = 20
dataB.value = 20
```

`dataA.value = 20` 코드를 실행하면 `dataA`가 가리키는 `x001` 인스턴스의 `value` 값을 10에서 20으로 변경 한다. 그런데 `dataA`와 `dataB`는 같은 `x001` 인스턴스를 참조하기 때문에 `dataA.value`와 `dataB.value`는 둘 다 같은 값인 20을 출력한다.

`dataB.value = 30`



1. `dataB.value = 30`

실행 코드

```
dataB.value = 30;  
System.out.println("dataA.value = " + dataA.value);  
System.out.println("dataB.value = " + dataB.value);
```

출력 결과

```
dataA.value = 30  
dataB.value = 30
```

`dataB.value = 30` 코드를 실행하면 `dataB`가 가리키는 `x001` 인스턴스의 `value` 값을 20에서 30으로 변경 한다. 그런데 `dataA`와 `dataB`는 같은 `x001` 인스턴스를 참조하기 때문에 `dataA.value`와 `dataB.value`는 같은 값인 30을 출력한다.

여기서 핵심은 `Data dataB = dataA`라고 했을 때 변수에 들어있는 값을 복사해서 사용한다는 점이다. 그런데 그 값이 참조값이다. 따라서 `dataA`와 `dataB`는 같은 참조값을 가지게 되고, 두 변수는 같은 객체 인스턴스를 참조하게 된다.

기본형 vs 참조형3 - 메서드 호출

이번에는 기본형과 참조형이 메서드 호출에 따라서 어떻게 달라지는지 알아보자.

대원칙: 자바는 항상 변수의 값을 복사해서 대입한다.

자바에서 변수에 값을 대입하는 것은 변수에 들어 있는 값을 복사해서 대입하는 것이다.

기본형, 참조형 모두 항상 변수에 있는 값을 복사해서 대입한다. 기본형이면 변수에 들어 있는 실제 사용하는 값을 복사해서 대입하고, 참조형이면 변수에 들어 있는 참조값을 복사해서 대입한다.

메서드 호출도 마찬가지이다. 메서드를 호출할 때 사용하는 매개변수(파라미터)도 결국 변수일 뿐이다. 따라서 메서드를 호출할 때 매개변수에 값을 전달하는 것도 앞서 설명한 내용과 같이 값을 복사해서 전달한다.

다음 메서드 호출 코드를 보고 어떤 결과가 나올지 먼저 생각해보자. 너무 쉽고 당연한 내용이지만, 이후에 나올 참조형과 비교를 위해서 한번 정리해보자.

기본형과 메서드 호출

```
package ref;

public class MethodChange1 {

    public static void main(String[] args) {
        int a = 10;
        System.out.println("메서드 호출 전: a = " + a);
        changePrimitive(a);
        System.out.println("메서드 호출 후: a = " + a);
    }

    static void changePrimitive(int x) {
        x = 20;
    }
}
```

실행 결과

```
메서드 호출 전: a = 10
메서드 호출 후: a = 10
```

1. 메서드 호출

```
int a = 10  
changePrimitive(a)
```

1. 메서드 호출, 값 전달: 10

```
changePrimitive(int x) {  
    x = 20  
}
```



메서드를 호출할 때 매개변수 `x`에 변수 `a`의 값을 전달한다. 이 코드는 다음과 같이 해석할 수 있다.

```
int x = a
```

자바에서 변수에 값을 대입하는 것은 항상 값을 복사해서 대입한다. 따라서 변수 `a`, `x` 각각 숫자 10을 가지고 있다.

2. 메서드 안에서 값을 변경

```
int a = 10  
changePrimitive(a)
```

```
changePrimitive(int x) {  
    x = 20  
}
```



메서드 안에서 `x = 20`으로 새로운 값을 대입한다.

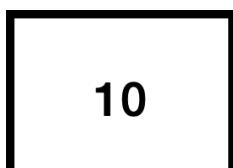
결과적으로 `x`의 값만 20으로 변경되고, `a`의 값은 10으로 유지된다.

3. 메서드 종료

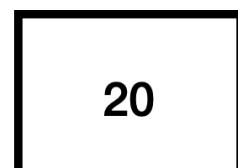
```
int a = 10  
changePrimitive(a)
```

```
changePrimitive(int x) {  
    x = 20  
}
```

3. 메서드 종료



a



x

메서드 종료후 값을 확인해보면 a는 10이 출력되는 것을 확인할 수 있다. 참고로 메서드가 종료되면 매개변수 x는 제거된다.

참조형과 메서드 호출

이번에는 참조형 변수의 메서드 호출에 대해 알아보자.

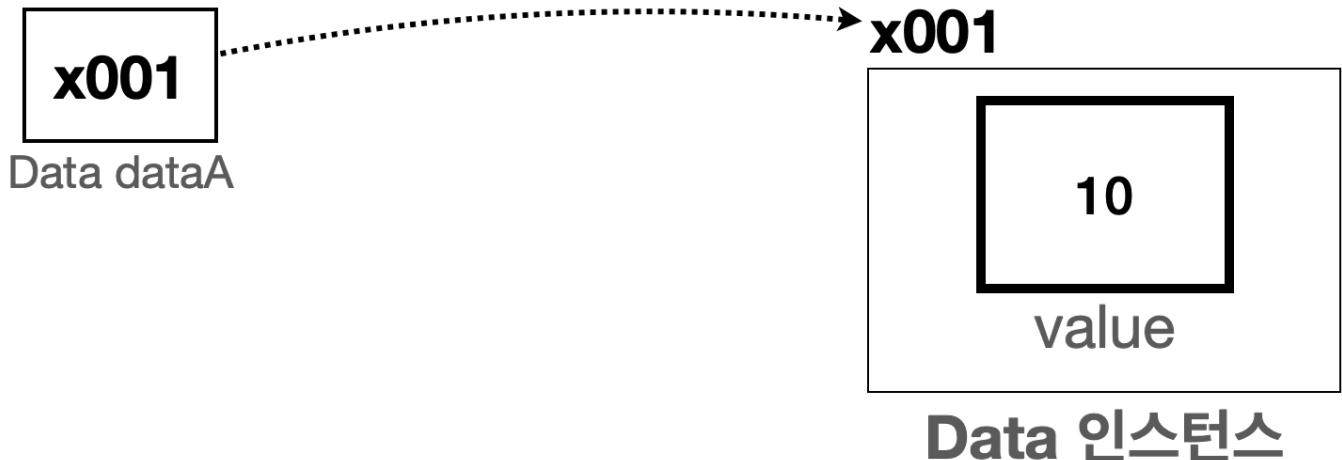
다음 메서드 호출 코드를 보고 어떤 결과가 나올지 먼저 생각해보자.

```
package ref;  
  
public class MethodChange2 {  
  
    public static void main(String[] args) {  
        Data dataA = new Data();  
        dataA.value = 10;  
        System.out.println("메서드 호출 전: dataA.value = " + dataA.value);  
        changeReference(dataA);  
        System.out.println("메서드 호출 후: dataA.value = " + dataA.value);  
    }  
  
    static void changeReference(Data dataX) {  
        dataX.value = 20;  
    }  
}
```

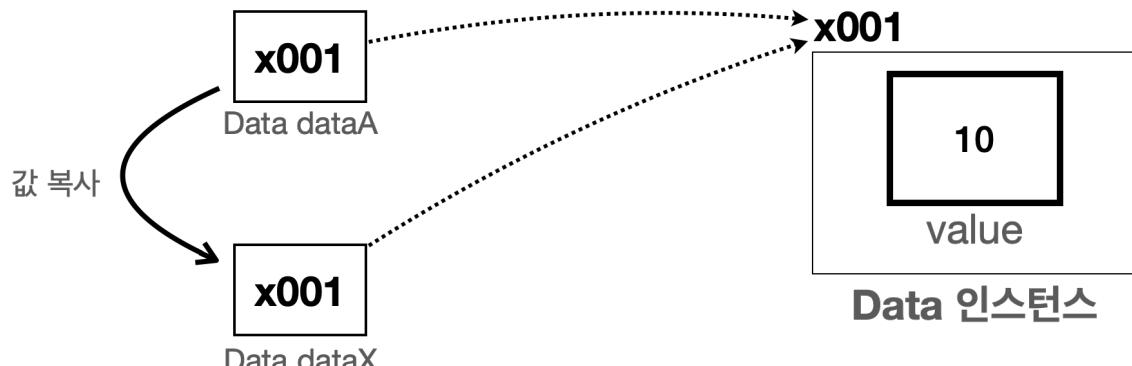
실행 결과

```
메서드 호출 전: dataA.value = 10  
메서드 호출 후: dataA.value = 20
```

Data 인스턴스를 생성하고, 참조값을 `dataA` 변수에 담고 `value`에 숫자 10을 할당한 상태는 다음과 같다.



1. 메서드 호출



메서드를 호출할 때 매개변수 `dataX`에 변수 `dataA`의 값을 전달한다. 이 코드는 다음과 같이 해석할 수 있다.

```
Data dataX = dataA
```

자바에서 변수에 값을 대입하는 것은 항상 값을 복사해서 대입한다. 변수 `dataA`는 참조값 `x001`을 가지고 있으므로

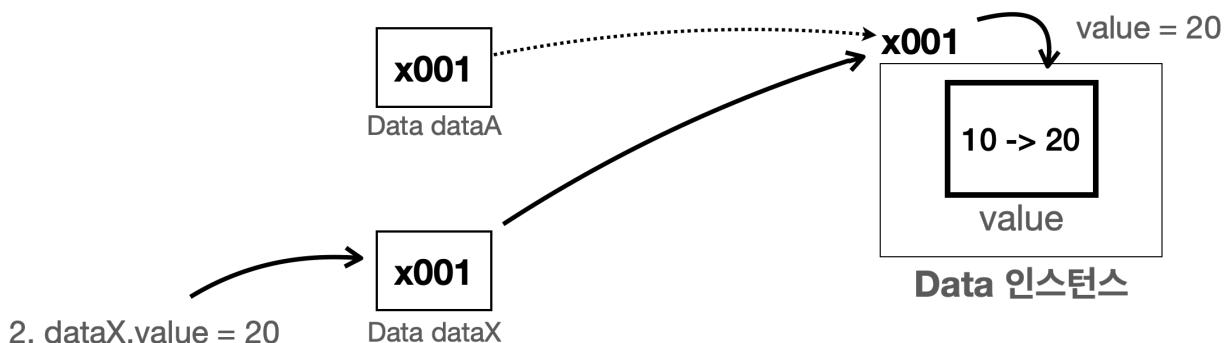
참조값을 복사해서 전달했다. 따라서 변수 `dataA`, `dataX` 둘다 같은 참조값인 `x001`을 가지게 된다.

이제 `dataX`를 통해서도 `x001`에 있는 `Data` 인스턴스에 접근할 수 있다.

2. 메서드 안에서 값을 변경

`changeReference(dataA)`

```
changeReference(Data dataX) {  
    dataX.value = 20  
}
```



메서드 안에서 `dataX.value = 20`으로 새로운 값을 대입한다.

참조값을 통해 `x001` 인스턴스에 접근하고 그 안에 있는 `value`의 값을 20으로 변경했다.

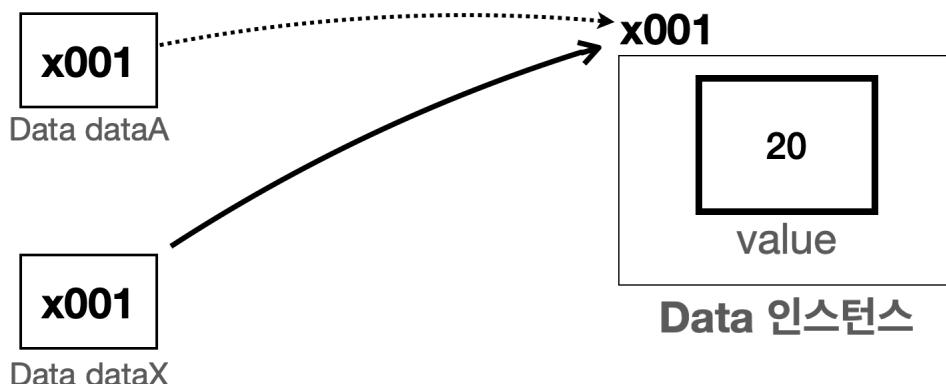
`dataA`, `dataX` 모두 같은 `x001` 인스턴스를 참조하기 때문에 `dataA.value`와 `dataX.value`는 둘다 20이라는 값을 가진다.

3. 메서드 종료

`changeReference(dataA)`

```
changeReference(Data dataX) {  
    dataX.value = 20  
}
```

3. 메서드 종료



메서드 종료후 `dataA.value`의 값을 확인해보면 다음과 같이 20으로 변경된 것을 확인할 수 있다.

```
메서드 호출 전: dataA.value = 10  
메서드 호출 후: dataA.value = 20
```

기본형과 참조형의 메서드 호출

자바에서 메서드의 매개변수(파라미터)는 항상 값에 의해 전달된다. 그러나 이 값이 실제 값이냐, 참조(메모리 주소)값이냐에 따라 동작이 달라진다.

- **기본형**: 메서드로 기본형 데이터를 전달하면, 해당 값이 복사되어 전달된다. 이 경우, 메서드 내부에서 매개변수(파라미터)의 값을 변경해도, 호출자의 변수 값에는 영향이 없다.
- **참조형**: 메서드로 참조형 데이터를 전달하면, 참조값이 복사되어 전달된다. 이 경우, 메서드 내부에서 매개변수(파라미터)로 전달된 객체의 멤버 변수를 변경하면, 호출자의 객체도 변경된다.

참조형과 메서드 호출 - 활용

이전에 보았던 `class1.ClassStart3` 코드에는 중복되는 부분이 2가지 있다.

- `name`, `age`, `grade`에 값을 할당
- 학생 정보를 출력

ClassStart3

```
package class1;

public class ClassStart3 {
    public static void main(String[] args) {
        Student student1;
        student1 = new Student();
        student1.name = "학생1";
        student1.age = 15;
        student1.grade = 90;

        Student student2 = new Student();
        student2.name = "학생2";
        student2.age = 16;
        student2.grade = 80;

        System.out.println("이름:" + student1.name + " 나이:" + student1.age + " 성
적:" + student1.grade);
        System.out.println("이름:" + student2.name + " 나이:" + student2.age + " 성
```

```
적：" + student2.grade);  
}  
}  
}
```

이런 중복은 메서드를 통해 손쉽게 제거할 수 있다.

메서드에 객체 전달

다음과 같이 코드를 작성해보자.

Student

```
package ref;  
  
public class Student {  
    String name;  
    int age;  
    int grade;  
}
```

- ref 패키지에도 Student 클래스를 만든다.

Method1

```
package ref;  
  
public class Method1 {  
    public static void main(String[] args) {  
        Student student1 = new Student();  
        initStudent(student1, "학생1", 15, 90);  
  
        Student student2 = new Student();  
        initStudent(student2, "학생2", 16, 80);  
  
        printStudent(student1);  
        printStudent(student2);  
    }  
  
    static void initStudent(Student student, String name, int age, int grade) {  
        student.name = name;  
        student.age = age;  
        student.grade = grade;  
    }  
}
```

```

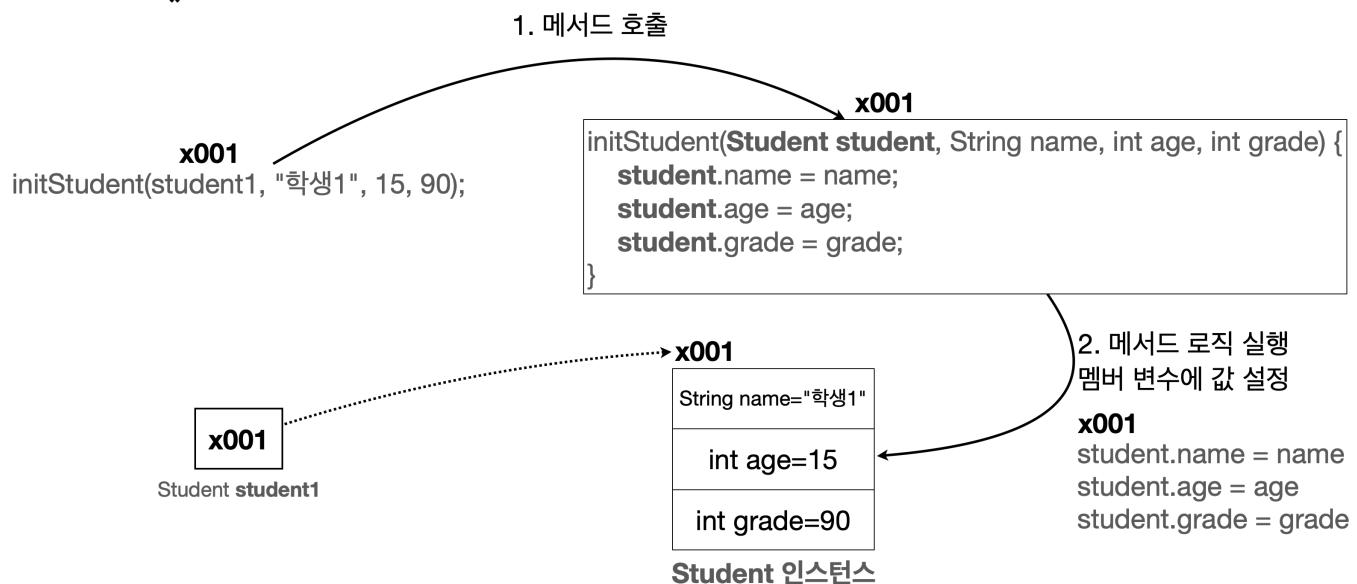
static void printStudent(Student student1) {
    System.out.println("이름:" + student1.name + " 나이:" + student1.age + " 성
적:" + student1.grade);
}

```

참조형은 메서드를 호출할 때 참조값을 전달한다. 따라서 메서드 내부에서 전달된 참조값을 통해 객체의 값을 변경하거나, 값을 읽어서 사용할 수 있다.

- `initStudent(Student student, ...)`: 전달한 학생 객체의 필드에 값을 설정한다.
- `printStudent(Student student, ...)`: 전달한 학생 객체의 필드 값을 읽어서 출력한다.

initStudent() 메서드 호출 분석



- `student1`이 참조하는 `Student` 인스턴스에 값을 편리하게 할당하고 싶어서 `initStudent()` 메서드를 만들었다.
- 이 메서드를 호출하면서 `student1`을 전달한다. 그러면 `student1`의 참조값이 매개변수 `student`에 전달된다. 이 참조값을 통해 `initStudent()` 메서드 안에서 `student1`이 참조하는 것과 동일한 `x001` `Student` 인스턴스에 접근하고 값을 변경할 수 있다.

주의!

```

package ref;

import class1.Student;

public class Method1 {
    ...
}

• import class1.Student; 이 선언되어 있으면 안된다.

```

- 이렇게 되면 `class1` 패키지에서 선언한 `Student`를 사용하게 된다. 이 경우 접근 제어자 때문에 컴파일 오류가 발생한다.
- 만약 선언되어 있다면 삭제하자. 삭제하면 같은 패키지에 있는 `ref.Student`를 사용한다.

메서드에서 객체 반환

조금 더 나아가보자. 다음 코드에도 중복이 있다.

```
Student student1 = new Student();
initStudent(student1, "학생1", 15, 90);

Student student2 = new Student();
initStudent(student2, "학생2", 16, 80);
```

바로 객체를 생성하고, 초기값을 설정하는 부분이다. 이렇게 2번 반복되는 부분을 하나로 합쳐보자.

다음과 같이 기존 코드를 변경해보자.

Method2

```
package ref;

public class Method2 {
    public static void main(String[] args) {
        Student student1 = createStudent("학생1", 15, 90);
        Student student2 = createStudent("학생2", 16, 80);

        printStudent(student1);
        printStudent(student2);
    }

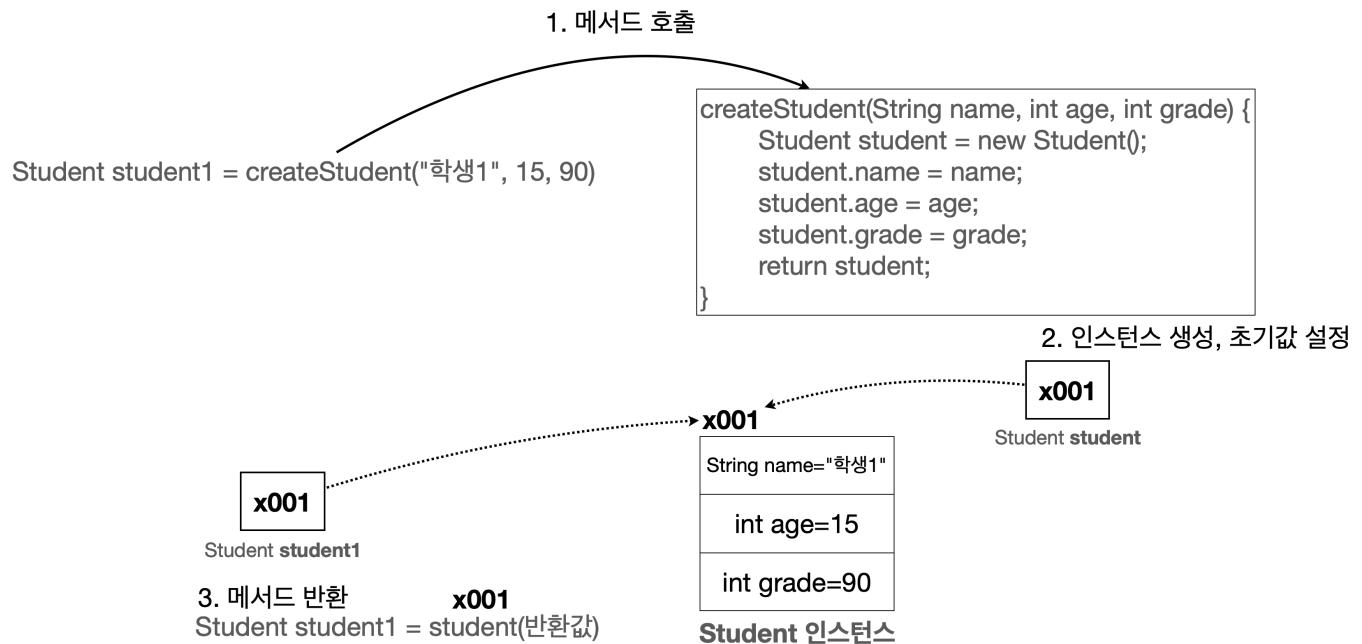
    static Student createStudent(String name, int age, int grade) {
        Student student = new Student();
        student.name = name;
        student.age = age;
        student.grade = grade;
        return student;
    }

    static void printStudent(Student student1) {
        System.out.println("이름:" + student1.name + " 나이:" + student1.age + " 성
적:" + student1.grade);
    }
}
```

`createStudent()`라는 메서드를 만들고 객체를 생성하는 부분도 이 메서드안에 함께 포함했다. 이제 이 메서드 하나로 객체의 생성과 초기값 설정을 모두 처리한다.

그런데 메서드 안에서 객체를 생성했기 때문에 만들어진 객체를 메서드 밖에서 사용할 수 있게 돌려주어야 한다. 그래야 메서드 밖에서 이 객체를 사용할 수 있다. 메서드는 호출 결과를 반환(`return`)을 할 수 있다. 메서드의 반환 기능을 사용해서 만들어진 객체의 참조값을 메서드 밖으로 반환하면 된다.

`createStudent()` 메서드 호출 분석



메서드 내부에서 인스턴스를 생성한 후에 참조값을 메서드 외부로 반환했다. 이 참조값만 있으면 해당 인스턴스에 접근할 수 있다. 여기서는 `student1`에 참조값을 보관하고 사용한다.

진행 과정

```
Student student1 = createStudent("학생1", 15, 90) //메서드 호출후 결과 반환  
Student student1 = student(x001) //참조형인 student를 반환  
Student student1 = x001 //student의 참조값 대입  
student1 = x001
```

`createStudent()`는 생성한 `Student` 인스턴스의 참조값을 반환한다. 이렇게 반환된 참조값을 `student1` 변수에 저장했다. 앞으로는 `student1`을 통해 `Student` 인스턴스를 사용할 수 있다.

변수와 초기화

변수의 종류

- 멤버 변수(필드): 클래스에 선언
- 지역 변수: 메서드에 선언, 매개변수도 지역 변수의 한 종류이다.

멤버 변수, 필드 예시

```
public class Student {  
    String name;  
    int age;  
    int grade;  
}
```

name, age, grade는 멤버 변수이다.

지역 변수 예시

```
public class ClassStart3 {  
    public static void main(String[] args) {  
        Student student1;  
        student1 = new Student();  
        Student student2 = new Student();  
    }  
}
```

student1, student2는 지역 변수이다.

```
public class MethodChange1 {  
  
    public static void main(String[] args) {  
        int a = 10;  
        System.out.println("메서드 호출 전: a = " + a);  
        changePrimitive(a);  
        System.out.println("메서드 호출 후: a = " + a);  
    }  
  
    public static void changePrimitive(int x) {  
        x = 20;  
    }  
}
```

a, x(매개변수)는 지역 변수이다.

지역 변수는 이름 그대로 특정 지역에서만 사용되는 변수라는 뜻이다. 예를 들어서 변수 x는 changePrimitive() 메서드의 블록에서만 사용된다. changePrimitive() 메서드가 끝나면 제거된다. a 변수도 마찬가지이다.

```
main() 메서드가 끝나면 제거된다.
```

변수의 값 초기화

- 멤버 변수: 자동 초기화
 - 인스턴스의 멤버 변수는 인스턴스를 생성할 때 자동으로 초기화된다.
 - 숫자(`int`) = `0`, `boolean` = `false`, 참조형 = `null` (`null` 값은 참조할 대상이 없다는 뜻으로 사용된다.)
 - 개발자가 초기값을 직접 지정할 수 있다.
- 지역 변수: 수동 초기화
 - 지역 변수는 항상 직접 초기화해야 한다.

멤버 변수의 초기화를 살펴보자

InitData

```
package ref;

public class InitData {
    int value1; //초기화 하지 않음
    int value2 = 10; //10으로 초기화
}
```

`value1`은 초기값을 지정하지 않았고, `value2`는 초기값을 10으로 지정했다.

InitMain

```
package ref;

public class InitMain {
    public static void main(String[] args) {
        InitData data = new InitData();
        System.out.println("value1 = " + data.value1);
        System.out.println("value2 = " + data.value2);
    }
}
```

실행 결과

```
value1 = 0
value2 = 10
```

`value1`은 초기값을 지정하지 않았지만 멤버 변수는 자동으로 초기화 된다. 숫자는 0으로 초기화된다.

`value2`는 10으로 초기값을 지정해두었기 때문에 객체를 생성할 때 10으로 초기화된다.

null

택배를 보낼 때 제품은 준비가 되었지만, 보낼 주소지가 아직 결정되지 않아서, 주소지가 결정될 때 까지는 주소지를 비워두어야 할 수 있다.

참조형 변수에는 항상 객체가 있는 위치를 가리키는 참조값이 들어간다. 그런데 아직 가리키는 대상이 없거나, 가리키는 대상을 나중에 입력하고 싶다면 어떻게 해야할까?

참조형 변수에서 아직 가리키는 대상이 없다면 `null`이라는 특별한 값을 넣어둘 수 있다. `null`은 값이 존재하지 않는, 없다는 뜻이다.

코드를 통해서 `null` 값에 대해 알아보자.

null 값 할당

```
package ref;

public class Data {
    int value;
}
```

앞서 만들었던 `Data` 클래스이다. 기존 코드를 그대로 유지하면 된다.

```
package ref;

public class NullMain1 {
    public static void main(String[] args) {
        Data data = null;
        System.out.println("1. data = " + data);
        data = new Data();
        System.out.println("2. data = " + data);
        data = null;
        System.out.println("3. data = " + data);
    }
}
```

실행 결과

```
1. data = null  
2. data = ref.Data@x001  
3. data = null
```

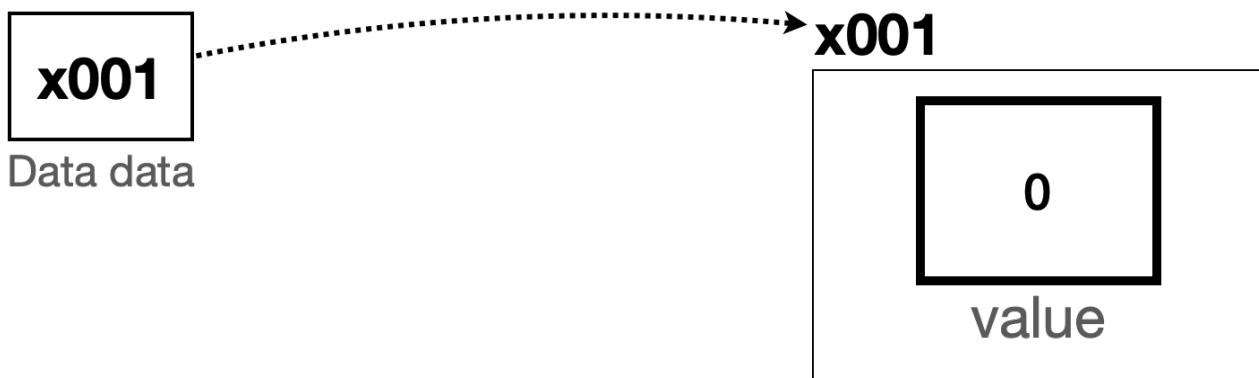
Data data = null



Data data

Data 타입을 받을 수 있는 참조형 변수 data를 만들었다. 그리고 여기에 null 값을 할당했다. 따라서 data 변수에는 아직 가리키는 객체가 없다는 뜻이다.

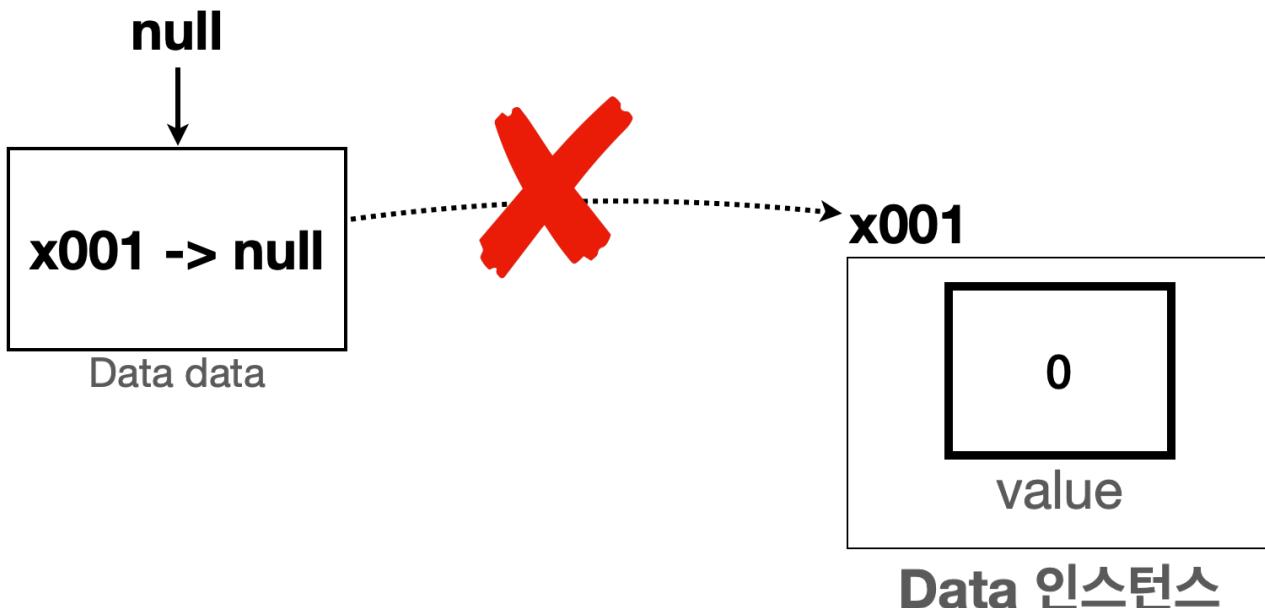
data = new Data()



Data 인스턴스

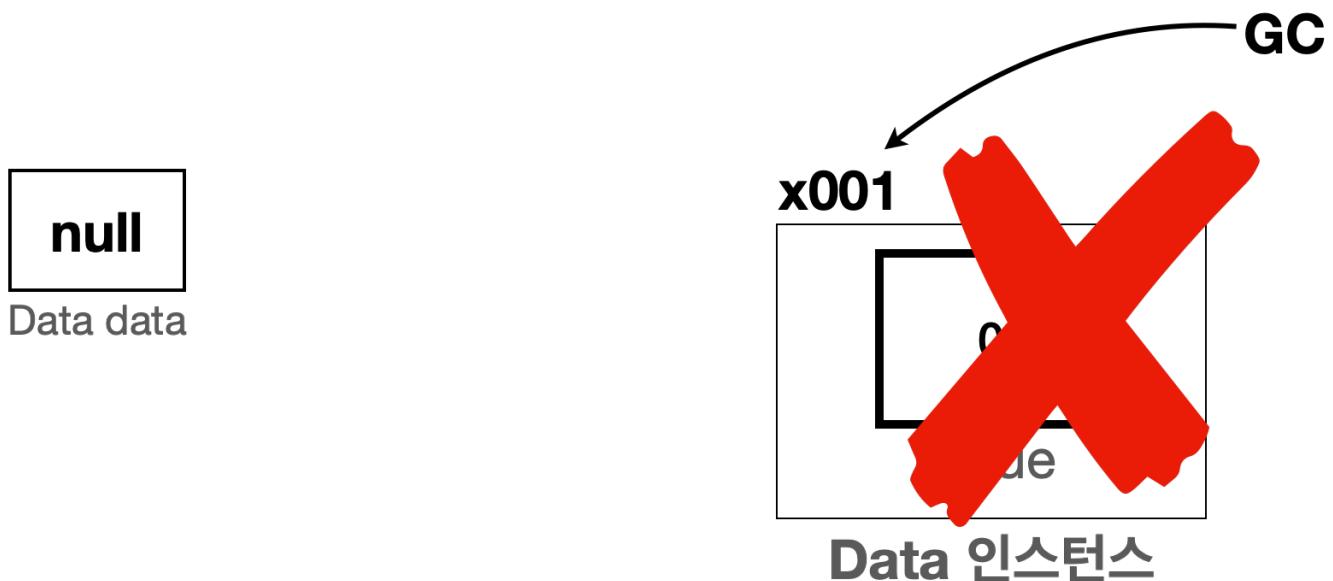
이후에 새로운 Data 객체를 생성해서 그 참조값을 data 변수에 할당했다. 이제 data 변수가 참조할 객체가 존재한다.

data = null



마지막에는 `data`에 다시 `null` 값을 할당했다. 이렇게 하면 `data` 변수는 앞서 만든 `Data` 인스턴스를 더는 참조하지 않는다.

GC - 아무도 참조하지 않는 인스턴스의 최후



`data`에 `null`을 할당했다. 따라서 앞서 생성한 `x001` `Data` 인스턴스를 더는 아무도 참조하지 않는다. 이렇게 아무도 참조하지 않게 되면 `x001`이라는 참조값을 다시 구할 방법이 없다. 따라서 해당 인스턴스에 다시 접근할 방법이 없다.

이렇게 아무도 참조하지 않는 인스턴스는 사용되지 않고 메모리 용량만 차지할 뿐이다.

C와 같은 과거 프로그래밍 언어는 개발자가 직접 명령어를 사용해서 인스턴스를 메모리에서 제거해야 했다. 만약 실제로 인스턴스 삭제를 누락하면 메모리에 사용하지 않는 객체가 가득해져서 메모리 부족 오류가 발생하게 된다.

자바는 이런 과정을 자동으로 처리해준다. 아무도 참조하지 않는 인스턴스가 있으면 JVM의 GC(가비지 컬렉션)가 더 이상 사용하지 않는 인스턴스라 판단하고 해당 인스턴스를 자동으로 메모리에서 제거해준다.

객체는 해당 객체를 참조하는 곳이 있으면, JVM이 종료할 때 까지 계속 생존한다. 그런데 중간에 해당 객체를 참조하는 곳이 모두 사라지면 그때 JVM은 필요 없는 객체로 판단하고 GC(가비지 컬렉션)를 사용해서 제거한다.

NullPointerException

택배를 보낼 때 주소지 없이 택배를 발송하면 어떤 문제가 발생할까? 만약 참조값 없이 객체를 찾아가면 어떤 문제가 발생할까?

이 경우 `NullPointerException`이라는 예외가 발생하는데, 개발자를 가장 많이 괴롭히는 예외이다.

`NullPointerException`은 이름 그대로 `null`을 가리킨다(Pointer)인데, 이때 발생하는 예외(Exception)다. `null`은 없다는 뜻이므로 결국 주소가 없는 곳을 찾아갈 때 발생하는 예외이다.

객체를 참조할 때는 `.` (dot)을 사용한다. 이렇게 하면 참조값을 사용해서 해당 객체를 찾아갈 수 있다. 그런데 참조값이 `null`이라면 값이 없다는 뜻이므로, 찾아갈 수 있는 객체(인스턴스)가 없다. `NullPointerException`은 이처럼 `null`에 `.` (dot)을 찍었을 때 발생한다.

예제를 통해서 확인해보자.

```
package ref;

public class NullMain2 {
    public static void main(String[] args) {
        Data data = null;
        data.value = 10; // NullPointerException 예외 발생
        System.out.println("data = " + data.value);
    }
}
```

`data` 참조형 변수에는 `null` 값이 들어가 있다. 그런데 `data.value = 10`이라고 하면 어떻게 될까?

```
data.value = 10
null.value = 10 //data에는 null 값이 들어있다.
```

결과적으로 `null` 값은 참조할 주소가 존재하지 않는다는 뜻이다. 따라서 참조할 객체 인스턴스가 존재하지 않으므로 다음과 같이 `java.lang.NullPointerException`이 발생하고, 프로그램이 종료된다. 참고로 예외가 발생했기 때문에 그 다음 로직은 수행되지 않는다.

실행 결과

```
Exception in thread "main" java.lang.NullPointerException: Cannot assign field
"value" because "data" is null
at ref.NullMain2.main(NullMain2.java:6)
```

멤버 변수와 null

앞선 예제와 같이 지역 변수의 경우에는 null 문제를 파악하는 것이 어렵지 않다. 다음과 같이 멤버 변수가 null인 경우에는 주의가 필요하다.

```
package ref;

public class Data {
    int value;
}
```

기존의 Data 클래스를 사용한다.

```
package ref;

public class BigData {
    Data data;
    int count;
}
```

BigData 클래스는 Data data, int count 두 변수를 가진다.

```
package ref;

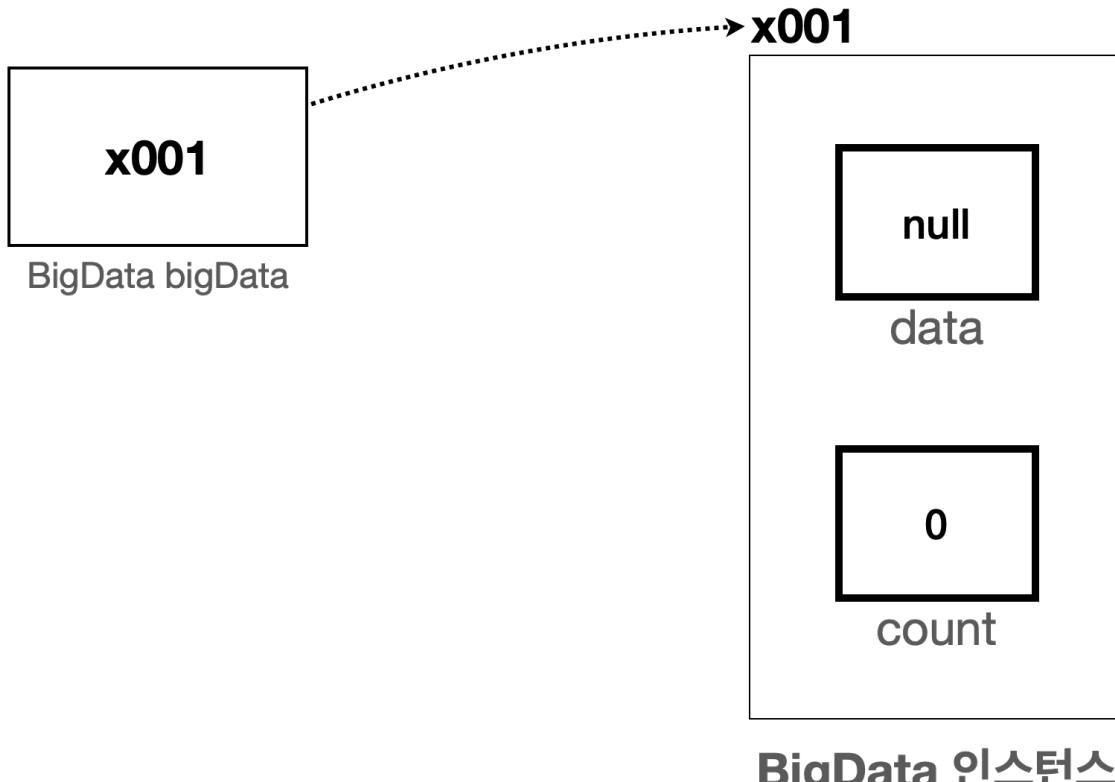
public class NullMain3 {
    public static void main(String[] args) {
        BigData bigData = new BigData();
        System.out.println("bigData.count=" + bigData.count);
        System.out.println("bigData.data=" + bigData.data);

        //NullPointerException
        System.out.println("bigData.data.value=" + bigData.data.value);
    }
}
```

실행 결과

```
bigData.count=0
```

```
bigData.data=null  
Exception in thread "main" java.lang.NullPointerException: Cannot read field  
"value" because "bigData.data" is null  
at ref.NullMain3.main(NullMain3.java:10)
```



BigData 인스턴스

`BigData`를 생성하면 `BigData`의 인스턴스가 생성된다. 이때 `BigData` 인스턴스의 멤버 변수에 초기화가 일어나는데, `BigData`의 `data` 멤버 변수는 참조형이므로 `null`로 초기화 된다. `count` 멤버 변수는 숫자이므로 `0`으로 초기화된다.

- `bigData.count`를 출력하면 `0`이 출력된다.
- `bigData.data`를 출력하면 참조값인 `null`이 출력된다. 이 변수는 아직 아무것도 참조하고 있지 않다.
- `bigData.data.value`를 출력하면 `data`의 값이 `null`이므로 `null`에 `.(dot)`을 찍게 되고, 따라서 참조할 곳이 없으므로 `NullPointerException` 예외가 발생한다.

예외 발생 과정

```
bigData.data.value  
x001.data.value //bigData는 x001 참조값을 가진다.  
null.value //x001.data는 null 값을 가진다.  
NullPointerException //null 값에 .(dot)을 찍으면 예외가 발생한다.
```

이 문제를 해결하려면 `Data` 인스턴스를 만들고 `BigData.data` 멤버 변수에 참조값을 할당하면 된다.

```

package ref;

public class NullMain4 {
    public static void main(String[] args) {
        BigData bigData = new BigData();
        bigData.data = new Data();
        System.out.println("bigData.count=" + bigData.count);
        System.out.println("bigData.data=" + bigData.data);
        System.out.println("bigData.data.value=" + bigData.data.value);
    }
}

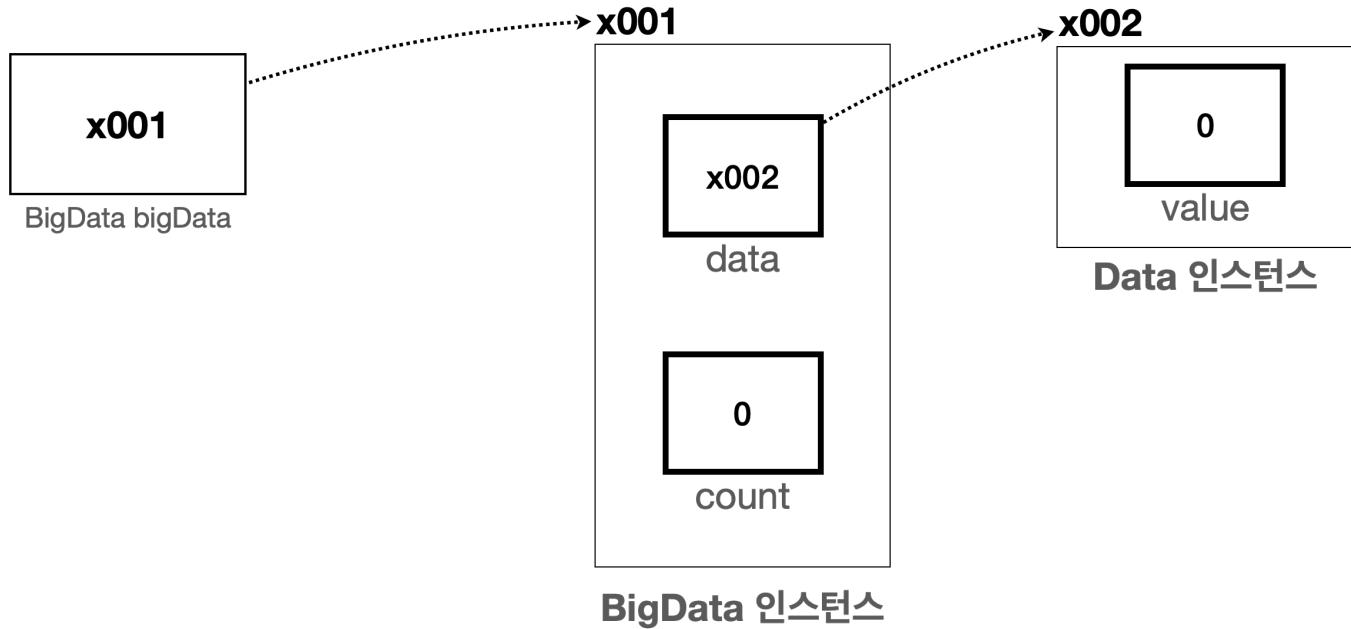
```

실행 결과

```

bigData.count=0
bigData.data=ref.Data@x002
bigData.data.value=0

```



실행 과정

```

bigData.data.value
x001.data.value //bigData는 x001 참조값을 가진다.
x002.value //x001.data는 x002 값을 가진다.
0 // 최종 결과

```

정리

NullPointerException이 발생하면 null 값에 .(dot)을 찍었다고 생각하면 문제를 쉽게 찾을 수 있다.

문제와 풀이

문제: 상품 주문 시스템 개발 - 리팩토링

문제 설명

앞서 만들었던 다음 클래스에 있는 "상품 주문 시스템"을 리팩토링 하자.

```
class1.ex.ProductOrderMain
```

당신은 온라인 상점의 주문 관리 시스템을 만들려고 한다.

먼저, 상품 주문 정보를 담을 수 있는 `ProductOrder` 클래스를 만들어보자.

요구 사항

`ProductOrder` 클래스는 다음과 같은 멤버 변수를 포함해야 한다.

- 상품명 (`productName`)
- 가격 (`price`)
- 주문 수량 (`quantity`)

예시 코드 구조

```
package ref.ex;

public class ProductOrder {
    String productName;
    int price;
    int quantity;
}
```

- 이 코드도 `ref.ex` 패키지에 새로 만들어야 한다.

다음으로 `ref.ex.ProductOrderMain2` 클래스 안에 `main()` 메서드를 포함하여, 여러 상품의 주문 정보를 배열로 관리하고, 그 정보들을 출력하고, 최종 결제 금액을 계산하여 출력하자. 이 클래스에서는 다음과 같은 메서드를 포함해야 합니다:

- `static ProductOrder createOrder(String productName, int price, int quantity)`
 - `ProductOrder`를 생성하고 매개변수로 초기값을 설정합니다. 마지막으로 생성한 `ProductOrder`를 반환합니다.
- `static void printOrders(ProductOrder[] orders)`

- 배열을 받아서 배열에 들어있는 전체 `ProductOrder`의 상품명, 가격, 수량을 출력합니다.
- `static int getTotalAmount(ProductOrder[] orders)`
 - 배열을 받아서 배열에 들어있는 전체 `ProductOrder`의 총 결제 금액을 계산하고, 계산 결과를 반환합니다.

예시 코드 구조

```
package ref.ex;

public class ProductOrderMain2 {
    public static void main(String[] args) {
        // 여러 상품의 주문 정보를 담는 배열 생성
        // createOrder()를 여러번 사용해서 상품 주문 정보들을 생성하고 배열에 저장
        // printOrders()를 사용해서 상품 주문 정보 출력
        // getTotalAmount()를 사용해서 총 결제 금액 계산
        // 총 결제 금액 출력
    }
}
```

출력 예시

```
상품명: 두부, 가격: 2000, 수량: 2
상품명: 김치, 가격: 5000, 수량: 1
상품명: 콜라, 가격: 1500, 수량: 2
총 결제 금액: 12000
```

정답

```
package ref.ex;

public class ProductOrder {
    String productName;
    int price;
    int quantity;
}
```

```
package ref.ex;

public class ProductOrderMain2 {

    public static void main(String[] args) {
        ProductOrder[] orders = new ProductOrder[3];
        orders[0] = createOrder("두부", 2000, 2);
    }
}
```

```

        orders[1] = createOrder("김치", 5000, 1);
        orders[2] = createOrder("콜라", 1500, 2);

        printOrders(orders);
        int totalAmount = getTotalAmount(orders);
        System.out.println("총 결제 금액: " + totalAmount);
    }

    static ProductOrder createOrder(String productName, int price, int quantity)
    {
        ProductOrder order = new ProductOrder();
        order.productName = productName;
        order.price = price;
        order.quantity = quantity;
        return order;
    }

    static void printOrders(ProductOrder[] orders) {
        for (ProductOrder order : orders) {
            System.out.println("상품명: " + order.productName + ", 가격: " +
order.price + ", 수량: " + order.quantity);
        }
    }

    static int getTotalAmount(ProductOrder[] orders) {
        int totalAmount = 0;
        for (ProductOrder order : orders) {
            totalAmount += order.price * order.quantity;
        }
        return totalAmount;
    }
}

```

문제: 상품 주문 시스템 개발 - 사용자 입력

문제 설명

앞서 만든 상품 주문 시스템을 사용자 입력을 받도록 개선해보자.

요구 사항

- 아래 입력, 출력 예시를 참고해서 다음 사항을 적용하자.

- 주문 수량을 입력 받자
 - 예) 입력할 주문의 개수를 입력하세요:
- 가격, 수량, 상품명을 입력 받자
 - 입력시 상품 순서를 알 수 있게 "n번째 주문 정보를 입력하세요." 라는 메시지를 출력하세요.
- 입력이 끝나면 등록한 상품과 총 결제 금액을 출력하자.

입력, 출력 예시

입력할 주문의 개수를 입력하세요: 3

1번째 주문 정보를 입력하세요.

상품명: 두부

가격: 2000

수량: 2

2번째 주문 정보를 입력하세요.

상품명: 김치

가격: 5000

수량: 1

3번째 주문 정보를 입력하세요.

상품명: 콜라

가격: 1500

수량: 2

상품명: 두부, 가격: 2000, 수량: 2

상품명: 김치, 가격: 5000, 수량: 1

상품명: 콜라, 가격: 1500, 수량: 2

총 결제 금액: 12000

정답

```
package ref.ex;

import java.util.Scanner;

public class ProductOrderMain3 {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("입력할 주문의 개수를 입력하세요: ");
        int n = scanner.nextInt();
        scanner.nextLine();

        ProductOrder[] orders = new ProductOrder[n];

        for (int i = 0; i < orders.length; i++) {
            System.out.print("상품명: ");
            String productName = scanner.nextLine();
            System.out.print("가격: ");
            int price = scanner.nextInt();
            scanner.nextLine();
            System.out.print("수량: ");
            int quantity = scanner.nextInt();
            scanner.nextLine();

            orders[i] = new ProductOrder(productName, price, quantity);
        }

        System.out.println("총 결제 금액: " + calculateTotalPrice(orders));
    }

    private static int calculateTotalPrice(ProductOrder[] orders) {
        int totalPrice = 0;
        for (ProductOrder order : orders) {
            totalPrice += order.getPrice() * order.getQuantity();
        }
        return totalPrice;
    }
}
```

```

        System.out.println((i + 1) + "번째 주문 정보를 입력하세요.");
    }

    System.out.print("상품명: ");
    String productName = scanner.nextLine();

    System.out.print("가격: ");
    int price = scanner.nextInt();

    System.out.print("수량: ");
    int quantity = scanner.nextInt();
    scanner.nextLine(); // 입력 버퍼를 비우기 위한 코드

    orders[i] = createOrder(productName, price, quantity);
}

printOrders(orders);
int totalAmount = getTotalAmount(orders);
System.out.println("총 결제 금액: " + totalAmount);
}

static ProductOrder createOrder(String productName, int price, int quantity)
{
    ProductOrder order1 = new ProductOrder();
    order1.productName = productName;
    order1.price = price;
    order1.quantity = quantity;
    return order1;
}

static void printOrders(ProductOrder[] orders) {
    for (ProductOrder order : orders) {
        System.out.println("상품명: " + order.productName + ", 가격: " +
order.price + ", 수량: " + order.quantity);
    }
}

static int getTotalAmount(ProductOrder[] orders) {
    int totalAmount = 0;
    for (ProductOrder order : orders) {
        totalAmount += order.price * order.quantity;
    }
    return totalAmount;
}

```

}

정리

대원칙: 자바는 항상 변수의 값을 복사해서 대입한다.

자바에서 변수에 값을 대입하는 것은 변수에 들어 있는 값을 복사해서 대입하는 것이다.

기본형, 참조형 모두 항상 변수에 있는 값을 복사해서 대입한다. 기본형이면 변수에 들어 있는 실제 사용하는 값을 복사해서 대입하고, 참조형이면 변수에 들어 있는 참조값을 복사해서 대입한다.

기본형이든 참조형이든 변수의 값을 대입하는 방식은 같다. 하지만 기본형과 참조형에 따라 동작하는 방식이 달라진다.

기본형 vs 참조형 - 기본

- 자바의 데이터 타입을 가장 크게 보면 기본형과 참조형으로 나눌 수 있다.
- 기본형을 제외한 나머지 변수는 모두 참조형이다. 클래스와 배열을 다루는 변수는, 참조형이다.
- 기본형 변수는 값을 직접 저장하지만, 참조형 변수는 참조(주소)를 저장한다.
- 기본형 변수는 산술 연산을 수행할 수 있지만, 참조형 변수는 산술 연산을 수행할 수 없다.
- 기본형 변수는 `null`을 할당할 수 없지만, 참조형 변수는 `null`을 할당할 수 있다.

기본형 vs 참조형 - 대입

- 기본형과 참조형 모두 대입시 변수 안에 있는 값을 읽고 복사해서 전달한다.
- 기본형은 사용하는 값을 복사해서 전달하고, 참조형은 참조값을 복사해서 전달한다! 이것이 중요하다. 실제 인스턴스가 복사되는 것이 아니다. 인스턴스를 가리키는 참조값을 복사해서 전달하는 것이다! 따라서 하나의 인스턴스를 여러곳에서 참조할 수 있다.
- 헛갈리면 그냥 변수 안에 들어간 값을 떠올려보자. 기본형은 사용하는 값이, 참조형은 참조값이 들어있다! 변수에 어떤 값이 들어있든간에 그 값을 그대로 복사해서 전달한다.

기본형 vs 참조형 - 메서드 호출

- 메서드 호출시 기본형은 메서드 내부에서 매개변수(파라미터)의 값을 변경해도 호출자의 변수 값에는 영향이 없다.
- 메서드 호출시 참조형은 메서드 내부에서 매개변수(파라미터)로 전달된 객체의 멤버 변수를 변경하면, 호출자의 객체도 변경된다.

3. 객체 지향 프로그래밍

#1.인강/0.자바/2.자바-기본

- /절차 지향 프로그래밍1 - 시작
- /절차 지향 프로그래밍2 - 데이터 묶음
- /절차 지향 프로그래밍3 - 메서드 추출
- /클래스와 메서드
- /객체 지향 프로그래밍
- /문제와 풀이
- /정리

절차 지향 프로그래밍1 - 시작

절차 지향 프로그래밍 vs 객체 지향 프로그래밍

프로그래밍 방식은 크게 절차 지향 프로그래밍과 객체 지향 프로그래밍으로 나눌 수 있다.

절차 지향 프로그래밍

- 절차 지향 프로그래밍은 이름 그대로 절차를 지향한다. 쉽게 이야기해서 실행 순서를 중요하게 생각하는 방식이다.
- 절차 지향 프로그래밍은 프로그램의 흐름을 순차적으로 따르며 처리하는 방식이다. 즉, "어떻게"를 중심으로 프로그래밍 한다.

객체 지향 프로그래밍

- 객체 지향 프로그래밍은 이름 그대로 객체를 지향한다. 쉽게 이야기해서 객체를 중요하게 생각하는 방식이다.
- 객체 지향 프로그래밍은 실제 세계의 사물이나 사건을 객체로 보고, 이러한 객체들 간의 상호작용을 중심으로 프로그래밍하는 방식이다. 즉, "무엇을" 중심으로 프로그래밍 한다.

둘의 중요한 차이

- 절차 지향은 데이터와 해당 데이터에 대한 처리 방식이 분리되어 있다. 반면 객체 지향에서는 데이터와 그 데이터에 대한 행동(메서드)이 하나의 '객체' 안에 함께 포함되어 있다.

우리는 지금까지 클래스와 객체를 사용해서 관련 데이터를 묶어서 사용하는 방법을 학습했다.

그럼 앞서 배운 것처럼 단순히 객체를 사용하기만 하면 객체 지향 프로그래밍이라 할 수 있을까?

사실 지금까지 우리가 작성한 모든 프로그램은 절차 지향 프로그램이다.

그렇다면 무엇이 객체 지향 프로그래밍이란 말인가?

절차 지향에서 객체 지향으로 점진적으로 코드를 변경해보면서 객체 지향 프로그래밍을 이해해보자.

문제: 음악 플레이어 만들기

음악 플레이어를 만들어보자.

요구 사항:

1. 음악 플레이어를 켜고 끌 수 있어야 한다.
2. 음악 프레이어의 볼륨을 증가, 감소할 수 있어야 한다.
3. 음악 플레이어의 상태를 확인할 수 있어야 한다.

예시 출력:

```
음악 플레이어를 시작합니다  
음악 플레이어 볼륨:1  
음악 플레이어 볼륨:2  
음악 플레이어 볼륨:1  
음악 플레이어 상태 확인  
음악 플레이어 ON, 볼륨:1  
음악 플레이어를 종료합니다
```

절차 지향 음악 플레이어1

```
package oop1;  
  
public class MusicPlayerMain1 {  
  
    public static void main(String[] args) {  
        int volume = 0;  
        boolean isOn = false;  
  
        //음악 플레이어 켜기  
        isOn = true;  
        System.out.println("음악 플레이어를 시작합니다");  
  
        //볼륨 증가  
        volume++;  
        System.out.println("음악 플레이어 볼륨:" + volume);  
  
        //볼륨 증가
```

```

volume++;
System.out.println("음악 플레이어 볼륨:" + volume);

//볼륨 감소
volume--;
System.out.println("음악 플레이어 볼륨:" + volume);

//음악 플레이어 상태
System.out.println("음악 플레이어 상태 확인");
if (isOn) {
    System.out.println("음악 플레이어 ON, 볼륨:" + volume);
} else {
    System.out.println("음악 플레이어 OFF");
}

//음악 플레이어 끄기
isOn = false;
System.out.println("음악 플레이어를 종료합니다");
}

}

```

실행 결과

```

음악 플레이어를 시작합니다
음악 플레이어 볼륨:1
음악 플레이어 볼륨:2
음악 플레이어 볼륨:1
음악 플레이어 상태 확인
음악 플레이어 ON, 볼륨:1
음악 플레이어를 종료합니다

```

순서대로 프로그램이 작동하도록 단순하게 작성했다. 이 코드를 점진적으로 변경해보자.

절차 지향 프로그래밍2 - 데이터 묶음

앞서 작성한 코드에 클래스를 도입하자. `MusicPlayerData`라는 클래스를 만들고, 음악 플레이어에 사용되는 데이터들을 여기에 묶어서 멤버 변수로 사용하자.

절차 지향 음악 플레이어2 - 데이터 묶음

```
package oop1;

public class MusicPlayerData {
    int volume = 0;
    boolean isOn = false;
}
```

음악 플레이어에 사용되는 `volume`, `isOn` 속성을 `MusicPlayerData`의 멤버 변수에 포함했다.

```
package oop1;

/**
 * 음악 플레이어와 관련된 데이터 묶기
 */
public class MusicPlayerMain2 {

    public static void main(String[] args) {

        MusicPlayerData data = new MusicPlayerData();

        //음악 플레이어 켜기
        data.isOn = true;
        System.out.println("음악 플레이어를 시작합니다");

        //볼륨 증가
        data.volume++;
        System.out.println("음악 플레이어 볼륨:" + data.volume);

        //볼륨 증가
        data.volume++;
        System.out.println("음악 플레이어 볼륨:" + data.volume);

        //볼륨 감소
        data.volume--;
        System.out.println("음악 플레이어 볼륨:" + data.volume);

        //음악 플레이어 상태
        System.out.println("음악 플레이어 상태 확인");
        if (data.isOn) {
            System.out.println("음악 플레이어 ON, 볼륨:" + data.volume);
        } else {
    }
```

```

        System.out.println("음악 플레이어 OFF");
    }

    //음악 플레이어 끄기
    data.isOn = false;
    System.out.println("음악 플레이어를 종료합니다");
}

}

```

음악 플레이어와 관련된 데이터는 `MusicPlayerData` 클래스에 존재한다. 이제 이 클래스를 사용하도록 기존 로직을 변경했다. 이후에 프로그램 로직이 더 복잡해져서 다양한 변수들이 추가되더라도 음악 플레이어와 관련된 변수들은 `MusicPlayerData data` 객체에 속해있으므로 쉽게 구분할 수 있다.

절차 지향 프로그래밍3 - 메서드 추출

코드를 보면 다음과 같이 중복되는 부분들이 있다.

```

//볼륨 증가
data.volume++;
System.out.println("음악 플레이어 볼륨:" + data.volume);

//볼륨 증가
data.volume++;
System.out.println("음악 플레이어 볼륨:" + data.volume);

```

그리고 각각의 기능들은 이후에 재사용 될 가능성이 높다.

- 음악 플레이어 켜기, 끄기
- 볼륨 증가, 감소
- 음악 플레이어 상태 출력

메서드를 사용해서 각각의 기능을 구분해보자.

절차 지향 음악 플레이어3 - 메서드 추출

```

package oop1;

/**
 * 메서드 추출

```

```
*/  
public class MusicPlayerMain3 {  
  
    public static void main(String[] args) {  
        MusicPlayerData data = new MusicPlayerData();  
        //음악 플레이어 켜기  
        on(data);  
        //볼륨 증가  
        volumeUp(data);  
        //볼륨 증가  
        volumeUp(data);  
        //볼륨 감소  
        volumeDown(data);  
        //음악 플레이어 상태  
        showStatus(data);  
        //음악 플레이어 끄기  
        off(data);  
    }  
  
    static void on(MusicPlayerData data) {  
        data.isOn = true;  
        System.out.println("음악 플레이어를 시작합니다");  
    }  
  
    static void off(MusicPlayerData data) {  
        data.isOn = false;  
        System.out.println("음악 플레이어를 종료합니다");  
    }  
  
    static void volumeUp(MusicPlayerData data) {  
        data.volume++;  
        System.out.println("음악 플레이어 볼륨:" + data.volume);  
    }  
  
    static void volumeDown(MusicPlayerData data) {  
        data.volume--;  
        System.out.println("음악 플레이어 볼륨:" + data.volume);  
    }  
  
    static void showStatus(MusicPlayerData data) {  
        System.out.println("음악 플레이어 상태 확인");  
        if (data.isOn) {  
            System.out.println("음악 플레이어 ON, 볼륨:" + data.volume);  
        }  
    }  
}
```

```
        } else {
            System.out.println("음악 플레이어 OFF");
        }
    }

}
```

각각의 기능을 메서드로 만든 덕분에 각각의 기능이 모듈화 되었다. 덕분에 다음과 같은 장점이 생겼다.

- **중복 제거:** 로직 중복이 제거되었다. 같은 로직이 필요하면 해당 메서드를 여러번 호출하면 된다.
- **변경 영향 범위:** 기능을 수정할 때 해당 메서드 내부만 변경하면 된다.
- **메서드 이름 추가:** 메서드 이름을 통해 코드를 더 쉽게 이해할 수 있다.

모듈화: 쉽게 이야기해서 레고 블럭을 생각하면 된다. 필요한 블럭을 가져다 꽉아서 사용할 수 있다. 여기서는 음악 플레이어의 기능이 필요하면 해당 기능을 메서드 호출 만으로 손쉽게 사용할 수 있다. 이제 음악 플레이어와 관련된 메서드를 조립해서 프로그램을 작성할 수 있다.

절차 지향 프로그래밍의 한계

지금까지 클래스를 사용해서 관련된 데이터를 하나로 묶고, 또 메서드를 사용해서 각각의 기능을 모듈화했다. 덕분에 상당히 깔끔하고 읽기 좋고, 유지보수하기 좋은 코드를 작성할 수 있었다. 하지만 여기서 더 개선할 수는 없을까?

우리가 작성한 코드의 한계는 바로 데이터와 기능이 분리되어 있다는 점이다. 음악 플레이어의 데이터는 `MusicPlayerData`에 있는데, 그 데이터를 사용하는 기능은 `MusicPlayerMain3`에 있는 각각의 메서드에 분리되어 있다. 그래서 음악 플레이어와 관련된 데이터는 `MusicPlayerData`를 사용해야 하고, 음악 플레이어와 관련된 기능은 `MusicPlayerMain3`의 각 메서드를 사용해야 한다.

데이터와 그 데이터를 사용하는 기능은 매우 밀접하게 연관되어 있다. 각각의 메서드를 보면 대부분 `MusicPlayerData`의 데이터를 사용한다. 따라서 이후에 관련 데이터가 변경되면 `MusicPlayerMain3` 부분의 메서드들도 함께 변경해야 한다. 그리고 이렇게 데이터와 기능이 분리되어 있으면 유지보수 관점에서도 관리 포인트가 2곳으로 늘어난다.

객체 지향 프로그래밍이 나오기 전까지는 지금과 같이 데이터와 기능이 분리되어 있었다. 따라서 지금과 같은 코드가 최선이었다. 하지만 객체 지향 프로그래밍이 나오면서 데이터와 기능을 온전히 하나로 묶어서 사용할 수 있게 되었다.

데이터와 기능을 하나로 온전히 묶는다는 것이 어떤 의미인지 이해하기 위해 간단한 예제를 만들어보자.

클래스와 메서드

클래스는 데이터인 멤버 변수 뿐 아니라 기능 역할을 하는 메서드도 포함할 수 있다.

먼저 멤버 변수만 존재하는 클래스로 간단한 코드를 작성해보자.

```
package oop1;
```

```
public class ValueData {  
    int value;  
}
```

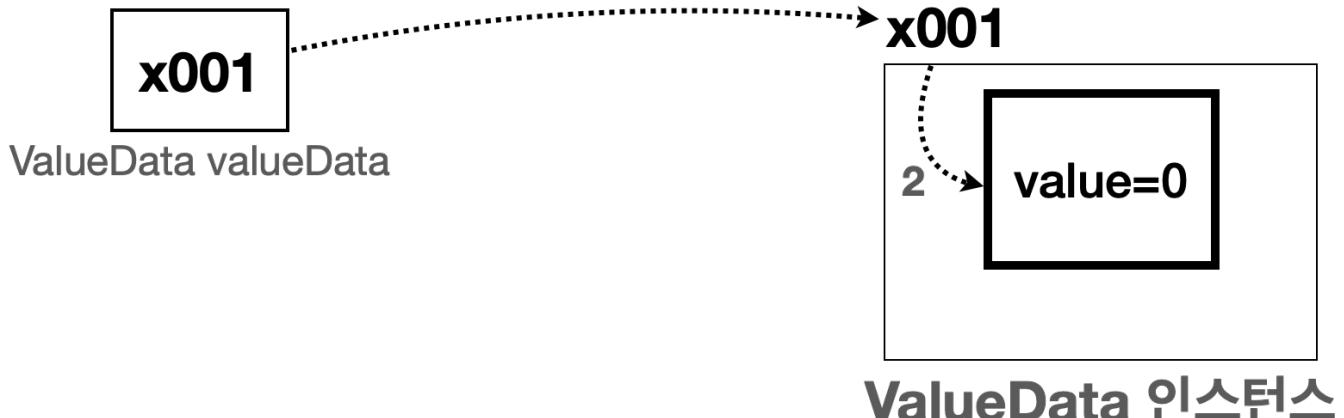
```
package oop1;
```

```
public class ValueDataMain {  
  
    public static void main(String[] args) {  
        ValueData valueData = new ValueData();  
        add(valueData);  
        add(valueData);  
        add(valueData);  
        System.out.println("최종 숫자=" + valueData.value);  
    }  
  
    static void add(ValueData valueData) {  
        valueData.value++;  
        System.out.println("숫자 증가 value=" + valueData.value);  
    }  
}
```

실행 결과

```
숫자 증가 value=1  
숫자 증가 value=2  
숫자 증가 value=3  
최종 숫자=3
```

1: valueData.value



ValueData 인스턴스

`ValueData`라는 인스턴스를 생성하고 외부에서 `ValueData.value`에 접근해 숫자를 하나씩 증가시키는 단순한 코드이다. 코드를 보면 데이터인 `value`와 `value`의 값을 증가시키는 기능인 `add()` 메서드가 서로 분리되어 있다.

자바 같은 객체 지향 언어는 클래스 내부에 속성(데이터)과 기능(메서드)을 함께 포함할 수 있다. 클래스 내부에 멤버 변수 뿐만 아니라 메서드도 함께 포함할 수 있다는 뜻이다.

이번에는 숫자를 증가시키는 기능도 클래스에 함께 포함해서 새로운 클래스를 정의해보자.

```
package oop1;

public class ValueObject {
    int value;

    void add() {
        value++;
        System.out.println("숫자 증가 value=" + value);
    }
}
```

이 클래스에는 데이터인 `value`와 해당 데이터를 사용하는 기능인 `add()` 메서드를 함께 정의했다.

이제 이 클래스가 어떻게 사용되는지 확인해보자.

참고: 여기서 만드는 `add()` 메서드에는 `static` 키워드를 사용하지 않는다.

메서드는 객체를 생성해야 호출할 수 있다. 그런데 `static`이 붙으면 객체를 생성하지 않고도 메서드를 호출할 수 있다.

`static`에 대한 자세한 내용은 뒤에서 설명한다.

```

package oop1;

public class ValueObjectMain {

    public static void main(String[] args) {
        ValueObject valueObject = new ValueObject();
        valueObject.add();
        valueObject.add();
        valueObject.add();
        System.out.println("최종 숫자= " + valueObject.value);
    }
}

```

실행 결과

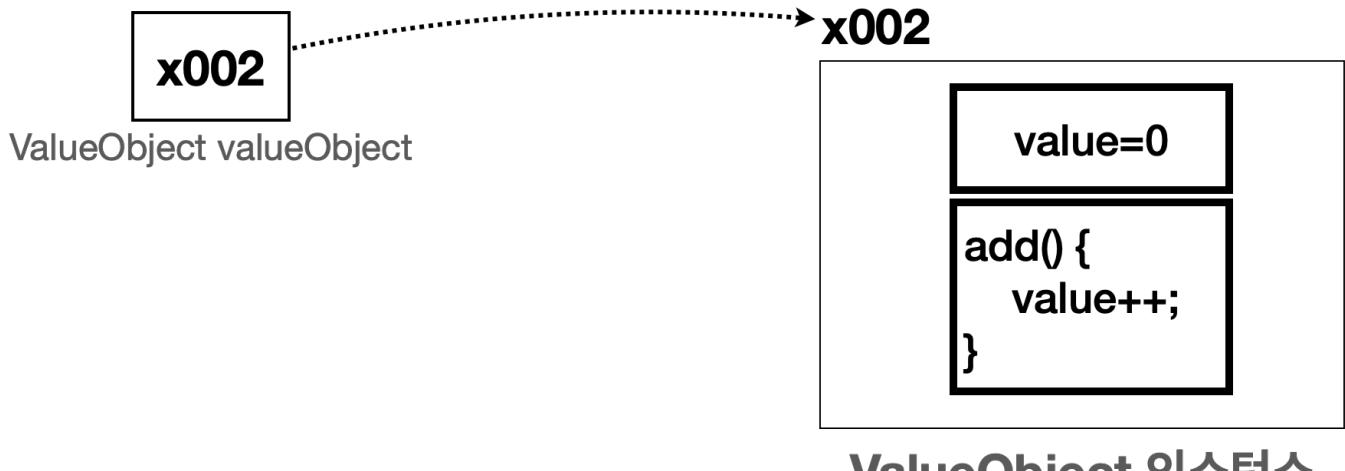
```

숫자 증가 value=1
숫자 증가 value=2
숫자 증가 value=3
최종 숫자=3

```

인스턴스 생성

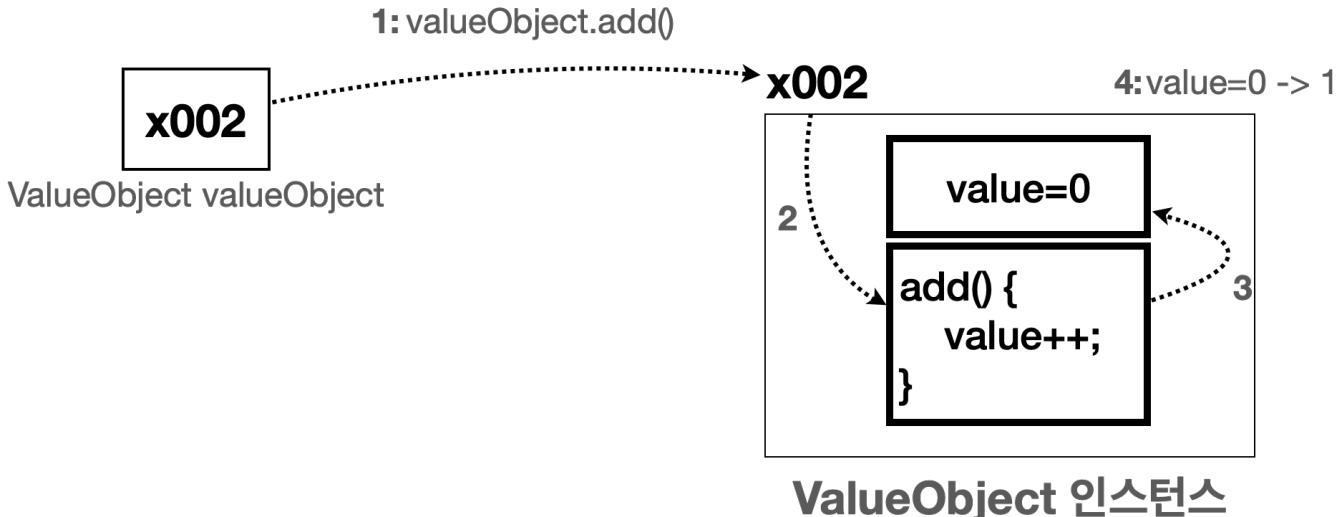
```
ValueObject valueObject = new ValueObject();
```



ValueObject 인스턴스

`valueObject`라는 객체를 생성했다. 이 객체는 멤버 변수 뿐만 아니라 내부에 기능을 수행하는 `add()` 메서드도 함께 존재한다.

인스턴스의 메서드 호출



인스턴스의 메서드를 호출하는 방법은 멤버 변수를 사용하는 방법과 동일하다. `.` (dot)을 찍어서 객체 접근한 다음에 원하는 메서드를 호출하면 된다.

```
valueObject.add(); //1
x002.add(); //2: x002 ValueObject 인스턴스에 있는 add() 메서드를 호출한다.
```

3: `add()` 메서드를 호출하면 메서드 내부에서 `value++`을 호출하게 된다. 이때 `value`에 접근해야 하는데, 기본으로 본인 인스턴스에 있는 멤버 변수에 접근한다. 본인 인스턴스가 `x002` 참조값을 사용하므로 자기 자신인 `x002.value`에 접근하게 된다.

4: `++` 연산으로 `value`의 값을 하나 증가시킨다.

정리

- 클래스는 속성(데이터, 멤버 변수)과 기능(메서드)을 정의할 수 있다.
- 객체는 자신의 메서드를 통해 자신의 멤버 변수에 접근할 수 있다.
 - 객체의 메서드 내부에서 접근하는 멤버 변수는 객체 자신의 멤버 변수이다.

객체 지향 프로그래밍

지금까지 개발한 음악 플레이어는 데이터와 기능이 분리되어 있었다. 이제 데이터와 기능을 하나로 묶어서 음악 플레이어라는 개념을 온전히 하나의 클래스에 담아보자. 프로그램을 작성하는 절차도 중요하지만 지금은 음악 플레이어라는 개념을 객체로 온전히 만드는 것이 더 중요하다. 음악 플레이어라는 객체를 지향해보자!

그러기 위해서는 프로그램의 실행 순서 보다는 음악 플레이어 클래스를 만드는 것 자체에 집중해야 한다. 음악 플레이어가 어떤 속성(데이터)을 가지고 어떤 기능(메서드)을 제공하는지 이 부분에 초점을 맞추어야 한다.

지금부터 우리는 음악 플레이어를 개발하는 개발자가 될 것이다. 이것을 어떻게 사용할지는 분리해서 생각하자. 쉽게 이야기해서 음악 플레이어를 만들어서 제공하는 개발자와 음악 플레이어를 사용하는 개발자가 분리되어 있다고 생각하면

된다.

음악 플레이어

- 속성: volume, isOn
- 기능: on(), off(), volumeUp(), volumeDown(), showStatus()

이것을 가지고 음악 플레이어를 만들어보자.

객체 지향 음악 플레이어

```
package oop;

public class MusicPlayer {

    int volume = 0;
    boolean isOn = false;

    void on() {
        isOn = true;
        System.out.println("음악 플레이어를 시작합니다");
    }

    void off() {
        isOn = false;
        System.out.println("음악 플레이어를 종료합니다");
    }

    void volumeUp() {
        volume++;
        System.out.println("음악 플레이어 볼륨:" + volume);
    }

    void volumeDown() {
        volume--;
        System.out.println("음악 플레이어 볼륨:" + volume);
    }

    void showStatus() {
        System.out.println("음악 플레이어 상태 확인");
        if (isOn) {
            System.out.println("음악 플레이어 ON, 볼륨:" + volume);
        } else {
```

```

        System.out.println("음악 플레이어 OFF");
    }
}

}

```

`MusicPlayer` 클래스에 음악 플레이어에 필요한 속성과 기능을 모두 정의했다. 이제 음악 플레이어가 필요한 곳에서 이 클래스만 있으면 온전한 음악 플레이어를 생성해서 사용할 수 있다. 음악 플레이어를 사용하는데 필요한 모든 속성과 기능이 하나의 클래스에 포함되어 있다!

```

package oop1;

/**
 * 객체 지향
 */
public class MusicPlayerMain4 {

    public static void main(String[] args) {
        MusicPlayer player = new MusicPlayer();
        //음악 플레이어 켜기
        player.on();
        //볼륨 증가
        player.volumeUp();
        //볼륨 증가
        player.volumeUp();
        //볼륨 감소
        player.volumeDown();
        //음악 플레이어 상태
        player.showStatus();
        //음악 플레이어 끄기
        player.off();
    }
}

```

실행 결과

```

음악 플레이어를 시작합니다
음악 플레이어 볼륨:1
음악 플레이어 볼륨:2
음악 플레이어 볼륨:1
음악 플레이어 상태 확인

```

음악 플레이어 ON, 볼륨: 1

음악 플레이어를 종료합니다

MediaPlayer를 사용하는 코드를 보자. MediaPlayer 객체를 생성하고 필요한 기능(메서드)을 호출하기만 하면 된다. 필요한 모든 것은 MediaPlayer 안에 들어있다!

- MediaPlayer를 사용하는 입장에서는 MediaPlayer의 데이터인 volume, isOn 같은 데이터는 전혀 사용하지 않는다.
- MediaPlayer를 사용하는 입장에서는 이제 MediaPlayer 내부에 어떤 속성(데이터)이 있는지 전혀 몰라도 된다. MediaPlayer를 사용하는 입장에서는 단순하게 MediaPlayer가 제공하는 기능 중에 필요한 기능을 호출해서 사용하기만 하면 된다.

캡슐화

MediaPlayer를 보면 음악 플레이어를 구성하기 위한 속성과 기능이 마치 하나의 캡슐에 쌓여있는 것 같다. 이렇게 속성과 기능을 하나로 묶어서 필요한 기능을 메서드를 통해 외부에 제공하는 것을 캡슐화라 한다.

객체 지향 프로그래밍 덕분에 음악 플레이어 객체를 사용하는 입장에서 진짜 음악 플레이어를 만들고 사용하는 것처럼 친숙하게 느껴진다. 그래서 코드가 더 읽기 쉬운 것은 물론이고, 속성과 기능이 한 곳에 있기 때문에 변경도 더 쉬워진다. 예를 들어서 MediaPlayer 내부 코드가 변하는 경우에 다른 코드는 변경하지 않아도 된다. MediaPlayer의 volume이라는 필드 이름이 다른 이름으로 변한다고 할 때 MediaPlayer 내부만 변경하면 된다. 또 음악 플레이어가 내부에서 출력하는 메시지를 변경할 때도 MediaPlayer 내부만 변경하면 된다. 이 경우 MediaPlayer를 사용하는 개발자는 코드를 전혀 변경하지 않아도 된다. 물론 외부에서 호출하는 MediaPlayer의 메서드 이름을 변경한다면 MediaPlayer를 사용하는 곳의 코드도 변경해야 한다.

문제와 풀이

문제1 - 절차 지향 직사각형 프로그램을 객체 지향으로 변경하기

다음은 직사각형의 넓이(Area), 둘레 길이(Perimeter), 정사각형 여부(square)를 구하는 프로그램이다.

- 절차 지향 프로그래밍 방식으로 되어 있는 코드를 객체 지향 프로그래밍 방식으로 변경해라.
- Rectangle 클래스를 만들어라.
- RectangleOopMain에 해당 클래스를 사용하는 main() 코드를 만들어라.

절차 지향 코드

```

package oop.ex;

public class RectangleProceduralMain {
    public static void main(String[] args) {
        int width = 5;
        int height = 8;
        int area = calculateArea(width, height);
        System.out.println("넓이: " + area);

        int perimeter = calculatePerimeter(width, height);
        System.out.println("둘레 길이: " + perimeter);

        boolean square = isSquare(width, height);
        System.out.println("정사각형 여부: " + square);
    }

    static int calculateArea(int width, int height) {
        return width * height;
    }

    static int calculatePerimeter(int width, int height) {
        return 2 * (width + height);
    }

    static boolean isSquare(int width, int height) {
        return width == height;
    }
}

```

실행 결과

```

넓이: 40
둘레 길이: 26
정사각형 여부: false

```

정답

```

package oop.ex;

public class Rectangle {
    int width;
    int height;
}

```

```

int calculateArea() {
    return width * height;
}

int calculatePerimeter() {
    return 2 * (width + height);
}

boolean isSquare() {
    return this.width == this.height;
}

}

```

```

package oop.ex;

public class RectangleOopMain {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle();
        rectangle.width = 5;
        rectangle.height = 8;

        int area = rectangle.calculateArea();
        System.out.println("넓이: " + area);

        int perimeter = rectangle.calculatePerimeter();
        System.out.println("둘레 길이: " + perimeter);

        boolean square = rectangle.isSquare();
        System.out.println("정사각형 여부: " + square);
    }
}

```

문제2 - 객체 지향 계좌

은행 계좌를 객체로 설계해야 한다.

- Account 클래스를 만들어라.
 - int balance 잔액
 - deposit(int amount): 입금 메서드
 - ◆ 입금시 잔액이 증가한다.

- withdraw(int amount) : 출금 메서드
 - ◆ 출금 시 잔액이 감소한다.
 - ◆ 만약 잔액이 부족하면 **잔액 부족**을 출력해야 한다.
- AccountMain 클래스를 만들고 main() 메서드를 통해 프로그램을 시작해라.
 - 계좌에 10000원을 입금해라.
 - 계좌에서 9000원을 출금해라.
 - 계좌에서 2000원을 출금 시도해라. → **잔액 부족** 출력을 확인해라.
 - 잔고를 출력해라. 잔고 : 1000

실행 결과

```
잔액 부족
잔고 : 1000
```

정답

```
package oop.ex;

class Account {
    int balance; // 잔액

    void deposit(int amount) {
        balance += amount;
    }

    void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
        } else {
            System.out.println("잔액 부족");
        }
    }
}
```

```
package oop.ex;

public class AccountMain {

    public static void main(String[] args) {
        Account account = new Account();
        account.deposit(10000);
        account.withdraw(9000);
    }
}
```

```
        account.withdraw(2000);
        System.out.println("잔고: " + account.balance);
    }
}
```

정리

객체 지향 프로그래밍 vs 절차 지향 프로그래밍

객체 지향 프로그래밍과 절차 지향 프로그래밍은 서로 대치되는 개념이 아니다. 객체 지향이라도 프로그램의 작동 순서는 중요하다. 다만 어디에 더 초점을 맞추는가에 둘의 차이가 있다. 객체 지향의 경우 객체의 설계와 관계를 중시한다. 반면 절차 지향의 경우 데이터와 기능이 분리되어 있고, 프로그램이 어떻게 작동하는지 그 순서에 초점을 맞춘다.

절차 지향 프로그래밍

- 절차 지향 프로그래밍은 이름 그대로 절차를 지향한다. 쉽게 이야기해서 실행 순서를 중요하게 생각하는 방식이다.
- 절차 지향 프로그래밍은 프로그램의 흐름을 순차적으로 따르며 처리하는 방식이다. 즉, "어떻게"를 중심으로 프로그래밍 한다.

객체 지향 프로그래밍

- 객체 지향 프로그래밍은 이름 그대로 객체를 지향한다. 쉽게 이야기해서 객체를 중요하게 생각하는 방식이다.
- 객체 지향 프로그래밍은 실제 세계의 사물이나 사건을 객체로 보고, 이러한 객체들 간의 상호작용을 중심으로 프로그래밍하는 방식이다. 즉, "무엇을" 중심으로 프로그래밍 한다.

둘의 중요한 차이

- 절차 지향은 데이터와 해당 데이터에 대한 처리 방식이 분리되어 있다. 반면 객체 지향에서는 데이터와 그 데이터에 대한 행동(메서드)이 하나의 '객체' 안에 함께 포함되어 있다.

객체란?

세상의 모든 사물을 단순하게 추상화해보면 속성(데이터)과 기능 딱 2가지로 설명할 수 있다.

자동차

- 속성: 차량 색상, 현재 속도
- 기능: 엑셀, 브레이크, 문 열기, 문 닫기

동물

- 속성: 색상, 키, 온도

- 기능: 먹는다. 걷는다.

게임 캐릭터

- 속성: 레벨, 경험치, 소유한 아이템들
- 기능: 이동, 공격, 아이템 획득

객체 지향 프로그래밍은 모든 사물을 속성과 기능을 가진 객체로 생각하는 것이다. 객체에는 속성과 기능만 존재한다.

이렇게 단순화하면 세상에 있는 객체들을 컴퓨터 프로그램으로 쉽게 설계할 수 있다.

이런 장점들 덕분에 지금은 객체 지향 프로그래밍이 가장 많이 사용된다.

참고로 실세계와 객체가 항상 1:1로 매칭되는 것은 아니다.

객체 지향의 특징은 속성과 기능을 하나로 묶는 것 뿐만 아니라 캡슐화, 상속, 다형성, 추상화, 메시지 전달 같은 다양한 특징들이 있다. 앞으로 이런 특징들을 하나씩 알아가보자.

4. 생성자

#1.인강/0.자바/2.자바-기본

- /생성자 - 필요한 이유
- /this
- /생성자 - 도입
- /기본 생성자
- /생성자 - 오버로딩과 this()
- /문제와 풀이
- /정리

생성자 - 필요한 이유

객체를 생성하는 시점에 어떤 작업을 하고 싶다면 생성자(Constructor)를 이용하면 된다.
생성자를 알아보기 전에 먼저 생성자가 왜 필요한지 코드로 간단히 알아보자.

MemberInit

```
package construct;

public class MemberInit {
    String name;
    int age;
    int grade;
}
```

MethodInitMain1

```
package construct;

public class MethodInitMain1 {
    public static void main(String[] args) {
        MemberInit member1 = new MemberInit();
        member1.name = "user1";
        member1.age = 15;
        member1.grade = 90;

        MemberInit member2 = new MemberInit();
        member2.name = "user2";
```

```

        member2.age = 16;
        member2.grade = 80;

        MemberInit[] members = {member1, member2};

        for (MemberInit s : members) {
            System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" +
s.grade);
        }
    }
}

```

실행 결과

```

이름:user1 나이:15 성적:90
이름:user2 나이:16 성적:80

```

회원 객체를 생성하고 나면 `name`, `age`, `grade` 같은 변수에 초기값을 설정한다. 아마도 회원 객체를 제대로 사용하기 위해서는 객체를 생성하자 마자 이런 초기값을 설정해야 할 것이다. 이 코드에는 회원의 초기값을 설정하는 부분이 계속 반복된다. 메서드를 사용해서 반복을 제거해보자.

```

package construct;

public class MethodInitMain2 {
    public static void main(String[] args) {
        MemberInit member1 = new MemberInit();
        initMember(member1, "user1", 15, 90);

        MemberInit member2 = new MemberInit();
        initMember(member2, "user2", 16, 80);

        MemberInit[] members = {member1, member2};

        for (MemberInit s : members) {
            System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" +
s.grade);
        }
    }

    static void initMember(MemberInit member, String name, int age, int grade) {
        member.name = name;
    }
}

```

```
        member.age = age;
        member.grade = grade;
    }
}
```

`initMember(...)` 메서드를 사용해서 반복을 제거했다. 그런데 이 메서드는 대부분 `MemberInit` 객체의 멤버 변수를 사용한다. 우리는 앞서 객체 지향에 대해서 학습했다. 이런 경우 속성과 기능을 한 곳에 두는 것이 더 나은 방법이다. 쉽게 이야기해서 `MemberInit`이 자기 자신의 데이터를 변경하는 기능(메서드)을 제공하는 것이 좋다.

this

`MemberInit - initMember()` 추가

```
package construct;

public class MemberInit {
    String name;
    int age;
    int grade;

    //추가
    void initMember(String name, int age, int grade) {
        this.name = name;
        this.age = age;
        this.grade = grade;
    }
}
```

```
package construct;

public class MethodInitMain3 {
    public static void main(String[] args) {
        MemberInit member1 = new MemberInit();
        member1.initMember("user1", 15, 90);

        MemberInit member2 = new MemberInit();
        member2.initMember("user2", 16, 80);
```

```
MemberInit[] members = {member1, member2};

for (MemberInit s : members) {
    System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" +
s.grade);
}
}
```

`initMember(...)` 는 `Member`에 초기값 설정 기능을 제공하는 메서드이다.

다음과 같이 메서드를 호출하면 객체의 멤버 변수에 인자로 넘어온 값을 채운다.

```
member1.initMember("user1", 15, 90)
```

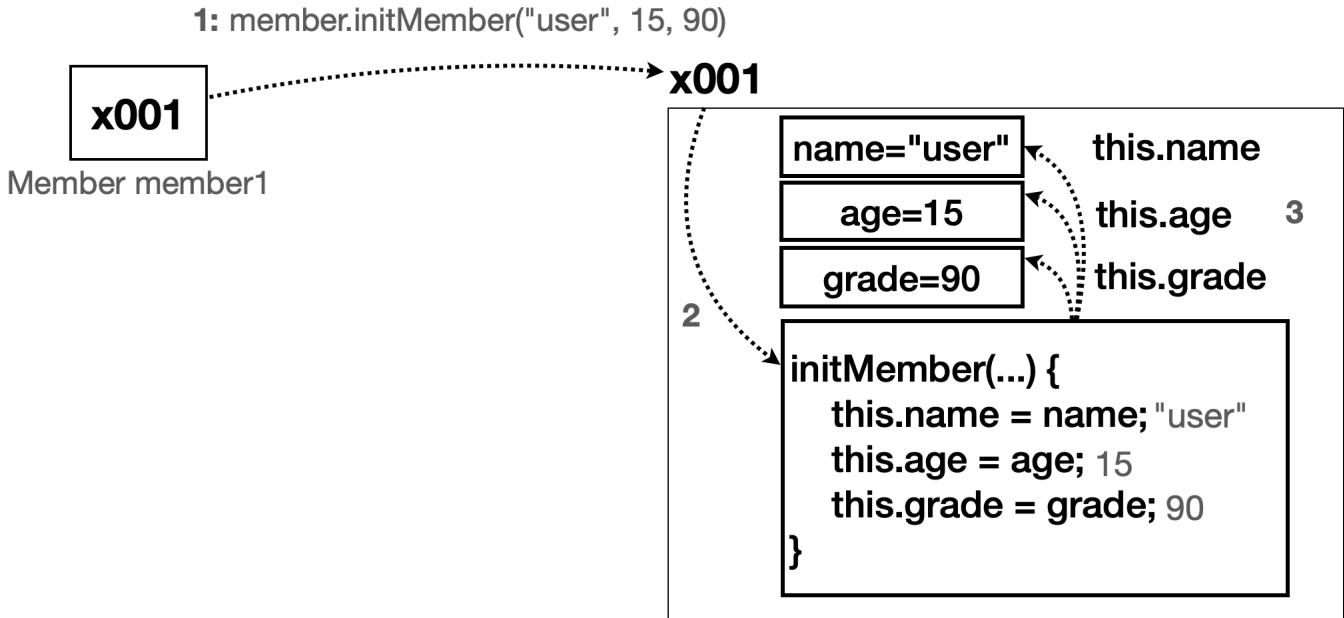
this

`Member`의 코드를 다시 보자

`initMember(String name...)`의 코드를 보면 메서드의 매개변수에 정의한 `String name`과 `Member`의 멤버 변수의 이름이 `String name`으로 둘다 똑같다. 나머지 변수 이름도 `name`, `age`, `grade`로 모두 같다.

멤버 변수와 메서드의 매개변수의 이름이 같으면 둘을 어떻게 구분해야 할까?

- 이 경우 멤버 변수보다 매개변수가 코드 블럭의 더 안쪽에 있기 때문에 **매개변수가 우선순위**를 가진다. 따라서 `initMember(String name, ...)` 메서드 안에서 `name`이라고 적으면 매개변수에 접근하게 된다.
- 멤버 변수에 접근하려면 앞에 `this.`이라고 해주면 된다. 여기서 `this`는 인스턴스 자신의 참조값을 가리킨다.



Member 인스턴스

진행 과정

```
this.name = name; //1. 오른쪽의 name은 매개변수에 접근
this.name = "user"; //2. name 매개변수의 값 사용
x001.name = "user"; //3. this.은 인스턴스 자신의 참조값을 뜻함, 따라서 인스턴스의 멤버 변수에 접근
```

this 제거

만약 이 예제에서 `this`를 제거하면 어떻게 될까?

```
this.name = name
```

다음과 같이 수정하면 `name`은 둘다 매개변수를 뜻하게 된다. 따라서 멤버변수의 값이 변경되지 않는다.

```
name = name
```

정리

- 매개변수의 이름과 멤버 변수의 이름이 같은 경우 `this`를 사용해서 둘을 명확하게 구분해야 한다.
- `this`는 인스턴스 자신을 가리킨다.

this의 생략

`this`는 생략할 수 있다. 이 경우 변수를 찾을 때 가까운 지역변수(매개변수도 지역변수다)를 먼저 찾고 없으면 그 다음으로 멤버 변수를 찾는다. 멤버 변수도 없으면 오류가 발생한다.

다음 예제는 필드 이름과 매개변수의 이름이 서로 다르다.

```
package construct;

public class MemberThis {
    String nameField;

    void initMember(String nameParameter) {
        nameField = nameParameter;
    }
}
```

예를 들어서 `nameField`는 앞에 `this`가 없어도 멤버 변수에 접근한다.

- `nameField`은 먼저 지역변수(매개변수)에서 같은 이름이 있는지 찾는다. 이 경우 없으므로 멤버 변수에서 찾는다.
- `nameParameter`은 먼저 지역변수(매개변수)에서 같은 이름이 있는지 찾는다. 이 경우 매개변수가 있으므로 매개변수를 사용한다.

this와 코딩 스타일

다음과 같이 멤버 변수에 접근하는 경우에 항상 `this`를 사용하는 코딩 스타일도 있다.

```
package construct;

public class MemberThis {
    String nameField;

    void initMember(String nameParameter) {
        this.nameField = nameParameter;
    }
}
```

`this.nameField`를 보면 `this`를 생략할 수 있지만, 생략하지 않고 사용해도 된다.

이렇게 `this`를 사용하면 이 코드가 멤버 변수를 사용한다는 것을 눈으로 쉽게 확인할 수 있다. 그래서 과거에 이런 스타일을 많이 사용하기도 했다. 쉽게 이야기해서 `this`를 강제로 사용해서, 지역 변수(매개변수)와 멤버 변수를 눈에 보이도록 구분하는 것이다.

하지만 최근에 IDE가 발전하면서 IDE가 멤버 변수와 지역 변수를 색상으로 구분해준다.

다음을 보면 멤버 변수와 지역 변수의 색상이 다른 것을 확인할 수 있다.

```
no usages
public class MemberThis {
    1 usage
    public String nameField;

    no usages
    public void initMember(String nameParameter) {
        nameField = nameParameter;
    }
}
```

이런 점 때문에 `this`는 앞서 설명한 것처럼 이름이 중복되는 것처럼, 꼭 필요한 경우에만 사용해도 충분하다 생각한다.

생성자 - 도입

프로그래밍을 하다보면 객체를 생성하고 이후에 바로 초기값을 할당해야 하는 경우가 많다. 따라서 앞서 `initMember(...)`와 같은 메서드를 매번 만들어야 한다.

그래서 대부분의 객체 지향 언어는 객체를 생성하자마자 즉시 필요한 기능을 좀 더 편리하게 수행할 수 있도록 생성자라는 기능을 제공한다. 생성자를 사용하면 객체를 생성하는 시점에 즉시 필요한 기능을 수행할 수 있다.

생성자는 앞서 살펴본 `initMember(...)`와 같이 메서드와 유사하지만 몇 가지 다른 특징이 있다.

기존 코드를 유지하기 위해 `MemberConstruct`라는 새로운 클래스를 작성하겠다.

MemberConstruct

```
package construct;

public class MemberConstruct {
    String name;
    int age;
    int grade;
```

```

MemberConstruct(String name, int age, int grade) {
    System.out.println("생성자 호출 name=" + name + ",age=" + age + ",grade=" +
grade);
    this.name = name;
    this.age = age;
    this.grade = grade;
}

}

```

다음 부분이 바로 생성자이다.

```

MemberConstruct(String name, int age, int grade) {
    System.out.println("생성자 호출 name=" + name + ",age=" + age + ",grade=" +
grade);
    this.name = name;
    this.age = age;
    this.grade = grade;
}

```

생성자는 메서드와 비슷하지만 다음과 같은 차이가 있다.

- 생성자의 이름은 클래스 이름과 같아야 한다. 따라서 첫 글자도 대문자로 시작한다.
- 생성자는 반환 타입이 없다. 비워두어야 한다.
- 나머지는 메서드와 같다.

```

package construct;

public class ConstructMain1 {
    public static void main(String[] args) {
        MemberConstruct member1 = new MemberConstruct("user1", 15, 90);
        MemberConstruct member2 = new MemberConstruct("user2", 16, 80);

        MemberConstruct[] members = {member1, member2};

        for (MemberConstruct s : members) {
            System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" +
s.grade);
        }
    }
}

```

실행 결과

```
생성자 호출 name=user1, age=15, grade=90
생성자 호출 name=user2, age=16, grade=80
이름:user1 나이:15 성적:90
이름:user2 나이:16 성적:80
```

생성자 호출

생성자는 인스턴스를 생성하고 나서 즉시 호출된다. 생성자를 호출하는 방법은 다음 코드와 같이 `new` 명령어 다음에 생성자 이름과 매개변수에 맞추어 인수를 전달하면 된다.

```
new 생성자이름( 생성자에 맞는 인수 목록)
new 클래스이름( 생성자에 맞는 인수 목록)
```

참고로 생성자이름이 클래스 이름이기 때문에 둘다 맞는 표현이다.

```
new MemberConstruct("user1", 15, 90)
```

이렇게 하면 인스턴스를 생성하고 즉시 해당 생성자를 호출한다. 여기서는 `Member` 인스턴스를 생성하고 바로 `MemberConstruct(String name, int age, int grade)` 생성자를 호출한다.

참고로 `new` 키워드를 사용해서 객체를 생성할 때 마지막에 괄호`()`도 포함해야 하는 이유가 바로 생성자 때문이다. 객체를 생성하면서 동시에 생성자를 호출한다는 의미를 포함한다.

생성자 장점

중복 호출 제거

생성자가 없던 시절에는 생성 직후에 어떤 작업을 수행하기 위해 다음과 같이 메서드를 직접 한번 더 호출해야 했다. 생성자 덕분에 객체를 생성하면서 동시에 생성 직후에 필요한 작업을 한번에 처리할 수 있게 되었다.

```
//생성자 등장 전
MemberInit member = new MemberInit();
member.initMember("user1", 15, 90);

//생성자 등장 후
MemberConstruct member = new MemberConstruct("user1", 15, 90);
```

제약 - 생성자 호출 필수

방금 코드에서 생성자 등장 전 코드를 보자. 이 경우 `initMember(...)`를 실수로 호출하지 않으면 어떻게 될까? 이 메서드를 실수로 호출하지 않아도 프로그램은 작동한다. 하지만 회원의 이름과 나이, 성적 데이터가 없는 상태로 프로그램이 동작하게 된다. 만약에 이 값들을 필수로 반드시 입력해야 한다면, 시스템에 큰 문제가 발생할 수 있다. 결국 아무 정보가 없는 유령 회원이 시스템 내부에 등장하게 된다.

생성자의 진짜 장점은 객체를 생성할 때 직접 정의한 생성자가 있다면 **직접 정의한 생성자를 반드시 호출해야 한다는 점**이다. 참고로 생성자를 메서드 오버로딩처럼 여러개 정의할 수 있는데, 이 경우에는 하나만 호출하면 된다.

`MemberConstruct` 클래스의 경우 다음 생성자를 직접 정의했기 때문에 직접 정의한 생성자를 반드시 호출해야 한다.

```
MemberConstruct(String name, int age, int grade) {...}
```

다음과 같이 직접 정의한 생성자를 호출하지 않으면 컴파일 오류가 발생한다.

```
MemberConstruct member3 = new MemberConstruct(); //컴파일 오류 발생  
member3.name = "user1";
```

컴파일 오류 메시지

```
no suitable constructor found for MemberConstruct(no arguments)
```

컴파일 오류는 IDE에서 즉시 확인할 수 있는 좋은 오류이다. 이 경우 개발자는 객체를 생성할 때, 직접 정의한 생성자를 필수로 호출해야 한다는 것을 바로 알 수 있다. 그래서 필요한 생성자를 찾아서 다음과 같이 호출할 것이다.

```
MemberConstruct member = new MemberConstruct("user1", 15, 90);
```

생성자 덕분에 회원의 이름, 나이, 성적은 항상 필수로 입력하게 된다. 따라서 아무 정보가 없는 유령 회원이 시스템 내부에 등장할 가능성을 원천 차단한다!

생성자를 사용하면 필수값 입력을 보장할 수 있다

| **참고:** 좋은 프로그램은 무한한 자유도가 주어지는 프로그램이 아니라 적절한 제약이 있는 프로그램이다.

기본 생성자

생각해보면 생성자를 만들지 않았는데, 생성자를 호출한 적이 있다. 다음 코드들을 다시 확인해보자.

```
public class MemberInit {  
    String name;  
    int age;  
    int grade;  
}
```

```
public class MethodInitMain1 {  
    public static void main(String[] args) {  
        MemberInit member1 = new MemberInit();  
        ...  
    }  
}
```

여기서 `new MemberInit()` 이 부분은 분명히 매개변수가 없는 다음과 같은 생성자가 필요할 것이다.

```
public class MemberInit {  
    String name;  
    int age;  
    int grade;  
  
    MemberInit() { //생성자 필요  
    }  
}
```

기본 생성자

- 매개변수가 없는 생성자를 기본 생성자라 한다.
- 클래스에 생성자가 하나도 없으면 자바 컴파일러는 매개변수가 없고, 작동하는 코드가 없는 기본 생성자를 자동으로 만들어준다.
- 생성자가 하나라도 있으면 자바는 기본 생성자를 만들지 않는다.

`MemberInit` 클래스의 경우 생성자를 만들지 않았으므로 자바가 자동으로 기본 생성자를 만들어준 것이다.

예제를 통해서 기본 생성자를 확인해보자.

MemberDefault

```
package construct;  
  
public class MemberDefault {  
    String name;  
}
```

MemberDefaultMain

```
package construct;
```

```
public class MemberDefaultMain {  
  
    public static void main(String[] args) {  
        MemberDefault memberDefault = new MemberDefault();  
    }  
}
```

MemberDefault 클래스에는 생성자가 하나도 없으므로 자바는 자동으로 다음과 같은 기본 생성자를 만들어준다.
(우리 눈에 보이지는 않는다.)

MemberDefault - 기본 생성자

```
package construct;  
  
public class MemberDefault {  
    String name;  
  
    //기본 생성자  
    public MemberDefault() {  
    }  
}
```

참고: 자바가 자동으로 생성해주시는 기본 생성자는 클래스와 같은 접근 제어자를 가진다. public은 뒤에 접근 제어자에서 자세히 설명한다.

물론 다음과 같이 기본 생성자를 직접 정의해도 된다.

```
package construct;  
  
public class MemberDefault {  
    String name;  
  
    MemberDefault() {  
        System.out.println("생성자 호출");  
    }  
}
```

실행 결과

생성자 호출

기본 생성자를 왜 자동으로 만들어줄까?

만약 자바에서 기본 생성자를 만들어주지 않는다면 생성자 기능이 필요하지 않은 경우에도 모든 클래스에 개발자가 직접 기본 생성자를 정의해야 한다. 생성자 기능을 사용하지 않는 경우도 많기 때문에 이런 편의 기능을 제공한다.

정리

- 생성자는 반드시 호출되어야 한다.
- 생성자가 없으면 기본 생성자가 제공된다.
- 생성자가 하나라도 있으면 기본 생성자가 제공되지 않는다. 이 경우 개발자가 정의한 생성자를 직접 호출해야 한다.

생성자 - 오버로딩과 this()

생성자도 메서드 오버로딩처럼 매개변수만 다르게 해서 여러 생성자를 제공할 수 있다.

MemberConstruct - 생성자 추가

```
package construct;

public class MemberConstruct {
    String name;
    int age;
    int grade;

    // 추가
    MemberConstruct(String name, int age) {
        this.name = name;
        this.age = age;
        this.grade = 50;
    }

    MemberConstruct(String name, int age, int grade) {
        System.out.println("생성자 호출 name=" + name + ", age=" + age + ", grade=" +
grade);
        this.name = name;
        this.age = age;
        this.grade = grade;
    }
}
```

기존 MemberConstruct에 생성자를 하나 추가해서 생성자가 2개가 되었다.

```
MemberConstruct(String name, int age)
MemberConstruct(String name, int age, int grade)
```

새로 추가한 생성자는 grade를 받지 않는다. 대신에 grade는 50점이 된다.

```
package construct;

public class ConstructMain2 {
    public static void main(String[] args) {
        MemberConstruct member1 = new MemberConstruct("user1", 15, 90);
        MemberConstruct member2 = new MemberConstruct("user2", 16);

        MemberConstruct[] members = {member1, member2};

        for (MemberConstruct s : members) {
            System.out.println("이름:" + s.name + " 나이:" + s.age + " 성적:" +
s.grade);
        }
    }
}
```

실행 결과

```
생성자 호출 name=user1,age=15,grade=90
이름:user1 나이:15 성적:90
이름:user2 나이:16 성적:50
```

생성자를 오버로딩 한 덕분에 성적 입력이 꼭 필요한 경우에는 grade가 있는 생성자를 호출하면 되고, 그렇지 않은 경우에는 grade가 없는 생성자를 호출하면 된다. grade가 없는 생성자를 호출하면 성적은 50점이 된다.

this()

두 생성자를 비교해 보면 코드가 중복 되는 부분이 있다.

```
public MemberConstruct(String name, int age) {
    this.name = name;
    this.age = age;
    this.grade = 50;
}
```

```
public MemberConstruct(String name, int age, int grade) {  
    this.name = name;  
    this.age = age;  
    this.grade = grade;  
}
```

바로 다음 부분이다.

```
this.name = name;  
this.age = age;
```

이때 `this()`라는 기능을 사용하면 생성자 내부에서 자신의 생성자를 호출할 수 있다. 참고로 `this`는 인스턴스 자신의 참조값을 가리킨다. 그래서 자신의 생성자를 호출한다고 생각하면 된다.

코드를 다음과 같이 수정해보자.

MemberConstruct - this() 사용

```
package construct;  
  
public class MemberConstruct {  
    String name;  
    int age;  
    int grade;  
  
    MemberConstruct(String name, int age) {  
        this(name, age, 50); //변경  
    }  
  
    MemberConstruct(String name, int age, int grade) {  
        System.out.println("생성자 호출 name=" + name + ",age=" + age + ",grade=" +  
grade);  
        this.name = name;  
        this.age = age;  
        this.grade = grade;  
    }  
}
```

이 코드는 첫번째 생성자 내부에서 두번째 생성자를 호출한다.

```
MemberConstruct(String name, int age) -> MemberConstruct(String name, int age,  
int grade)
```

`this()` 를 사용하면 생성자 내부에서 다른 생성자를 호출할 수 있다. 이 부분을 잘 활용하면 지금과 같이 중복을 제거할 수 있다. 물론 실행 결과는 기존과 같다.

this() 규칙

- `this()` 는 생성자 코드의 첫줄에만 작성할 수 있다.

다음은 규칙 위반이다. 이 경우 컴파일 오류가 발생한다.

```
public MemberConstruct(String name, int age) {  
    System.out.println("go");  
    this(name, age, 50);  
}
```

`this()` 가 생성자 코드의 첫줄에 사용되지 않았다.

문제와 풀이

문제 - Book과 생성자

`BookMain` 코드가 작동하도록 `Book` 클래스를 완성하세요.
특히 `Book` 클래스의 생성자 코드에 중복이 없도록 주의하세요.

```
package construct.ex;  
  
public class Book {  
    String title; //제목  
    String author; //저자  
    int page; //페이지 수  
  
    //TODO 코드를 완성하세요.  
}
```

```
package construct.ex;  
  
public class BookMain {  
    public static void main(String[] args) {  
        // 기본 생성자 사용
```

```

Book book1 = new Book();
book1.displayInfo();

// title과 author만을 매개변수로 받는 생성자
Book book2 = new Book("Hello Java", "Seo");
book2.displayInfo();

// 모든 필드를 매개변수로 받는 생성자
Book book3 = new Book("JPA 프로그래밍", "kim", 700);
book3.displayInfo();
}
}

```

실행 결과

```

제목: , 저자: , 페이지: 0
제목: Hello Java, 저자: Seo, 페이지: 0
제목: JPA 프로그래밍, 저자: kim, 페이지: 700

```

정답

```

package construct.ex;

public class Book {
    String title;
    String author;
    int page;

    // 기본생성자
    Book() {
        this("", "", 0);
    }

    // title과 author만을 매개변수로 받는 생성자
    Book(String title, String author) {
        this(title, author, 0);
    }

    // 모든 필드를 매개변수로 받는 생성자
    Book(String title, String author, int page) {
        this.title = title;
        this.author = author;
        this.page = page;
    }
}

```

```
}

void displayInfo() {
    System.out.println("제목: " + title + ", 저자: " + author + ", 페이지: " +
page);
}

}
```

정리

생성자는 객체 생성 직후 객체를 초기화 하기 위한 특별한 메서드로 생각할 수 있다.

5. 패키지

#1.인강/0.자바/2.자바-기본

- /패키지 - 시작
- /패키지 - import
- /패키지 규칙
- /패키지 활용
- /정리

패키지 - 시작

여러분이 쇼핑몰 시스템을 개발한다고 가정해보자. 다음과 같이 프로그램이 매우 작고 단순해서 클래스가 몇개 없다면 크게 고민할 거리가 없겠지만, 기능이 점점 추가되어서 프로그램이 아주 커지게 된다면 어떻게 될까?

아주 작은 프로그램

```
Order  
User  
Product
```

큰 프로그램

```
User  
UserManager  
UserHistory  
Product  
ProductCatalog  
ProductImage  
Order  
OrderService  
OrderHistory  
ShoppingCart  
CartItem  
Payment  
PaymentHistory  
Shipment  
ShipmentTracker
```

매우 많은 클래스가 등장하면서 관련 있는 기능들을 분류해서 관리하고 싶을 것이다.

컴퓨터는 보통 파일을 분류하기 위해 폴더, 디렉토리라는 개념을 제공한다. 자바도 이런 개념을 제공하는데, 이것이 바로 패키지이다.

다음과 같이 카테고리를 만들고 분류해보자.

```
* user
  * User
  * UserManager
  * UserHistory
* product
  * Product
  * ProductCatalog
  * ProductImage
* order
  * Order
  * OrderService
  * OrderHistory
* cart
  * ShoppingCart
  * CartItem
* payment
  * Payment
  * PaymentHistory
* shipping
  * Shipment
  * ShipmentTracker
```

여기서 `user`, `product` 등이 바로 패키지이다. 그리고 해당 패키지 안에 관련된 자바 클래스들을 넣으면 된다.
패키지(package)는 이를 그대로 물건을 운송하기 위한 포장 용기나 그 포장 뚝음을 뜻한다.

패키지 사용

패키지 사용법을 코드로 확인해보자.

패키지를 먼저 만들고 그 다음에 클래스를 만들어야 한다.

패키지 위치에 주의하자.

pack.Data

```
package pack;

public class Data {
    public Data() {
```

```
        System.out.println("패키지 pack Data 생성");
    }
}
```

- 패키지를 사용하는 경우 항상 코드 첫줄에 package pack과 같이 패키지 이름을 적어주어야 한다.
- 여기서는 pack 패키지에 Data 클래스를 만들었다.
- 이후에 Data 인스턴스가 생성되면 생성자를 통해 정보를 출력한다.

pack.a.User

```
package pack.a;

public class User {

    public User() {
        System.out.println("패키지 pack.a 회원 생성");
    }
}
```

- pack 하위에 a라는 패키지를 먼저 만들자.
- pack.a 패키지에 User 클래스를 만들었다.
- 이후에 User 인스턴스가 생성되면 생성자를 통해 정보를 출력한다.

참고: 생성자에 public을 사용했다. 다른 패키지에서 이 클래스의 생성자를 호출하려면 public을 사용해야 한다. 자세한 내용은 접근 제어자에서 설명한다. 지금은 코드와 같이 생성자에 public 키워드를 넣어두자.

pack.PackageMain1

```
package pack;

public class PackageMain1 {

    public static void main(String[] args) {
        Data data = new Data();
        pack.a.User user = new pack.a.User();
    }
}
```

pack 패키지 위치에 PackageMain1 클래스를 만들었다.

실행 결과

```
패키지 pack Data 생성
패키지 pack.a 회원 생성
```

- **사용자와 같은 위치:** PackageMain1 과 Data 는 같은 pack 이라는 패키지에 소속되어 있다. 이렇게 같은 패키지에 있는 경우에는 패키지 경로를 생략해도 된다.
- **사용자와 다른 위치:** PackageMain1 과 User 는 서로 다른 패키지다. 이렇게 패키지가 다르면 pack.a.User 와 같이 패키지 전체 경로를 포함해서 클래스를 적어주어야 한다.

패키지 - import

import

이전에 본 코드와 같이 패키지가 다르다고 pack.a.User 와 같이 항상 전체 경로를 적어주는 것은 불편하다. 이때는 import 를 사용하면 된다.

```
package pack;

import pack.a.User;

public class PackageMain2 {

    public static void main(String[] args) {
        Data data = new Data();
        User user = new User(); //import 사용으로 패키지 명 생략 가능
    }
}
```

실행 결과

```
패키지 pack Data 생성
패키지 pack.a 회원 생성
```

코드에서 첫줄에는 package 를 사용하고, 다음 줄에는 import 를 사용할 수 있다.

import 를 사용하면 다른 패키지에 있는 클래스를 가져와서 사용할 수 있다.

import 를 사용한 덕분에 코드에서는 패키지 명을 생략하고 클래스 이름만 적을 수 있다.

참고로 특정 패키지에 포함된 모든 클래스를 포함해서 사용하고 싶으면 import 시점에 *(별) 을 사용하면 된다.

패키지 별(*) 사용

```
package pack;
```

```
import pack.a.*; //pack.a의 모든 클래스 사용

public class PackageMain2 {

    public static void main(String[] args) {
        Data data = new Data();
        User user = new User(); //import 사용으로 패키지 명 생략 가능
    }
}
```

이렇게 하면 pack.a 패키지에 있는 모든 클래스를 패키지 명을 생략하고 사용할 수 있다.

클래스 이름 중복

패키지 덕분에 클래스 이름이 같아도 패키지 이름으로 구분해서 같은 이름의 클래스를 사용할 수 있다.

```
pack.a.User
pack.b.User
```

이런 경우 클래스 이름이 둘다 User 이지만 패키지 이름으로 대상을 구분할 수 있다.

이렇게 이름이 같은 경우 둘다 사용하고 싶으면 어떻게 해야할까?

pack.b.User

```
package pack.b;

public class User {

    public User() {
        System.out.println("패키지 pack.b 회원 생성");
    }
}
```

```
package pack;

import pack.a.User;

public class PackageMain3 {

    public static void main(String[] args) {
        User userA = new User();
        pack.b.User userB = new pack.b.User();
    }
}
```

}

같은 이름의 클래스가 있다면 `import` 는 둘중 하나만 선택할 수 있다. 이때는 자주 사용하는 클래스를 `import` 하고 나머지를 패키지를 포함한 전체 경로를 적어주면 된다. 물론 둘다 전체 경로를 적어준다면 `import` 를 사용하지 않아도 된다.

패키지 규칙

패키지 규칙

- 패키지의 이름과 위치는 폴더(디렉토리) 위치와 같아야 한다. (필수)
- 패키지 이름은 모두 소문자를 사용한다. (관례)
- 패키지 이름의 앞 부분에는 일반적으로 회사의 도메인 이름을 거꾸로 사용한다. 예를 들어,
`com.company.myapp` 과 같이 사용한다. (관례)
 - 이 부분은 필수는 아니다. 하지만 수 많은 외부 라이브러리가 함께 사용되면 같은 패키지에 같은 클래스 이름이 존재할 수도 있다. 이렇게 도메인 이름을 거꾸로 사용하면 이런 문제를 방지할 수 있다.
 - 내가 오픈소스나 라이브러리를 만들어서 외부에 제공한다면 꼭 지키는 것이 좋다.
 - 내가 만든 애플리케이션을 다른 곳에 공유하지 않고, 직접 배포한다면 보통 문제가 되지 않는다.

패키지와 계층 구조

패키지는 보통 다음과 같이 계층 구조를 이룬다.

- a
 - b
 - c

이렇게 하면 다음과 같이 총 3개의 패키지가 존재한다.

`a, a.b, a.c`

계층 구조상 `a` 패키지 하위에 `a.b` 패키지와 `a.c` 패키지가 있다.

그런데 이것은 우리 눈에 보기에 계층 구조를 이를 뿐이다. `a` 패키지와 `a.b, a.c` 패키지는 서로 완전히 다른 패키지이다.

따라서 `a` 패키지의 클래스에서 `a.b` 패키지의 클래스가 필요하면 `import` 해서 사용해야 한다. 반대도 물론 마찬가지이다.

정리하면 패키지가 계층 구조를 이루더라도 모든 패키지는 서로 다른 패키지이다.

물론 사람이 이해하기 쉽게 계층 구조를 잘 활용해서 패키지를 분류하는 것은 좋다. 참고로 카테고리는 보통 큰 분류에서 세세한 분류로 점점 나누어진다. 패키지도 마찬가지이다.

패키지 활용

실제 패키지가 어떤 식으로 사용되는지 예제를 통해서 알아보자. 실제 동작하는 코드는 아니지만, 큰 애플리케이션은 대략 이런식으로 패키지를 구성한다고 이해하면 된다. 참고로 이것은 정답은 아니고 프로젝트 규모와 아키텍처에 따라서 달라진다.

전체 구조도

- com.helloshop
 - user
 - ◆ User
 - ◆ UserService
 - product
 - ◆ Product
 - ◆ ProductService
 - order
 - ◆ Order
 - ◆ OrderService
 - ◆ OrderHistory

com.helloshop.user 패키지

```
package com.helloshop.user;

public class User {
    String userId;
    String name;
}
```

```
package com.helloshop.user;

public class UserService {
```

```
}
```

com.helloshop.product 패키지

```
package com.helloshop.product;

public class Product {
    String productId;
    int price;
}
```

```
package com.helloshop.product;

public class ProductService {
```

com.helloshop.order 패키지

```
package com.helloshop.order;

import com.helloshop.product.Product;
import com.helloshop.user.User;

public class Order {

    User user;
    Product product;

    public Order(User user, Product product) {
        this.user = user;
        this.product = product;
    }
}
```

다른 패키지의 기능이 필요하면 `import`를 사용하면 된다.

생성자를 보면 `public`이 붙어있다. `public`이 붙어있어야 다른 패키지에서 생성자를 호출할 수 있다.

```
package com.helloshop.order;

import com.helloshop.product.Product;
import com.helloshop.user.User;
```

```
public class OrderService {  
    public void order() {  
        User user = new User();  
        Product product = new Product();  
        Order order = new Order(user, product);  
    }  
}
```

```
package com.helloshop.order;  
  
public class OrderHistory {  
}
```

패키지를 구성할 때 서로 관련된 클래스는 하나의 패키지에 모으고, 관련이 적은 클래스는 다른 패키지로 분리하는 것이 좋다.

정리

6. 접근 제어자

#1.인강/0.자바/2.자바-기본

- /접근 제어자 이해1
- /접근 제어자 이해2
- /접근 제어자 종류
- /접근 제어자 사용 - 필드, 메서드
- /접근 제어자 사용 - 클래스 레벨
- /캡슐화
- /문제와 풀이
- /정리

접근 제어자 이해1

자바는 `public`, `private` 같은 접근 제어자(access modifier)를 제공한다. 접근 제어자를 사용하면 해당 클래스 외부에서 특정 필드나 메서드에 접근하는 것을 허용하거나 제한할 수 있다.

이런 접근 제어자가 왜 필요할까? 예제를 통해 접근 제어자가 필요한 이유를 알아보자.

여러분은 스피커에 들어가는 소프트웨어를 개발하는 개발자다.

스피커의 음량은 절대로 100을 넘으면 안된다는 요구사항이 있다. (**100을 넘어가면 스피커의 부품들이 고장난다.**)

스피커 객체를 만들어보자.

스피커는 음량을 높이고, 내리고, 현재 음량을 확인할 수 있는 단순한 기능을 제공한다.

요구사항대로 스피커의 음량은 100까지만 증가할 수 있다. 절대 100을 넘어가면 안된다.

Speaker

```
package access;

public class Speaker {

    int volume;

    Speaker(int volume) {
        this.volume = volume;
    }
}
```

```

void volumeUp() {
    if (volume >= 100) {
        System.out.println("음량을 증가할 수 없습니다. 최대 음량입니다.");
    } else {
        volume += 10;
        System.out.println("음량을 10 증가합니다.");
    }
}

void volumeDown() {
    volume -= 10;
    System.out.println("volumeDown 호출");
}

void showVolume() {
    System.out.println("현재 음량: " + volume);
}
}

```

생성자를 통해 초기 음량 값을 지정할 수 있다.

`volumeUp()` 메서드를 보자. 음량을 한번에 10씩 증가한다. 단 음량이 100을 넘게되면 더는 음량을 증가하지 않는다.

SpeakerMain

```

package access;

public class SpeakerMain {
    public static void main(String[] args) {
        Speaker speaker = new Speaker(90);
        speaker.showVolume();

        speaker.volumeUp();
        speaker.showVolume();

        speaker.volumeUp();
        speaker.showVolume();
    }
}

```

실행 결과

```

현재 음량: 90
음량을 10 증가합니다.

```

현재 음량: 100

음량을 증가할 수 없습니다. 최대 음량입니다.

현재 음량: 100

초기 음량 값을 90으로 지정했다. 그리고 음량을 높이는 메서드를 여러번 호출했다.

기대한 대로 음량은 100을 넘지 않았다. 프로젝트는 성공적으로 끝났다.

오랜 시간이 흘러서 업그레이드 된 다음 버전의 스피커를 출시하게 되었다. 이때는 새로운 개발자가 급하게 기존 코드를 이어받아서 개발을 하게 되었다. 참고로 새로운 개발자는 기존 요구사항을 잘 몰랐다. 코드를 실행해보니 이상하게 음량이 1000이상 올라가지 않았다. 소리를 더 올리면 좋겠다고 생각한 개발자는 다양한 방면으로 고민했다.

`Speaker` 클래스를 보니 `volume` 필드를 직접 사용할 수 있었다. `volume` 필드의 값을 200으로 설정하고 이 코드를 실행한 순간 스피커의 부품들에 과부하가 걸리면서 폭발했다.

SpeakerMain - 필드 직접 접근 코드 추가

```
package access;

public class SpeakerMain {
    public static void main(String[] args) {
        Speaker speaker = new Speaker(90);
        speaker.showVolume();

        speaker.volumeUp();
        speaker.showVolume();

        speaker.volumeUp();
        speaker.showVolume();

        //필드에 직접 접근
        System.out.println("volume 필드 직접 접근 수정");
        speaker.volume = 200;
        speaker.showVolume();
    }
}
```

실행 결과

현재 음량: 90

음량을 10 증가합니다.

현재 음량: 100

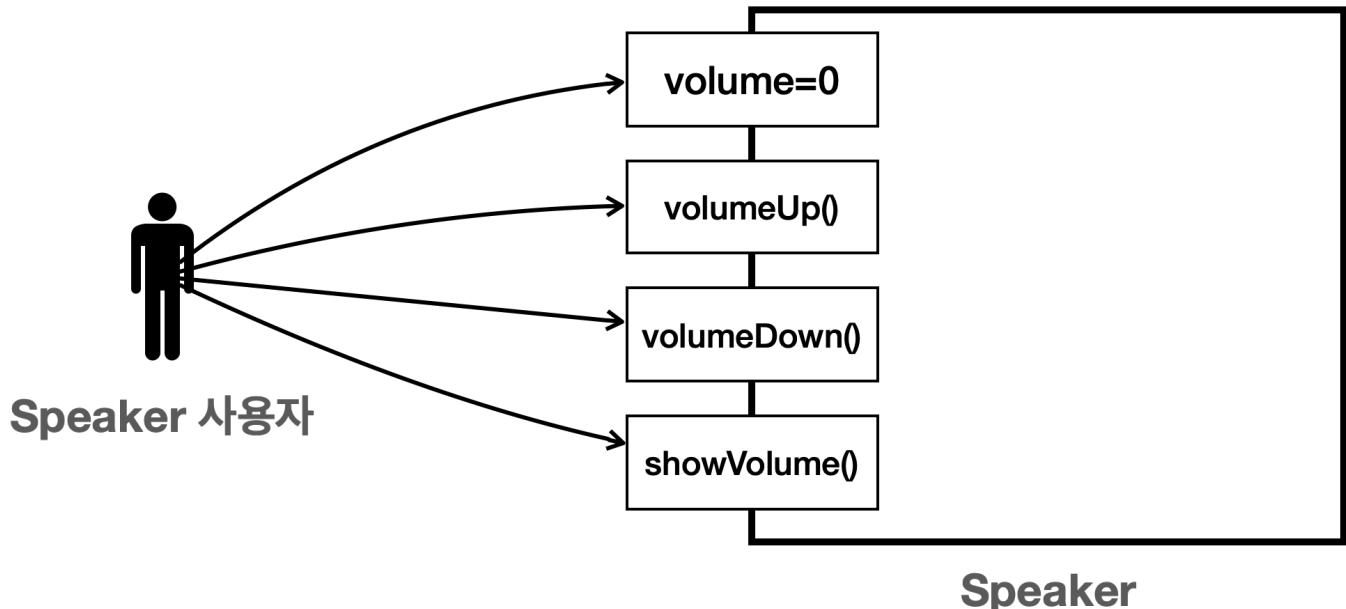
음량을 증가할 수 없습니다. 최대 음량입니다.

현재 음량: 100

volume 필드 직접 접근 수정

현재 음량: 200

volume 필드



Speaker 객체를 사용하는 사용자는 **Speaker**의 **volume** 필드와 메서드에 모두 접근할 수 있다.

앞서 **volumeUp()** 과 같은 메서드를 만들어서 음량이 100을 넘지 못하도록 기능을 개발했지만 소용이 없다. 왜냐하면 **Speaker**를 사용하는 입장에서는 **volume** 필드에 직접 접근해서 원하는 값을 설정할 수 있기 때문이다.

이런 문제를 근본적으로 해결하기 위해서는 **volume** 필드의 외부 접근을 막을 수 있는 방법이 필요하다.

접근 제어자 이해2

이 문제를 근본적으로 해결하는 방법은 **volume** 필드를 **Speaker** 클래스 외부에서는 접근하지 못하게 막는 것이다.

Speaker - volume 접근 제어자를 **private**으로 설정

```
package access;

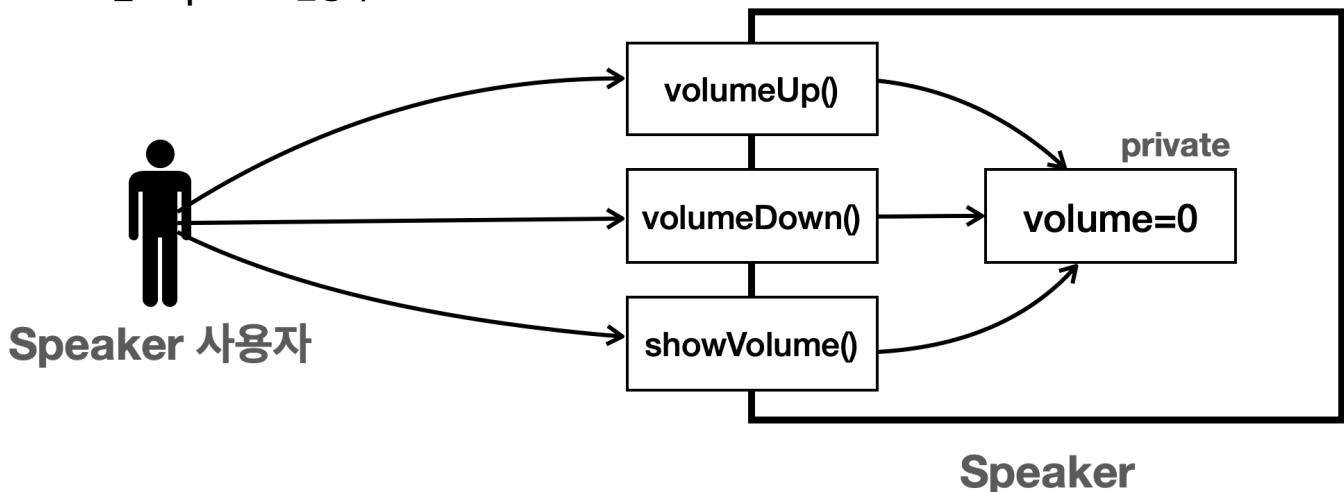
public class Speaker {

    private int volume; //private 사용
    ...
}
```

private 접근 제어자는 모든 외부 호출을 막는다. 따라서 **private**이 붙은 경우 해당 클래스 내부에서만 호출할 수

있다.

volume 필드 - private 변경 후



그림을 보면 volume 필드를 private을 사용해서 Speaker 내부에 숨겼다.

외부에서 volume 필드에 직접 접근할 수 없게 막은 것이다. volume 필드는 이제 Speaker 내부에서만 접근할 수 있다.

이제 SpeakerMain 코드를 다시 실행해보자.

```
//필드에 직접 접근
System.out.println("volume 필드 직접 접근 수정");
speaker.volume = 200; //private 접근 오류
```

IDE에서 speaker.volume = 200 부분에 오류가 발생하는 것을 확인할 수 있다. 실행해보면 다음과 같은 컴파일 오류가 발생한다.

컴파일 오류 메시지

```
volume has private access in access.Speaker
```

volume 필드는 private으로 설정되어 있기 때문에 외부에서 접근할 수 없다는 오류이다.

volume 필드 직접 접근 - 주석 처리

```
//필드에 직접 접근
System.out.println("volume 필드 직접 접근 수정");
//speaker.volume = 200; //private 접근 오류
speaker.showVolume();
```

이제 Speaker 외부에서 volume 필드에 직접 접근하는 것은 불가능하다. 이 경우 자바 컴파일러가 컴파일 오류를 발생시킨다.

프로그램을 실행하기 위해서 volume 필드에 직접 접근하는 코드를 주석 처리하자.

만약 Speaker 클래스를 개발하는 개발자가 처음부터 `private`을 사용해서 `volume` 필드의 외부 접근을 막아두었다면 어떠했을까? 새로운 개발자도 `volume` 필드에 직접 접근하지 않고, `volumeUp()`과 같은 메서드를 통해서 접근했을 것이다. 결과적으로 Speaker 가 폭팔하는 문제는 발생하지 않았을 것이다.

| 참고: 좋은 프로그램은 무한한 자유도가 주어지는 프로그램이 아니라 적절한 제약을 제공하는 프로그램이다.

접근 제어자 종류

자바는 4가지 종류의 접근 제어자를 제공한다.

접근 제어자의 종류

- `private`: 모든 외부 호출을 막는다.
- `default` (package-private): 같은 패키지안에서 호출은 허용한다.
- `protected`: 같은 패키지안에서 호출은 허용한다. 패키지가 달라도 상속 관계의 호출은 허용한다.
- `public`: 모든 외부 호출을 허용한다.

순서대로 `private`이 가장 많이 차단하고, `public`이 가장 많이 허용한다.

`private -> default -> protected -> public`

참고로 `protected`는 상속 관계에서 자세히 설명한다.

package-private

접근 제어자를 명시하지 않으면 같은 패키지 안에서 호출을 허용하는 `default` 접근 제어자가 적용된다.

`default`라는 용어는 해당 접근 제어자가 기본값으로 사용되기 때문에 붙여진 이름이지만, 실제로는 `package-private`이 더 정확한 표현이다. 왜냐하면 해당 접근 제어자를 사용하는 멤버는 동일한 패키지 내의 다른 클래스에서만 접근이 가능하기 때문이다. 참고로 두 용어를 함께 사용한다.

접근 제어자 사용 위치

접근 제어자는 필드와 메서드, 생성자에 사용된다.

추가로 클래스 레벨에도 일부 접근 제어자를 사용할 수 있다. 이 부분은 뒤에서 따로 설명한다.

접근 제어자 예시

```
public class Speaker { //클래스 레벨
```

```

private int volume; //필드

public Speaker(int volume) {} //생성자

public void volumeUp() {} //메서드
public void volumeDown() {}
public void showVolume() {}

}

```

접근 제어자의 핵심은 속성과 기능을 외부로부터 숨기는 것이다.

- `private`은 나의 클래스 안으로 속성과 기능을 숨길 때 사용, 외부 클래스에서 해당 기능을 호출할 수 없다.
- `default`는 나의 패키지 안으로 속성과 기능을 숨길 때 사용, 외부 패키지에서 해당 기능을 호출할 수 없다.
- `protected`는 상속 관계로 속성과 기능을 숨길 때 사용, 상속 관계가 아닌 곳에서 해당 기능을 호출할 수 없다.
- `public`은 기능을 숨기지 않고 어디서든 호출할 수 있게 공개한다.

접근 제어자 사용 - 필드, 메서드

다양한 상황에 따른 접근 제어자를 확인해보자.

주의! 지금부터는 패키지 위치가 매우 중요하다. 패키지 위치에 주의하자

필드, 메서드 레벨의 접근 제어자

AccessData

```

package access.a;

public class AccessData {

    public int publicField;
    int defaultField;
    private int privateField;

    public void publicMethod() {
        System.out.println("publicMethod 호출 " + publicField);
    }

    void defaultMethod() {

```

```

        System.out.println("defaultMethod 호출 " + defaultField);
    }

private void privateMethod() {
    System.out.println("privateMethod 호출 " + privateField);
}

public void innerAccess() {
    System.out.println("내부 호출");
    publicField = 100;
    defaultField = 200;
    privateField = 300;
    publicMethod();
    defaultMethod();
    privateMethod();
}
}

```

- 패키지 위치는 package access.a이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- 순서대로 public, default, private을 필드와 메서드에 사용했다.
- 마지막에 innerAccess() 가 있는데, 이 메서드는 내부 호출을 보여준다. 내부 호출은 자기 자신에게 접근하는 것이다. 따라서 private을 포함한 모든 곳에 접근할 수 있다.

이제 외부에서 이 클래스에 접근해보자.

AccessInnerMain

```

package access.a;

public class AccessInnerMain {
    public static void main(String[] args) {
        AccessData data = new AccessData();
        //public 호출 가능
        data.publicField = 1;
        data.publicMethod();

        //같은 패키지 default 호출 가능
        data.defaultField = 2;
        data.defaultMethod();

        //private 호출 불가
        //data.privateField = 3;
        //data.privateMethod();
    }
}

```

```
        data.innerAccess();
    }
}
```

- 패키지 위치는 package access.a 이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- public 은 모든 접근을 허용하기 때문에 필드, 메서드 모두 접근 가능하다.
- default 는 같은 패키지에서 접근할 수 있다. AccessInnerMain 은 AccessData 와 같은 패키지이다. 따라서 default 접근 제어자에 접근할 수 있다.
- private 은 AccessData 내부에서만 접근할 수 있다. 따라서 호출 불가다.
- AccessData.innerAccess() 메서드는 public 이다. 따라서 외부에서 호출할 수 있다.
innerAccess() 메서드는 외부에서 호출되었지만 innerAccess() 메서드는 AccessData 에 포함되어 있다. 이 메서드는 자신의 private 필드와 메서드에 모두 접근할 수 있다.

실행 결과

```
publicMethod 호출 1
defaultMethod 호출 2
내부 호출
publicMethod 호출 100
defaultMethod 호출 200
privateMethod 호출 300
```

AccessOuterMain

```
package access.b;

import access.a.AccessData;

public class AccessOuterMain {
    public static void main(String[] args) {
        AccessData data = new AccessData();
        //public 호출 가능
        data.publicField = 1;
        data.publicMethod();

        //다른 패키지 default 호출 불가
        //data.defaultField = 2;
        //data.defaultMethod();

        //private 호출 불가
        //data.privateField = 3;
        //data.privateMethod();
```

```
    data.innerAccess();  
}  
}
```

- 패키지 위치는 package access.b 이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- public은 모든 접근을 허용하기 때문에 필드, 메서드 모두 접근할 수 있다.
- default는 같은 패키지에서 접근할 수 있다. access.b.AccessOuterMain은 access.a.AccessData와 다른 패키지이다. 따라서 default 접근 제어자에 접근할 수 없다.
- private은 AccessData 내부에서만 접근할 수 있다. 따라서 호출 불가다.
- AccessData.innerAccess() 메서드는 public이다. 따라서 외부에서 호출할 수 있다. innerAccess() 메서드는 외부에서 호출되었지만 해당 메서드 안에서는 자신의 private 필드와 메서드에 접근할 수 있다.

실행 결과

```
publicMethod 호출 1  
내부 호출  
publicMethod 호출 100  
defaultMethod 호출 200  
privateMethod 호출 300
```

참고로 생성자도 접근 제어자 관점에서 메서드와 같다.

접근 제어자 사용 - 클래스 레벨

클래스 레벨의 접근 제어자 규칙

- 클래스 레벨의 접근 제어자는 public, default만 사용할 수 있다.
 - private, protected는 사용할 수 없다.
- public 클래스는 반드시 파일명과 이름이 같아야 한다.
 - 하나의 자바 파일에 public 클래스는 하나만 등장할 수 있다.
 - 하나의 자바 파일에 default 접근 제어자를 사용하는 클래스는 무한정 만들 수 있다.

PublicClass.java 파일

```
package access.a;  
  
public class PublicClass {  
    public static void main(String[] args) {
```

```

        PublicClass publicClass = new PublicClass();
        DefaultClass1 class1 = new DefaultClass1();
        DefaultClass2 class2 = new DefaultClass2();
    }
}

class DefaultClass1 {

}

class DefaultClass2 {
}

```

- 패키지 위치는 package access.a 이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- PublicClass 라는 이름의 클래스를 만들었다. 이 클래스는 public 접근 제어자다. 따라서 파일명과 이 클래스의 이름이 반드시 같아야 한다. 이 클래스는 public 이기 때문에 외부에서 접근할 수 있다.
- DefaultClass1, DefaultClass2 는 default 접근 제어자다. 이 클래스는 default 이기 때문에 같은 패키지 내부에서만 접근할 수 있다.
- PublicClass 의 main() 을 보면 각각의 클래스를 사용하는 예를 보여준다.
 - PublicClass 는 public 접근 제어다. 따라서 어디서든 사용할 수 있다. DefaultClass1, DefaultClass2 와는 같은 패키지에 있으므로 사용할 수 있다.

PublicClassInnerMain

```

package access.a;

public class PublicClassInnerMain {
    public static void main(String[] args) {
        PublicClass publicClass = new PublicClass();
        DefaultClass1 class1 = new DefaultClass1();
        DefaultClass2 class2 = new DefaultClass2();
    }
}

```

- 패키지 위치는 package access.a 이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- PublicClass 는 public 클래스이다. 따라서 외부에서 접근할 수 있다.
- PublicClassInnerMain 와 DefaultClass1, DefaultClass2 는 같은 패키지이다. 따라서 접근할 수 있다.

```

package access.b;

//import access.a.DefaultClass1;
import access.a.PublicClass;

```

```
public class PublicClassOuterMain {  
    public static void main(String[] args) {  
        PublicClass publicClass = new PublicClass();  
  
        //다른 패키지 접근 불가  
        //DefaultClass1 class1 = new DefaultClass1();  
        //DefaultClass2 class2 = new DefaultClass2();  
    }  
}
```

- 패키지 위치는 package access.b 이다. 패키지 위치를 꼭 맞추어야 한다. 주의하자.
- PublicClass 는 public 클래스이다. 따라서 외부에서 접근할 수 있다.
- PublicClassOuterMain 와 DefaultClass1 , DefaultClass2 는 다른 패키지이다. 따라서 접근할 수 없다.

캡슐화

캡슐화(Encapsulation)는 객체 지향 프로그래밍의 중요한 개념 중 하나다. 캡슐화는 데이터와 해당 데이터를 처리하는 메서드를 하나로 묶어서 외부에서의 접근을 제한하는 것을 말한다. 캡슐화를 통해 데이터의 직접적인 변경을 방지하거나 제한할 수 있다.

캡슐화는 쉽게 이야기해서 속성과 기능을 하나로 묶고, 외부에 꼭 필요한 기능만 노출하고 나머지는 모두 내부로 숨기는 것이다.

이전에 객체 지향 프로그래밍을 설명하면서 캡슐화에 대해 알아보았다. 이때는 데이터와 데이터를 처리하는 메서드를 하나로 모으는 것에 초점을 맞추었다. 여기서 한발짝 더 나아가 캡슐화를 안전하게 완성할 수 있게 해주는 장치가 바로 접근 제어자다.

그럼 어떤 것을 숨기고 어떤 것을 노출해야 할까?

1. 데이터를 숨겨라

객체에는 속성(데이터)과 기능(메서드)이 있다. 캡슐화에서 가장 필수로 숨겨야 하는 것은 속성(데이터)이다.

Speaker 의 volume 을 떠올려보자. 객체 내부의 데이터를 외부에서 함부로 접근하게 두면, 클래스 안에서 데이터를 다른 모든 로직을 무시하고 데이터를 변경할 수 있다. 결국 모든 안전망을 다 빠져나가게 된다. 따라서 캡슐화가 깨진다.

우리가 자동차를 운전할 때 자동차 부품을 다 열어서 그 안에 있는 속도계를 직접 조절하지 않는다. 단지 자동차가 제공

하는 엑셀 기능을 사용해서 엑셀을 밟으면 자동차가 나머지는 다 알아서 하는 것이다.

우리가 일상에서 생각할 수 있는 음악 플레이어를 떠올려보자. 음악 플레이어를 사용할 때 그 내부에 들어있는 전원부나, 볼륨 상태의 데이터를 직접 수정할 일이 있을까? 우리는 그냥 음악 플레이어의 켜고, 끄고, 볼륨을 조절하는 버튼을 누를 뿐이다. 그 내부에 있는 전원부나, 볼륨의 상태 데이터를 직접 수정하지 않는다. 전원 버튼을 눌렀을 때 실제 전원을 받아서 전원을 켜는 것은 음악 플레이어의 일이다. 볼륨을 높였을 때 내부에 있는 볼륨 장치들을 움직이고 볼륨 수치를 조절하는 것도 음악 플레이어가 스스로 해야하는 일이다. 쉽게 이야기해서 우리는 음악 플레이어가 제공하는 기능을 통해서 음악 플레이어를 사용하는 것이다. 복잡하게 음악 플레이어의 내부를 까서 그 내부 데이터까지 우리가 직접 사용하는 것은 아니다.

객체의 데이터는 객체가 제공하는 기능인 메서드를 통해서 접근해야 한다.

2. 기능을 숨겨라

객체의 기능 중에서 외부에서 사용하지 않고 내부에서만 사용하는 기능들이 있다. 이런 기능도 모두 감추는 것이 좋다. 우리가 자동차를 운전하기 위해 자동차가 제공하는 복잡한 엔진 조절 기능, 배기 기능까지 우리가 알 필요는 없다. 우리는 단지 엑셀과 핸들 정도의 기능만 알면 된다.

만약 사용자에게 이런 기능까지 모두 알려준다면, 사용자가 자동차에 대해 너무 많은 것을 알아야 한다.

사용자 입장에서 꼭 필요한 기능만 외부에 노출하자. 나머지 기능은 모두 내부로 숨기자

정리하면 데이터는 모두 숨기고, 기능은 꼭 필요한 기능만 노출하는 것이 좋은 캡슐화이다.

이번에는 잘 캡슐화된 예제를 하나 만들어보자.

BankAccount

```
package access;

public class BankAccount {

    private int balance;

    public BankAccount() {
        balance = 0;
    }

    // public 메서드: deposit
    public void deposit(int amount) {
        if (isAmountValid(amount)) {
            balance += amount;
        } else {
    }
}
```

```

        System.out.println("유효하지 않은 금액입니다.");
    }
}

// public 메서드: withdraw
public void withdraw(int amount) {
    if (isAmountValid(amount) && balance - amount >= 0) {
        balance -= amount;
    } else {
        System.out.println("유효하지 않은 금액이거나 잔액이 부족합니다.");
    }
}

// public 메서드: getBalance
public int getBalance() {
    return balance;
}

// private 메서드: isAmountValid
private boolean isAmountValid(int amount) {
    // 금액이 0보다 커야함
    return amount > 0;
}
}

```

```

package access;

public class BankAccountMain {

    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.deposit(10000);
        account.withdraw(3000);
        System.out.println("balance = " + account.getBalance());
    }
}

```

은행 계좌 기능을 다룬다. 다음과 같은 기능을 가지고 있다.

private

- balance: 데이터 필드는 외부에 직접 노출하지 않는다. BankAccount 가 제공하는 메서드를 통해서만 접근할 수 있다.
- isAmountValid(): 입력 금액을 검증하는 기능은 내부에서만 필요한 기능이다. 따라서 private 을 사용했

다.

public

- `deposit()` : 입금
- `withdraw()` : 출금
- `getBalance()` : 잔고

`BankAccount` 를 사용하는 입장에서는 단 3가지 메서드만 알면 된다. 나머지 복잡한 내용은 모두 `BankAccount` 내부에 숨어있다.

만약 `isAmountValid()` 를 외부에 노출하면 어떻게 될까? `BankAccount` 를 사용하는 개발자 입장에서는 사용할 수 있는 메서드가 하나 더 늘었다. 여러분이 `BankAccount` 를 사용하는 개발자라면 어떤 생각을 할까? 아마도 입금과 출금 전에 본인이 먼저 `isAmountValid()` 를 사용해서 검증을 해야 하나? 라고 의문을 가질 것이다.

만약 `balance` 필드를 외부에 노출하면 어떻게 될까? `BankAccount` 를 사용하는 개발자 입장에서는 이 필드를 직접 사용해도 된다고 생각할 수 있다. 왜냐하면 외부에 공개하는 것은 그것을 외부에서 사용해도 된다는 뜻이기 때문이다. 결국 모든 검증과 캡슐화가 깨지고 잔고를 무한정 늘리고 출금하는 심각한 문제가 발생할 수 있다.

접근 제어자와 캡슐화를 통해 데이터를 안전하게 보호하는 것은 물론이고, `BankAccount` 를 사용하는 개발자 입장에서 해당 기능을 사용하는 복잡도도 낮출 수 있다.

문제와 풀이

문제 - 최대 카운터와 캡슐화

`MaxCounter` 클래스를 만드세요.

이 클래스는 최대값을 지정하고 최대값 까지만 값이 증가하는 기능을 제공합니다.

- `int count` : 내부에서 사용하는 숫자입니다. 초기값은 0입니다.
- `int max` : 최대값입니다. 생성자를 통해 입력합니다.
- `increment()` 숫자를 하나 증가합니다.
- `getCount()` 지금까지 증가한 값을 반환합니다.

요구사항

- 접근 제어자를 사용해서 데이터를 캡슐화 하세요.

- 해당 클래스는 다른 패키지에서도 사용할 수 있어야 합니다.

```
package access.ex;

public class CounterMain {
    public static void main(String[] args) {
        MaxCounter counter = new MaxCounter(3);
        counter.increment();
        counter.increment();
        counter.increment();
        counter.increment();
        int count = counter.getCount();
        System.out.println(count);
    }
}
```

실행 결과

최대값을 초과할 수 없습니다.

3

정답

```
package access.ex;

public class MaxCounter {
    private int count = 0;
    private int max;

    public MaxCounter(int max) {
        this.max = max;
    }

    public void increment() {
        if (count >= max) {
            System.out.println("최대값을 초과할 수 없습니다.");
            return;
        }
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

```
    }  
}
```

문제 - 쇼핑 카트

ShoppingCartMain 코드가 작동하도록 Item, ShoppingCart 클래스를 완성해라.

요구사항

- 접근 제어자를 사용해서 데이터를 캡슐화 하세요.
- 해당 클래스는 다른 패키지에서도 사용할 수 있어야 합니다.
- 장바구니에는 상품을 최대 10개만 담을 수 있다.
 - 10개 초과 등록시: "장바구니가 가득 찼습니다." 출력

```
package access.ex;  
  
public class ShoppingCartMain {  
  
    public static void main(String[] args) {  
        ShoppingCart cart = new ShoppingCart();  
  
        Item item1 = new Item("마늘", 2000, 2);  
        Item item2 = new Item("상추", 3000, 4);  
  
        cart.addItem(item1);  
        cart.addItem(item2);  
  
        cart.displayItems();  
    }  
}
```

실행 결과

```
장바구니 상품 출력  
상품명:마늘, 합계:4000  
상품명:상추, 합계:12000  
전체 가격 합:16000
```

Item 클래스

```
package access.ex;
```

```
public class Item {  
    private String name;  
    private int price;  
    private int quantity;  
  
    //TODO 나머지 코드를 완성해라.  
}
```

ShoppingCart 클래스

```
package access.ex;  
  
public class ShoppingCart {  
    private Item[] items = new Item[10];  
    private int itemCount;  
  
    //TODO 나머지 코드를 완성해라.  
}
```

정답

```
package access.ex;  
  
public class Item {  
    private String name;  
    private int price;  
    private int quantity;  
  
    public Item(String name, int price, int quantity) {  
        this.name = name;  
        this.price = price;  
        this.quantity = quantity;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getTotalPrice() {  
        return price * quantity;  
    }  
}
```

각각의 `Item`의 가격과 수량을 곱하면 각 상품별 합계를 구할 수 있다. `price`와 `quantity`를 외부에 반환한 다음에 외부에서 곱해서 상품별 합계를 구해도 되지만, `getTotalPrice()` 메서드를 제공하면 외부에서는 단순히 이 메서드를 호출하면 된다. 이 메서드의 핵심은 자신이 가진 데이터를 사용한다는 점이다.

```
package access.ex;

public class ShoppingCart {
    private Item[] items = new Item[10];
    private int itemCount;

    public void addItem(Item item) {
        if (itemCount >= items.length) {
            System.out.println("장바구니가 가득 찼습니다.");
            return;
        }

        items[itemCount] = item;
        itemCount++;
    }

    public void displayItems() {
        System.out.println("장바구니 상품 출력");
        for (int i = 0; i < itemCount; i++) {
            Item item = items[i];
            System.out.println("상품명:" + item.getName() + ", 합계:" +
item.getTotalPrice());
        }
        System.out.println("전체 가격 합:" + calculateTotalPrice());
    }

    private int calculateTotalPrice() {
        int totalPrice = 0;
        for (int i = 0; i < itemCount; i++) {
            Item item = items[i];
            totalPrice += item.getTotalPrice();
        }
        return totalPrice;
    }
}

● calculateTotalPrice(): 이 메서드 내부에서만 사용되므로 private 접근 제어자를 사용한다.
```

정리

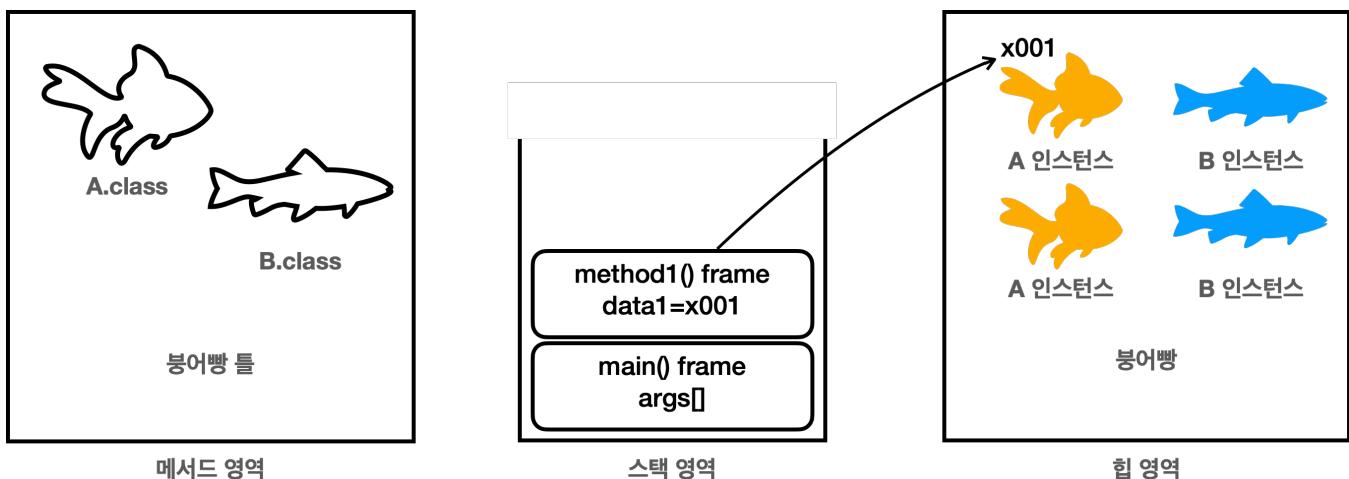
7. 자바 메모리 구조와 static

#1.인강/0.자바/2.자바-기본

- /자바 메모리 구조
- /스택과 큐 자료 구조
- /스택 영역
- /스택 영역과 힙 영역
- /static 변수1
- /static 변수2
- /static 변수3
- /static 메서드1
- /static 메서드2
- /static 메서드3
- /문제와 풀이
- /정리

자바 메모리 구조

자바 메모리 구조 - 비유

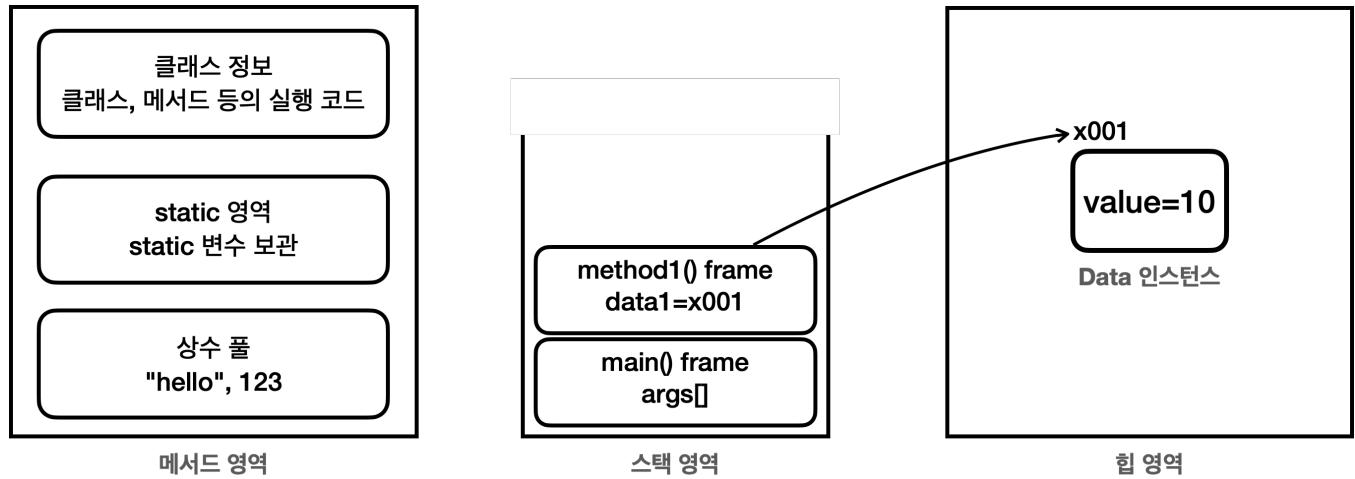


자바의 메모리 구조는 크게 메서드 영역, 스택 영역, 힙 영역 3개로 나눌 수 있다.

- **메서드 영역:** 클래스 정보를 보관한다. 이 클래스 정보가 붕어빵 틀이다.
- **스택 영역:** 실제 프로그램이 실행되는 영역이다. 메서드를 실행할 때마다 하나씩 쌓인다.
- **힙 영역:** 객체(인스턴스)가 생성되는 영역이다. `new` 명령어를 사용하면 이 영역을 사용한다. 쉽게 이야기해서 붕어빵 틀로부터 생성된 붕어빵이 존재하는 공간이다. 참고로 배열도 이 영역에 생성된다.

방금 설명한 내용은 쉽게 비유로 한 것이고 실제는 다음과 같다.

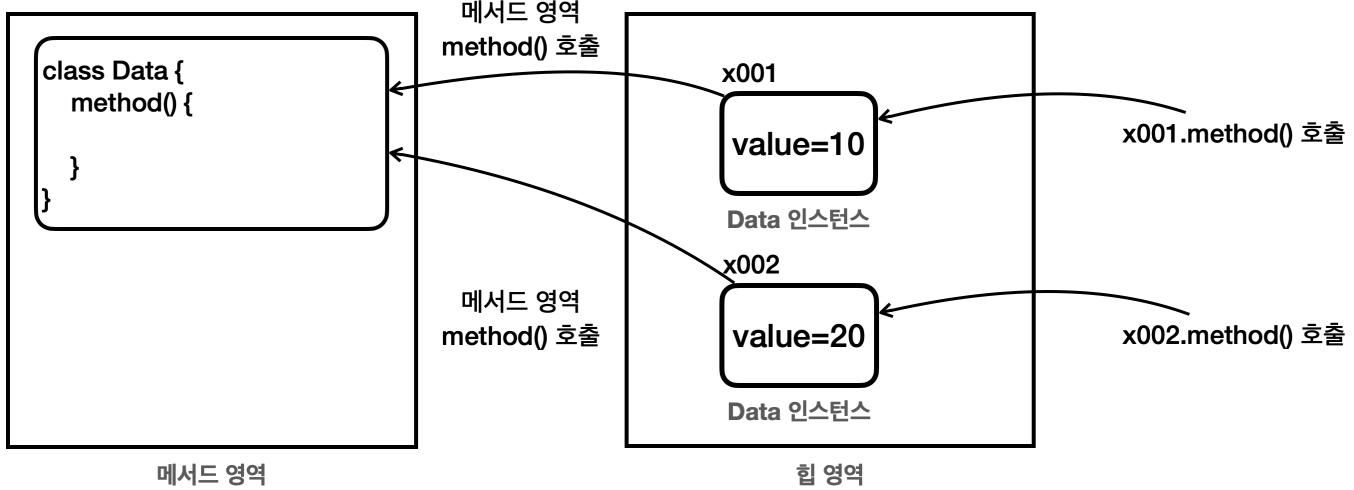
자바 메모리 구조 - 실제



- **메서드 영역(Method Area):** 메서드 영역은 프로그램을 실행하는데 필요한 공통 데이터를 관리한다. 이 영역은 프로그램의 모든 영역에서 공유한다.
 - 클래스 정보: 클래스의 실행 코드(바이트 코드), 필드, 메서드와 생성자 코드 등 모든 실행 코드가 존재한다.
 - static 영역: static 변수들을 보관한다. 뒤에서 자세히 설명한다.
 - 런타임 상수 풀: 프로그램을 실행하는데 필요한 공통 리터럴 상수를 보관한다. 예를 들어서 프로그램에 "hello"라는 리터럴 문자가 있으면 이런 문자를 공통으로 묶어서 관리한다. 이 외에도 프로그램을 효율적으로 관리하기 위한 상수들을 관리한다.
- **스택 영역(Stack Area):** 자바 실행 시, 하나의 실행 스택이 생성된다. 각 스택 프레임은 지역 변수, 중간 연산 결과, 메서드 호출 정보 등을 포함한다.
 - 스택 프레임: 스택 영역에 쌓이는 네모 박스가 하나의 스택 프레임이다. 메서드를 호출할 때마다 하나의 스택 프레임이 쌓이고, 메서드가 종료되면 해당 스택 프레임이 제거된다.
- **힙 영역(Heap Area):** 객체(인스턴스)와 배열이 생성되는 영역이다. 가비지 컬렉션(GC)이 이루어지는 주요 영역이며, 더 이상 참조되지 않는 객체는 GC에 의해 제거된다.

참고: 스택 영역은 더 정확히는 각 쓰레드별로 하나의 실행 스택이 생성된다. 따라서 쓰레드 수 만큼 스택 영역이 생성된다. 지금은 쓰레드를 1개만 사용하므로 스택 영역도 하나이다. 쓰레드에 대한 부분은 멀티 쓰레드를 학습해야 이해할 수 있다.

메서드 코드는 메서드 영역에



자바에서 특정 클래스로 100개의 인스턴스를 생성하면, 힙 메모리에 100개의 인스턴스가 생긴다. 각각의 인스턴스는 내부에 변수와 메서드를 가진다. 같은 클래스로 부터 생성된 객체라도, 인스턴스 내부의 변수 값은 서로 다를 수 있지만, 메서드는 공통된 코드를 공유한다. 따라서 객체가 생성될 때, 인스턴스 변수에는 메모리가 할당되지만, 메서드에 대한 새로운 메모리 할당은 없다. 메서드는 메서드 영역에서 공통으로 관리되고 실행된다.

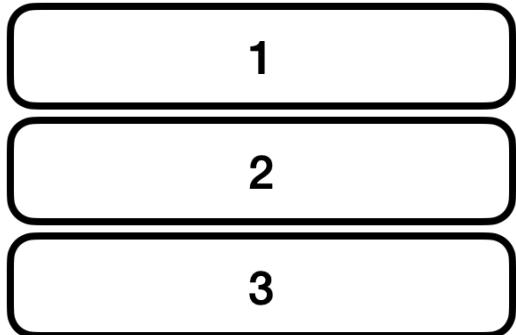
정리하면 인스턴스의 메서드를 호출하면 실제로는 메서드 영역에 있는 코드를 불러서 수행한다.

스택과 큐 자료 구조

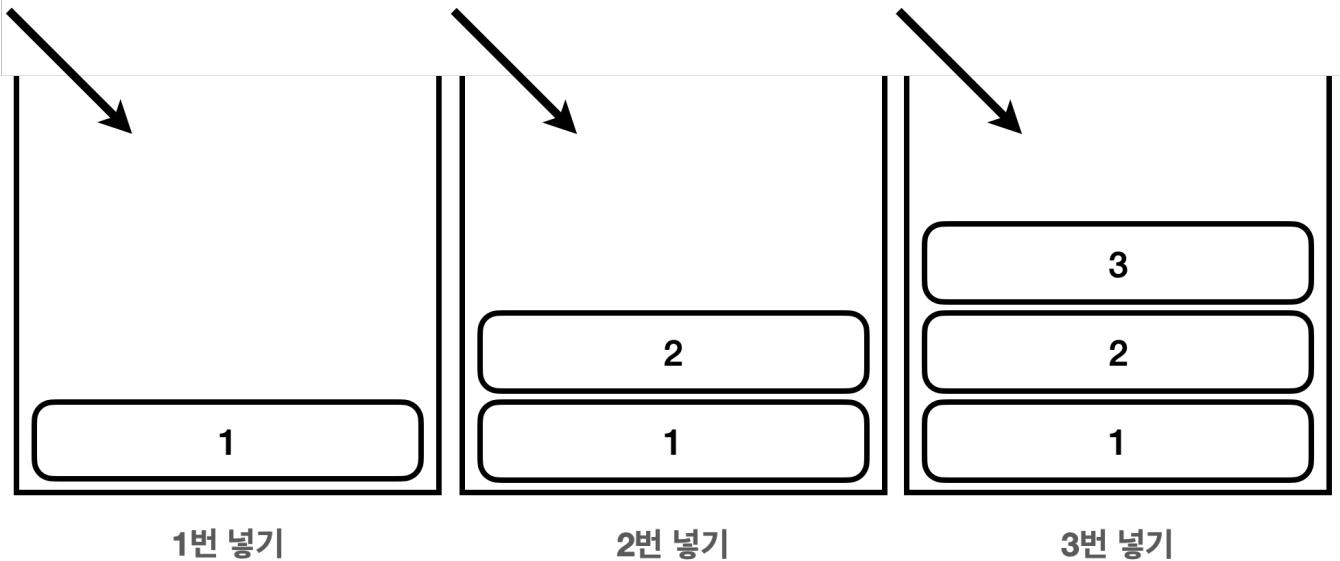
자바 메모리 구조 중 스택 영역에 대해 알아보기 전에 먼저 스택(Stack)이라는 자료 구조에 대해서 알아보자.

스택 구조

다음과 같은 1, 2, 3 이름표가 붙은 블럭이 있다고 가정하자.

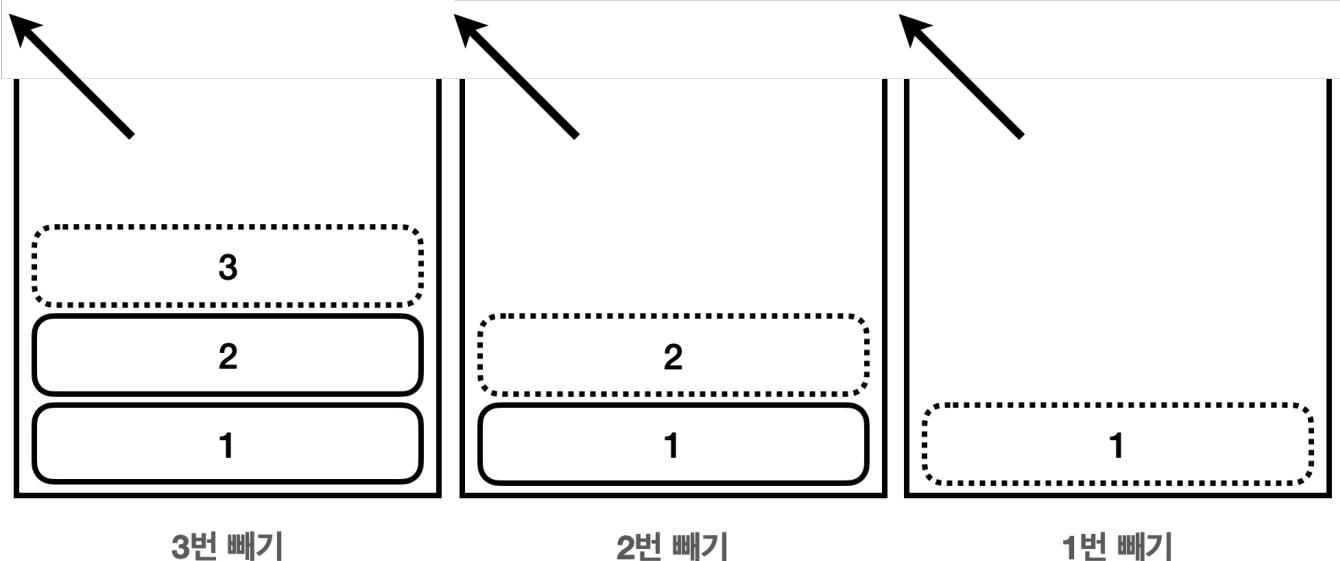


이 블럭을 다음과 같이 생긴 통에 넣는다고 생각해보자. 위쪽만 열려있기 때문에 위쪽으로 블럭을 넣고, 위쪽으로 블럭을 빼야 한다. 쉽게 이야기해서 넣는 곳과 빼는 곳이 같다.



블럭은 $1 \rightarrow 2 \rightarrow 3$ 순서대로 넣을 수 있다.

이번에는 넣은 블럭을 빼자.



블럭을 빼려면 위에서부터 순서대로 빼야한다.

블럭은 $3 \rightarrow 2 \rightarrow 1$ 순서로 뺄 수 있다.

정리하면 다음과 같다.

1(넣기) \rightarrow 2(넣기) \rightarrow 3(넣기) \rightarrow 3(빼기) \rightarrow 2(빼기) \rightarrow 1(빼기)

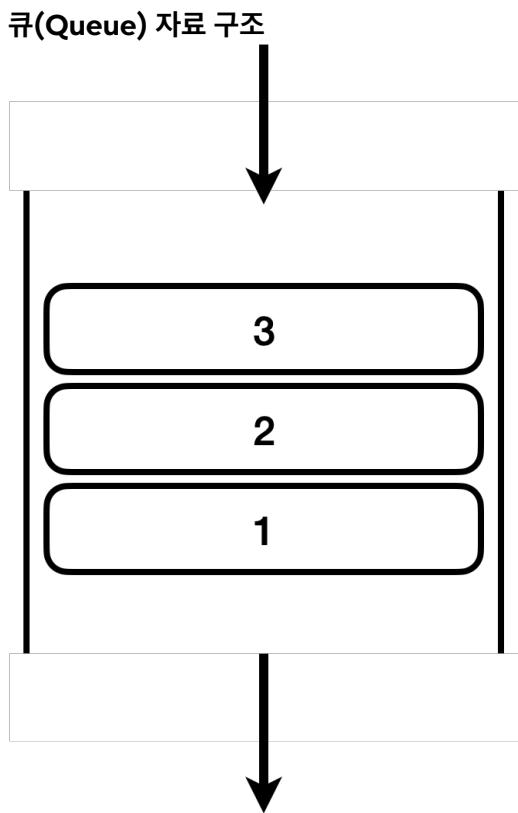
후입 선출(LIFO, Last In First Out)

여기서 가장 마지막에 넣은 3번이 가장 먼저 나온다. 이렇게 나중에 넣은 것이 가장 먼저 나오는 것을 후입 선출이라 하고, 이런 자료 구조를 스택이라 한다.

선입 선출(FIFO, First In First Out)

후입 선출과 반대로 가장 먼저 넣은 것이 가장 먼저 나오는 것을 선입 선출이라 한다. 이런 자료 구조를 큐(Queue)라

한다.



정리하면 다음과 같다.

1(넣기) → 2(넣기) → 3(넣기) → 1(빼기) → 2(빼기) → 3(빼기)

이런 자료 구조는 각자 필요한 영역이 있다. 예를 들어서 선착순 이벤트를 하는데 고객이 대기해야 한다면 큐 자료 구조를 사용해야 한다.

이번시간에 중요한 것은 스택이다. 프로그램 실행과 메서드 호출에는 스택 구조가 적합하다. 스택 구조를 학습했으니, 자바에서 스택 영역이 어떤 방식으로 작동하는지 알아보자.

스택 영역

다음 코드를 실행하면 스택 영역에서 어떤 변화가 있는지 확인해보자.

JavaMemoryMain1

```
package memory;

public class JavaMemoryMain1 {
```

```

public static void main(String[] args) {
    System.out.println("main start");
    method1(10);
    System.out.println("main end");
}

static void method1(int m1) {
    System.out.println("method1 start");
    int cal = m1 * 2;
    method2(cal);
    System.out.println("method1 end");
}

static void method2(int m2) {
    System.out.println("method2 start");
    System.out.println("method2 end");
}

}

```

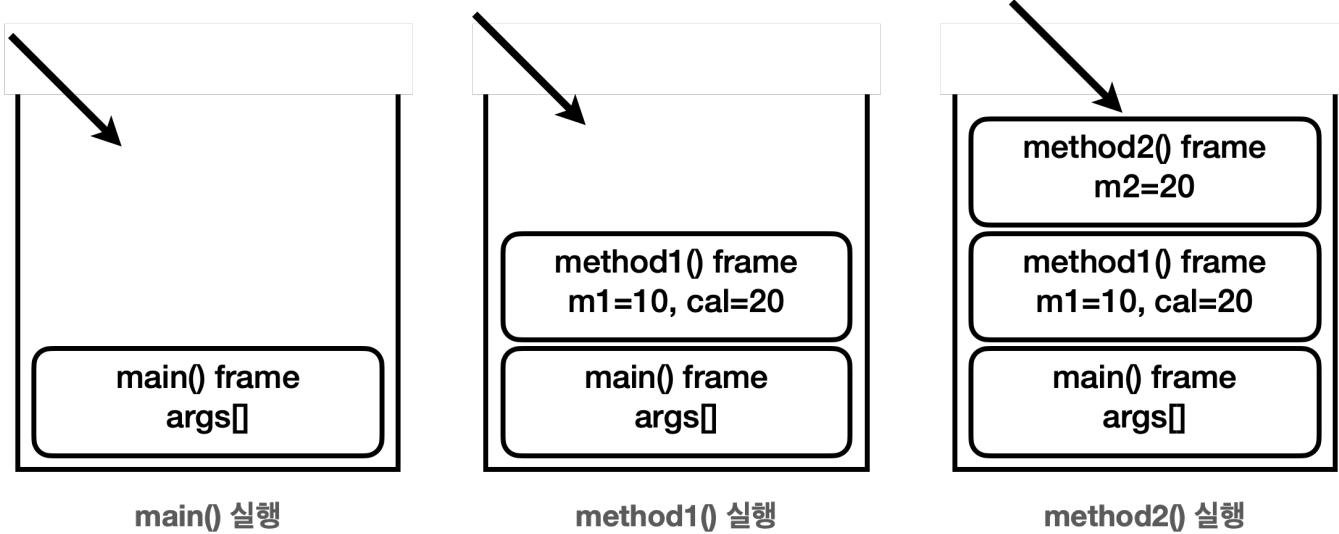
실행 결과

```

main start
method1 start
method2 start
method2 end
method1 end
main end

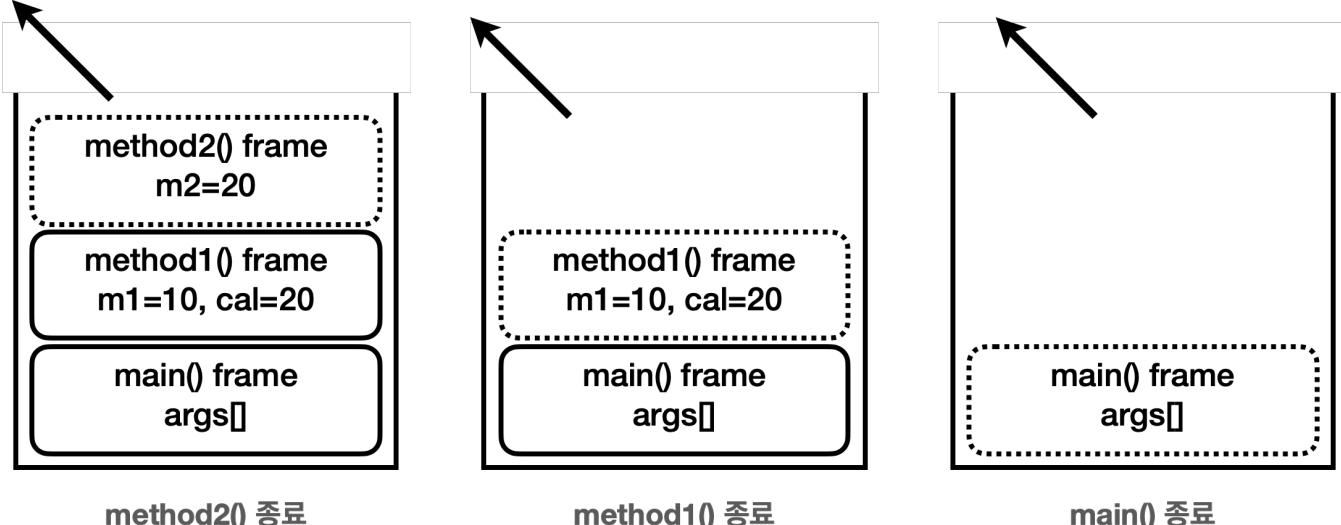
```

호출 그림



- 처음 자바 프로그램을 실행하면 `main()` 을 실행한다. 이때 `main()` 을 위한 스택 프레임이 하나 생성된다.
 - `main()` 스택 프레임은 내부에 `args`라는 매개변수를 가진다. `args`는 뒤에서 다룬다.
- `main()` 은 `method1()` 을 호출한다. `method1()` 스택 프레임이 생성된다.
 - `method1()` 는 `m1`, `cal` 지역 변수(매개변수 포함)를 가지므로 해당 지역 변수들이 스택 프레임에 포함된다.
- `method1()` 은 `method2()` 를 호출한다. `method2()` 스택 프레임이 생성된다. `method2()` 는 `m2` 지역 변수(매개변수 포함)를 가지므로 해당 변수가 스택 프레임에 포함된다.

종료 그림



- `method2()` 가 종료된다. 이때 `method2()` 스택 프레임이 제거되고, 매개변수 `m2` 도 제거된다. `method2()` 스택 프레임이 제거 되었으므로 프로그램은 `method1()` 로 돌아간다. 물론 `method1()` 을 처음부터 시작하는 것이 아니라 `method1()` 에서 `method2()` 를 호출한 지점으로 돌아간다.
- `method1()` 이 종료된다. 이때 `method1()` 스택 프레임이 제거되고, 지역 변수(매개변수 포함) `m1`, `cal` 도 제거된다. 프로그램은 `main()` 으로 돌아간다.
- `main()` 이 종료된다. 더 이상 호출할 메서드가 없고, 스택 프레임도 완전히 비워졌다. 자바는 프로그램을 정리하고 종료한다.

정리

- 자바는 스택 영역을 사용해서 메서드 호출과 지역 변수(매개변수 포함)를 관리한다.
- 메서드를 계속 호출하면 스택 프레임이 계속 쌓인다.
- 지역 변수(매개변수 포함)는 스택 영역에서 관리한다.
- 스택 프레임이 종료되면 지역 변수도 함께 제거된다.
- 스택 프레임이 모두 제거되면 프로그램도 종료된다.

스택 영역과 힙 영역

이번에는 스택 영역과 힙 영역이 함께 사용되는 경우를 알아보자.

Data

```
package memory;

public class Data {
    private int value;

    public Data(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

JavaMemoryMain2

```
package memory;

public class JavaMemoryMain2 {

    public static void main(String[] args) {
        System.out.println("main start");
        method1();
        System.out.println("main end");
    }

    static void method1() {
        System.out.println("method1 start");
        Data data1 = new Data(10);
        method2(data1);
        System.out.println("method1 end");
    }

    static void method2(Data data2) {
        System.out.println("method2 start");
        System.out.println("data.value=" + data2.getValue());
    }
}
```

```
        System.out.println("method2 end");
    }

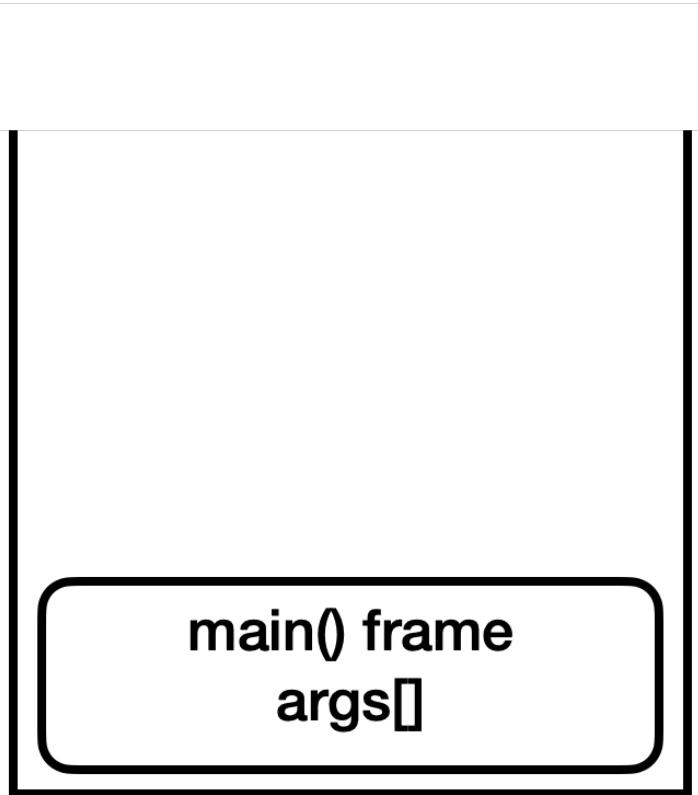
}
```

- `main()` → `method1()` → `method2()` 순서로 호출하는 단순한 코드이다.
- `method1()`에서 `Data` 클래스의 인스턴스를 생성한다.
- `method1()`에서 `method2()`를 호출할 때 매개변수에 `Data` 인스턴스의 참조값을 전달한다.

실행 결과

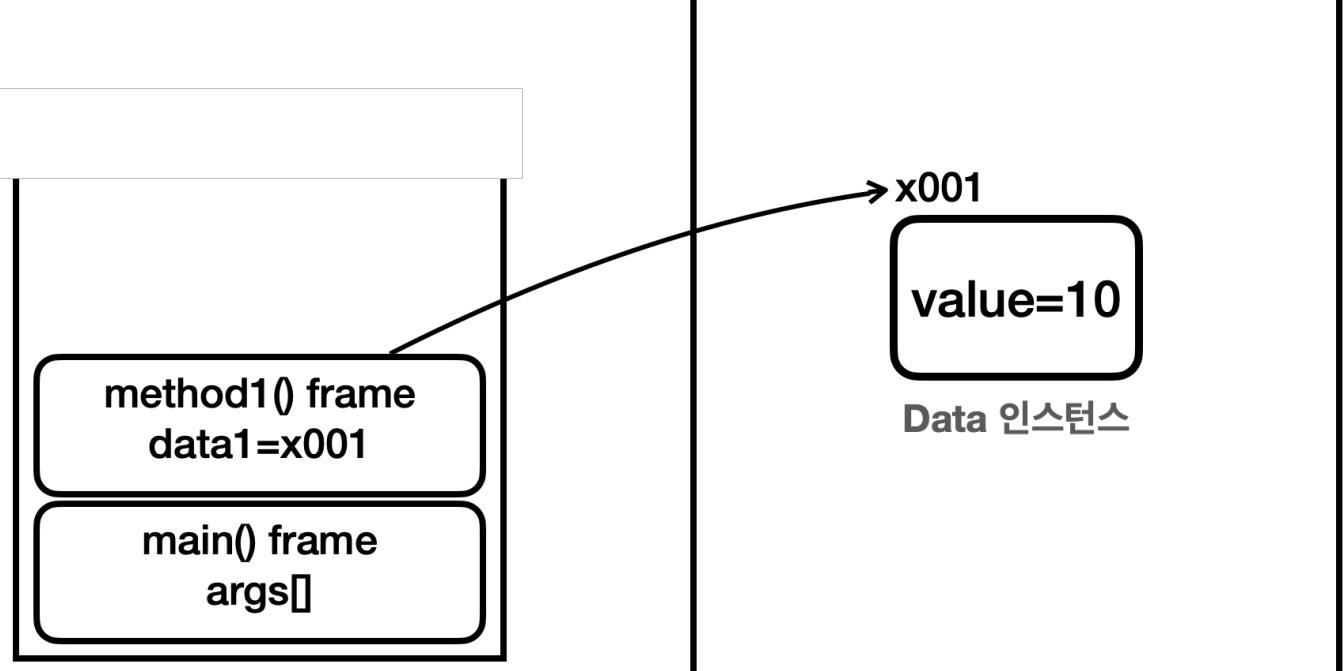
```
main start
method1 start
method2 start
data.value=10
method2 end
method1 end
main end
```

그림을 통해 순서대로 알아보자.



main() 실행

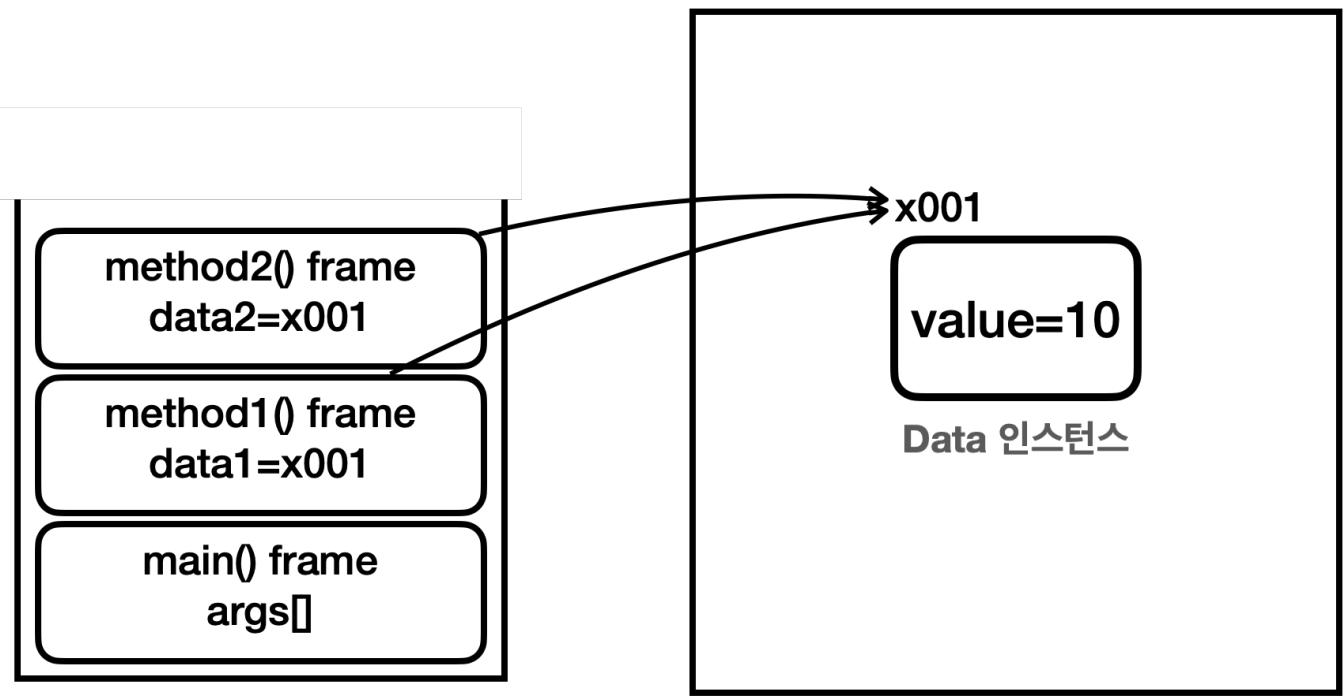
- 처음 `main()` 메서드를 실행한다. `main()` 스택 프레임이 생성된다.



method1() 실행

힙 영역

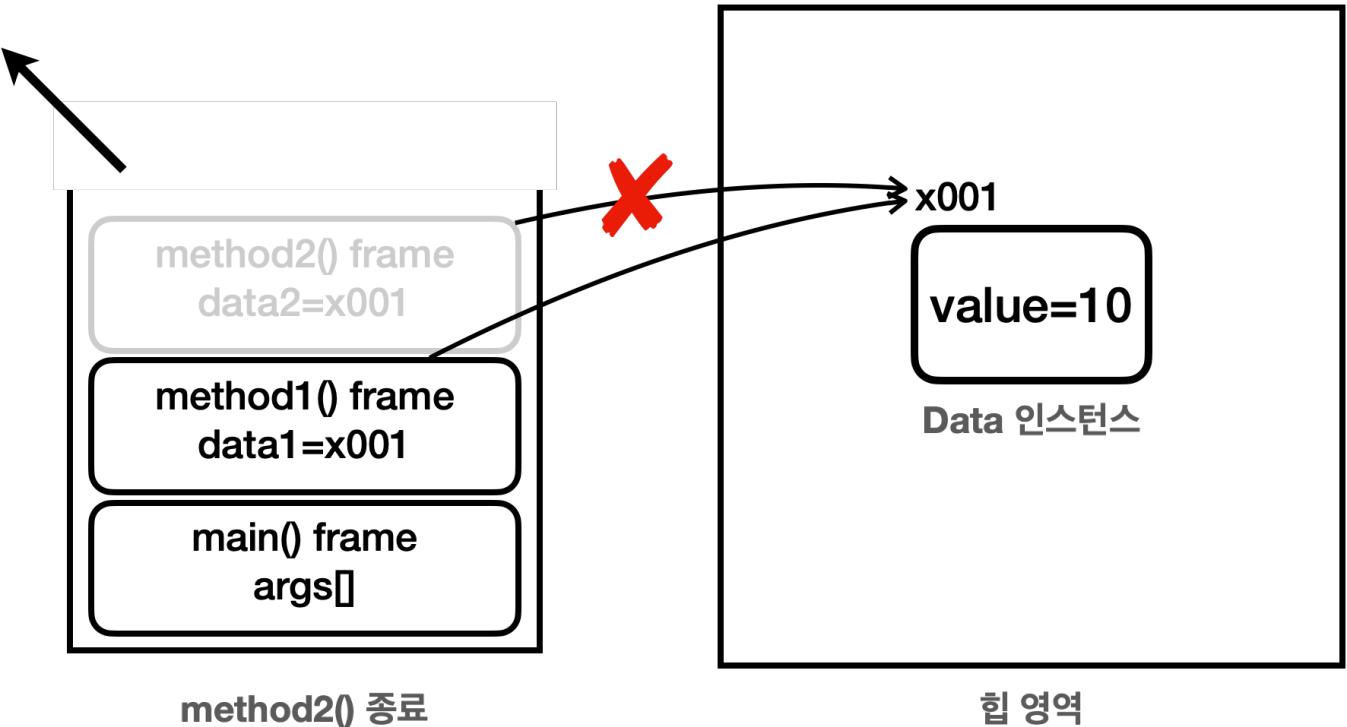
- `main()`에서 `method1()`을 실행한다. `method1()` 스택 프레임이 생성된다.
- `method1()`은 지역 변수로 `Data data1`을 가지고 있다. 이 지역 변수도 스택 프레임에 포함된다.
- `method1()`은 `new Data(10)`를 사용해서 힙 영역에 `Data` 인스턴스를 생성한다. 그리고 참조값을 `data1`에 보관한다.



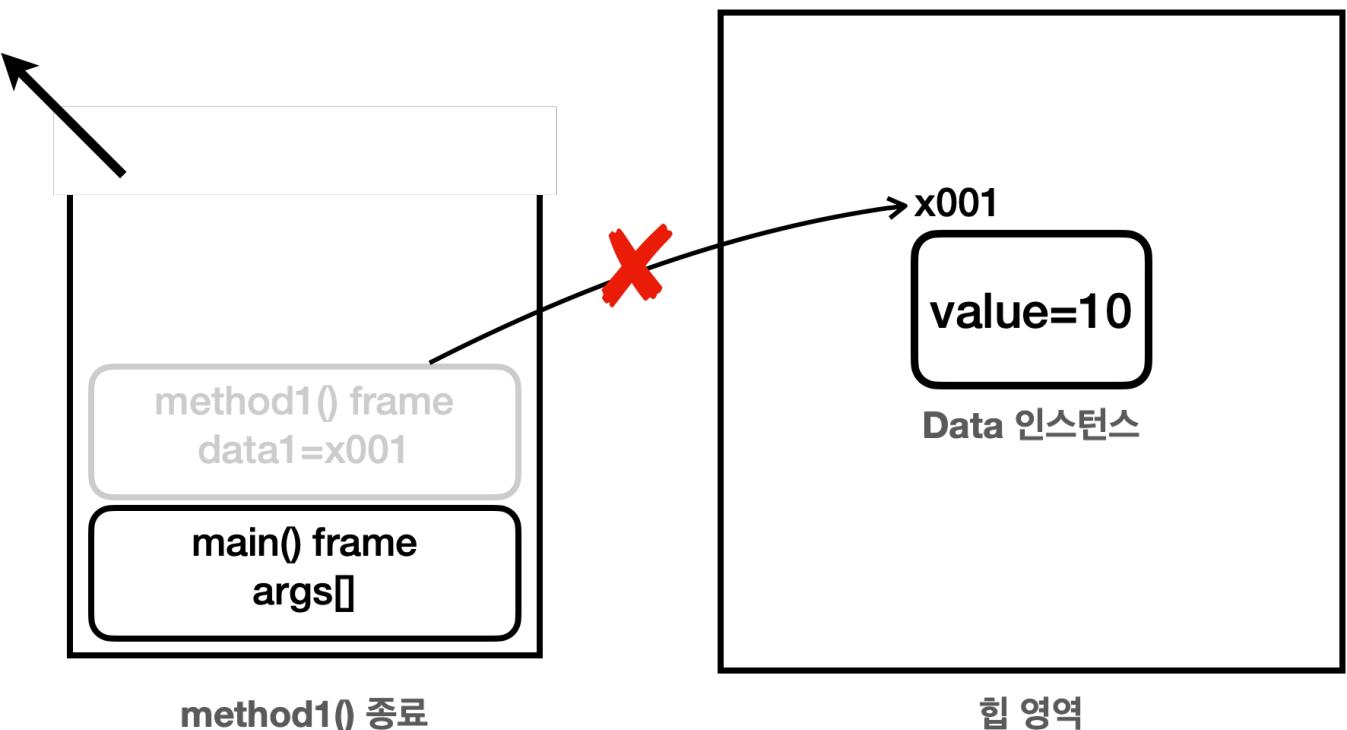
method2() 실행

힙 영역

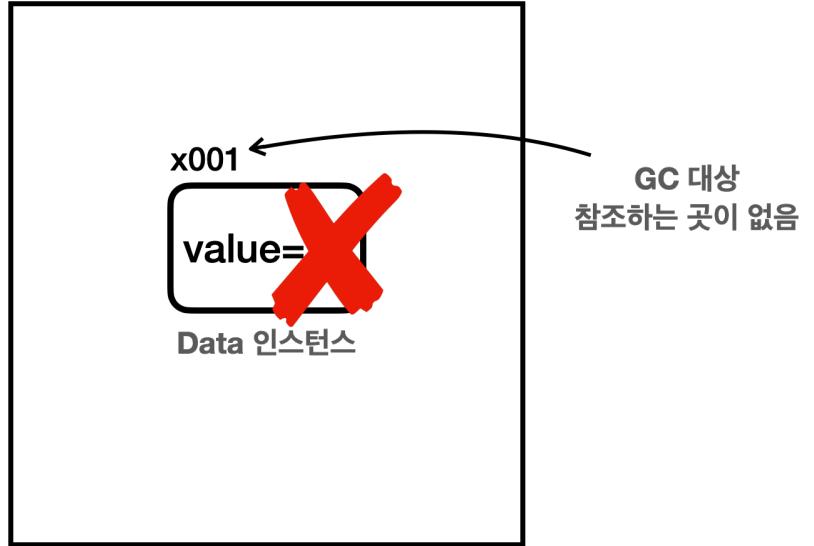
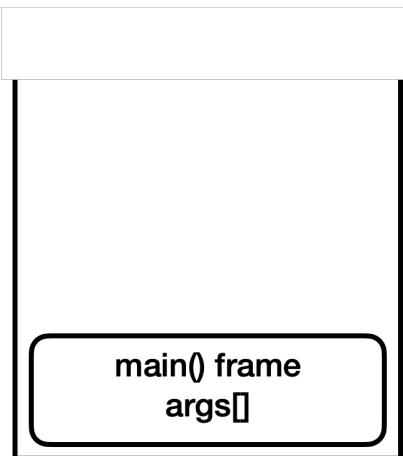
- `method1()`은 `method2()`를 호출하면서 `Data data2` 매개변수에 `x001` 참조값을 넘긴다.
- 이제 `method1()`에 있는 `data1`과 `method2()`에 있는 `data2` 지역 변수(매개변수 포함)는 둘다 같은 `x001` 인스턴스를 참조한다.



- method2() 가 종료된다. method2() 의 스택 프레임이 제거되면서 매개변수 data2 도 함께 제거된다.



- method1() 이 종료된다. method1() 의 스택 프레임이 제거되면서 지역 변수 data1 도 함께 제거된다.



- method1() 이 종료된 직후의 상태를 보자. method1() 의 스택 프레임이 제거되고 지역 변수 data1 도 함께 제거되었다.
- 이제 x001 참조값을 가진 Data 인스턴스를 참조하는 곳이 더는 없다.
- 참조하는 곳이 없으므로 사용되는 곳도 없다. 결과적으로 프로그램에서 더는 사용하지 않는 객체인 것이다. 이런 객체는 메모리만 차지하게 된다.
- GC(가비지 컬렉션)은 이렇게 참조가 모두 사라진 인스턴스를 찾아서 메모리에서 제거한다.

참고: 힙 영역 외부가 아닌, 힙 영역 안에서만 인스턴스끼리 서로 참조하는 경우에도 GC의 대상이 된다.

정리

지역 변수는 스택 영역에, 객체(인스턴스)는 힙 영역에 관리되는 것을 확인했다. 이제 나머지 하나가 남았다. 바로 메서드 영역이다. 메서드 영역이 관리하는 변수도 있다. 이것을 이해하기 위해서는 먼저 static 키워드를 알아야 한다. static 키워드는 메서드 영역과 밀접한 연관이 있다.

static 변수1

이번에는 새로운 키워드인 static 키워드에 대해 학습해보자.

static 키워드는 주로 멤버 변수와 메서드에 사용된다.

먼저 멤버 변수에 static 키워드가 왜 필요한지 이해하기 위해 간단한 예제를 만들어보자.

특정 클래스를 통해서 생성된 객체의 수를 세는 단순한 프로그램이다.

인스턴스 내부 변수에 카운트 저장

먼저 생성할 인스턴스 내부에 카운트를 저장하겠다.

Data1

```
package static1;

public class Data1 {
    public String name;
    public int count;

    public Data1(String name) {
        this.name = name;
        count++;
    }
}
```

생성된 객체의 수를 세어야 한다. 따라서 객체가 생성될 때마다 생성자를 통해 인스턴스의 멤버 변수인 `count` 값을 증가시킨다.

참고로 예제를 단순하게 만들기 위해 필드에 `public`을 사용했다.

DataCountMain1

```
package static1;

public class DataCountMain1 {

    public static void main(String[] args) {
        Data1 data1 = new Data1("A");
        System.out.println("A count=" + data1.count);

        Data1 data2 = new Data1("B");
        System.out.println("B count=" + data2.count);

        Data1 data3 = new Data1("C");
        System.out.println("C count=" + data3.count);
    }
}
```

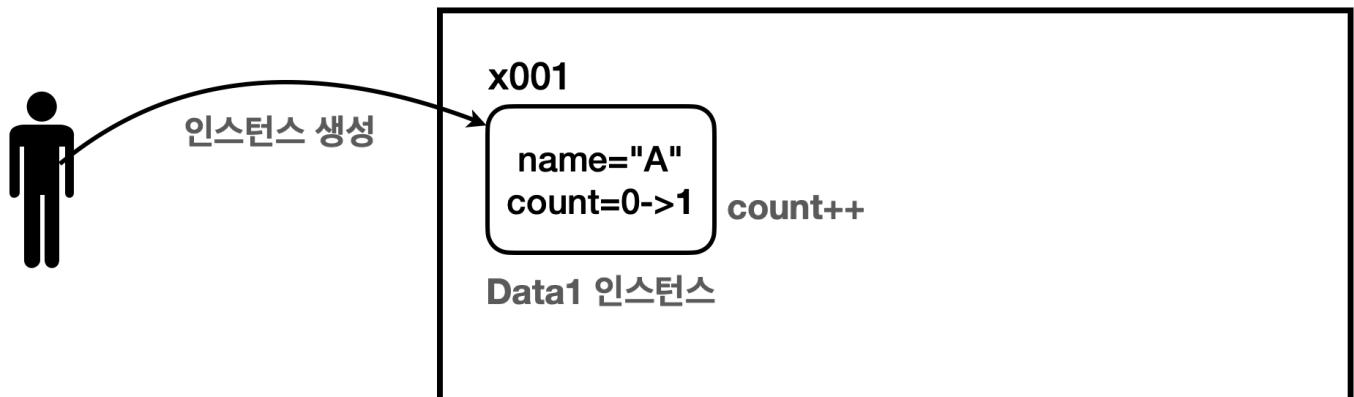
객체를 생성하고 카운트 값을 출력한다.

실행 결과

```
A count=1
B count=1
```

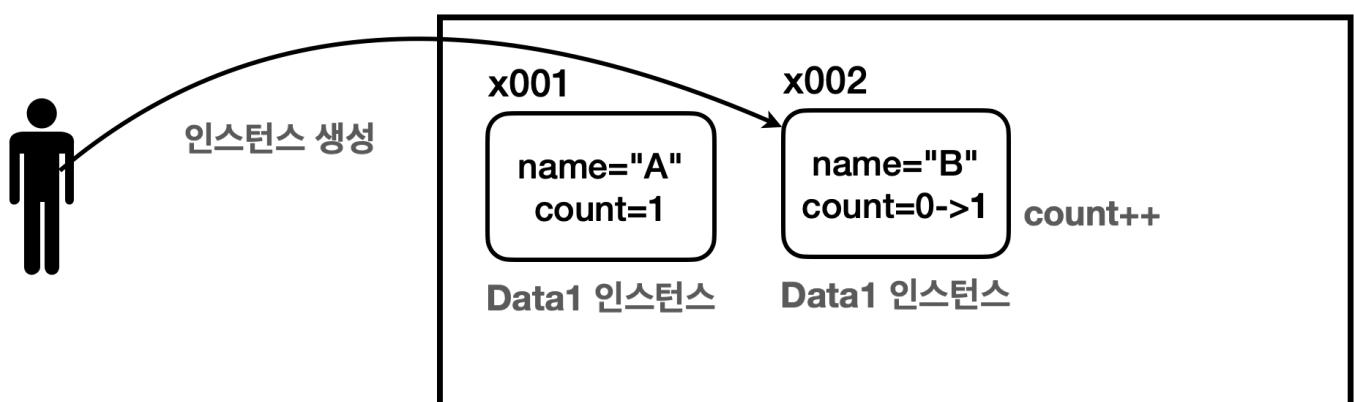
```
C count=1
```

이 프로그램은 당연히 기대한 대로 작동하지 않는다. 객체를 생성할 때마다 `Data1` 인스턴스는 새로 만들어진다. 그리고 인스턴스에 포함된 `count` 변수도 새로 만들어지기 때문이다.



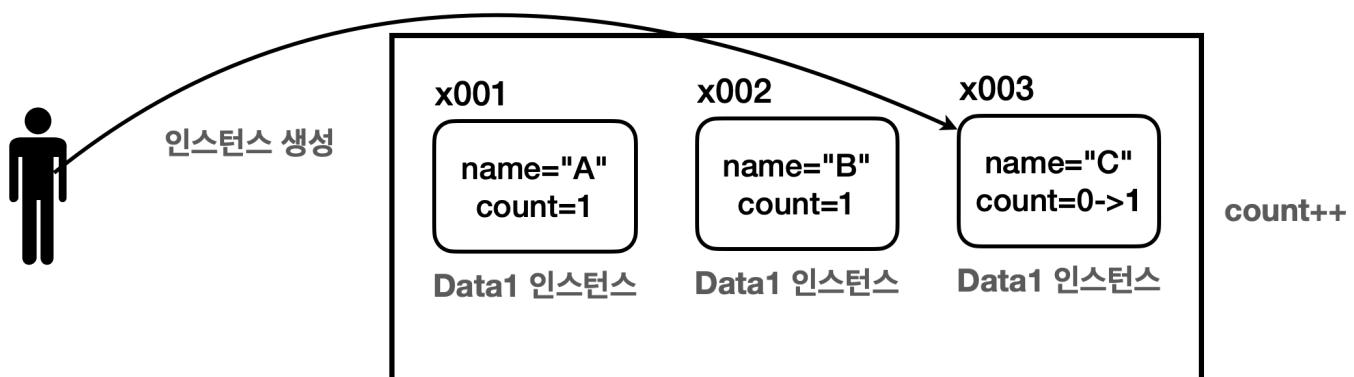
힙 영역

처음 `Data1("A")` 인스턴스를 생성하면 `count` 값은 0으로 초기화 된다. 생성자에서 `count++` 을 호출했으므로 `count` 의 값은 1이 된다.



힙 영역

다음으로 `Data1("B")` 인스턴스를 생성하면 완전 새로운 인스턴스를 생성한다. 이 새로운 인스턴스의 `count` 값은 0으로 초기화 된다. 생성자에서 `count++` 을 호출했으므로 `count` 의 값은 1이 된다.



힙 영역

다음으로 `Data1("C")` 인스턴스를 생성하면 이전 인스턴스는 관계없는 새로운 인스턴스를 생성한다. 이 새로운 인스턴스의 `count` 값은 0으로 초기화 된다. 생성자에서 `count++`을 호출했으므로 `count`의 값은 1이 된다.

인스턴스에 사용되는 멤버 변수 `count` 값은 인스턴스끼리 서로 공유되지 않는다. 따라서 원하는 답을 구할 수 없다. 이 문제를 해결하려면 변수를 서로 공유해야 한다.

외부 인스턴스에 카운트 저장

이번에는 카운트 값을 저장하는 별도의 객체를 만들어보자.

Counter

```
package static1;

public class Counter {
    public int count;
}
```

- 이 객체를 공유해서 필요할 때마다 카운트 값을 증가할 것이다.

Data2

```
package static1;

public class Data2 {
    public String name;

    public Data2(String name, Counter counter) {
        this.name = name;
        counter.count++;
    }
}
```

- 기존 코드를 유지하기 위해 새로운 `Data2` 클래스를 만들었다. 여기에는 `count` 멤버 변수가 없다. 대신에 생성자에서 `Counter` 인스턴스를 추가로 전달 받는다.
- 생성자가 호출되면 `counter` 인스턴스에 있는 `count` 변수의 값을 하나 증가시킨다.

DataCountMain2

```
package static1;

public class DataCountMain2 {
```

```

public static void main(String[] args) {
    Counter counter = new Counter();
    Data2 data1 = new Data2("A", counter);
    System.out.println("A count=" + counter.count);

    Data2 data2 = new Data2("B", counter);
    System.out.println("B count=" + counter.count);

    Data2 data3 = new Data2("C", counter);
    System.out.println("C count=" + counter.count);
}
}

```

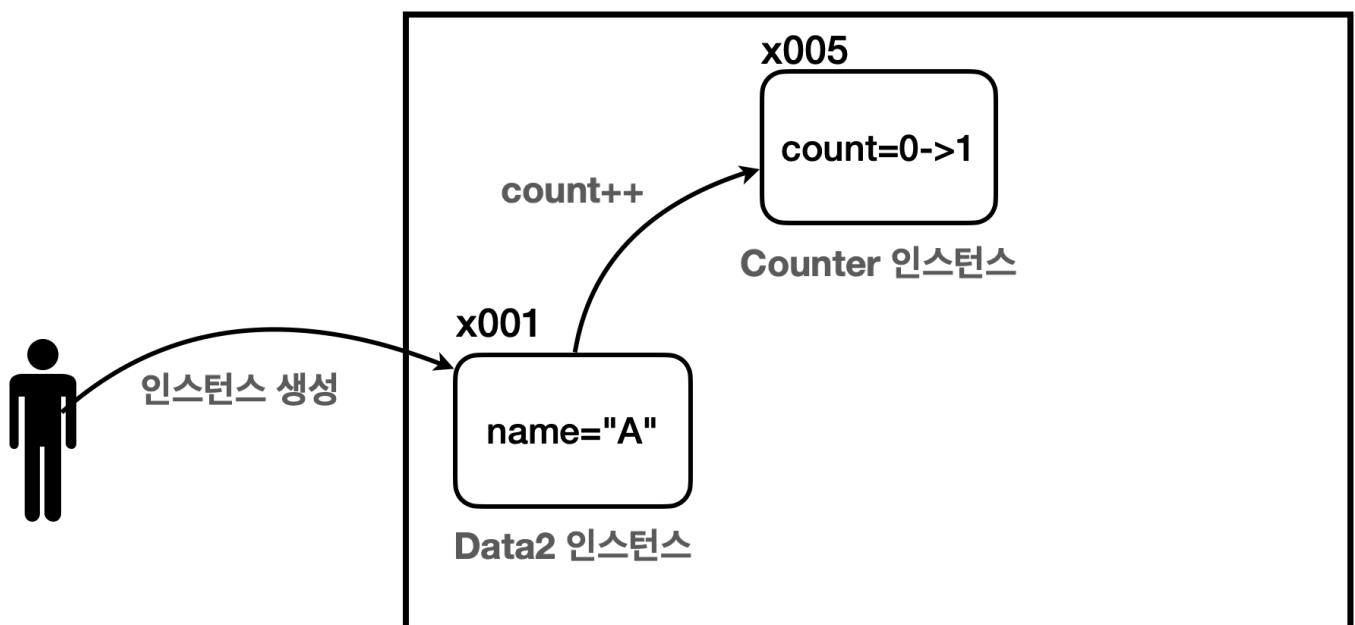
실행 결과

```

A count=1
B count=2
C count=3

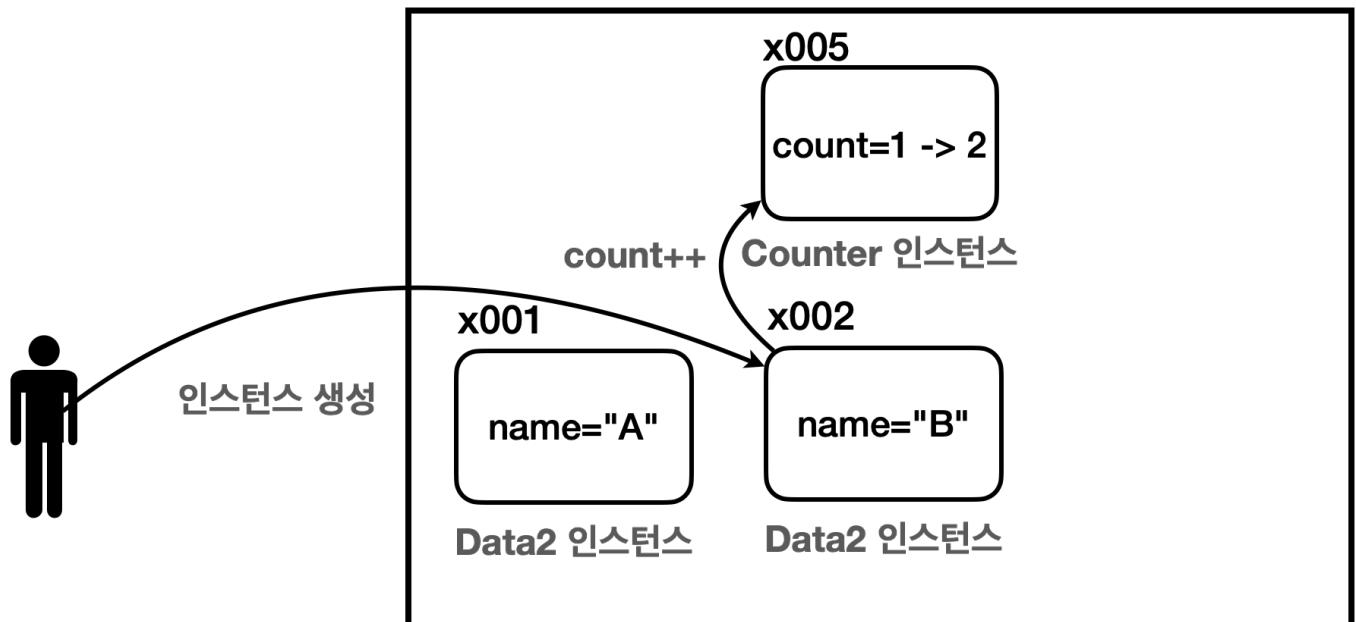
```

Counter 인스턴스를 공용으로 사용한 덕분에 객체를 생성할 때마다 값을 정확하게 증가시킬 수 있다.



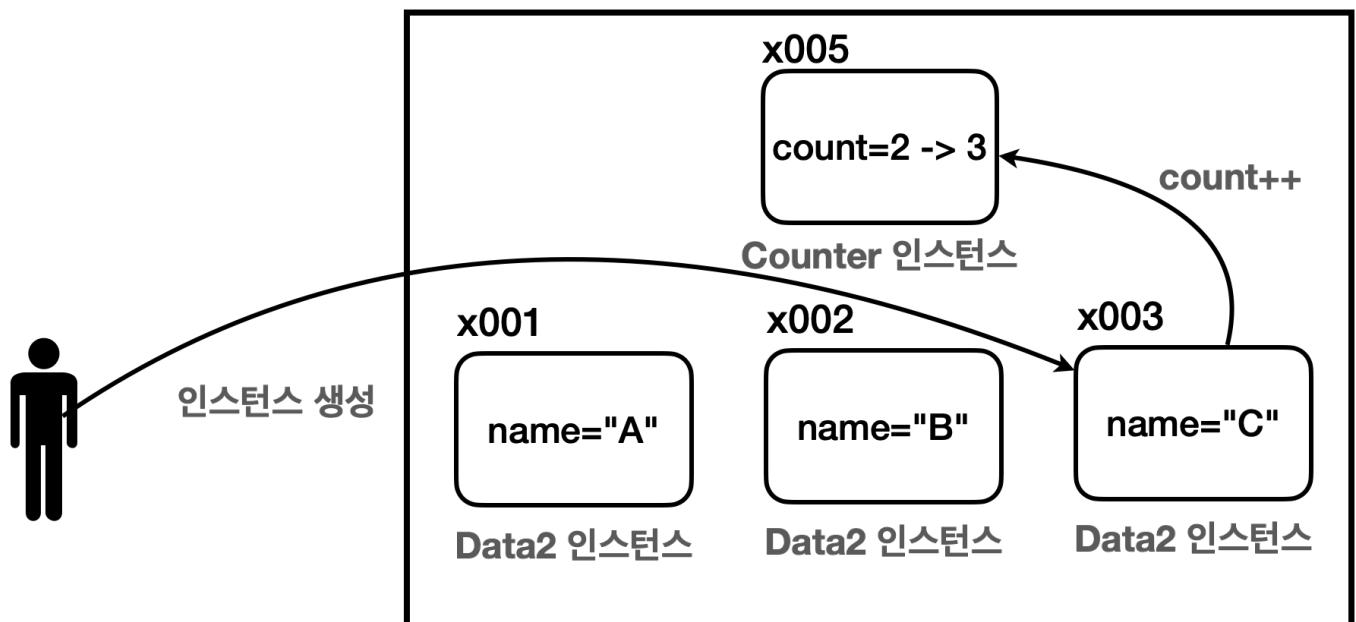
힙 영역

`Data2("A")` 인스턴스를 생성하면 생성자를 통해 `Counter` 인스턴스에 있는 `count` 값을 하나 증가시킨다. `count` 값은 10이 된다.



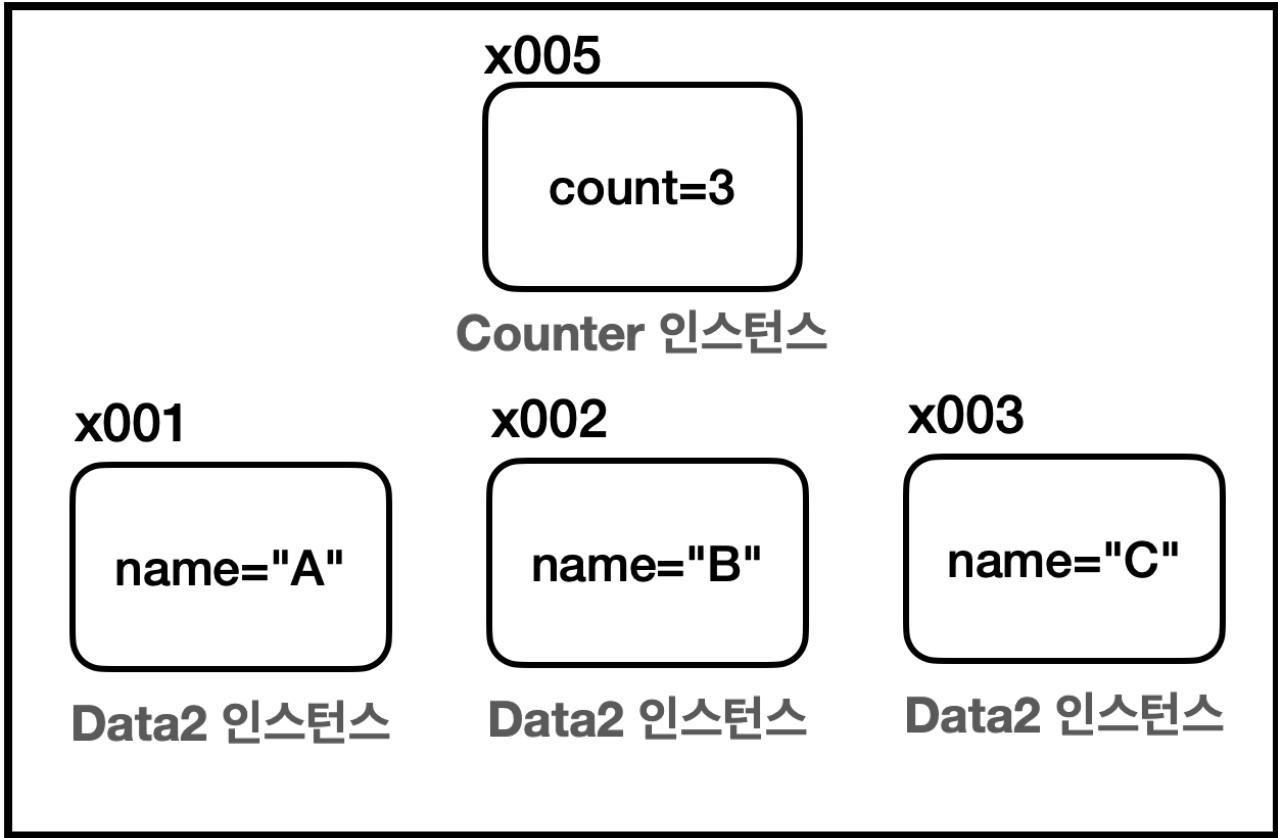
힙 영역

Data2("B") 인스턴스를 생성하면 생성자를 통해 Counter 인스턴스에 있는 count 값을 하나 증가시킨다.
count 값은 2가 된다.



힙 영역

Data2("C") 인스턴스를 생성하면 생성자를 통해 Counter 인스턴스에 있는 count 값을 하나 증가시킨다.
count 값은 3이 된다.



힙 영역

결과적으로 `Data2` 의 인스턴스가 3개 생성되고, `count` 값도 인스턴스 숫자와 같은 3으로 정확하게 측정된다.

그런데 여기에는 약간 불편한 점들이 있다.

- `Data2` 클래스와 관련된 일인데, `Counter`라는 별도의 클래스를 추가로 사용해야 한다.
- 생성자의 매개변수도 추가되고, 생성자가 복잡해진다. 생성자를 호출하는 부분도 복잡해진다.

static 변수2

static 변수 사용

특정 클래스에서 공용으로 함께 사용할 수 있는 변수를 만들 수 있다면 편리할 것이다.

`static` 키워드를 사용하면 공용으로 함께 사용하는 변수를 만들 수 있다.

Data3

```

package static1;

public class Data3 {
  
```

```

public String name;
public static int count; //static

public Data3(String name) {
    this.name = name;
    count++;
}

}

```

- 기존 코드를 유지하기 위해 새로운 클래스 Data3을 만들었다.
- static int count 부분을 보자. 변수 타입(int) 앞에 static 키워드가 붙어있다.
- 이렇게 멤버 변수에 static을 붙이게 되면 static 변수, 정적 변수 또는 클래스 변수라 한다.
- 객체가 생성되면 생성자에서 정적 변수 count의 값을 하나 증가시킨다.

DataCountMain3

```

package static1;

public class DataCountMain3 {

    public static void main(String[] args) {
        Data3 data1 = new Data3("A");
        System.out.println("A count=" + Data3.count);

        Data3 data2 = new Data3("B");
        System.out.println("B count=" + Data3.count);

        Data3 data3 = new Data3("C");
        System.out.println("C count=" + Data3.count);
    }
}

```

코드를 보면 count 정적 변수에 접근하는 방법이 조금 특이한데 Data3.count와 같이 클래스명에 .(dot)을 사용 한다. 마치 클래스에 직접 접근하는 것처럼 느껴진다.

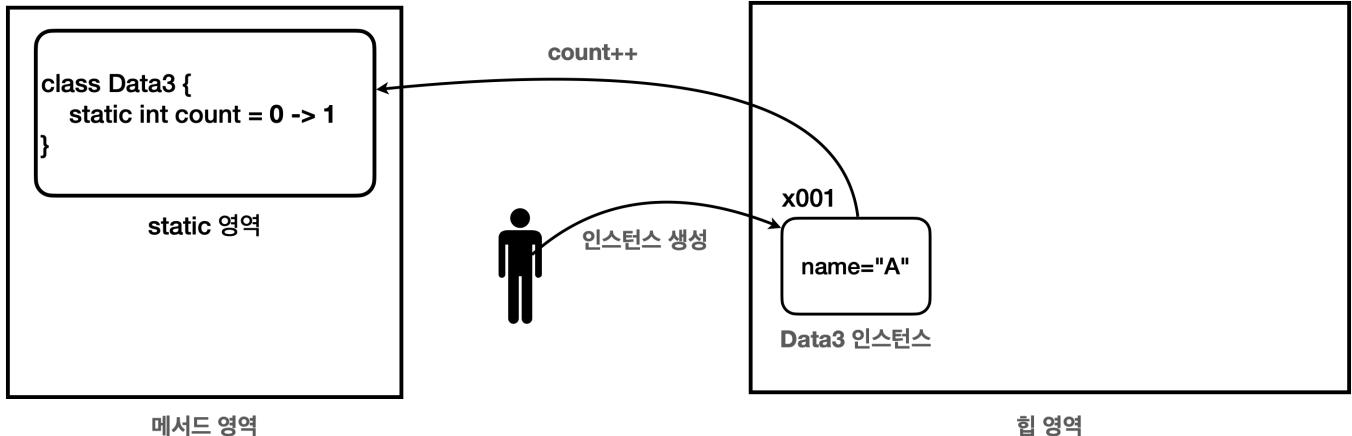
실행 결과

```

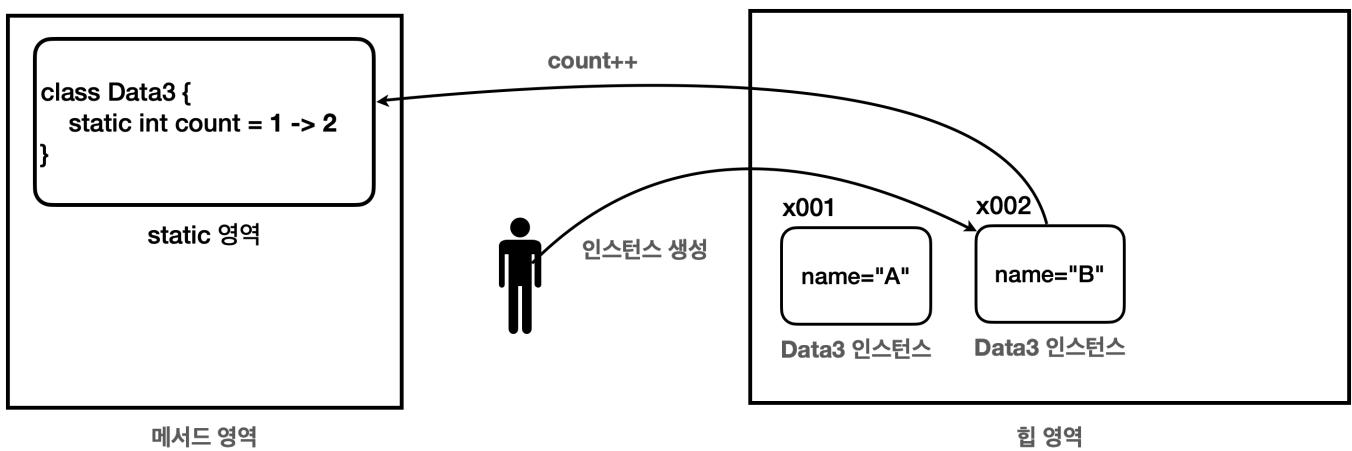
A count=1
B count=2
C count=3

```

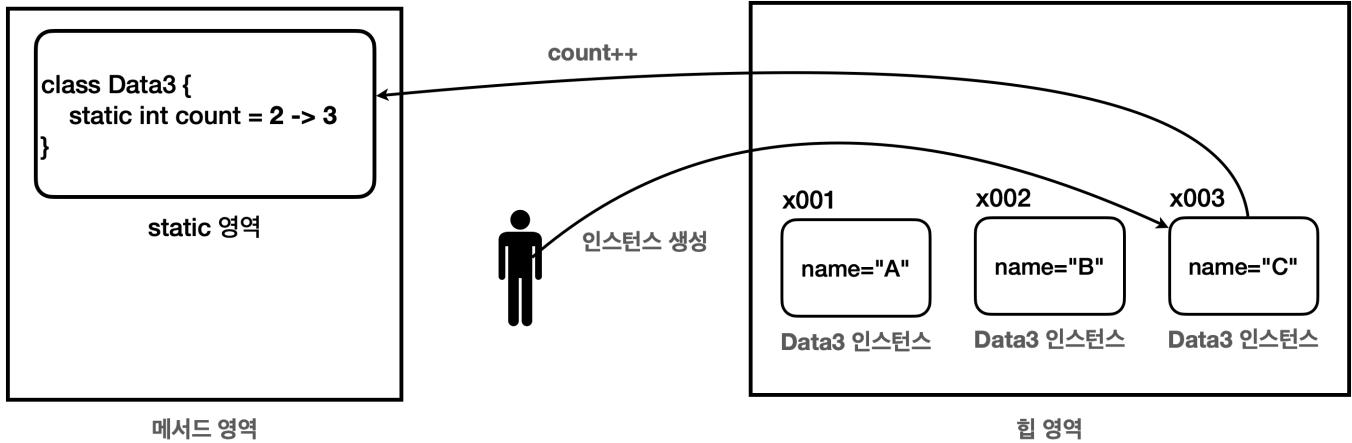
그림으로 알아보자.



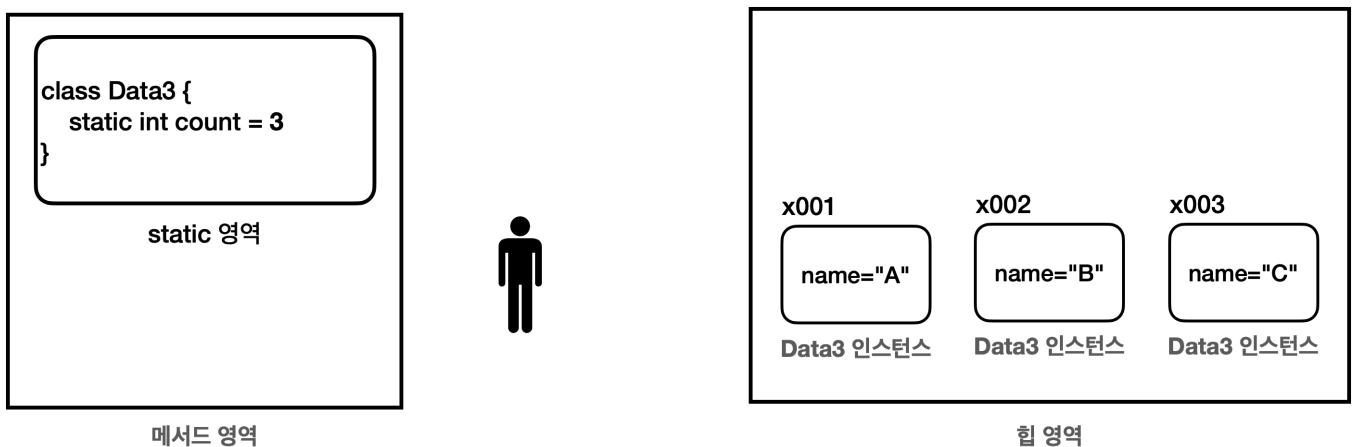
- `static`이 붙은 멤버 변수는 메서드 영역에서 관리한다.
 - `static`이 붙은 멤버 변수 `count`는 인스턴스 영역에 생성되지 않는다. 대신에 메서드 영역에서 이 변수를 관리한다.
- `Data3("A")` 인스턴스를 생성하면 생성자가 호출된다
- 생성자에는 `count++` 코드가 있다. `count`는 `static`이 붙은 정적 변수다. 정적 변수는 인스턴스 영역이 아니라 메서드 영역에서 관리한다. 따라서 이 경우 메서드 영역에 있는 `count`의 값이 하나 증가된다.



- `Data3("B")` 인스턴스를 생성하면 생성자가 호출된다
- `count++` 코드가 있다. `count`는 `static`이 붙은 정적 변수다. 메서드 영역에 있는 `count` 변수의 값이 하나 증가된다.



- `Data3("C")` 인스턴스를 생성하면 생성자가 호출된다
- `count++` 코드가 있다. `count`는 `static`이 붙은 정적 변수다. 메서드 영역에 있는 `count` 변수의 값이 하나 증가된다.



최종적으로 메서드 영역에 있는 `count` 변수의 값은 3이 된다.

`static`이 붙은 정적 변수에 접근하려면 `Data3.count`와 같이 클래스명 + `.` (dot) + 변수명으로 접근하면 된다.

참고로 `Data3`의 생성자와 같이 자신의 클래스에 있는 정적 변수라면 클래스명을 생략할 수 있다.

`static` 변수를 사용한 덕분에 공용 변수를 사용해서 편리하게 문제를 해결할 수 있었다.

정리

`static` 변수는 쉽게 이야기해서 클래스인 봉어빵 틀이 특별히 관리하는 변수이다. 봉어빵 틀은 1개이므로 클래스 변수도 하나만 존재한다. 반면에 인스턴스 변수는 봉어빵인 인스턴스의 수 만큼 존재한다.

static 변수3

이번에는 `static` 변수를 정리해보자.

용어 정리

```
public class Data3 {  
    public String name;  
    public static int count; //static  
}
```

예제 코드에서 `name`, `count`는 둘다 멤버 변수이다.

멤버 변수(필드)는 `static`이 붙은 것과 아닌 것에 따라 다음과 같이 분류할 수 있다.

멤버 변수(필드)의 종류

- **인스턴스 변수:** `static`이 붙지 않은 멤버 변수, 예) `name`
 - `static`이 붙지 않은 멤버 변수는 인스턴스를 생성해야 사용할 수 있고, 인스턴스에 소속되어 있다. 따라서 인스턴스 변수라 한다.
 - 인스턴스 변수는 인스턴스를 만들 때마다 새로 만들어진다.
- **클래스 변수:** `static`이 붙은 멤버 변수, 예) `count`
 - 클래스 변수, 정적 변수, `static` 변수등으로 부른다. **용어를 모두 사용하니 주의하자**
 - `static`이 붙은 멤버 변수는 인스턴스와 무관하게 클래스에 바로 접근해서 사용할 수 있고, 클래스 자체에 소속되어 있다. 따라서 클래스 변수라 한다.
 - 클래스 변수는 자바 프로그램을 시작할 때 딱 1개가 만들어진다. 인스턴스와는 다르게 보통 여走路에서 공유하는 목적으로 사용된다.

변수와 생명주기

- **지역 변수(매개변수 포함):** 지역 변수는 스택 영역에 있는 스택 프레임 안에 보관된다. 메서드가 종료되면 스택 프레임도 제거 되는데 이때 해당 스택 프레임에 포함된 지역 변수도 함께 제거된다. 따라서 지역 변수는 생존 주기가 짧다.
- **인스턴스 변수:** 인스턴스에 있는 멤버 변수를 인스턴스 변수라 한다. 인스턴스 변수는 힙 영역을 사용한다. 힙 영역은 GC(가비지 컬렉션)가 발생하기 전까지는 생존하기 때문에 보통 지역 변수보다 생존 주기가 길다.
- **클래스 변수:** 클래스 변수는 메서드 영역의 `static` 영역에 보관되는 변수이다. 메서드 영역은 프로그램 전체에서 사용하는 공용 공간이다. 클래스 변수는 해당 클래스가 JVM에 로딩 되는 순간 생성된다. 그리고 JVM이 종료될 때 까지 생명주기가 어어진다. 따라서 가장 긴 생명주기를 가진다.

`static`이 정적이라는 이유는 바로 여기에 있다. 힙 영역에 생성되는 인스턴스 변수는 동적으로 생성되고, 제거된다.

반면에 `static`인 정적 변수는 거의 프로그램 실행 시점에 딱 만들어지고, 프로그램 종료 시점에 제거된다. 정적 변수는 이름 그대로 정적이다.

정적 변수 접근 법

`static` 변수는 클래스를 통해 바로 접근할 수도 있고, 인스턴스를 통해서도 접근할 수 있다.

`DataCountMain3` 마지막 코드에 다음 부분을 추가하고 실행해보자.

`DataCountMain3` - 추가

```
//추가  
//인스턴스를 통한 접근  
Data3 data4 = new Data3("D");  
System.out.println(data4.count);  
  
//클래스를 통한 접근  
System.out.println(Data3.count);
```

실행 결과 - 추가된 부분

```
4  
4
```

둘의 차이는 없다. 둘다 결과적으로 정적 변수에 접근한다.

인스턴스를 통한 접근 `data4.count`

정적 변수의 경우 인스턴스를 통한 접근은 추천하지 않는다. 왜냐하면 코드를 읽을 때 마치 인스턴스 변수에 접근하는 것처럼 오해할 수 있기 때문이다.

클래스를 통한 접근 `Data3.count`

정적 변수는 클래스에서 공용으로 관리하기 때문에 클래스를 통해서 접근하는 것이 더 명확하다. 따라서 정적 변수에 접근할 때는 클래스를 통해서 접근하자.

static 메서드1

이번에는 `static` 이 붙은 메서드에 대해 알아보자.

이해를 돋기 위해 간단한 예제를 만들어보자.

특정 문자열을 꾸며주는 간단한 기능을 만들어보자.

예를 들어서 "hello"라는 문자열 앞 뒤에 *을 붙여서 "*hello*"와 같이 꾸며주는 기능이다.

인스턴스 메서드

먼저 지금까지 학습한 방식을 통해 해당 기능을 개발해보자.

```
package static2;

public class DecoUtil1 {

    public String deco(String str) {
        String result = "*" + str + "*";
        return result;
    }
}
```

`deco()`는 문자열을 꾸미는 기능을 제공한다. 문자열이 들어오면 앞 뒤에 *을 붙여서 반환한다.

```
package static2;

public class DecoMain1 {

    public static void main(String[] args) {
        String s = "hello java";
        DecoUtil1 utils = new DecoUtil1();
        String deco = utils.deco(s);

        System.out.println("before: " + s);
        System.out.println("after: " + deco);
    }
}
```

실행 결과

```
before: hello java
after: *hello java*
```

앞서 개발한 `deco()` 메서드를 호출하기 위해서는 `DecoUtil1`의 인스턴스를 먼저 생성해야 한다. 그런데 `deco()`라는 기능은 멤버 변수도 없고, 단순히 기능만 제공할 뿐이다. 인스턴스가 필요한 이유는 멤버 변수(인스턴스 변수) 등을 사용하는 목적이 큰데, 이 메서드는 사용하는 인스턴스 변수도 없고 단순히 기능만 제공한다.

static 메서드

먼저 예제를 만들어서 실행해보자.

```
package static2;

public class DecoUtil2 {

    public static String deco(String str) {
        String result = "*" + str + "*";
        return result;
    }

}
```

DecoUtil2는 앞선 예제와 비슷한데, 메서드 앞에 `static`이 붙어있다. 이 부분에 주의하자. 이렇게 하면 정적 메서드를 만들 수 있다. 그리고 이 정적 메서드는 정적 변수처럼 인스턴스 생성 없이 클래스 명을 통해서 바로 호출할 수 있다.

```
package static2;

public class DecoMain2 {

    public static void main(String[] args) {
        String s = "hello java";
        String deco = DecoUtil2.deco(s);

        System.out.println("before: " + s);
        System.out.println("after: " + deco);
    }
}
```

실행 결과

```
before: hello java
after: *hello java*
```

DecoUtil2.deco(s) 코드를 보자.

`static`이 붙은 정적 메서드는 객체 생성 없이 클래스명 + `.` (dot) + 메서드 명으로 바로 호출할 수 있다. 정적 메서드 덕분에 불필요한 객체 생성 없이 편리하게 메서드를 사용했다.

클래스 메서드

메서드 앞에도 `static`을 붙일 수 있다. 이것을 정적 메서드 또는 클래스 메서드라 한다. 정적 메서드라는 용어는 `static`이 정적이라는 뜻이기 때문이고, 클래스 메서드라는 용어는 인스턴스 생성 없이 마치 클래스에 있는 메서드를 바로 호출하는 것처럼 느껴지기 때문이다.

인스턴스 메서드

`static`이 붙지 않은 메서드는 인스턴스를 생성해야 호출할 수 있다. 이것을 인스턴스 메서드라 한다.

static 메서드2

정적 메서드는 객체 생성없이 클래스에 있는 메서드를 바로 호출할 수 있다는 장점이 있다.
하지만 정적 메서드는 언제나 사용할 수 있는 것이 아니다.

정적 메서드 사용법

- 정적 메서드는 `static`만 사용할 수 있다.
 - 클래스 내부의 기능을 사용할 때, 정적 메서드는 `static`이 붙은 정적 메서드나 정적 변수만 사용할 수 있다.
 - 클래스 내부의 기능을 사용할 때, 정적 메서드는 인스턴스 변수나, 인스턴스 메서드를 사용할 수 없다.
- 반대로 모든 곳에서 `static`을 호출할 수 있다.
 - 정적 메서드는 공용 기능이다. 따라서 접근 제어자만 허락한다면 클래스를 통해 모든 곳에서 `static`을 호출할 수 있다.

예제를 통해 정적 메서드의 사용법을 확인해보자.

DecoData

```
package static2;

public class DecoData {

    private int instanceValue;
    private static int staticValue;

    public static void staticCall() {
        //instanceValue++; //인스턴스 변수 접근, compile error
        //instanceMethod(); //인스턴스 메서드 접근, compile error
    }
}
```

```

        staticValue++; //정적 변수 접근
        staticMethod(); //정적 메서드 접근
    }

    public void instanceCall() {
        instanceValue++; //인스턴스 변수 접근
        instanceMethod(); //인스턴스 메서드 접근

        staticValue++; //정적 변수 접근
        staticMethod(); //정적 메서드 접근
    }

    private void instanceMethod() {
        System.out.println("instanceValue=" + instanceValue);
    }
    private static void staticMethod() {
        System.out.println("staticValue=" + staticValue);
    }
}

```

이번 예제에서는 접근 제어자를 적극 활용해서 필드를 포함한 외부에서 직접 필요하지 않은 기능은 모두 막아두었다.

- `instanceValue`는 인스턴스 변수이다.
- `staticValue`는 정적 변수(클래스 변수)이다.
- `instanceMethod()`는 인스턴스 메서드이다.
- `staticMethod()`는 정적 메서드(클래스 메서드)이다.

`staticCall()` 메서드를 보자.

이 메서드는 정적 메서드이다. 따라서 `static`만 사용할 수 있다. 정적 변수, 정적 메서드에는 접근할 수 있지만, `static`이 없는 인스턴스 변수나 인스턴스 메서드에 접근하면 **컴파일 오류가 발생한다**.

코드를 보면 `staticCall()` → `staticMethod()`로 `static`에서 `static`을 호출하는 것을 확인할 수 있다.

`instanceCall()` 메서드를 보자.

이 메서드는 인스턴스 메서드이다. 모든 곳에서 공용인 `static`을 호출할 수 있다. 따라서 정적 변수, 정적 메서드에 접근할 수 있다. 물론 인스턴스 변수, 인스턴스 메서드에도 접근할 수 있다.

DecoDataMain

```

package static2;

public class DecoDataMain {

```

```

public static void main(String[] args) {
    System.out.println("1.정적 호출");
    DecoData.staticCall();

    System.out.println("2.인스턴스 호출1");
    DecoData data1 = new DecoData();
    data1.instanceCall();

    System.out.println("3.인스턴스 호출2");
    DecoData data2 = new DecoData();
    data2.instanceCall();
}
}

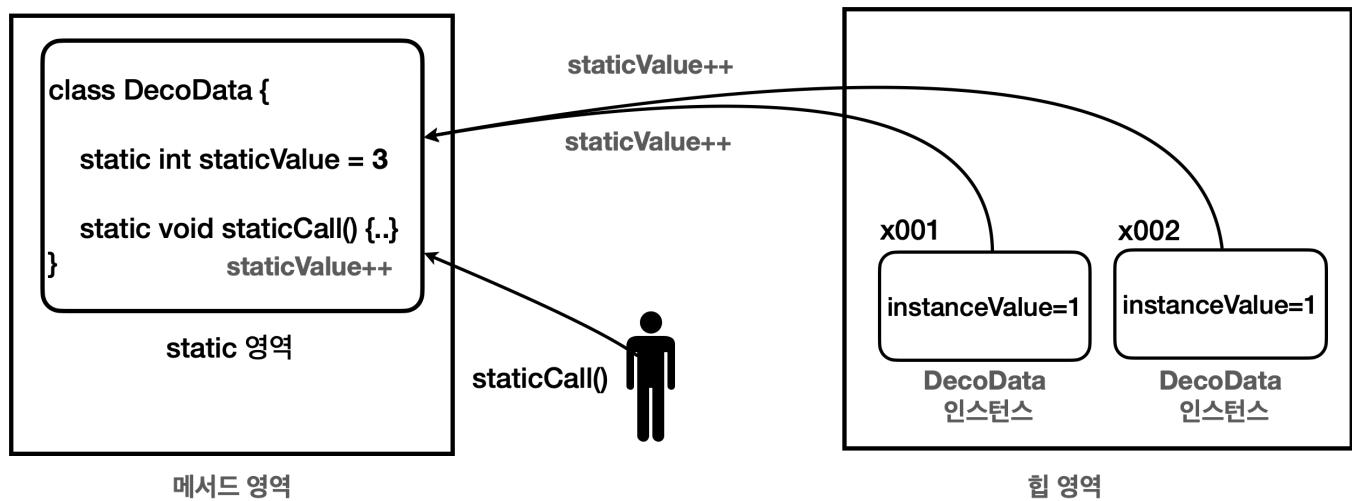
```

실행 결과

```

1.정적 호출
staticValue=1
2.인스턴스 호출1
instanceValue=1
staticValue=2
3.인스턴스 호출2
instanceValue=1
staticValue=3

```



정적 메서드가 인스턴스의 기능을 사용할 수 없는 이유

정적 메서드는 클래스의 이름을 통해 바로 호출할 수 있다. 그래서 인스턴스처럼 참조값의 개념이 없다.

특정 인스턴스의 기능을 사용하려면 참조값을 알아야 하는데, 정적 메서드는 참조값 없이 호출한다. 따라서 정적 메서드 내부에서 인스턴스 변수나 인스턴스 메서드를 사용할 수 없다.

물론 당연한 이야기지만 다음과 같이 객체의 참조값을 직접 매개변수로 전달하면 정적 메서드도 인스턴스의 변수나 메서드를 호출할 수 있다.

```
public static void staticCall(DecoData data) {  
    data.instanceValue++;  
    data.instanceMethod();  
}
```

static 메서드3

용어 정리

멤버 메서드의 종류

- **인스턴스 메서드**: static이 붙지 않은 멤버 메서드
- **클래스 메서드**: static이 붙은 멤버 메서드
 - 클래스 메서드, 정적 메서드, static 메서드등으로 부른다.

static이 붙지 않은 멤버 메서드는 인스턴스를 생성해야 사용할 수 있고, 인스턴스에 소속되어 있다. 따라서 인스턴스 메서드라 한다. static이 붙은 멤버 메서드는 인스턴스와 무관하게 클래스에 바로 접근해서 사용할 수 있고, 클래스 자체에 소속되어 있다. 따라서 클래스 메서드라 한다.

참고로 방금 설명한 내용은 멤버 변수에도 똑같이 적용된다.

정적 메서드 활용

정적 메서드는 객체 생성이 필요 없이 메서드의 호출만으로 필요한 기능을 수행할 때 주로 사용한다.

예를 들어 간단한 메서드 하나로 끝나는 유틸리티성 메서드에 자주 사용한다. 수학의 여러가지 기능을 담은 클래스를 만들 수 있는데, 이 경우 인스턴스 변수 없이 입력한 값을 계산하고 반환하는 것이 대부분이다. 이럴 때 정적 메서드를 사용해서 유틸리티성 메서드를 만들면 좋다.

정적 메서드 접근 법

static 메서드는 static 변수와 마찬가지로 클래스를 통해 바로 접근할 수 있고, 인스턴스를 통해서도 접근할 수 있다.

DataCountMain3 - 추가

```
//추가  
//인스턴스를 통한 접근  
DecoData data3 = new DecoData();  
data3.staticCall();  
  
//클래스를 통한 접근  
DecoData.staticCall();
```

실행 결과 - 추가된 부분

```
staticValue=4  
staticValue=5
```

둘의 차이는 없다. 둘다 결과적으로 정적 메서드에 접근한다.

인스턴스를 통한 접근 `data3.staticCall()`

정적 메서드의 경우 인스턴스를 통한 접근은 추천하지 않는다. 왜냐하면 코드를 읽을 때 마치 인스턴스 메서드에 접근하는 것처럼 오해할 수 있기 때문이다.

클래스를 통한 접근 `DecoData.staticCall()`

정적 메서드는 클래스에서 공용으로 관리하기 때문에 클래스를 통해서 접근하는 것이 더 명확하다. 따라서 정적 메서드에 접근할 때는 클래스를 통해서 접근하자.

static import

정적 메서드를 사용할 때 해당 메서드를 다음과 같이 자주 호출해야 한다면 `static import` 기능을 고려하자.

```
DecoData.staticCall();  
DecoData.staticCall();  
DecoData.staticCall();
```

이 기능을 사용하면 다음과 같이 클래스 명을 생략하고 메서드를 호출할 수 있다.

```
staticCall();  
staticCall();  
staticCall();
```

DecoDataMain - static import 적용

```
package static2;  
  
//import static static2.DecoData.staticCall;  
import static static2.DecoData.*;
```

```
public class DecoDataMain {  
  
    public static void main(String[] args) {  
        System.out.println("1.정적 호출");  
        staticCall(); //클래스 명 생략 가능  
  
        ...  
    }  
}
```

특정 클래스의 정적 메서드 하나만 적용하려면 다음과 같이 생략할 메서드 명을 적어주면 된다.

```
import static static2.DecoData.staticCall;
```

특정 클래스의 모든 정적 메서드에 적용하려면 다음과 같이 * 을 사용하면 된다.

```
import static static2.DecoData.*;
```

참고로 import static 은 정적 메서드 뿐만 아니라 정적 변수에도 사용할 수 있다.

main() 메서드는 정적 메서드

인스턴스 생성 없이 실행하는 가장 대표적인 메서드가 바로 main() 메서드이다.

main() 메서드는 프로그램을 시작하는 시작점이 되는데, 생각해보면 객체를 생성하지 않아도 main() 메서드가 작동했다. 이것은 main() 메서드가 static 이기 때문이다.

정적 메서드는 정적 메서드만 호출할 수 있다. 따라서 정적 메서드인 main() 이 호출하는 메서드에는 정적 메서드를 사용했다.

물론 더 정확히 말하자면 정적 메서드는 같은 클래스 내부에서 정적 메서드만 호출할 수 있다. 따라서 정적 메서드인 main() 메서드가 같은 클래스에서 호출하는 메서드도 정적 메서드로 선언해서 사용했다.

main() 메서드와 static 메서드 호출 예

```
public class ValueDataMain {  
  
    public static void main(String[] args) {  
        ValueData valueData = new ValueData();  
        add(valueData);  
    }  
  
    static void add(ValueData valueData) {  
        valueData.value++;  
        System.out.println("숫자 증가 value=" + valueData.value);  
    }  
}
```

```
    }  
}
```

문제와 풀이

문제1: 구매한 자동차 수

다음 코드를 참고해서 생성한 차량 수를 출력하는 프로그램을 작성하자.

Car 클래스를 작성하자.

```
package static2.ex;  
  
public class CarMain {  
  
    public static void main(String[] args) {  
        Car car1 = new Car("K3");  
        Car car2 = new Car("G80");  
        Car car3 = new Car("Model Y");  
  
        Car.showTotalCars(); //구매한 차량 수를 출력하는 static 메서드  
    }  
}
```

실행 결과

```
차량 구입, 이름: K3  
차량 구입, 이름: G80  
차량 구입, 이름: Model Y  
구매한 차량 수: 3
```

정답

```
package static2.ex;  
  
public class Car {  
  
    private static int totalCars;  
    private String name;
```

```

public Car(String name) {
    System.out.println("차량 구입, 이름: " + name);
    this.name = name;
    totalCars++;
}

public static void showTotalCars() {
    System.out.println("구매한 차량 수: " + totalCars);
}

}

```

문제2: 수학 유틸리티 클래스

다음 기능을 제공하는 배열용 수학 유틸리티 클래스(`MathArrayUtils`)를 만드세요.

- `sum(int[] array)`: 배열의 모든 요소를 더하여 합계를 반환합니다.
- `average(int[] array)`: 배열의 모든 요소의 평균값을 계산합니다.
- `min(int[] array)`: 배열에서 최소값을 찾습니다.
- `max(int[] array)`: 배열에서 최대값을 찾습니다.

요구사항

- `MathArrayUtils`은 객체를 생성하지 않고 사용해야 합니다. 누군가 실수로 `MathArrayUtils`의 인스턴스를 생성하지 못하게 막으세요.
- 실행 코드에 `static import`를 사용해도 됩니다.

실행 코드와 실행 결과를 참고하세요.

실행 코드

```

package static2.ex;

public class MathArrayUtilsMain {

    public static void main(String[] args) {
        int[] values = {1, 2, 3, 4, 5};
        System.out.println("sum=" + MathArrayUtils.sum(values));
        System.out.println("average=" + MathArrayUtils.average(values));
        System.out.println("min=" + MathArrayUtils.min(values));
        System.out.println("max=" + MathArrayUtils.max(values));
    }
}

```

```
    }  
}
```

실행 결과

```
sum=15  
average=3.0  
min=1  
max=5
```

정답

```
package static2.ex;  
  
public class MathArrayUtils {  
  
    private MathArrayUtils() {  
        //private 인스턴스 생성을 막는다.  
    }  
  
    public static int sum(int[] values) {  
        int total = 0;  
        for (int value : values) {  
            total += value;  
        }  
        return total;  
    }  
  
    public static double average(int[] values) {  
        return (double) sum(values) / values.length;  
    }  
  
    public static int min(int[] values) {  
        int minValue = values[0];  
        for (int value : values) {  
            if (value < minValue) {  
                minValue = value;  
            }  
        }  
        return minValue;  
    }  
  
    public static int max(int[] values) {
```

```
int maxValue = values[0];
for (int value : values) {
    if (value > maxValue) {
        maxValue = value;
    }
}
return maxValue;
}
```

정리

8. final

#1.인강/0.자바/2.자바-기본

- /final 변수와 상수1
- /final 변수와 상수2
- /final 변수와 참조
- /정리

final 변수와 상수1

`final` 키워드는 이름 그대로 끝! 이라는 뜻이다.

변수에 `final` 키워드가 붙으면 더는 값을 변경할 수 없다.

참고로 `final`은 `class`, `method`를 포함한 여러 곳에 붙을 수 있다. 지금은 변수에 붙는 `final` 키워드를 알아보자. 나머지는 `final`의 사용법은 상속을 설명한 이후에 설명한다.

final - 지역 변수

```
package final1;

public class FinalLocalMain {

    public static void main(String[] args) {
        //final 지역 변수1
        final int data1;
        data1 = 10; //최초 한번만 할당 가능
        //data1 = 20; //컴파일 오류

        //final 지역 변수2
        final int data2 = 10;
        //data2 = 20; //컴파일 오류

        method(10);
    }

    //final 매개변수
    static void method(final int parameter) {
        //parameter = 20; 컴파일 오류
    }
}
```

```
}
```

- `final`을 지역 변수에 설정할 경우 최초 한번만 할당할 수 있다. 이후에 변수의 값을 변경하려면 컴파일 오류가 발생한다.
- `final`을 지역 변수 선언시 바로 초기화 한 경우 이미 값이 할당되었기 때문에 값을 할당할 수 없다.
- 매개변수에 `final`이 붙으면 메서드 내부에서 매개변수의 값을 변경할 수 없다. 따라서 메서드 호출 시점에 사용된 값이 끝까지 사용된다.

final - 필드(멤버 변수)

```
package final1;

//final 필드 - 생성자 초기화
public class ConstructInit {
    final int value;

    public ConstructInit(int value) {
        this.value = value;
    }
}
```

- `final`을 필드에 사용할 경우 해당 필드는 생성자를 통해서 한번만 초기화 될 수 있다.

```
package final1;

//final 필드 - 필드 초기화
public class FieldInit {
    static final int CONST_VALUE = 10;
    final int value = 10;
}
```

- `final` 필드를 필드에서 초기화하면 이미 값이 설정되었기 때문에 생성자를 통해서도 초기화 할 수 없다. `value` 필드를 참고하자.
- 코드에서 보는 것처럼 `static` 변수에도 `final`을 선언할 수 있다.
 - `CONST_VALUE`로 변수 작성 방법이 대문자를 사용했는데, 이 부분은 바로 뒤에 상수에서 설명한다.

```
package final1;

public class FinalFieldMain {

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

```

//final 필드 - 생성자 초기화
System.out.println("생성자 초기화");
ConstructInit constructInit1 = new ConstructInit(10);
ConstructInit constructInit2 = new ConstructInit(20);
System.out.println(constructInit1.value);
System.out.println(constructInit2.value);

//final 필드 - 필드 초기화
System.out.println("필드 초기화");
FieldInit fieldInit1 = new FieldInit();
FieldInit fieldInit2 = new FieldInit();
FieldInit fieldInit3 = new FieldInit();
System.out.println(fieldInit1.value);
System.out.println(fieldInit2.value);
System.out.println(fieldInit3.value);

//상수
System.out.println("상수");
System.out.println(FieldInit.CONST_VALUE);
}

}

```

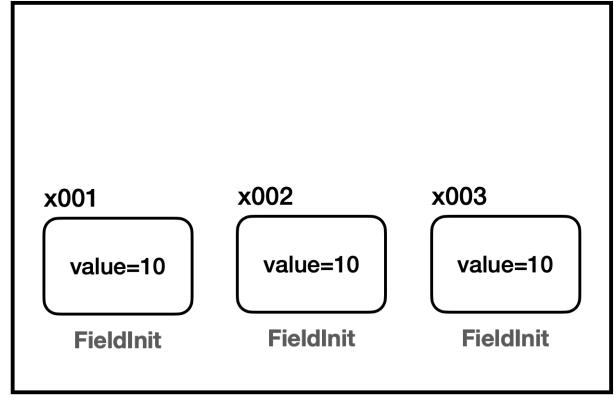
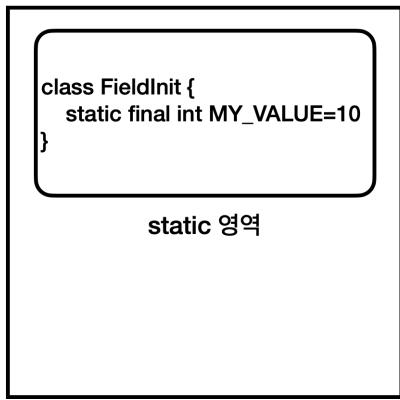
실행 결과

```

생성자 초기화
10
20
필드 초기화
10
10
10
상수
10

```

ConstructInit과 같이 생성자를 사용해서 final 필드를 초기화 하는 경우, 각 인스턴스마다 final 필드에 다른 값을 할당할 수 있다. 물론 final을 사용했기 때문에 생성 이후에 이 값을 변경하는 것은 불가능하다.



메서드 영역

힙 영역

- FieldInit과 같이 `final` 필드를 필드에서 초기화 하는 경우, 모든 인스턴스가 다음 오른쪽 그림과 같이 같은 값을 가진다.
- 여기서는 FieldInit 인스턴스의 모든 `value` 값은 10이 된다.
- 왜냐하면 생성자 초기화와 다르게 필드 초기화는 필드의 코드에 해당 값이 미리 정해져있기 때문이다.
- 모든 인스턴스가 같은 값을 사용하기 때문에 결과적으로 메모리를 낭비하게 된다.(물론 JVM에 따라서 내부 최적화를 시도할 수 있다) 또 메모리 낭비를 떠나서 같은 값이 계속 생성되는 것은 개발자가 보기에 명확한 중복이다. 이럴 때 사용하면 좋은 것이 바로 `static` 영역이다.

static final

- `FieldInit.MY_VALUE`는 `static` 영역에 존재한다. 그리고 `final` 키워드를 사용해서 초기화 값이 변하지 않는다.
- `static` 영역은 단 하나만 존재하는 영역이다. `MY_VALUE` 변수는 JVM 상에서 하나만 존재하므로 앞서 설명 한 중복과 메모리 비효율 문제를 모두 해결할 수 있다.

이런 이유로 필드에 `final` + 필드 초기화를 사용하는 경우 `static`을 붙여서 사용하는 것이 효과적이다.

final 변수와 상수2

상수(Constant)

상수는 변하지 않고, 항상 일정한 값을 갖는 수를 말한다. 자바에서는 보통 단 하나만 존재하는 변하지 않는 고정된 값을 상수라 한다.

이런 이유로 상수는 `static final` 키워드를 사용한다.

자바 상수 특징

- `static final` 키워드를 사용한다.

- 대문자를 사용하고 구분은 _ (언더스코어)로 한다. (관례)
 - 일반적인 변수와 상수를 구분하기 위해 이렇게 한다.
- 필드를 직접 접근해서 사용한다.
 - 상수는 기능이 아니라 고정된 값 자체를 사용하는 것이 목적이다.
 - 상수는 값을 변경할 수 없다. 따라서 필드에 직접 접근해도 데이터가 변하는 문제가 발생하지 않는다.

```
package final1;

//상수
public class Constant {
    //수학 상수
    public static final double PI = 3.14;
    //시간 상수
    public static final int HOURS_IN_DAY = 24;
    public static final int MINUTES_IN_HOUR = 60;
    public static final int SECONDS_IN_MINUTE = 60;
    //애플리케이션 설정 상수
    public static final int MAX_USERS = 1000;
}
```

- 애플리케이션 안에는 다양한 상수가 존재할 수 있다. 수학, 시간 등등 실생활에서 사용하는 상수부터, 애플리케이션의 다양한 설정을 위한 상수들도 있다.
- 보통 이런 상수들은 애플리케이션 전반에서 사용되기 때문에 `public` 를 자주 사용한다. 물론 특정 위치에서만 사용된다면 다른 접근 제어자를 사용하면 된다.
- 상수는 중앙에서 값을 하나로 관리할 수 있다는 장점도 있다.
- 상수는 런타임에 변경할 수 없다. 상수를 변경하려면 프로그램을 종료하고, 코드를 변경한 다음에 프로그램을 다시 실행해야 한다.

추가로 상수는 중앙에서 값을 하나로 관리할 수 있다는 장점도 있다. 다음 두 예제를 비교해보자.

ConstantMain1 - 상수 없음

```
package final1;

public class ConstantMain1 {

    public static void main(String[] args) {
        System.out.println("프로그램 최대 참여자 수 " + 1000);
        int currentUserCount = 999;
        process(currentUserCount++);
        process(currentUserCount++);
```

```

        process(currentUserCount++);
    }

    private static void process(int currentUserCount) {
        System.out.println("참여자 수:" + currentUserCount);
        if (currentUserCount > 1000) {
            System.out.println("대기자로 등록합니다.");
        } else {
            System.out.println("게임에 참가합니다.");
        }
    }
}

```

이 코드에는 다음과 같은 문제가 있다.

- 만약 프로그램 최대 참여자 수를 현재 1000명에서 2000명으로 변경해야 하면 2곳의 변경 포인트가 발생한다.
만약 애플리케이션의 100곳에서 이 숫자를 사용했다면 100곳을 모두 변경해야 한다.
- 매직 넘버 문제가 발생했다. 숫자 1000이라는 것이 무슨 뜻일까? 이 값만 보고 이해하기 어렵다.

실행 결과

```

프로그램 최대 참여자 수 1000
참여자 수:999
게임에 참가합니다.
참여자 수:1000
게임에 참가합니다.
참여자 수:1001
대기자로 등록합니다.

```

ConstantMain2 - 상수 사용

```

package final1;

public class ConstantMain2 {

    public static void main(String[] args) {
        System.out.println("프로그램 최대 참여자 수 " + Constant.MAX_USERS);
        int currentUserCount = 999;
        process(currentUserCount++);
        process(currentUserCount++);
        process(currentUserCount++);
    }

    private static void process(int currentUserCount) {

```

```

        System.out.println("참여자 수:" + currentUserCount);
        if (currentUserCount > Constant.MAX_USERS) {
            System.out.println("대기자로 등록합니다.");
        } else {
            System.out.println("게임에 참가합니다.");
        }
    }
}

```

- Constant.MAX_USERS 상수를 사용했다. 만약 프로그램 최대 참여자 수를 변경해야 하면 Constant.MAX_USERS의 상수 값만 변경하면 된다.
- 매직 넘버 문제를 해결했다. 숫자 1000이 아니라 사람이 인지할 수 있게 MAX_USERS라는 변수명으로 코드로 이해할 수 있다.

final 변수와 참조

final은 변수의 값을 변경하지 못하게 막는다. 그런데 여기서 변수의 값이라는 것이 뭘까?

- 변수는 크게 기본형 변수와 참조형 변수가 있다.
- 기본형 변수는 10, 20 같은 값을 보관하고, 참조형 변수는 객체의 참조값을 보관한다.
 - final을 기본형 변수에 사용하면 값을 변경할 수 없다.
 - final을 참조형 변수에 사용하면 참조값을 변경할 수 없다.

여기까지는 이해하는데 어려움이 없을 것이다. 이번에는 약간 복잡한 예제를 만들어 보자.

```

package final1;

public class Data {
    public int value;
}

• int value: final이 아니다. 변경할 수 있는 변수다.

```

```

package final1;

public class FinalRefMain {

    public static void main(String[] args) {
        final Data data = new Data();

```

```

//data = new Data(); //final 변경 불가 컴파일 오류

//참조 대상의 값은 변경 가능
data.value = 10;
System.out.println(data.value);
data.value = 20;
System.out.println(data.value);
}
}

```

```

final Data data = new Data()
//data = new Data(); //final 변경 불가 컴파일 오류

```

참조형 변수 `data`에 `final`이 붙었다. 변수 선언 시점에 참조값을 할당했으므로 더는 참조값을 변경할 수 없다.

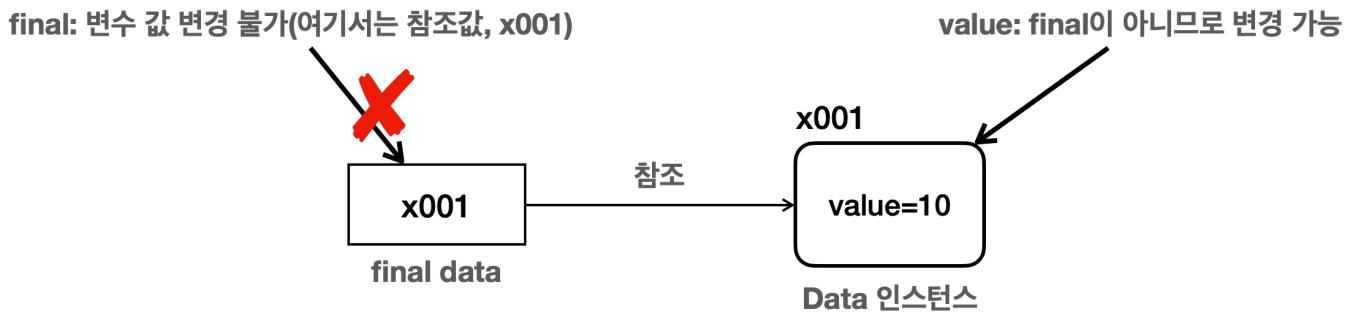
```

data.value = 10
data.value = 20

```

그런데 참조 대상의 객체 값은 변경할 수 있다.

- 참조형 변수 `data`에 `final`이 붙었다. 이 경우 참조형 변수에 들어있는 참조값을 다른 값으로 변경하지 못한다. 쉽게 이야기해서 이제 다른 객체를 참조할 수 없다. 그런데 이것의 정확한 뜻을 잘 이해해야 한다. 참조형 변수에 들어있는 참조값만 변경하지 못한다는 뜻이다. 이 변수 이외에 다른 곳에 영향을 주는 것이 아니다.
- `Data.value`는 `final`이 아니다. 따라서 값을 변경할 수 있다.



정리하면 참조형 변수에 `final`이 붙으면 참조 대상을 자체를 다른 대상으로 변경하지 못하는 것이지, 참조하는 대상의 값을 변경할 수 있다.

정리

`final`은 매우 유용한 제약이다. 만약 특정 변수의 값을 할당한 이후에 변경하지 않아야 한다면 `final`을 사용하자.

예를 들어서 고객의 `id` 변경하면 큰 문제가 발생한다면 `final`로 선언하고 생성자로 값을 할당하자.
만약 어디선가 실수로 `id` 값을 변경한다면 컴파일러가 문제를 찾아줄 것이다.

```
package final1.ex;

public class Member {

    private final String id; //final 키워드 사용
    private String name;

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

    public void changeData(String id, String name) {
        //this.id = id; //컴파일 오류 발생
        this.name = name;
    }

    public void print() {
        System.out.println("id:" + id + ", name:" + name);
    }
}
```

- `changeData()` 메서드에서 `final`인 `id` 값 변경을 시도하면 컴파일 오류가 발생한다.

```
package final1.ex;

public class MemberMain {

    public static void main(String[] args) {
        Member member = new Member("myId", "kim");
        member.print();
        member.changeData("myId2", "seo");
        member.print();
    }
}
```

실행 결과

```
id:myId, name:kim
```

id:myId, name:seo

9. 상속

#1.인강/0.자바/2.자바-기본

- /상속 - 시작
- /상속 관계
- /상속과 메모리 구조
- /상속과 기능 추가
- /상속과 메서드 오버라이딩
- /상속과 접근 제어
- /super - 부모 참조
- /super - 생성자
- /문제와 풀이
- /정리

상속 - 시작

상속 관계가 왜 필요한지 이해하기 위해 다음 예제 코드를 만들어서 실행해보자.

예제 코드

패키지 위치에 주의하자

```
package extends1.ex1;

public class ElectricCar {

    public void move() {
        System.out.println("차를 이동합니다.");
    }

    public void charge() {
        System.out.println("충전합니다.");
    }
}
```

```
package extends1.ex1;

public class GasCar {
```

```

public void move() {
    System.out.println("차를 이동합니다.");
}

public void fillUp() {
    System.out.println("기름을 주유합니다.");
}

```

```

package extends1.ex1;

public class CarMain {

    public static void main(String[] args) {
        ElectricCar electricCar = new ElectricCar();
        electricCar.move();
        electricCar.charge();

        GasCar gasCar = new GasCar();
        gasCar.move();
        gasCar.fillUp();
    }
}

```

실행 결과

차를 이동합니다.
충전합니다.
차를 이동합니다.
기름을 주유합니다.

+ move()
+ charge()

ElectricCar

+ move()
+ fillUp()

GasCar

전기차(ElectricCar)와 가솔린차(GasCar)를 만들었다. 전기차는 이동(move()), 충전(charge()) 기능이 있고, 가솔린차는 이동(move()), 주유(fillUp()) 기능이 있다.

전기차와 가솔린차는 자동차(Car)의 좀 더 구체적인 개념이다. 반대로 자동차(Car)는 전기차와 가솔린차를 포함하는 추상적인 개념이다. 그래서인지 잘 보면 둘의 공통 기능이 보인다. 바로 이동(move())이다.

전기차든 가솔린차든 주유하는 방식이 다른 것이지 이동하는 것은 똑같다. 이런 경우 상속 관계를 사용하는 것이 효과적이다.

상속 관계

상속은 객체 지향 프로그래밍의 핵심 요소 중 하나로, 기존 클래스의 필드와 메서드를 새로운 클래스에서 재사용하게 해준다. 이를 그대로 기존 클래스의 속성과 기능을 그대로 물려받는 것이다. 상속을 사용하려면 extends 키워드를 사용하면 된다. 그리고 extends 대상은 하나만 선택할 수 있다.

용어 정리

- 부모 클래스 (슈퍼 클래스): 상속을 통해 자신의 필드와 메서드를 다른 클래스에 제공하는 클래스
- 자식 클래스 (서브 클래스): 부모 클래스로부터 필드와 메서드를 상속받는 클래스

주의!

지금부터 코드를 작성할 때 기존 코드를 유지하기 위해, 새로운 패키지에 기존 코드를 옮겨가면서 코드를 작성할 것이다. 클래스의 이름이 같기 때문에 패키지 명과 import 사용에 주의해야 한다.

상속 관계를 사용하도록 코드를 작성해보자.

기존 코드를 유지하기 위해 ex2 패키지를 새로 만들자

```
package extends1.ex2;

public class Car {
    public void move() {
        System.out.println("차를 이동합니다.");
    }
}
```

Car는 부모 클래스가 된다. 여기에는 자동차의 공통 기능인 move()가 포함되어 있다.

```
package extends1.ex2;

public class ElectricCar extends Car {

    public void charge() {
        System.out.println("충전합니다.");
    }
}
```

전기차는 `extends Car`를 사용해서 부모 클래스인 `Car`를 상속 받는다. 상속 덕분에 `ElectricCar`에서도 `move()`를 사용할 수 있다.

```
package extends1.ex2;

public class GasCar extends Car {

    public void fillUp() {
        System.out.println("기름을 주유합니다.");
    }
}
```

가솔린차도 전기차와 마찬가지로 `extends Car`를 사용해서 부모 클래스인 `Car`를 상속 받는다. 상속 덕분에 여기서도 `move()`를 사용할 수 있다.

```
package extends1.ex2;

public class CarMain {

    public static void main(String[] args) {
        ElectricCar electricCar = new ElectricCar();
        electricCar.move();
        electricCar.charge();

        GasCar gasCar = new GasCar();
        gasCar.move();
        gasCar.fillUp();
    }
}
```

실행 결과

차를 이동합니다.

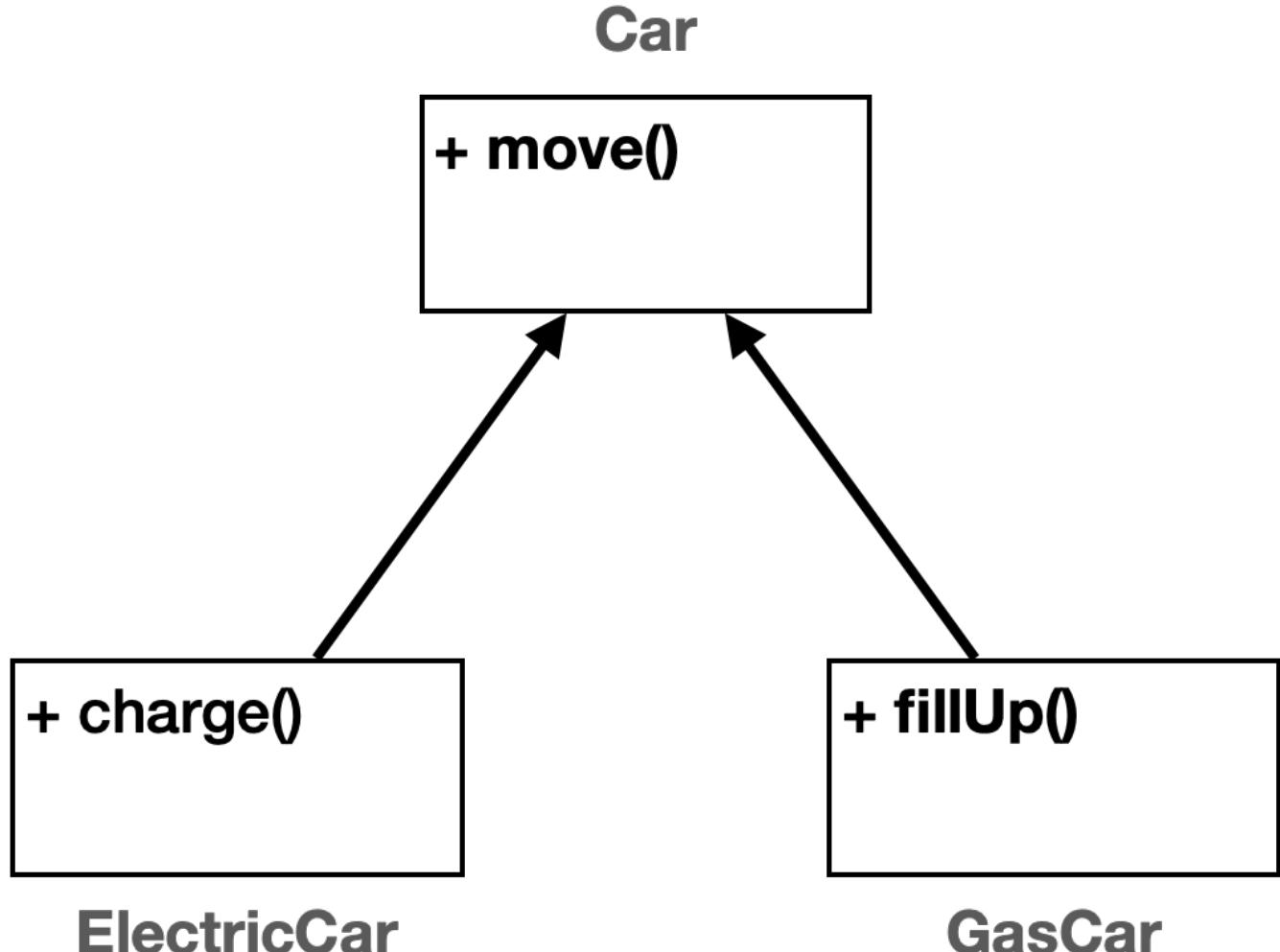
충전합니다.

차를 이동합니다.

기름을 주유합니다.

실행 결과는 기존 예제와 완전히 동일하다.

상속 구조도



전기차와 가솔린차가 `Car`를 상속 받은 덕분에 `electricCar.move()`, `gasCar.move()`를 사용할 수 있다.

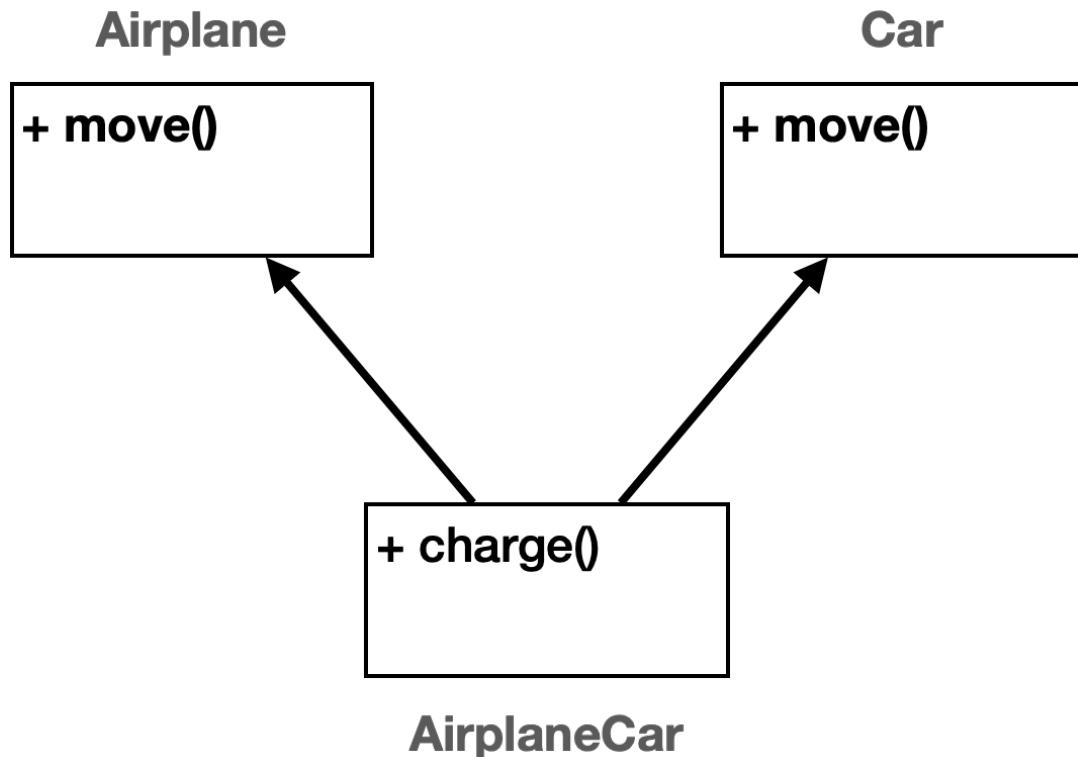
참고로 당연한 이야기지만 상속은 부모의 기능을 자식이 물려 받는 것이다. 따라서 자식이 부모의 기능을 물려 받아서 사용할 수 있다. 반대로 부모 클래스는 자식 클래스에 접근할 수 없다. 자식 클래스는 부모 클래스의 기능을 물려 받기 때문에 접근할 수 있지만, 그 반대는 아니다. 부모 코드를 보자! 자식에 대한 정보가 하나도 없다. 반면에 자식 코드는 `extends Car`를 통해서 부모를 알고 있다.

단일 상속

참고로 자바는 다중 상속을 지원하지 않는다. 그래서 `extend` 대상은 하나만 선택할 수 있다. 부모를 하나만 선택할 수

있다는 뜻이다. 물론 부모가 또 다른 부모를 하나 가지는 것은 괜찮다.

다중 상속 그림



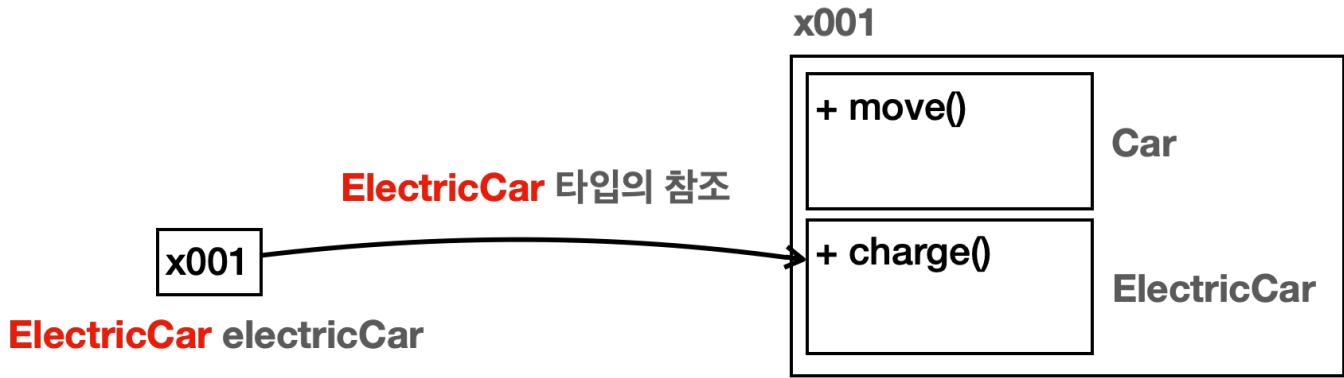
만약 비행기와 자동차를 상속 받아서 하늘을 나는 자동차를 만든다고 가정해보자. 만약 그림과 같이 다중 상속을 사용하게 되면 `AirplaneCar` 입장에서 `move()`를 호출할 때 어떤 부모의 `move()`를 사용해야 할지 애매한 문제가 발생한다. 이것을 다이아몬드 문제라 한다. 그리고 다중 상속을 사용하면 클래스 계층 구조가 매우 복잡해지 수 있다. 이런 문제점 때문에 자바는 클래스의 다중 상속을 허용하지 않는다. 대신에 이후에 설명한 인터페이스의 다중 구현을 허용해서 이러한 문제를 피한다.

상속과 메모리 구조

이 부분을 제대로 이해하는 것이 앞으로 정말 중요하다!

상속 관계를 객체로 생성할 때 메모리 구조를 확인해보자.

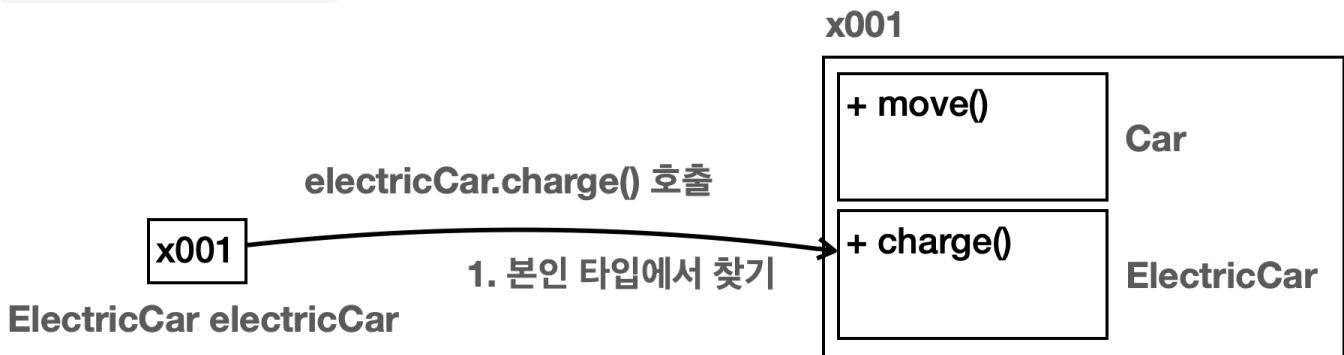
```
ElectricCar electricCar = new ElectricCar();
```



`new ElectricCar()` 를 호출하면 `ElectricCar` 뿐만 아니라 상속 관계에 있는 `Car` 까지 함께 포함해서 인스턴스를 생성한다. 참조값은 `x001`로 하나이지만 실제로 그 안에서는 `Car`, `ElectricCar`라는 두가지 클래스 정보가 공존하는 것이다.

상속이라고 해서 단순하게 부모의 필드와 메서드만 물려 받는게 아니다. 상속 관계를 사용하면 부모 클래스도 함께 포함해서 생성된다. 외부에서 볼때는 하나의 인스턴스를 생성하는 것 같지만 내부에서는 부모와 자식이 모두 생성되고 공간도 구분된다.

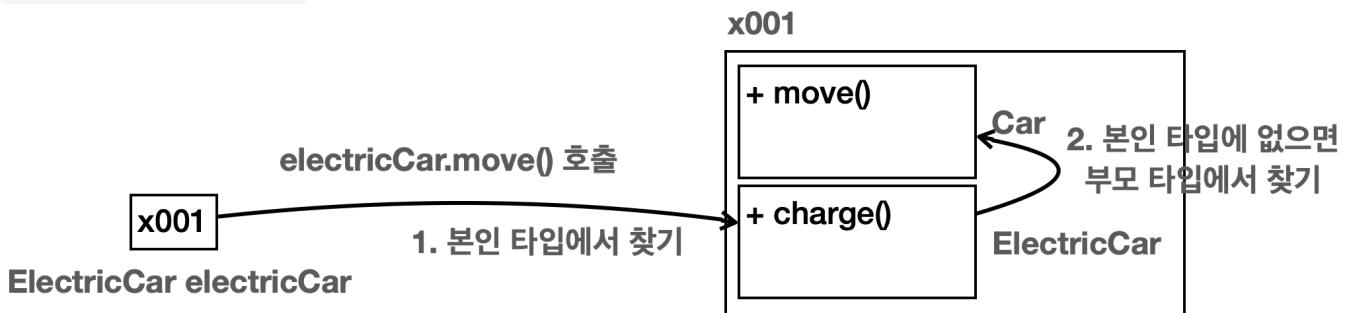
`electricCar.charge() 호출`



`electricCar.charge()` 를 호출하면 참조값을 확인해서 `x001.charge()` 를 호출한다. 따라서 `x001`을 찾아서 `charge()` 를 호출하면 되는 것이다. 그런데 상속 관계의 경우에는 내부에 부모와 자식이 모두 존재한다. 이때 부모인 `Car` 를 통해서 `charge()` 를 찾을지 아니면 `ElectricCar` 를 통해서 `charge()` 를 찾을지 선택해야 한다.

이때는 **호출하는 변수의 타입(클래스)을 기준으로 선택한다.** `electricCar` 변수의 타입이 `ElectricCar` 이므로 인스턴스 내부에 같은 타입인 `ElectricCar`를 통해서 `charge()` 를 호출한다.

`electricCar.move() 호출`



`electricCar.move()` 를 호출하면 먼저 x001 참조로 이동한다. 내부에는 `Car`, `ElectricCar` 두가지 타입이 있다. 이때 호출하는 변수인 `electricCar` 의 타입이 `ElectricCar` 이므로 이 타입을 선택한다. 그런데 `ElectricCar` 에는 `move()` 메서드가 없다. 상속 관계에서는 자식 타입에 해당 기능이 없으면 부모 타입으로 올라가서 찾는다. 이 경우 `ElectricCar` 의 부모인 `Car` 로 올라가서 `move()` 를 찾는다. 부모인 `Car` 에 `move()` 가 있으므로 부모에 있는 `move()` 메서드를 호출한다.

만약 부모에서도 해당 기능을 찾지 못하면 더 상위 부모에서 필요한 기능을 찾아본다. 부모에 부모로 계속 올라가면서 필드나 메서드를 찾는 것이다. 물론 계속 찾아도 없으면 컴파일 오류가 발생한다.

지금까지 설명한 상속과 메모리 구조는 반드시 이해해야 한다!

- 상속 관계의 객체를 생성하면 그 내부에는 부모와 자식이 모두 생성된다.
- 상속 관계의 객체를 호출할 때, 대상 타입을 정해야 한다. 이때 호출자의 타입을 통해 대상 타입을 찾는다.
- 현재 타입에서 기능을 찾지 못하면 상위 부모 타입으로 기능을 찾아서 실행한다. 기능을 찾지 못하면 컴파일 오류가 발생한다.

상속과 기능 추가

이번에는 상속 관계의 장점을 알아보기 위해, 상속 관계에 다음 기능을 추가해보자.

- 모든 차량에 문열기(`openDoor()`) 기능을 추가해야 한다.
- 새로운 수소차(`HydrogenCar`)를 추가해야 한다.
 - 수소차는 `fillHydrogen()` 기능을 통해 수소를 충전할 수 있다.

기존 코드를 유지하기 위해 `ex3` 패키지를 새로 만들자

```
package extends1.ex3;

public class Car {

    public void move() {
        System.out.println("차를 이동합니다.");
    }

    //추가
    public void openDoor() {
        System.out.println("문을 엽니다.");
    }
}
```

모든 차량에 문열기 기능을 추가할 때는 상위 부모인 Car에 openDoor() 기능을 추가하면 된다. 이렇게 하면 Car의 자식들은 해당 기능을 모두 물려받게 된다. 만약 상속 관계가 아니었다면 각각의 차량에 해당 기능을 모두 추가해야 한다.

```
package extends1.ex3;

public class ElectricCar extends Car {

    public void charge() {
        System.out.println("충전합니다.");
    }
}
```

기존 코드와 같다.

```
package extends1.ex3;

public class GasCar extends Car {

    public void fillUp() {
        System.out.println("기름을 주유합니다.");
    }
}
```

기존 코드와 같다.

```
package extends1.ex3;

//추가
public class HydrogenCar extends Car {

    public void fillHydrogen() {
        System.out.println("수소를 충전합니다.");
    }
}
```

수소차를 추가했다. Car를 상속받은 덕분에 move(), openDoor() 와 같은 기능을 바로 사용할 수 있다. 수소차는 전용 기능인 수소 충전(fillHydrogen()) 기능을 제공한다.

```
package extends1.ex3;

public class CarMain {
```

```
public static void main(String[] args) {
    ElectricCar electricCar = new ElectricCar();
    electricCar.move();
    electricCar.charge();
    electricCar.openDoor();

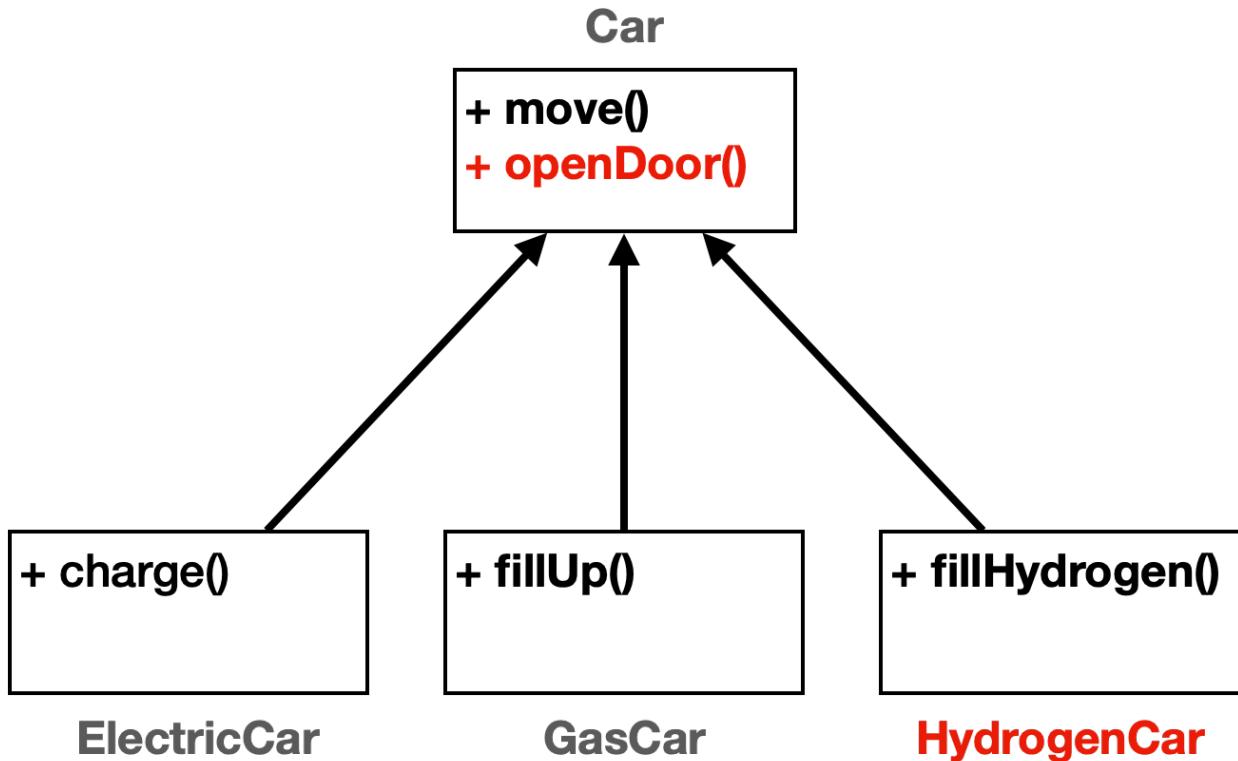
    GasCar gasCar = new GasCar();
    gasCar.move();
    gasCar.fillUp();
    gasCar.openDoor();

    HydrogenCar hydrogenCar = new HydrogenCar();
    hydrogenCar.move();
    hydrogenCar.fillHydrogen();
    hydrogenCar.openDoor();
}
}
```

실행 결과

```
차를 이동합니다.  
충전합니다.  
문을 엽니다.  
차를 이동합니다.  
기름을 주유합니다.  
문을 엽니다.  
차를 이동합니다.  
수소를 충전합니다.  
문을 엽니다.
```

기능 추가와 클래스 확장



상속 관계 덕분에 중복은 줄어들고, 새로운 수소차를 편리하게 확장(extend)한 것을 알 수 있다.

상속과 메서드 오버라이딩

부모 타입의 기능을 자식에서는 다르게 재정의 하고 싶을 수 있다.

예를 들어서 자동차의 경우 `Car.move()`라는 기능이 있다. 이 기능을 사용하면 단순히 "차를 이동합니다."라고 출력 한다. 전기차의 경우 보통 더 빠르기 때문에 전자가가 `move()`를 호출한 경우에는 "전기차를 빠르게 이동합니다."라고 출력을 변경하고 싶다.

이렇게 부모에게서 상속 받은 기능을 자식이 재정의 하는 것을 **메서드 오버라이딩(Overriding)**이라 한다.

기존 코드를 유지하기 위해 **overriding** 패키지를 새로 만들자

```

package extends1.overriding;

public class Car {

    public void move() {
        System.out.println("차를 이동합니다.");
    }
}

```

```
public void openDoor() {  
    System.out.println("문을 엽니다.");  
}  
}
```

기존 코드와 같다.

```
package extends1.overriding;  
  
public class GasCar extends Car {  
  
    public void fillUp() {  
        System.out.println("기름을 주유합니다.");  
    }  
}
```

기존 코드와 같다.

```
package extends1.overriding;  
  
public class ElectricCar extends Car {  
  
    @Override  
    public void move() {  
        System.out.println("전기차를 빠르게 이동합니다.");  
    }  
  
    public void charge() {  
        System.out.println("충전합니다.");  
    }  
}
```

ElectricCar는 부모인 Car의 move() 기능을 그대로 사용하고 싶지 않다. 메서드 이름은 같지만 새로운 기능을 사용하고 싶다. 그래서 ElectricCar의 move() 메서드를 새로 만들었다.

이렇게 부모의 기능을 자식이 새로 재정의하는 것을 메서드 오버라이딩이라 한다.

이제 ElectricCar의 move()를 호출하면 Car의 move()가 아니라 ElectricCar의 move()가 호출된다.

@Override

@이 붙은 부분을 애노테이션이라 한다. 애노테이션은 주석과 비슷한데, 프로그램이 읽을 수 있는 특별한 주석이라 생각하면 된다. 애노테이션에 대한 자세한 내용은 따로 설명한다.

이 애노테이션은 상위 클래스의 메서드를 오버라이드하는 것임을 나타낸다.

이름 그대로 오버라이딩한 메서드 위에 이 애노테이션을 붙여야 한다.

컴파일러는 이 애노테이션을 보고 메서드가 정확히 오버라이드 되었는지 확인한다. 오버라이딩 조건을 만족시키지 않으면 컴파일 에러를 발생시킨다. 따라서 실수로 오버라이딩을 못하는 경우를 방지해준다. 예를 들어서 이 경우에 만약 부모에 move() 메서드가 없다면 컴파일 오류가 발생한다. 참고로 이 기능은 필수는 아니지만 코드의 명확성을 위해 붙여주는 것이 좋다.

```
package extends1.overriding;

public class CarMain {

    public static void main(String[] args) {
        ElectricCar electricCar = new ElectricCar();
        electricCar.move();

        GasCar gasCar = new GasCar();
        gasCar.move();
    }
}
```

실행 결과

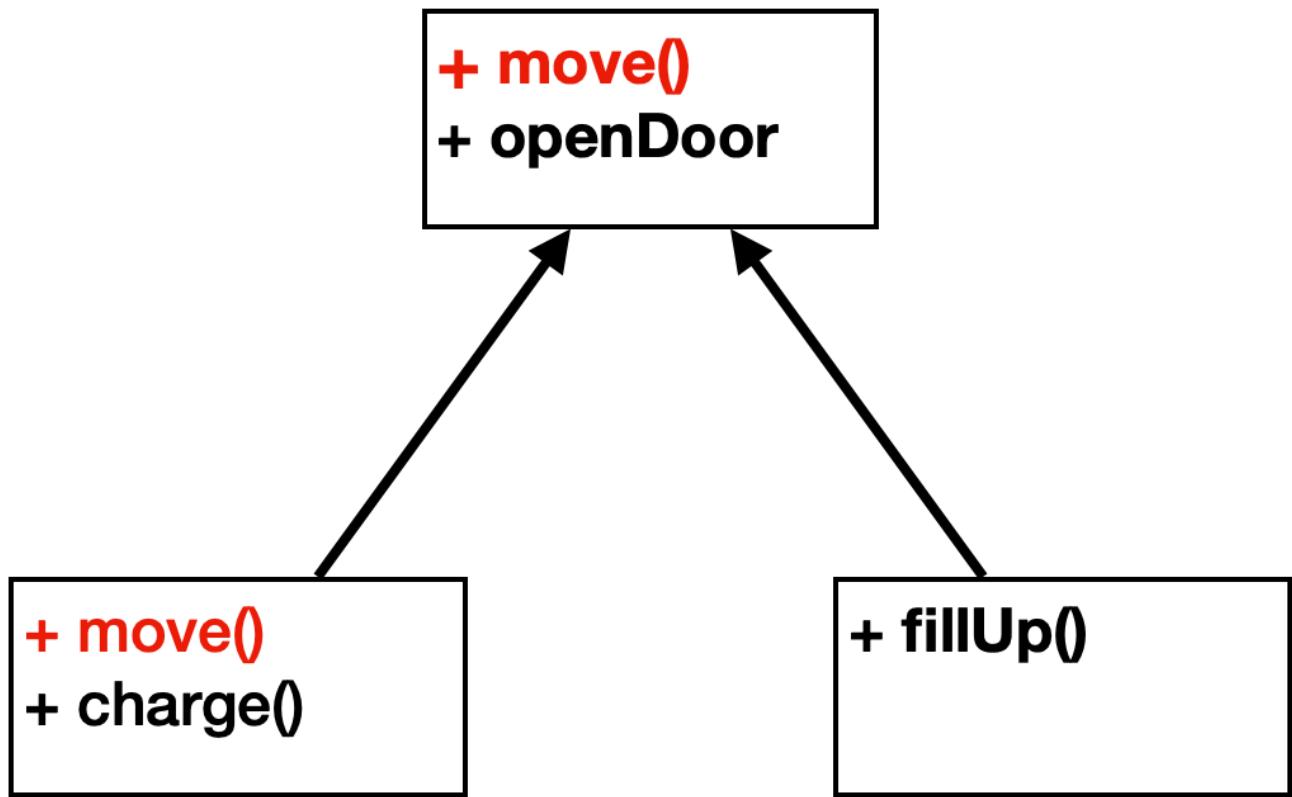
전기차를 빠르게 이동합니다.

차를 이동합니다.

실행 결과를 보면 electricCar.move() 를 호출했을 때 오버라이딩한 ElectricCar.move() 메서드가 실행된 것을 확인할 수 있다.

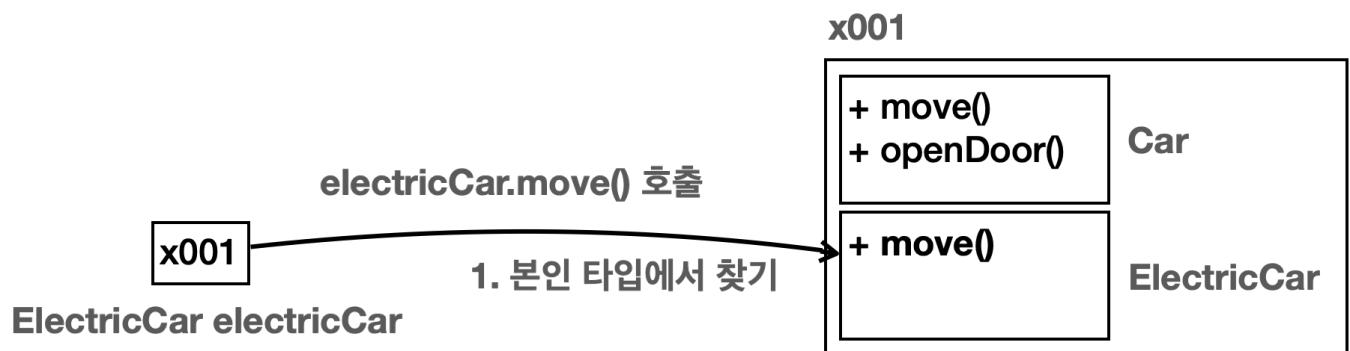
오버라이딩과 클래스

Car



Car 의 `move()` 메서드를 ElectricCar에서 오버라이딩 했다.

오버라이딩과 메모리 구조



1. `electricCar.move()` 를 호출한다.
2. 호출한 `electricCar`의 타입은 `ElectricCar`이다. 따라서 인스턴스 내부의 `ElectricCar` 타입에서 시작 한다.
3. `ElectricCar` 타입에 `move()` 메서드가 있다. 해당 메서드를 실행한다. 이때 실행할 메서드를 이미 찾았으므로 부모 타입을 찾지 않는다.

오버로딩(Overloading)과 오버라이딩(Overriding)

- **메서드 오버로딩:** 메서드 이름이 같고 매개변수(파라미터)가 다른 메서드를 여러개 정의하는 것을 메서드 오버로

딩(Overloading)이라 한다. 오버로딩은 번역하면 과적인데, 과하게 물건을 담았다는 뜻이다. 따라서 같은 이름의 메서드를 여러개 정의했다고 이해하면 된다.

- **메서드 오버라이딩:** 메서드 오버라이딩은 하위 클래스에서 상위 클래스의 메서드를 재정의하는 과정을 의미한다. 따라서 상속 관계에서 사용한다. 부모의 기능을 자식이 다시 정의하는 것이다. 오버라이딩을 단순히 해석하면 무언가를 넘어서 타는 것을 말한다. 자식의 새로운 기능이 부모의 기존 기능을 넘어 타서 기존 기능을 새로운 기능으로 덮어버린다고 이해하면 된다. 오버라이딩을 우리말로 번역하면 무언가를 다시 정의한다고 해서 **재정의**라 한다. 상속 관계에서는 기존 기능을 다시 정의한다고 이해하면 된다. 실무에서는 메서드 오버라이딩, 메서드 재정의 둘 다 사용한다.

메서드 오버라이딩 조건

메서드 오버라이딩은 다음과 같은 까다로운 조건을 가지고 있다.

다음 내용은 아직 학습하지 않은 내용들도 있으므로 모두 이해하려고 하기 보다는 참고만 하자.

지금은 단순히 **부모 메서드와 같은 메서드를 오버라이딩 할 수 있다 정도로 이해하면 충분하다.**

메서드 오버라이딩 조건

- **메서드 이름:** 메서드 이름이 같아야 한다.
- **메서드 매개변수(파라미터):** 매개변수(파라미터) 타입, 순서, 개수가 같아야 한다.
- **반환 타입:** 반환 타입이 같아야 한다. 단 반환 타입이 하위 클래스 타입일 수 있다.
- **접근 제어자:** 오버라이딩 메서드의 접근 제어자는 상위 클래스의 메서드보다 더 제한적이어서는 안된다. 예를 들어, 상위 클래스의 메서드가 `protected`로 선언되어 있으면 하위 클래스에서 이를 `public` 또는 `protected`로 오버라이드할 수 있지만, `private` 또는 `default`로 오버라이드 할 수 없다.
- **예외:** 오버라이딩 메서드는 상위 클래스의 메서드보다 더 많은 체크 예외를 `throws`로 선언할 수 없다. 하지만 더 적거나 같은 수의 예외, 또는 하위 타입의 예외는 선언할 수 있다. 예외를 학습해야 이해할 수 있다. 예외는 뒤에서 다룬다.
- **static, final, private:** 키워드가 붙은 메서드는 오버라이딩 될 수 없다.
 - `static`은 클래스 레벨에서 작동하므로 인스턴스 레벨에서 사용하는 오버라이딩이 의미가 없다. 쉽게 이야기해서 그냥 클래스 이름을 통해 필요한 곳에 직접 접근하면 된다.
 - `final` 메서드는 재정의를 금지한다.
 - `private` 메서드는 해당 클래스에서만 접근 가능하기 때문에 하위 클래스에서 보이지 않는다. 따라서 오버라이딩 할 수 없다.
- **생성자 오버라이딩:** 생성자는 오버라이딩 할 수 없다.

상속과 접근 제어

상속 관계와 접근 제어에 대해 알아보자. 참고로 접근 제어를 자세히 설명하기 위해 부모와 자식의 패키지를 따로 분리하였다. 이 부분에 유의해서 예제를 만들어보자.

패키지: parent

Parent

+ **publicValue**
protectedValue
~ **defaultValue**
- **privateValue**

+ **publicMethod()**
protectedMethod()
~ **defaultMethod()**
- **privateMethod()**
+ **printParent()**

패키지: child

Child

+ **call()**



접근 제어자를 표현하기 위해 UML 표기법을 일부 사용했다.

- + : public
- # : protected
- ~ : default
- - : private

접근 제어자를 잠시 복습해보자.

접근 제어자의 종류

- **private** : 모든 외부 호출을 막는다.
- **default** (package-private) : 같은 패키지안에서 호출은 허용한다.
- **protected** : 같은 패키지안에서 호출은 허용한다. 패키지가 달라도 상속 관계의 호출은 허용한다.
- **public** : 모든 외부 호출을 허용한다.

순서대로 **private**이 가장 많이 차단하고, **public**이 가장 많이 허용한다.

private -> **default** -> **protected** -> **public**

그림과 같이 다양한 접근 제어자를 사용하도록 코드를 작성해보자.

```
package extends1.access.parent;

public class Parent {

    public int publicValue;
    protected int protectedValue;
    int defaultValue;
    private int privateValue;

    public void publicMethod() {
        System.out.println("Parent.publicMethod");
    }
    protected void protectedMethod() {
        System.out.println("Parent.protectedMethod");
    }
    void defaultMethod() {
        System.out.println("Parent.defaultMethod");
    }
    private void privateMethod() {
        System.out.println("Parent.privateMethod");
    }
}
```

```

public void printParent() {
    System.out.println("==Parent 메서드 안==");
    System.out.println("publicValue = " + publicValue);
    System.out.println("protectedValue = " + protectedValue);
    System.out.println("defaultValue = " + defaultValue); //부모 메서드 안에서 접근 가능
    System.out.println("privateValue = " + privateValue); //부모 메서드 안에서 접근 가능
}

}

//부모 메서드 안에서 모두 접근 가능
defaultMethod();
privateMethod();
}
}

```

부모 클래스인 Parent에는 public, protected, default, private과 같은 모든 접근 제어자가 필드와 메서드에 모두 존재한다.

```

package extends1.access.child;

import extends1.access.parent.Parent;

public class Child extends Parent {

    public void call() {
        publicValue = 1;
        protectedValue = 1; //상속 관계 or 같은 패키지
        //defaultValue = 1; //다른 패키지 접근 불가, 컴파일 오류
        //privateValue = 1; //접근 불가, 컴파일 오류

        publicMethod();
        protectedMethod(); //상속 관계 or 같은 패키지
        //defaultMethod(); //다른 패키지 접근 불가, 컴파일 오류
        //privateMethod(); //접근 불가, 컴파일 오류

        printParent();
    }
}

```

둘의 패키지가 다르다는 부분의 유의하자

자식 클래스인 Child에서 부모 클래스인 Parent에 얼마나 접근할 수 있는지 확인해보자.

- `publicValue = 1`: 부모의 `public` 필드에 접근한다. `public` 이므로 접근할 수 있다.
- `protectedValue = 1`: 부모의 `protected` 필드에 접근한다. 자식과 부모는 다른 패키지이지만, 상속 관계 이므로 접근할 수 있다.
- `defaultValue = 1`: 부모의 `default` 필드에 접근한다. 자식과 부모가 다른 패키지이므로 접근할 수 없다.
- `privateValue = 1`: 부모의 `private` 필드에 접근한다. `private`은 모든 외부 접근을 막으므로 자식이라도 호출할 수 없다.

메서드의 경우도 앞서 설명한 필드와 동일하다.

```
package extends1.access;

import extends1.access.child.Child;

public class ExtendsAccessMain {

    public static void main(String[] args) {
        Child child = new Child();
        child.call();
    }
}
```

실행 결과

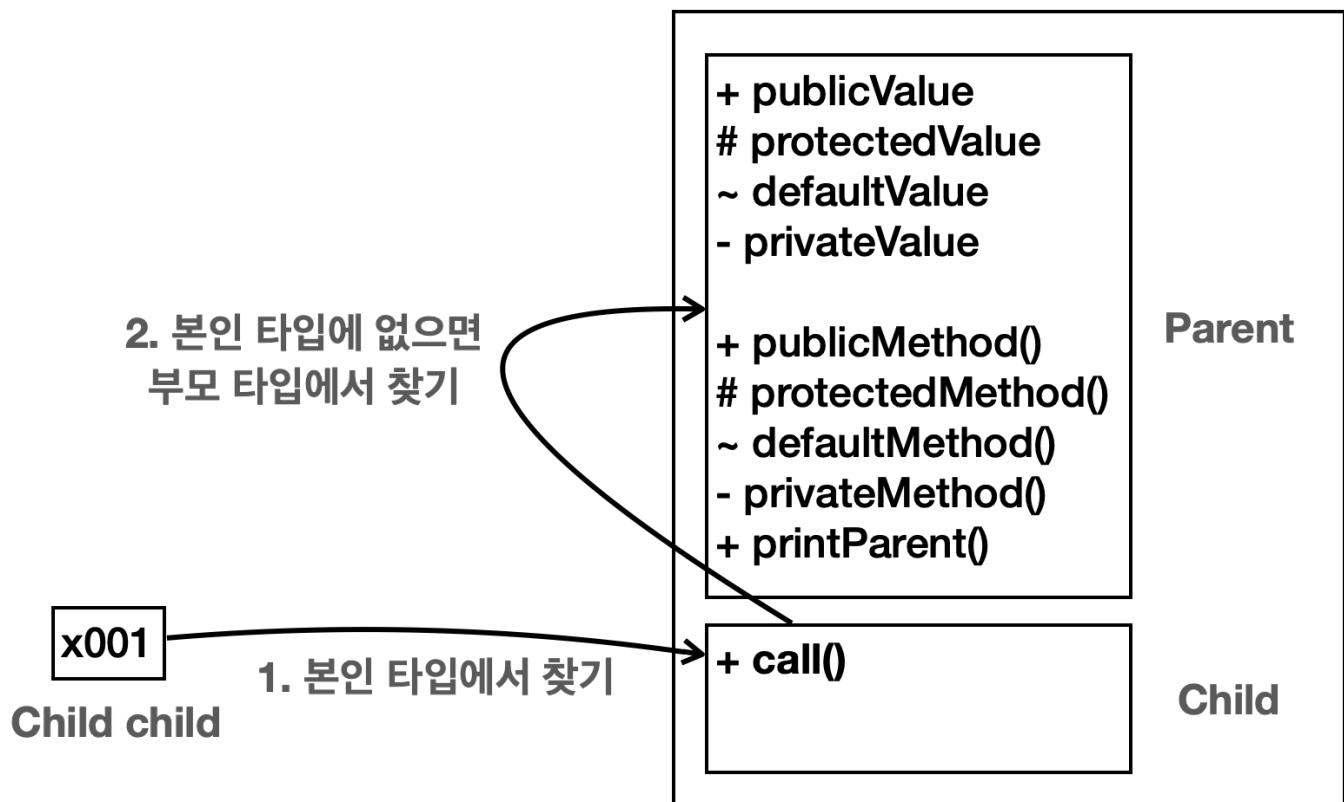
```
Parent.publicMethod
Parent.protectedMethod
==Parent 메서드 안==
publicValue = 1
protectedValue = 1
defaultValue = 0
privateValue = 0
Parent.defaultMethod
Parent.privateMethod
```

코드를 실행해보면 `Child.call()` → `Parent.printParent()` 순서로 호출한다.

`Child`는 부모의 `public`, `protected` 필드나 메서드만 접근할 수 있다. 반면에 `Parent.printParent()`의 경우 `Parent` 안에 있는 메서드이기 때문에 `Parent` 자신의 모든 필드와 메서드에 얼마든지 접근할 수 있다.

접근 제어와 메모리 구조

x001



본인 타입에 없으면 부모 타입에서 기능을 찾는데, 이때 접근 제어자가 영향을 준다. 왜냐하면 객체 내부에서는 자식과 부모가 구분되어 있기 때문이다. 결국 자식 타입에서 부모 타입의 기능을 호출할 때, 부모 입장에서 보면 외부에서 호출한 것과 같다.

super - 부모 참조

부모와 자식의 필드명이 같거나 메서드가 오버라이딩 되어 있으면, 자식에서 부모의 필드나 메서드를 호출할 수 없다. 이때 `super` 키워드를 사용하면 부모를 참조할 수 있다. `super`는 이름 그대로 부모 클래스에 대한 참조를 나타낸다.

다음 예를 보자. 부모의 필드명과 자식의 필드명이 둘다 `value`로 똑같다. 메서드도 `hello()`로 자식에서 오버라이딩 되어 있다. 이때 자식 클래스에서 부모 클래스의 `value`와 `hello()`를 호출하고 싶다면 `super` 키워드를 사용하면 된다.

Parent

```
+ value="parent"  
+ hello()
```



```
+ value="child"  
+ hello()  
+ call()
```

Child

```
package extends1.super1;  
  
public class Parent {  
    public String value = "parent";  
  
    public void hello() {  
        System.out.println("Parent.hello");  
    }  
}
```

```
package extends1.super1;  
  
public class Child extends Parent {  
    public String value = "child";  
  
    @Override  
    public void hello() {
```

```

        System.out.println("Child.hello");
    }

    public void call() {
        System.out.println("this value = " + this.value); //this 생략 가능
        System.out.println("super value = " + super.value);

        this.hello(); //this 생략 가능
        super.hello();
    }
}

```

`call()` 메서드를 보자.

- `this`는 자기 자신의 참조를 뜻한다. `this`는 생략할 수 있다.
- `super`는 부모 클래스에 대한 참조를 뜻한다.
- 필드 이름과 메서드 이름이 같지만 `super`를 사용해서 부모 클래스에 있는 기능을 사용할 수 있다.

```

package extends1.super1;

public class Super1Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.call();
    }
}

```

실행 결과

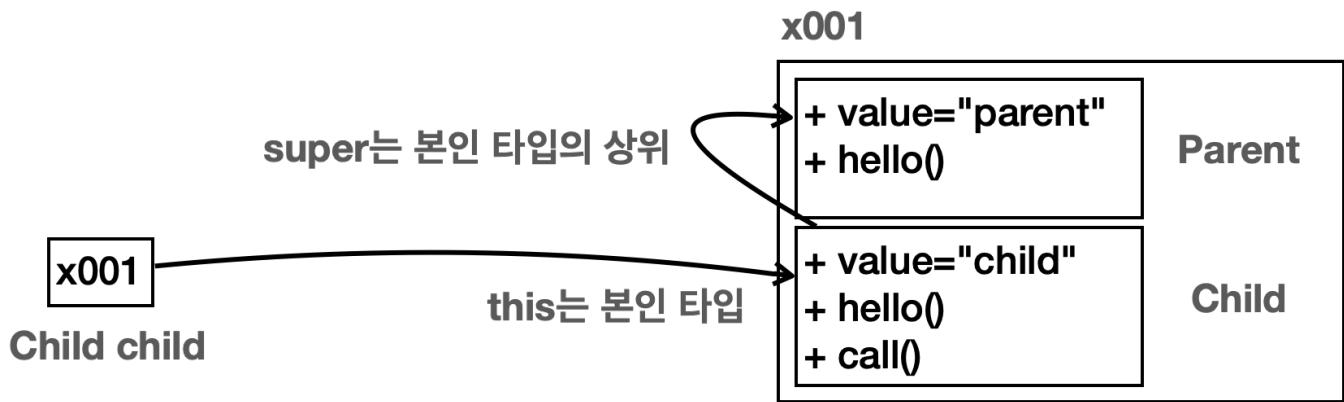
```

this value = child
super value = parent
Child.hello
Parent.hello

```

실행 결과를 보면 `super`를 사용한 경우 부모 클래스의 기능을 사용한 것을 확인할 수 있다.

super 메모리 그림



super - 생성자

상속 관계의 인스턴스를 생성하면 결국 메모리 내부에는 자식과 부모 클래스가 각각 다 만들어진다. Child 를 만들면 부모인 Parent 까지 함께 만들어지는 것이다. 따라서 각각의 생성자도 모두 호출되어야 한다.

상속 관계를 사용하면 자식 클래스의 생성자에서 부모 클래스의 생성자를 반드시 호출해야 한다.(규칙)

상속 관계에서 부모의 생성자를 호출할 때는 super(...) 를 사용하면 된다.

예제를 통해 상속 관계에서 생성자를 어떻게 사용하는지 알아보자.

```
package extends1.super2;

public class ClassA {

    public ClassA() {
        System.out.println("ClassA 생성자");
    }
}
```

- ClassA 는 최상위 부모 클래스이다.

```
package extends1.super2;

public class ClassB extends ClassA {

    public ClassB(int a) {
        super(); //기본 생성자 생략 가능
        System.out.println("ClassB 생성자 a="+a);
    }
}
```

```
}
```

```
public ClassB(int a, int b) {  
    super(); //기본 생성자 생략 가능  
    System.out.println("ClassB 생성자 a=" + a + " b=" + b);  
}
```

- ClassB는 ClassA를 상속 받았다. 상속을 받으면 생성자의 첫줄에 super(...)를 사용해서 부모 클래스의 생성자를 호출해야 한다.
 - 예외로 생성자 첫줄에 this(...)를 사용할 수는 있다. 하지만 super(...)는 자신의 생성자 안에서 언젠가는 반드시 호출해야 한다.
- 부모 클래스의 생성자가 기본 생성자(파라미터가 없는 생성자)인 경우에는 super()를 생략할 수 있다.
 - 상속 관계에서 첫줄에 super(...)를 생략하면 자바는 부모의 기본 생성자를 호출하는 super()를자동으로 만들어준다.
 - 참고로 기본 생성자를 많이 사용하기 때문에 편의상 이런 기능을 제공한다.

```
package extends1.super2;  
  
public class ClassC extends ClassB {  
  
    public ClassC() {  
        super(10, 20);  
        System.out.println("ClassC 생성자");  
    }  
}
```

- ClassC는 ClassB를 상속 받았다. ClassB 다음 두 생성자가 있다.
 - ClassB(int a)
 - ClassB(int a, int b)
- 생성자는 하나만 호출할 수 있다. 두 생성자 중에 하나를 선택하면 된다.
 - super(10, 20)를 통해 부모 클래스의 ClassB(int a, int b) 생성자를 선택했다.
- 참고로 ClassC의 부모인 ClassB에는 기본 생성자가 없다. 따라서 부모의 기본 생성자를 호출하는 super()를 사용하거나 생략할 수 없다.

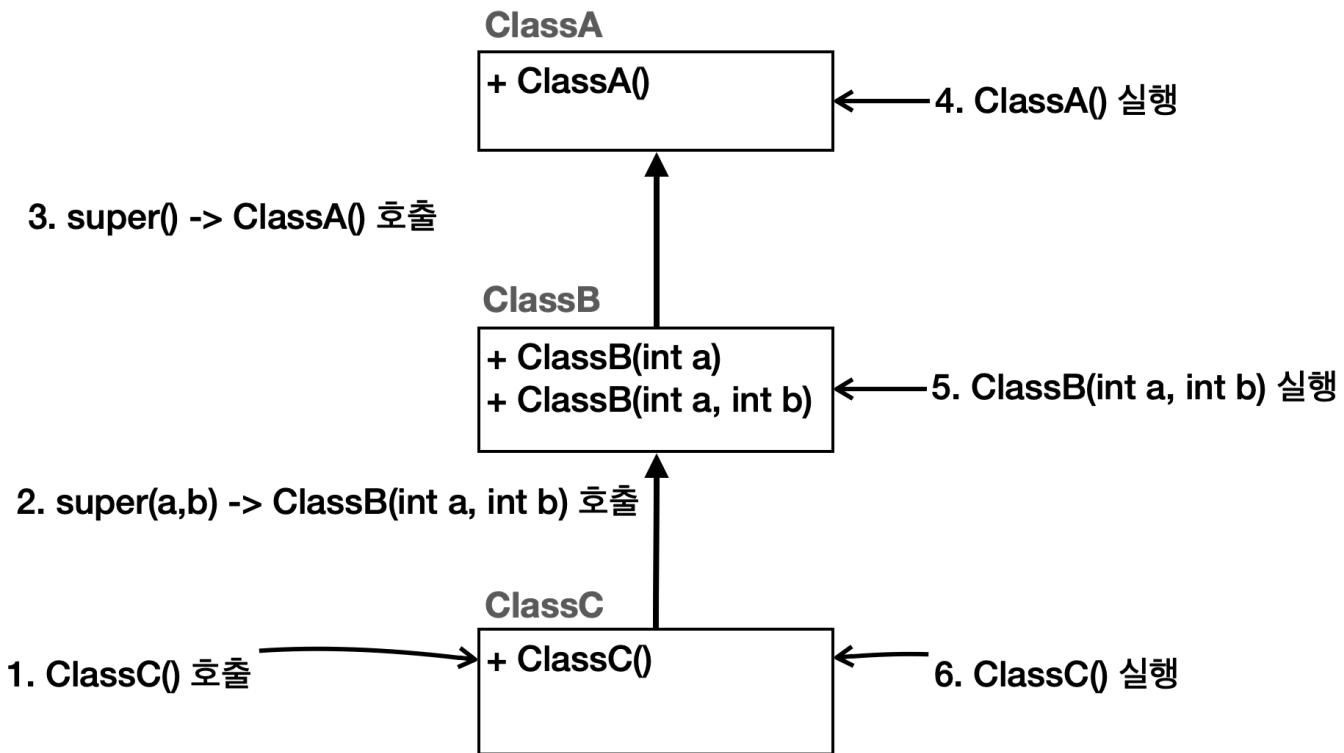
```
package extends1.super2;  
  
public class Super2Main {  
  
    public static void main(String[] args) {  
        ClassC classC = new ClassC();
```

```
    }  
}
```

실행 결과

```
ClassA 생성자  
ClassB 생성자 a=10 b=20  
ClassC 생성자
```

실행해보면 ClassA → ClassB → ClassC 순서로 실행된다. 생성자의 실행 순서가 결과적으로 최상위 부모부터 실행되어서 하나씩 아래로 내려오는 것이다. 따라서 초기화는 최상위 부모부터 이루어진다. 왜냐하면 자식 생성자의 첫 줄에서 부모의 생성자를 호출해야 하기 때문이다.



1~3까지의 과정

`new ClassC()`를 통해 ClassC 인스턴스를 생성한다. 이때 `ClassC()`의 생성자가 먼저 호출되는 것이 맞다. 하지만 `ClassC()`의 생성자는 가장 먼저 `super(..)`를 통해 `ClassB(...)`의 생성자를 호출한다. `ClassB()`의 생성자도 부모인 `ClassA()`의 생성자를 가장 먼저 호출한다.

4~6까지의 과정

- `ClassA()`의 생성자는 최상위 부모이다. 생성자 코드를 실행하면서 "ClassA 생성자"를 출력한다. `ClassA()` 생성자 호출이 끝나면 `ClassA()`를 호출한 `ClassB(...)` 생성자로 제어권이 돌아간다.
- `ClassB(...)` 생성자가 코드를 실행하면서 "ClassB 생성자 a=10 b=20"를 출력한다. 생성자 호출이 끝나면 `ClassB(...)`를 호출한 `ClassC()`의 생성자로 제어권이 돌아간다.
- `ClassC()`가 마지막으로 생성자 코드를 실행하면서 "ClassC 생성자"를 출력한다.

정리

- 상속 관계의 생성자 호출은 결과적으로 부모에서 자식 순서로 실행된다. 따라서 부모의 데이터를 먼저 초기화하고 그 다음에 자식의 데이터를 초기화한다.
- 상속 관계에서 자식 클래스의 생성자 첫줄에 반드시 `super(...)`을 호출해야 한다. 단 기본 생성자 (`super()`)인 경우 생략할 수 있다.

this(...)와 함께 사용

코드의 첫줄에 `this(...)`을 사용하더라도 반드시 한번은 `super(...)`을 호출해야 한다.

코드 변경

```
package extends1.super2;

public class ClassB extends ClassA {

    public ClassB(int a) {
        this(a, 0); //기본 생성자 생략 가능
        System.out.println("ClassB 생성자 a=" + a);
    }

    public ClassB(int a, int b) {
        super(); //기본 생성자 생략 가능
        System.out.println("ClassB 생성자 a=" + a + " b=" + b);
    }
}
```

```
package extends1.super2;

public class Super2Main {

    public static void main(String[] args) {
        //ClassC classC = new ClassC();
        ClassB classB = new ClassB(100);
    }
}
```

실행 결과

```
ClassA 생성자
ClassB 생성자 a=100 b=0
ClassB 생성자 a=100
```

문제와 풀이

문제: 상속 관계 상품

쇼핑몰의 판매 상품을 만들어보자.

- Book, Album, Movie 이렇게 3가지 상품을 클래스로 만들어야 한다.
- 코드 중복이 없게 상속 관계를 사용하자. 부모 클래스는 Item이라는 이름을 사용하면 된다.
- 공통 속성: name, price
 - Book: 저자(author), isbn(isbn)
 - Album: 아티스트(artist)
 - Movie: 감독(director), 배우(actor)

다음 코드와 실행결과를 참고해서 Item, Book, Album, Movie 클래스를 만들어보자.

```
package extends1.ex;

public class ShopMain {

    public static void main(String[] args) {
        Book book = new Book("JAVA", 10000, "han", "12345");
        Album album = new Album("앨범1", 15000, "seo");
        Movie movie = new Movie("영화1", 18000, "감독1", "배우1");

        book.print();
        album.print();
        movie.print();

        int sum = book.getPrice() + album.getPrice() + movie.getPrice();
        System.out.println("상품 가격의 합: " + sum);
    }
}
```

실행 결과

```
이름:JAVA, 가격:10000
- 저자:han, isbn:12345
이름:앨범1, 가격:15000
- 아티스트:seo
```

이름:영화1, 가격:18000

- 감독:감독1, 배우:배우1

상품 가격의 합: 43000

정답

```
package extends1.ex;

public class Item {
    private String name;
    private int price;

    public Item(String name, int price) {
        this.name = name;
        this.price = price;
    }

    public int getPrice() {
        return price;
    }

    public void print() {
        System.out.println("이름:" + name + ", 가격:" + price);
    }
}
```

```
package extends1.ex;
```

```
public class Book extends Item {
    private String author;
    private String isbn;

    public Book(String name, int price, String author, String isbn) {
        super(name, price);
        this.author = author;
        this.isbn = isbn;
    }

    @Override
    public void print() {
        super.print();
        System.out.println("- 저자:" + author + ", isbn:" + isbn);
    }
}
```

```
}
```

```
}
```

```
package extends1.ex;

public class Album extends Item {
    private String artist;

    public Album(String name, int price, String artist) {
        super(name, price);
        this.artist = artist;
    }

    @Override
    public void print() {
        super.print();
        System.out.println("- 아티스트:" + artist);
    }
}
```

```
package extends1.ex;

public class Movie extends Item {
    private String director;
    private String actor;

    public Movie(String name, int price, String director, String actor) {
        super(name, price);
        this.director = director;
        this.actor = actor;
    }

    @Override
    public void print() {
        super.print();
        System.out.println("- 감독:" + director + ", 배우:" + actor);
    }
}
```

정리

클래스와 메서드에 사용되는 final

클래스에 final

- 상속 끝!
- final로 선언된 클래스는 확장될 수 없다. 다른 클래스가 final로 선언된 클래스를 상속받을 수 없다.
- 예: public final class MyFinalClass { ... }

메서드에 final

- 오버라이딩 끝!
- final로 선언된 메서드는 오버라이드 될 수 없다. 상속받은 서브 클래스에서 이 메서드를 변경할 수 없다.
- 예: public final void myFinalMethod() { ... }

10. 다형성1

#1.인강/0.자바/2.자바-기본

- /다형성 시작
- /다형성과 캐스팅
- /캐스팅의 종류
- /다운캐스팅과 주의점
- /instanceof
- /다형성과 메서드 오버라이딩
- /정리

다형성 시작

객체지향 프로그래밍의 대표적인 특징으로는 캡슐화, 상속, 다형성이 있다. 그 중에서 다형성은 객체지향 프로그래밍의 꽃이라 불린다.

앞서 학습한 캡슐화나 상속은 직관적으로 이해하기 쉽다. 반면에 다형성은 제대로 이해하기도 어렵고, 잘 활용하기는 더 어렵다. 하지만 좋은 개발자가 되기 위해서는 다형성에 대한 이해가 필수다.

다형성(Polymorphism)은 이름 그대로 "다양한 형태", "여러 형태"를 뜻한다.

프로그래밍에서 다형성은 한 객체가 여러 타입의 객체로 취급될 수 있는 능력을 뜻한다. 보통 하나의 객체는 하나의 타입으로 고정되어 있다. 그런데 다형성을 사용하면 하나의 객체가 다른 타입으로 사용될 수 있다는 뜻이다. 지금은 이 내용을 이해하기보다는 참고만 해두자.

이제부터 본격적으로 다형성을 학습해보자.

다형성을 이해하기 위해서는 크게 2가지 핵심 이론을 알아야 한다.

- **다형적 참조**
- **메서드 오버라이딩**

먼저 다형적 참조라 불리는 개념에 대해 알아보자.

다형적 참조

다형적 참조를 이해하기 위해 다음과 같은 간단한 상속 관계를 코드로 만들어보자.

Parent

```
+ parentMethod()
```



```
+ childMethod()
```

Child

부모와 자식이 있고, 각각 다른 메서드를 가진다.

```
package poly.basic;

public class Parent {

    public void parentMethod() {
        System.out.println("Parent.parentMethod");
    }
}
```

```
package poly.basic;

public class Child extends Parent {

    public void childMethod() {
        System.out.println("Child.childMethod");
    }
}
```

```
package poly.basic;
```

```

/**
 * 다형적 참조: 부모는 자식을 품을 수 있다.
 */
public class PolyMain {
    public static void main(String[] args) {
        //부모 변수가 부모 인스턴스 참조
        System.out.println("Parent -> Parent");
        Parent parent = new Parent();
        parent.parentMethod();

        //자식 변수가 자식 인스턴스 참조
        System.out.println("Child -> Child");
        Child child = new Child();
        child.parentMethod();
        child.childMethod();

        //부모 변수가 자식 인스턴스 참조(다형적 참조)
        System.out.println("Parent -> Child");
        Parent poly = new Child();
        poly.parentMethod();

        //Child child1 = new Parent(); 자식은 부모를 담을 수 없다.

        //자식의 기능은 호출할 수 없다. 컴파일 오류 발생
        //poly.childMethod();
    }
}

```

실행 결과

```

Parent -> Parent
Parent.parentMethod

Child -> Child
Parent.parentMethod
Child.childMethod

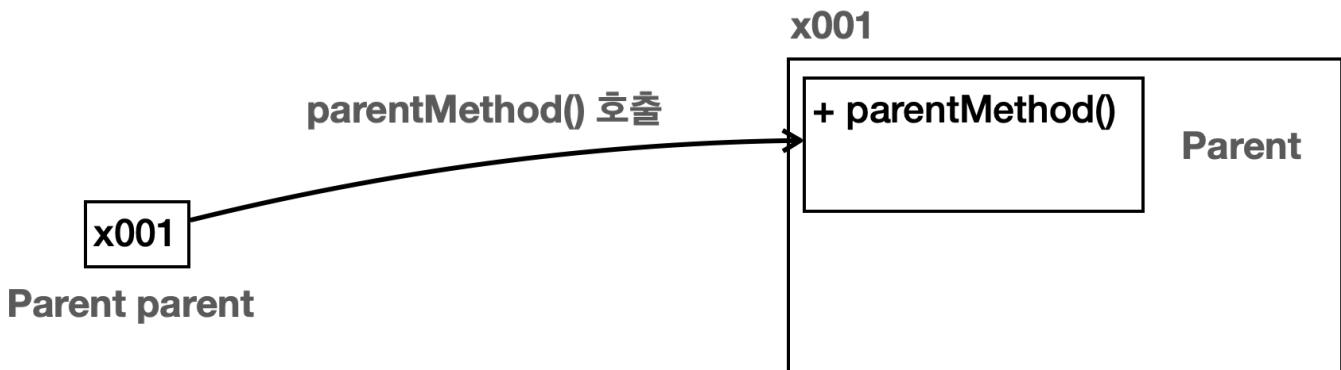
Parent -> Child
Parent.parentMethod

```

그림을 통해 코드를 하나씩 분석해보자.

부모 타입의 변수가 부모 인스턴스 참조

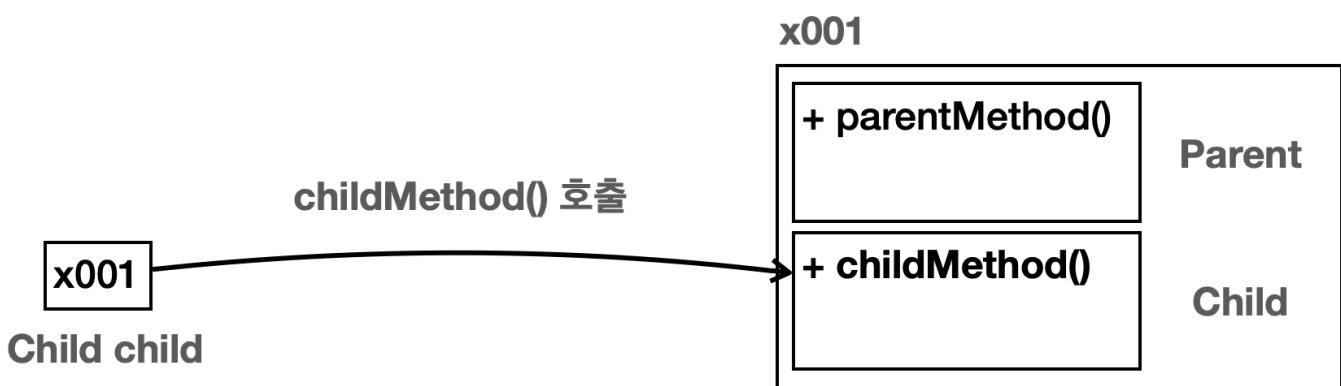
Parent → Parent: parent.parentMethod()



- 부모 타입의 변수가 부모 인스턴스를 참조한다.
- Parent parent = new Parent()
- Parent 인스턴스를 만들었다. 이 경우 부모 타입인 Parent 를 생성했기 때문에 메모리 상에 Parent 만 생성된다.(자식은 생성되지 않는다.)
- 생성된 참조값을 Parent 타입의 변수인 parent 에 담아둔다.
- parent.parentMethod() 를 호출하면 인스턴스의 Parent 클래스에 있는 parentMethod() 가 호출된다.

자식 타입의 변수가 자식 인스턴스 참조

Child → Child: child.childMethod()

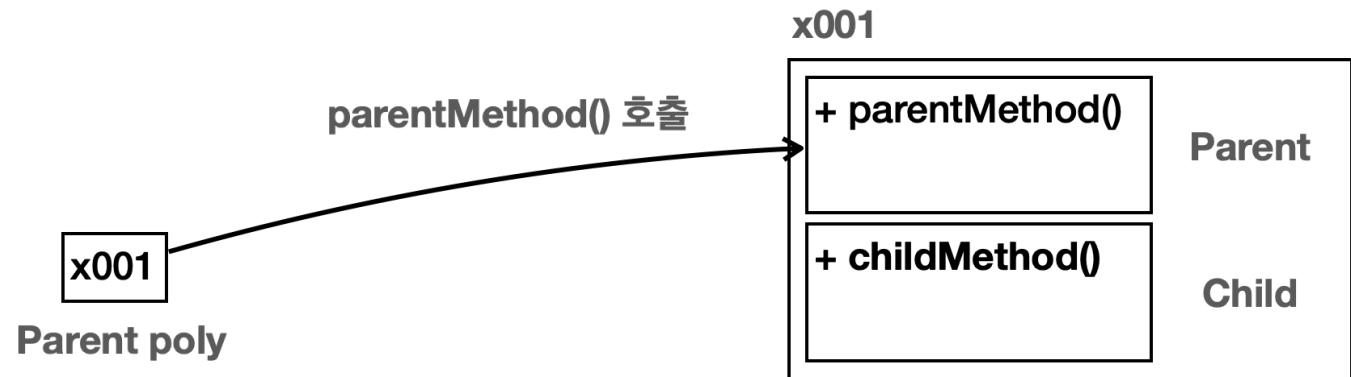


- 자식 타입의 변수가 자식 인스턴스를 참조한다.
- Child child = new Child()
- Child 인스턴스를 만들었다. 이 경우 자식 타입인 Child 를 생성했기 때문에 메모리 상에 Child 와 Parent 가 모두 생성된다.
- 생성된 참조값을 Child 타입의 변수인 child 에 담아둔다.
- child.childMethod() 를 호출하면 인스턴스의 Child 클래스에 있는 childMethod() 가 호출된다.

여기까지는 지금까지 배운 내용이므로 이해하는데 어려움은 없을 것이다. 이제부터가 중요하다.

다형적 참조: 부모 타입의 변수가 자식 인스턴스 참조

Parent → Child: poly.parentMethod()



- 부모 타입의 변수가 자식 인스턴스를 참조한다.
- Parent poly = new Child()
- Child 인스턴스를 만들었다. 이 경우 자식 타입인 Child를 생성했기 때문에 메모리 상에 Child와 Parent가 모두 생성된다.
- 생성된 참조값을 Parent 타입의 변수인 poly에 담아둔다.

부모는 자식을 담을 수 있다.

- 부모 타입은 자식 타입을 담을 수 있다.
- Parent poly는 부모 타입이다. new Child()를 통해 생성된 결과는 Child 타입이다. 자바에서 부모 타입은 자식 타입을 담을 수 있다!
 - Parent poly = new Child(): 성공
- 반대로 자식 타입은 부모 타입을 담을 수 없다.
 - Child child1 = new Parent(): 컴파일 오류 발생

다형적 참조

지금까지 학습한 내용을 떠올려보면 항상 같은 타입에 참조를 대입했다. 그래서 보통 한 가지 형태만 참조할 수 있다.

- Parent parent = new Parent()
- Child child = new Child()

그런데 Parent 타입의 변수는 다음과 같이 자신인 Parent는 물론이고, 자식 타입까지 참조할 수 있다. 만약 손자가 있다면 손자도 그 하위 타입도 참조할 수 있다.

- Parent poly = new Parent()
- Parent poly = new Child()
- Parent poly = new Grandson(): Child 하위에 손자가 있다면 가능

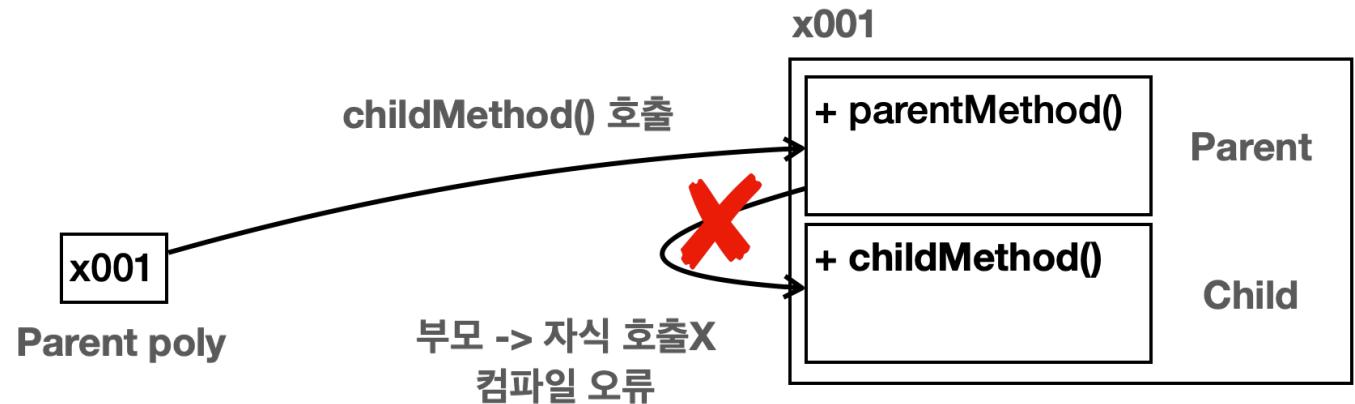
자바에서 부모 타입은 자신은 물론이고, 자신을 기준으로 모든 자식 타입을 참조할 수 있다. 이것이 바로 다양한 형태를 참조할 수 있다고 해서 다형적 참조라 한다.

다형적 참조와 인스턴스 실행

앞의 그림을 참고하자. `poly.parentMethod()` 를 호출하면 먼저 참조값을 사용해서 인스턴스를 찾는다. 그리고 다음으로 인스턴스 안에서 실행할 타입도 찾아야 한다. `poly` 는 `Parent` 타입이다. 따라서 `Parent` 클래스부터 시작해서 필요한 기능을 찾는다. 인스턴스의 `Parent` 클래스에 `parentMethod()` 가 있다. 따라서 해당 메서드가 호출된다.

다형적 참조의 한계

`Parent → Child: poly.childMethod()`



`Parent poly = new Child()` 이렇게 자식을 참조한 상황에서 `poly` 가 자식 타입인 `Child`에 있는 `childMethod()` 를 호출하면 어떻게 될까?

`poly.childMethod()` 를 실행하면 먼저 참조값을 통해 인스턴스를 찾는다. 그리고 다음으로 인스턴스 안에서 실행할 타입을 찾아야 한다. 호출자인 `poly` 는 `Parent` 타입이다. 따라서 `Parent` 클래스부터 시작해서 필요한 기능을 찾는다. 그런데 상속 관계는 부모 방향으로 찾아 올라갈 수는 있지만 자식 방향으로 찾아 내려갈 수는 없다. `Parent` 는 부모 타입이고 상위에 부모가 없다. 따라서 `childMethod()` 를 찾을 수 없으므로 컴파일 오류가 발생한다.

이런 경우 `childMethod()` 를 호출하고 싶으면 어떻게 해야 할까? 바로 캐스팅이 필요하다.

다형적 참조의 핵심은 부모는 자식을 품을 수 있다는 것이다.

그런데 이런 다형적 참조가 왜 필요하지? 라는 의문이 들 수 있다. 이 부분은 다형성의 다른 이론들도 함께 알아야 이해할 수 있다. 지금은 우선 다형성의 문법과 이론을 익히는데 집중하자.

다형성과 캐스팅

`Parent poly = new Child()` 와 같이 부모 타입의 변수를 사용하게 되면 `poly.childMethod()` 와 같이 자

식 타입에 있는 기능은 호출할 수 없다.

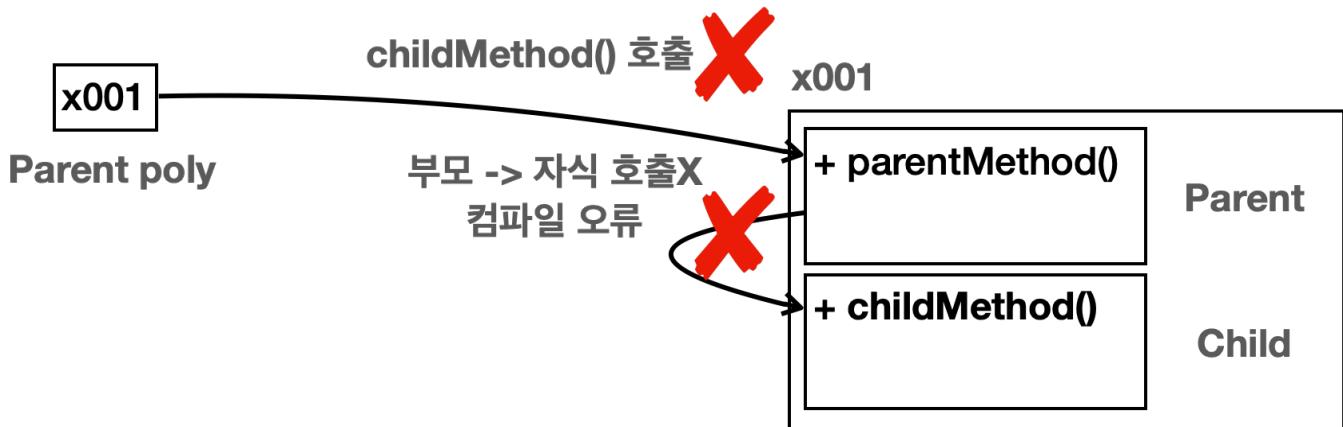
```
package poly.basic;

public class CastingMain1 {
    public static void main(String[] args) {
        //부모 변수가 자식 인스턴스 참조(다형적 참조)
        Parent poly = new Child();
        //단 자식의 기능은 호출할 수 없다. 컴파일 오류 발생
        //poly.childMethod();

        //다운캐스팅(부모 타입 -> 자식 타입)
        Child child = (Child) poly;
        child.childMethod();
    }
}
```

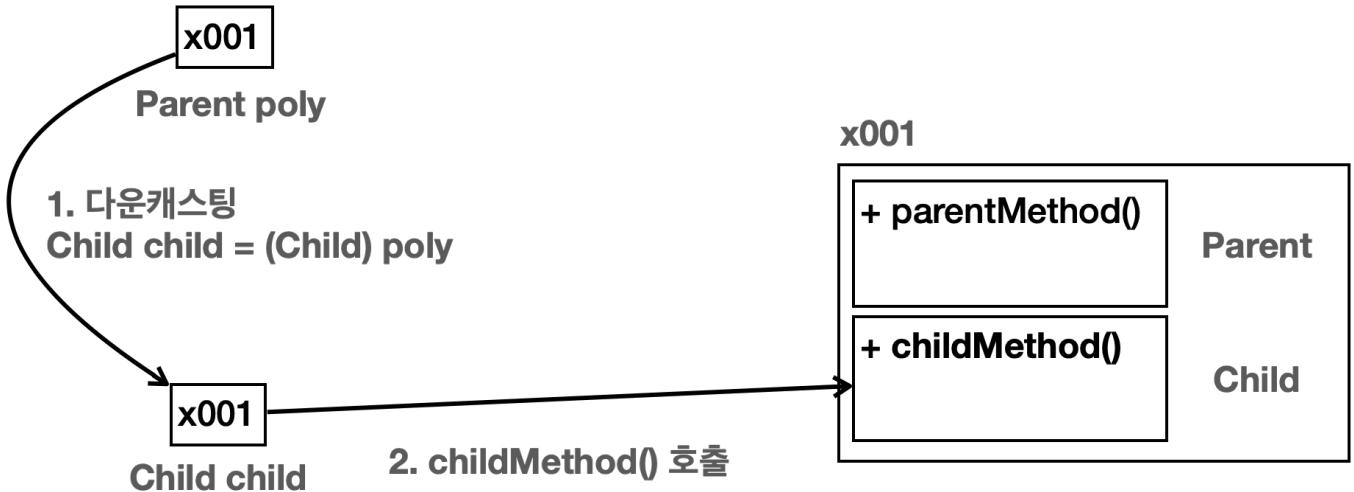
실행 결과

```
Child.childMethod
```



- `poly.childMethod()`를 호출하면 먼저 참조값을 사용해서 인스턴스를 찾는다.
- 인스턴스 안에서 사용할 타입을 찾아야 한다. `poly`는 `Parent` 타입이다.
- `Parent`는 최상위 부모이다. 상속 관계는 부모로만 찾아서 올라갈 수 있다. `childMethod()`는 자식 타입에 있으므로 호출할 수 없다. 따라서 컴파일 오류가 발생한다.

다운캐스팅



이럴때는 어떻게 하면 될까? 호출하는 타입을 자식인 **Child** 타입으로 변경하면 인스턴스의 **Child**에 있는 **childMethod()**를 호출할 수 있다. 하지만 다음과 같은 문제에 봉착한다.

부모는 자식을 담을 수 있지만 자식은 부모를 담을 수 없다.

- **Parent parent = new Child();**: 부모는 자식을 담을 수 있다.
- **Parent parent = child //Child child** 변수: 부모는 자식을 담을 수 있다.

반면에 다음과 같이 자식은 부모를 담을 수 없다.

```
Child child = poly //Parent poly 변수
```

부모 타입을 사용하는 변수를 자식 타입에 대입하려고 하면 컴파일 오류가 발생한다. 자식은 부모를 담을 수 없다.

이때는 다운캐스팅이라는 기능을 사용해서 부모 타입을 잠깐 자식 타입으로 변경하면 된다.

다음 코드를 분석해보자.

```
Child child = (Child) poly //Parent poly
```

(타입) 처럼 팔호와 그 사이에 타입을 지정하면 참조 대상을 특정 타입으로 변경할 수 있다. 이렇게 특정 타입으로 변경하는 것을 캐스팅이라 한다.

오른쪽에 있는 **(Child) poly** 코드를 먼저 보자. **poly**는 **Parent** 타입이다. 이 타입을 **(Child)**를 사용해서 일시적으로 자식 타입인 **Child** 타입으로 변경한다. 그리고 나서 왼쪽에 있는 **Child child**에 대입한다.

실행 순서

```
Child child = (Child) poly //다운캐스팅을 통해 부모타입을 자식 타입으로 변환한 다음에 대입 시도
Child child = (Child) x001 //참조값을 읽은 다음 자식 타입으로 지정
Child child = x001 //최종 결과
```

참고로 캐스팅을 한다고 해서 **Parent poly**의 타입이 변하는 것은 아니다. 해당 참조값을 꺼내고 꺼낸 참조값이 **Child** 타입이 되는 것이다. 따라서 **poly**의 타입은 **Parent**로 기존과 같이 유지된다.

캐스팅

- 업캐스팅(upcasting): 부모 타입으로 변경
- 다운캐스팅(downcasting): 자식 타입으로 변경

캐스팅 용어

"캐스팅"은 영어 단어 "cast"에서 유래되었다. "cast"는 금속이나 다른 물질을 녹여서 특정한 형태나 모양으로 만드는 과정을 의미한다.

`Child child = (Child) poly` 경우 `Parent poly`라는 부모 타입을 `Child`라는 자식 타입으로 변경했다. 부모 타입을 자식 타입으로 변경하는 것을 다운캐스팅이라 한다. 반대로 부모 타입으로 변경하는 것은 업캐스팅이라 한다.

다운캐스팅과 실행

```
//다운캐스팅(부모 타입 -> 자식 타입)
Child child = (Child) poly;
child.childMethod();
```

다운캐스팅 덕분에 `child.childMethod()`를 호출할 수 있게 되었다. `childMethod()`를 호출하기 위해 해당 인스턴스를 찾아간 다음 `Child` 타입을 찾는다. `Child` 타입에는 `childMethod()`가 있으므로 해당 기능을 호출할 수 있다. 앞의 그림을 참고하자.

캐스팅의 종류

자식 타입의 기능을 사용하려면 다음과 같이 다운캐스팅 결과를 변수에 담아두고 이후에 기능을 사용하면 된다.

```
Child child = (Child) poly
child.childMethod();
```

하지만 다운캐스팅 결과를 변수에 담아두는 과정이 번거롭다. 이런 과정 없이 일시적으로 다운캐스팅을 해서 인스턴스에 있는 하위 클래스의 기능을 바로 호출할 수 있다.

다음 코드를 보자.

일시적 다운 캐스팅

```
package poly.basic;

public class CastingMain2 {
    public static void main(String[] args) {
        //부모 변수가 자식 인스턴스 참조(다형적 참조)
```

```

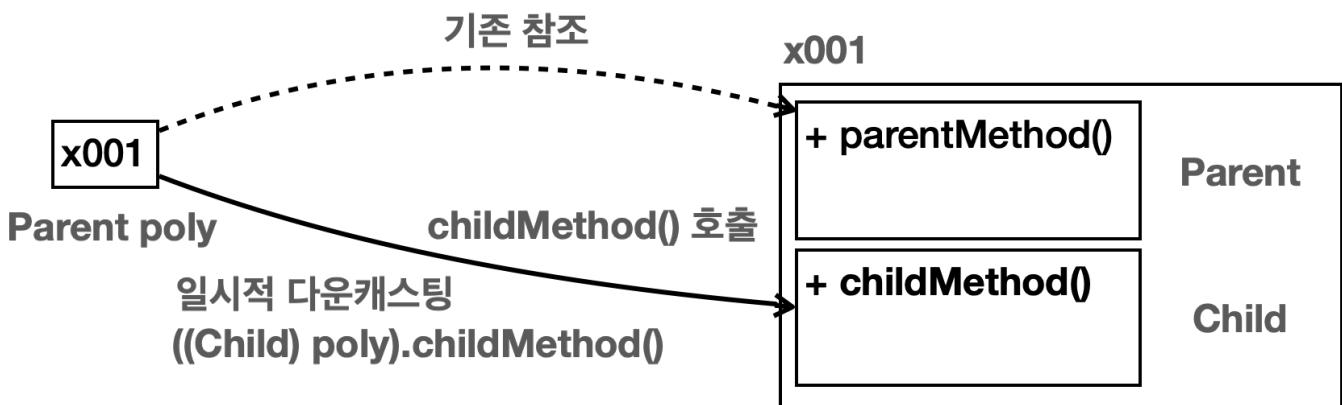
Parent poly = new Child();
//단 자식의 기능은 호출할 수 없다. 컴파일 오류 발생
//poly.childMethod();

//일시적 다운캐스팅 - 해당 메서드를 호출하는 순간만 다운캐스팅
((Child) poly).childMethod();
}
}

```

실행 결과

Child.childMethod



`((Child) poly).childMethod()`

`poly`는 `Parent` 타입이다. 그런데 이 코드를 실행하면 `Parent` 타입을 임시로 `Child`로 변경한다. 그리고 메서드를 호출할 때 `Child` 타입에서 찾아서 실행한다.

정확히는 `poly`가 `Child` 타입으로 바뀌는 것은 아니다.

```

((Child) poly).childMethod() //다운캐스팅을 통해 부모타입을 자식 타입으로 변환 후 기능 호출
((Child) x001).childMethod() //참조값을 읽은 다음 자식 타입으로 다운캐스팅

```

참고로 캐스팅을 한다고 해서 `Parent poly`의 타입이 변하는 것은 아니다. 해당 참조값을 꺼내고 꺼낸 참조값이 `Child` 타입이 되는 것이다. 따라서 `poly`의 타입은 `Parent`로 그대로 유지된다.

이렇게 일시적 다운캐스팅을 사용하면 별도의 변수 없이 인스턴스의 자식 타입의 기능을 사용할 수 있다.

업캐스팅

다운캐스팅과 반대로 현재 타입을 부모 타입으로 변경하는 것을 업캐스팅이라 한다.

다음 코드를 보자.

```
package poly.basic;

//upcasting vs downcasting
public class CastingMain3 {
    public static void main(String[] args) {
        Child child = new Child();
        Parent parent1 = (Parent) child; //업캐스팅은 생략 가능, 생략 권장
        Parent parent2 = child; //업캐스팅 생략

        parent1.parentMethod();
        parent2.parentMethod();
    }
}
```

실행 결과

```
Parent.parentMethod
Parent.parentMethod
```

다음 코드를 보자.

```
Parent parent1 = (Parent) child;
Child 타입을 Parent 타입에 대입해야 한다. 따라서 타입을 변환하는 캐스팅이 필요하다.
```

그런데 부모 타입으로 변환하는 경우에는 다음과 같이 캐스팅 코드인 (타입) 를 생략할 수 있다.

```
Parent parent2 = child
Parent parent2 = new Child()
```

업캐스팅은 생략할 수 있다. 다운캐스팅은 생략할 수 없다. 참고로 업캐스팅은 매우 자주 사용하기 때문에 생략을 권장한다.

자바에서 부모는 자식을 담을 수 있다. 하지만 그 반대는 안된다. (꼭 필요하다면 다운캐스팅을 해야 한다.)

업캐스팅은 생략해도 되고, 다운캐스팅은 왜 개발자가 직접 명시적으로 캐스팅을 해야 할까?

다운캐스팅과 주의점

다운캐스팅은 잘못하면 심각한 런타임 오류가 발생할 수 있다.

다음 코드를 통해 다운캐스팅에서 발생할 수 있는 문제를 확인해보자.

```
package poly.basic;

//다운캐스팅을 자동으로 하지 않는 이유
public class CastingMain4 {
    public static void main(String[] args) {
        Parent parent1 = new Child();
        Child child1 = (Child) parent1;
        child1.childMethod(); //문제 없음

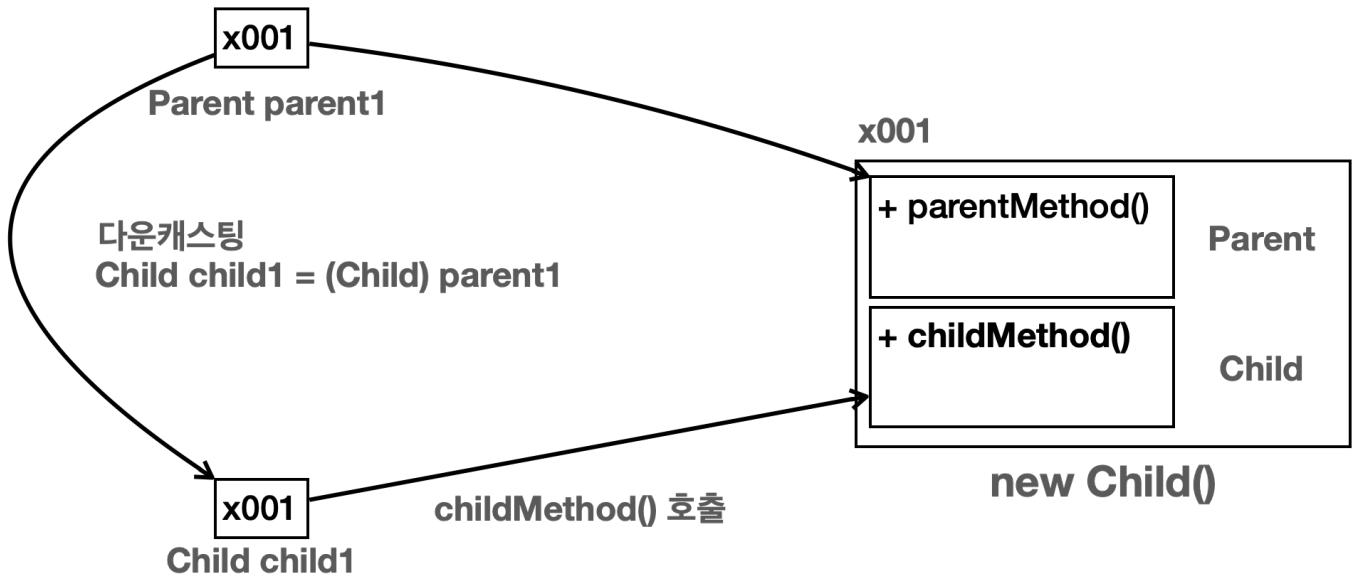
        Parent parent2 = new Parent();
        Child child2 = (Child) parent2; //런타임 오류 - ClassCastException
        child2.childMethod(); //실행 불가
    }
}
```

실행 결과

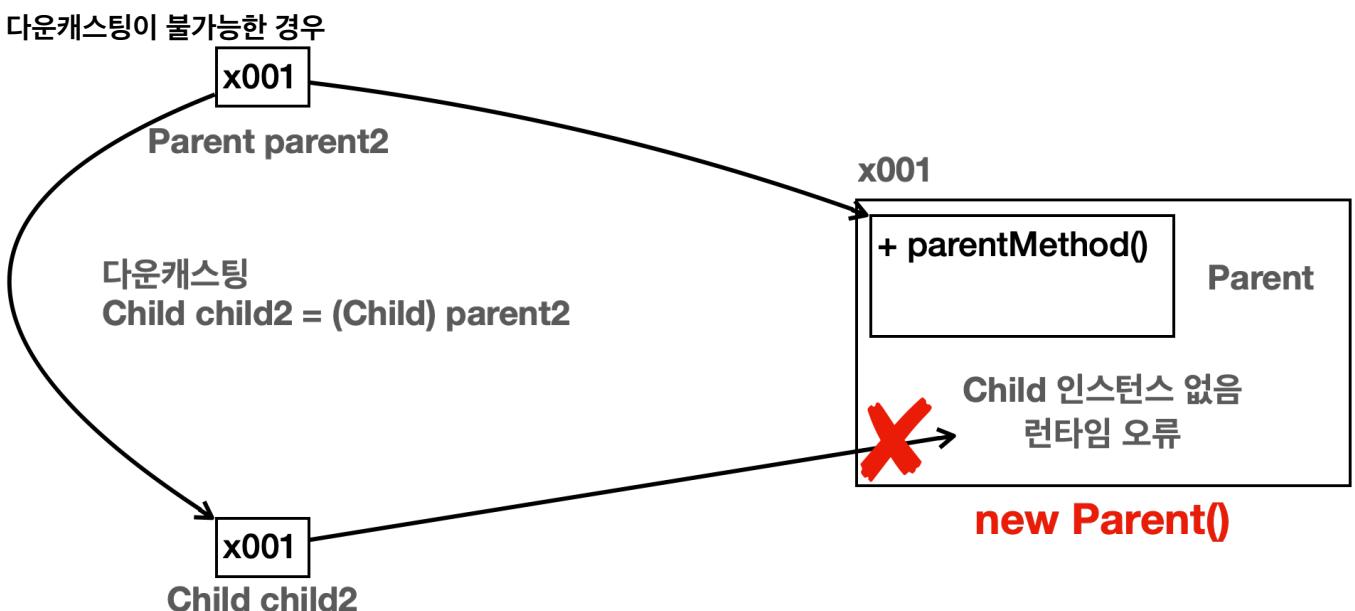
```
Child.childMethod
Exception in thread "main" java.lang.ClassCastException: class poly.basic.Parent
cannot be cast to class poly.basic.Child (poly.basic.Parent and poly.basic.Child
are in unnamed module of loader 'app')
at poly.basic.CastingMain4.main(CastingMain4.java:11)
```

실행 결과를 보면 `child1.childMethod()`는 잘 호출되었지만, `child2.childMethod()`는 실행되지 못하고, 그 전에 오류가 발생했다.

다운캐스팅이 가능한 경우



예제의 `parent1`의 경우 다운캐스팅을 해도 문제가 되지 않는다.



예제의 `parent2`를 다운캐스팅하면 `ClassCastException` 이라는 심각한 런타임 오류가 발생한다.

이 코드를 자세히 알아보자.

```
Parent parent2 = new Parent()
```

먼저 `new Parent()`로 부모 타입으로 객체를 생성한다. 따라서 메모리 상에 자식 타입은 전혀 존재하지 않는다. 생성 결과를 `parent2`에 담아둔다. 이 경우 같은 타입이므로 여기서는 문제가 발생하지 않는다.

```
Child child2 = (Child) parent2
```

다음으로 `parent2`를 `Child` 타입으로 다운캐스팅한다. 그런데 `parent2`는 `Parent`로 생성이 되었다. 따라서 메모리 상에 `Child` 자체가 존재하지 않는다. `Child` 자체를 사용할 수 없는 것이다.

자바에서는 이렇게 사용할 수 없는 타입으로 다운캐스팅하는 경우에 `ClassCastException`이라는 예외를 발생시킨다. 예외가 발생하면 다음 동작이 실행되지 않고, 프로그램이 종료된다. 따라서 `child2.childMethod()` 코드 자체가 실행되지 않는다.

업캐스팅이 안전하고 다운캐스팅이 위험한 이유

업캐스팅의 경우 이런 문제가 절대로 발생하지 않는다. 왜냐하면 객체를 생성하면 해당 타입의 상위 부모 타입은 모두 함께 생성된다! 따라서 위로만 타입을 변경하는 업캐스팅은 메모리 상에 인스턴스가 모두 존재하기 때문에 항상 안전하다. 따라서 캐스팅을 생략할 수 있다.

반면에 다운캐스팅의 경우 인스턴스에 존재하지 않는 하위 타입으로 캐스팅하는 문제가 발생할 수 있다. 왜냐하면 객체를 생성하면 부모 타입은 모두 함께 생성되지만 자식 타입은 생성되지 않는다. 따라서 개발자가 이런 문제를 인지하고 사용해야 한다는 의미로 명시적으로 캐스팅을 해주어야 한다.

그림으로 설명 - 업캐스팅

Class A

+ methodA();

A a = new C()

x001

Class B

+ methodB();

B b = new C()

A

B

C

Class C

+ methodC();

C c = new C()

new C()

클래스 A, B, C는 상속 관계다.

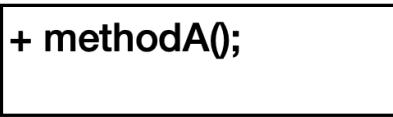
`new C()`로 인스턴스를 생성하면 인스턴스 내부에 자신과 부모인 A, B, C가 모두 생성된다. 따라서 C의 부모 타입인 A, B, C 모두 C 인스턴스를 참조할 수 있다. 상위로 올라가는 업캐스팅은 인스턴스 내부에 부모가 모두 생성되기 때문에 문제가 발생하지 않는다.

- `A a = new C()`: A로 업캐스팅
- `B b = new C()`: B로 업캐스팅

- `C c = new C()` : 자신과 같은 타입

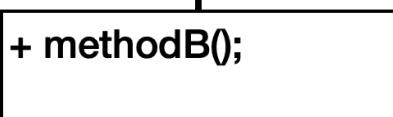
그림으로 설명 - 다운캐스팅

Class A



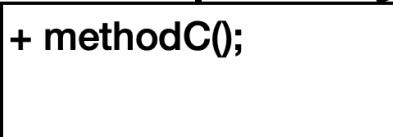
`A a = new B()`

Class B



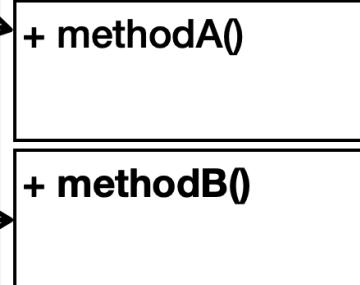
`B b = new B()`

Class C



`C c = (C) new B()`

x001



`new B()`

`new B()`로 인스턴스를 생성하면 인스턴스 내부에 자신과 부모인 A, B가 생성된다. 따라서 B의 부모 타입인 A, B 모두 B 인스턴스를 참조할 수 있다. 상위로 올라가는 업캐스팅은 인스턴스 내부에 부모가 모두 생성되기 때문에 문제가 발생하지 않는다. 하지만 객체를 생성할 때 하위 자식은 생성되지 않기 때문에 하위로 내려가는 다운캐스팅은 인스턴스 내부에 없는 부분을 선택하는 문제가 발생할 수 있다.

- `A a = new B()` : A로 업캐스팅
- `B b = new B()` : 자신과 같은 타입
- `C c = new B()` : 하위 타입은 대입할 수 없음, **컴파일 오류**
- `C c = (C) new B()` : 하위 타입으로 강제 다운캐스팅, 하지만 B 인스턴스에 C와 관련된 부분이 없으므로 잘못된 캐스팅, `ClassCastException` 런타임 오류 발생

컴파일 오류 vs 런타임 오류

컴파일 오류는 변수명 오타, 잘못된 클래스 이름 사용등 자바 프로그램을 실행하기 전에 발생하는 오류이다. 이런 오류는 IDE에서 즉시 확인할 수 있기 때문에 안전하고 좋은 오류이다.

반면에 런타임 오류는 이름 그대로 프로그램이 실행되고 있는 시점에 발생하는 오류이다. 런타임 오류는 매우 안좋은 오류이다. 왜냐하면 보통 고객이 해당 프로그램을 실행하는 도중에 발생하기 때문이다.

instanceof

다형성에서 참조형 변수는 이를 그대로 다양한 자식을 대상으로 참조할 수 있다. 그런데 참조하는 대상이 다양하기 때문에 어떤 인스턴스를 참조하고 있는지 확인하려면 어떻게 해야 할까?

```
Parent parent1 = new Parent()  
Parent parent2 = new Child()
```

여기서 `Parent`는 자신과 같은 `Parent`의 인스턴스도 참조할 수 있고, 자식 타입인 `Child`의 인스턴스도 참조할 수 있다. 이때 `parent1`, `parent2` 변수가 참조하는 **인스턴스의 타입을 확인하고** 싶다면 `instanceof` 키워드를 사용하면 된다.

예제를 보자.

```
package poly.basic;  
  
public class CastingMain5 {  
  
    public static void main(String[] args) {  
        Parent parent1 = new Parent();  
        System.out.println("parent1 호출");  
        call(parent1);  
  
        Parent parent2 = new Child();  
        System.out.println("parent2 호출");  
        call(parent2);  
    }  
  
    private static void call(Parent parent) {  
        parent.parentMethod();  
        if (parent instanceof Child) {  
            System.out.println("Child 인스턴스 맞음");  
            Child child = (Child) parent;  
            child.childMethod();  
        }  
    }  
}
```

실행 결과

```
parent1 호출  
Parent.parentMethod  
parent2 호출  
Parent.parentMethod  
Child 인스턴스 맞음  
Child.childMethod
```

`call(Parent parent)` 메서드를 보자.

이 메서드는 매개변수로 넘어온 `parent` 가 참조하는 타입에 따라서 다른 명령을 수행한다.

참고로 지금처럼 다운캐스팅을 수행하기 전에는 먼저 `instanceof` 를 사용해서 원하는 타입으로 변경이 가능한지 확인한 다음에 다운캐스팅을 수행하는 것이 안전하다.

해당 메서드를 처음 호출할 때 `parent` 는 `Parent` 의 인스턴스를 참조한다.

```
parent instanceof Child //parent는 Parent의 인스턴스  
new Parent() instanceof Child //false
```

`parent` 는 `Parent` 의 인스턴스를 참조하므로 `false` 를 반환한다.

해당 메서드를 다음으로 호출할 때 `parent` 는 `Child` 의 인스턴스를 참조한다.

```
parent instanceof Child //parent는 Child의 인스턴스  
new Child() instanceof Child //true  
parent는 Child의 인스턴스를 참조하므로 true를 반환한다.
```

참고로 `instanceof` 키워드는 오른쪽 대상의 자식 타입을 왼쪽에서 참조하는 경우에도 `true` 를 반환한다.

```
parent instanceof Parent //parent는 Child의 인스턴스
```

```
new Parent() instanceof Parent //parent가 Parent의 인스턴스를 참조하는 경우: true  
new Child() instanceof Parent //parent가 Child의 인스턴스를 참조하는 경우: true
```

쉽게 이야기해서 오른쪽에 있는 타입에 왼쪽에 있는 인스턴스의 타입이 들어갈 수 있는지 대입해보면 된다. 대입이 가능하면 `true`, 불가능하면 `false` 가 된다.

```
new Parent() instanceof Parent  
Parent p = new Parent() //같은 타입 true  
  
new Child() instanceof Parent  
Parent p = new Child() //부모는 자식을 담을 수 있다. true  
  
new Parent() instanceof Child
```

```
Child c = new Parent() //자식은 부모를 담을 수 없다. false  
  
new Child() instanceof Child  
Child c = new Child() //같은 타입 true
```

자바 16 - Pattern Matching for instanceof

자바 16부터는 `instanceof` 를 사용하면서 동시에 변수를 선언할 수 있다.

다음 코드를 참고하자.

```
package poly.basic;  
  
public class CastingMain6 {  
  
    public static void main(String[] args) {  
        Parent parent1 = new Parent();  
        System.out.println("parent1 호출");  
        call(parent1);  
  
        Parent parent2 = new Child();  
        System.out.println("parent2 호출");  
        call(parent2);  
    }  
  
    private static void call(Parent parent) {  
        parent.parentMethod();  
        //Child 인스턴스인 경우 childMethod() 실행  
        if (parent instanceof Child child) {  
            System.out.println("Child 인스턴스 맞음");  
            child.childMethod();  
        }  
    }  
}
```

실행 결과

```
parent1 호출  
Parent.parentMethod  
parent2 호출  
Parent.parentMethod  
Child 인스턴스 맞음
```

Child.childMethod

덕분에 인스턴스가 맞는 경우 직접 다운캐스팅 하는 코드를 생략할 수 있다.

다형성과 메서드 오버라이딩

다형성을 이루는 또 하나의 중요한 핵심 이론은 바로 메서드 오버라이딩이다.

메서드 오버라이딩에서 꼭 기억해야 할 점은 **오버라이딩 된 메서드가 항상 우선권을 가진다**는 점이다.

그래서 이름도 기존 기능을 덮어 새로운 기능을 재정의 한다는 뜻의 오버라이딩이다.

앞서 메서드 오버라이딩을 학습했지만 지금까지 학습한 메서드 오버라이딩은 반쪽짜리다. 메서드 오버라이딩의 진짜 힘은 다형성과 함께 사용할 때 나타난다. 다음 코드를 통해 다형성과 메서드 오버라이딩을 알아보자.

Parent

```
+ value = "parent"  
+ method()
```



```
+ value = "child"  
+ method(): 오버라이딩
```

Child

- Parent, Child 모두 value라는 같은 멤버 변수를 가지고 있다.
 - 멤버 변수는 오버라이딩 되지 않는다.
- Parent, Child 모두 method()라는 같은 메서드를 가지고 있다. Child에서 메서드를 오버라이딩 했다.
 - 메서드는 오버라이딩 된다.

```
package poly.overriding;

public class Parent {

    public String value = "parent";

    public void method() {
        System.out.println("Parent.method");
    }
}
```

```
package poly.overriding;

public class Child extends Parent {

    public String value = "child";

    @Override
    public void method() {
        System.out.println("Child.method");
    }
}
```

- Child에서 Parent의 method()를 재정의(오버라이딩) 했다.

```
package poly.overriding;

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

        //자식 변수가 자식 인스턴스 참조
        Child child = new Child();
        System.out.println("Child -> Child");
        System.out.println("value = " + child.value);
        child.method();

        //부모 변수가 부모 인스턴스 참조
        Parent parent = new Parent();
        System.out.println("Parent -> Parent");
        System.out.println("value = " + parent.value);
        parent.method();

        //부모 변수가 자식 인스턴스 참조(다형적 참조)
    }
}
```

```

        Parent poly = new Child();
        System.out.println("Parent -> Child");
        System.out.println("value = " + poly.value); //변수는 오버라이딩X
        poly.method(); //메서드 오버라이딩!
    }
}

```

실행 결과

Child -> Child

value = child

Child.method

Parent -> Parent

value = parent

Parent.method

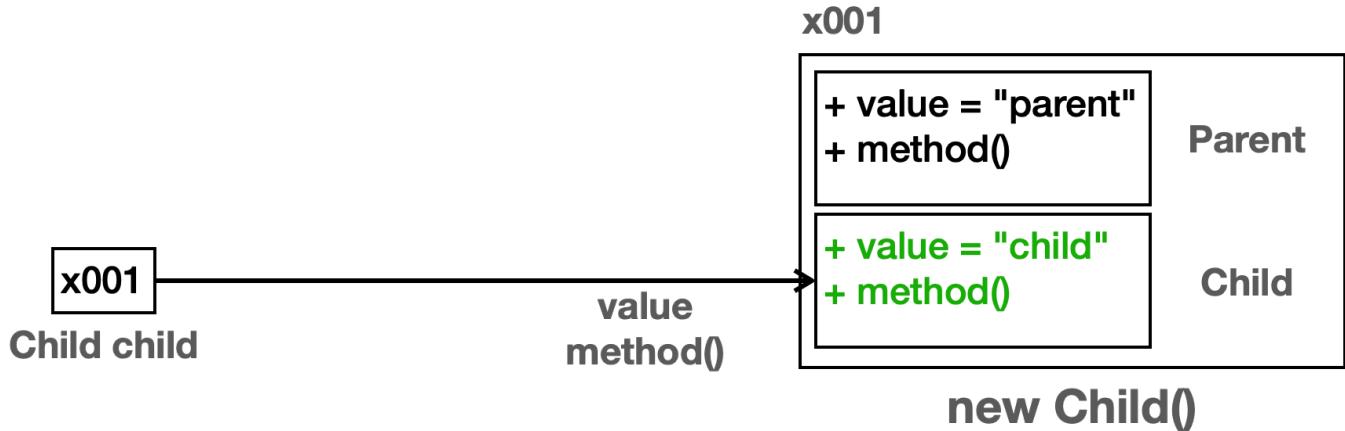
Parent -> Child

value = parent

Child.method

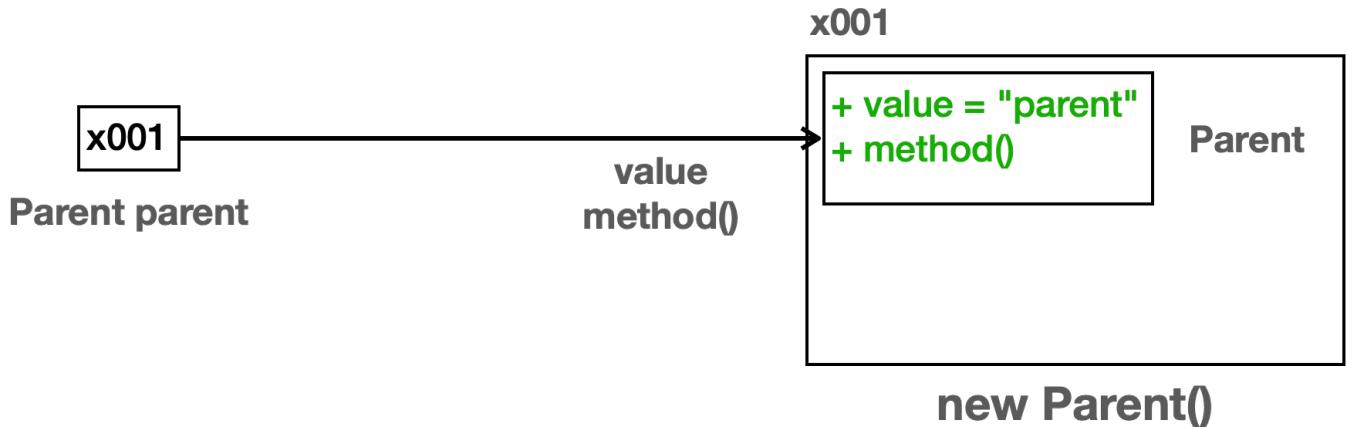
그림을 통해 코드를 분석해보자.

Child → Child



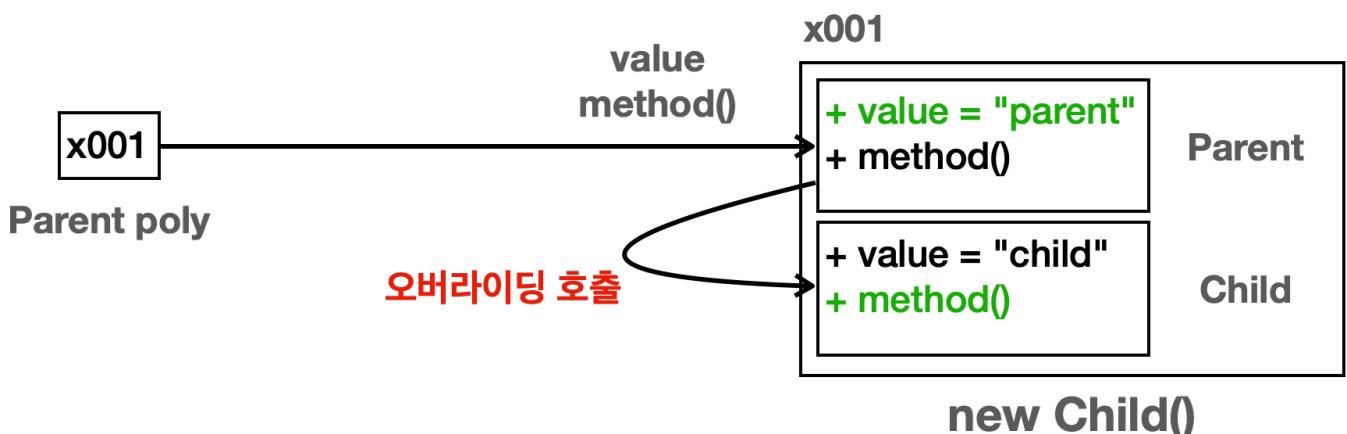
- `child` 변수는 `Child` 타입이다. 따라서 `child.value`, `child.method()` 를 호출하면 인스턴스의 `Child` 타입에서 기능을 찾아서 실행한다.

Parent → Parent



- `parent` 변수는 `Parent` 타입이다. 따라서 `parent.value`, `parent.method()` 를 호출하면 인스턴스의 `Parent` 타입에서 기능을 찾아서 실행한다.

Parent → Child



- 이 부분이 중요하다.
- `poly` 변수는 `Parent` 타입이다. 따라서 `poly.value`, `poly.method()` 를 호출하면 인스턴스의 `Parent` 타입에서 기능을 찾아서 실행한다.
 - `poly.value`: `Parent` 타입에 있는 `value` 값을 읽는다.
 - `poly.method()`: `Parent` 타입에 있는 `method()` 를 실행하려고 한다. 그런데 하위 타입인 `Child.method()` 가 오버라이딩 되어 있다. **오버라이딩 된 메서드는 항상 우선권을 가진다.** 따라서 `Parent.method()` 가 아니라 `Child.method()` 가 실행된다.

오버라이딩 된 메서드는 항상 우선권을 가진다. 오버라이딩은 부모 타입에서 정의한 기능을 자식 타입에서 재정의하는 것이다. 만약 자식에서도 오버라이딩 하고 손자에서도 같은 메서드를 오버라이딩을 하면 손자의 오버라이딩 메서드가 우선권을 가진다. 더 하위 자식의 오버라이딩 된 메서드가 우선권을 가지는 것이다.

지금까지 다형성을 이루는 핵심 이론인 다형적 참조와 메서드 오버라이딩에 대해 학습했다.

- **다형적 참조**: 하나의 변수 타입으로 다양한 자식 인스턴스를 참조할 수 있는 기능
- **메서드 오버라이딩**: 기존 기능을 하위 타입에서 새로운 기능으로 재정의

이 둘을 이해하고 나면 진정한 다형성의 위력을 맛볼 수 있다.

다음 시간에는 지금까지 학습한 이론들이 다형성에 어떻게 활용되는지 다형성의 힘을 예제를 통해 알아보자.

정리

11. 다형성2

#1.인강/0.자바/2.자바-기본

- /다형성 활용1
- /다형성 활용2
- /다형성 활용3
- /추상 클래스1
- /추상 클래스2
- /인터페이스
- /인터페이스 - 다중 구현
- /클래스와 인터페이스 활용
- /정리

다형성 활용1

지금까지 학습한 다형성을 왜 사용하는지, 그 장점을 알아보기 위해 우선 다형성을 사용하지 않고 프로그램을 개발한 다음에 다형성을 사용하도록 코드를 변경해보자.
아주 단순하고 전통적인 동물 소리 문제로 접근해보자.

sound()

Dog

sound()

Cat

sound()

Caw

개, 고양이, 소의 울음 소리를 테스트하는 프로그램을 작성해보자. 먼저 다형성을 사용하지 않고 코드를 작성해보자.

예제1

```
package poly.ex1;

public class Dog {
    public void sound() {
        System.out.println("멍멍");
    }
}
```

```
package poly.ex1;

public class Cat {
    public void sound() {
        System.out.println("나옹");
    }
}
```

```
package poly.ex1;

public class Caw {
    public void sound() {
        System.out.println("음매");
    }
}
```

```
package poly.ex1;

public class AnimalSoundMain {

    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        Caw caw = new Caw();

        System.out.println("동물 소리 테스트 시작");
        dog.sound();
        System.out.println("동물 소리 테스트 종료");

        System.out.println("동물 소리 테스트 시작");
        cat.sound();
        System.out.println("동물 소리 테스트 종료");

        System.out.println("동물 소리 테스트 시작");
        caw.sound();
        System.out.println("동물 소리 테스트 종료");
    }
}
```

실행 결과

```
동물 소리 테스트 시작
멍멍
```

```
동물 소리 테스트 종료
```

```
동물 소리 테스트 시작
```

```
나옹
```

```
동물 소리 테스트 종료
```

```
동물 소리 테스트 시작
```

```
음매
```

```
동물 소리 테스트 종료
```

단순히 개, 고양이, 소 동물들의 울음 소리를 출력하는 프로그램이다. 만약 여기서 새로운 동물이 추가되면 어떻게 될까?

만약 기존 코드에 소가 없었다고 가정해보자, 소가 추가된다고 가정하면 `Caw` 클래스를 만들고 다음 코드도 추가해야 한다.

```
//Caw를 생성하는 코드
Caw caw = new Caw();

//Caw를 사용하는 코드
System.out.println("동물 소리 테스트 시작");
caw.sound();
System.out.println("동물 소리 테스트 종료");
```

`Caw`를 생성하는 부분은 당연히 필요하니 크게 상관이 없지만, `Dog`, `Cat`, `Caw`를 사용해서 출력하는 부분은 계속 중복이 증가한다.

중복 코드

```
System.out.println("동물 소리 테스트 시작");
dog.sound();
System.out.println("동물 소리 테스트 종료");

System.out.println("동물 소리 테스트 시작");
cat.sound();
System.out.println("동물 소리 테스트 종료");
...
```

이 부분의 중복을 제거할 수 있을까?

중복을 제거하기 위해서는 메서드를 사용하거나, 또는 배열과 `for` 문을 사용하면 된다.

그런데 `Dog`, `Cat`, `Caw`는 서로 완전히 다른 클래스다.

중복 제거 시도

메서드로 중복 제거 시도

메서드를 사용하면 다음과 같이 매개변수의 클래스를 Caw, Dog, Cat 중에 하나로 정해야 한다.

```
private static void soundCaw(Caw caw) {  
    System.out.println("동물 소리 테스트 시작");  
    caw.sound();  
    System.out.println("동물 소리 테스트 종료");  
}
```

따라서 이 메서드는 Caw 전용 메서드가 되고 Dog, Cat은 인수로 사용할 수 없다.

Dog, Cat, Caw의 타입(클래스)이 서로 다르기 때문에 soundCaw 메서드를 함께 사용하는 것은 불가능하다.

배열과 for문을 통한 중복 제거 시도

```
Caw[] cawArr = {cat, dog, caw}; //컴파일 오류 발생!  
System.out.println("동물 소리 테스트 시작");  
for (Caw caw : cawArr) {  
    cawArr.sound();  
}  
System.out.println("동물 소리 테스트 종료");
```

배열과 for문 사용해서 중복을 제거하려고 해도 배열의 타입을 Dog, Cat, Caw 중에 하나로 지정해야 한다. 같은 Caw들을 배열에 담아서 처리하는 것은 가능하지만 타입이 서로 다른 Dog, Cat, Caw을 하나의 배열에 담는 것은 불가능하다.

결과적으로 지금 상황에서는 해결 방법이 없다. 새로운 동물이 추가될 때마다 더 많은 중복 코드를 작성해야 한다.

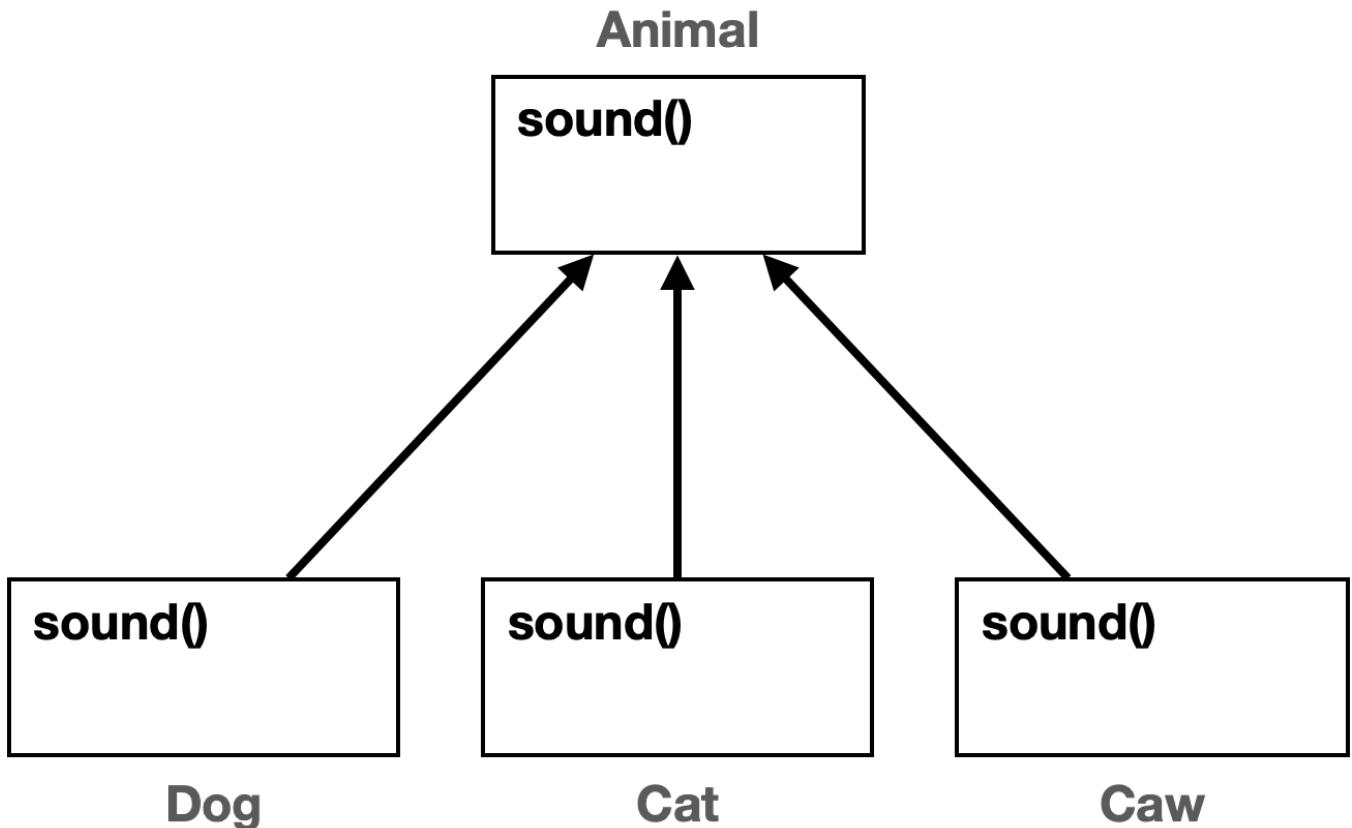
지금까지 설명한 모든 중복 제거 시도가 Dog, Cat, Caw의 타입이 서로 다르기 때문에 불가능하다. 문제의 핵심은 바로 타입이 다르다는 점이다. 반대로 이야기하면 Dog, Cat, Caw가 모두 같은 타입을 사용할 수 있는 방법이 있다면 메서드와 배열을 활용해서 코드의 중복을 제거할 수 있다는 것이다.

다형성의 핵심은 다형적 참조와 메서드 오버라이딩이다. 이 둘을 활용하면 Dog, Cat, Caw가 모두 같은 타입을 사용하고, 각자 자신의 메서드도 호출할 수 있다.

다형성 활용2

이번에는 앞서 설명한 예제를 다형성을 사용하도록 변경해보자.

예제2



다형성을 사용하기 위해 여기서는 상속 관계를 사용한다. `Animal` (동물)이라는 부모 클래스를 만들고 `sound()` 메서드를 정의한다. 이 메서드는 자식 클래스에서 오버라이딩 할 목적으로 만들었다.

`Dog`, `Cat`, `Caw`는 `Animal` 클래스를 상속받았다. 그리고 각각 부모의 `sound()` 메서드를 오버라이딩 한다.

기존 코드를 유지하기 위해 새로운 패키지를 만들고 새로 코드를 작성하자.

주의! 패키지 이름에 주의하자 `import` 를 사용해서 다른 패키지에 있는 같은 이름의 클래스를 사용하면 안된다.

```

package poly.ex2;

public class Animal {
    public void sound() {
        System.out.println("동물 울음 소리");
    }
}
  
```

```

package poly.ex2;

public class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("멍멍");
    }
}
  
```

```
}
```

```
package poly.ex2;

public class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("냐옹");
    }
}
```

```
package poly.ex2;

public class Caw extends Animal{
    @Override
    public void sound() {
        System.out.println("음매");
    }
}
```

```
package poly.ex2;

public class AnimalPolyMain1 {

    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        Caw caw = new Caw();

        soundAnimal(dog);
        soundAnimal(cat);
        soundAnimal(caw);
    }

    //동물이 추가 되어도 변하지 않는 코드
    private static void soundAnimal(Animal animal) {
        System.out.println("동물 소리 테스트 시작");
        animal.sound();
        System.out.println("동물 소리 테스트 종료");
    }
}
```

실행 결과

```
동물 소리 테스트 시작
```

```
멍멍
```

```
동물 소리 테스트 종료
```

```
동물 소리 테스트 시작
```

```
냐옹
```

```
동물 소리 테스트 종료
```

```
동물 소리 테스트 시작
```

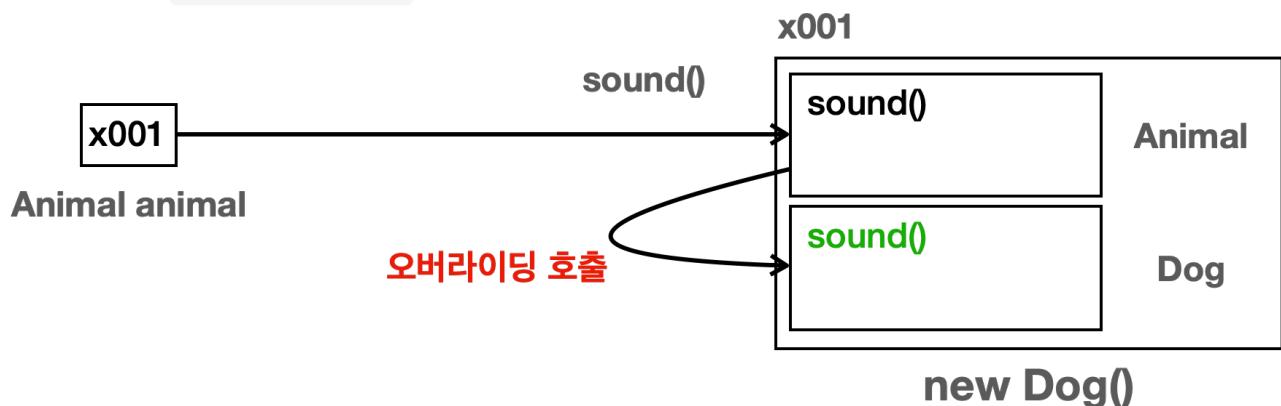
```
음매
```

```
동물 소리 테스트 종료
```

실행 결과는 기존 코드와 같다.

코드를 분석해보자.

- `soundAnimal(dog)` 을 호출하면
- `soundAnimal(Animal animal)`에 `Dog` 인스턴스가 전달된다.
 - `Animal animal = dog` 로 이해하면 된다. 부모는 자식을 담을 수 있다. `Animal`은 `Dog`의 부모다.
- 메서드 안에서 `animal.sound()` 메서드를 호출한다.



- `animal` 변수의 타입은 `Animal`이므로 `Dog` 인스턴스에 있는 `Animal` 클래스 부분을 찾아서 `sound()` 메서드를 실행한다. 그런데 하위 클래스인 `Dog`에서 `sound()` 메서드를 오버라이딩 했다. 따라서 오버라이딩한 메서드가 우선권을 가진다.
- `Dog` 클래스에 있는 `sound()` 메서드가 호출되므로 "멍멍"이 출력된다.

이 코드의 핵심은 `Animal animal` 부분이다.

- **다형적 참조** 덕분에 `animal` 변수는 자식인 `Dog`, `Cat`, `Caw`의 인스턴스를 참조할 수 있다. (부모는 자식을 담을 수 있다)
- **메서드 오버라이딩** 덕분에 `animal.sound()` 를 호출해도 `Dog.sound()`, `Cat.sound()`, `Caw.sound()` 와 같이 각 인스턴스의 메서드를 호출할 수 있다. 만약 자바에 메서드 오버라이딩이 없었다면 모

두 Animal의 sound() 가 호출되었을 것이다.

다형성 덕분에 이후에 새로운 동물을 추가해도 다음 코드를 그대로 재사용 할 수 있다. 물론 다형성을 사용하기 위해 새로운 동물은 Animal을 상속 받아야 한다.

```
private static void soundAnimal(Animal animal) {
    System.out.println("동물 소리 테스트 시작");
    animal.sound();
    System.out.println("동물 소리 테스트 종료");
}
```

다형성 활용3

이번에는 배열과 for문을 사용해서 중복을 제거해보자.

```
package poly.ex2;

public class AnimalPolyMain2 {

    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        Caw caw = new Caw();
        Animal[] animalArr = {dog, cat, caw};

        //변하지 않는 부분
        for (Animal animal : animalArr) {
            System.out.println("동물 소리 테스트 시작");
            animal.sound();
            System.out.println("동물 소리 테스트 종료");
        }
    }
}
```

실행 결과

```
동물 소리 테스트 시작
멍멍
동물 소리 테스트 종료
동물 소리 테스트 시작
```

냐옹

동물 소리 테스트 종료

동물 소리 테스트 시작

음매

동물 소리 테스트 종료

배열은 같은 타입의 데이터를 나열할 수 있다.

Dog, Cat, Caw는 모두 Animal의 자식이므로 Animal 타입이다.

Animal 타입의 배열을 만들고 다형적 참조를 사용하면 된다.

```
//둘은 같은 코드이다.  
Animal[] animalArr = new Animal[]{dog, cat, caw};  
Animal[] animalArr = {dog, cat, caw}
```

다형적 참조 덕분에 Dog, Cat, Caw의 부모 타입인 Animal 타입으로 배열을 만들고, 각각을 배열에 포함했다.

이제 배열을 for문을 사용해서 반복하면 된다.

```
//변하지 않는 부분  
for (Animal animal : animalArr) {  
    System.out.println("동물 소리 테스트 시작");  
    animal.sound();  
    System.out.println("동물 소리 테스트 종료");  
}
```

animal.sound()를 호출하지만 배열에는 Dog, Cat, Caw의 인스턴스가 들어있다. 메서드 오버라이딩에 의해 각 인스턴스의 오버라이딩 된 sound() 메서드가 호출된다.

조금 더 개선

이번에는 배열과 메서드 모두 활용해서 기존 코드를 완성해보자.

```
package poly.ex2;  
  
public class AnimalPolyMain3 {  
  
    public static void main(String[] args) {  
        Animal[] animalArr = {new Dog(), new Cat(), new Caw()};  
        for (Animal animal : animalArr) {  
            soundAnimal(animal);  
        }  
    }  
}
```

```
//동물이 추가 되어도 변하지 않는 코드
private static void soundAnimal(Animal animal) {
    System.out.println("동물 소리 테스트 시작");
    animal.sound();
    System.out.println("동물 소리 테스트 종료");
}
}
```

- Animal[] animalArr 를 통해서 배열을 사용한다.
- soundAnimal(Animal animal)
 - 하나의 동물을 받아서 로직을 처리한다.

새로운 동물이 추가되어도 soundAnimal(..) 메서드는 코드 변경 없이 유지할 수 있다. 이렇게 할 수 있는 이유는 이 메서드는 Dog, Cat, Caw 같은 구체적인 클래스를 참조하는 것이 아니라 Animal이라는 추상적인 부모를 참조하기 때문이다. 따라서 Animal을 상속 받은 새로운 동물이 추가되어도 이 메서드의 코드는 변경 없이 유지할 수 있다.

여기서 잘 보면 새로운 동물이 추가되었을 때 코드가 변하는 부분과 변하지 않는 부분이 있다.

main() 은 코드가 변하는 부분이다. 새로운 동물을 생성하고 필요한 메서드를 호출한다.

soundAnimal(..) 는 코드가 변하지 않는 부분이다.

새로운 기능이 추가되었을 때 변하는 부분을 최소화 하는 것이 잘 작성된 코드이다. 이렇게 하기 위해서는 코드에서 변하는 부분과 변하지 않는 부분을 명확하게 구분하는 것이 좋다.

남은 문제

지금까지 설명한 코드에는 사실 2가지 문제가 있다.

- Animal 클래스를 생성할 수 있는 문제
- Animal 클래스를 상속 받는 곳에서 sound() 메서드 오버라이딩을 하지 않을 가능성

Animal 클래스를 생성할 수 있는 문제

Animal 클래스는 동물이라는 클래스이다. 이 클래스를 다음과 같이 직접 생성해서 사용할 일이 있을까?

```
Animal animal = new Animal();
```

개, 고양이, 소가 실제 존재하는 것은 당연하지만, 동물이라는 추상적인 개념이 실제로 존재하는 것은 이상하다. 사실 이 클래스는 다형성을 위해서 필요한 것이지 직접 인스턴스를 생성해서 사용할 일은 없다.

하지만 Animal도 클래스이기 때문에 인스턴스를 생성하고 사용하는데 아무런 제약이 없다. 누군가 실수로 new Animal() 을 사용해서 Animal의 인스턴스를 생성할 수 있다는 것이다. 이렇게 생성된 인스턴스는 작동은 하지만 제대로된 기능을 수행하지는 않는다.

Animal 클래스를 상속 받는 곳에서 sound() 메서드 오버라이딩을 하지 않을 가능성

예를 들어서 Animal을 상속 받은 Pig 클래스를 만든다고 가정해보자. 우리가 기대하는 것은 Pig 클래스가 sound() 메서드를 오버라이딩 해서 "꿀꿀"이라는 소리가 나도록 하는 것이다. 그런데 개발자가 실수로 sound() 메서드를 오버라이딩 하는 것을 빠트릴 수 있다. 이렇게 되면 부모의 기능을 상속 받는다. 따라서 코드상 아무런 문제가 발생하지 않는다. 물론 프로그램을 실행하면 기대와 다르게 "꿀꿀"이 아니라 부모 클래스에 있는 Animal.sound() 가 호출될 것이다.

좋은 프로그램은 제약이 있는 프로그램이다. 추상 클래스와 추상 메서드를 사용하면 이런 문제를 한번에 해결할 수 있다.

추상 클래스1

추상 클래스

동물(Animal)과 같이 부모 클래스는 제공하지만, 실제 생성되면 안되는 클래스를 추상 클래스라 한다.

추상 클래스는 이름 그대로 추상적인 개념을 제공하는 클래스이다. 따라서 실체인 인스턴스가 존재하지 않는다. 대신에 상속을 목적으로 사용되고, 부모 클래스 역할을 담당한다.

```
abstract class AbstractAnimal {...}
```

- 추상 클래스는 클래스를 선언할 때 앞에 추상이라는 의미의 abstract 키워드를 붙여주면 된다.
- 추상 클래스는 기존 클래스와 완전히 같다. 다만 new AbstractAnimal() 와 같이 직접 인스턴스를 생성하지 못하는 제약이 추가된 것이다.

추상 메서드

부모 클래스를 상속 받는 자식 클래스가 반드시 오버라이딩 해야 하는 메서드를 부모 클래스에 정의할 수 있다. 이것을 추상 메서드라 한다. 추상 메서드는 이름 그대로 추상적인 개념을 제공하는 메서드이다. 따라서 실체가 존재하지 않고, 메서드 바디가 없다.

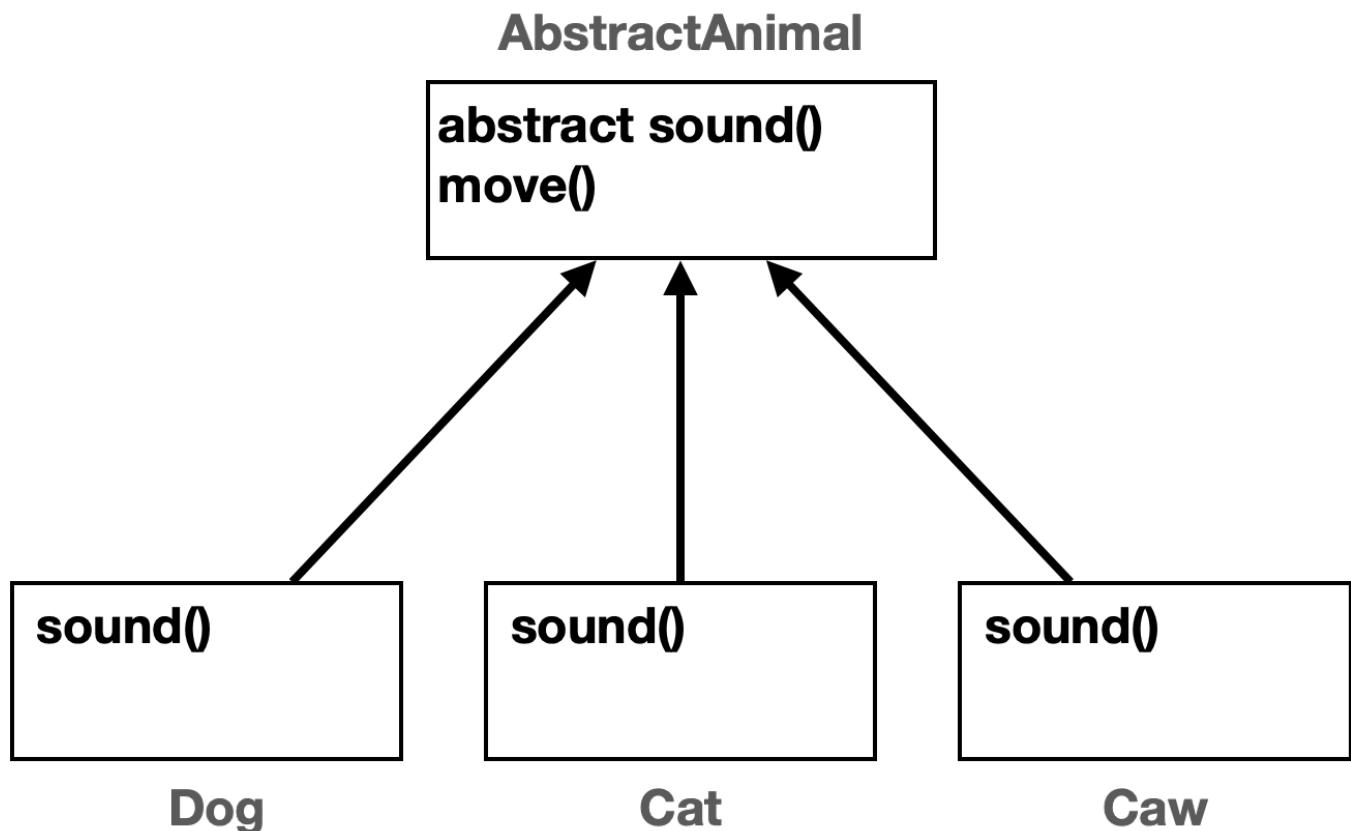
```
public abstract void sound();
```

- 추상 메서드는 선언할 때 메서드 앞에 추상이라는 의미의 abstract 키워드를 붙여주면 된다.
- 추상 메서드가 하나라도 있는 클래스는 추상 클래스로 선언해야 한다.
 - 그렇지 않으면 컴파일 오류가 발생한다.
 - 추상 메서드는 메서드 바디가 없다. 따라서 작동하지 않는 메서드를 가진 불완전한 클래스로 볼 수 있다. 따라서 직접 생성하지 못하도록 추상 클래스로 선언해야 한다.

- 추상 메서드는 상속 받는 자식 클래스가 반드시 오버라이딩 해서 사용해야 한다.
 - 그렇지 않으면 컴파일 오류가 발생한다.
 - 추상 메서드는 자식 클래스가 반드시 오버라이딩 해야 하기 때문에 메서드 바디 부분이 없다. 바디 부분을 만들면 컴파일 오류가 발생한다.
 - 오버라이딩 하지 않으면 자식도 추상 클래스가 되어야 한다.
- 추상 메서드는 기존 메서드와 완전히 같다. 다만 메서드 바디가 없고, 자식 클래스가 해당 메서드를 반드시 오버라이딩 해야 한다는 제약이 추가된 것이다.

이제 추상 클래스와 추상 메서드를 사용해서 예제를 만들어보자.

예제3



```

package poly.ex3;

public abstract class AbstractAnimal {
    public abstract void sound();

    public void move() {
        System.out.println("동물이 움직입니다.");
    }
}
  
```

- `AbstractAnimal`은 `abstract`가 붙은 추상 클래스이다. 이 클래스는 직접 인스턴스를 생성할 수 없다.
- `sound()`은 `abstract`가 붙은 추상 메서드이다. 이 메서드는 자식이 반드시 오버라이딩 해야 한다.

이 클래스는 `move()`라는 메서드를 가지고 있는데, 이 메서드는 추상 메서드가 아니다. 따라서 자식 클래스가 오버라이딩 하지 않아도 된다.

참고로 추상 클래스라고 `AbstractAnimal`처럼 클래스 이름 앞에 꼭 `Abstract`를 써야하는 것은 아니다. 그냥 `Animal`이라는 클래스 이름으로도 충분하다. 여기서는 예제 코드를 다른 예제 코드와 구분해서 설명하기 위해 앞에 `Abstract`를 붙였다.

```
package poly.ex3;

public class Dog extends AbstractAnimal {
    @Override
    public void sound() {
        System.out.println("멍멍");
    }
}
```

```
package poly.ex3;

public class Cat extends AbstractAnimal {
    @Override
    public void sound() {
        System.out.println("냐옹");
    }
}
```

```
package poly.ex3;

public class Caw extends AbstractAnimal {
    @Override
    public void sound() {
        System.out.println("음매");
    }
}
```

```
package poly.ex3;
```

```

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

        //추상클래스 생성 불가
        //AbstractAnimal animal = new AbstractAnimal();

        Dog dog = new Dog();
        Cat cat = new Cat();
        Caw caw = new Caw();

        cat.sound();
        cat.move();

        soundAnimal(cat);
        soundAnimal(dog);
        soundAnimal(caw);
    }

    //동물이 추가 되어도 변하지 않는 코드
    private static void soundAnimal(AbstractAnimal animal) {
        System.out.println("동물 소리 테스트 시작");
        animal.sound();
        System.out.println("동물 소리 테스트 종료");
    }
}

```

실행 결과

```

냐옹
동물이 움직입니다.

동물 소리 테스트 시작
냐옹
동물 소리 테스트 종료

동물 소리 테스트 시작
멍멍
동물 소리 테스트 종료

동물 소리 테스트 시작
음매
동물 소리 테스트 종료

```

추상 클래스는 생성이 불가능하다. 다음 코드의 주석을 풀고 실행하면 컴파일 오류가 발생한다.

```
//추상클래스 생성 불가  
AbstractAnimal animal = new AbstractAnimal();
```

컴파일 오류 - 인스턴스 생성

```
java: poly.ex3.AbstractAnimal is abstract; cannot be instantiated
```

AbstractAnimal 가 추상이어서 인스턴스 생성이 불가능하다는 뜻이다.

추상 메서드는 반드시 오버라이딩 해야 한다. 만약 자식에서 오버라이딩 메서드를 만들지 않으면 다음과 같이 컴파일 오류가 발생한다. Dog 의 sound() 메서드를 잠시 주식처리해보자.

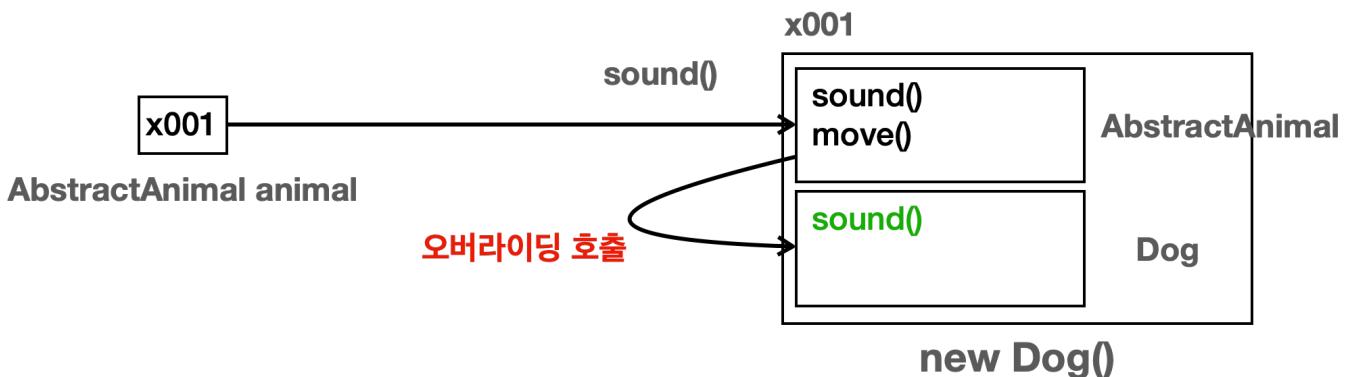
```
package poly.ex3;  
  
public class Dog extends AbstractAnimal {  
/*  
    @Override  
    public void sound() {  
        System.out.println("멍멍");  
    }  
*/  
}
```

컴파일 오류 - 오버라이딩X

```
java: poly.ex3.Dog is not abstract and does not override abstract method sound()  
in poly.ex3.AbstractAnimal
```

Dog 는 추상클래스가 아닌데 sound() 가 오버라이딩 되지 않았다는 뜻이다.

지금까지 설명한 제약을 제외하고 나머지는 모두 일반적인 클래스와 동일하다. 추상 클래스는 제약이 추가된 클래스일 뿐이다. 메모리 구조, 실행 결과 모두 동일하다.



정리

- 추상 클래스 덕분에 실수로 Animal 인스턴스를 생성할 문제를 근본적으로 방지해준다.
- 추상 메서드 덕분에 새로운 동물의 자식 클래스를 만들때 실수로 sound() 를 오버라이딩 하지 않을 문제를 근본적으로 방지해준다.

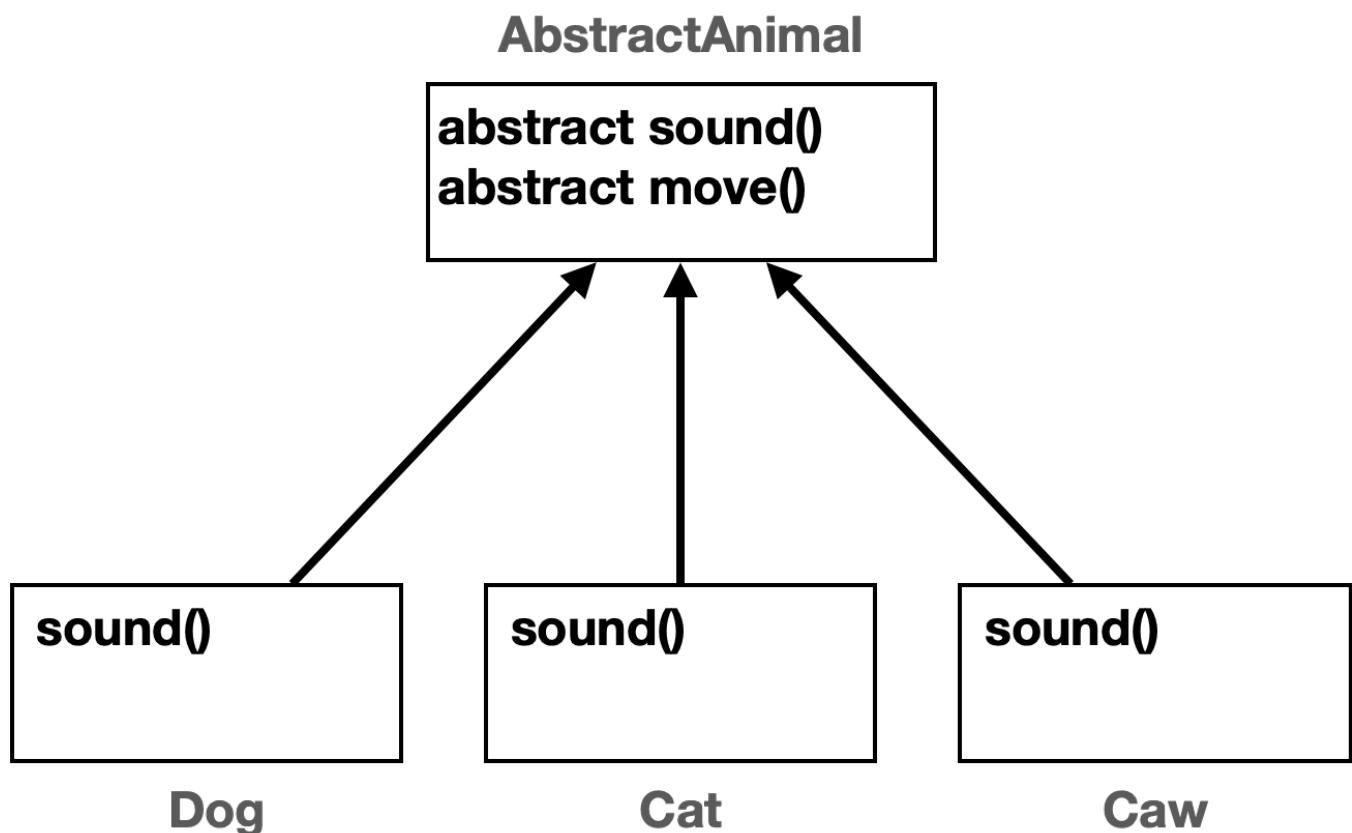
추상 클래스2

순수 추상 클래스: 모든 메서드가 추상 메서드인 추상 클래스

앞서 만든 예제에서 move() 도 추상 메서드로 만들어야 한다고 가정해보자.

이 경우 AbstractAnimal 클래스의 모든 메서드가 추상 메서드가 된다. 이런 클래스를 순수 추상 클래스라 한다.

move() 가 추상 메서드가 되었으니 자식들은 AbstractAnimal 의 모든 기능을 오버라이딩 해야 한다.



예제4

```
package poly.ex4;
```

```
public abstract class AbstractAnimal {  
    public abstract void sound();  
    public abstract void move();  
}
```

```
package poly.ex4;  
  
public class Dog extends AbstractAnimal {  
    @Override  
    public void sound() {  
        System.out.println("멍멍");  
    }  
  
    @Override  
    public void move() {  
        System.out.println("개 이동");  
    }  
}
```

```
package poly.ex4;  
  
public class Cat extends AbstractAnimal {  
    @Override  
    public void sound() {  
        System.out.println("냐옹");  
    }  
  
    @Override  
    public void move() {  
        System.out.println("고양이 이동");  
    }  
}
```

```
package poly.ex4;  
  
public class Caw extends AbstractAnimal {  
    @Override  
    public void sound() {  
        System.out.println("음매");  
    }  
  
    @Override
```

```

public void move() {
    System.out.println("소 이동");
}
}

package poly.ex4;

public class AbstractMain {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        Caw caw = new Caw();

        soundAnimal(cat);
        soundAnimal(dog);
        soundAnimal(caw);

        moveAnimal(cat);
        moveAnimal(dog);
        moveAnimal(caw);
    }

    //동물이 추가 되어도 변하지 않는 코드
    private static void soundAnimal(AbstractAnimal animal) {
        System.out.println("동물 소리 테스트 시작");
        animal.sound();
        System.out.println("동물 소리 테스트 종료");
    }

    //동물이 추가 되어도 변하지 않는 코드
    private static void moveAnimal(AbstractAnimal animal) {
        System.out.println("동물 이동 테스트 시작");
        animal.move();
        System.out.println("동물 이동 테스트 종료");
    }
}

```

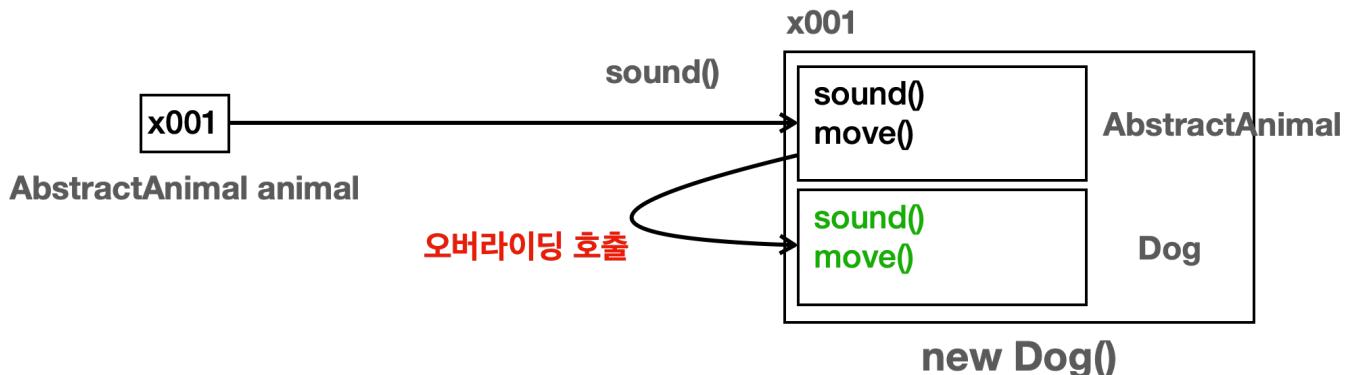
실행 결과

```

동물 소리 테스트 시작
냐옹
동물 소리 테스트 종료
...

```

```
동물 이동 테스트 시작
고양이 이동
동물 이동 테스트 종료
동물 이동 테스트 시작
개 이동
동물 이동 테스트 종료
동물 이동 테스트 시작
소 이동
동물 이동 테스트 종료
```



코드를 이해하는데 어려움은 없을 것이다.

순수 추상 클래스

모든 메서드가 추상 메서드인 순수 추상 클래스는 코드를 실행할 바디 부분이 전혀 없다.

```
public abstract class AbstractAnimal {
    public abstract void sound();
    public abstract void move();
}
```

이러한 순수 추상 클래스는 실행 로직을 전혀 가지고 있지 않다. 단지 다형성을 위한 부모 타입으로써 껍데기 역할만 제공할 뿐이다.

순수 추상 클래스는 다음과 같은 특징을 가진다.

- 인스턴스를 생성할 수 없다.
- 상속시 자식은 모든 메서드를 오버라이딩 해야 한다.
- 주로 다형성을 위해 사용된다.

상속하는 클래스는 모든 메서드를 구현해야 한다.

"상속시 자식은 모든 메서드를 오버라이딩 해야 한다."라는 특징은 상속 받는 클래스 입장에서 보면 부모의 모든 메서드를 구현해야 하는 것이다.

이런 특징을 잘 생각해보면 순수 추상 클래스는 마치 어떤 규격을 지켜서 구현해야 하는 것처럼 느껴진다.

AbstractAnimal의 경우 sound(), move()라는 규격에 맞추어 구현을 해야 한다.

이것은 우리가 일반적으로 이야기하는 인터페이스와 같이 느껴진다. 예를 들어서 USB 인터페이스를 생각해보자. USB 인터페이스는 분명한 규격이 있다. 이 규격에 맞추어 제품을 개발해야 연결이 된다. 순수 추상 클래스가 USB 인터페이스 규격이라고 한다면 USB 인터페이스에 맞추어 마우스, 키보드 같은 연결 장치들을 구현할 수 있다.

이런 순수 추상 클래스의 개념은 프로그래밍에서 매우 자주 사용된다. 자바는 순수 추상 클래스를 더 편리하게 사용할 수 있도록 인터페이스라는 개념을 제공한다.

인터페이스

자바는 순수 추상 클래스를 더 편리하게 사용할 수 있는 인터페이스라는 기능을 제공한다.

순수 추상 클래스

```
public abstract class AbstractAnimal {  
    public abstract void sound();  
    public abstract void move();  
}
```

인터페이스는 `class` 가 아니라 `interface` 키워드를 사용하면 된다.

인터페이스

```
public interface InterfaceAnimal {  
    public abstract void sound();  
    public abstract void move();  
}
```

인터페이스 - `public abstract` 키워드 생략 가능

```
public interface InterfaceAnimal {  
    void sound();  
    void move();  
}
```

순수 추상 클래스는 다음과 같은 특징을 가진다.

- 인스턴스를 생성할 수 없다.
- 상속시 모든 메서드를 오버라이딩 해야 한다.

- 주로 다형성을 위해 사용된다.

인터페이스는 앞서 설명한 순수 추상 클래스와 같다. 여기에 약간의 편의 기능이 추가된다.

- 인터페이스의 메서드는 모두 `public`, `abstract`이다.
- 메서드에 `public abstract`를 생략할 수 있다. 참고로 생략이 권장된다.
- 인터페이스는 다중 구현(다중 상속)을 지원한다.

인터페이스와 멤버 변수

```
public interface InterfaceAnimal {  
    public static final int MY_PI = 3.14;  
}
```

인터페이스에서 멤버 변수는 `public`, `static`, `final`이 모두 포함되었다고 간주된다. `final`은 변수의 값을 한 번 설정하면 수정할 수 없다는 뜻이다.

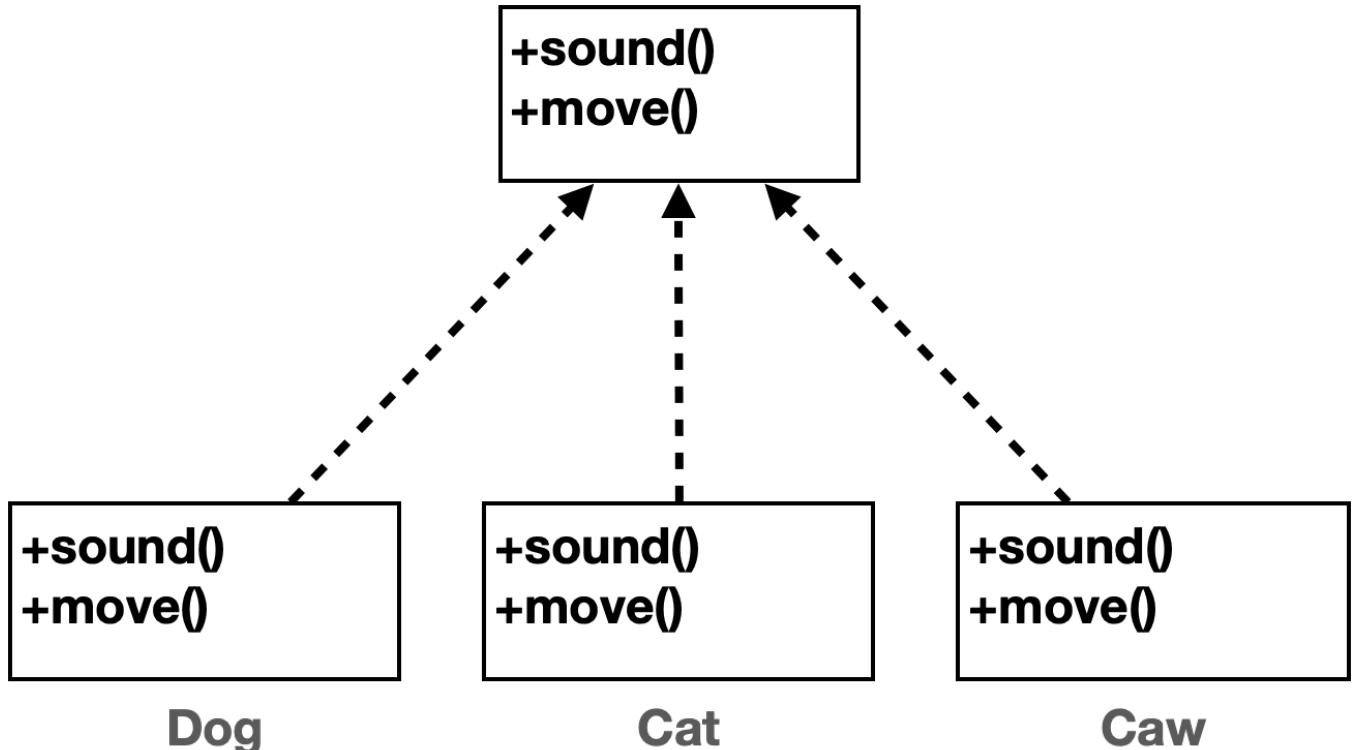
자바에서 `static final`을 사용해 정적이면서 고칠 수 없는 변수를 상수라 하고, 관례상 상수는 대문자에 언더스코어(_)로 구분한다.

해당 키워드는 다음과 같이 생략할 수 있다. (생략이 권장된다.)

```
public interface InterfaceAnimal {  
    int MY_PI = 3.14;  
}
```

예제5

InterfaceAnimal



클래스 상속 관계는 UML에서 실선을 사용하지만, 인터페이스 구현(상속) 관계는 UML에서 점선을 사용한다.

```
package poly.ex5;

public interface InterfaceAnimal {
    void sound();
    void move();
}
```

인터페이스는 `class` 대신에 `interface`로 선언하면 된다.

`sound()`, `move()` 는 앞에 `public abstract` 가 생략되어 있다. 따라서 상속 받는 곳에서 모든 메서드를 오버라이딩 해야 한다.

```
package poly.ex5;

public class Dog implements InterfaceAnimal {
    @Override
    public void sound() {
        System.out.println("멍멍");
    }

    @Override
    public void move() {
```

```
        System.out.println("개 이동");
    }
}
```

인터페이스를 상속 받을 때는 `extends` 대신에 `implements`라는 구현이라는 키워드를 사용해야 한다. 인터페이스는 그래서 상속이라 하지 않고 구현이라 한다.

```
package poly.ex5;

public class Cat implements InterfaceAnimal {
    @Override
    public void sound() {
        System.out.println("냐옹");
    }

    @Override
    public void move() {
        System.out.println("고양이 이동");
    }
}
```

```
package poly.ex5;

public class Caw implements InterfaceAnimal {
    @Override
    public void sound() {
        System.out.println("음매");
    }

    @Override
    public void move() {
        System.out.println("소 이동");
    }
}
```

```
package poly.ex5;

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

        //인터페이스 생성 불가
        //InterfaceAnimal interfaceMain1 = new InterfaceAnimal();
```

```

Cat cat = new Cat();
Dog dog = new Dog();
Caw caw = new Caw();

soundAnimal(cat);
soundAnimal(dog);
soundAnimal(caw);
}

//동물이 추가 되어도 변하지 않는 코드
private static void soundAnimal(InterfaceAnimal animal) {
    System.out.println("동물 소리 테스트 시작");
    animal.sound();
    System.out.println("동물 소리 테스트 종료");
}
}

```

동물 소리 테스트 시작

냐옹

동물 소리 테스트 종료

동물 소리 테스트 시작

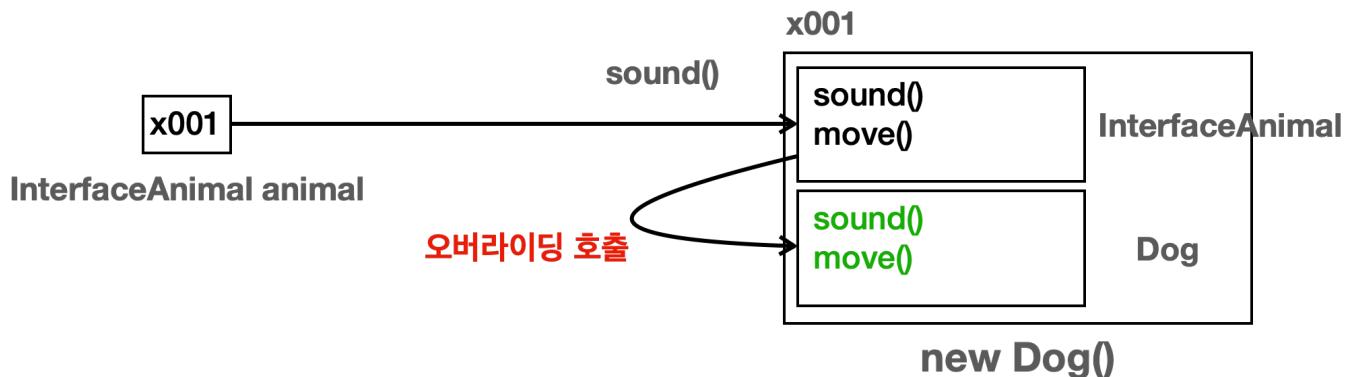
멍멍

동물 소리 테스트 종료

동물 소리 테스트 시작

음매

동물 소리 테스트 종료



앞서 설명한 순수 추상 클래스 예제와 거의 유사하다. 순수 추상 클래스가 인터페이스가 되었을 뿐이다.

클래스, 추상 클래스, 인터페이스는 모두 똑같다.

- 클래스, 추상 클래스, 인터페이스는 프로그램 코드, 메모리 구조상 모두 똑같다. 모두 자바에서는 `.class`로 다루어진다. 인터페이스를 작성할 때도 `.java`에 인터페이스를 정의한다.
- 인터페이스는 순수 추상 클래스와 비슷하다고 생각하면 된다.

상속 vs 구현

부모 클래스의 기능을 자식 클래스가 상속 받을 때, 클래스는 상속 받는다고 표현하지만, 부모 인터페이스의 기능을 자식이 상속 받을 때는 인터페이스를 구현한다고 표현한다. 이렇게 서로 다르게 표현하는 이유는 알아보자. 상속은 이름 그대로 부모의 기능을 물려 받는 것이 목적이다. 하지만 인터페이스는 모든 메서드가 추상 메서드이다. 따라서 물려받을 수 있는 기능이 없고, 오히려 인터페이스에 정의한 모든 메서드를 자식이 오버라이딩 해서 기능을 구현해야 한다. 따라서 구현한다고 표현한다.

인터페이스는 메서드 이름만 있는 설계도이고, 이 설계도가 실제 어떻게 작동하는지는 하위 클래스에서 모두 구현해야 한다. 따라서 인터페이스의 경우 상속이 아니라 해당 인터페이스를 구현한다고 표현한다.

상속과 구현은 사람이 표현하는 단어만 다를 뿐이지 자바 입장에서는 똑같다. 일반 상속 구조와 동일하게 작동한다.

인터페이스를 사용해야 하는 이유

모든 메서드가 추상 메서드인 경우 순수 추상 클래스를 만들어도 되고, 인터페이스를 만들어도 된다. 그런데 왜 인터페이스를 사용해야 할까? 단순히 편리하다는 이유를 넘어서 다음과 같은 이유가 있다.

- 제약:** 인터페이스를 만드는 이유는 인터페이스를 구현하는 곳에서 인터페이스의 메서드를 반드시 구현해라는 규약(제약)을 주는 것이다. USB 인터페이스를 생각해보자. USB 인터페이스에 맞추어 키보드, 마우스를 개발하고 연결해야 한다. 그렇지 않으면 작동하지 않는다. 인터페이스의 규약(제약)은 반드시 구현해야 하는 것이다. 그런데 순수 추상 클래스의 경우 미래에 누군가 그곳에 실행 가능한 메서드를 끼워 넣을 수 있다. 이렇게 되면 추가된 기능을 자식 클래스에서 구현하지 않을 수도 있고, 또 더는 순수 추상 클래스가 아니게 된다. 인터페이스는 모든 메서드가 추상 메서드이다. 따라서 이런 문제를 원천 차단할 수 있다.
- 다중 구현:** 자바에서 클래스 상속은 부모를 하나만 지정할 수 있다. 반면에 인터페이스는 부모를 여러명 두는 다중 구현(다중 상속)이 가능하다.

좋은 프로그램은 제약이 있는 프로그램이다.

참고

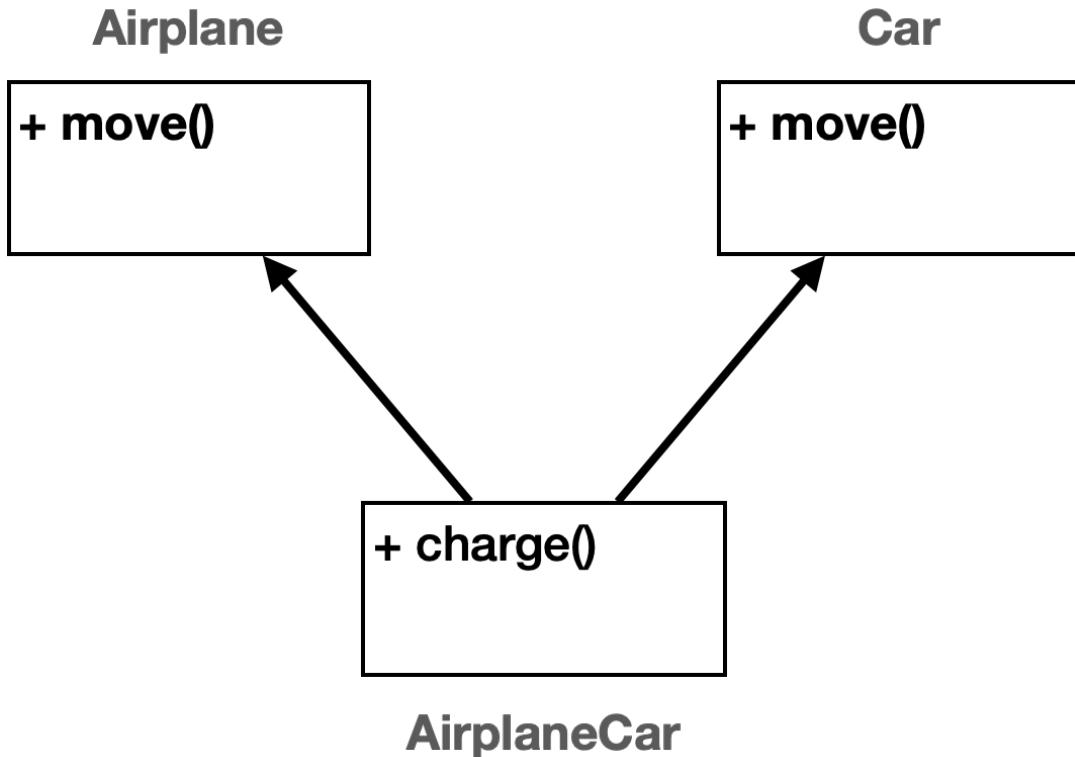
자바8에 등장한 `default` 메서드를 사용하면 인터페이스도 메서드를 구현할 수 있다. 하지만 이것은 예외적으로 아주 특별한 경우에만 사용해야 한다. 자바9에서 등장한 인터페이스의 `private` 메서드도 마찬가지이다. 지금 학습 단계에서는 이 부분들을 고려하지 않는 것이 좋다. 이 부분은 뒤에서 따로 다룬다.

인터페이스 - 다중 구현

자바가 다중 상속을 지원하지 않는 이유 - 복습

자바는 다중 상속을 지원하지 않는다. 그래서 `extends` 대상은 하나만 선택할 수 있다. 부모를 하나만 선택할 수 있다는 뜻이다. 물론 부모가 또 부모를 가지는 것은 괜찮다.

다중 상속 그림



만약 비행기와 자동차를 상속 받아서 하늘을 나는 자동차를 만든다고 가정해보자. 만약 그림과 같이 다중 상속을 사용하게 되면 `AirplaneCar` 입장에서 `move()`를 호출할 때 어떤 부모의 `move()`를 사용해야 할지 애매한 문제가 발생한다. 이것을 다이아몬드 문제라 한다. 그리고 다중 상속을 사용하면 클래스 계층 구조가 매우 복잡해지 수 있다. 이런 문제점 때문에 자바는 클래스의 다중 상속을 허용하지 않는다. 대신에 인터페이스의 다중 구현을 허용하여 이러한 문제를 피한다.

클래스는 앞서 설명한 이유로 다중 상속이 안되는데, 인터페이스의 다중 구현은 허용한 이유는 뭘까?

인터페이스는 모두 추상 메서드로 이루어져 있기 때문이다.

다음 예제를 보자.

인터페이스 다중 구현 그림

InterfaceA

```
+methodA()  
+methodCommon()
```

InterfaceB

```
+methodB()  
+methodCommon()
```

```
+methodA()  
+methodB()  
+methodCommon()
```

Child

InterfaceA, InterfaceB는 둘다 같은 `methodCommon()`을 가지고 있다. 그리고 Child는 두 인터페이스를 구현했다. 상속 관계의 경우 두 부모 중에 어떤 한 부모의 `methodCommon()`을 사용해야 할지 결정해야 하는 다이아몬드 문제가 발생한다.

하지만 인터페이스 자신은 구현을 가지지 않는다. 대신에 인터페이스를 구현하는 곳에서 해당 기능을 모두 구현해야 한다. 여기서 InterfaceA, InterfaceB는 같은 이름의 `methodCommon()`를 제공하지만 이것의 기능은 Child가 구현한다. 그리고 오버라이딩에 의해 어차피 Child에 있는 `methodCommon()`이 호출된다. 결과적으로 두 부모 중에 어떤 한 부모의 `methodCommon()`을 선택하는 것이 아니라 그냥 인터페이스들을 구현한 Child에 있는 `methodCommon()`이 사용된다. 이런 이유로 인터페이스는 다이아몬드 문제가 발생하지 않는다. 따라서 인터페이스의 경우 다중 구현을 허용한다.

예제를 코드로 작성해보자.

```
package poly.diamond;  
  
public interface InterfaceA {  
    void methodA();  
    void methodCommon();  
}
```

```
package poly.diamond;
```

```

public interface InterfaceB {
    void methodB();
    void methodCommon();
}

package poly.diamond;

public class Child implements InterfaceA, InterfaceB {
    @Override
    public void methodA() {
        System.out.println("Child.methodA");
    }

    @Override
    public void methodB() {
        System.out.println("Child.methodB");
    }

    @Override
    public void methodCommon() {
        System.out.println("Child.methodCommon");
    }
}

```

- implements InterfaceA, InterfaceB 와 같이 다중 구현을 할 수 있다. implements 키워드 위에 , 로 여러 인터페이스를 구분하면 된다.
- methodCommon() 의 경우 양쪽 인터페이스에 다 있지만 같은 메서드이므로 구현은 하나만 하면 된다.

```

package poly.diamond;

//인터페이스 다중 구현
public class DiamondMain {
    public static void main(String[] args) {
        InterfaceA a = new Child();
        a.methodA();
        a.methodCommon();

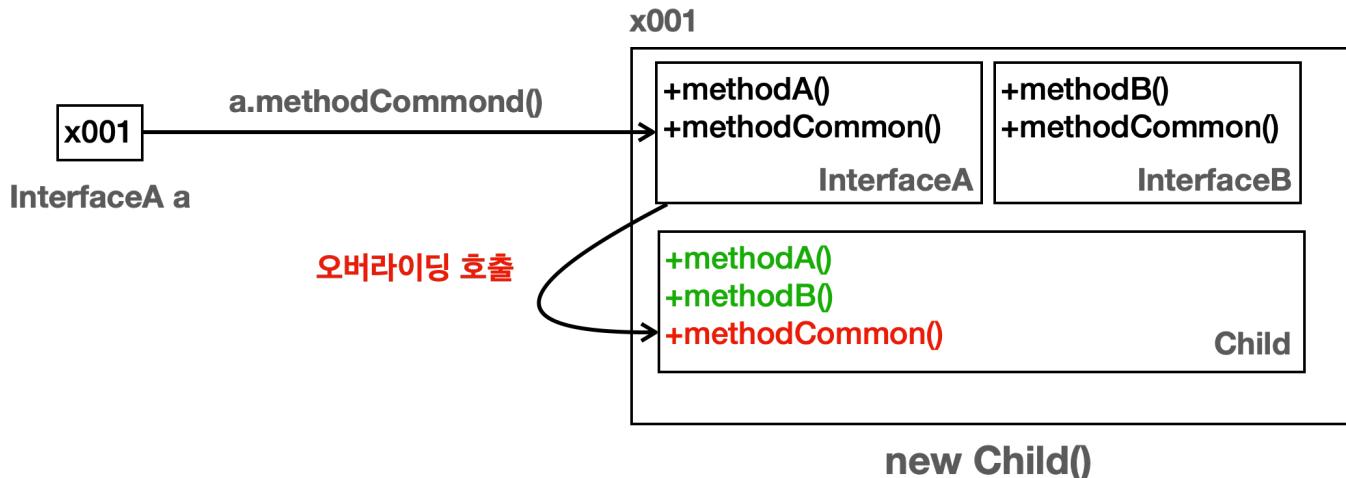
        InterfaceB b = new Child();
        b.methodB();
        b.methodCommon();
    }
}

```

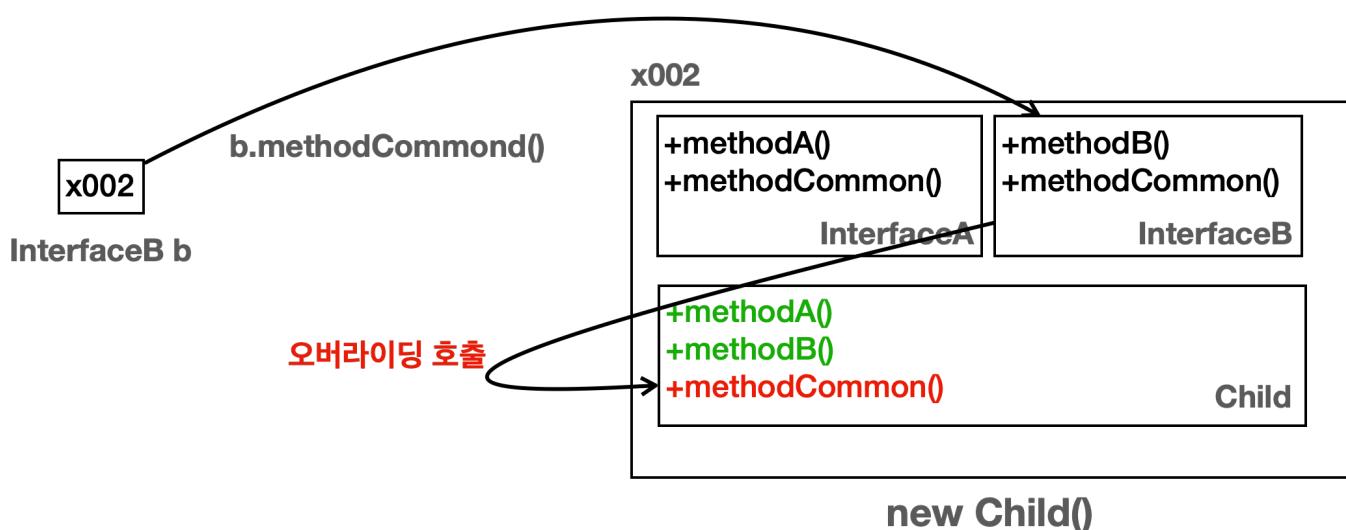
```
    }  
}
```

실행 결과

```
Child.methodA  
Child.methodCommon  
Child.methodB  
Child.methodCommon
```



1. `a.methodCommon()` 을 호출하면 먼저 x001 `Child` 인스턴스를 찾는다.
2. 변수 `a` 가 `InterfaceA` 타입이므로 해당 타입에서 `methodCommon()` 을 찾는다.
3. `methodCommon()` 은 하위 타입인 `Child`에서 오버라이딩 되어 있다. 따라서 `Child`의 `methodCommon()` 이 호출된다.

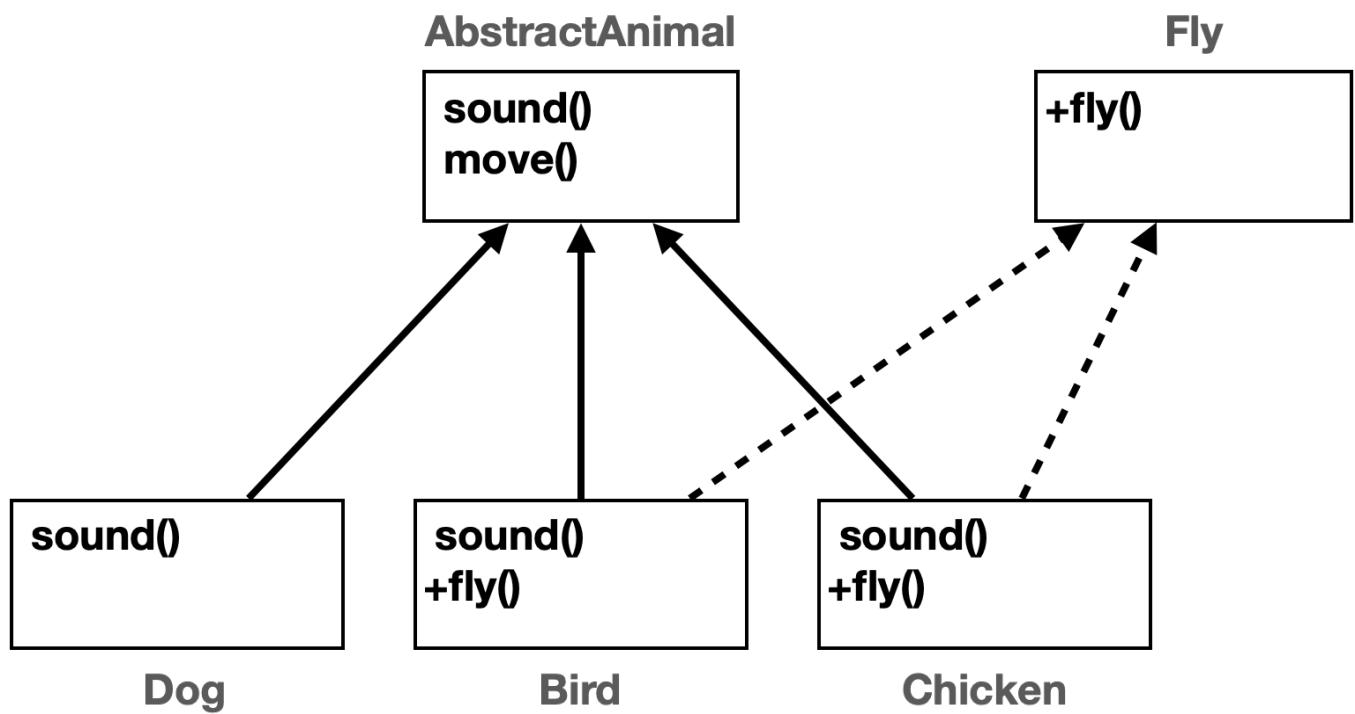


4. `b.methodCommon()` 을 호출하면 먼저 x001 `Child` 인스턴스를 찾는다.
5. 변수 `b` 가 `InterfaceB` 타입이므로 해당 타입에서 `methodCommon()` 을 찾는다.
6. `methodCommon()` 은 하위 타입인 `Child`에서 오버라이딩 되어 있다. 따라서 `Child`의 `methodCommon()`

이 호출된다.

클래스와 인터페이스 활용

이번에는 클래스 상속과 인터페이스 구현을 함께 사용하는 예를 알아보자.



- **AbstractAnimal**은 추상 클래스다.
 - **sound()** : 동물의 소리를 내기 위한 **sound()** 추상 메서드를 제공한다.
 - **move()** : 동물의 이동을 표현하기 위한 메서드이다. 이 메서드는 추상 메서드가 아니다. 상속을 목적으로 사용된다.
- **Fly**는 인터페이스이다. 나는 동물은 이 인터페이스를 구현할 수 있다.
 - **Bird**, **Chicken**은 날 수 있는 동물이다. **fly()** 메서드를 구현해야 한다.

예제6

```
package poly.ex6;

public abstract class AbstractAnimal {
    public abstract void sound();
    public void move() {
        System.out.println("동물이 이동합니다.");
    }
}
```

```
}
```

```
package poly.ex6;
```

```
public interface Fly {  
    void fly();  
}
```

```
package poly.ex6;
```

```
public class Dog extends AbstractAnimal {  
    @Override  
    public void sound() {  
        System.out.println("멍멍");  
    }  
}
```

Dog는 AbstractAnimal만 상속 받는다.

```
package poly.ex6;  
  
public class Bird extends AbstractAnimal implements Fly {  
    @Override  
    public void sound() {  
        System.out.println("짹짹");  
    }  
  
    @Override  
    public void fly() {  
        System.out.println("새 날기");  
    }  
}
```

Bird는 AbstractAnimal 클래스를 상속하고 Fly 인터페이스를 구현한다.

하나의 클래스 여러 인터페이스 예시

```
public class Bird extends AbstractAnimal implements Fly, Swim {
```

extends를 통한 상속은 하나만 할 수 있고 implements를 통한 인터페이스는 다중 구현 할 수 있기 때문에 둘이 함께 나온 경우 extends가 먼저 나와야 한다.

```
package poly.ex6;

public class Chicken extends AbstractAnimal implements Fly {
    @Override
    public void sound() {
        System.out.println("꼬끼오");
    }

    @Override
    public void fly() {
        System.out.println("닭 날기");
    }
}
```

```
package poly.ex6;

public class SoundFlyMain {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Bird bird = new Bird();
        Chicken chicken = new Chicken();

        soundAnimal(dog);
        soundAnimal(bird);
        soundAnimal(chicken);

        flyAnimal(bird);
        flyAnimal(chicken);
    }

    //AbstractAnimal 사용 가능
    private static void soundAnimal(AbstractAnimal animal) {
        System.out.println("동물 소리 테스트 시작");
        animal.sound();
        System.out.println("동물 소리 테스트 종료");
    }

    //Fly 인터페이스가 있으면 사용 가능
    private static void flyAnimal(Fly fly) {
        System.out.println("날기 테스트 시작");
        fly.fly();
    }
}
```

```

        System.out.println("날기 테스트 종료");
    }

}

```

실행 결과

동물 소리 테스트 시작

멍멍

동물 소리 테스트 종료

동물 소리 테스트 시작

꽥꽥

동물 소리 테스트 종료

동물 소리 테스트 시작

꼬끼오

동물 소리 테스트 종료

날기 테스트 시작

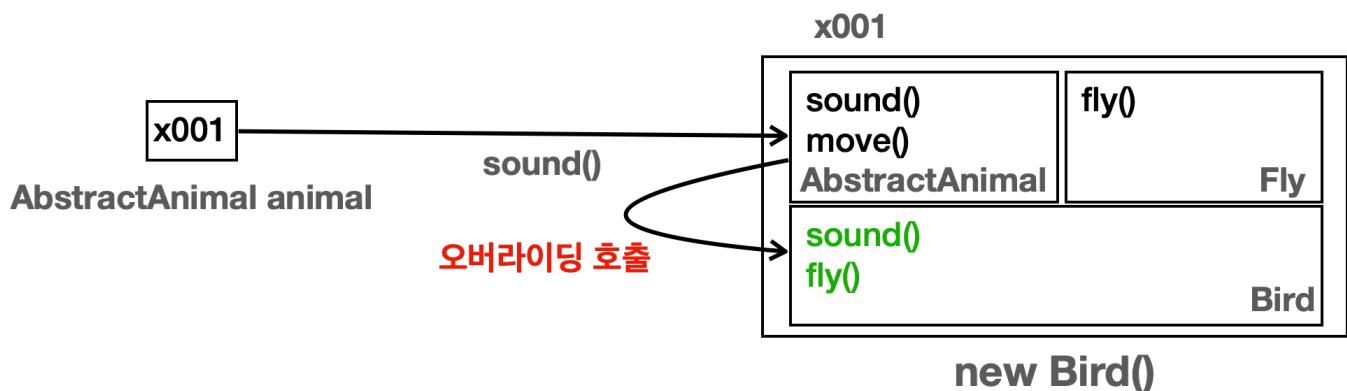
새 날기

날기 테스트 종료

날기 테스트 시작

닭 날기

날기 테스트 종료



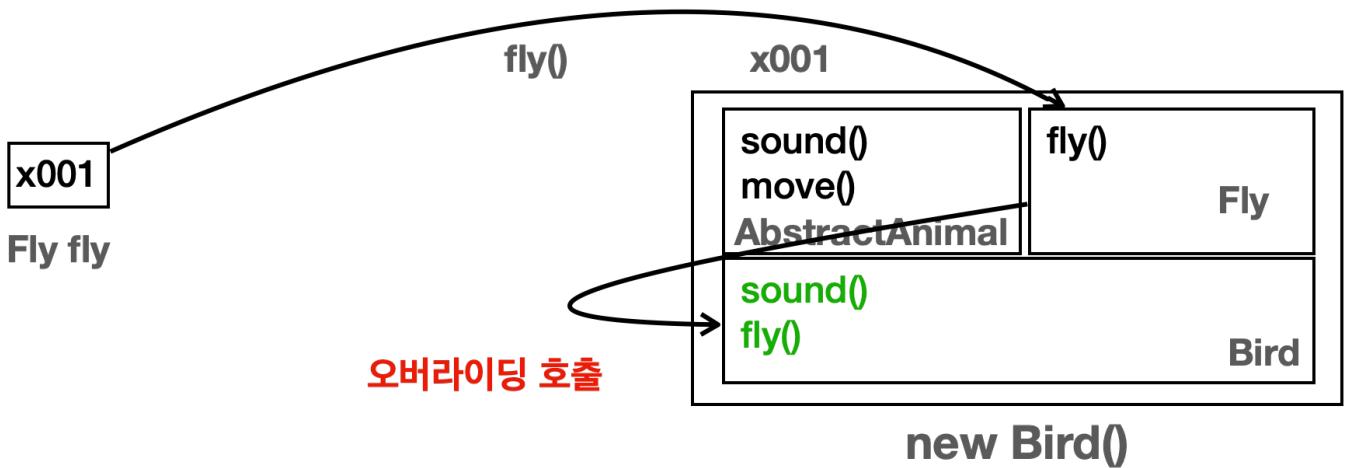
`soundAnimal(AbstractAnimal animal)`

`AbstractAnimal`를 상속한 `Dog`, `Bird`, `Chicken`을 전달해서 실행할 수 있다.

실행 과정

- `soundAnimal(bird)` 를 호출한다고 가정하자.
- 메서드 안에서 `animal.sound()` 를 호출하면 참조 대상인 `x001 Bird` 인스턴스를 찾는다.

- 호출한 `animal` 변수는 `AbstractAnimal` 타입이다. 따라서 `AbstractAnimal.sound()`를 찾는다. 해당 메서드는 `Bird.sound()`에 오버라이딩 되어 있다.
- `Bird.sound()` 가 호출된다.



`flyAnimal(Fly fly)`

`Fly` 인터페이스를 구현한 `Bird`, `Chicken`을 전달해서 실행할 수 있다.

실행 과정

- `fly(bird)` 를 호출한다고 가정하자.
- 메서드 안에서 `fly.fly()` 를 호출하면 참조 대상인 `x001` `Bird` 인스턴스를 찾는다.
- 호출한 `fly` 변수는 `Fly` 타입이다. 따라서 `Fly.fly()` 를 찾는다. 해당 메서드는 `Bird.fly()`에 오버라이딩 되어 있다.
- `Bird.fly()` 가 호출된다.

정리

12. 다형성과 설계

#1.인강/0.자바/2.자바-기본

- /좋은 객체 지향 프로그래밍이란?
- /다형성 - 역할과 구현 예제1
- /다형성 - 역할과 구현 예제2
- /다형성 - 역할과 구현 예제3
- /OCP(Open-Closed Principle) 원칙
- /문제와 풀이
- /정리

좋은 객체 지향 프로그래밍이란?

객체 지향 특징

- 추상화
- 캡슐화
- 상속
- 다형성

객체 지향 프로그래밍

- 객체 지향 프로그래밍은 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 단위, 즉 "객체"들의 모임으로 파악하고자 하는 것이다. 각각의 객체는 메시지를 주고받고, 데이터를 처리할 수 있다. (협력)
- 객체 지향 프로그래밍은 프로그램을 유연하고 변경이 용이하게 만들기 때문에 대규모 소프트웨어 개발에 많이 사용된다.

유연하고, 변경이 용이?

- 레고 블럭 조립하듯이
- 키보드, 마우스 갈아 끼우듯이
- 컴퓨터 부품 갈아 끼우듯이
- 컴포넌트를 쉽고 유연하게 변경하면서 개발할 수 있는 방법



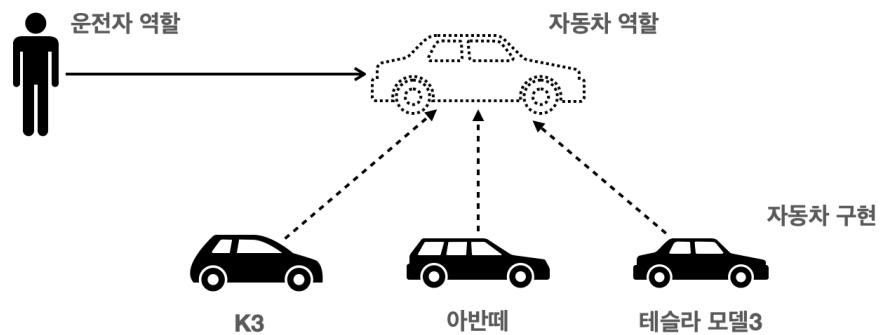
다형성

Polymorphism

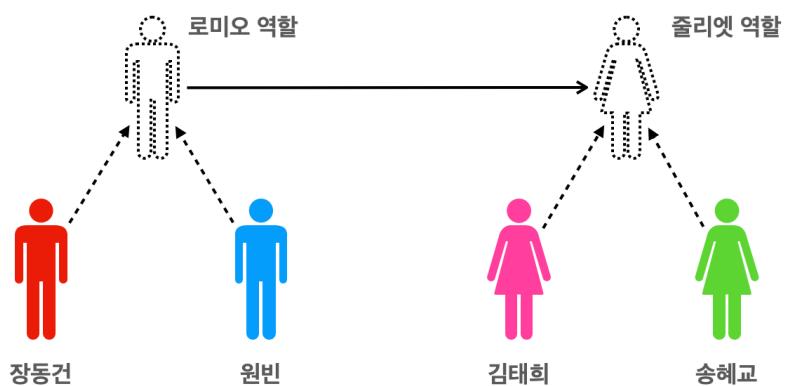
다형성의 실세계 비유

- 실세계와 객체 지향을 1:1로 매칭X
- 그래도 실세계의 비유로 이해하기에는 좋음
- 역할과 구현으로 세상을 구분

운전자 - 자동차



공연 무대 로미오와 줄리엣 공연



다형성의 실세계 비유

예시

- 운전자 - 자동차
- 공연 무대
- 키보드, 마우스, 세상의 표준 인터페이스들
- 정렬 알고리즘
- 할인 정책 로직

역할과 구현을 분리

- 역할과 구현으로 구분하면 세상이 단순해지고, 유연해지며 변경도 편리해진다.
- 장점
 - 클라이언트는 대상의 역할(인터페이스)만 알면 된다.
 - 클라이언트는 구현 대상의 내부 구조를 몰라도 된다.
 - 클라이언트는 구현 대상의 내부 구조가 변경되어도 영향을 받지 않는다.
 - 클라이언트는 구현 대상 자체를 변경해도 영향을 받지 않는다.

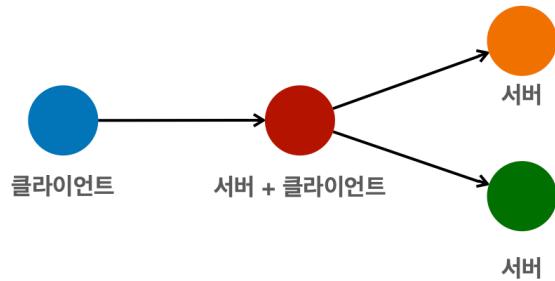
역할과 구현을 분리

자바 언어

- 자바 언어의 다형성을 활용
 - 역할 = 인터페이스
 - 구현 = 인터페이스를 구현한 클래스, 구현 객체
- 객체를 설계할 때 역할과 구현을 명확히 분리
- 객체 설계시 역할(인터페이스)을 먼저 부여하고, 그 역할을 수행하는 구현 객체 만들기

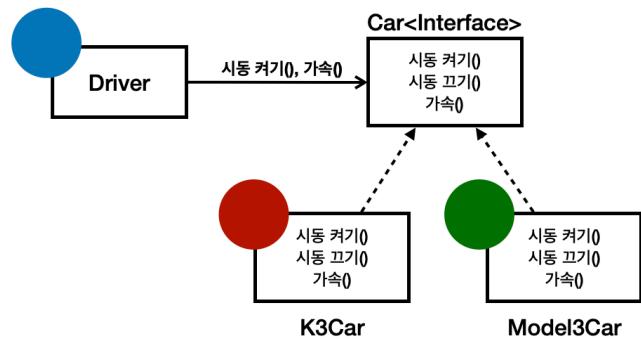
객체의 협력이라는 관계부터 생각

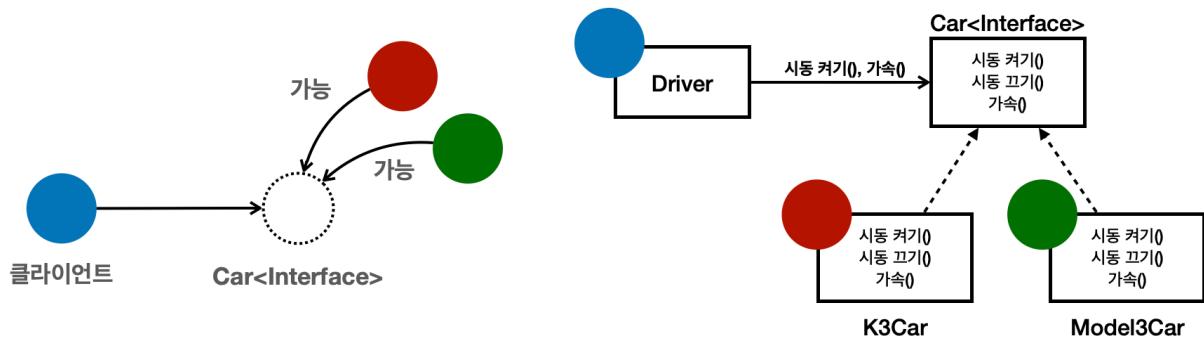
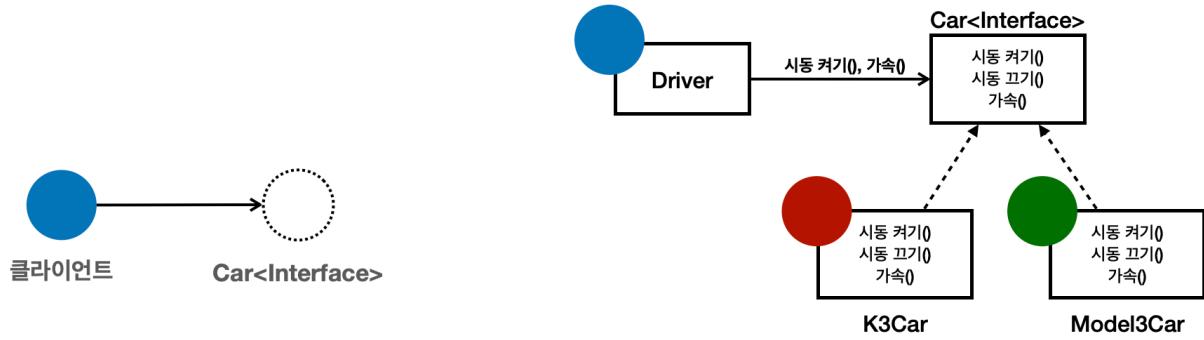
- 혼자 있는 객체는 없다.
- 클라이언트: 요청, 서버: 응답
- 수 많은 객체 클라이언트와 객체 서버는 서로 협력 관계를 가진다.

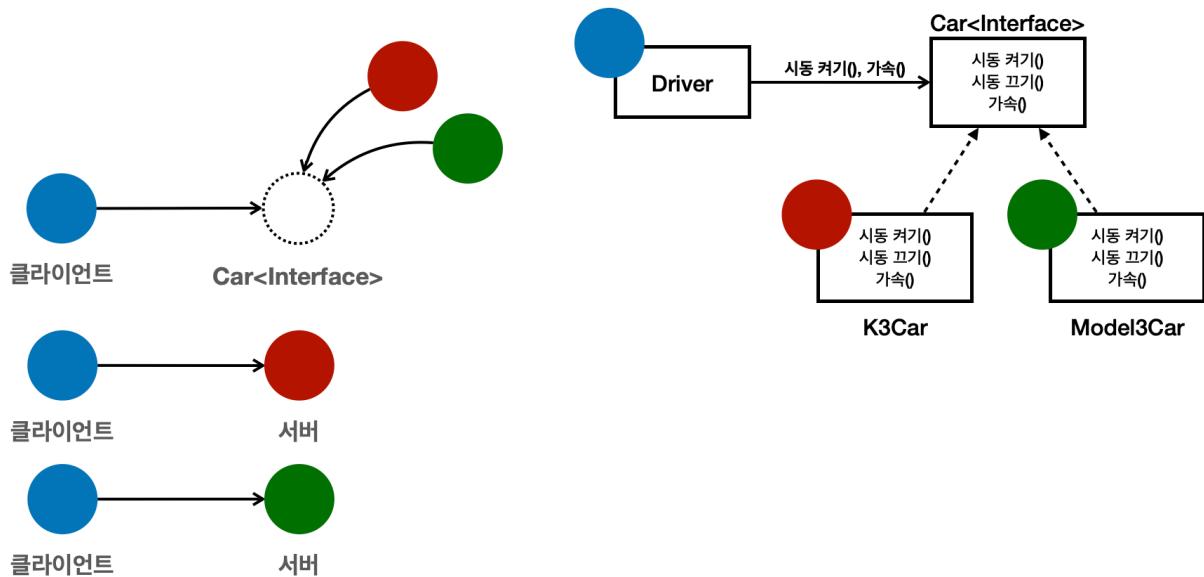
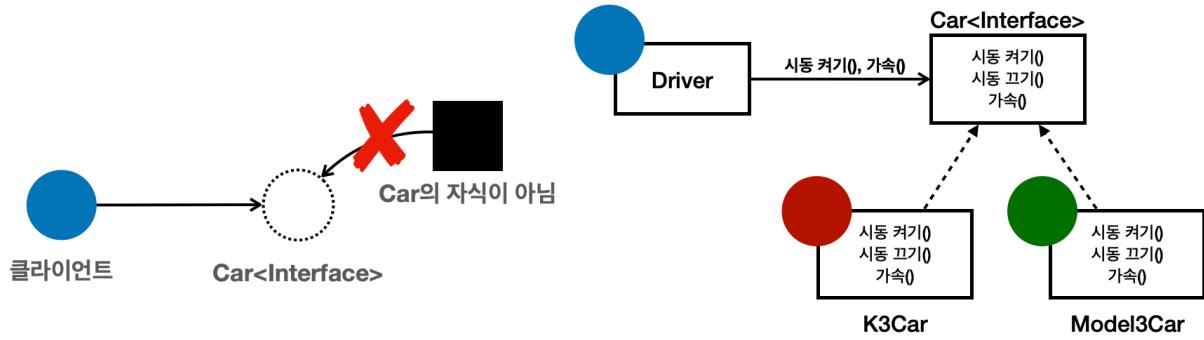


자바 언어의 다형성

- 오버라이딩을 떠올려보자
- 오버라이딩은 자바 기본 문법
- 오버라이딩 된 메서드가 실행
- 다형성으로 인터페이스를 구현한 객체를 실행 시점에 유연하게 변경할 수 있다.
- 물론 클래스 상속 관계도 다형성, 오버라이딩 적용 가능







다형성의 본질

- 인터페이스를 구현한 객체 인스턴스를 실행 시점에 유연하게 변경할 수 있다.
- 다형성의 본질을 이해하려면 협력이라는 객체사이의 관계에서 시작해야함
- 클라이언트를 변경하지 않고, 서버의 구현 기능을 유연하게 변경할 수 있다.

역할과 구현을 분리 정리

- 실세계의 역할과 구현이라는 편리한 컨셉을 다형성을 통해 객체 세상으로 가져올 수 있음
- 유연하고, 변경이 용이
- 확장 가능한 설계
- 클라이언트에 영향을 주지 않는 변경 가능
- 인터페이스를 안정적으로 잘 설계하는 것이 중요

역할과 구현을 분리 한계

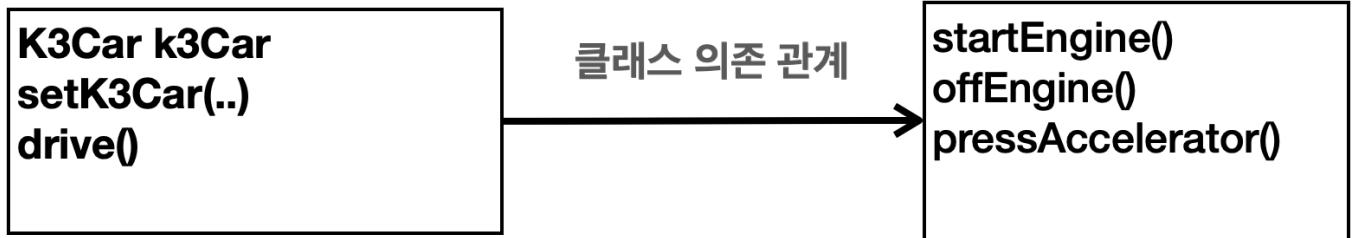
- 역할(인터페이스) 자체가 변하면, 클라이언트, 서버 모두에 큰 변경이 발생한다.
- 자동차를 비행기로 변경해야 한다면?
- 대본 자체가 변경된다면?
- USB 인터페이스가 변경된다면?
- 인터페이스를 안정적으로 잘 설계하는 것이 중요

정리

- 다형성이 가장 중요하다!
- 디자인 패턴 대부분은 다형성을 활용하는 것이다.
- 스프링의 핵심인 제어의 역전(IoC), 의존관계 주입(DI)도 결국 다형성을 활용하는 것이다.
- 스프링을 사용하면 마치 레고 블럭 조립하듯이! 공연 무대의 배우를 선택하듯이! 구현을 편리하게 변경할 수 있다.

다형성 - 역할과 구현 예제1

앞서 설명한 내용을 더 깊이있게 이해하기 위해, 간단한 운전자와 자동차의 관계를 개발해보자. 먼저 다형성을 사용하지 않고, 역할과 구현을 분리하지 않고 단순하게 개발해보자.



Driver는 **K3Car**를 운전하는 프로그램이다.

```

package poly.car0;

public class K3Car {
    public void startEngine() {
        System.out.println("K3Car.startEngine");
    }

    public void offEngine() {
        System.out.println("K3Car.offEngine");
    }

    public void pressAccelerator() {
        System.out.println("K3Car.pressAccelerator");
    }
}

```

```

package poly.car0;

public class Driver {

    private K3Car k3Car;

    public void setK3Car(K3Car k3Car) {
        this.k3Car = k3Car;
    }

    public void drive() {
        System.out.println("자동차를 운전합니다.");
        k3Car.startEngine();
        k3Car.pressAccelerator();
        k3Car.offEngine();
    }
}

```

```
    }  
}
```

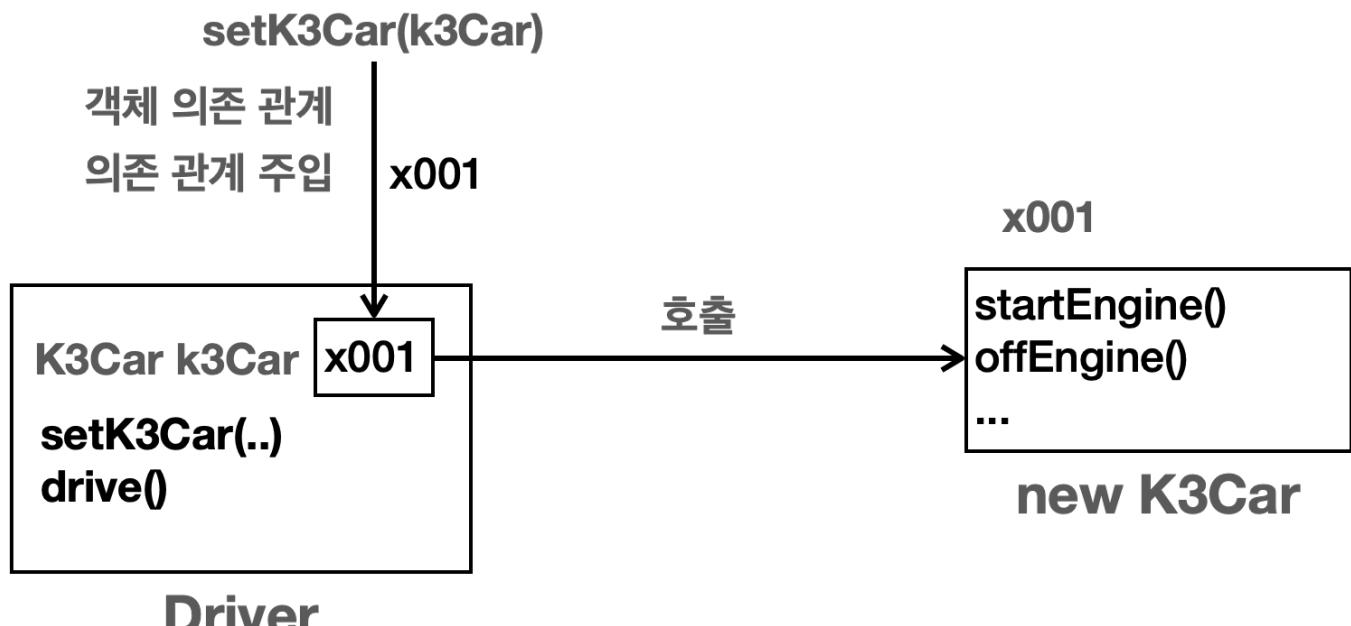
```
package poly.car0;  
  
public class CarMain0 {  
  
    public static void main(String[] args) {  
        Driver driver = new Driver();  
        K3Car k3Car = new K3Car();  
        driver.setK3Car(k3Car);  
        driver.drive();  
    }  
}
```

- `Driver`와 `K3Car`을 먼저 생성한다. 그리고 `driver.setK3Car(..)`를 통해 `driver`에게 `k3Car`의 참조를 넘겨준다.
- `driver.driver()`를 호출한다.

실행 결과

```
자동차를 운전합니다.  
K3Car.startEngine  
K3Car.pressAccelerator  
K3Car.offEngine
```

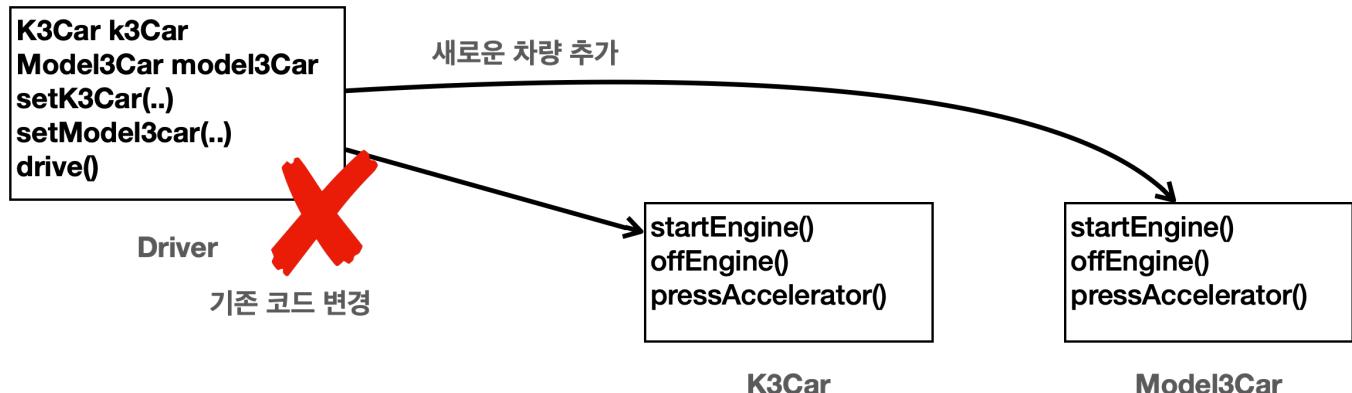
메모리 그림



다형성 - 역할과 구현 예제2

새로운 Model3 차량을 추가해야 하는 요구사항이 들어왔다. 이 요구사항을 맞추려면 기존에 `Driver` 코드를 많이 변경해야 한다.

드라이버는 `K3Car` 도 운전할 수 있고, `Model3Car` 도 운전할 줄 있어야 한다. 참고로 둘을 동시에 운전하는 것은 아니다.



```
package poly.car0;

public class Model3Car {
    public void startEngine() {
        System.out.println("Model3Car.startEngine");
    }

    public void offEngine() {
        System.out.println("Model3Car.offEngine");
    }

    public void pressAccelerator() {
        System.out.println("Model3Car.pressAccelerator");
    }
}
```

Driver - 코드 변경

```
package poly.car0;

public class Driver {

    private K3Car k3Car;
```

```

private Model3Car model3Car; //추가

public void setK3Car(K3Car k3Car) {
    this.k3Car = k3Car;
}

//추가
public void setModel3Car(Model3Car model3Car) {
    this.model3Car = model3Car;
}

//변경
public void drive() {
    System.out.println("자동차를 운전합니다.");
    if (k3Car != null) {
        k3Car.startEngine();
        k3Car.pressAccelerator();
        k3Car.offEngine();
    } else if (model3Car != null) {
        model3Car.startEngine();
        model3Car.pressAccelerator();
        model3Car.offEngine();
    }
}
}

```

드라이버는 K3Car, Model3Car를 모두 운전할 줄 알아야 한다. 다음과 같은 코드 변경이 발생한다.

- Model3Car 용 필드 추가
- setModel3Car(..) 메서드 추가
- drive() 메서드에서 가지고 있는 차량에 따른 분기

CarMain0 - 코드 변경

```

package poly.car0;

public class CarMain0 {

    public static void main(String[] args) {
        Driver driver = new Driver();
        K3Car k3Car = new K3Car();
        driver.setK3Car(k3Car);
        driver.drive();

        //추가
    }
}

```

```

        Model3Car model3Car = new Model3Car();
        driver.setK3Car(null);
        driver.setModel3Car(model3Car);
        driver.drive();
    }
}

```

- K3를 운전하던 운전자가 Model3로 차량을 변경해서 운전하는 코드이다.
- `driver.setK3Car(null)`을 통해서 기존 K3Car의 참조를 제거한다.
- `driver.setModel3Car(model3Car)`을 통해서 새로운 `model3Car`의 참조를 추가한다.
- `driver.driver()`를 호출한다.

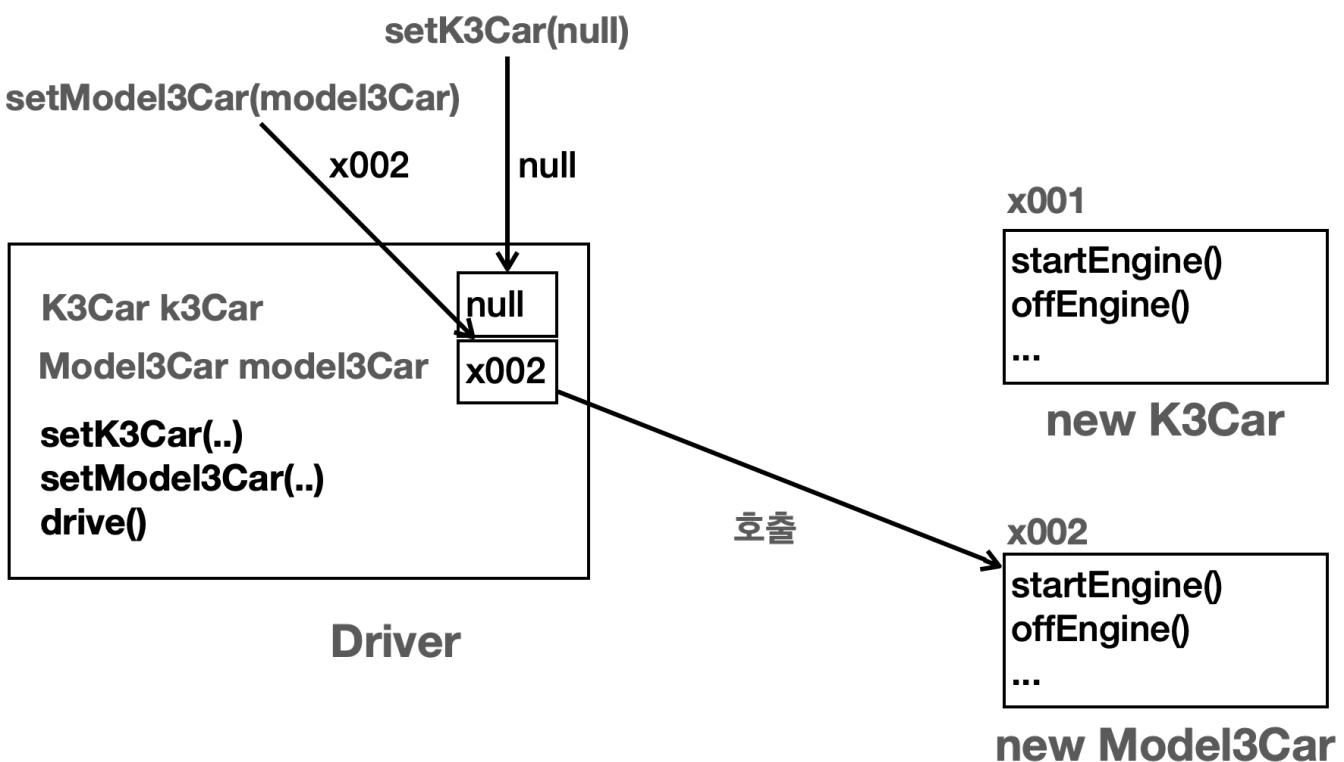
실행 결과

```

자동차를 운전합니다.
K3Car.startEngine
K3Car.pressAccelerator
K3Car.offEngine
자동차를 운전합니다.
Model3Car.startEngine
Model3Car.pressAccelerator
Model3Car.offEngine

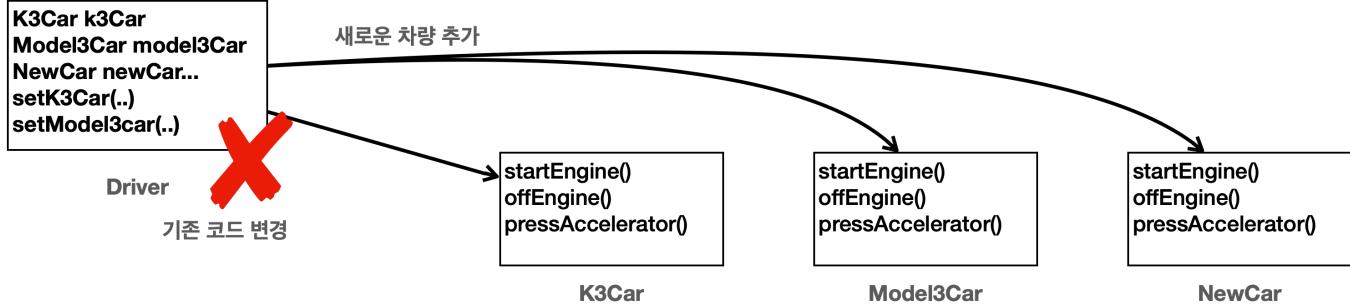
```

메모리 그림



여기서 새로운 차량을 추가한다면 또 다시 `Driver` 코드를 많이 변경해야 한다. 만약 운전할 수 있는 차량의 종류가 계

속 늘어난다면 점점 더 변경해야 하는 코드가 많아질 것이다.

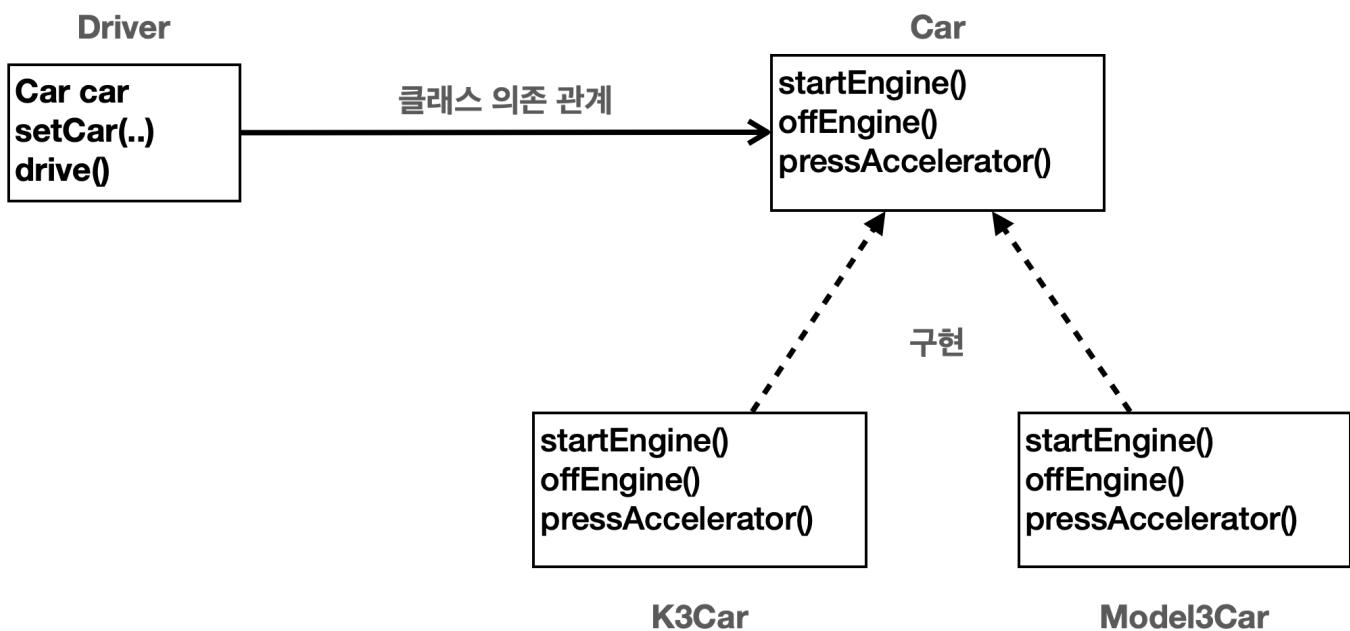


다형성 - 역할과 구현 예제3

다형성을 활용하면 역할과 구현을 분리해서, 클라이언트 코드의 변경 없이 구현 객체를 변경할 수 있다.

다음 관계에서 **Driver** 가 클라이언트이다.

예제를 통해서 자세히 알아보자.



앞서 설명한 자동차 예제를 코드로 구현해보자.

- **Driver**: 운전자는 자동차(Car)의 역할에만 의존한다. 구현인 K3, Model3 자동차에 의존하지 않는다.
 - **Driver** 클래스는 Car car 멤버 변수를 가진다. 따라서 Car 인터페이스를 참조한다.
 - 인터페이스를 구현한 K3Car, Model3Car에 의존하지 않고, Car 인터페이스에만 의존한다.
 - 여기서 설명하는 의존은 클래스 의존 관계를 뜻한다. 클래스 상에서 어떤 클래스를 알고 있는가를 뜻한다.
 Driver 클래스 코드를 보면 Car 인터페이스만 사용하는 것을 확인할 수 있다.
- **Car**: 자동차의 역할이고 인터페이스이다. K3Car, Model3Car 클래스가 인터페이스를 구현한다.

```
package poly.car1;

public interface Car {
    void startEngine();
    void offEngine();
    void pressAccelerator();
}
```

```
package poly.car1;

public class K3Car implements Car {
    @Override
    public void startEngine() {
        System.out.println("K3Car.startEngine");
    }

    @Override
    public void offEngine() {
        System.out.println("K3Car.offEngine");
    }

    @Override
    public void pressAccelerator() {
        System.out.println("K3Car.pressAccelerator");
    }
}
```

```
package poly.car1;

public class Model3Car implements Car {
    @Override
    public void startEngine() {
        System.out.println("Model3Car.startEngine");
    }

    @Override
    public void offEngine() {
        System.out.println("Model3Car.offEngine");
    }
}
```

```

@Override
public void pressAccelerator() {
    System.out.println("Model3Car.pressAccelerator");
}

}

```

```

package poly.car1;

public class Driver {

    private Car car;

    public void setCar(Car car) {
        System.out.println("자동차를 설정합니다: " + car);
        this.car = car;
    }

    public void drive() {
        System.out.println("자동차를 운전합니다.");
        car.startEngine();
        car.pressAccelerator();
        car.offEngine();
    }
}

```

- `Driver`는 멤버 변수로 `Car car`를 가진다.
- `setCar(Car car)`: 멤버 변수에 자동차를 설정한다. 외부에서 누군가 이 메서드를 호출해주어야 `Driver`는 새로운 자동차를 참조하거나 변경할 수 있다.
- `drive()`: `Car` 인터페이스가 제공하는 기능들을 통해 자동차를 운전한다.

```

package poly.car1;

/**
 * 다형성을 활용한 런타임 변경
 * 런타임: 애플리케이션 실행 도중에 변경 가능
 */
public class CarMain1 {
    public static void main(String[] args) {
        Driver driver = new Driver();
    }
}

```

```

//차량 선택(k3)
Car k3Car = new K3Car();
driver.setCar(k3Car);
driver.drive();

//차량 변경(k3 -> model3)
Car model3Car = new Model3Car();
driver.setCar(model3Car);
driver.drive();

}

}

```

실행 결과

자동차를 설정합니다: poly.car.K3Car@24d46ca6

자동차를 운전합니다.

K3Car.startEngine

K3Car.pressAccelerator

K3Car.offEngine

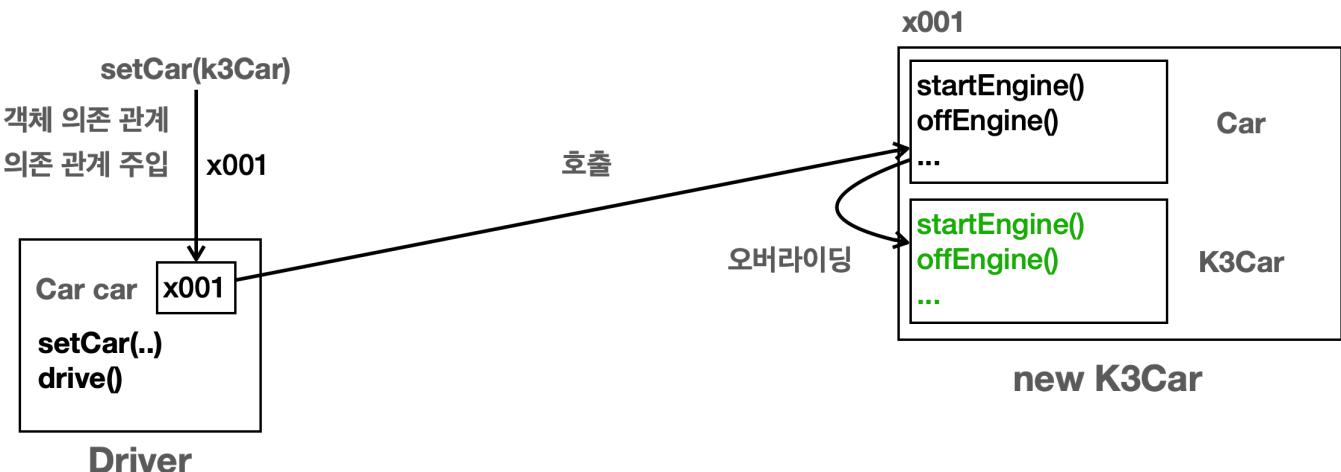
자동차를 설정합니다: poly.car.Model3Car@372f7a8d

자동차를 운전합니다.

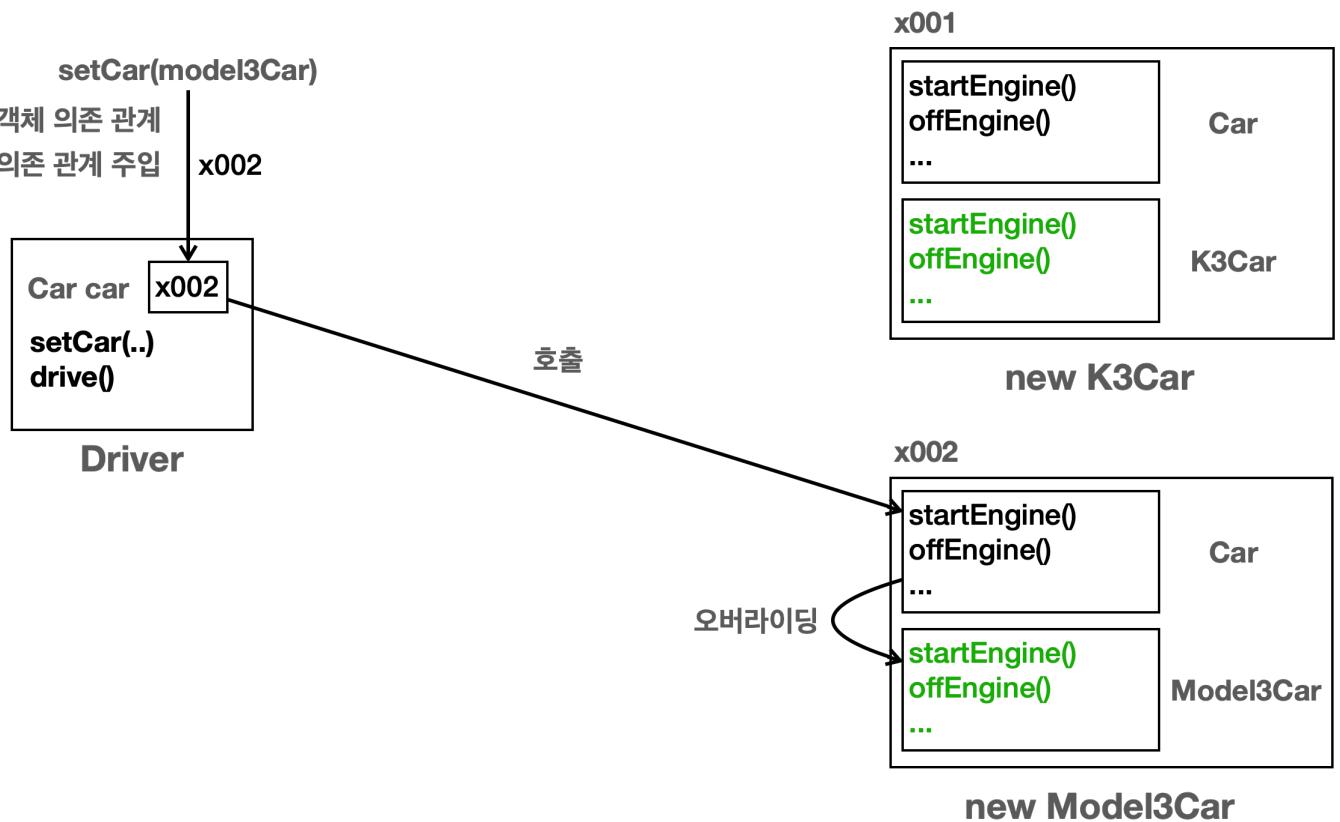
Model3Car.startEngine

Model3Car.pressAccelerator

Model3Car.offEngine



- 먼저 **Driver**와 **K3Car**를 생성한다.
- driver.setCar(k3Car)**를 호출해서 **Driver**의 **Car car** 필드가 **K3Car**의 인스턴스를 참조하도록 한다.
- driver.drive()**를 호출하면 **x001**을 참조한다. **car** 필드가 **Car** 타입이므로 **Car** 타입을 찾아서 실행 하지만 메서드 오버라이딩에 의해 **K3Car**의 기능이 호출된다.



- Model3Car 를 생성한다.
- driver.setCar(model3Car) 를 호출해서 Driver 의 Car car 필드가 Model3Car 의 인스턴스를 참조하도록 변경한다.
- driver.drive() 를 호출하면 x002 을 참조한다. car 필드가 Car 타입이므로 Car 타입을 찾아서 실행 하지만 메서드 오버라이딩에 의해 Model3Car 의 기능이 호출된다.

OCP(Open-Closed Principle) 원칙

좋은 객체 지향 설계 원칙 중 하나로 OCP 원칙이라는 것이다.

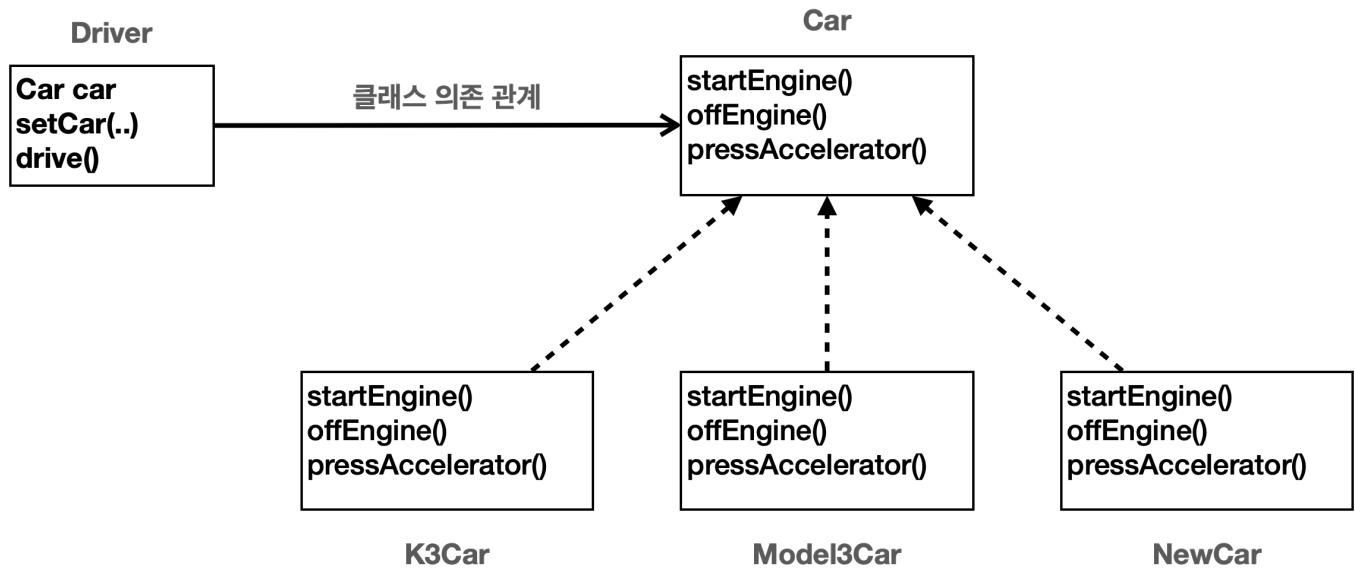
- **Open for extension:** 새로운 기능의 추가나 변경 사항이 생겼을 때, 기존 코드는 확장할 수 있어야 한다.
- **Closed for modification:** 기존의 코드는 수정되지 않아야 한다.

확장에는 열려있고, 변경에는 닫혀 있다는 뜻인데, 쉽게 이야기해서 기존의 코드 수정 없이 새로운 기능을 추가할 수 있다는 의미다. 약간 말이 안 맞는 것 같지만 우리가 앞서 개발한 코드가 바로 OCP 원칙을 잘 지키고 있는 코드다.

새로운 차량의 추가

여기서 새로운 차량을 추가해도 Driver 의 코드는 전혀 변경하지 않는다. 운전할 수 있는 차량의 종류가 계속 늘어나

도 Car 를 사용하는 Driver 의 코드는 전혀 변경하지 않는다. 기능을 확장해도 main() 일부를 제외한 프로그램의 핵심 부분의 코드는 전혀 수정하지 않아도 된다.



확장에 열려있다는 의미

Car 인터페이스를 사용해서 새로운 차량을 자유롭게 추가할 수 있다. Car 인터페이스를 구현해서 기능을 추가할 수 있다는 의미이다. 그리고 Car 인터페이스를 사용하는 클라이언트 코드인 Driver 도 Car 인터페이스를 통해 새롭게 추가된 차량을 자유롭게 호출할 수 있다. 이것이 확장에 열려있다는 의미이다.

코드 수정은 닫혀 있다는 의미

새로운 차를 추가하게 되면 기능이 추가되기 때문에 기존 코드의 수정은 불가피하다. 당연히 어딘가의 코드는 수정해야 한다.

변하지 않는 부분

새로운 자동차를 추가할 때 가장 영향을 받는 중요한 클라이언트는 바로 Car 의 기능을 사용하는 Driver 이다. 핵심은 Car 인터페이스를 사용하는 클라이언트인 Driver 의 코드를 수정하지 않아도 된다는 뜻이다.

변하는 부분

main() 과 같이 새로운 차를 생성하고 Driver 에게 필요한 차를 전달해주는 역할은 당연히 코드 수정이 발생한다. main() 은 전체 프로그램을 설정하고 조율하는 역할을 한다. 이런 부분은 OCP를 지켜도 변경이 필요하다.

정리

- Car 를 사용하는 클라이언트 코드인 Driver 코드의 변경없이 새로운 자동차를 확장할 수 있다.
- 다양성을 활용하고 역할과 구현을 잘 분리한 덕분에 새로운 자동차를 추가해도 대부분의 핵심 코드들을 그대로 유지할 수 있게 되었다.

전략 패턴(Strategy Pattern)

디자인 패턴 중에 가장 중요한 패턴을 하나 뽑으라고 하면 전략 패턴을 뽑을 수 있다. 전략 패턴은 알고리즘을 클라이언트 코드의 변경 없이 쉽게 교체할 수 있다. 방금 설명한 코드가 바로 전략 패턴을 사용한 코드이다. `Car` 인터페이스가 바로 전략을 정의하는 인터페이스가 되고, 각각의 차량이 전략의 구체적인 구현이 된다. 그리고 전략을 클라이언트 코드(`Driver`)의 변경 없이 손쉽게 교체할 수 있다.

문제와 풀이

문제1: 다중 메시지 발송

한번에 여러 곳에 메시지를 발송하는 프로그램을 개발하자.

다음 코드를 참고해서 클래스를 완성하자

요구사항

- 다형성을 활용하세요.
- `Sender` 인터페이스를 사용하세요.
- `EmailSender`, `SmsSender`, `FaceBookSender`를 구현하세요.

```
package poly.ex.sender;

public class SendMain {

    public static void main(String[] args) {
        Sender[] senders = {new EmailSender(), new SmsSender(), new
FaceBookSender()};
        for (Sender sender : senders) {
            sender.sendMessage("환영합니다!");
        }
    }
}
```

실행 결과

메일을 발송합니다: 환영합니다!

SMS를 발송합니다: 환영합니다!

페이스북에 발송합니다: 환영합니다!

정답

```
package poly.ex.sender;

public interface Sender {
    void sendMessage(String message);
}
```

```
package poly.ex.sender;

public class EmailSender implements Sender {
    @Override
    public void sendMessage(String message) {
        System.out.println("메일을 발송합니다: " + message);
    }
}
```

```
package poly.ex.sender;

public class SmsSender implements Sender {
    @Override
    public void sendMessage(String message) {
        System.out.println("SMS를 발송합니다: " + message);
    }
}
```

```
package poly.ex.sender;

public class FaceBookSender implements Sender {
    @Override
    public void sendMessage(String message) {
        System.out.println("페이스북에 발송합니다: " + message);
    }
}
```

문제2: 결제 시스템 개발

여러분은 기대하던 결제 시스템 개발팀에 입사하게 되었다.

이 팀은 현재 2가지 결제 수단을 지원한다. 앞으로 5개의 결제 수단을 추가로 지원할 예정이다.

새로운 결제수단을 쉽게 추가할 수 있도록, 기존 코드를 리펙토링해라.

요구사항

- OCP 원칙을 지키세요.
- 메서드를 포함한 모든 코드를 변경해도 됩니다. 클래스나 인터페이스를 추가해도 됩니다.
- 단 프로그램을 실행하는 `PayMain0` 코드는 변경하지 않고, 그대로 유지해야 합니다.
- 리펙토링 후에도 실행 결과는 기존과 같아야 합니다.

```
package poly.ex.pay0;

public class KakaoPay {

    public boolean pay(int amount) {
        System.out.println("카카오페이지 시스템과 연결합니다.");
        System.out.println(amount + "원 결제를 시도합니다.");
        return true;
    }
}
```

```
package poly.ex.pay0;

public class NaverPay {

    public boolean pay(int amount) {
        System.out.println("네이버페이지 시스템과 연결합니다.");
        System.out.println(amount + "원 결제를 시도합니다.");
        return true;
    }
}
```

```
package poly.ex.pay0;

public class PayService {

    public void processPay(String option, int amount) {
        boolean result;
```

```

        System.out.println("결제를 시작합니다: option=" + option + ", amount=" +
amount);
        if (option.equals("kakao")) {
            KakaoPay kakaoPay = new KakaoPay();
            result = kakaoPay.pay(amount);
        } else if (option.equals("naver")) {
            NaverPay naverPay = new NaverPay();
            result = naverPay.pay(amount);
        } else {
            System.out.println("결제 수단이 없습니다.");
            result = false;
        }

        if (result) {
            System.out.println("결제가 성공했습니다.");
        } else {
            System.out.println("결제가 실패했습니다.");
        }
    }

}

```

```

package poly.ex.pay0;

public class PayMain0 {

    public static void main(String[] args) {
        PayService payService = new PayService();

        //kakao 결제
        String payOption1 = "kakao";
        int amount1 = 5000;
        payService.processPay(payOption1, amount1);

        //naver 결제
        String payOption2 = "naver";
        int amount2 = 10000;
        payService.processPay(payOption2, amount2);

        //잘못된 결제 수단 선택
        String payOption3 = "bad";
        int amount3 = 15000;
        payService.processPay(payOption3, amount3);
    }
}

```

```
}
```

```
}
```

실행 결과

```
결제를 시작합니다: option=kakao, amount=5000
```

```
카카오페이지 시스템과 연결합니다.
```

```
5000원 결제를 시도합니다.
```

```
결제가 성공했습니다.
```

```
결제를 시작합니다: option=naver, amount=10000
```

```
네이버페이 시스템과 연결합니다.
```

```
10000원 결제를 시도합니다.
```

```
결제가 성공했습니다.
```

```
결제를 시작합니다: option=bad, amount=15000
```

```
결제 수단이 없습니다.
```

```
결제가 실패했습니다.
```

정답

이 문제에 정답은 없습니다.

새로운 결제 수단을 추가했을 때 Pay 를 사용하는 클라이언트 코드인 PayService 의 변경을 최소화 할 수 있다면 성공입니다.

```
package poly.ex.pay1;

public interface Pay {
    boolean pay(int amount);
}
```

```
package poly.ex.pay1;

public class KakaoPay implements Pay {
    @Override
    public boolean pay(int amount) {
        System.out.println("카카오페이지 시스템과 연결합니다.");
        System.out.println(amount + "원 결제를 시도합니다.");
        return true;
}
}
```

```

package poly.ex.pay1;

public class NaverPay implements Pay {
    @Override
    public boolean pay(int amount) {
        System.out.println("네이버페이 시스템과 연결합니다.");
        System.out.println(amount + "원 결제를 시도합니다.");
        return true;
    }
}

```

```

package poly.ex.pay1;

public class DefaultPay implements Pay {
    @Override
    public boolean pay(int amount) {
        System.out.println("결제 수단이 없습니다.");
        return false;
    }
}

```

- 결제 수단을 찾지 못했을 때 사용한다. `DefaultPay` 사용하면 `null` 체크를 피할 수 있다.

```

package poly.ex.pay1;

public abstract class PayStore {

    //결제 수단 추가시 변하는 부분
    public static Pay findPay(String option) {
        if (option.equals("kakao")) {
            return new KakaoPay();
        } else if (option.equals("naver")) {
            return new NaverPay();
        } else {
            return new DefaultPay();
        }
    }
}

```

- 결제 수단 이름으로 실제 결제 수단 구현체를 찾는다.
- `static`으로 기능을 제공한다. 추상 클래스로 선언해서 객체 생성을 막는다.
- 결제 수단을 찾지 못했을 때 `null` 대신에 항상 실패하는 결제 수단을 사용한다.

```

package poly.ex.pay1;

public class PayService {

    public void processPay(String option, int amount) {
        System.out.println("결제를 시작합니다: option=" + option + ", amount=" +
amount);

        Pay pay = PayStore.findPay(option);
        boolean result = pay.pay(amount);

        if (result) {
            System.out.println("결제가 성공했습니다.");
        } else {
            System.out.println("결제가 실패했습니다.");
        }
    }

}

```

- `PayService`는 구체적인 결제 수단이 아니라 `Pay`에 의존한다.
- 따라서 결제수단을 추가해도 `PayService`의 코드에는 변경이 없다.

```

package poly.ex.pay1;

public class PayMain1 {

    public static void main(String[] args) {
        PayService payService = new PayService();

        //kakao 결제
        String payOption1 = "kakao";
        int amount1 = 5000;
        payService.processPay(payOption1, amount1);

        //naver 결제
        String payOption2 = "naver";
        int amount2 = 10000;
        payService.processPay(payOption2, amount2);

        //잘못된 결제 수단 선택
        String payOption3 = "bad";
    }
}

```

```
    int amount3 = 15000;
    payService.processPay(payOption3, amount3);
}

}
```

실행 결과

```
결제를 시작합니다: option=kakao, amount=5000
카카오페이지 시스템과 연결합니다.
5000원 결제를 시도합니다.
결제가 성공했습니다.

결제를 시작합니다: option=naver, amount=10000
네이버페이 시스템과 연결합니다.
10000원 결제를 시도합니다.
결제가 성공했습니다.

결제를 시작합니다: option=bad, amount=15000
결제 수단이 없습니다.
결제가 실패했습니다.
```

문제3: 결제 시스템 개발 - 사용자 입력

기존 결제 시스템이 사용자 입력을 받도록 수정하세요.

```
결제 수단을 입력하세요:kakao
결제 금액을 입력하세요:5000
결제를 시작합니다: option=kakao, amount=5000
카카오페이지 시스템과 연결합니다.
5000원 결제를 시도합니다.
결제가 성공했습니다.

결제 수단을 입력하세요:exit
프로그램을 종료합니다.
```

정답

```
package poly.ex.pay1;

import java.util.Scanner;

public class PayMain2 {

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

```
Scanner scanner = new Scanner(System.in);
PayService payService = new PayService();

while (true) {
    System.out.print("결제 수단을 입력하세요:");
    String payOption = scanner.nextLine();
    if (payOption.equals("exit")) {
        System.out.println("프로그램을 종료합니다.");
        return;
    }
    System.out.print("결제 금액을 입력하세요:");
    int amount = scanner.nextInt();
    scanner.nextLine();

    payService.processPay(payOption, amount);
}
}
```

정리

13. 다음으로

#1.인강/0.자바/2.자바-기본

학습 내용 정리

전체 목차

1. 클래스와 데이터

- /프로젝트 환경 구성
- /클래스가 필요한 이유
- /클래스 도입
- /객체 사용
- /클래스, 객체, 인스턴스 정리
- /배열 도입 - 시작
- /배열 도입 - 리펙토링
- /문제와 풀이
- /정리

2. 기본형과 참조형

- /기본형 vs 참조형1 - 시작
- /기본형 vs 참조형2 - 변수 대입
- /기본형 vs 참조형3 - 메서드 호출
- /참조형과 메서드 호출 - 활용
- /변수와 초기화
- /null
- /NullPointerException
- /문제와 풀이
- /정리

3. 객체 지향 프로그래밍

- /절차 지향 프로그래밍1 - 시작

- /절차 지향 프로그래밍2 - 데이터 묶음
- /절차 지향 프로그래밍3 - 메서드 추출
- /클래스와 메서드
- /객체 지향 프로그래밍
- /문제와 풀이
- /정리

4. 생성자

- /생성자 - 필요한 이유
- /this
- /생성자 - 도입
- /기본 생성자
- /생성자 - 오버로딩과 this()
- /문제와 풀이
- /정리

5. 패키지

- /패키지 - 시작
- /패키지 - import
- /패키지 규칙
- /패키지 활용
- /정리

6. 접근 제어자

- /접근 제어자 이해1
- /접근 제어자 이해2
- /접근 제어자 종류
- /접근 제어자 사용 - 필드, 메서드
- /접근 제어자 사용 - 클래스 레벨
- /캡슐화
- /문제와 풀이
- /정리

7. 자바 메모리 구조와 static

- /자바 메모리 구조
- /스택과 큐 자료 구조
- /스택 영역
- /스택 영역과 힙 영역
- /static 변수1
- /static 변수2
- /static 변수3
- /static 메서드1
- /static 메서드2
- /static 메서드3
- /문제와 풀이
- /정리

8. final

- /final 변수와 상수1
- /final 변수와 상수2
- /final 변수와 참조
- /정리

9. 상속

- /상속 - 시작
- /상속 관계
- /상속과 메모리 구조
- /상속과 기능 추가
- /상속과 메서드 오버라이딩
- /상속과 접근 제어
- /super - 부모 참조
- /super - 생성자
- /문제와 풀이
- /정리

10. 다형성1

- /다형성 시작
- /다형성과 캐스팅
- /캐스팅의 종류
- /다운캐스팅과 주의점
- /instanceof
- /다형성과 메서드 오버라이딩
- /정리

11. 다형성2

- /다형성 활용1
- /다형성 활용2
- /다형성 활용3
- /추상 클래스1
- /추상 클래스2
- /인터페이스
- /인터페이스 - 다중 구현
- /클래스와 인터페이스 활용
- /정리

12. 다형성과 설계

- /좋은 객체 지향 프로그래밍이란?
- /다형성 - 역할과 구현 예제1
- /다형성 - 역할과 구현 예제2
- /다형성 - 역할과 구현 예제3
- /OCP(Open-Closed Principle) 원칙
- /문제와 풀이
- /정리

로드맵 소개

실전 자바 로드맵

- 김영한의 자바 입문 - 코드로 시작하는 자바 첫걸음 (오픈)
- 김영한의 실전 자바 - 기본편 (오픈)
- 김영한의 실전 자바 - 중급편 (2024년 초 예정)
- 김영한의 실전 자바 - 고급편 (2024년 초 예정)

실전 데이터베이스 로드맵(2024년 중순 예정)

백엔드 개발자 로드맵 소개



김영한 백엔드 개발자 자바 스프링 JPA 실무 로드맵

백엔드 개발자 로드맵 소개 영상 링크: <https://youtu.be/ZgtvcyH58ys>

스프링 완전 정복 로드맵

- 스프링을 완전히 마스터 할 수 있는 로드맵
- URL: <https://www.inflearn.com/roadmaps/373>

스프링 부트와 JPA 실무 완전 정복 로드맵

- 최신 실무 기술로 웹 애플리케이션을 만들어보면서 학습
- URL: <https://www.inflearn.com/roadmaps/149>

하고 싶은 이야기

객체 지향 개념 이해?

사실 객체 지향 개념을 이해하는게 쉬운 것이 아님

실무에 있는 개발자들도 제대로 이해하지 못하는 분들이 상당히 많음

이번 강의가 80% 이상 이해가 된다면 프로그래머로 적성이 확실히 있음

이해가 어려운 분은 한번 정도 복습하는 것을 추천(아직 익숙하지 않을 수 있음)

- 프로그래밍을 처음 공부한다면 한번에 이해가 안되는 경우가 많음, 실제로 익숙해지는 과정이 필요
- 많은 개발자가 복습하고 정리하는 단계에서 이해

물론 실무를 하려면 앞으로 공부할 내용이 많으니 꾸준한 노력이 뒷받침 되어야 한다.

실무 개발자가 되는 학습 방향

프로그래밍 언어를 배우는 3단계

- 기본 문법 및 개념 이해 - **자바 입문편**
 - 기본 문법, 변수, 타입, 조건문, 반복문, 함수 사용법 등
- 고급 개념과 해당 언어의 라이브러리 활용 - **자바 기본편 ~ 고급편**
 - 객체지향 개념
 - 고급 문법, 자바의 주요 라이브러리
- 프레임워크 및 생태계 탐색 → **백엔드 개발자 로드맵**
 - 스프링 프레임워크를 포함한 다양한 오픈소스 학습
 - 실제 프로젝트나 업무에서의 적용에 중점
 - 개발 생태계 내에서의 베스트 프랙티스
 - 어떤 프레임워크를 사용하고, 어떤 라이브러리를 조합해서 사용하는 것이 좋은지 등등
 - 어떤 스타일로 코딩하는게 좋은지 등등

스프링?

- 백엔드 웹 애플리케이션은 대부분 스프링을 사용
- 객체 지향 개념을 편리하게 사용하도록 도와주는 도구 + 알파
- 자바가 이렇게 잘 되는 이유는 80% 정도는 스프링 덕분
- 수많은 베스트 프렉티스가 스프링에 포함되어 있음, 사실 알파가 엄청 큼
 - 수많은 공통 기능을 이미 만들어둠
 - 수많은 오픈 소스 라이브러리를 중앙에서 통합해서 관리
 - 좋은 베스트 프렉티스 제공

스프링을 학습하기 전에 준비할 것?

자바 로드맵 → 데이터베이스 로드맵 → 스프링

하지만 취업 준비 등으로 스프링 공부를 빨리해야 하는 상황이라면?

백엔드 개발자 로드맵에 진입하기 전에 최소한 다음 내용 정도 추가 학습하고 넘어가기

- 자바 예외 처리, 자바 컬렉션
- 데이터베이스는 스프링 로드맵 진행 도중에 데이터베이스 관련된 부분이 있을 때 학습

하고 싶은 이야기 정리 링크

개발 인생 전반의 이야기

EO 인터뷰 영상

- 한국 개발자 최고 1타강사 김영한의 인생 [1부]: https://youtu.be/_HTj5b59Em0
- 한국 개발자 최고 1타강사 김영한의 인생 [2부]: <https://youtu.be/MNyNRraMU8Y>

개발바닥 - 시골 청년 개발왕 되다

- 1편: <https://youtu.be/Pb69UQ6f8n0>
- 2편: <https://youtu.be/b4QP5RsuJts>
- 3편: <https://youtu.be/l0h1pQ96u2g>

취업과 이직에 대한 고민

인프콘 - 어느 날 고민 많은 주니어 개발자가 찾아왔다, 성장과 취업, 이직 이야기

- <https://youtu.be/QHlyr8soUDM>

인프런 최초 20만 명 달성 기념 QA

- <https://youtu.be/psXdWq008DA>