

# 파워자바(개정3판)

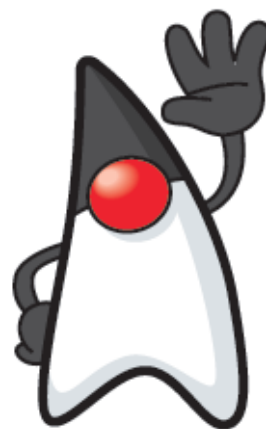


4장 클래스와 객체 |



## 4장의 목표

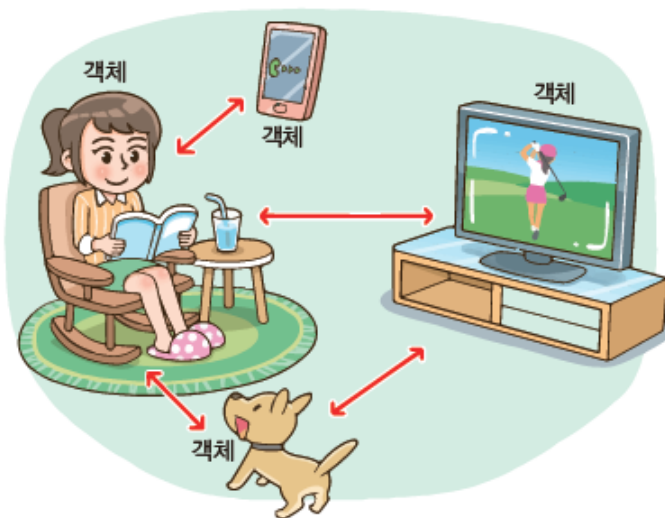
1. 객체 지향과 절차 지향을 비교해서 설명할 수 있나요?
2. 특정한 객체를 찍어내는 클래스를 정의할 수 있나요?
3. 메소드 오버로딩을 사용하여 이름이 동일한 여러 개의 메소드를 정의할 수 있나요?
4. 생성자를 작성하여 객체를 초기화할 수 있나요?
5. 접근자와 생성자를 만들어서 접근을 제어할 수 있나요?





# 객체지향 프로그래밍

- 객체(object)를 사용하는 프로그래밍 방식을 객체 지향 프로그래밍(OOP: Object-Oriented Programming)이라고 한다.
- OOP는 실세계와 비슷하게 소프트웨어도 작성해보자는 방법론이다.



실세계는 객체들로 가득  
차 있습니다.





# 객체지향 프로그래밍

- 객체들은 객체 나름대로의 고유한 기능을 수행하면서 다른 객체들과 상호 작용한다.
- 객체들은 메시지(message)를 통하여 상호 작용하고 있다.





# 객체

- 객체는 상태와 동작을 가지고 있다.
- 객체의 상태(**state**)는 객체의 속성이다.
- 객체의 동작(**behavior**)은 객체가 취할 수 있는 동작이다.

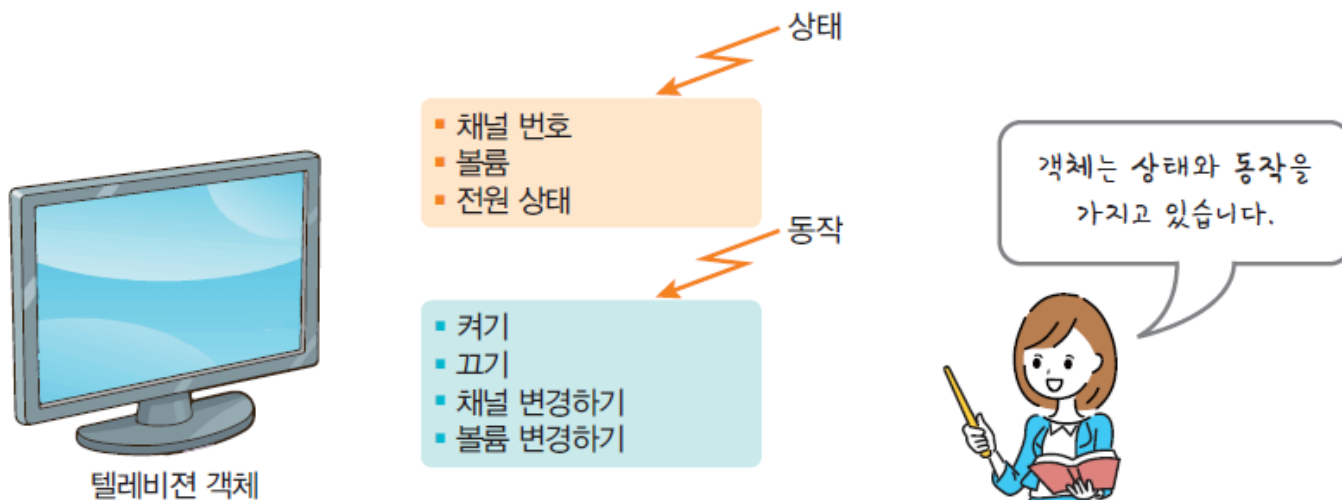
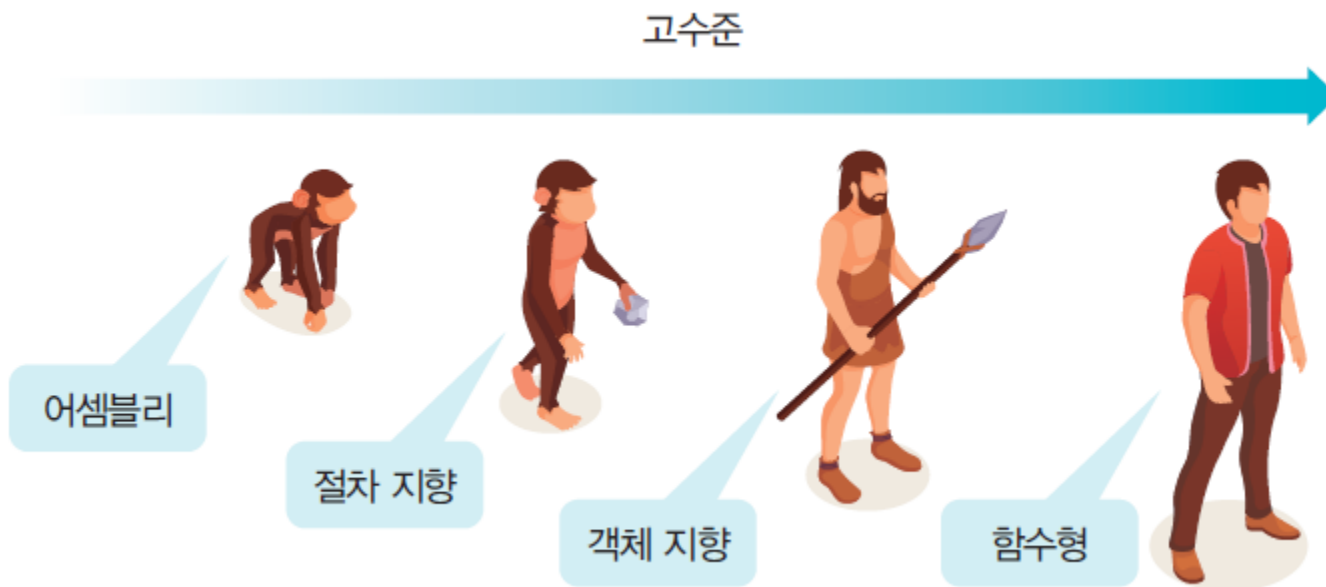


그림 4.1 텔레비전 객체의 예



# 프로그래밍 기법의 발전





# 절차지향 프로그래밍

- 절차 지향 프로그래밍은 프로시저(procedure)에 기반을 두고 있다.
- “절차”라는 용어는 “procedure”를 번역한 것이며, 프로시저는 함수 또는 서브루틴을 뜻한다.
- 절차 지향 프로그래밍에서 프로그램은 함수(프로시저)들의 집합으로 이루어진다.

```
int main(void) {  
    double radius = 10.0;  
    double area;  
    area = 3.14*radius*radius;  
    printf("면적=%f\n", area);  
  
    return 0;  
}
```



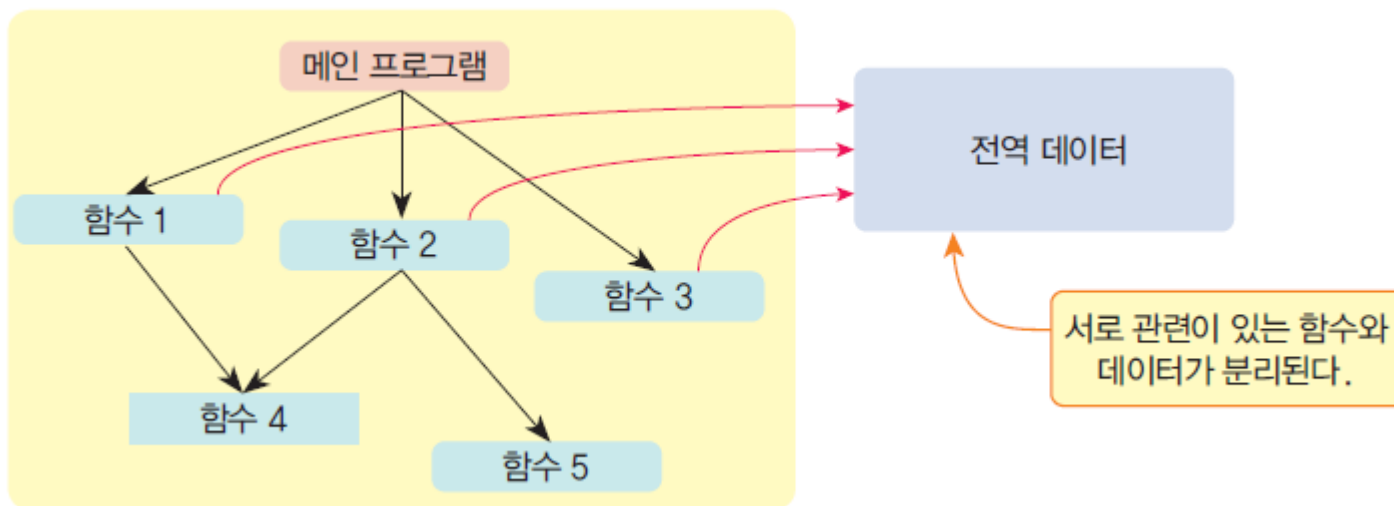
```
double getCircleArea(double r) {  
    double area;  
    area = 3.14*r*r;  
    return area;  
}  
  
int main(void) {  
    int radius = 10.0;  
    double area = getCircleArea(radius);  
    printf("면적=%f\n", area);  
    return 0;  
}
```

함수를 사용하여 소스를 작성한다.



# 절차지향의 문제점

- 서로 관련있는 함수와 데이터가 분리된다.
- (예) 앞장의 코드에서 원의 반지름을 나타내는 `radius`와 이 데이터를 사용하는 함수 `getCircleArea()`는 서로 분리되어 있다.

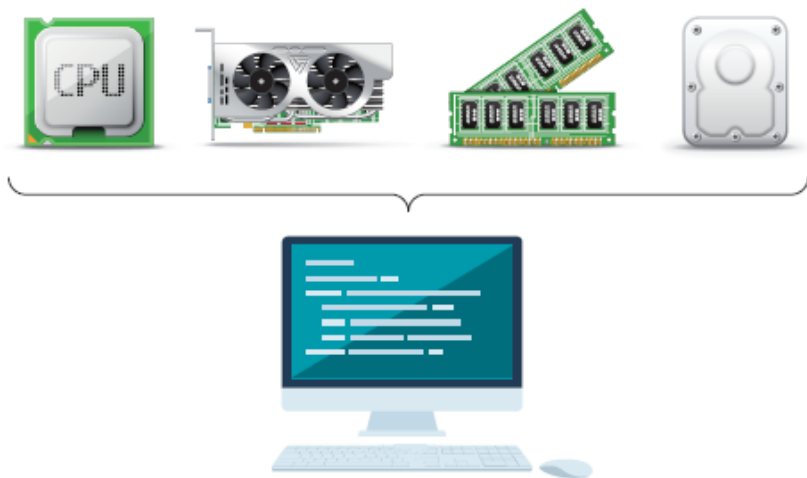






# 객체 지향이 나오게 된 동기

- 컴퓨터 하드웨어는 원하는 부품을 조립하여 만들 수 있다.
- 컴퓨터 소프트웨어는 원하는 함수를 조립하여 만들 수 없다. 코드와 데이터를 묶는 방법이 없었기 때문이다.



부품을 조립하여 PC를 만들듯이  
객체를 조합하여 소프트웨어를 만든다

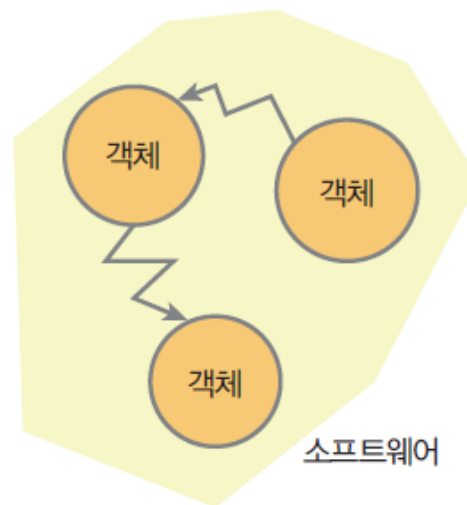
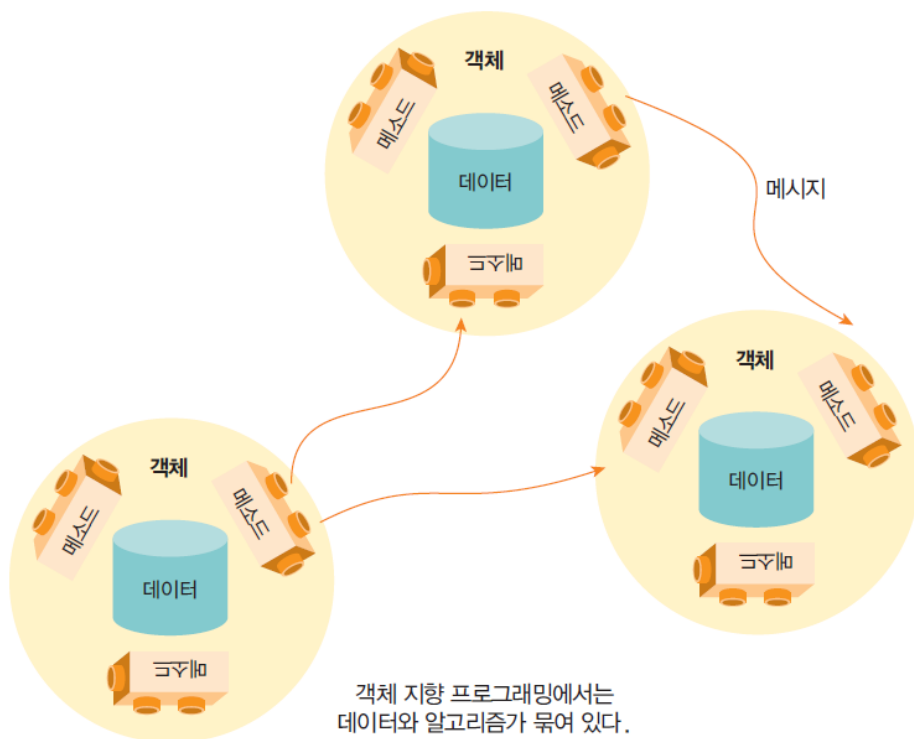


그림 4.2 객체 지향 방법



# 객체 지향 프로그래밍

- 객체 지향 프로그래밍(object-oriented programming)은 데이터와 함수를 하나의 덩어리로 묶어서 생각하는 방법이다. 데이터와 함수를 하나의 덩어리(객체)로 묶는 것을 캡슐화(encapsulation)라고 부른다.





# 개체 지향 프로그래밍의 간단한 예

- 앞의 코드에서 원의 반지름과 면적을 계산하는 함수를 하나로 묶으면 다음과 같다.

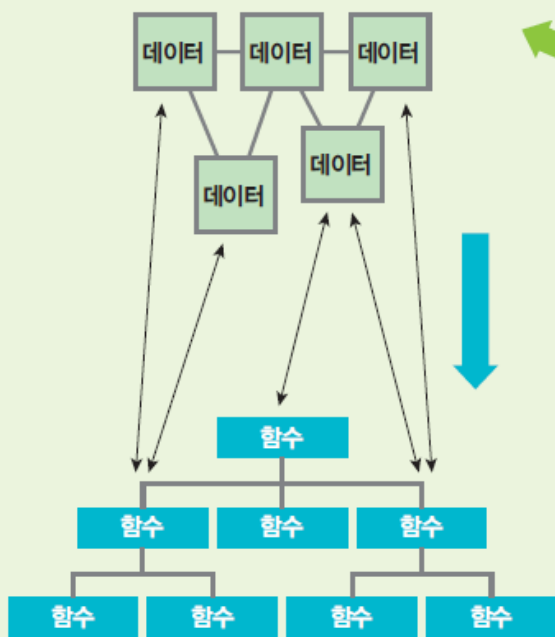
```
public class Circle {  
    double radius;           // 데이터  
    String color;           // 데이터  
    double getArea() { return 3.14*radius*radius; } // 함수  
}
```



# 절차 지향 vs 객체 지향

## 절차 지향 프로그램

데이터와 함수가 분리된다.

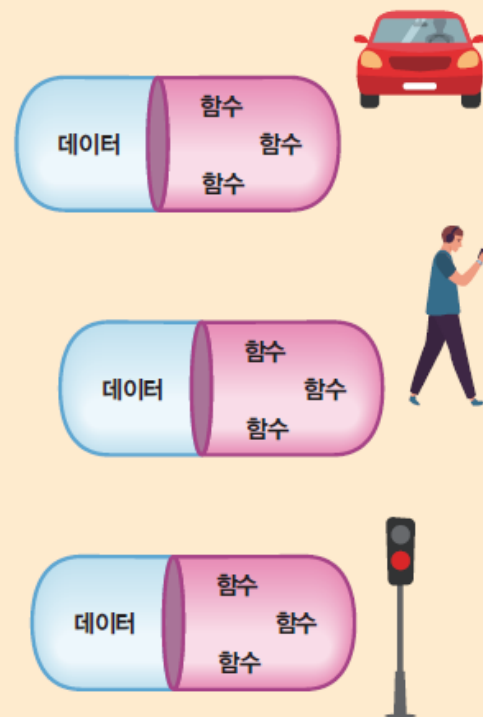


문제 영역



## 객체 지향 프로그램

데이터와 함수가 객체로 캡슐화된다.





# 객체지향의 특징

- 객체지향 언어는 캡슐화, 상속 및 다형성으로 정의된다

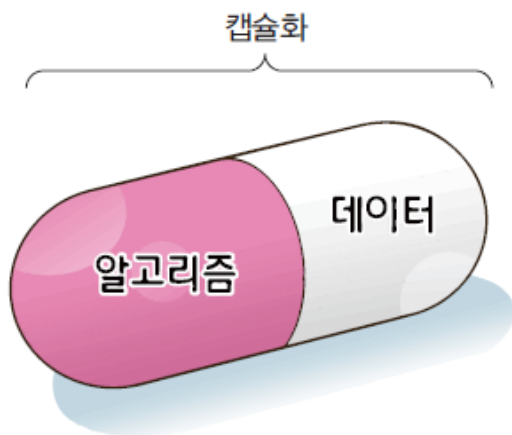


OOP의 4개의 기둥



# 캡슐화

- 관련된 데이터와 알고리즘이 하나의 묶음으로 정리되어 있어야 한다. 객체 지향 프로그래밍에서는 이것을 캡슐화(encapsulation)라고 부른다.





# 캡슐화의 목적

- 캡슐화에는 2가지의 목적이 있다. 캡슐화의 첫 번째 목적은 서로 관련된 데이터와 알고리즘을 하나로 묶는 것이다.
- 캡슐화의 두 번째 목적은 객체를 캡슐로 싸서 객체의 내부를 보호하는 것이다. 즉 객체의 실제 구현 내용을 외부에 감추는 것이다



캡슐로 싸서 내부의 코드를  
보호합니다.





# 정보 은닉

- 정보 은닉(**information hiding**)은 객체의 외부에서는 객체의 내부 데이터를 볼 수 없게 한다는 의미이다.
- 객체 안의 데이터와 알고리즘은 외부에서 변경하지 못하게 막고, 공개된 인터페이스를 통해서만 객체에 접근하도록 하는 개념이다.

사용자가 책을 마음대로  
꺼내고 꽂을 수 있기 때문에  
서가 엉망이 될 수 있다.



(a) 절차 지향 방법

사서를 통하여  
책을 꺼내면  
서가는 안전하다.



(b) 객체 지향 방법

그림 4.4 어떤 방법이 내부 데이터를 안전하게 보관할까?





# 소프트웨어 캡슐

소프트웨어 캡슐 내부가 어떻게 구성되어 있으며 어떻게 돌아가는지 전혀 신경 쓸 필요가 없다. 그냥 사용법만 익혀서 사용하면 된다

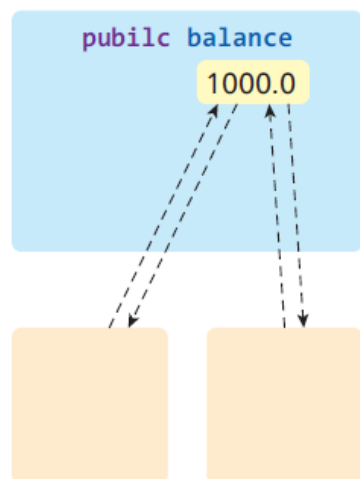




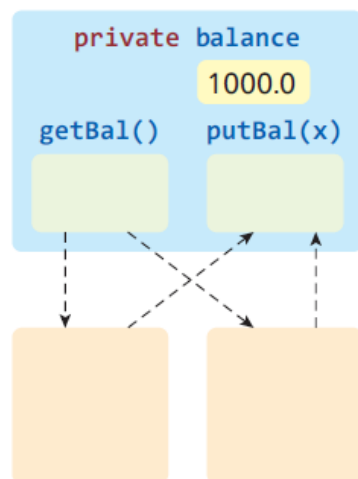
# 공개 인터페이스

- 세탁기 위에있는 버튼이 이것이 바로 클래스에 정의된 공개 인터페이스에 해당된다.

직접적인 접근

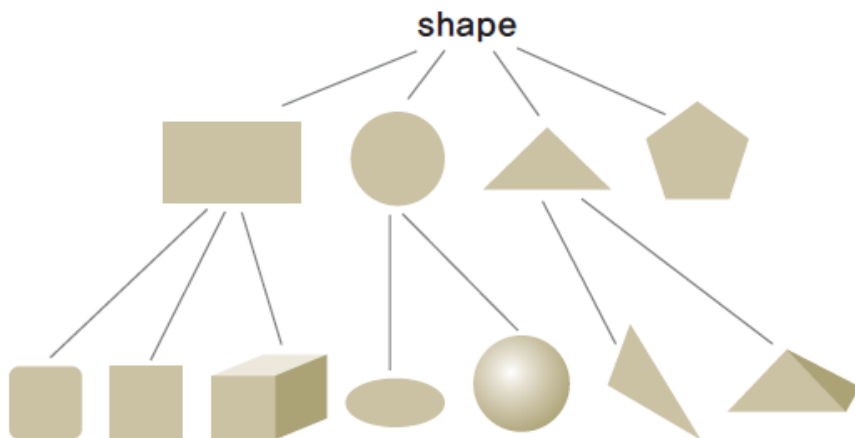


간접적인 접근





- 자동차 회사에서 새로운 모델을 개발할 때, 기존의 모델을 수정하는 방법도 많이 선택한다(페이스리프트).
- 우리는 이미 작성된 클래스(부모 클래스)를 이어받아서 새로운 클래스(자식 클래스)를 생성할 수 있다. 자식 클래스는 부모 클래스의 속성과 동작을 물려받는다.



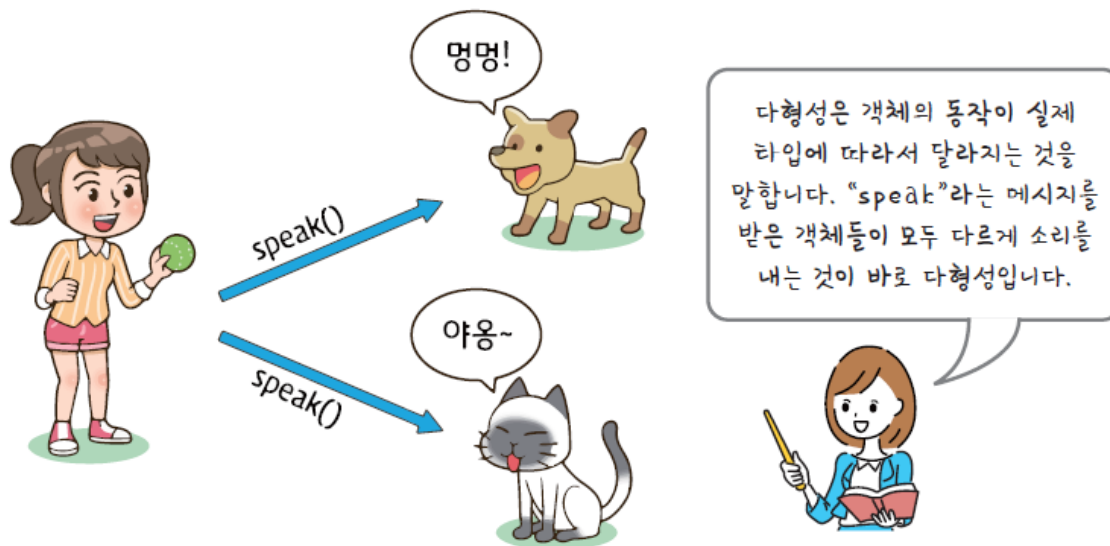
상속은 기존에 만들어진 코드를 이어받아서 보다 쉽게 코드를 작성하는 기법입니다.





# 다형성

- 다형성이란, 동일한 이름의 동작이라고 하여도 객체의 실제 타입에 따라서 동작의 내용이 달라질수 있다는 것을 의미한다.





# 추상화

- 추상화(**abstraction**)는 불필요한 정보는 숨기고 중요한 정보만을 표현함으로써 프로그램을 간단히 만드는 기법이다. 추상화는 요즘 유행하는 “미니멀리스트”를 생각하면 이해하기 쉽다.



실제 객체



추상화된 객체

추상화는 필요한 것만을 남겨 놓는  
것입니다. 추상화 과정이 없다면  
사소한 것도 신경 써야 합니다.





# 주간점검

1. 객체 지향의 개념에 속하는 것들을 나열해보자.
2. 객체 지향의 최종 목적은 무엇일까? 즉 무엇 때문에 객체 지향적인 방법으로 설계하는가?
3. 캡슐화의 장점은 무엇인가?
4. 정보 은닉이란 무엇인가?
5. 절차 지향 방법과 객체 지향 방법을 비교해서 설명해보자.



# 클래스란?

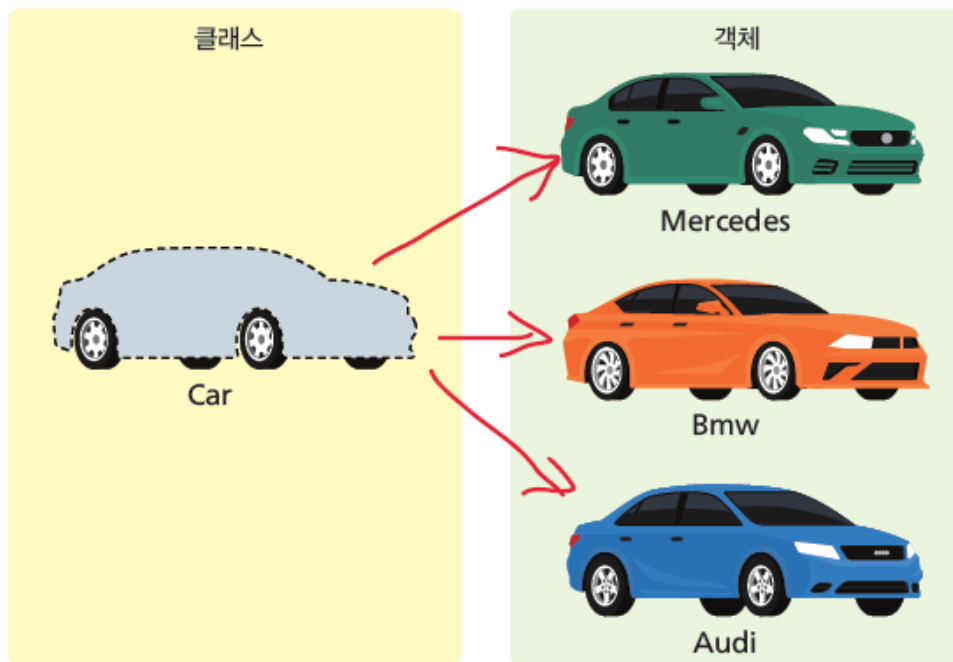
- 객체에 대한 설계도를 클래스(class)라고 한다. 클래스란 특정한 종류의 객체들을 찍어내는 형틀(template) 또는 청사진(blueprint)이라고도 할 수 있다. 클래스로부터 만들어지는 각각의 객체를 그 클래스의 인스턴스(instance)라고 한다.





# 클래스란?

- 하나의 클래스로 여러개의 인스턴스를 찍어내지만, 인스턴스마다 속성의 값은 다르다.







# 클래스의 작성

## Syntax: 클래스 정의

```
public class 클래스이름 {
```

```
    자료형    필드1;
```

```
    자료형    필드2;
```

```
    ...
```

```
    반환형    메소드1() { ... }
```

```
    반환형    메소드2() { ... }
```

```
    ...
```

```
}
```

클래스는 class라는 키워드로 시작한다.  
이어서 클래스 이름을 적고, 중괄호 안에  
필드와 메소드를 나열하면 된다.

필드

메소드



# 클래스의 작성

```
Circle.java
01 public class Circle {
02
03     public int radius;
04     public String color;
05
06     public double getArea() {
07         return 3.14*radius*radius;
08     }
09 }
```

키워드


클래스 이름

접근 지정자

필드

메소드

클래스





# 객체 생성

- 클래스는 객체를 만들기 위한 설계도(틀)에 해당된다. 설계도를 가지고 어떤 작업을 할 수는 없다.
- 실제로 어떤 작업을 하려면 객체를 생성하여야 한다

*CircleTest.java*

```
01 public class CircleTest {  
02     public static void main(String[] args) {  
03         Circle obj; ← ① 참조 변수 선언  
04         obj = new Circle(); ← ② 객체 생성  
05         obj.radius = 100;  
06         obj.color = "blue"; ← ③ 객체의 필드 접근  
07         double area = obj.getArea(); ← ④ 객체 메소드 접근  
08         System.out.println("원의 면적=" + area);  
09     }  
10 }
```

원의 면적=31400.0



# 객체 생성 단계

## 1. 참조 변수를 선언

```
Circle obj;
```

obj



## 2. new 연산자를 사용하여 객체를 생성하고 객체의 참조값을 참조 변수에 저장

```
obj = new Circle();
```

obj



|                      |  |
|----------------------|--|
| radius               |  |
| color                |  |
| getArea()<br>{ ... } |  |

## 3. 객체의 필드와 메소드를 사용

```
obj.radius = 100;
```

```
obj.color = "blue";
```

```
double area = obj.getArea();
```

obj

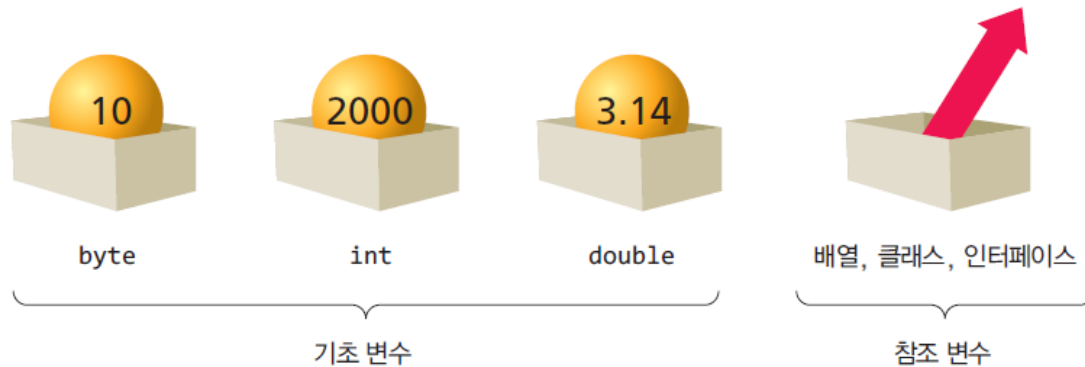


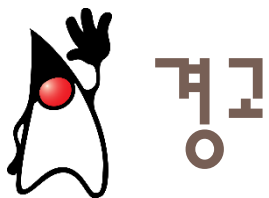
|                      |        |
|----------------------|--------|
| radius               | 100    |
| color                | "blue" |
| getArea()<br>{ ... } |        |



# 참조 변수

- 참조 변수는 객체를 참조할 때 사용되는 변수로서 여기에는 객체의 참조값이 저장된다. 참조값은 일반적으로 객체의 주소이다.





프로그래밍시 혼동을 많이 일으키는 부분은, 다음과 같이 선언하면 객체가 생성된다고 생각하는 것이다.

```
Circle obj;
```

위의 문장은 단순히 객체를 참조하는 변수 `obj`를 선언해놓은 것이다. 객체의 이름만 정해놓은 것이라고 생각해도 좋다. 아직 객체는 생성되지 않았다. 참조 변수만 생성된 것이다. 자바에서 모든 객체는 `new` 연산자를 이용해야만 비로소 생성된다. 하지만 C++ 언어에서는 위의 문장으로 실제 객체가 생성된다. 자바랑 혼동하지 말자.



#### public 클래스와 소스 파일 이름

원칙적으로 하나의 소스 파일에는 하나의 `public` 클래스만 있어야 하고 `public` 클래스의 이름은 소스 파일 이름과 동일하여야 한다. 하지만 `public` 클래스가 아니라면 다른 클래스도 동일한 소스 파일에 추가할 수 있다. `CircleTest` 클래스에 `public`을 뺀 `Circle` 클래스를 추가해도 된다.





# 예제:

## 예제 4-1

## DeskLamp 클래스 작성하고 객체 생성해보기



집에서 사용하는 데스크 램프를 클래스로 작성하여 보면 다음과 같다.

| DeskLamp     |
|--------------|
| -isOn : bool |
| +turnOn()    |
| +turnOff()   |



### DeskLamp.java

```
01 public class DeskLamp {
02     // 인스턴스 변수 정의
03     private boolean isOn;           // 켜짐이나 꺼짐과 같은 램프의 상태
04
05     // 메소드 정의
06     public void turnOn()    {    isOn = true;    }
07     public void turnOff()   {    isOn = false;   }
08     public String toString() {
09         return "현재 상태는 " + (isOn == true ? "켜짐" : "꺼짐");
10     }
11 }
```



# 예제:

*DeskLampTest.java*

```
01 public class DeskLampTest {  
02     public static void main(String[] args) {  
03         // 객체를 생성하려면 new 예약어를 사용한다.  
04         DeskLamp myLamp = new DeskLamp();  
05  
06         // 객체의 메소드를 호출하려면 도트 연산자인 .을 사용한다.  
07         myLamp.turnOn();  
08         System.out.println(myLamp);  
09         myLamp.turnOff();  
10         System.out.println(myLamp);  
11     }  
12 }
```

현재 상태는 켜짐

현재 상태는 꺼짐

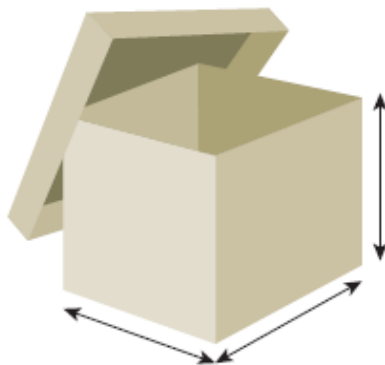




## 예제:

상자를 나타내는 Box 클래스를 작성하여 보자. Box 클래스는 가로 길이, 세로 길이, 높이를 나타내는 필드를 가지고. 상자의 부피를 계산하는 메소드를 가진다.

Box 클래스를 정의하고 Box 객체를 하나 생성한다. Box 객체 안의 가로 길이, 세로 길이, 높이를 20, 20, 30으로 설정하여 보자. 상자의 부피를 출력하여 본다.



상자의 가로, 세로, 높이는 20, 20, 30입니다.

상자의 부피는 12000.0입니다.



# 예제:

*BoxTest.java*

```
01  class Box {
02      int width;
03      int length;
04      int height;
05      double getVoume() {          return (double) width*height*length;    }
06  }
07  public class BoxTest {
08      public static void main(String[] args) {
09          Box b;
10          b = new Box();
11          b.width = 20;
12          b.length = 20;
13          b.height = 30;
14          System.out.println("상자의 가로, 세로, 높이는 " + b.width + ", " +
15                              b.length + ", " + b.height + "입니다.");
16          System.out.println("상자의 부피는 " + b.getVoume() + "입니다.");
17      }
18  }
```



# 예제: Television 클래스 작성하고 객체 생성해보기



클래스



객체



객체

Television.java

```
01 public class Television {
02     int channel;        // 채널 번호
03     int volume;         // 볼륨
04     boolean onOff;      // 전원 상태
05
06     public static void main(String[] args) {
07         Television myTv = new Television();
08         myTv.channel = 7;
09         myTv.volume = 10;
10         myTv.onOff = true;
11
12         Television yourTv = new Television();
13         yourTv.channel = 9;
14         yourTv.volume = 12;
15         yourTv.onOff = true;
16         System.out.println("나의 텔레비전의 채널은 " + myTv.channel +
17                             "이고 볼륨은 " + myTv.volume + "입니다.");
18         System.out.println("너의 텔레비전의 채널은 " + yourTv.channel +
19                             "이고 볼륨은 " + yourTv.volume + "입니다.");
20     }
21 }
```

myTv와 yourTv는  
별개의 객체를 가리킨다.

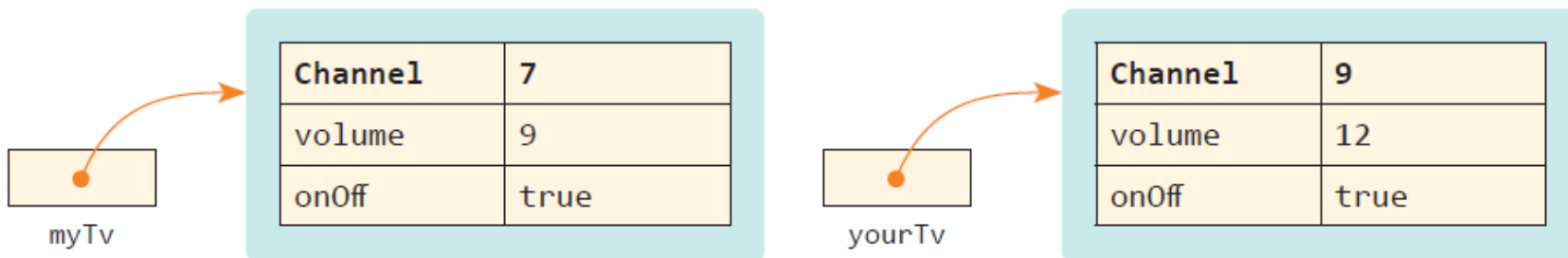
실행 결과

나의 텔레비전의 채널은 7이고 볼륨은 10입니다.  
너의 텔레비전의 채널은 9이고 볼륨은 12입니다



# 예제: Television 클래스 작성하고 객체 생성해보기

아래 그림에서 보듯이 myTv의 데이터는 yourTv의 데이터와 완전히 분리되어 있다.





# 중간 점검

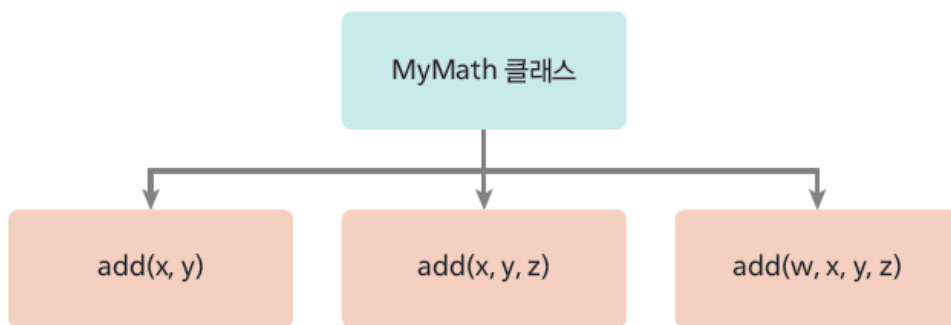
1. 객체를 생성시키는 연산자는 무엇인가?
2. 클래스와 객체와의 관계는 무엇이라고 할 수 있는가?
3. 각 객체가 가지고 있는 데이터는 모두 동일한가? 아니면 객체마다 다른가?
4. 자바에서 객체를 참조하려면 변수를 어떻게 선언하여야 하는가?
5. 다음의 코드에서 잘못된 부분은 어디인가?

```
Circle obj;  
obj.radius = 100;  
obj.color = "blue";
```



# 메소드 오버로딩

- 자바에서는 같은 이름의 메소드가 여러 개 존재할 수 있다. 이것을 메소드 오버로딩(**method overloading**)이라고 한다. 오버로딩을 우리말로 번역하자면 “중복정의”라고 할 수 있다.



메소드 오버로딩이란 이름이 같은 메소드를 여러 개 정의하는 것입니다. 다만 각각의 메소드가 가지고 있는 매개 변수는 달라야 합니다.





# 예제

MyMath.java

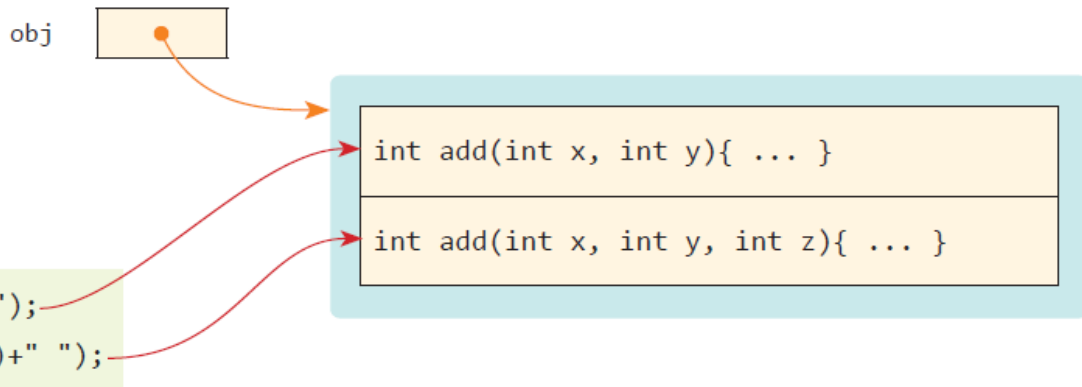
```
01 public class MyMath {  
02  
03     int add(int x, int y)          { return x+y; }  
04     int add(int x, int y, int z)   { return x+y+z; }  
05     int add(int x, int y, int z, int w) { return x+y+z+w; }  
06  
07     public static void main(String[] args) {  
08         MyMath obj;  
09         obj = new MyMath();  
10         System.out.print(obj.add(10, 20)+" ");  
11         System.out.print(obj.add(10, 20, 30)+" ");  
12         System.out.print(obj.add(10, 20, 30, 40)+" ");  
13     }  
14 }
```

매개 변수만 다르면 메소드 이름은 같아도 된다. 이것을 메소드 중복지의라고 한다.

30 60 100



# 메소드 오버로딩



메소드의 반환형만 다르다고 중복정의할 수는 없다. 반드시 메소드 매개변수가 달라야 한다.

```
int add(int x, int y) { ... }  
double add(int x, int y) { ... } // 중복 안 됨!
```

TIP

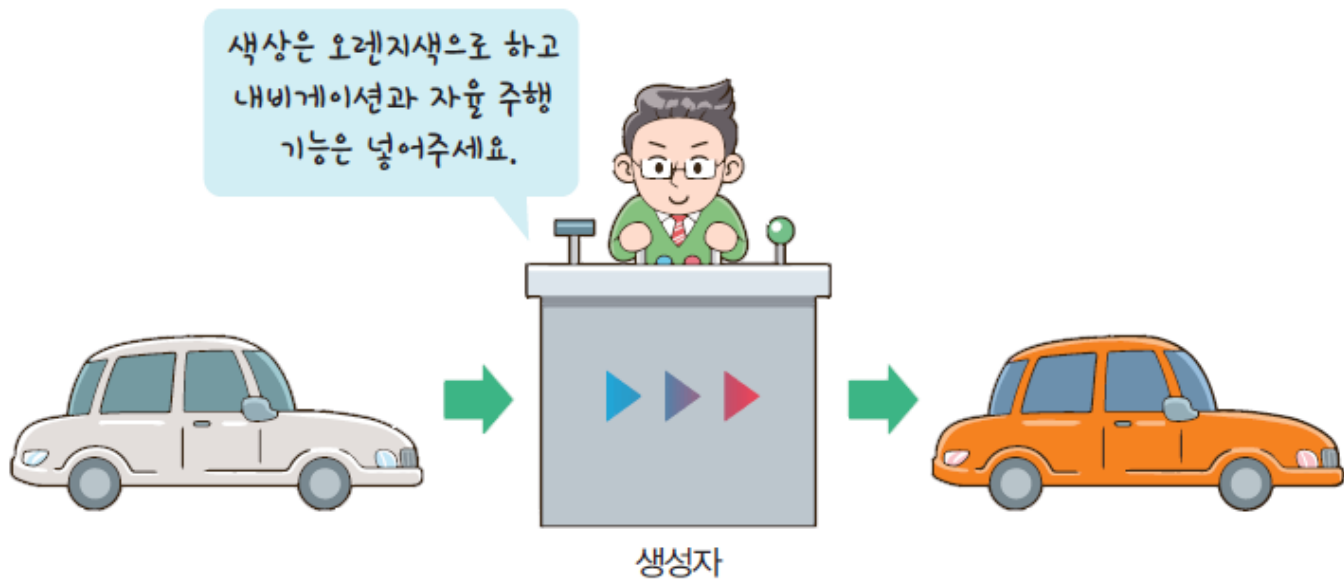






# 생성자

- 생성자(constructor)는 객체가 생성될 때 객체를 초기화하는 특수한 메소드이다.





# 생성자

PizzaTest.java

```
01  class Pizza
02  {
03      int size;
04      String type;
05
06      public Pizza() {
07          size = 12;
08          type = "슈퍼슈프림";
09      }
10
11      public Pizza(int s, String t) {
12          size = s;
13          type = t;
14      }
15  }
16  public class PizzaTest {
17      public static void main(String[] args) {
18          Pizza obj1 = new Pizza();
19          System.out.println("(" + obj1.type + " , " + obj1.size + ",)");
20
21          Pizza obj2 = new Pizza(24, "포테이토");
22          System.out.println("(" + obj2.type + " , " + obj2.size + ",)");
23      }
24  }
```

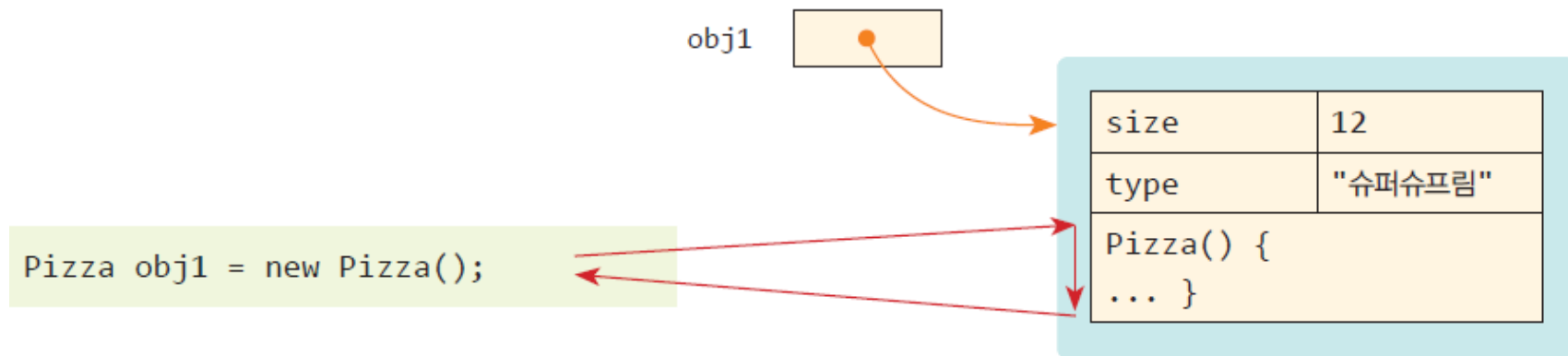
매개 변수가 없는 생성자

매개 변수가 있는 생성자

(슈퍼슈프림 , 12,)  
(포테이토 , 24,)



# 생성자





# 예제



## 생성자 만들어 보기

## 예제 4-4

앞의 Television 클래스에 생성자를 추가하여서 업그레이드 하여 보자. 생성자는 객체가 생성될 때, channel, volume, onOff 필드를 초기화한다.

*Television.java*

```
25 public class Television {
26     private int channel; // 채널 번호
27     private int volume; // 볼륨
28     private boolean onOff; // 전원 상태
29
30     Television(int c, int v, boolean o) {
31         channel = c;
32         volume = v;
33         onOff = o;
34     }
35
36     void print() {
37         System.out.println("채널은 " + channel + "이고 볼륨은 " + volume + "입니다.");
```

생성자가 정의되었다.



# 예제

```
38     }  
39 }  
40 public class TelevisionTest {  
41     public static void main(String[] args) {  
42         Television myTv = new Television(7, 10, true);  
43         myTv.print();  
44  
45         Television yourTv = new Television(11, 20, true);  
46         yourTv.print();  
47     }  
48 }
```

채널은 7이고 볼륨은 10입니다.

채널은 11이고 볼륨은 20입니다.



# 기본 생성자

- 기본 생성자(default constructor)는 매개 변수가 없는 생성자이다. 만약 개발자가 생성자를 하나도 정의하지 않으면 자바 컴파일러는 기본 생성자를 자동으로 만든다

*BoxTest.java*

```
01 public class Box {  
02     int width, height, depth;  
03  
04 }  
05 public class BoxTest {  
06     public static void main(String[] args) {  
07         Box b = new Box();  
08         System.out.println("상자의 크기: (" + b.width + "," + b.height + ","  
09                                     + b.depth + ")");  
10     }  
11 }
```

상자의 크기: (0,0,0)



# 기본 생성자가 추가되는 않는 경우

Box.java

```
01 public class Box {
02     int width, height, depth;
03     public Box(int w, int h, int d) { width=w; height=h; depth=d; }
04
05
06
07     public static void main(String[] args) {
08         Box b = new Box(); // 오류 발생!!
09         System.out.println("상자의 크기: (" + b.width + "," + b.height + ","
10                               + b.depth + ")");
11     }
12 }
```



# this 참조 변수

- this는 현재 객체 자신을 가리키는 참조 변수이다. this는 컴파일러에서 자동으로 생성한다. 흔히 생성자에서 매개 변수 이름과 필드 이름이 동일한 경우에 혼동을 막기 위해서 사용한다

```
01 public class Circle {  
02     int radius;  
03  
04     public Circle(int radius) {  
05         this.radius = radius;  
06     }  
07     double getArea() {  
08         return 3.14*radius*radius;  
09     }  
10 }
```

this.radius는 필드이고  
radius는 매개 변수라는  
것을 알 수 있네요.



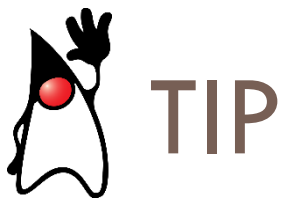


# this()

- this()는 다른 생성자를 의미한다.

```
01 public class Circle {  
02     int radius;  
03  
04     public Circle(int radius) {  
05         this.radius = radius;  
06     }  
07  
08     public Circle() {  
09         this(0);  
10     }  
11  
12     double getArea() {  
13         return 3.14 * radius * radius;  
14     }  
15 }
```

Circle(0)을 호출한다.



# TIP

이클립스는 생성자를 자동으로 만들어주는 메뉴를 가지고 있다. 클래스 안에 커서를 두고 [Source] → [Generate Constructor using Fields]를 사용해보자. 한 번만 사용해보면 금방 익숙해질 것이다. 생성자에서 초기화할 필드만 체크하면 된다.

TIP





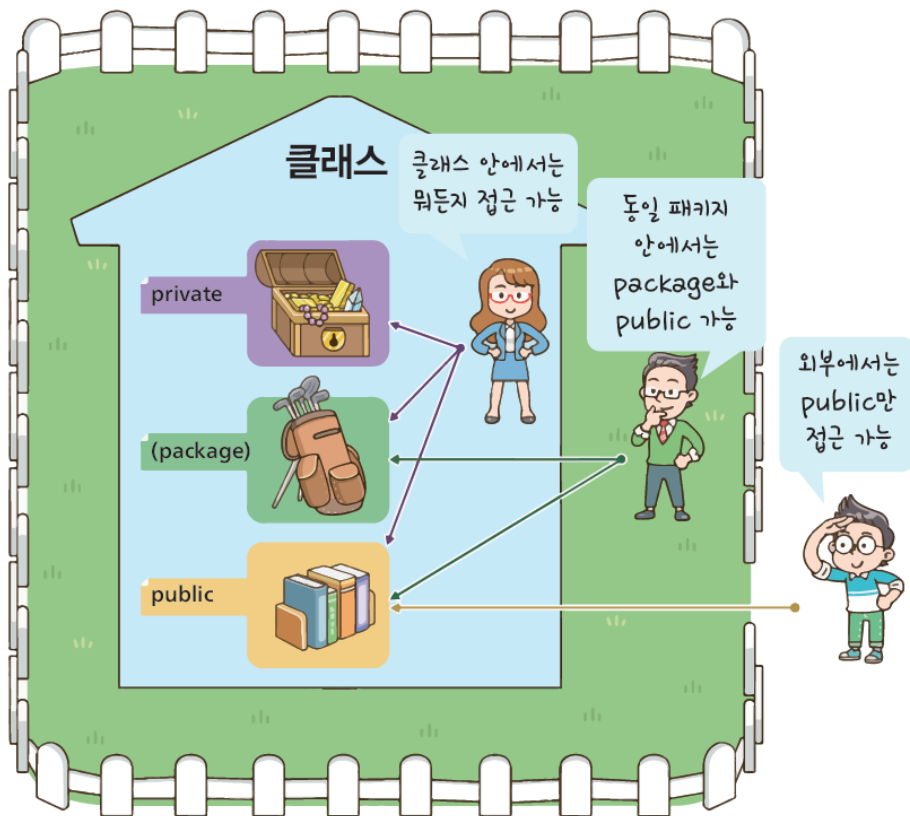
# 중간점검

1. 만약 클래스 이름이 MyClass라면 생성자의 이름은 무엇이어야 하는가?
2. 생성자의 반환형은 무엇인가?
3. 생성자 안에서 this()의 의미는 무엇인가?
4. this의 주된 용도는 무엇인가?
5. 같은 이름의 메소드를 중복하여 정의하는 것을 \_\_\_\_\_라고 한다.
6. 메소드 : 오버로딩에서는 무엇으로 이름이 동일한 메소드를 구별하는가?



# 접근 제어

- 접근 제어(access control)란 클래스의 멤버에 접근하는 것을 제어하는 것이다. **public**이나 **private**의 접근 지정자를 멤버 앞에 붙여서 접근을 제한하게 된다





# 자바의 접근 제어 지정자

| 접근 지정자    | 동일한 클래스 | 패키지 | 자식 클래스 | 전체 |
|-----------|---------|-----|--------|----|
| public    | O       | O   | O      | O  |
| protected | O       | O   | O      | X  |
| 없음        | O       | O   | X      | X  |
| private   | O       | X   | X      | X  |



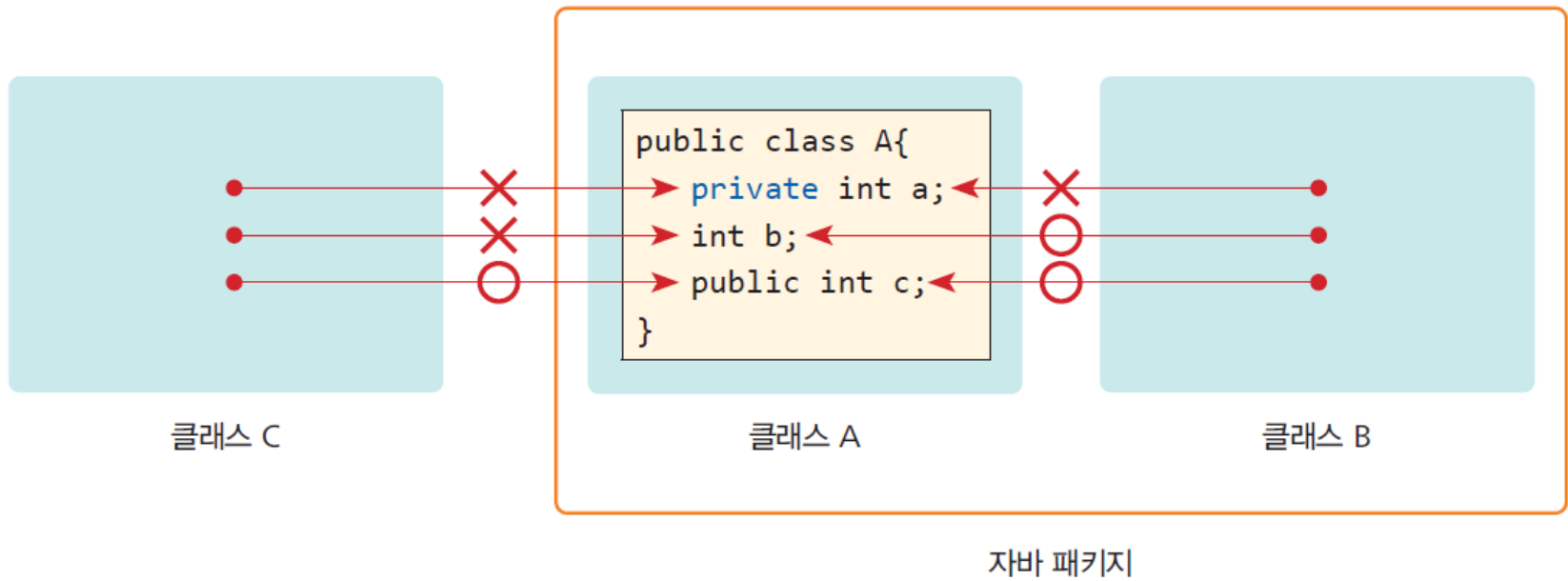
# 예제

Test.java

```
01  class A {
02      private int a;      // 전용
03      int b;              // 디폴트
04      public int c;       // 공용
05  }
06
07  public class Test {
08      public static void main(String args[]) {
09
10          A obj = new A();  // 객체 생성
11
12          obj.a = 10;      // 전용 멤버는 다른 클래스에서는 접근 안 됨
13          obj.b = 20;       // 디폴트 멤버는 접근할 수 있음
14          obj.c = 30;       // 공용 멤버는 접근할 수 있음
15      }
16  }
```



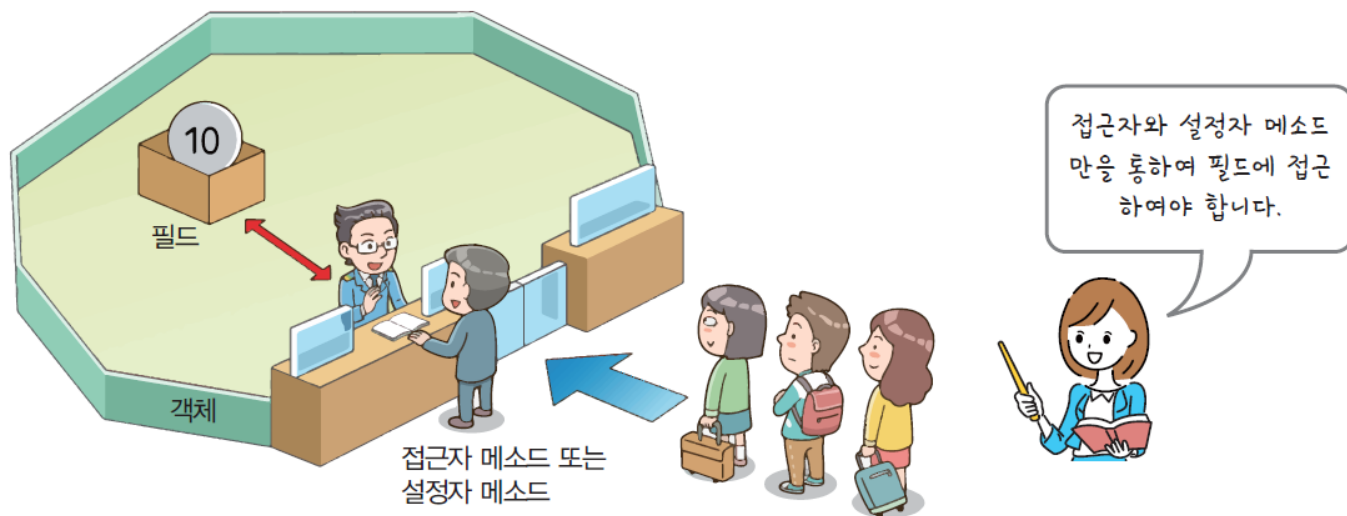
# 접근 제어





# 접근자와 설정자

- 필드값을 반환하는 접근자(getters), 필드값을 설정하는 설정자(setters)이다. 이러한 메소드는 대개 **get**이나 **set**이 메소드 이름 앞에 붙여진다. 예를 들면 **getBalance()**는 접근자이고 **setBalance()**는 설정자이다.







# 접근자와 설정자 예

Account.java

```
01 public class Account {  
02     private int regNumber;  
03     private String name;  
04     private int balance;  
05  
06     public String getName() { return name; }  
07     public void setName(String name) { this.name = name; }  
08     public int getBalance() { return balance; }  
09     public void setBalance(int balance) { this.balance = balance; }  
10  
11 }
```

필드가 모두 private로 선언되었다. 클래스 내부에서만 사용이 가능하다.

접근자와 설정자를 사용하고 있다.

```
12 public class AccountTest {  
13     public static void main(String[] args) {  
14         Account obj = new Account();  
15         obj.setName("Tom");  
16         obj.setBalance(100000);  
17         System.out.println("이름은 " + obj.getName() + " 통장 잔고는 "  
18             + obj.getBalance() + "입니다.");  
19     }  
20 }
```

이름은 Tom 통장 잔고는 100000입니다.



# 접근자와 설정자를 사용하는 이유

- 접근자와 설정자를 사용해야만 나중에 클래스를 업그레이드할 때 편하다.
- 접근자에서 매개 변수를 통하여 잘못된 값이 넘어오는 경우, 이를 사전에 차단할 수 있다.
- 필요할 때마다 필드값을 동적으로 계산하여 반환할 수 있다.
- 접근자만을 제공하면 자동적으로 읽기만 가능한 필드를 만들 수 있다.

```
public void setAge(int age)
{
    if( age < 0 )
        this.age = 0;
    else
        this.age = age;
}
```



# 접근 제어

1. 필드의 경우, private로 만드는 것이 바람직한 이유는 무엇인가?
2. 필드를 정의할 때 아무런 접근 제어 수식자를 붙이지 않으면 어떻게 되는가?



# Lab: 안전한 배열 만들기

- 만약 인덱스가 배열의 크기를 벗어나게 되면 실행 오류가 발생한다. 따라서 실행 오류를 발생하지 않는 안전한 배열을 작성하여 보자.

| <<SafeArray>>  |
|--|
| -a[]: int<br>+length: int                            |
| +get(index: int):int<br>+put(index: int, value: int) |



# Lab: 안전한 배열 만들기

*SafeArrayTest.java*

```
01  public class SafeArray {
02      private int a[];
03      public int length;
04
05      public SafeArray(int size) {
06          a = new int[size];
07          length = size;
08      }
09
10      public int get(int index) {
11          if (index >= 0 && index < length) {
12              return a[index];
13          }
14          return -1;
15      }
```



# Lab: 안전한 배열 만들기

```
15
16
17 public void put(int index, int value) {
18     if (index >= 0 && index < length) {
19         a[index] = value;
20     } else
21         System.out.println("잘못된 인덱스 " + index);
22 }
23
24 public class SafeArrayTest {
25     public static void main(String args[]) {
26         SafeArray array = new SafeArray(3);
27
28         for (int i = 0; i < (array.length + 1); i++) {
29             array.put(i, i * 10);
30         }
31     }
32 }
```

설정자에서 잘못된 인덱스  
번호를 차단할 수 있다.

잘못된 인덱스 3



# 무엇을 클래스로 만들어야 할까?

- TV, 자동차와 같이, 실제 생활에서 사용되는 객체들은, 클래스로 작성하기가 비교적 쉽다. 하지만 프로그램 안에서는 실제 생활에는 사용되지 않는, 추상적인 객체들도 많이 존재한다.
- 요구 사항이 문서화되면 클래스 식별 과정을 시작할 수 있다. 요구 사항에서 클래스를 식별하는 한 가지 방법은 모든 “명사”를 표시하는 것이다.

은행은 정기 예금 계좌와 보통 예금 계좌를 제공한다. 고객들은 자신의 계좌에 돈을 입금할 수 있으며 계좌에서 돈을 인출할 수 있다. 그리고 각 계좌는 기간에 따라 이자를 지급한다. 계좌마다 이자는 달라진다.





# 메소드 결정

- 요구 사항 문서에서 “동사”에 해당되는 부분이 메소드가 되는 경우가 많다. 앞의 요구 사항 문서에서 “고객들은 자신의 계좌에 돈을 입금할 수 있으며 계좌에서 돈을 인출할 수 있다.”라는 문장이 있다. 여기서 “입금한다”, “인출한다”와 같은 동사는 **deposit()**, **withdraw()** 메소드로 매핑시킬 수 있다.

은행은 정기 예금 계좌와 보통 예금 계좌를 제공한다. 고객들은 자신의 계좌에 돈을 입금할 수 있으며 계좌에서 돈을 인출할 수 있다. 그리고 각 계좌는 기간에 따라 이자를 지급한다. 계좌마다 이자는 달라진다.



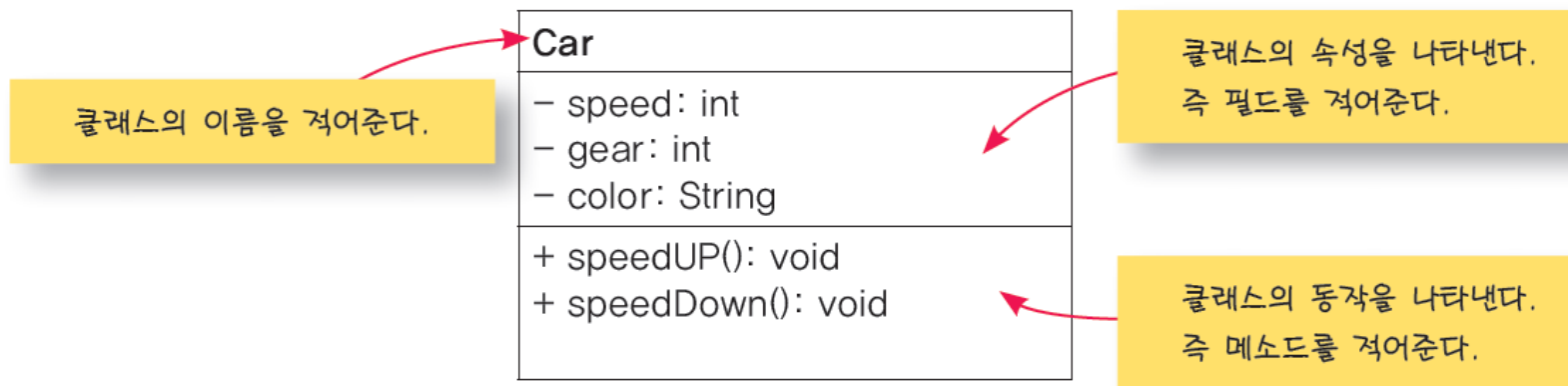




# 클래스 간의 관계를 결정한다.

- 이러한 클래스 간의 관계를 설정하는 부분은 객체 지향 설계에서 가장 어려운 부분이다. 개발자는 많은 결정을 내려야 한다. 가장 중요한 결정은 다음의 2가지이다.
  - 상속(Inheritance)
  - 구성(Composition)

- **UML(Unified Modeling Language):** UML은 클래스만을 그리는 도구는 아니고 객체지향설계 시에 사용되는 일반적인 모델링 언어라고 할 수 있다.
- **UML**을 사용하면 소프트웨어를 본격적으로 작성하기 전에 구현하고자 하는 시스템을 시각화하여 검토할 수 있다.





# 가시성 표시자


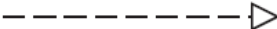





- 필드나 메소드의 이름 앞에는 가시성 표시자(visibility indicator)가 올 수 있다.
- +는 public을, -는 private을 의미한다.

|   |           |
|---|-----------|
| + | Public    |
| - | Private   |
| # | Protected |
| / | Derived   |
| ~ | Package   |



# 클래스 간의 관계

표 4-1 UML에서 사용되는 화살표의 종류

| 관계                                   | 화살표   |
|--------------------------------------|---|
| 일반화(generalization), 상속(inheritance) |  |
| 구현(realization)                      |  |
| 구성관계(composition)                    |  |
| 집합관계(aggregation)                    |  |
| 유향 연관(direct association)            |  |
| 양방향 연관(bidirectional association)    |  |
| 의존(dependency)                       |  |



# 의존 관계

- 의존(dependency)이란 하나의 클래스가 다른 클래스를 사용하는 관계이다.

*CarTest.java*

```
01 public class CarTest {  
02     public static void main(String[] args) {  
03         Car myCar = new Car();  
04         myCar.speedUp();  
05     }  
06 }
```

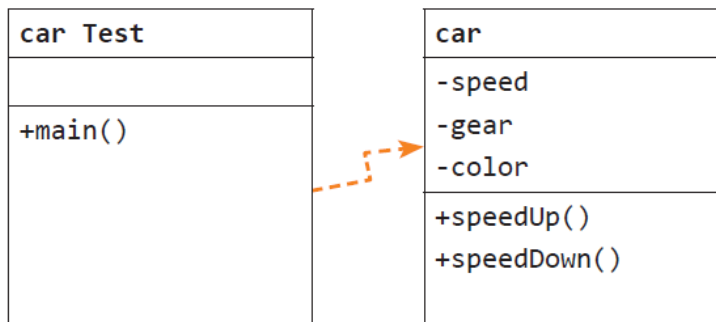


그림 4.6 Car 예제의 UML



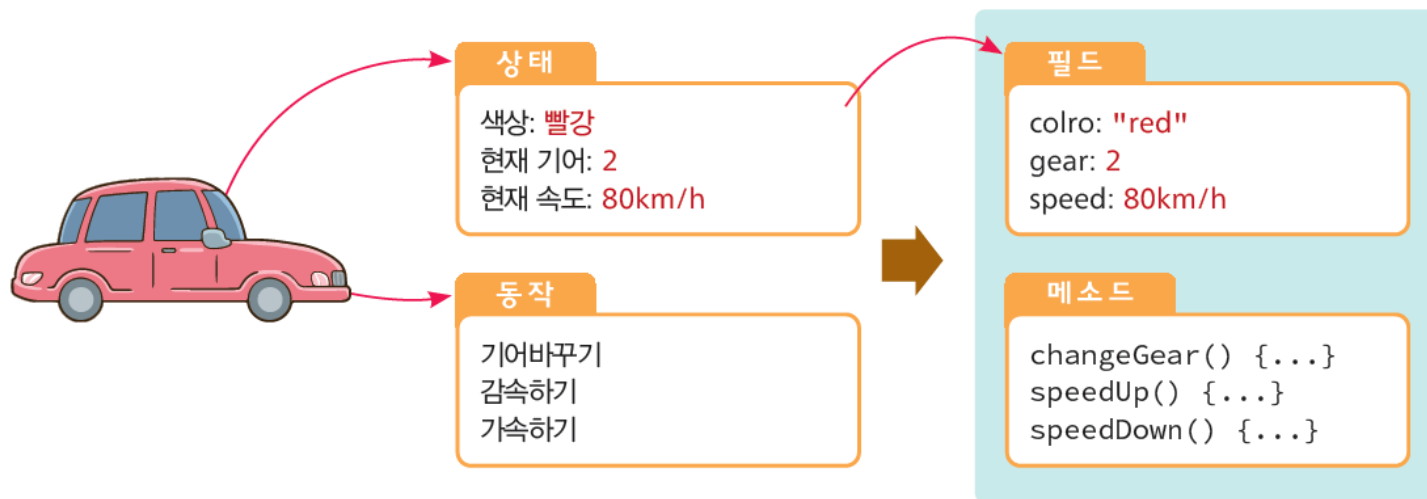
# 장갑점검

1. 작업 명세서가 주어졌다고 하자. 클래스 후보는 어떻게 찾을 수 있는가?
2. 작업 명세서가 주어졌다고 하자. 메소드 후보는 어떻게 찾을 수 있는가?
3. TV를 나타내는 클래스를 정의하고 UML의 클래스 다이어그램으로 표현하여 보라.



# Lab: 자동차 클래스 작성

- 자동차를 나타내는 클래스를 정의하여 보자. 예를 들어, 자동차 객체의 경우 속성은 색상, 현재 속도, 현재 기어 등이다. 자동차의 동작은 기어 변속하기, 가속하기, 감속하기 등이 있다.





# Sol:

CarTest.java

```
01  class Car {
02      String color; // 색상
03      int speed;    // 속도
04      int gear;     // 기어
05
06      @Override
07      public String toString() {
08          return "Car [color=" + color + ", speed=" + speed + ", gear=" + gear + "];"
09      }
10
11      void changeGear(int g) {          gear = g;          }
12      void speedUp() {                  speed = speed + 10;  }
13      void speedDown() {                speed = speed - 10;  }
14  }
15
16  public class CarTest {
17      public static void main(String[] args) {
18
19          Car myCar = new Car();
20          myCar.changeGear(1);
21          myCar.speedUp();
22          System.out.println(myCar);
23      }
24  }
```

toString() 메소드는 이클립스에서 자동으로 생성시킬 수 있다. [Source] → [Generate toString()...] 메뉴를 사용해보자.

Car [color=null, speed=10, gear=1]





# Lab: 은행 계좌 클래스 작성

- 은행 계좌를 클래스로 정의하여 보자. 먼저 잔액은 변수 **balance**로 표시한다. 예금 입금은 **deposit()** 메소드로, 예금 인출은 **withdraw()** 메소드로 나타내면 된다.

| BankAccount   |
|---|
| -owner : string<br>-accountNumber : int<br>-balance : int |
| +deposit()<br>+withdraw()                                 |





# Sol:

*BankAccount.java*

```
01 class BankAccount { // 은행 계좌
02     int accountNumber; // 계좌 번호
03     String owner; // 예금주
04     int balance; // 잔액을 표시하는 변수
05
06     void deposit(int amount) {           balance += amount;   }
07     void withdraw(int amount) {          balance -= amount;   }
08     public String toString(){
09         return "현재 잔액은 " + balance + "입니다.";
10     }
11 }
12
13 public class BankAccountTest {
14     public static void main(String[] args) {
15         BankAccount myAccount = new BankAccount();
16         myAccount.deposit(10000);
17         System.out.println(myAccount);
18         myAccount.withdraw(8000);
19         System.out.println(myAccount);
20
21     }
22 }
```

현재 잔액은 10000입니다.

현재 잔액은 2000입니다.



# Lab: 윈도우 생성해보기

- 자바에서 윈도우를 나타내는 클래스는 **JFrame**이다. 윈도우를 만들 때 가장 힘든 작업이 윈도우의 기본 값들을 설정하는 부분이다. 객체 지향 방법에서는 “생성자” 개념이 있어서 이 부분이 아주 쉽게 가능하다. 개발자가 아무 것도 설정하지 않아도 생성자에서 기본값으로 설정된다





# Sol: 윈도우 생성해보기

FrameTest.java

01 **import** javax.swing.\*;

① javax.swing이라는 패키지  
안의 모든 클래스를 읽어들인다.

02

03 **public class** FrameTest {

04 **public static void** main(String[] args) {

05 JFrame f = **new** JFrame("Frame Test");

② JFrame 클래스의 객체를 생  
성하고, 생성자를 호출한다.

06

07 f.setSize(300, 200);

08 f.setVisible(**true**);

③ JFrame 클래스의 메소드를  
호출한다.

09

}

10 }



# Mini Project: 주사위 클래스

- 주사위를 **Dice** 클래스로 모델링한다. **Dice** 클래스는 주사위면(**face**)을 필드로 가지고 있고 **roll()**, **getValue()**, **setValue()** 등의 메소드를 가지고 있다. 생성자에서는 주사위면을 0으로 초기화한다.



주사위1= 5 주사위2= 5  
주사위1= 3 주사위2= 4  
주사위1= 1 주사위2= 1  
(1, 1)이 나오는데 걸린 횟수= 3



# Summary

- 객체 지향 방법은 실세계가 객체로 구성되어 있는 것처럼 소프트웨어도 객체로 구성하는 방법론이다.
- 절차 지향은 함수를 사용하여 프로그램을 작성하는 방법이다. 함수는 재사용하기가 어렵다.
- 객체 지향을 이루고 있는 핵심적인 개념에는 “캡슐화”, “상속”, “다형성”, “추상화” 등이 있다.
- 캡슐화는 객체의 속성과 동작을 하나로 묶는 것을 의미한다. 내부 구현을 알 수 없게 은닉하는 것도 캡슐화에 포함된다.
- 상속은 다른 클래스를 재사용하는 강력한 방법이다.
- 객체는 속성과 동작으로 정의된다. 프로그램에서는 속성은 필드로, 동작은 메소드로 구현된다.
- 자바에서는 **new**를 사용하여 객체를 생성한다.
- 객체를 생성하기 전에 반드시 객체를 참조하는 변수를 먼저 선언하여야 한다.
- 메소드 오버로딩이란 이름은 동일하지만 매개 변수가 다른 메소드를 여러 개 정의하는 것이다.





# Q & A

