객체지향 프로그래밍

Object-Oriented Programming

Chapter 04. 객체 지향

- 객체 지향의 개요
- 객체 지향 프로그래밍의 특징
- 클래스의 선언과 객체 생성
- 클래스의 구성 요소와 멤버 접근
- 접근자와 설정자
- 생성자
- 정적 멤버

객체 지향의 개요

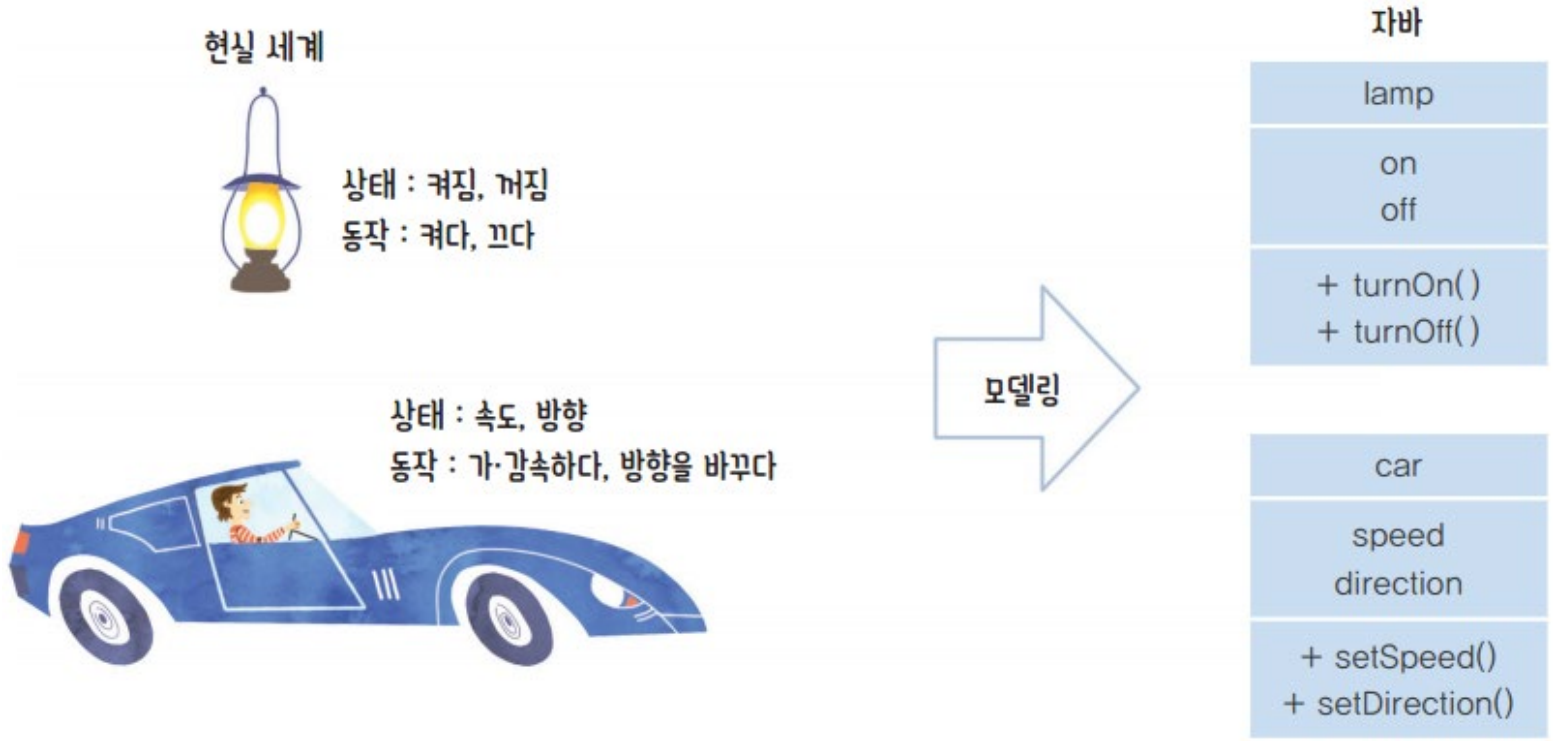
객체 (Object)

- **현실 세계에서의 객체:** 구체적이거나 추상적인 사물(개념)
  - 동작 (behavior)
  - 상태 (state)

객체 지향의 개요

객체 (Object)

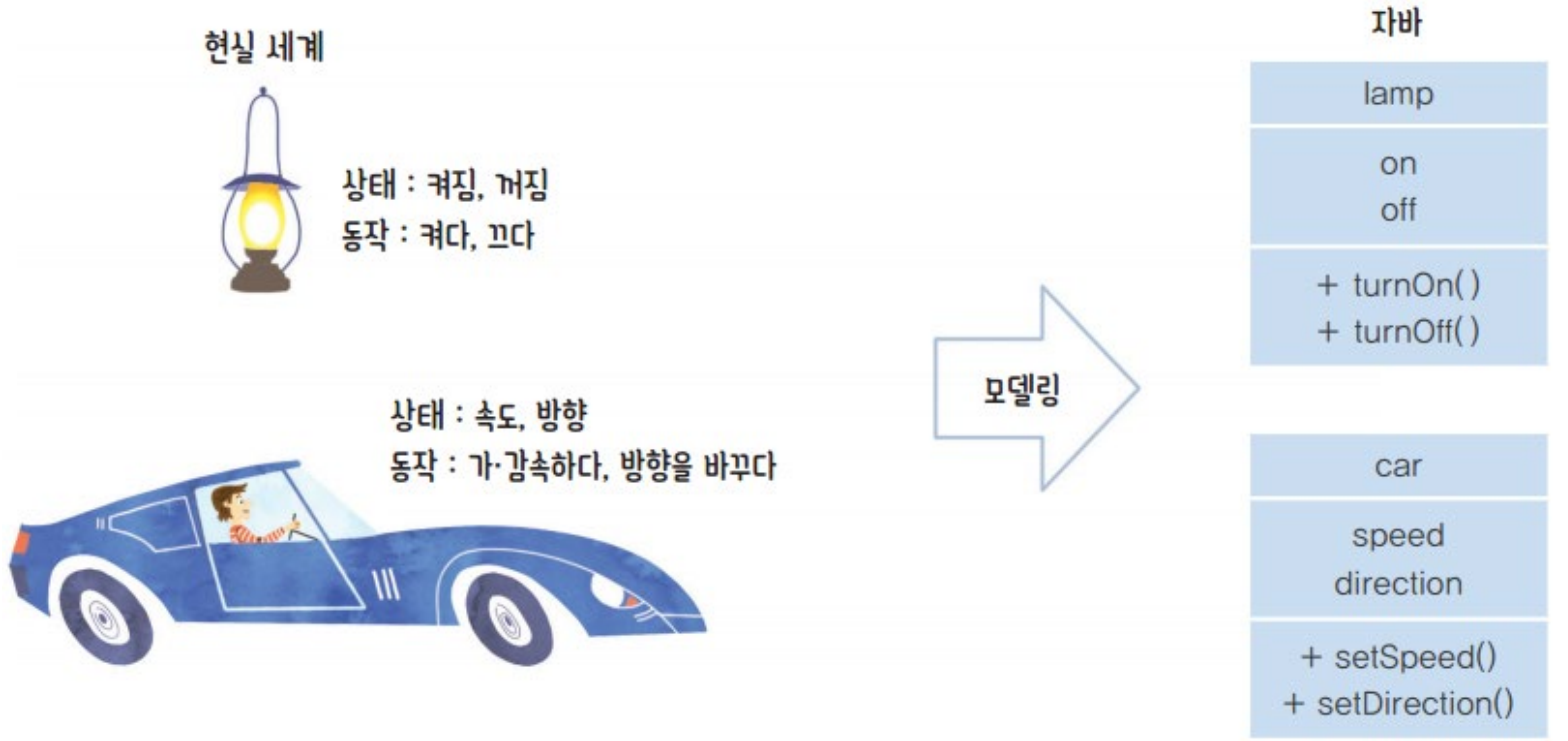
- **소프트웨어 객체:** 현실 세계의 객체를 필드와 메서드로 모델링한 것.
  - 상태 (state) → 필드 (field)
  - 동작 (behavior) → 메서드 (method)



객체 지향의 개요

객체 (Object)

- **소프트웨어 객체:** 현실 세계의 객체를 필드와 메서드로 모델링한 것.
  - 필드 (field): 객체를 통해 사용하는 변수
  - 메서드 (method): 객체를 통해 호출하는 동작



객체 지향의 개요

객체 지향 프로그래밍 (Object-oriented programming)

- 객체를 기반으로 하는 프로그래밍 패러다임

- 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나
  여러 개의 독립된 단위, 즉 '객체들의 모임'으로 파악하고자 하는 것

- 여러 가지 객체의 상호 작용을 통해 문제를 해결

객체 지향의 개요

절차 지향 프로그래밍 (procedural programming)

- 일련의 동작을 순서에 맞추어 단계적으로 실행하도록 명령어를 나열
- 데이터를 정의하는 방법보다는 명령어의 순서와 흐름에 중점
- 데이터와 동작이 분리됨

- C, BASIC, Fortran 등의 언어

객체 지향의 개요

절차 지향 프로그래밍의 한계

- 소프트웨어의 규모가 커지면서 서로 복잡하게 얽혀 있는 데이터간 관계를 파악하는데 어려움이 발생

- 변경, 확장의 어려움

- 현실 세계의 작업은 절차나 과정 보다는 다양한 물체의 상호작용으로 표현하는 것이 이해하기 쉬움
  → 이런 특성을 고려해 등장한 것이 객체 지향 프로그래밍

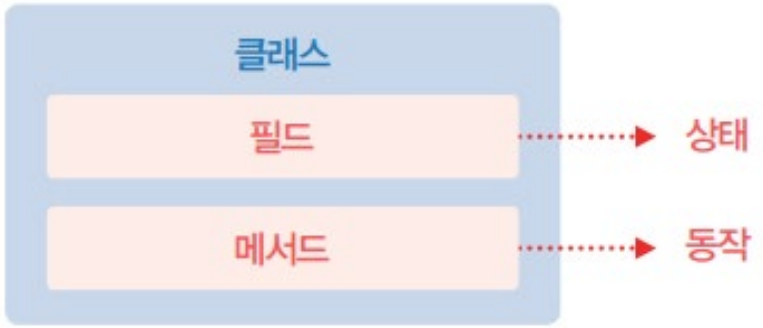
객체 지향의 개요

객체와 클래스

- 객체 지향 프로그래밍에서는 현실 세계를 객체 단위로 프로그래밍한다.
- **클래스(class)**는 객체의 설계도

형틀 = 클래스

제품 = 객체



객체 지향의 개요

인스턴스(instance)

- **인스턴스:** 클래스라는 틀로 만든 객체
- **인스턴스화**: 클래스에서 객체를 생성하는 과정

객체 지향 프로그래밍의 특징

주요 특징

- **캡슐화**
- **상속**
- **다형성**

객체 지향 프로그래밍의 특징

캡슐화 (encapsulation)

- 관련된 필드와 메서드를 하나의 캡슐처럼 포장해

  외부에서 알 수 없도록 감추는 것

= 정보 은닉(information hiding)

객체 지향 프로그래밍의 특징

캡슐화의 이점

- 세부 구현을 숨김으로써 외부 영향을 줄인다.

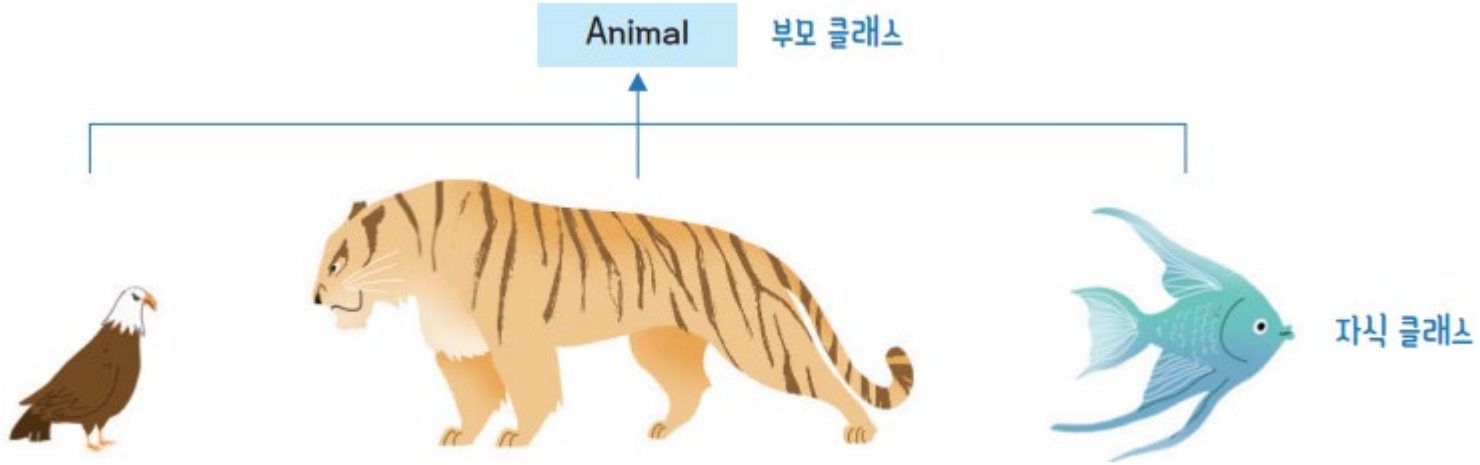
- 외부로부터 특정 메서드, 필드에 대한 접근을 제한.
  - ➢ 의도치 않은 동작(오류) 방지.

- 중요한 정보만을 외부에 공개함으로써 프로그램을 단순화.

객체 지향 프로그래밍의 특징

상속 (inheritance)

- **한 클래스가 기존의 클래스의 필드와 메서드를 이용할 수 있게 하는 기능**
- 코드 재사용성 향상
- 객체 조직화 가능



객체 지향 프로그래밍의 특징

다형성 (polymorphism)

- **객체에 따라서 메서드를 다르게 동작하도록 구현하는 기술**
  - 오버로딩
  - 오버라이딩

클래스 선언과 객체 생성

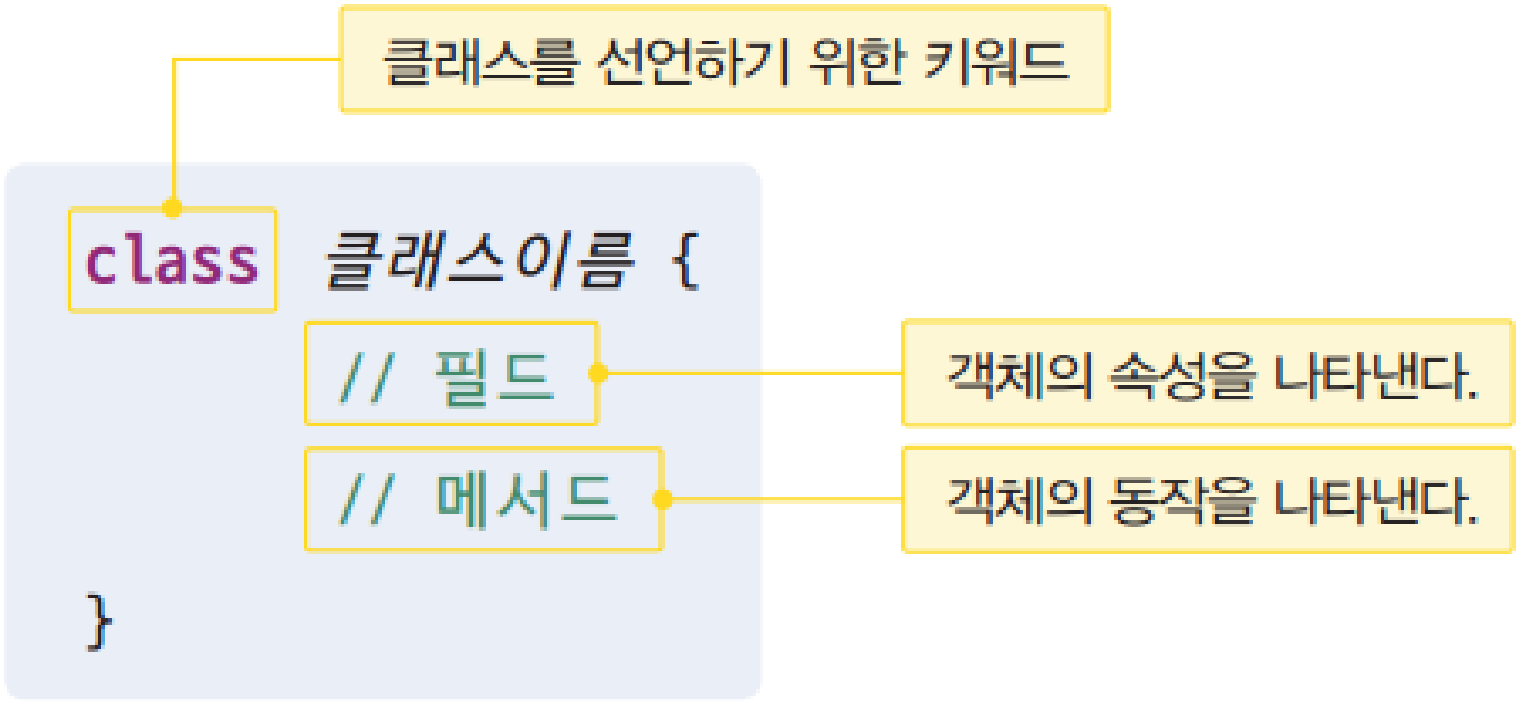
추상화 (abstraction)

- 현실 세계의 객체는 수많은 상태가 있고 다양한 동작을 하지만,
  클래스에 모두 포함하기는 어렵기에 추상화하는 과정이 필요
- 추상화는 현실 세계의 객체에서 불필요한 속성을 제거하고
  중요한 정보만 클래스로 표현하는 일종의 모델링 기법
- 사람마다 추상화하는 기법이 같지 않으므로 각 개발자는
  클래스를 다르게 정의 가능

클래스 선언과 객체 생성

클래스 선언

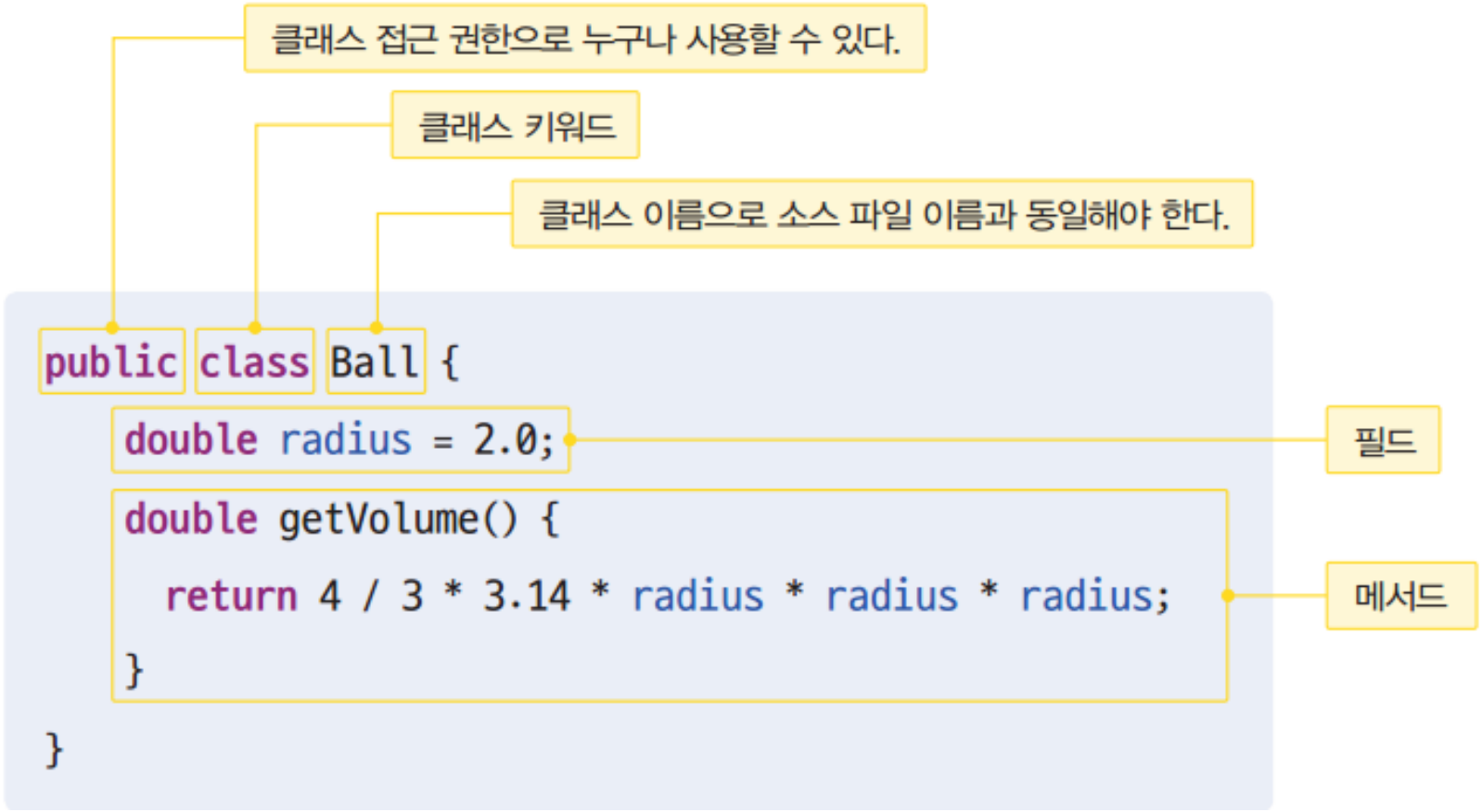
- **형식**



클래스 선언과 객체 생성

클래스 선언

- 예시

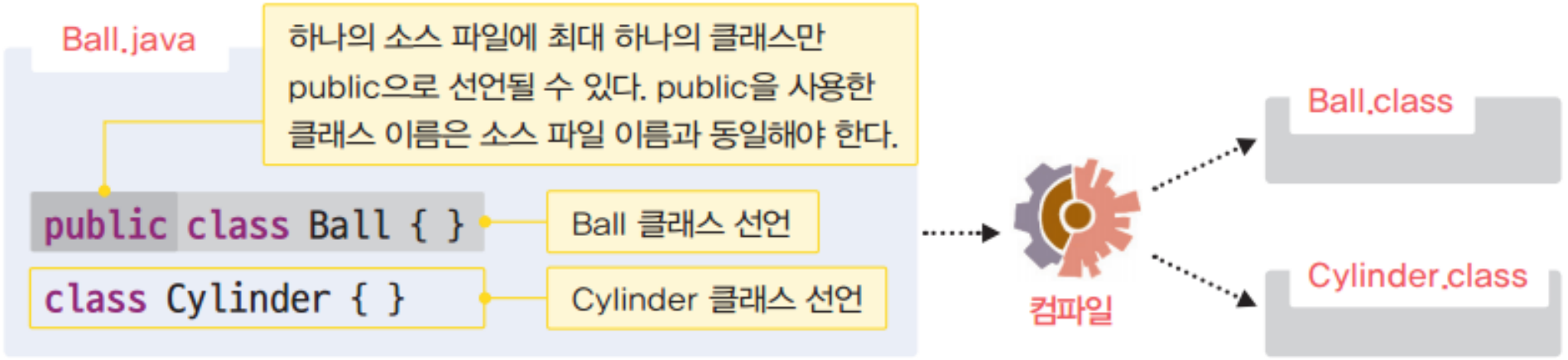


클래스 선언과 객체 생성

클래스 선언과 파일

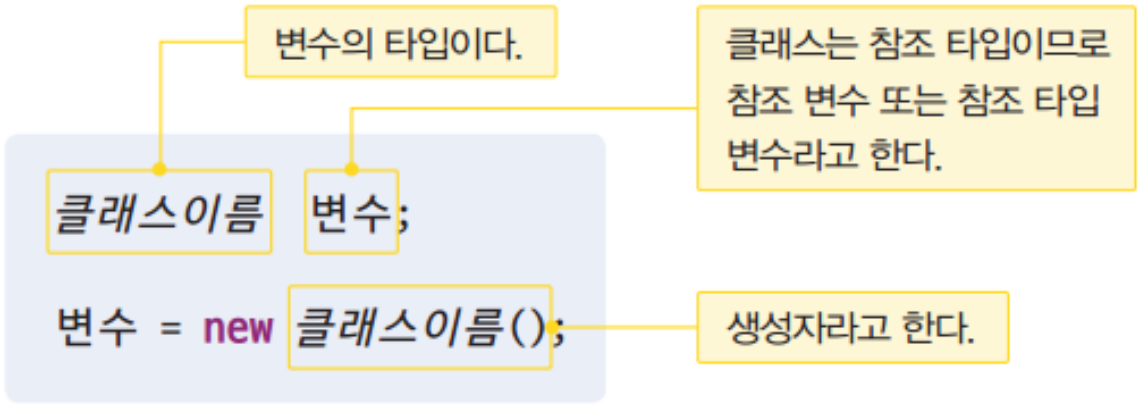
- 보통 소스 파일마다 하나의 클래스를 선언하지만,
  2개 이상의 클래스를 하나의 파일로 선언 가능

- 하나의 파일에 클래스가 둘 이상 있다면 하나만 public으로 선언할 수 있고,
  해당 클래스 이름은 소스 파일 이름과 동일해야 함



클래스 선언과 객체 생성

객체 생성



(a) 객체 변수 선언과 생성

```
new 클래스이름();
```

(b) 변수를 생략한 객체 생성

- **한 문장으로 변수 선언과 객체 생성**

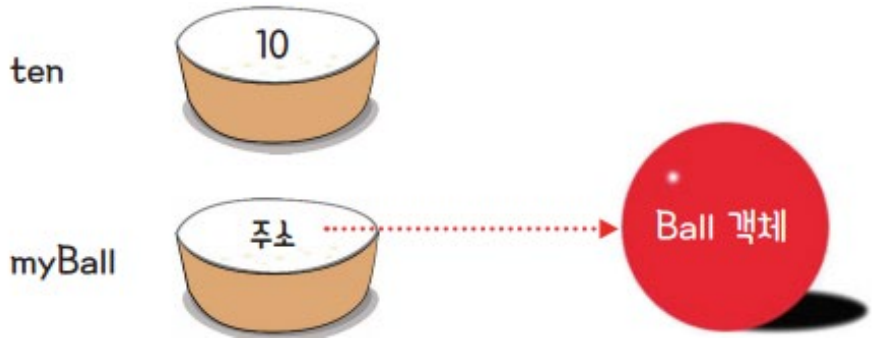
```
클래스이름 변수 = new 클래스이름();
```

클래스 선언과 객체 생성

참조 타입

- 클래스는 대표적인 참조 타입
- 참조 타입의 변수는 기초 타입과 달리 **주소**를 저장

```
int ten = 10;
Ball myBall = new myBall();
```



클래스 선언과 객체 생성

참조 타입

```
Ball myBall = new Ball();
Ball yourBall = new Ball();
myBall = yourBall;
```

(a) myBall = yourBall 연산 전

(b) myBall = yourBall 연산 후

클래스 선언과 객체 생성

참조 타입

```
Ball myBall = new Ball();
Ball yourBall = new Ball();
myBall = yourBall;
```

(a) myBall = yourBall 연산 전

(b) myBall = yourBall 연산 후

클래스 선언과 객체 생성

Practice

- **예제 4-1**: 클래스 선언과 객체 생성 (p.138)
  - package: chap04
  - class: PhoneDemo

클래스의 구성 요소와 멤버 접근

클래스의 구성 요소

- 멤버 : 필드, 메서드
- 생성자

클래스의 구성 요소와 멤버 접근

필드와 지역 변수의 차이

- 필드는 기본 값이 있지만, 지역 변수는 기본 값이 없어 반드시 초기화
- 필드는 클래스 전체에서 사용할 수 있지만, 지역 변수는 선언된 블록 내부에서만 선언된 사용 가능
- 지역 변수에는 접근 제어자 사용 불가

데이터 타입	기본 값
byte	0
char	\u0000
short	0
int	0
배열, 클래스, 인터페이스	null
long	0L
float	0.0F
double	0.0
boolean	false

클래스의 구성 요소와 멤버 접근

Practice

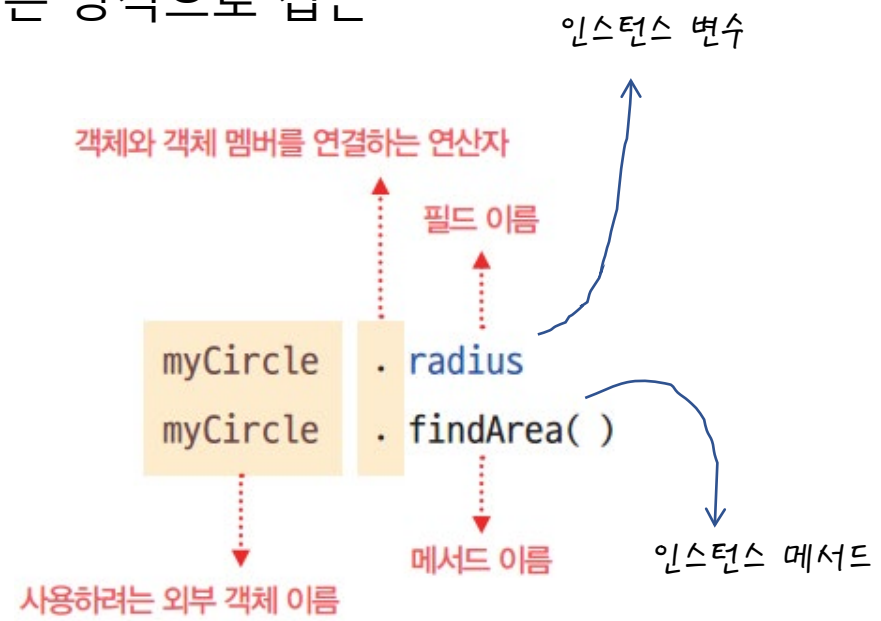
- **예제 4-2**: 지역 변수의 사용 범위 (p.140)
  - Package: chap04
  - Class: LocalVariableDemo

클래스의 구성 요소와 멤버 접근

필드와 메서드 접근

객체참조변수.멤버

- 클래스 내부에서 자신의 멤버에 접근하려면 참조 변수 this 혹은 참조 변수 없이 그냥 멤버 이름 그대로 사용하면 된다.
- 예를 들어, 외부 클래스 Circle의 객체 myCircle이 있다면 myCircle 객체의 radius와 findArea( )는 다음과 같은 방식으로 접근



클래스의 구성 요소와 멤버 접근

Practice

- **예제 4-3**: 클래스의 멤버 접근 (p.142)
  - Package: chap04
  - Class: CircleDemo

접근자와 설정자

접근 제어자 (access modifier)

- 메서드, 필드, 클래스가 공개될 범위를 설정
- private, public 키워드
- 캡슐화를 위해 필요

접근자와 설정자

접근자와 설정자

- private으로 지정된 필드의 값을 반환하는 접근자와
  값을 변경하는 설정자는 공개된 메서드
- 일반적으로 접근자는 get, 설정자는 set으로 시작하는 이름을 사용
- 필드 이름을 외부와 차단해서 독립시키기 때문에
  필드 이름 변경이나 데이터 검증도 가능

접근자와 설정자

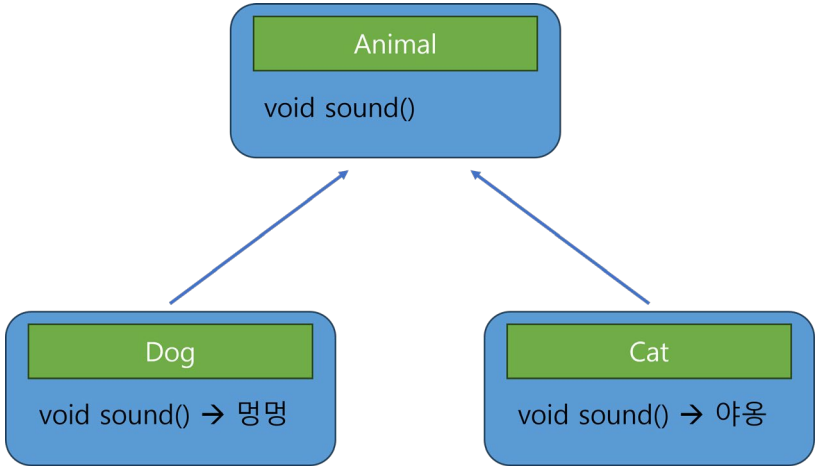
Practice

- **예제 4-4**: 접근자와 설정자 추가 (p.144)
  - Package: chap04
  - Class: CircleDemo

객체지향

Practice

- **예제: 상속 및 다형성**
  - Package: chap04
  - Class: InheritExample




```java
package chap04;

class Animal {
    void sound() {
        System.out.println("Animal default sound");
    }
}

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

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("야옹");
    }
}

public class InheritExample {
    public static void main(String[] args) {
        Animal a1 = new Dog();
        Animal a2 = new Cat();

        a1.sound();
        a2.sound();
        // 다형성: 자식 클래스(Dog, Cat)를 부모 클래스 타입(Animal)으로 다룸
    }
}
```

※ 상속, 다형성은 6주차(Chapter 6) 수업에서 자세히 다룰 예정

객체지향

Practice

- **예제: 캡슐화**
  - Package: chap04
  - Class: EncapsulationExample

- count 필드의 직접적 접근을 방지 (private 접근자)
- increment 메서드를 통해 안전한 방식으로 객체 상태 변경

```java
package chap04;

class Counter {
    private byte count = 0;

    public int getCount() {
        return count;
    }

    public void increment() {
        if (count == Byte.MAX_VALUE) {
            return;
        }
        count++;
    }
}

public class EncapsulationExample {
    public static void main(String[] args) {
        Counter counter = new Counter();
        for (int i = 0; i < 300; i++) {
            counter.increment();
            System.out.println(counter.getCount());
        }
    }
}
```

객체지향

Practice

- **예제: 다형성**
  - Package: chap04
  - Class: PolymorphismExample

- **java.util.*** :
  - 자료구조 등 유용한 클래스들 포함
  - 자바에서 기본적으로 제공 (JDK)

```java
package chap04;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;

public class PolymorphismExample {
    public static void main(String[] args) {
        // 다형성: List 타입으로 선언
        List<String> list1 = new ArrayList<>();
        List<String> list2 = new LinkedList<>();
        Scanner scanner = new Scanner(System.in);

        // 사용자 입력
        for (int i = 0; i < 5; i++) {
            System.out.print("요소를 입력하세요: ");
            String element = scanner.next();
            list1.add(element);
            list2.add(element);
        }

        // 같은 방식으로 출력 가능
        System.out.println("=== ArrayList 출력 ===");
        printList(list1);

        System.out.println("=== LinkedList 출력 ===");
        printList(list2);

        scanner.close();
    }

    // 다형성 활용: List 구현체가 무엇이든 출력 가능
    private static void printList(List<String> list) {
        for (String item : list) {
            System.out.println(item);
        }
    }
}
```