객체 지향

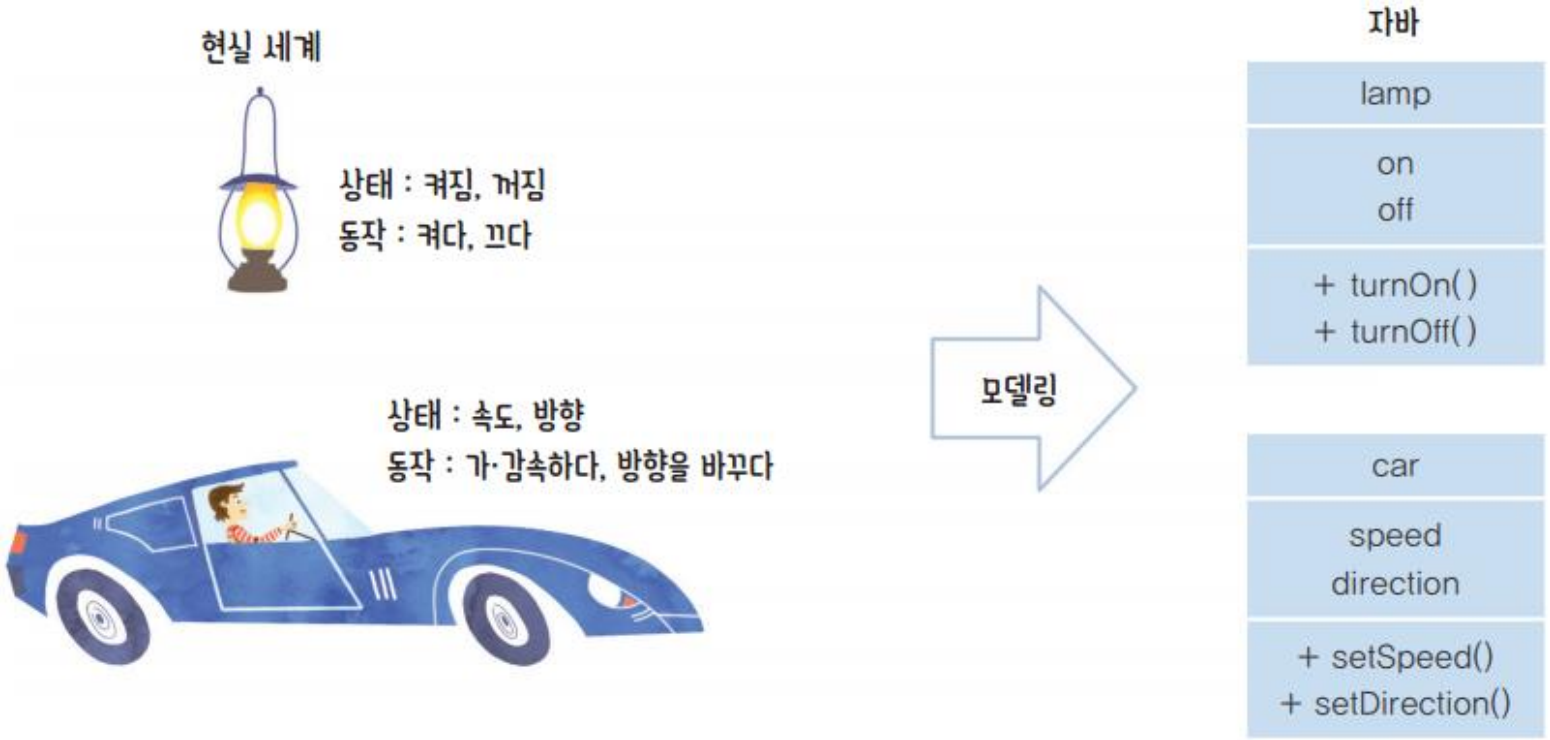
예제 소스 코드는 파일과 연결되어 있습니다.

editplus(유료), notepad++(무료)와 같은 편집 도구를 미리 설치하여 PPT를 슬라이드 쇼로 진행할 때 소스 파일과 연결하여 보면 강의하실 때 편리합니다.

객체지향 기초

객체의 개념

- 소프트웨어 객체는 현실 세계의 객체를 필드와 메서드로 모델링한 것
- 소프트웨어 객체는 상태를 필드(Field)로 정의하고, 동작을 메서드(Method)로 정의.
- 필드는 객체 내부에 선언된 변수를 의미하고, 메서드는 객체 내부에 정의된 동작



객체지향 기초

절차 지향 프로그래밍

- 일련의 동작을 순서에 맞추어 단계적으로 실행하도록 명령어를 나열
- 데이터를 정의하는 방법보다는 명령어의 순서와 흐름에 중점
- 수행할 작업을 예상할 수 있어 직관적인데, 규모가 작을 때는 프로그래밍과 이해하기가 용이
- 소프트웨어는 계산 위주이므로 절차 지향 프로그래밍이 적합

객체 지향 프로그래밍

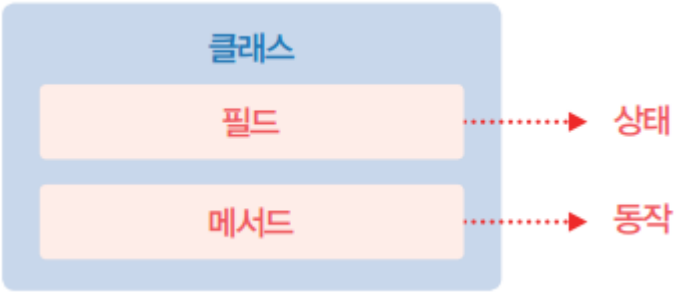
- 소프트웨어의 규모가 커지면서 동작과 분리되어 전 과정에서 서로 복잡하게 얽혀 있는 데이터를 사용했기 때문에 절차 지향 프로그래밍 방식의 한계
- 절차 지향 프로그램은 추후 변경하거나 확장하기도 어려움
- 현실 세계를 객체 단위로 프로그래밍하며, 객체는 필드(데이터)와 메서드(코드)를 하나로 묶어 표현

객체지향 기초

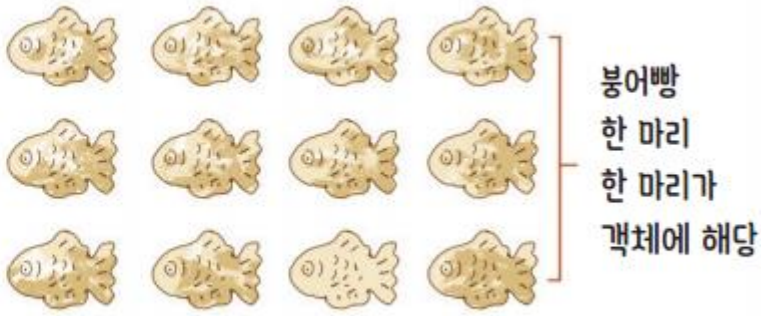
객체와 클래스

형틀 = 클래스

제품 = 객체



인스턴스화



인스턴스 = 붕어빵의 실체

객체 지향 프로그래밍

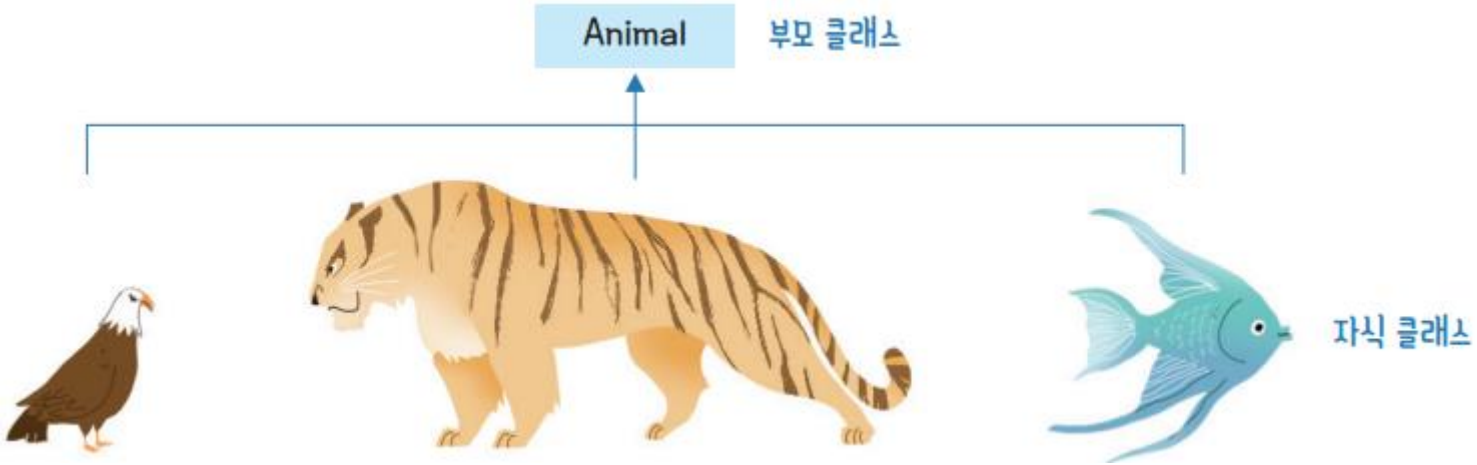
특징

- 캡슐화(정보 은닉) : 관련된 필드와 메서드를 하나의 캡슐처럼 포장해 세부 내용을 외부에서 알 수 없도록 감추는 것

객체 지향 프로그래밍

특징

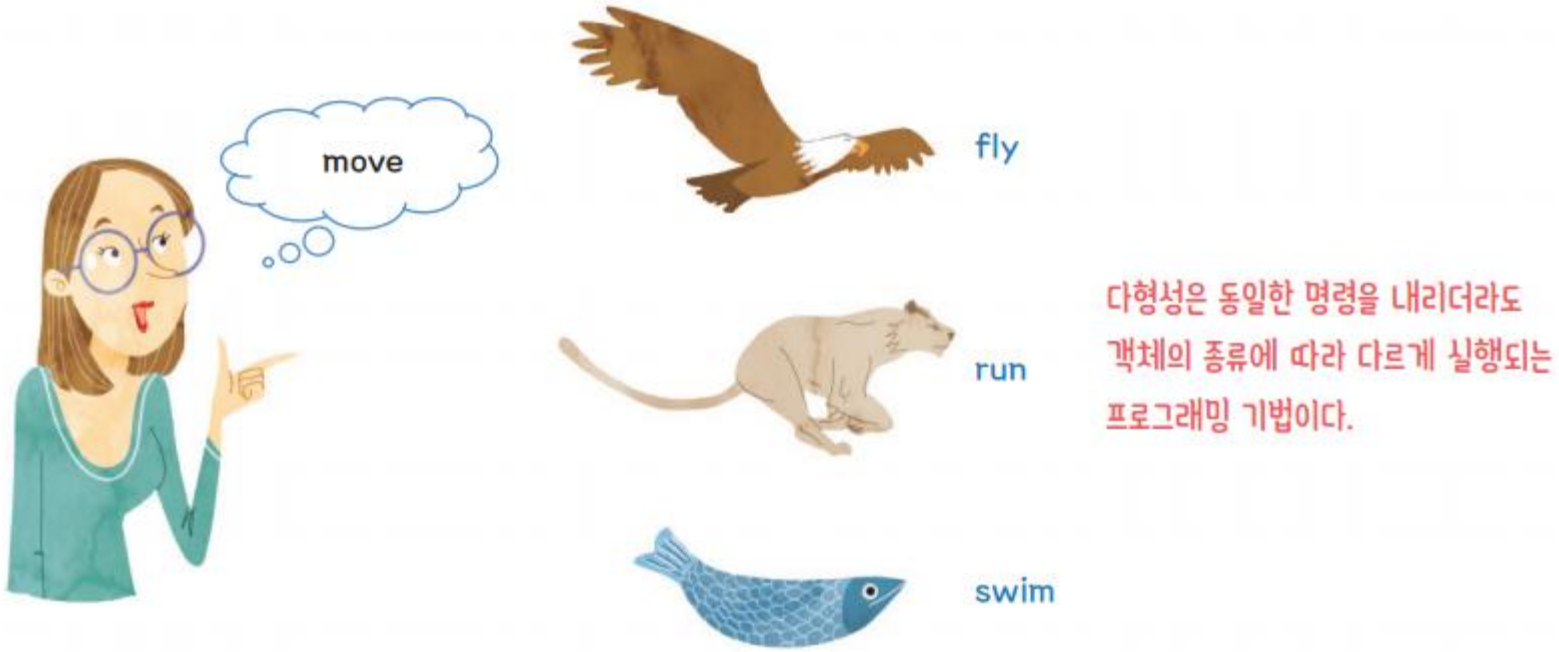
- 상속 : 자녀가 부모 재산을 상속받아 사용하듯이 상위 객체를 상속받은 하위 객체가 상위 객체의 메서드와 필드를 사용하는 것
- 상속은 개발된 객체를 재사용하는 방법 중 하나



객체 지향 프로그래밍

특징

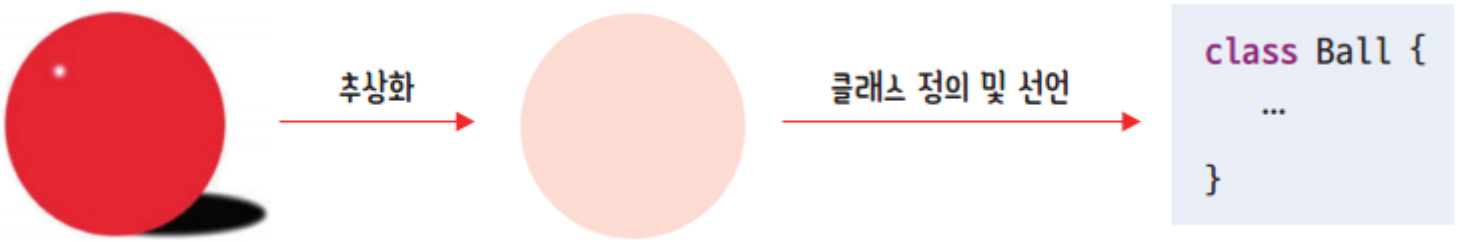
- 다형성 : 대입되는 객체에 따라서 메서드를 다르게 동작하도록 구현하는 기술. 실행 도중 동일한 이름의 다양한 구현체 중에서 메서드를 선택 가능



클래스 선언과 객체 생성

추상화

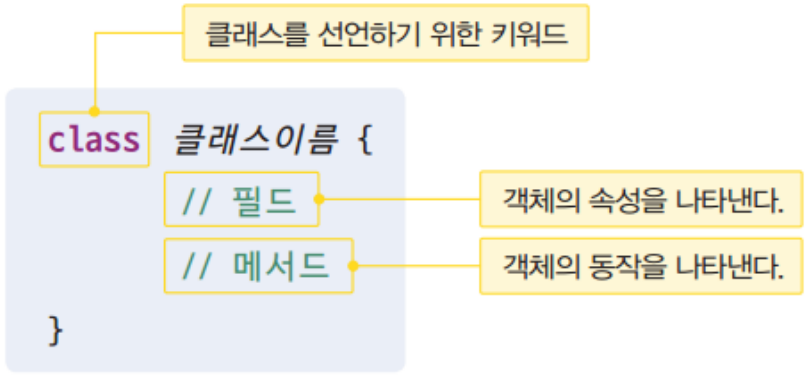
- 현실 세계의 객체는 수많은 상태가 있고 다양한 동작을 하지만, 클래스에 모두 포함하기는 어렵기에 추상화(Abstraction)하는 과정이 필요
- 추상화는 현실 세계의 객체에서 불필요한 속성을 제거하고 중요한 정보만 클래스로 표현하는 일종의 모델링 기법
- 따라서 사람마다 추상화하는 기법이 같지 않으므로 각 개발자는 클래스를 다르게 정의 가능



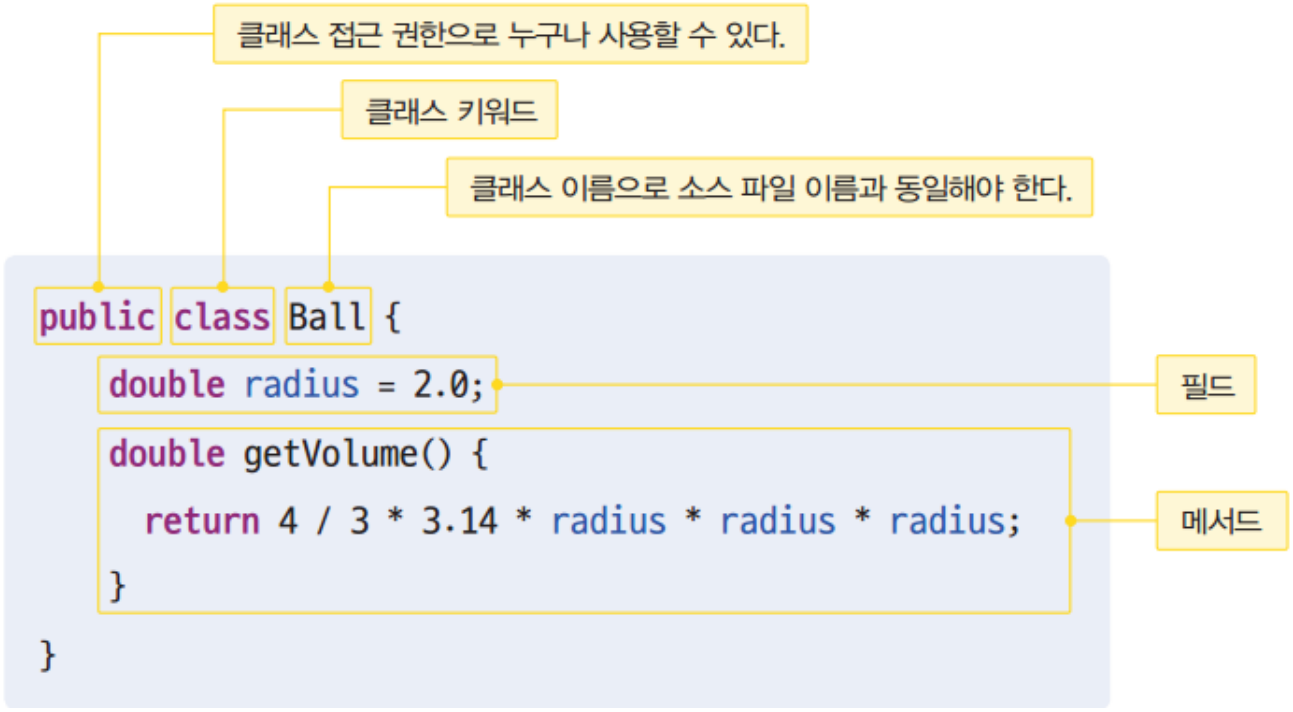
클래스 선언과 객체 생성

클래스 선언

- 형식



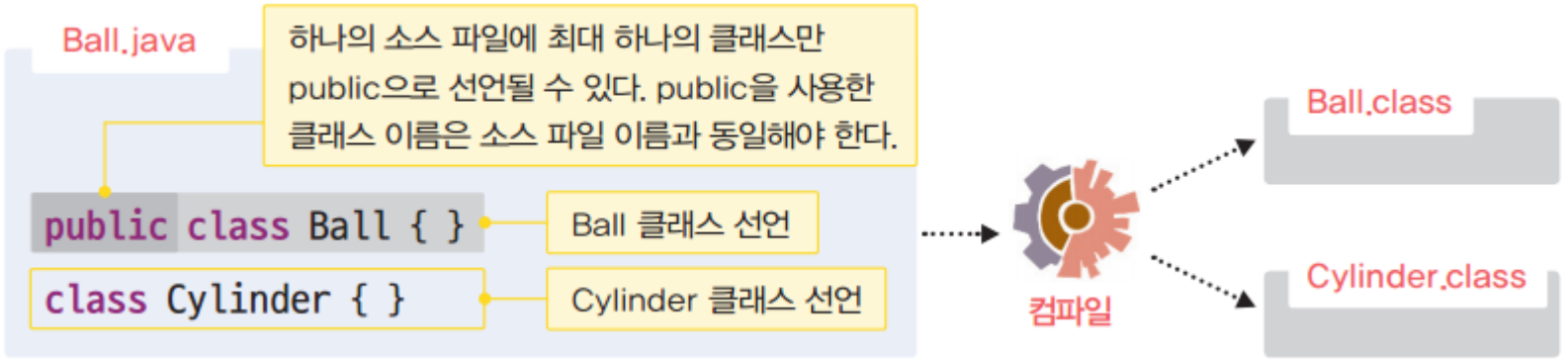
- 예



클래스 선언과 객체 생성

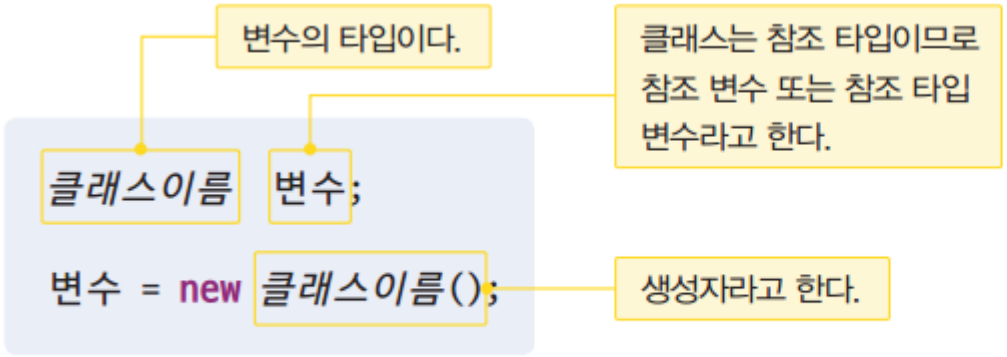
클래스 선언과 파일

- 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 생성과 참조 변수



(a) 객체 변수 선언과 생성

```
new 클래스이름();
```

(b) 변수를 생략한 객체 생성

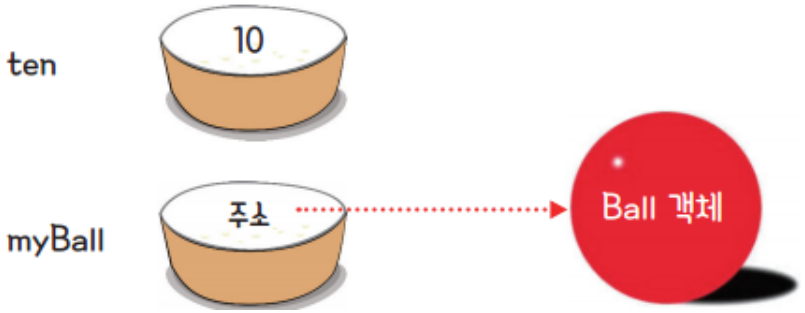
- 한 문장으로 변수 선언과 객체 생성

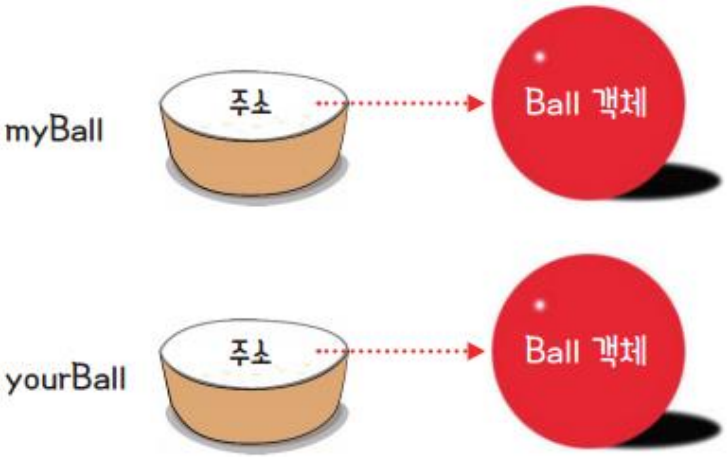
```
클래스이름 변수 = new 클래스이름();
```

클래스 선언과 객체 생성

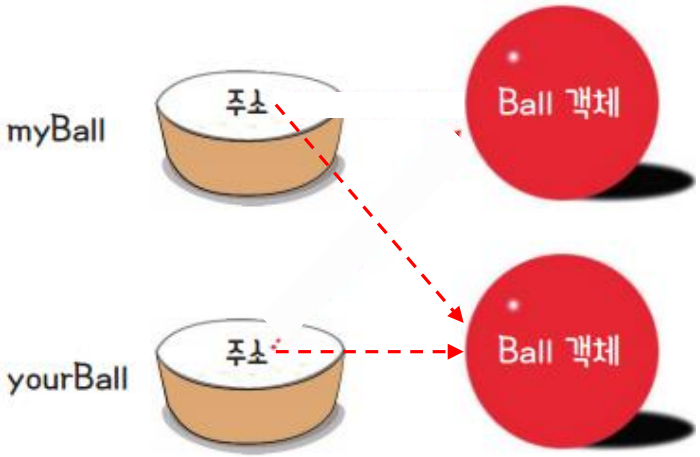
- 기초 타입과 참조 타입

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





(a) myBall = yourBall 연산 전



(b) myBall = yourBall 연산 후

클래스 선언과 객체 생성

- **기초 타입과 참조 타입**
  - 예제 : sec03/PhoneDemo

```
100만 원짜리 갤럭시 S8 스마트폰
85만 원짜리 G6 스마트폰
```

클래스의 구성 요소와 멤버 접근

클래스의 구성 요소

- 멤버 : 필드, 메서드
- 생성자

- 참고 : 지역 변수는 메서드 내부에 선언된 변수. 매개 변수도 일종의 지역 변수임

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 내부의 선언된 후에서만 사용 가능
- 필드와 달리 지역 변수는 final로만 지정 가능

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

필드와 지역 변수의 차이

- 예제 : sec04/LocalVariableDemo

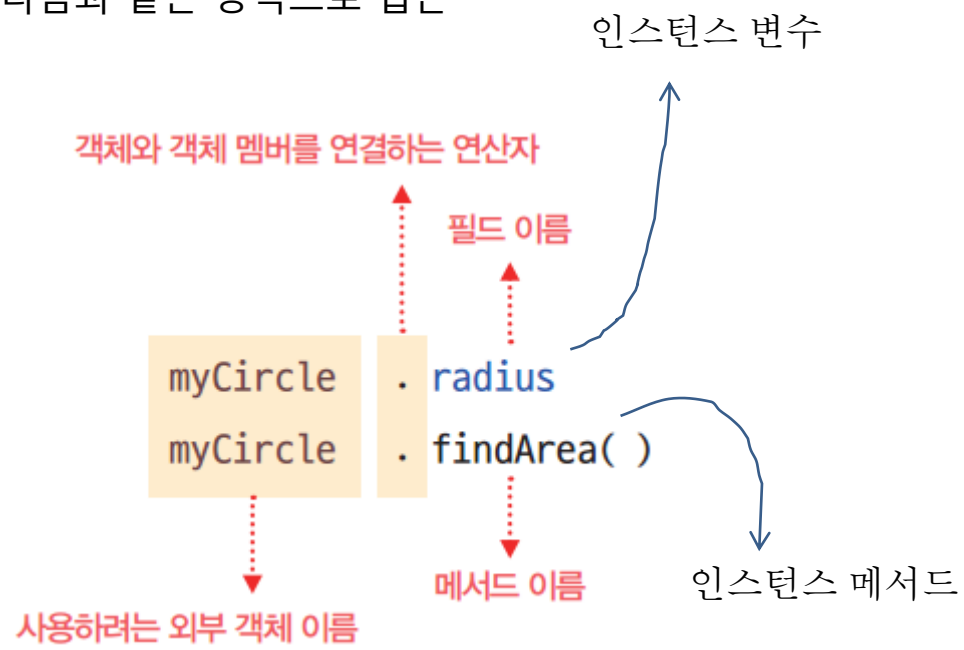
```
0123456789
```

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

필드와 메서드 접근

- 예를 들어, 클래스가 radius 필드와 findArea( ) 메서드를 포함한다면 클래스 내부에서는 다음과 같이 그대로 사용하면 된다.

```
radius 혹은 this. radius            // 필드 이름
findArea() 혹은 this.findArea       // 메서드 이름
```

- 예제 : sec04/CircleDemo

접근자와 설정자

필요성

- 클래스 내부에 캡슐화된 멤버를 외부에서 사용할 필요

접근자와 설정자

- private으로 지정된 필드에 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

- 예제 : sec05/CircleDemo

생성자

생성자의 의미와 선언

- 생성자의 역할 : 객체를 생성하는 시점에서 필드를 다양하게 초기화
- 생성자의 선언 방식

```
클래스이름 ( … ) { … }
```

일반적으로 공개되어야 하므로 public으로 선언되지만 아닐 수도 있다.

- 생성자 사용

```
클래스이름 변수 = new 클래스이름(…);
```

생성자

- 생성자 이름은 클래스 이름과 같다.
- 생성자의 반환 타입은 없다.
- 생성자는 new 연산자와 함께 사용하며, 객체를 생성할 때 호출한다.
- 생성자도 오버로딩할 수 있다.

생성자

디폴트 생성자

- 모든 클래스는 최소한 하나의 생성자가 있음
- 만약 생성자를 선언하지 않으면 컴파일러가 자동으로 디폴트 생성자를 추가

```
class Circle {
   private double radius;
   public double getRadius() { … }
   public void setRadius(double radius) { … }
   …
}
```

생성자를 하나도 선언하지 않았지만 컴파일러가 디폴트 생성자인 Circle( )를 자동으로 추가한다.

```
public class CircleDemo {
   public static void main(String[] args) {
      Circle myCircle = new Circle();
      …
   }
}
```

Circle 클래스에서 생성자를 선언하지 않았지만, Circle 생성자를 사용해 객체를 생성할 수 있다.

- 예제 : sec06/etc/CircleDemo

생성자

생성자 오버로딩

- 생성자도 메서드처럼 오버로딩(Overloading) 가능

- 예제 : sec06/CircleDemo

생성자

- this와 this()

```
class Square {
    private double side;

    public void setRadius(double s) {
        side = s;
    }
}
```

멤버 필드이다.

멤버 필드처럼 정사각형 변을 의미하지만, 변수 이름은 다르다.

```
class Square {
    private double side;

    public void setRadius(double side) {
        this.side = side;
    }
}
```

매개변수

멤버 필드

생성자

- **this와 this()**
  - 예제 : sec06/dis/Circle
  - 주의 사항

```
public Circle() {
   radius = 10.0;
   this("빨강");
}
```

기존 생성자를 호출하기 전에 다른 실행문이 있어 오류가 발생한다. 따라서 순서를 바꿔야 한다.

생성자

연속 호출

- 예를 들어 반환 타입이 void인 setName(String name), setAge(), sayHello()라는 메서드를 가진 Person 클래스가 있다고 가정

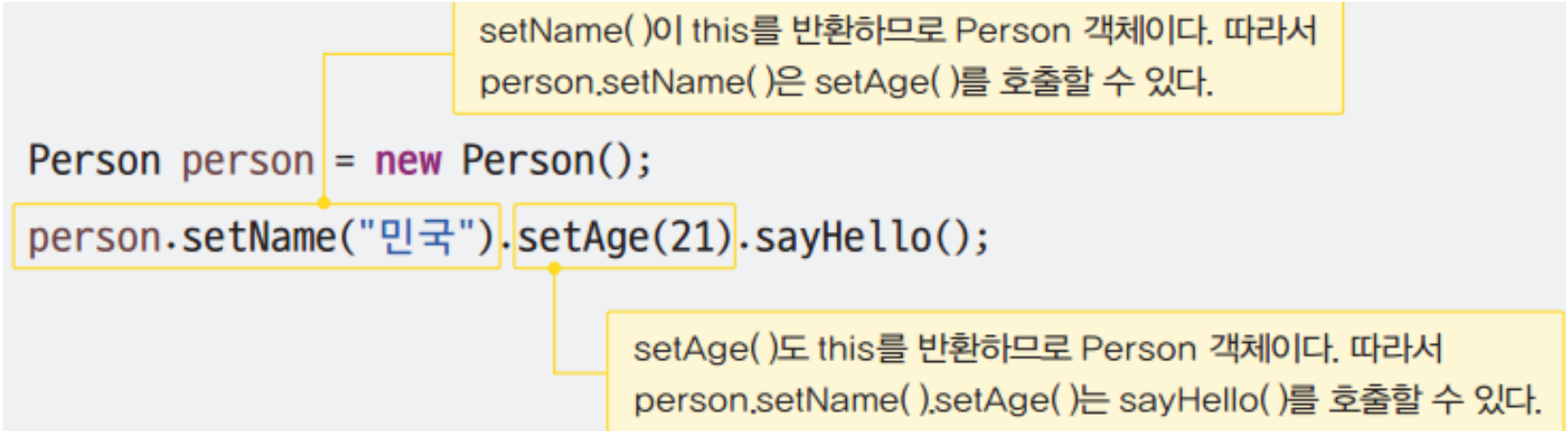
```
Person person = new Person();
person.setName("민국");
person.setAge(21);
person.sayHello();
```

메서드를 호출할 때마다
새로운 실행문을 사용해야 하므로
번거롭고 가독성도 떨어진다.

생성자

연속 호출

- 만약 setName()과 setAge()의 반환 타입이 this라면

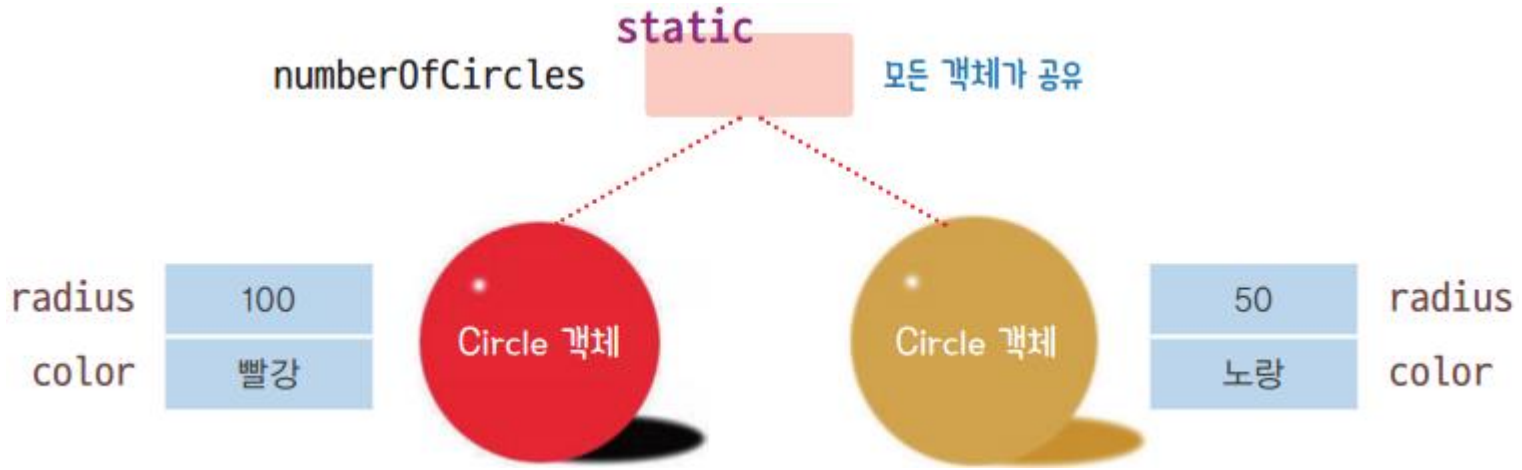


- 예제 : [sec06/MethodChainDemo]

```
안녕, 나는 민국이고 21살이야.
```

정적 멤버

인스턴스 멤버와 정적 멤버



- 자바는 static 키워드로 클래스의 필드를 공유할 수 있도록 지원
- 인스턴스 변수 : static 키워드로 지정되지 않아 공유되지 않은 필드로 인스턴스마다 자신의 필드를 생성
- 정적 변수 혹은 클래스 변수 : static 키워드로 지정하여 모든 인스턴스가 공유하는 필드

정적 멤버

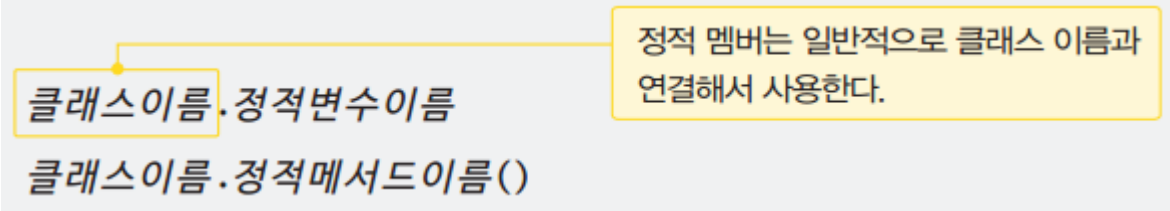
인스턴스 멤버와 정적 멤버

- 인스턴스 변수는 객체별로 관리. 객체를 생성할 때 인스턴스 변수도 객체가 소멸될 때는 자동으로 소멸
- 반면 정적 변수는 클래스 로더가 클래스를 메서드 영역에 적재할 때 생성

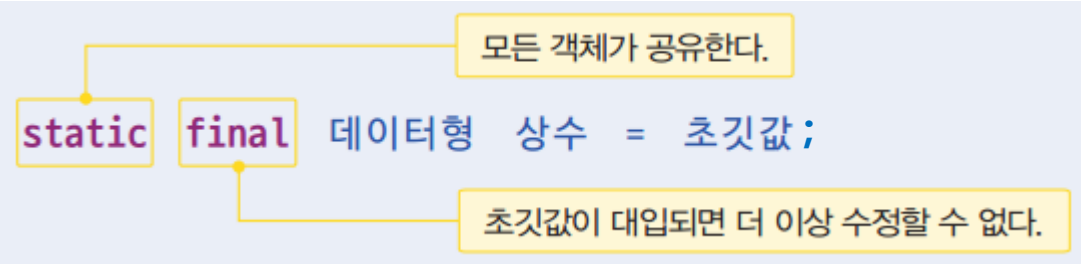
- 정적 메서드의 유의 사항
  - 객체와 관련된 인스턴스 변수를 사용할 수 없다.
  - 객체와 관련된 인스턴스 메서드를 호출할 수 없다.
  - 객체 자신을 가리키는 this 키워드를 사용할 수 없다.

정적 멤버

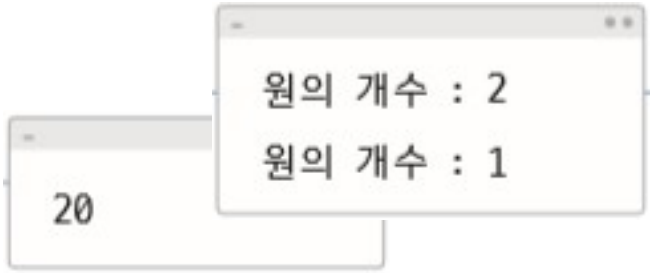
정적 멤버의 활용



- 상수는 변경되지 않는 변수이기 때문에 final 키워드로 지정하지만 final로만 지정하면 객체마다 자신의 기억 공간
- 상수는 값이 변경되지 않으므로 객체마다 따로 기억 공간을 할당할 필요가 없다. 따라서 static final로 지정해서 선언



- 예제 : sec07/CircleDemo
- 예제 : sec07/UtilDemo



정적 멤버

정적 블록

- 정적 변수의 초기화 과정이 for 문이나 오류 처리처럼 복잡하다면 과정이 그리 간단하지 않을 것이다. 대신에 정적 변수의 초기화가 복잡할 때는 다음과 같이 정적 블록을 사용할 수 있다.

- 예제 : sec07/OneToTenDemo