객체 지향 개념
클래스, 객체, 메소드

514760-1
2019년 봄학기
3/26/2019
박경신

OOP (Object-Oriented Programming)

- 소프트웨어의 생산성 향상
  - 컴퓨터 산업 발전에 따라 소프트웨어의 생명 주기(life cycle) 단축
  - 객체 지향 언어는 상속, 다형성, 객체, 캡슐화 등 소프트웨어 재사용을 위한 여러 장치 내장
    - 소프트웨어의 재사용과 부분 수정을 통해 소프트웨어를 다시 만드는 부담을 대폭 줄임으로써 소프트웨어의 생산성이 향상
- 실세계에 대한 쉬운 모델링
  - 과거
    - 수학 계산/통계 처리를 하는 등의 처리 과정, 계산 절차가 중요
  - 현재
    - 컴퓨터가 산업 전반에 활용
    - 실세계에서 발생하는 일을 프로그래밍
      - 실세계에서는 절차나 과정보다 일과 관련된 물체(객체)들의 상호 작용으로 묘사하는 것이 용이
  - 실세계의 일을 보다 쉽게 프로그래밍하기 위한 객체 중심의 객체 지향 언어 탄생

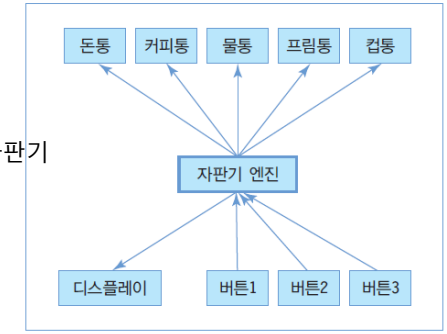
절차 지향 프로그래밍 vs OOP

- 절차 지향 프로그래밍 (Procedural Programming)
  - 작업 순서를 표현하는 컴퓨터 명령 집합
  - 함수들의 집합으로 프로그램 작성
- 객체 지향 프로그래밍 (Object Oriented Programming)
  - 프로그램을 실제 세상에 가깝게 모델링
  - 컴퓨터가 수행하는 작업을 객체들간의 상호 작용으로 표현
  - 클래스 혹은 객체들의 집합으로 프로그램 작성

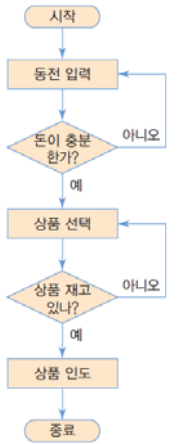
절차 지향 프로그래밍 vs OOP

- OOP – 실세계를 모델링하여 프로그래밍하는 방법

커피자판기

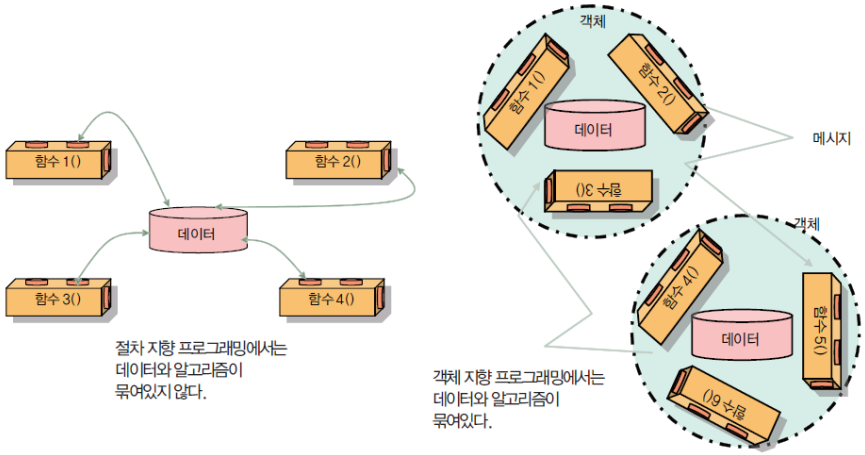


객체지향적 프로그래밍의 객체들의 상호 관련성



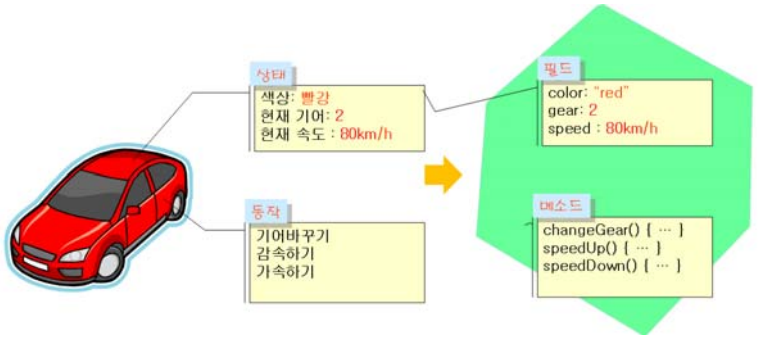
절차지향적 프로그래밍의 실행 절차

절차 지향 프로그래밍 vs OOP



객체(Object)

- 객체(object)는 상태(state)와 동작(behavior)을 가짐.
- 객체의 상태(state)는 객체의 속성임.
- 객체의 동작(behavior) 또는 행동은 객체가 할 수 있는 동작임.
- **상태**는 **필드(field)**로 **동작**은 **메소드(method)**로 구현됨.



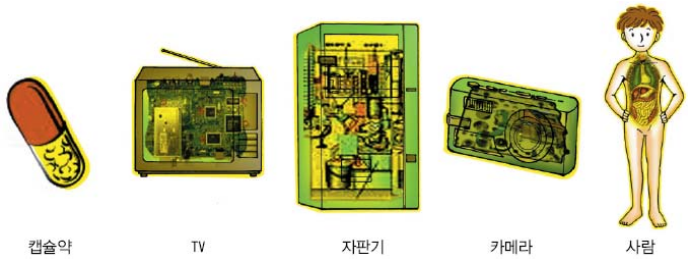
객체 지향의 3대 특징

- 캡슐화 (Encapsulation)
- 상속 (Inheritance)
- 다형성 (Polymorphism)

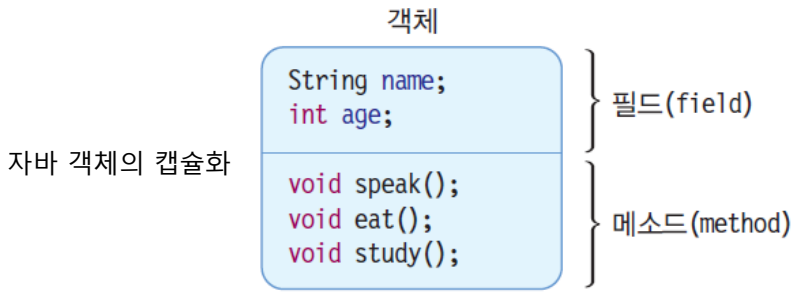
캡슐화(Encapsulation)

- 캡슐화(Encapsulation)
  - 관련된 데이터와 알고리즘 (코드)를 하나의 덩어리로 묶는 것
  - 메소드(함수)와 데이터를 클래스 내에 선언하고 구현
  - 외부에서는 공개된 메소드의 인터페이스만 접근 가능
    - 외부에서는 비공개 데이터에 직접 접근하거나 메소드의 구현 세부를 알 수 없음
  - 객체 내 데이터에 대한 보안, 보호, 외부 접근 제한

실세계의 캡슐화



캡슐화 (Encapsulation)

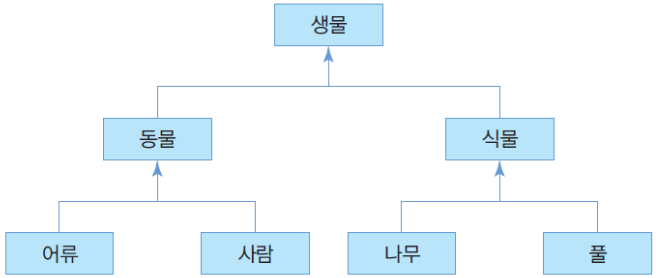


캡슐화와 정보은닉

- 정보 은닉이 가능하기 때문에 업그레이드기 쉽게 가능
- 정보 은닉**(information hiding)**은 객체를 캡슐로 싸서 객체의 내부를 보호하는 하는 것이다. 즉 객체의 실제 구현 내용을 외부에 감추는 것임.

상속(Inheritance)

- 상속(Inheritance): 이미 작성된 클래스(부모 클래스)를 이어받아서 새로운 클래스(자식 클래스)를 생성하는 기법
- 기존의 코드를 재활용하기 위한 기법



상속의 개념

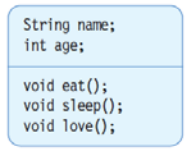
상속(Inheritance)

```
class Animal {
    String name;
    int age;
    void eat() {...}
    void sleep() {...}
    void love() {...}
}
```

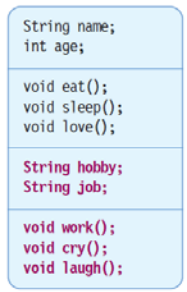
상속

```
class Human extends Animal {
    String hobby;
    String job;
    void work() {...}
    void cry() {...}
    void laugh() {...}
}
```

Animal의 객체



Human의 객체



상속

- 상위 클래스의 특성을 하위 클래스가 물려받음
  - 상위 클래스 : 수퍼 클래스, 하위 클래스 : 서브 클래스
- 서브 클래스
  - 수퍼 클래스 코드의 재사용
  - 새로운 특성 추가 가능
- 자바는 클래스 다중 상속 없음
  - 인터페이스를 통해 다중 상속과 같은 효과 얻음

다형성(Polymorphism)

- 다형성(Polymorphism)
  - 동일한 이름으로 많은 상황에 대처하는 기법
  - 자바의 다형성 사례
    - 슈퍼 클래스의 메소드를 서브 클래스마다 다르게 구현하는 메소드 오버라이딩(overriding)

클래스와 객체

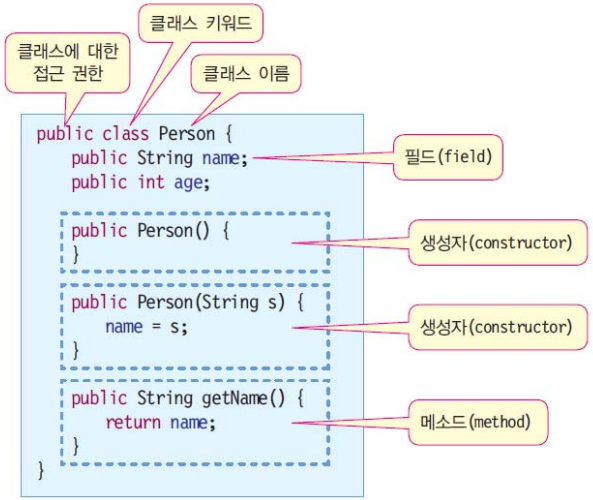
- 클래스
  - 객체의 속성과 행위 선언
  - 객체의 설계도 혹은 틀
- 객체
  - 클래스의 틀로 찍어낸 실체
    - 메모리 공간을 갖는 구체적인 실체
    - 클래스를 구체화한 객체를 **인스턴스(instance)**라고 부름
    - 객체와 인스턴스는 같은 뜻으로 사용
- 사례
  - 클래스: 소나타자동차, 객체: 출고된 실제 소나타 100대
  - 클래스: 벽시계, 객체: 우리집 벽에 걸린 벽시계들
  - 클래스: 책상, 객체: 우리가 사용중인 실제 책상들

클래스와 객체

클래스: Person

이름, 직업,나이, 성별, 혈액형
밥 먹기, 잠자기, 말하기, 걷기

클래스 구조



클래스 선언

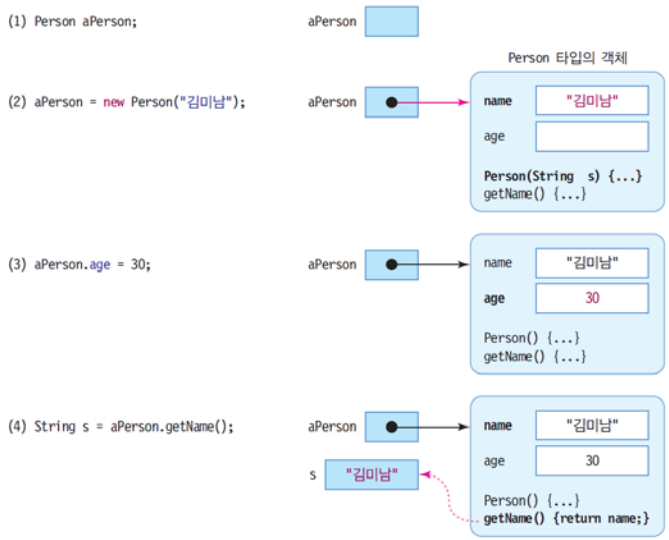
- 클래스 접근 권한, public
  - 다른 클래스들에서 이 클래스를 사용하거나 접근할 수 있음을 선언
- class Person
  - Person이라는 이름의 클래스 선언
  - 클래스는 {로 시작하여 }로 닫으며 이곳에 모든 필드와 메소드 구현
- 필드(field)
  - 값을 저장할 멤버 변수
    - 멤버 변수 혹은 필드라고 함
  - 필드의 접근 지정자 public
    - 필드를 다른 클래스의 메소드에서 접근할 수 있도록 공개한다는 의미
- 메소드(method)
  - 메소드는 함수이며 객체의 행위를 구현
  - 메소드의 접근 지정자 public
    - 메소드를 다른 클래스의 메소드에서 호출할 수 있도록 공개한다는 의미
- 생성자(constructor)
  - 클래스의 이름과 동일한 메소드
  - 클래스의 객체가 생성될 때만 호출되는 메소드

객체 생성

- 객체 생성
  - new 키워드를 이용하여 생성
    - new는 객체의 생성자 호출
- 객체 생성 과정
  1. 객체에 대한 레퍼런스 변수 선언
  2. 객체 생성

```
public static void main (String args[]) {
    Person aPerson; // 1. 레퍼런스 변수 aPerson 선언
    aPerson = new Person("김미남");  // 2. Person 객체 생성

    aPerson.age = 30;            // 객체 멤버 접근
    int i = aPerson.age;                    // 30
    String s = aPerson.getName();       // 객체 메소드 호출
}
```

객체 생성 및 사용 예



객체의 활용

- 객체의 멤버 접근 : **객체 레퍼런스.멤버**

```
public class ClassExample {
    public static void main (String args[]) {
        Person aPerson = new
        Person("홍길동");

        aPerson.age = 30;
        int i = aPerson.age;
        String s = aPerson.getName();
    }
}
```

객체의 필드에 값 대입

객체의 필드에서 값 읽기

객체의 메소드 호출

예제: 상품 (Goods) 클래스

```
public class Goods {
  String name;  // 상품 이름
  int price;       // 상품 가격
  int numberOfStock; // 재고 수량
  int numberOfSold;  // 팔린 수량
  public static void main(String[] args) {
    Goods camera = new Goods();
    camera.name = "Nikon";
    camera.price = 400000;
    camera.numberOfStock = 30;
    camera.numberOfSold = 50;
    System. out.println("상품 이름:" + camera.name);
    System.out.println("상품 가격:" + camera.price);
    System.out.println("재고 수량:" + camera.numberOfStock);
    System.out.println("팔린 수량:" + camera.numberOfSold);
  }
}
```

```
상품 이름:Nikon
상품 가격:400000
재고 수량:30
팔린 수량:50
```

예제: 자동차 (Car) 클래스

```
public class Car {
        String color; int speed; int gear;
        @Override
        public String toString() {
        return "Car [color=" + color + ", speed=" + speed + ", gear=" + gear + "]";
        }
        void setColor(String c) {          color = c;          }
        void speedUp() {          speed = speed + 10;          }
        void speedDown() {          speed = speed - 10;          }
        void changeGear(int g) {          gear = g;          }
}
public class CarTest {
        public static void main(String[] args) {
                Car myCar = new Car();
                myCar.setColor("red");
                myCar.changeGear(1);
                myCar.speedUp();
                System.out.println(myCar);
        }
}
```

```
Car [color=red, speed=10, gear=1]
```

예제 : 지수 클래스 (MyExp) 만들기

클래스 MyExp를 작성하라. MyExp는 지수값을 표현하는 클래스로서 두 개의 정수형 멤버 필드 base와 exp를 가진다. 2^3의 경우 base는 2이며, exp는 3이다. base와 exp는 양의 정수만을 가지는 것으로 가정한다. 또한 MyExp는 정수값을 리턴하는 getValue()라는 메소드를 제공한다. getValue()는 base와 exp 값으로부터 지수를 계산하여 정수 값으로 리턴한다. 예를 들어 MyExp객체의 base 필드가 2이고 exp가 3이라면 getValue()는 8을 리턴한다.

```
public class MyExp {
  int base;
  int exp;
  int getValue() {
    int res=1;
    for(int i=0; i<exp; i++)
      res = res * base;
    return res;
  }
  public static void main(String[] args) {
    MyExp number1 = new MyExp();
    number1.base = 2;
    number1.exp = 3;
    MyExp number2 = new MyExp();
    number2.base = 3;
    number2.exp = 4;
    System.out.println("2의 3승 = " + number1.getValue());
    System.out.println("3의 4승 = " + number2.getValue());
  }
}
```

```
2의 3승 = 8
3의 4승 = 81
```

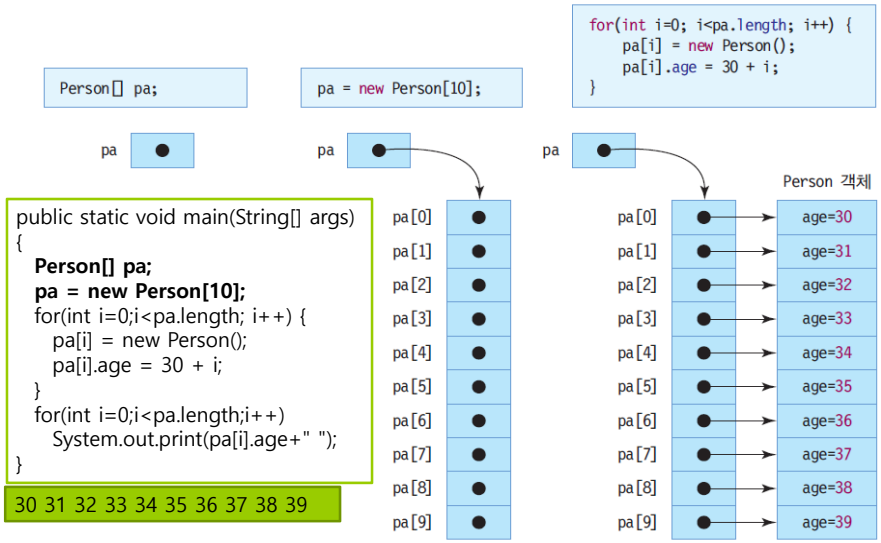
객체 배열

- 객체 배열 생성 및 사용

```
Person[] pa;                          // 배열에 대한 레퍼런스 선언
pa = new Person[10];                  // 레퍼런스 배열 생성
for(int i=0; i<pa.length; i++) {
   pa[i] = new Person();              // 배열의 원소 객체 생성
   pa[i].age = 30 + i;                // 객체 배열 사용
}
```

```
for(int i=0; i<pa.length; i++) // 배열 pa의 모든 원소 객체의 age를 출력한다.
   System.out.print(pa[i].age + " ");
```

객체 배열 선언과 생성 사례



예제: 객체 배열 생성

Scanner 클래스를 이용하여 상품을 입력 받아 Goods 객체를 생성하고 이들을 Goods 객체 배열에 저장하라. 상품을 3개 입력 받으면 이들을 모두 화면에 출력하라.

```
import java.util.Scanner;
public class GoodsArray {
  public static void main(String[] args) {
    Goods[] goodsArray;
    goodsArray = new Goods [3];
    Scanner s = new Scanner(System.in);
    for(int i=0; i<goodsArray.length; i++) {
      String name = s.next();
      int price = s.nextInt();
      int n = s.nextInt();
      int sold = s.nextInt();
      goodsArray[i] = new Goods(name, price, n, sold);
    }
    for(int i=0; i<goodsArray.length; i++) {
      System.out.print(goodsArray[i].getName()+" ");
      System.out.print(goodsArray[i].getPrice()+" ");
      System.out.print(goodsArray[i].getNumberOfStock()+" ");
      System.out.println(goodsArray[i].getSold());
    }
  }
}
```

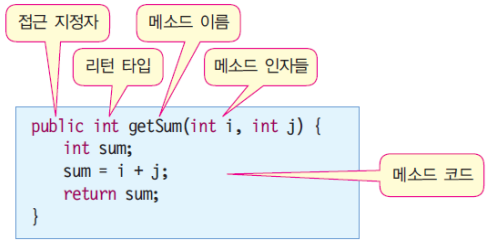
예제: 객체 배열 생성

```
class Goods {
  private String name;
  private int price;
  private int numberOfStock;
  private int sold;

  Goods(String n, int p, int nStack, int s) {
    name = n;
    price = p;
    numberOfStock = nStack;
    sold = s;
  }
  String getName() {return name;}
  int getPrice() {return price;}
  int getNumberOfStock() {return numberOfStock;}
  int getSold() {return sold;}
}
```

```
콜라 500 10 20
사이다 1000 20 30
맥주 2000 30 50
콜라 500 10 20
사이다 1000 20 30
맥주 2000 30 50
```

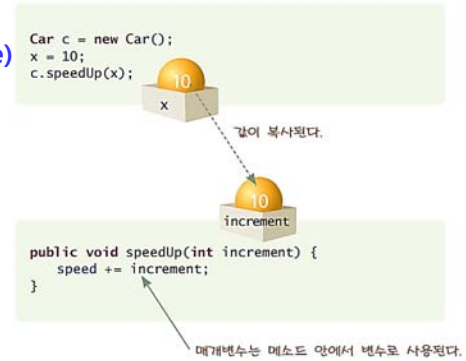
메소드 형식

- 메소드
  - 메소드는 C/C++의 함수와 동일
  - 자바의 모든 메소드는 반드시 클래스 안에 있어야 함(캡슐화 원칙)
- 메소드 구성 형식
  - 접근 지정자
    - public. private, protected, 디폴트(접근 지정자 생략된 경우)
  - 리턴 타입
    - 메소드가 반환하는 값의 데이터 타입



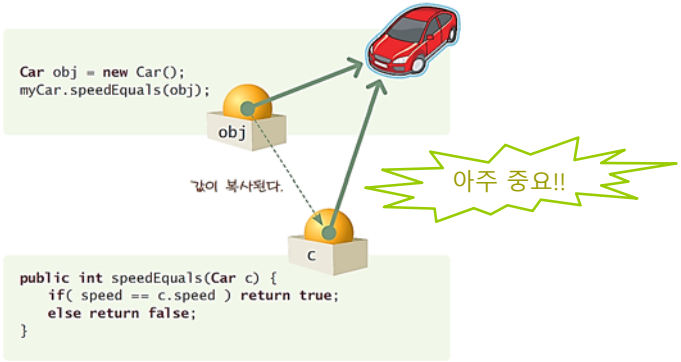
인자 전달 – 기본 타입

- 자바의 인자 전달 방식(Parameter Passing)
  - **값에 의한 호출(call by value)**
  - 기본 타입의 값을 전달하는 경우
    - 값이 복사되어 전달
    - 메소드의 매개 변수가 변경되어도 호출한 실인자 값은 변경되지 않음

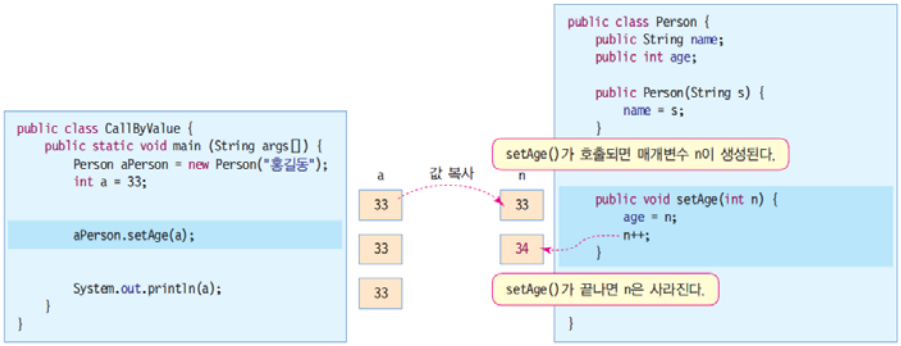


인자 전달 – 레퍼런스 타입

- 자바의 인자 전달 방식
  - 객체 혹은 배열을 전달하는 경우
    - 객체나 배열의 레퍼런스만 전달
      - 객체 혹은 배열이 통째로 복사되어 전달되는 것이 아님
    - 메소드의 매개 변수와 호출한 실인자가 객체나 배열을 공유하게 됨

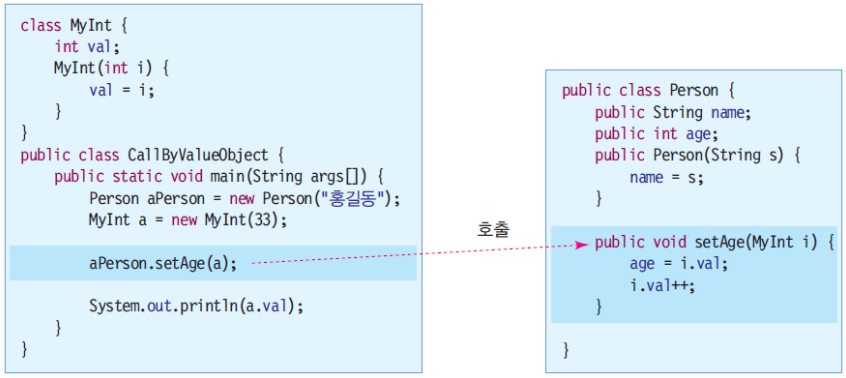


기본 타입의 값이 전달되는 경우



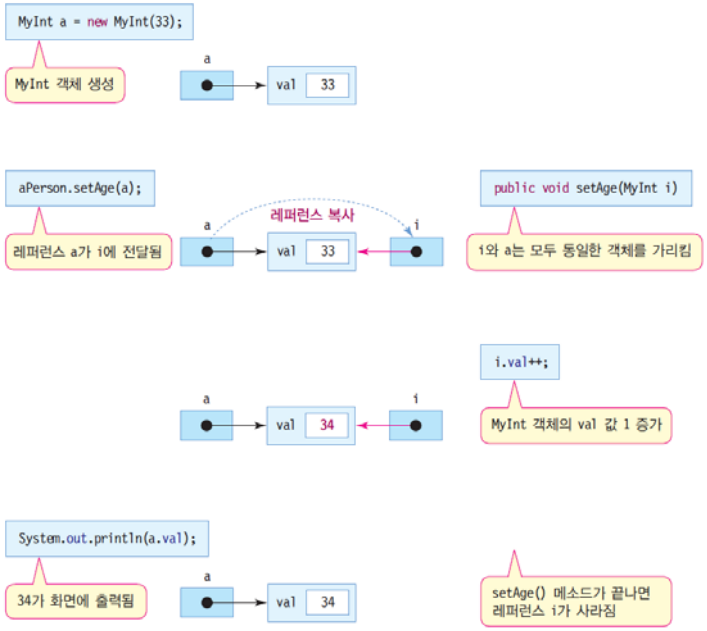
33

객체가 전달되는 경우



34

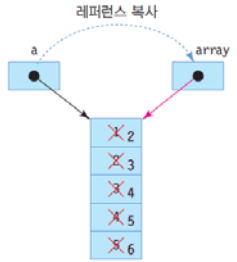
* 객체가 복사되어 전달되는 것이 아님
객체에 대한 레퍼런스만 복사되어 전달



매개 변수에 배열이 전달되는 경우

- 매개 변수에 배열의 레퍼런스만 전달

```
public class ArrayParameter {

    public static void main(String args[]) {
        int a[] = {1, 2, 3, 4, 5};

        increase(a);

        for(int i=0; i<a.length; i++)
            System.out.print(a[i]+" ");
    }
```



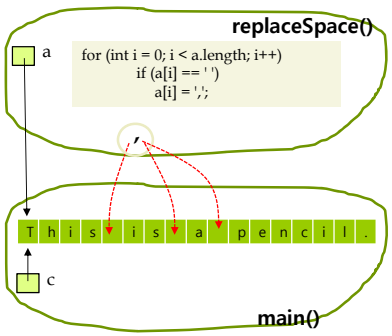

```
    static void increase(int[] array) {
        for(int i=0; i<array.length; i++) {
            array[i]++;
        }
    }
}
```

2 3 4 5 6

예제: 배열의 전달

```
public class ArrayParameter {
  static void replaceSpace(char[] a) {
    for (int i = 0; i < a.length; i++)
      if (a[i] == ' ')
        a[i] = ',';
  }
  static void printCharArray(char[] a) {
    for (int i = 0; i < a.length; i++)
      System.out.print(a[i]);
    System.out.println();
  }
  public static void main (String[] args) {
    char c[] = {'T','h','i','s',' ','i','s',' ','a',' '
','p','e','n','c','i','l','.'};
    printCharArray(c);
    replaceSpace(c);
    printCharArray(c);
  }
}
```

char 배열을 메소드의 인자로 전달하여 배열 속의 공백(' ')문자를 ',' 로 대치하는 프로그램을 작성하라.



This is a pencil.
This,is,a,pencil.

메소드 오버로딩

- 오버로딩(Overloading)
  - 한 클래스 내에서 두 개 이상의 이름이 같은 메소드 작성
    - 메소드 이름이 동일하여야 함
    - 매개 변수의 개수가 서로 다르거나, 타입이 서로 달라야 함
    - 리턴 타입은 오버로딩과 관련 없음

```
// 메소드 오버로딩이 성공한 사례
class MethodOverloading {
  public int getSum(int i, int j) {
    return i + j;
  }
  public int getSum(int i, int j, int k) {
    return i + j + k;
  }
  public double getSum(double i, double j) {
    return i + j;
  }
}
```

```
// 메소드 오버로딩이 실패한 사례
class MethodOverloadingFail {
  public int getSum(int i, int j) {
    return i + j;
  }
  public double getSum(int i, int j) {
    return (double)(i + j);
  }
}
```

오버로딩된 메소드 호출

```
public static void main(String args[]) {
    MethodSample a = new MethodSample();

    int i = a.getSum(1, 2);

    int j = a.getSum(1, 2, 3);

    double k = a.getSum(1.1, 2.2);

}
```

```
public class MethodSample {
    public int getSum(int i, int j) {
        return i + j;
    }

    public int getSum(int i, int j, int k) {
        return i + j + k;
    }

    public double getSum(double i, double j) {
        return i + j;
    }
}
```

this 레퍼런스

- this 란?
  - 현재 실행되는 메소드가 속한 객체에 대한 레퍼런스
    - 컴파일러에 의해 자동 선언 : 별도로 선언할 필요 없음

```
class Samp {
  int id;
  public Samp(int x) { id = x; }
  public void set(int x) { id = x; }
  public int get() {return id; }
}
```

=

```
class Samp {
  int id;
  public Samp(int x) { this.id = x; }
  public void set(int x) { this.id = x; }
  public int get() {return id; }
}
```

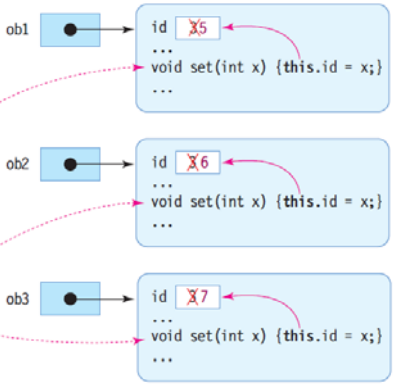
this가 필요한 경우

- this의 필요성
  - 객체의 멤버 변수와 메소드 변수의 이름이 같은 경우
  - 다른 메소드 호출 시 객체 자신의 레퍼런스를 전달할 때
  - 메소드가 객체 자신의 레퍼런스를 반환할 때

```
class Samp {
  int id;
  // 매개 변수 이름과 필드의 이름이 같을 때
  public Samp(int id) { this.id = id; }
  public void set(int id) { this.id = id; }
  public int get() {return this.id; }

  public Samp me() {
    return this; // 자신의 레퍼런스를 반환할 때
  }
}
```

this에 대한 이해

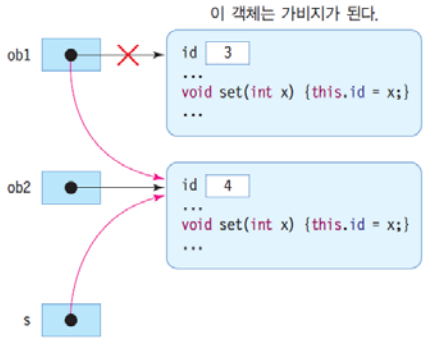
```
public class Samp {
    int id;
    public Samp(int x) {this.id = x;}
    public void set(int x) {this.id = x;}
    public int get() {return this.id;}

    public static void main(String [] args) {
        Samp ob1 = new Samp(3);
        Samp ob2 = new Samp(3);
        Samp ob3 = new Samp(3);

        ob1.set(5);
        ob2.set(6);
        ob3.set(7);
    }
}
```



객체의 치환

* 객체의 치환은 객체가 복사되는 것이 아니며 레퍼런스가 복사된다.

```
public class Samp {
    int id;
    public Samp(int x) {this.id = x;}
    public void set(int x) {this.id = x;}
    public int get() {return this.id;}

    public static void main(String [] args) {
        Samp ob1 = new Samp(3);
        Samp ob2 = new Samp(4);
        Samp s;

        s = ob2;
        ob1 = ob2; // 객체의 치환
        System.out.println("ob1.id="+ob1.id);
        System.out.println("ob2.id="+ob2.id);
    }
}
```




```
ob1.id=4
ob2.id=4
```

필드의 초기화

- 선언과 동시에 초기화 가능

```
public class ClassRoom {
    public static int capacity = 60;        // 60으로 초기화
    private boolean use = false;            // false로 초기화
}
```

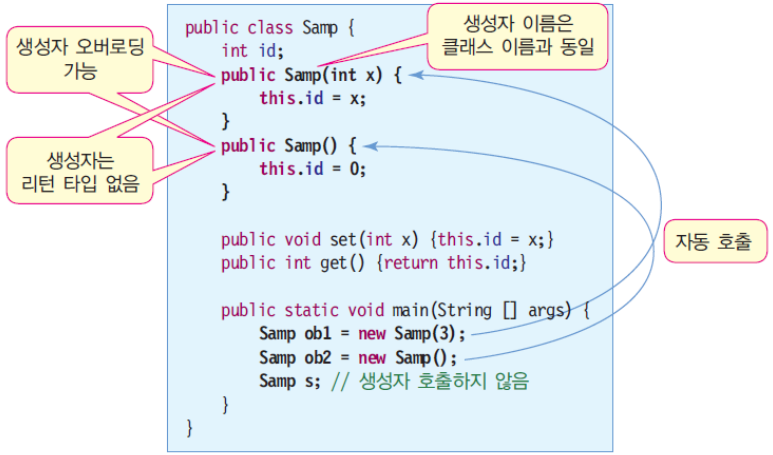
생성자 개념

- 생성자 - 객체가 생성될 때 초기화를 위해 실행되는 메소드

생성자

- 생성자의 특징
  - 생성자는 메소드
  - 생성자 이름은 클래스 이름과 동일
  - 생성자는 new를 통해 객체를 생성할 때만 호출됨
  - 생성자도 오버로딩하여 여러개 작성 가능
  - 생성자는 리턴 타입을 지정할 수 없음
  - 생성자는 하나 이상 선언되어야 함
    - 개발자가 생성자를 작성하지 않았으면 컴파일러에 의해 자동으로 기본 생성자가 선언됨
    - 기본 생성자를 디폴트 생성자(default constructor)라고도 함

생성자 정의와 생성자 호출




```
public class Samp {
    int id;
    public Samp(int x) {
        this.id = x;
    }
    public Samp() {
        this.id = 0;
    }

    public void set(int x) {this.id = x;}
    public int get() {return this.id;}

    public static void main(String [] args) {
        Samp ob1 = new Samp(3);
        Samp ob2 = new Samp();
        Samp s; // 생성자 호출하지 않음
    }
}
```

기본 생성자

- 기본 생성자(default constructor)
  - 클래스에 생성자가 하나도 선언되지 않은 경우, 컴파일러에 의해 자동으로 생성
    - 매개 변수 없는 생성자
    - 아무 작업 없이 단순 리턴
  - 디폴트 생성자라고도 부름

```
class DefaultConstructor{
  int x;
  public  void setX(int x) {this.x = x;}
  public  int getX() {return x;}

  public static void main(String [] args) {
    DefaultConstructor p = new DefaultConstructor();
    p.setX(3);
  }
}
```

개발자가 작성한 코드

기본 생성자

```
class DefaultConstructor{
  int x;
  public  void setX(int x) {this.x = x;}
  public  int getX() {return x;}

  public DefaultConstructor() { }

  public static void main(String [] args) {
    DefaultConstructor p= new DefaultConstructor();
    p.setX(3);
  }
}
```

컴파일러에 의해 자동 삽입된 기본 생성자

컴파일러가 자동으로 기본 생성자를 삽입한 코드

기본 생성자가 자동 생성되지 않는 경우

- 클래스에 생성자가 하나라도 존재하면 기본 생성자가 자동 삽입되지 않음

```
class DefaultConstructor{
  int x;
  public  void setX(int x) {this.x = x;}
  public  int getX() {return x;}

  public DefaultConstructor(int x) {
    this.x = x;
  }
  public static void main(String [] args) {
    DefaultConstructor p1= new DefaultConstructor(3);
    int n = p1.getX();
    DefaultConstructor p2= new DefaultConstructor();
    p2.setX(5);
  }
}
```

컴파일러가 기본 생성자를 자동 생성하지 않음

public DefaultConstructor() { }

컴파일 오류. 해당하는 생성자가 없음 !!!

this(), 생성자에서 다른 생성자 호출

- this() 생성자 호출
  - 같은 클래스의 다른 생성자 호출
  - 생성자 내에서만 사용 가능
    - 다른 메소드에서는 사용 불가
  - 반드시 생성자 코드의 제일 처음에 수행

```
public class Book {
  String title;
  String author;
  int ISBN;

  public Book(String title, String author, int ISBN) {
    this.title = title;
    this.author = author;
    this.ISBN = ISBN;
  }
  public Book(String title, int ISBN) {
    this(title, "Anonymous", ISBN);
  }
  public Book() {
    this(null, null, 0);
    System.out.println("생성자가 호출되었음");
  }
  public static void main(String [] args) {
    Book javaBook = new Book("Java JDK", "황기태",
3333);
    Book holyBible = new Book("Holy Bible", 1);
    Book emptyBook = new Book();
  }
}
```

title = "Holy Bible"
author = "Anonymous"
ISBN = 1

title = "Holy Bible"
ISBN = 1

this() 사용 실패 예

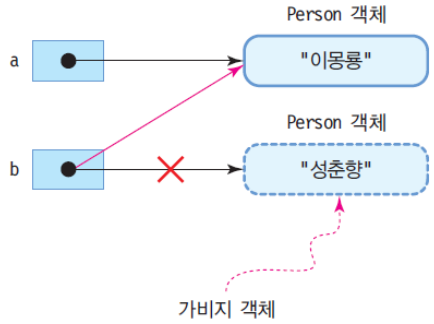
```
public class Book {
  String title;
  String author;
  int ISBN;
  public Book(String title, String author, int ISBN) {
    this.title = title;
    this.author = author;
    this.ISBN = ISBN;
  }
  public Book() {
    System.out.println("생성자가 호출되었음");
    //this(null, null, 0); // 생성자의 첫 번째 문장이 아니기 때문에 컴파일 오류
  }

  public static void main(String [] args) {
    Book javaBook = new Book("Java JDK", "황기태", 3333);
  }
}
```

객체의 소멸과 가비지

- 객체 소멸
  - new에 의해 생성된 객체 메모리를 자바 가상 기계로 되돌려 주는 행위
  - 소멸된 객체 공간은 가용 메모리에 포함
- 자바 응용프로그램에서 임의로 객체 소멸할 수 없음
  - 객체 소멸은 자바 가상 기계의 고유한 역할
  - 자바 개발자에게는 매우 다행스러운 기능
    - C/C++에서는 할당 받은 객체를 개발자가 프로그램 내에서 삭제해야 함
    - C/C++의 프로그램 작성을 어렵게 만드는 요인
- 가비지(Garbage)
  - 가비지(Gabage) :
    - 가리키는 레퍼런스가 하나도 없는 객체
    - 더 이상 접근하여 사용할 수 없게 되었음
  - 가비지 켈렉션(Gabage Collection)
    - 자바 가상 기계의 가비지 컬렉터가 자동으로 가비지를 수집하여 반환

가비지 사례

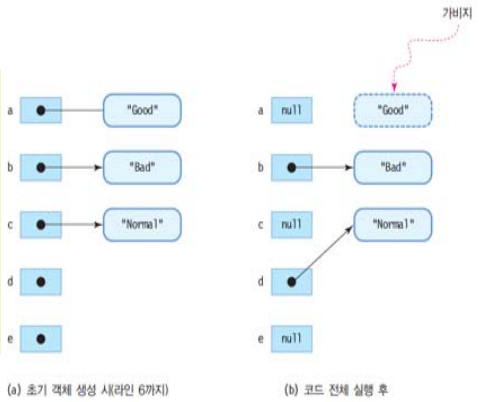
```
Person a, b;
a = new Person("이몽룡");
b = new Person("성춘향");
b = a; // b가 가리키던 객체는 가비지가 됨
```



예제 : 가비지 발생

다음 소스에서 언제 가비지가 발생하는지 설명하라.

```
public class GarbageEx {
  public static void main(String[] args) {
    String a = new String("Good");
    String b = new String("Bad");
    String c = new String("Normal");
    String d, e;
    a = null;
    d = c;
    c = null;
  }
}
```



(a) 초기 객체 생성 시(라인 6까지) (b) 코드 전체 실행 후

가비지 컬렉션

- 가비지 컬렉션
  - 자바에서 가비지 자동 회수
    - 가용 메모리 공간으로 확보
  - 가비지 컬렉터(garbage collector)에 의해 자동 수행
- 개발자에 의한 강제 가비지 컬렉션
  - System 또는 Runtime 객체의 gc() 메소드 호출

```
System.gc(); // 가비지 컬렉션 작동 요청
```

- 이 코드는 자바 가상 기계에 강력한 가비지 컬렉션 요청
  - 그러나 자바 가상 기계가 가비지 컬렉션 시점을 전적으로 판단

설정자와 접근자

- 설정자(mutator)
  - 필드의 값을 설정하는 메소드
  - setXXX() 형식
- 접근자(accessor)
  - 필드의 값을 반환하는 메소드
  - getXXX() 형식

접근자와 변경자 메소드만을 통하여 필드에 접근한다.

설정자와 접근자

CarTest1.java

```
01  class Car {
02      private String color;          // 색상
03      private int speed;             // 속도
04      private int gear;              // 기어
05      public String getColor() {
06          return color;
07      }
08      public void setColor(String c) {
09          color = c;
10      }
11      public int getSpeed()          {       return speed;   }
12      public void setSpeed(int s)    {       speed = s;      }
13      public int getGear()           {       return gear;    }
14      public void setGear(int g)     {       gear = g;       }
15  }
```

필드가 모두 private로 선언되었다.
클래스 내부에서만 사용이 가능하다.

color에 대한 접근자 메소드

color에 대한 설정자 메소드

이런 식으로 간략하게 표기하기도 한다.

예제: 설정자(set)와 접근자(get)

```
16  public class CarTest1 {
17     public static void main(String[] args) {
18        // 객체 생성
19        Car myCar = new Car();
20
21        myCar.setColor("red");
22        myCar.setSpeed(100);
23        myCar.setGear(1);
24
25        System.out.println("현재 자동차의 색상은 " + myCar.getColor());
26        System.out.println("현재 자동차의 속도는 " + myCar.getSpeed());
27        System.out.println("현재 자동차의 기어는 " + myCar.getGear());
28     }
29  }
```

객체 생성

설정자 메소드 호출

설정자와 접근자는 왜 사용하는가?

- 설정자에서 매개 변수를 통하여 잘못된 값이 넘어오는 경우, 이를 사전에 차단할 수 있음.
- 필요할 때마다 필드값을 계산하여 반환할 수 있음.
- 접근자만을 제공하면 자동적으로 읽기만 가능한 필드를 만들 수 있음.

```
public void setSpeed(int s)
{
    if( s < 0 )
        speed = 0;
    else
        speed = s;
}
```

속도가 음수이면 0으로 만든다.

지역 변수

- 메소드 안에 선언
- 메소드의 매개 변수도 지역 변수의 일종

```
class Box {
    int width=0, length=0, height=0;
    ...
    public int getVolume()
    {
        int volume;
        volume = width*length*height;
        return volume;
    }
}
```

필드: 전체 클래스 안에서 사용가능

지역 변수

지역 변수 volume의 사용 범위

주의

- 지역 변수를 초기화하지 않고 사용하면 오류

```
class BugClass {
    public int getSum() {
        int sum;
        for (int i = 0; i < 10; i++)
            sum += i;
        return sum;
    }
}
```

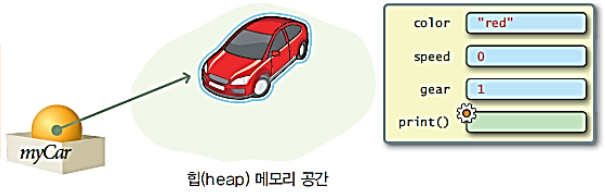
초기화되지 않은 지역변수를 사용하면 오류 발생

실행결과

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
The local variable sum may not have been initialized
```

객체의 생성

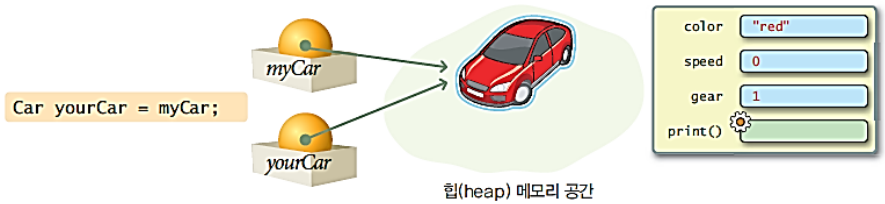
```
Car myCar = new Car();
myCar.color = "red";
myCar.speed = 0;
mycar.gear = 1;
```



참조값을 복사한다면

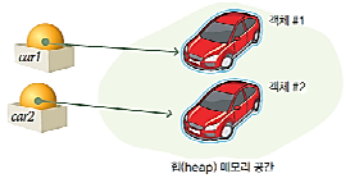
- 두 개의 참조 변수가 하나의 객체를 가리킬 수 있음.

```
Car yourCar = myCar;
```

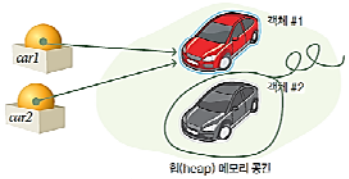


객체의 소멸

```
Car car1 = new Car();    // 첫 번째 객체
Car car2 = new Car();    // 두 번째 객체
```



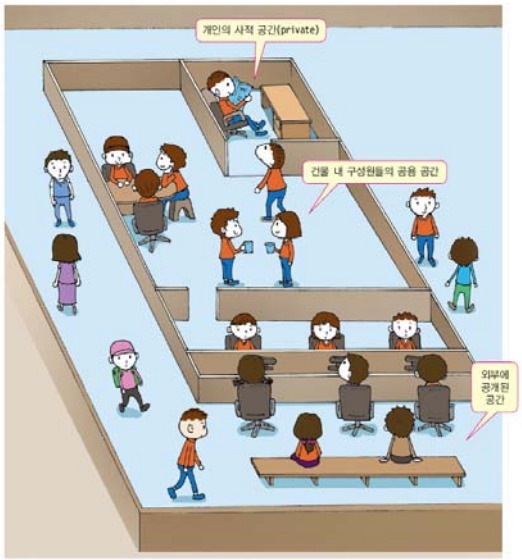

```
car2 = car1;    // car1과 car2는 같은 객체를 가리킨다.
                // car2가 가리켰던 객체는 쓰레기 수집기에 의하여 수거된다.
```



객체의 소멸

```
car1 = null;    // 객체 1은 아직도 활성화된 참조가 있기 때문에 소멸되지 않는다.
```

접근 지정자 이해



클래스 접근 지정자

- 클래스 앞에 올 수 있는 접근 지정자
  - public 접근 지정자

```
public class Person {}
```

    - 다른 모든 클래스가 접근 가능
  - 접근 지정자 생략 (**default** 접근 지정자)

```
class Person {}
```

    - **package-private**라고도 함
    - **같은 패키지 내에 있는 클래스**에서만 접근 가능
      - 같은 디렉토리에 있는 클래스끼리 접근 가능

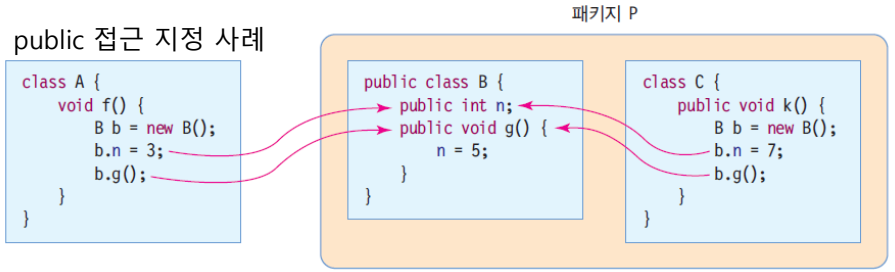
멤버 접근 지정자

- 디폴트(default) 멤버
  - 같은 패키지 내의 다른 클래스만 접근 가능
- public 멤버
  - 패키지에 관계 없이 모든 클래스에서 접근 가능
- private 멤버
  - 클래스 내에서만 접근 가능
  - 상속 받은 하위 클래스에서도 접근 불가
- protected 멤버
  - 같은 패키지 내의 다른 모든 클래스에서 접근 가능
  - 상속 받은 하위 클래스는 다른 패키지에 있어도 접근 가능

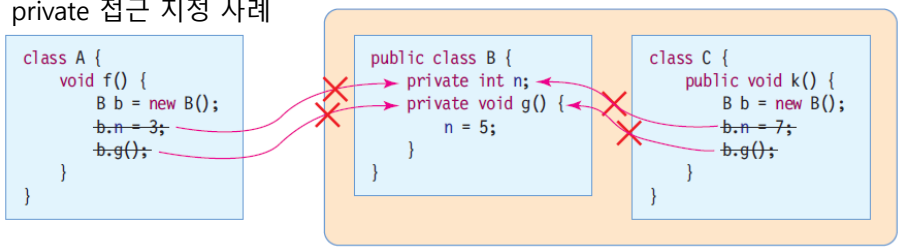
멤버에 접근하는 클래스	멤버의 접근 지정자			
	default	private	protected	public
같은 패키지의 클래스	○	×	○	○
다른 패키지의 클래스	×	×	×	○

멤버 접근 지정자의 이해

public 접근 지정 사례

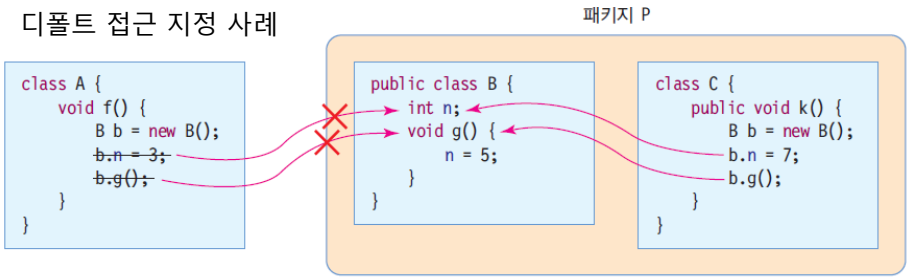


private 접근 지정 사례



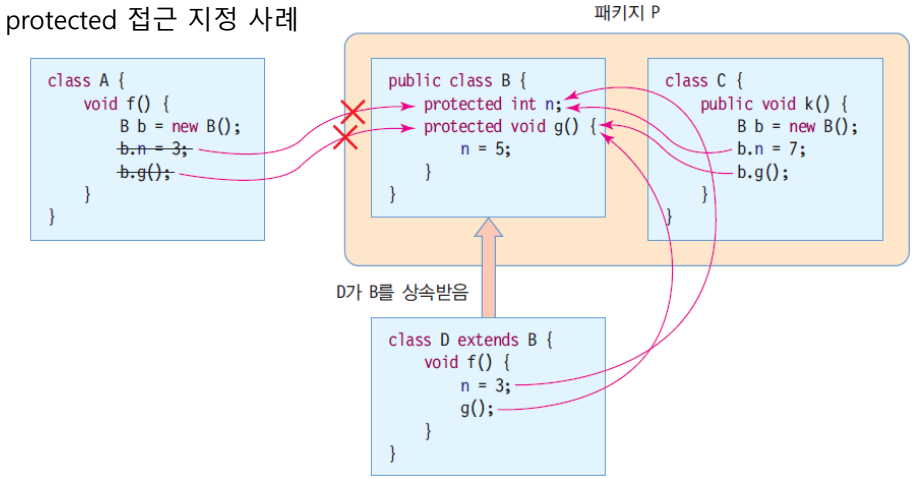
멤버 접근 지정자의 이해

디폴트 접근 지정 사례



멤버 접근 지정자의 이해

protected 접근 지정 사례



예제: 접근 지정자의 사용

다음의 소스를 컴파일 해보고 오류가 난 이유를 설명하고 오류를 수정하시오.

```
class Sample {
  public int a;
  private int b;
  int c;
}
public class AccessEx {
  public static void main(String[] args) {
    Sample aClass = new Sample();
    aClass.a = 10;
    //aClass.b = 10;
    aClass.c = 10;
  }
}
```

- Sample 클래스의 a와 c는 각각 public, default 지정자로 선언이 되었으므로, 같은 패키지에 속한 AccessEx 클래스에서 접근 가능
- b는 private으로 선언이 되었으므로 AccessEx 클래스에서 접근 불가능

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        The field Sample.b is not visible

        at AccessEx.main(AccessEx.java:11)
```

예제: 결과

오류가 수정된 소스

```
class Sample {
  public int a;
  private int b;
  int c;
  public int getB() {
    return b;
  }
  public void setB(int value) {   b = value;   }
}
public class AccessEx {
  public static void main(String[] args) {
    Sample aClass = new Sample();
    aClass.a = 10;
    aClass.setB(10);
    aClass.c = 10;
  }
}
```

- private 접근 지정자를 갖는 멤버 b를 위해 클래스 내부에 getB()/setB() 메소드 만들어 접근

static 멤버와 non-static 멤버

- non-static 멤버의 특성
  - 공간적 - 멤버들은 객체마다 독립적으로 별도 존재
    - 인스턴스 멤버라고도 부름
  - 시간적 - 필드와 메소드는 객체 생성 후 비로소 사용 가능
  - 비공유의 특성 - 멤버들은 여러 객체에 의해 공유되지 않고 배타적
- static 멤버란?
  - 객체를 생성하지 않고 사용가능
  - 클래스당 하나만 생성됨
    - 클래스 멤버라고도 부름
    - 객체마다 생기는 것이 아님
  - 특성
    - 공간적 특성 - static 멤버들은 클래스 당 하나만 생성.
    - 시간적 특성 - static 멤버들은 클래스가 로딩될 때 공간 할당.
    - 공유의 특성 - static 멤버들은 동일한 클래스의 모든 객체에 의해 공유

```
class StaticSample {
  int n; // non-static 필드
  void g() {...} // non-static 메소드

  static int m; // static 필드
  static void f() {...} // static 메소드
}
```

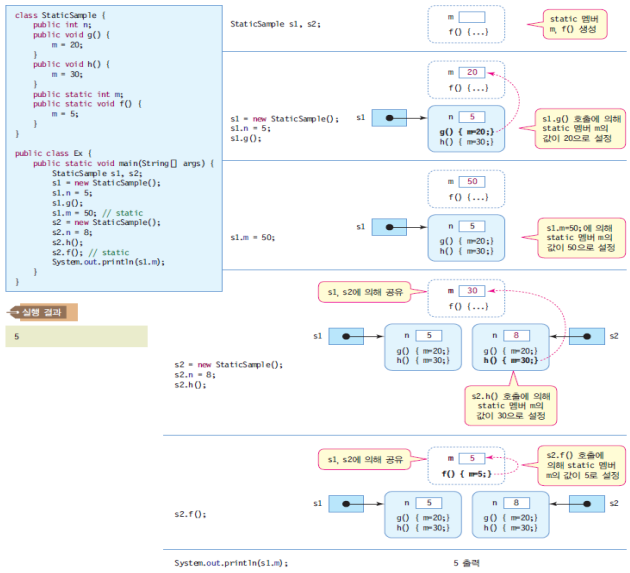
non-static 멤버와 static 멤버의 차이

	non-static 멤버	static 멤버

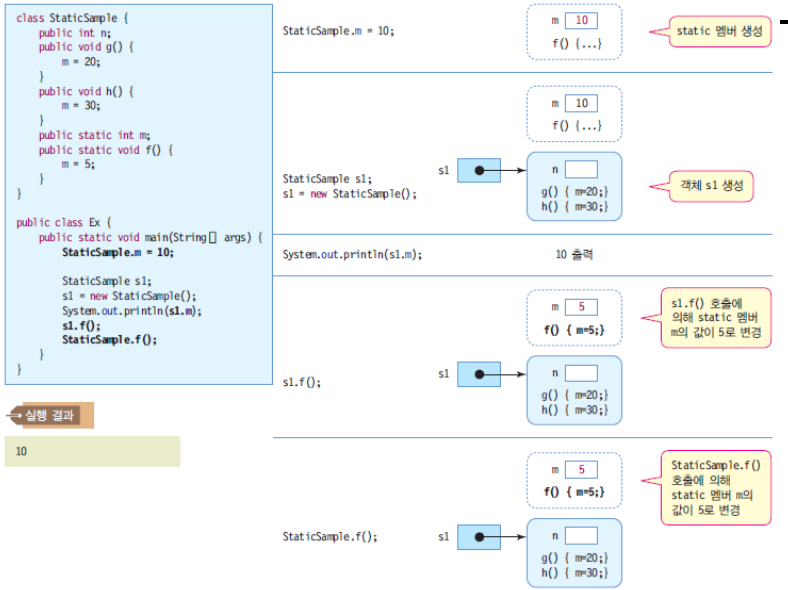
선언	class Sample {
	int n;
	void g() {...}
	}
		class Sample {
		static int m;
		static void g() {...}
		}

공간적 특성	멤버는 객체마다 별도 존재 • 인스턴스 멤버라고 부름	멤버는 클래스당 하나 생성 • 멤버는 객체 내부가 아닌 별도의 공간에 생성 • 클래스 멤버라고 부름
시간적 특성	객체 생성 시에 멤버 생성됨 • 객체가 생길 때 멤버도 생성 • 객체 생성 후 멤버 사용 가능 • 객체가 사라지면 멤버도 사라짐	클래스 로딩 시에 멤버 생성 • 객체가 생기기 전에 이미 생성 • 객체가 생기기 전에도 사용 가능 • 객체가 사라져도 멤버는 사라지지 않음 • 멤버는 프로그램이 종료될 때 사라짐
공유의 특성	공유되지 않음 • 멤버는 객체 내에 각각 공간 유지	동일한 클래스의 모든 객체들에 의해 공유됨

static 멤버를 객체의 멤버로 접근하는 사례



static 멤버를 클래스 이름으로 접근하는 사례



static의 활용

- 전역 변수와 전역 함수를 만들 때 활용
  - 자바의 캡슐화 원칙 지킴
    - 다른 클래스에서 공유하는 전역 변수나 전역 함수도 반드시 클래스 내부에 구현해야 함
- static 멤버를 가진 클래스 사례
  - java.lang.Math 클래스
    - JDK와 함께 배포되는 java.lang.Math 클래스
    - 모든 필드와 메소드가 public static으로 선언
    - 다른 모든 클래스에서 사용할 수 있음

```
public class Math {
   public static int abs(int a);
   public static double cos(double a);
   public static int max(int a, int b);
   public static double random();
   ...
}
```

// 잘못된 사용법

```
//Math() 생성자는 private
//Math m = new Math()
//int n = m.abs(-5)
```

// 바른 사용법

```
int n = Math.abs(-5);
```

static 메소드의 제약 조건 1

- static 메소드는 오직 static 멤버만 접근 가능
  - 객체가 생성되지 않은 상황에서도 static 메소드는 실행될 수 있기 때문에, non-static 메소드와 필드 사용 불가
  - non-static 메소드는 static 멤버 사용 가능

```
class StaticMethod {
   int n;
   void f1(int x) {n = x;} // 정상
   void f2(int x) {m = x;} // 정상

   static int m;
   static void s1(int x) {n = x;} // 컴파일 오류. static 메소드는 non-static 필드 사용 불가
   static void s2(int x) {f1(3);} // 컴파일 오류. static 메소드는 non-static 메소드 사용 불가

   static void s3(int x) {m = x;} // 정상. static 메소드는 static 필드 사용 가능
   static void s4(int x) {s3(3);} // 정상. static 메소드는 static 메소드 호출 가능
}
```

오류

static 메소드의 제약 조건 2

- static 메소드는 this 사용불가
  - static 메소드는 객체가 생성되지 않은 상황에서도 호출이 가능하므로, 현재 객체를 가리키는 this 레퍼런스 사용할 수 없음

```
class StaticAndThis {
   int n;
   static int m;
   void f1(int x) {this.n = x;} // 정상
   void f2(int x) {this.m = x;} // non-static 메소드에서는 static 멤버 접근 가능
   static void s1(int x) {this.n = x;} // 컴파일 오류. static 메소드는 this 사용 불가
}
```

오류

예제: static을 이용한 달러와 우리나라 원화 사이의 변환 예제

```
class CurrencyConverter {
  private static double rate; //  한국 원화에 대한 환율
  public static double toDollar(double won) {
    return won/rate; // 한국 원화를 달러로 변환
  }
  public static double toKWR(double dollar) {
    return dollar * rate; // 달러를 한국 원화로 변환
  }
  public static void setRate(double r) {
    rate = r; // 환율 설정. KWR/$1
  }
}
public class StaticMember {
  public static void main(String[] args) {
    CurrencyConverter.setRate(1121); // 미국 달러 환율 설정
    System.out.println("백만원은 "
        + CurrencyConverter.toDollar(1000000) + "달러입니다.");
    System.out.println("백달러는 "
                + CurrencyConverter.toKWR(100) +"원입니다.");
  }
}
```

static 필드와 메소드를 이용하여 달러와 한국 원화 사이의 변환을 해주는 환율 계산기를 만들어 보자.

```
백만원은 892.0606601248885달러입니다.
백달러는 112100.0원입니다.
```

final 클래스와 메소드

- final 클래스 - 더 이상 클래스 상속 불가능

```
final class FinalClass {
    .....
}
class DerivedClass extends FinalClass { // 컴파일 오류
    .....
}
```

- final 메소드 - 더 이상 오버라이딩 불가능

```
public class SuperClass {
    protected final int finalMethod() { ... }
}
class DerivedClass extends SuperClass {
    protected int finalMethod() { ... } // 컴파일 오류, 오버라이딩 할 수 없음
}
```

final 필드

- final 필드, 상수 선언
  - 상수를 선언할 때 사용

```
class SharedClass {
    public static final double PI = 3.141592653589793;
}
```

  - 상수 필드는 선언 시에 초기 값을 지정하여야 한다
  - 상수 필드는 실행 중에 값을 변경할 수 없다
  - 상수는 static으로 선언하는 것이 바람직함

```
public class FinalFieldClass {
    final int ROWS = 10; // 상수 정의, 이때 초기 값(10)을 반드시 설정
    void f() {
        int [] intArray = new int [ROWS]; // 상수 활용
        //ROWS = 30; // 컴파일 오류 발생, final 필드 값을 변경할 수 없다.
    }
}
```

String 클래스의 메소드

char	charAt(int index) 지정된 인덱스에 있는 문자를 반환한다.
int	compareTo(String anotherString) 사전적 순서로 문자열을 비교한다.앞에 있으면 -1, 같으면 0, 뒤에 있으면 1이 반환된다.
String	concat(String str) 주어진 문자열을 현재의 문자열 뒤에 붙인다.
boolean	equals(Object anObject) 주어진 객체와 현재의 문자열을 비교한다.
boolean	equalsIgnoreCase(String anotherString) 대소문자를 무시하고 비교한다.
boolean	isEmpty() length()가 0이면 true를 반환한다.
int	length() 현재 문자열의 길이를 반환한다.
String	replace(char oldChar, char newChar) 주어진 문자열에서 oldChar를 newChar로 변경한, 새로운 문자열을 생성하여 반환한다.
String	substring(int beginIndex, int endIndex) 현재 문자열의 일부를 반환한다.
String	toLowerCase() 문자열의 문자들을 모두 소문자로 변경한다.
String	toUpperCase() 문자열의 문자들을 모두 대문자로 변경한다.

문자열의 결합

- 두 개의 문자열은 + 연산자를 이용하여 결합될 수 있다.
- String subject = "Money";
- String other = " has no value if it is not used";
- String sentence = subject + other;

숫자를 문자열로 변환

- **int** x = 20;
- System.out.println("결과값은 " + x); // "결과값은 20" 이 출력된다.
- String answer = "The answer is " + 100; // "The answer is 100"