

# 클래스 상속과 다형성

# 클래스의 상속

# 클래스의 상속

4

## TIP

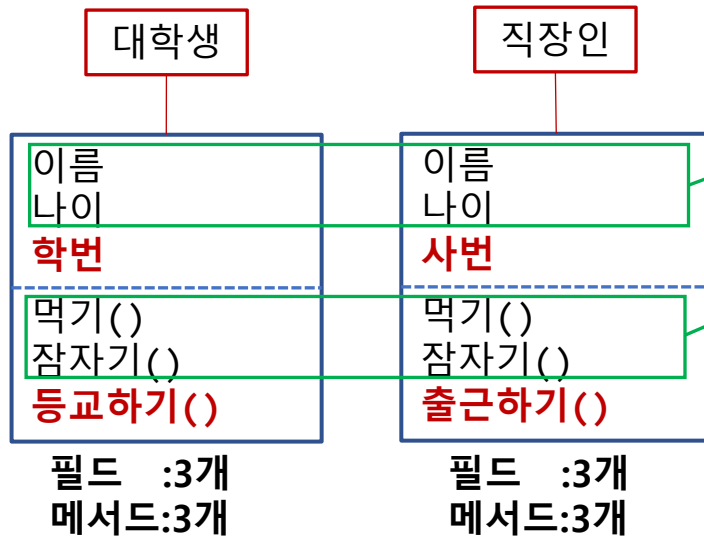
- 상속 다이어그램을 표기할 때 부모 클래스 쪽으로 화살표가 있음
- UML(Unified Modeling language):  
시스템을 모델로 표현해주는 대표적인 모델링 언어

## 1 상속의 개념

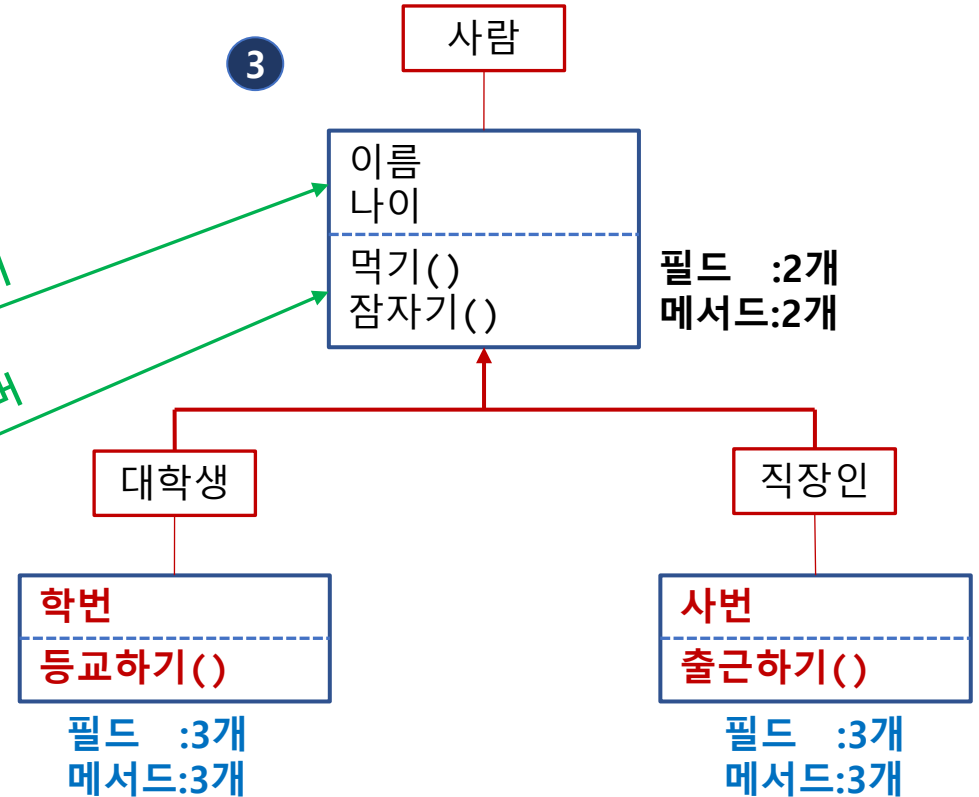
- 부모클래스의 멤버(필드, 메서드, inner클래스)를  
자식 클래스가 내려받아(상속) 클래스 내부에 포함

## 대학생 클래스와 직장인 클래스

2



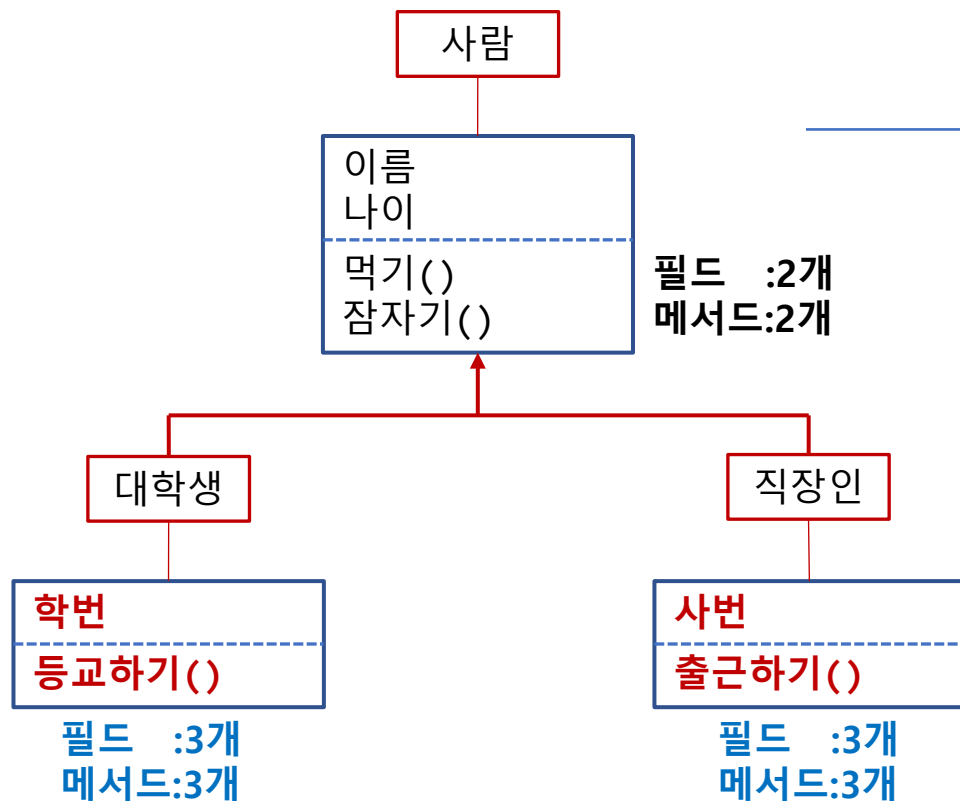
3



# 클래스의 상속

## ☞ 상속의 장점

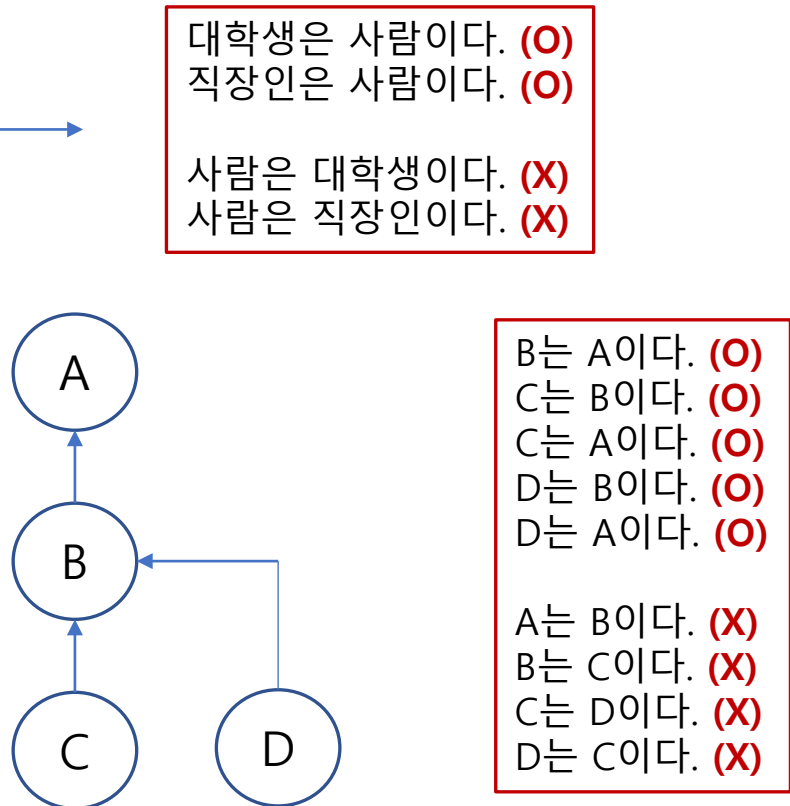
### 1 장점1. 코드의 중복성 제거



### TIP

- 상속 다이어그램을 표기할 때 부모 클래스 쪽으로 화살표가 있음
- UML(Unified Modeling language):  
시스템을 모델로 표현해주는 대표적인 모델링 언어

### 2 장점2. 다형적 표현 가능 (가장 중요한 장점)



# 클래스의 상속

## ☞ 상속의 장점

다형적 표현으로 얻을 수 있는 장점

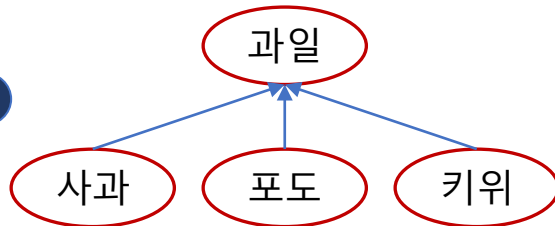
1



```
사과[] apple = {new 사과(), new 사과()};  
포도[] grape = {new 포도(), new 포도()};  
키위[] kiwi = {new 키위(), new 키위(), new 키위()};
```



2



```
사과는 과일이다. (과일 fruit = new 사과())  
포도는 과일이다. (과일 fruit = new 포도())  
키위는 과일이다. (과일 fruit = new 키위())
```

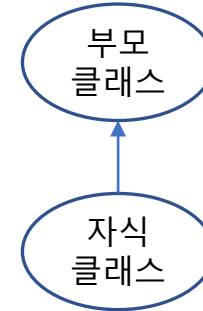
```
과일[] fruits = {new 사과(), new 사과(), new 포도(), new 포도(), new 키위(), new 키위(), new 키위()};
```

# 클래스의 상속

## ☞ 상속 문법

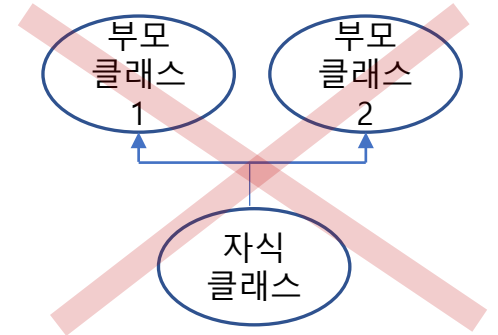
- 1 - **extends** 키워드 사용

```
class 자식클래스 extends 부모클래스 {  
    ...  
}
```

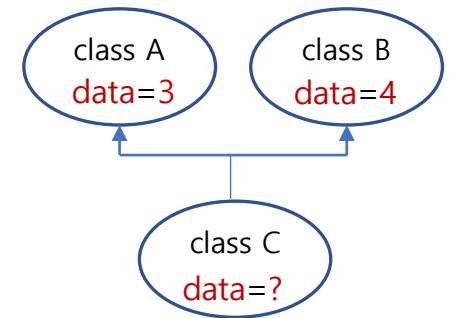
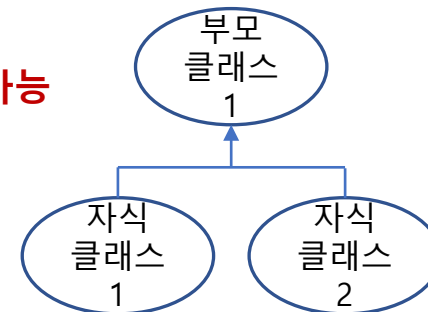


- 2 - **다중 상속 불가**

```
class 자식클래스 extends 부모클래스1, 부모클래스2 {  
    ...  
}
```

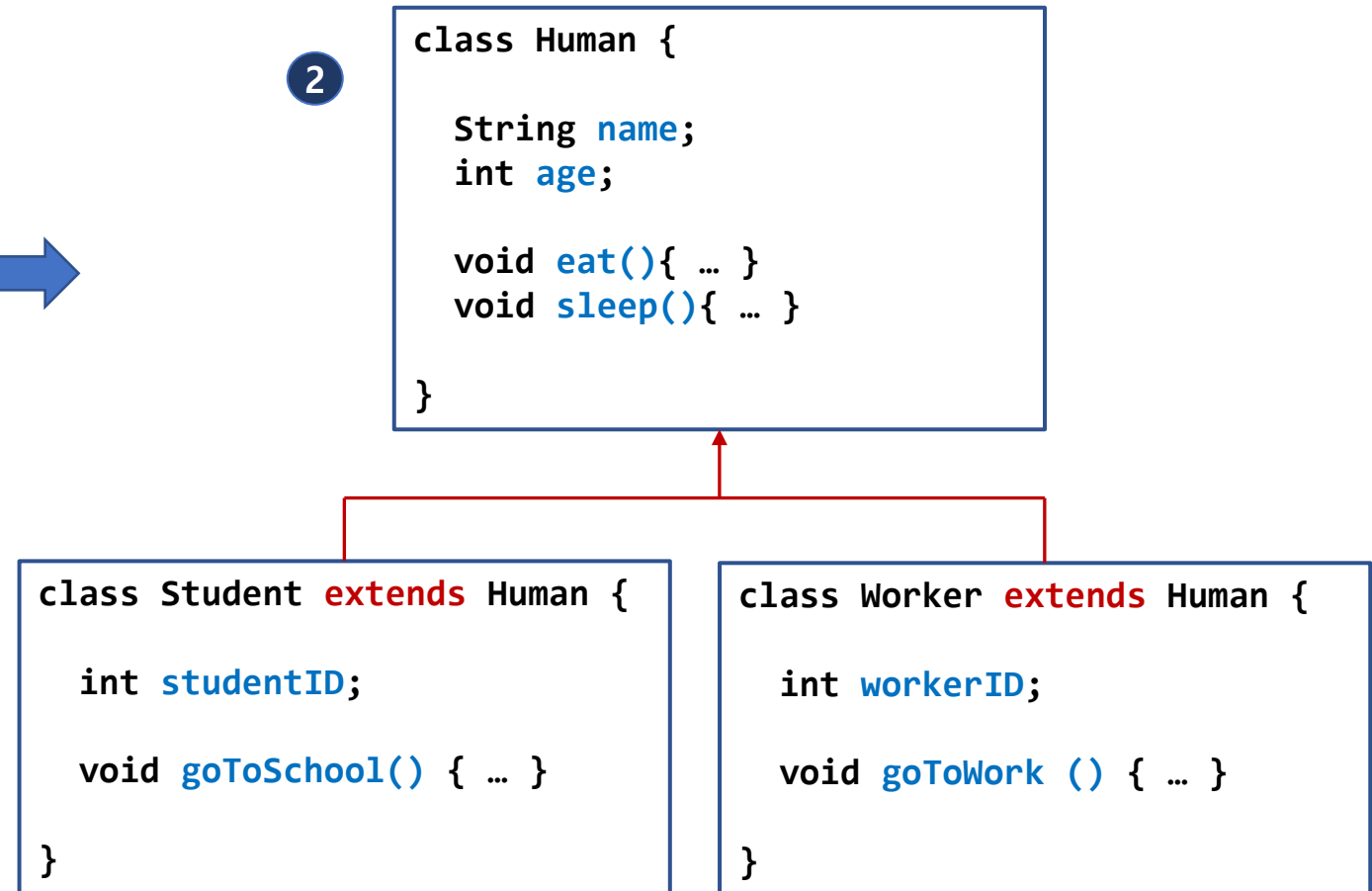
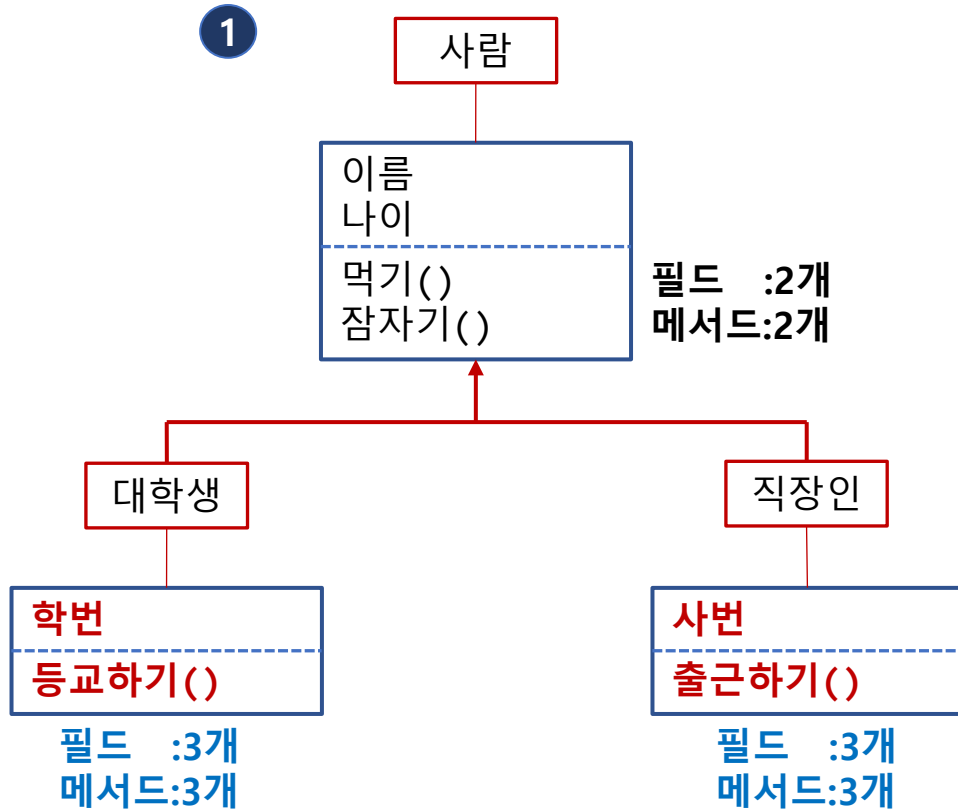


- 3 **cf) 부모클래스가 둘이 될 수는 없지만 자식클래스는 여러 개 가능**



# 클래스의 상속

☞ 상속 문법



# 클래스의 상속

☞ 상속 문법

1

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

```
class Student extends Human {  
  
    int studentID;  
  
    void goToSchool() { ... }  
  
}
```

```
class Worker extends Human {  
  
    int workerID;  
  
    void goToWork () { ... }  
  
}
```

2

```
public static void main(String[] ar) {  
  
    // #1. Human 객체 (필드2개, 메서드2개)  
    Human h = new Human();  
    h.name="홍길동";  
    h.age=16;  
    h.eat();  
    h.sleep();  
  
}
```

3

```
    // #2. Student 객체 (필드 2+1개, 메서드 2+1개)  
    Student s = new Student();  
    s.name="김민성";  
    s.age=15;  
    s.studentID=128;  
    s.eat();  
    s.sleep();  
    s.goToSchool();  
  
}
```

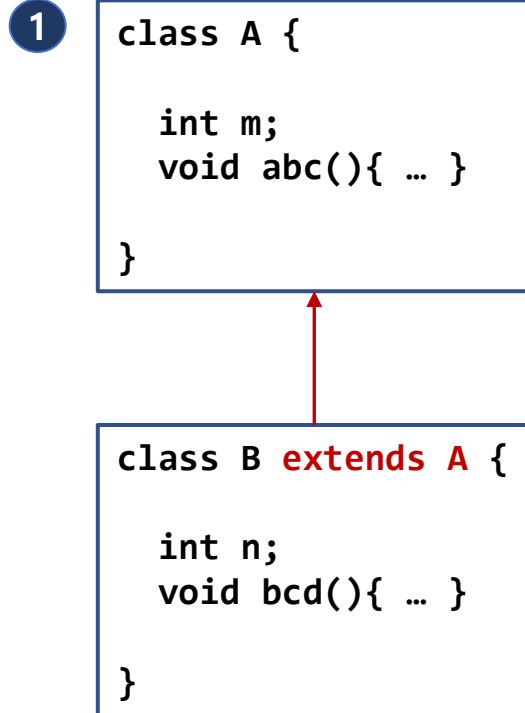
4

```
    // #3. Worker 객체 (필드 2+1개, 메서드 2+1개)  
    Worker w = new Worker();  
    w.name="김현지";  
    w.age=30;  
    w.workerID=128;  
    w.eat();  
    w.sleep();  
    w.goToWork();  
  
}
```



# 클래스의 상속

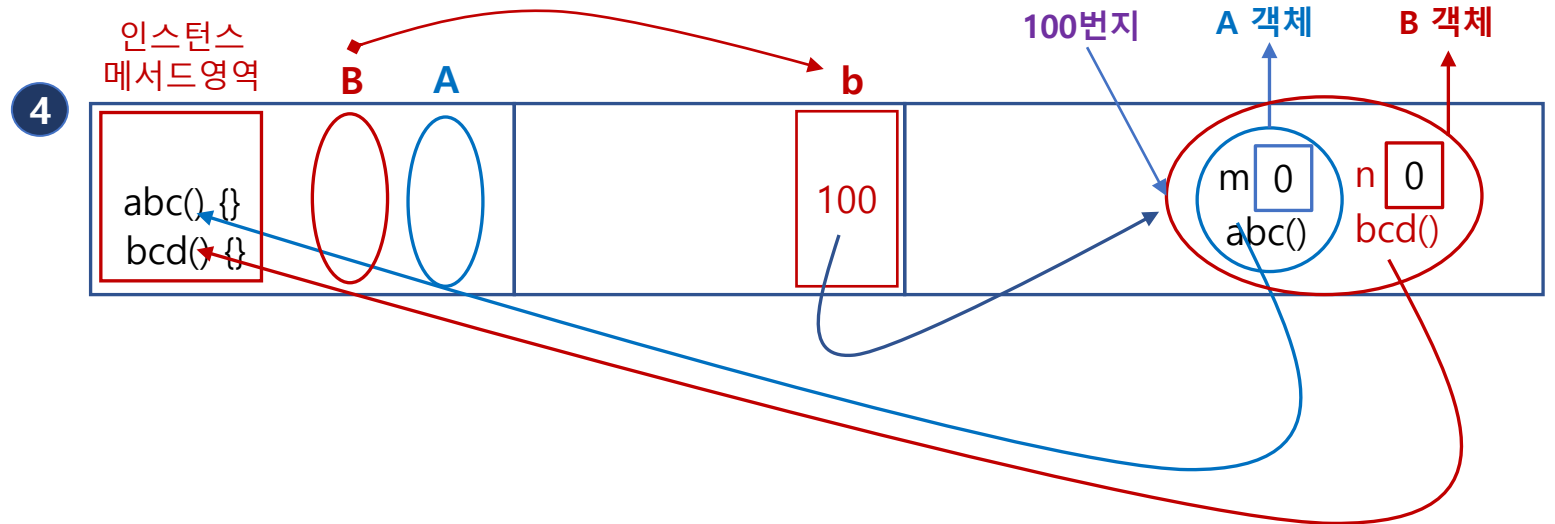
☞ 상속시 메모리의 구조



## 2 CHECK

- 상속을 받으면 부모클래스의 멤버를 가질 수 있는 이유는 객체 속에 부모클래스의 객체를 먼저 생성하여 포함하기 때문

3 `B b = new B();`



# 클래스의 상속

👉 생성자의 상속 여부

1 - Q. 생성자는 상속이 될까?

- 만일 생성자가 상속된다면 B 클래스 내에 A() 생성자가 존재하는 형태

2

```
class A {  
    A() {  
    }  
}
```

3

```
class B extends A  
{  
    A(){ }  
}
```

4

생성자의 두가지 조건

- 클래스 이름과 동일
- 리턴타입이 없음

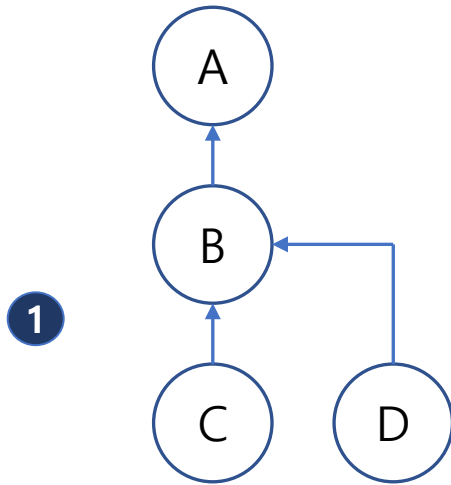
1. 이름이 달라 생성자 자격 없음
2. 리턴타입이 없어 메서드는 아님

5

ANS. 생성자는 상속되지 않는다.

# 클래스의 상속

👉 객체의 다형적 표현



## 상속관계의 코드 표현

A는 A이다. → A a1 = new A();  
B는 A이다. → A a2 = new B();

2

## 객체의 생성 #1

```
A a = new A(); (O)  
B b = new B(); (O)  
C c = new C(); (O)  
D d = new D(); (O)
```

좌우 타입은 동일

3

## 객체의 생성 #2

```
A a1 = new B(); (O)  
A a2 = new C(); (O)  
A a3 = new D(); (O)  
  
B b1 = new C(); (O)  
B b2 = new D(); (O)
```

업캐스팅으로 자동변환

4

## 잘못된 객체 생성

```
B b1 = new A(); (X)  
  
C c1 = new A(); (X)  
C c2 = new B(); (X)  
  
D d1 = new A(); (X)  
D d2 = new B(); (X)  
D d3 = new C(); (X)
```

# 클래스의 상속

👉 객체의 다형적 표현

1

```
class A{ }  
class B extends A{ }  
class C extends B{ }  
class D extends B{ }
```



2

```
public static void main(String[] ar) {  
    // # A  
    A aa = new A();  
    A ab = new B();  
    A ac = new C();  
    A ad = new D();  
  
    // # B  
    // B ba = new A(); //오류  
    B bb = new B();  
    B bc = new C();  
    B bd = new D();  
  
    // # C  
    // C ca = new A(); //오류  
    // C cb = new B(); //오류  
    C cc = new C();  
    // C cd = new D(); //오류  
  
    // # D  
    // D da = new A(); //오류  
    // D db = new B(); //오류  
    // D dc = new C(); //오류  
    D dd = new D();  
}
```

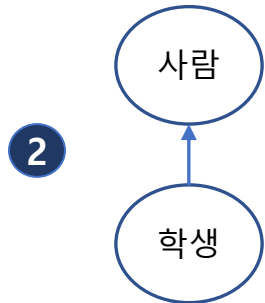
# The End

# 객체의 타입변환

# 객체의 타입변환

## ☞ 객체의 타입 변환 (업캐스팅과 다운캐스팅)

- 1
  - 상속관계에 있는 경우 객체도 타입변환이 가능
  - 업캐스팅은 항상 가능 : 생략시 컴파일러에 의해 자동캐스팅
  - 다운캐스팅은 때에 따라서 가능/불가능 : 가능한 경우에만 수동으로 직접 캐스팅 필요



(업캐스팅) 학생은 사람이다. → 항상 O

(다운캐스팅) 사람은 학생이다.  
→ 학생인 사람일때는 O  
→ 학생이 아닌 사람일때는 X

```
사람 human1 = new 사람();  
사람 human2 = new 학생();
```

학생과 학생이 아닌 사람이  
모두 포함된 사람 객체

학생인 사람 객체

3 human1 → 학생 (가능) ?

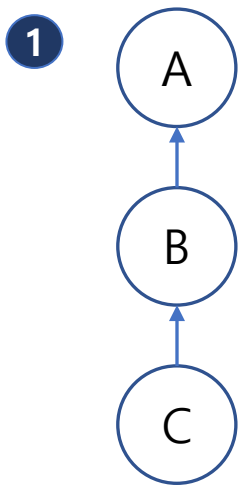
human2 → 학생 (가능) ?

4

```
Exception in thread "main" java.lang.ClassCastException:  
    at pack05_inheritanceandpolymorphism.sec02.EX03_T
```

# 객체의 타입 변환

☞ 객체의 타입 변환 (업캐스팅과 다운캐스팅)



2

**자동타입변환(업캐스팅)**

```
B b = new B();  
A a = (A) b;  
  
C c = new C();  
B b = (B) c;  
A a = (A) c;
```

컴파일러가  
자동으로 추가

4

**CHECK**

다운캐스팅은 언제 가능할까??

3

**수동타입변환(다운캐스팅)**

```
A a = new A();  
B b = (B) a; //→오류발생  
  
A a = new B();  
B b = (B) a; //→0  
C c = (C) a; //→오류발생
```

**CHECK!!**

다운캐스팅은 때에 따라  
가능하기도 불가능하기도 함



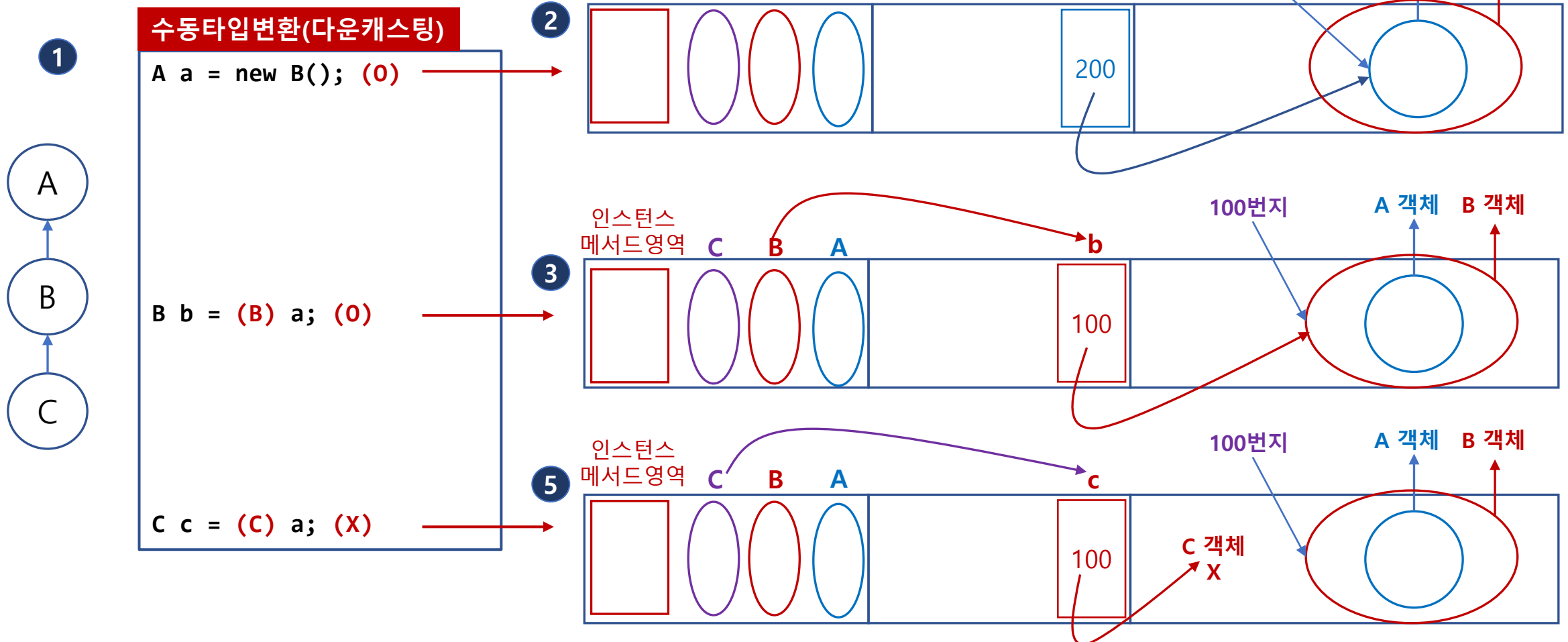
# 객체의 타입변환

4

CHECK

- 다운 캐스팅이 가능하기 위해서는  
Heap 메모리 내에 해당 객체가 있어야 함

☞ 메모리로 이해하는 다운캐스팅



# 객체의 타입변환

☞ 선언 타입에 따른 차이점

선언한 타입의 멤버만 사용 가능

1

```
class A {  
    int m=3;  
    void abc(){  
        System.out.println("A");  
    }  
}
```

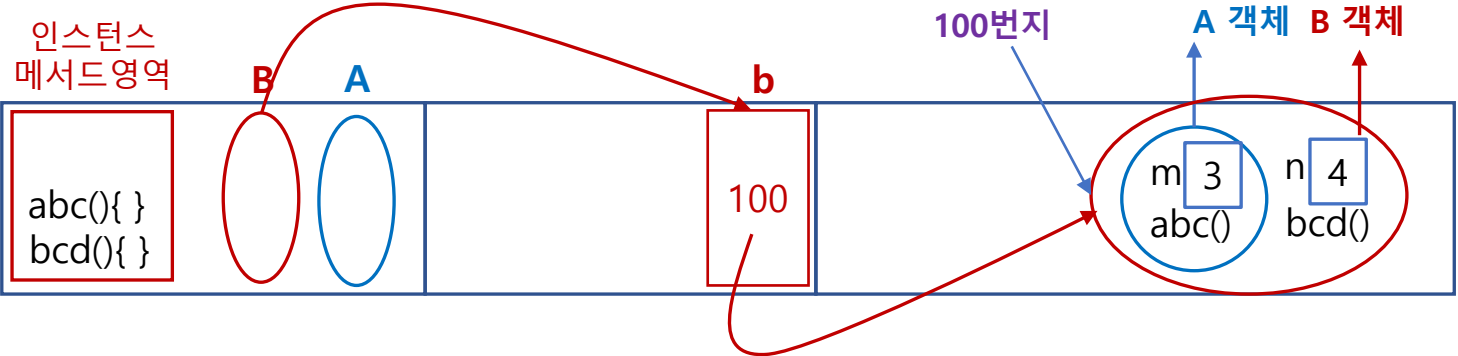
```
class B extends A {  
    int n=4;  
    void bcd(){  
        System.out.println("B");  
    }  
}
```

2

선언 타입에 따른 차이점

```
B b = new B();  
System.out.println(b.m); (O)  
System.out.println(b.n); (O)  
b.abc(); (O)  
b.bcd(); (O)
```

3

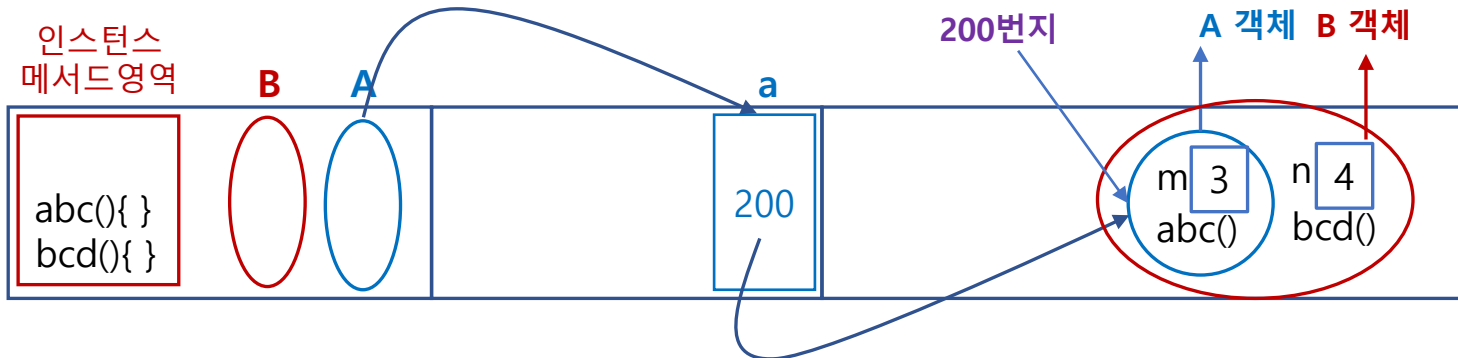


4

선언 타입에 따른 차이점

```
A a = new B();  
System.out.println(a.m); (O)  
System.out.println(a.n); (X)  
a.abc(); (O)  
a.bcd(); (X)
```

5



# 객체의 타입변환

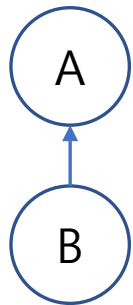
☞ 다운캐스팅 가능 여부 확인

1

참조변수 **instanceof** 타입

→ true / false

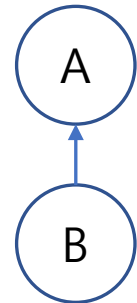
2



```
A a = new B();  
if( a instanceof B ){ //true  
    B b = (B)a;  
}
```

3

```
A a = new A();  
if( a instanceof B ){ //false  
    B b = (B)a;  
}
```



# 객체의 타입변환

## ☞ 객체의 다형적 표현

```
1 class A{  
  class B extends A{}
```

```
2 public static void main(String[] ar) {  
  
  // instanceof  
  A aa = new A();  
  A ab = new B();  
  
  System.out.println(aa instanceof A); //true  
  System.out.println(aa instanceof B); //false  
  
  System.out.println(ab instanceof A); //true  
  System.out.println(ab instanceof B); //true
```

```
3 if(aa instanceof B) {  
    B b = (B)aa;  
    System.out.println("aa를 B로 casting을 완료하였습니다.");  
} else {  
    System.out.println("aa는 B로 casting을 할 수 없습니다.");  
}  
  
4 if(ab instanceof B) {  
    B b = (B)ab;  
    System.out.println("ab를 B로 casting을 완료하였습니다.");  
} else {  
    System.out.println("ab는 B로 casting을 할 수 없습니다.");  
}  
  
5 if("안녕" instanceof String) {  
    System.out.println("\"안녕\"은 String 입니다.");  
}  
}
```

true  
false  
true  
true  
aa는 B로 casting을 할 수 없습니다.  
ab를 B로 casting을 완료하였습니다.  
"안녕"은 String 입니다.



# The End

# 메서드 오버라이딩(Overriding)

# 메서드 오버라이딩(Overriding)

## ☞ 메서드 오버라이딩 (Method Overriding)

### 1 - 메서드 오버라이딩이란?

부모클래스에게 상속받은 메서드를 재정의하여 사용 (덮어쓰기 개념)

### 3 - 메서드 오버라이딩을 위한 조건


부모클래스의 메서드와 **시그너처** 및 **리턴 타입** 동일

부모클래스의 메서드보다 **접근지정자**는 같거나 넓어야 함

2

```
class A {  
    void print(){  
        System.out.println("A 클래스");  
    }  
}
```

```
class B extends A {  
    void print(){  
        System.out.println("B 클래스");  
    }  
}
```



# 메서드 오버라이딩(Method Overriding)

☞ 메서드 오버라이딩 (Method Overriding)의 메모리 구조

1

```
class A {  
    void print(){  
        System.out.println("A 클래스");  
    }  
}
```

```
class B extends A {  
    void print(){  
        System.out.println("B 클래스");  
    }  
}
```

2

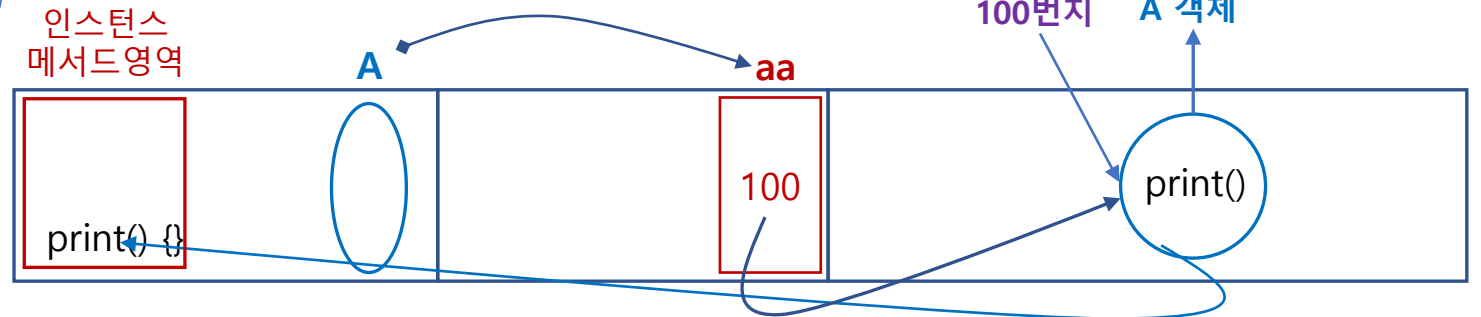
A aa = new A();

4

aa.print(); → A 클래스

3

인스턴스  
메서드영역





## 메서드 오버라이딩(Overriding)

## 👉 메서드 오버라이딩 (Method Overriding)의 메모리 구조

1

```
class A {  
  
    void print(){  
        System.out.println("A 클래스");  
    }  
  
}
```

```
class B extends A {  
  
    void print(){  
        System.out.println("B 클래스");  
    }  
  
}
```

5

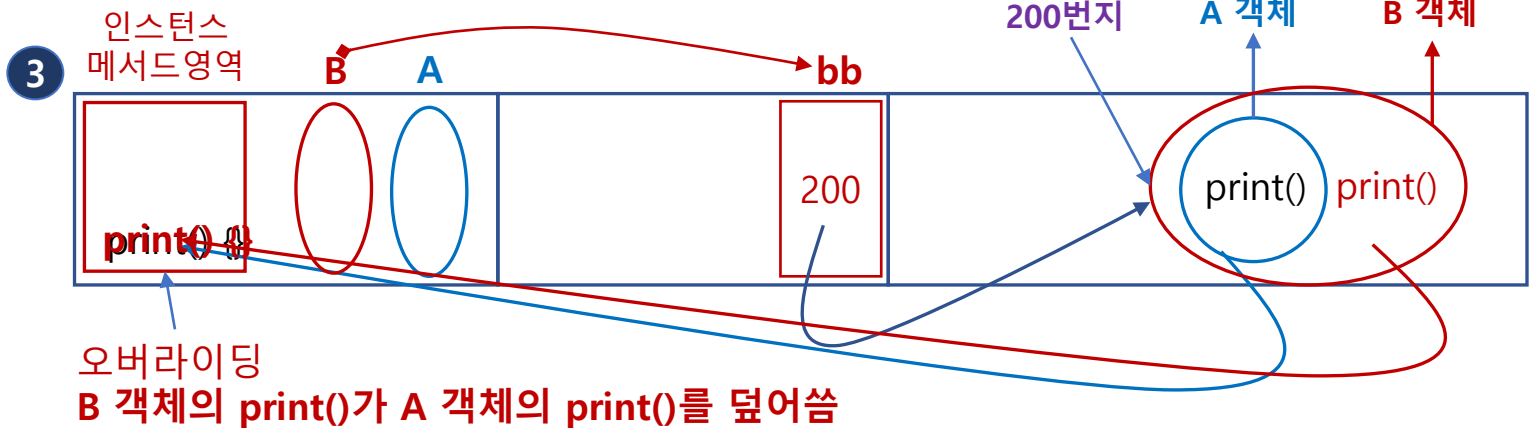
- 객체내에 동일한 멤버가 두개 이상 있는 경우  
: 참조변수가 가리키는 객체의 바깥쪽부터  
안쪽으로 들어가면서 첫번째 만나는 멤버가 실행

2

```
B bb = new B();
```

4

bb.print(); → B 클래스



# 메서드 오버라이딩(Method Overriding)

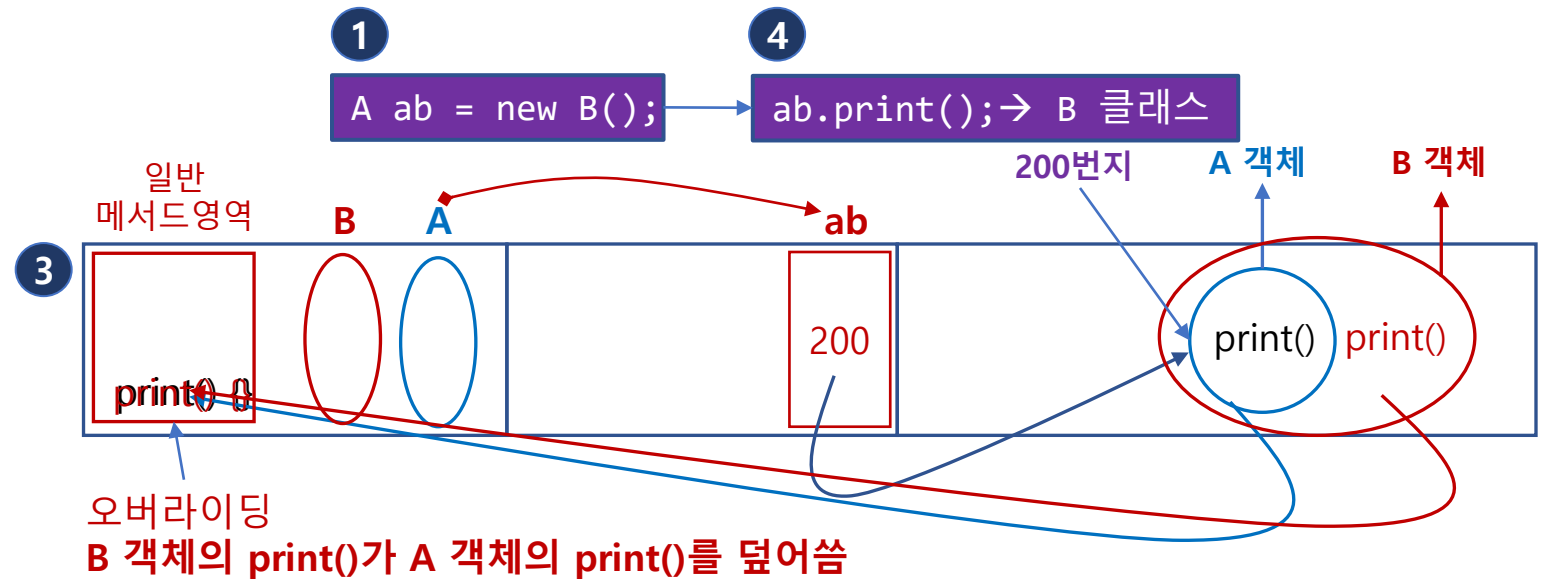
☞ 메서드 오버라이딩 (Method Overriding)의 메모리 구조

```
class A {  
    void print(){  
        System.out.println("A 클래스");  
    }  
}
```

```
class B extends A {  
    void print(){  
        System.out.println("B 클래스");  
    }  
}
```

2 CHECK

A a = new B();  
상속에 의한 다형적 표현 ( B는 A이다)



# 메서드 오버라이딩(Overriding)

☞ 메서드 오버라이딩 (Method Overriding)의 메모리 구조

```
class A {  
    void print(){  
        System.out.println("A 클래스");  
    }  
}
```

```
class B extends A {  
    void print(){  
        System.out.println("B 클래스");  
    }  
}
```

1

// #1. A 타입 선언 A 객체 생성

```
A aa = new A();  
aa.print(); // A 클래스
```

2

// #2. B 타입 선언 B 객체 생성

```
B bb = new B();  
bb.print(); // B 클래스
```

3

// #3. A 타입 선언 B 객체 생성 (다형적 표현)

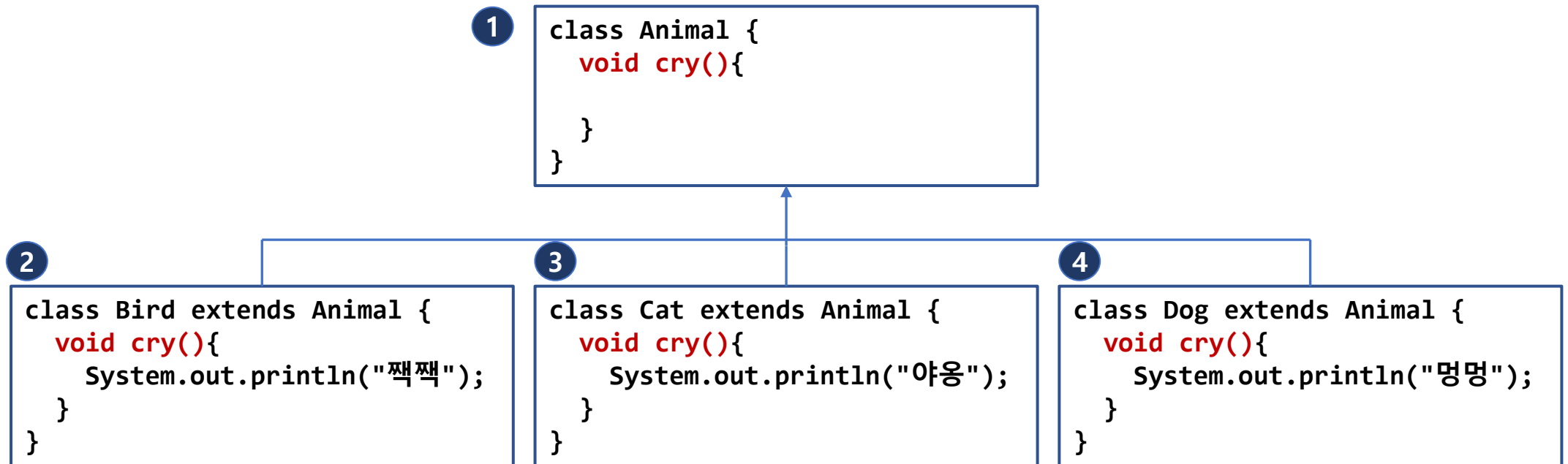
```
A ab = new B();  
ab.print(); // B 클래스
```

}

```
A 클래스  
B 클래스  
B 클래스
```

# 메서드 오버라이딩(Method Overriding)

☞ 메서드 오버라이딩 (Method Overriding) 대표적인 예



# 메서드 오버라이딩(Method Overriding)

☞ 메서드 오버라이딩 (Method Overriding) 대표적인 예

1

각각의 타입으로 선언 + 각각의 타입으로 객체 생성

```
Animal aa = new Animal();  
Bird bb = new Bird();  
Cat cc = new Cat();  
Dog dd = new Dog();  
  
aa.cry(); //출력없음  
bb.cry(); // 짹짹  
cc.cry(); //야옹  
dd.cry(); //멍멍
```

2

Animal 타입으로 선언 + 자식클래스 타입으로 객체 생성 (다형적 표현)

```
Animal ab = new Bird();  
Animal ac = new Cat();  
Animal ad = new Dog();  
  
ab.cry(); // 짹짹  
ac.cry(); //야옹  
ad.cry(); //멍멍
```



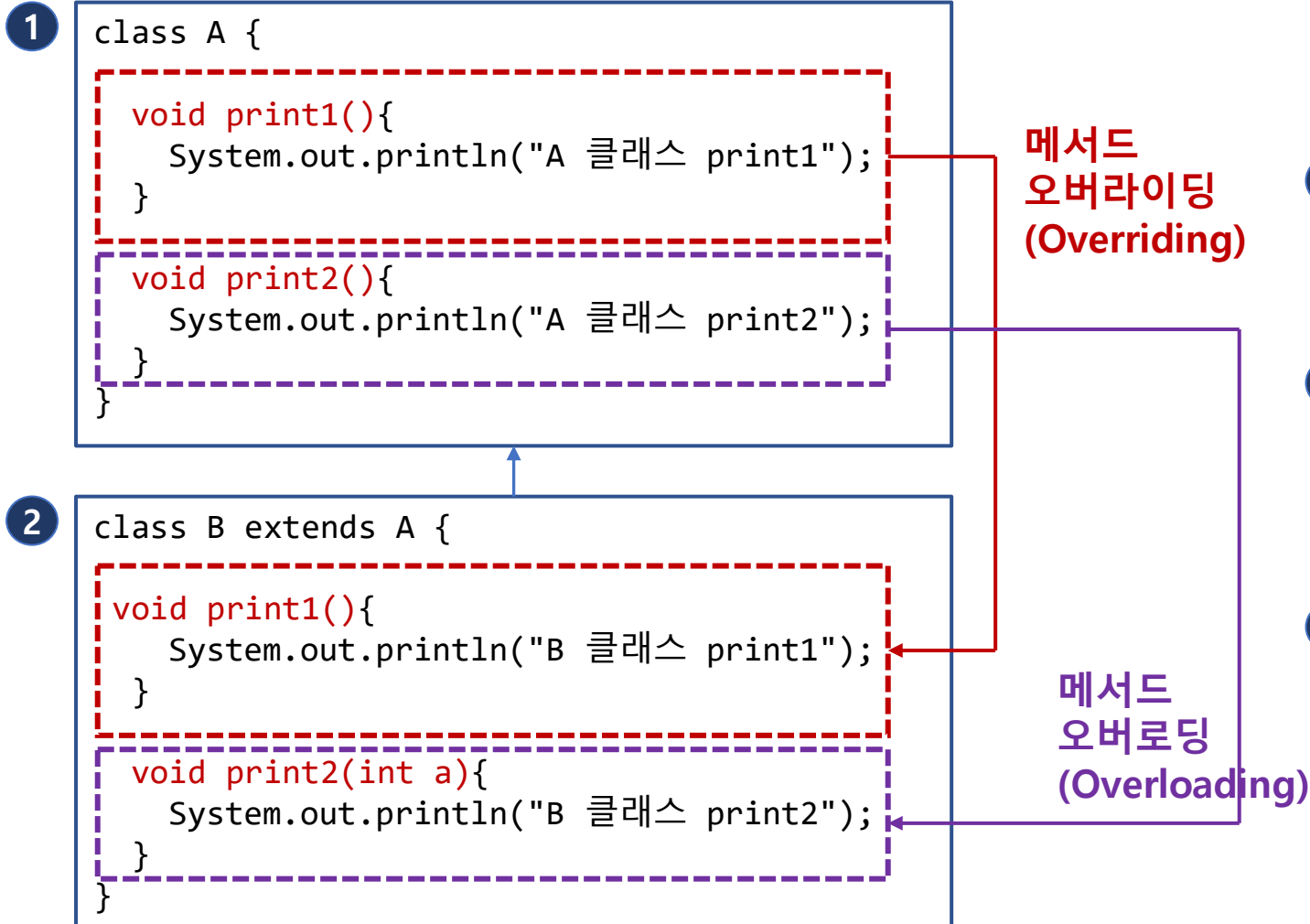
3

배열로 한번에 관리 가능

```
Animal[] animals = new Animal[] {new Bird(), new Cat(), new Dog()};  
for(Animal animal : animals) {  
    animal.cry();  
} // 짹짹, 야옹, 멍멍  
}
```

# 메서드 오버라이딩(Overriding)

👉 주의. 메서드 오버라이딩 (Method Overriding) vs. 메서드 오버로딩 (Method Overloading)

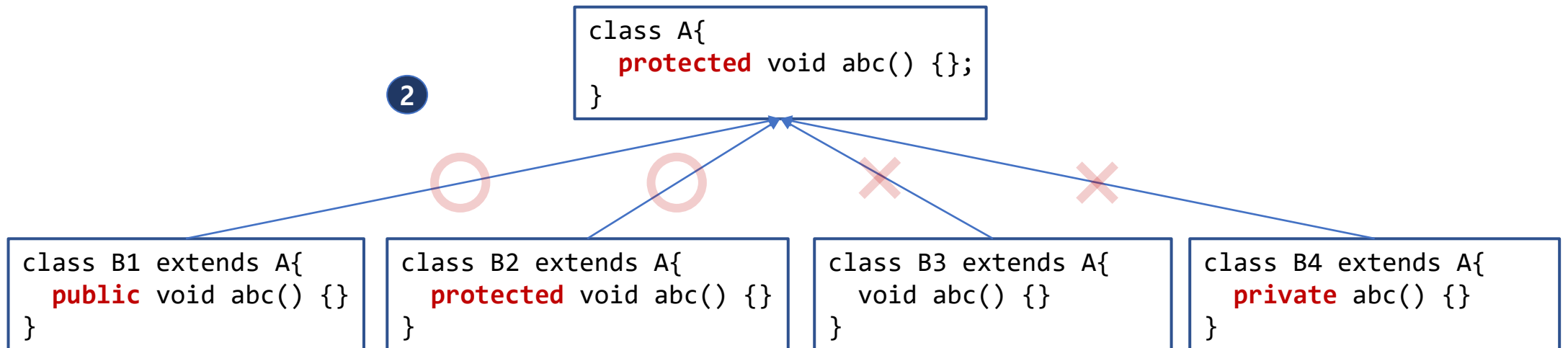


```
public static void main(String[] ar) {  
  
    3 // #1. A 타입 선언 A 객체 생성  
      A aa = new A();  
      aa.print1(); // A 클래스 print1  
      aa.print2(); // A 클래스 print2  
  
    4 // #2. B 타입 선언 B 객체 생성  
      B bb = new B();  
      bb.print1(); // B 클래스 print1  
      bb.print2(); // A 클래스 print2  
      bb.print2(3); // B 클래스 print2  
  
    5 // #3. A 타입 선언 B 객체 생성 (다형적 표현)  
      A ab = new B();  
      ab.print1(); // B 클래스 print1  
      ab.print2(); // A 클래스 print2  
      // ab.print2(3); // 오류  
}
```

# 메서드 오버라이딩(Method Overriding)

☞ 메서드 오버라이딩 (Method Overriding)과 접근지정자

- 1 - 오버라이딩 시 **자식클래스의 메서드 접근지정자는 부모의 접근지정자보다 같거나 커야 함** (즉, 좁혀질 수 없음)



# The End

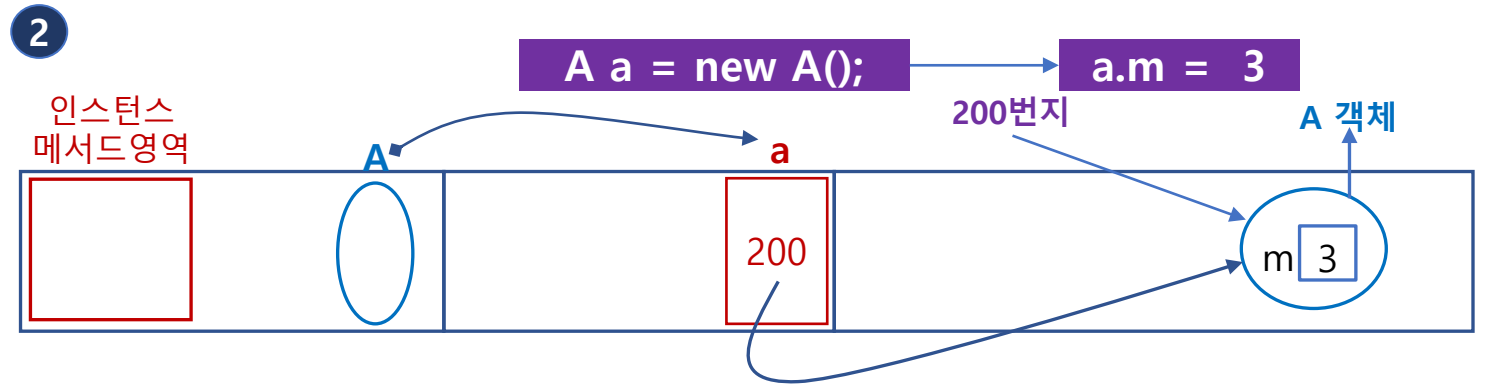
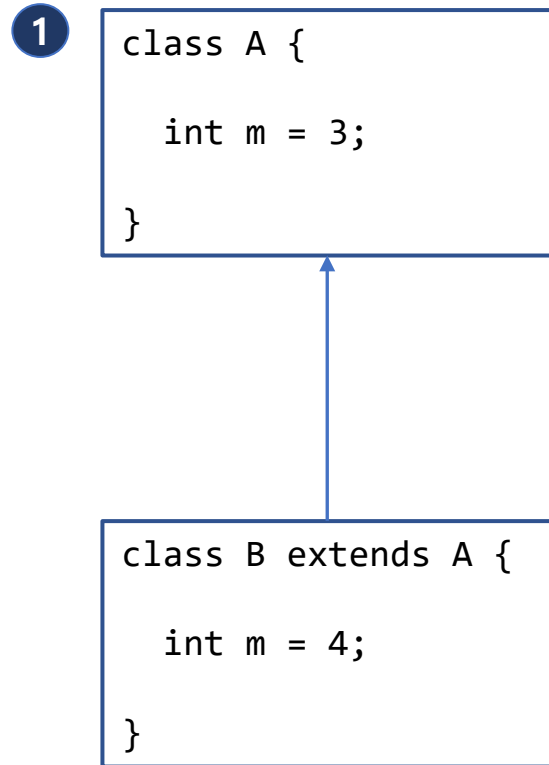


# 필드와 static 멤버(필드/메서드)의 중복

# 필드와 static 멤버(필드/메서드)의 중복

**NO! 인스턴스 필드는 오버라이딩이 되지 않음**

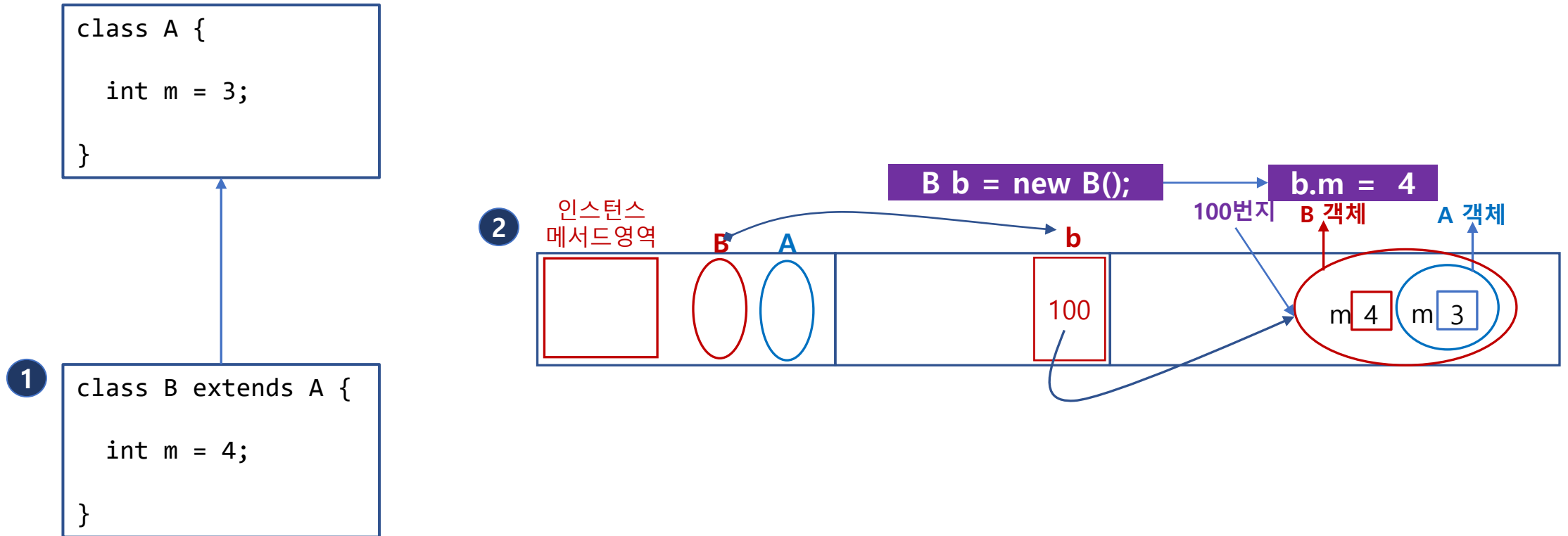
☞ 인스턴스 필드도 오버라이딩(Overriding)?



# 필드와 static 멤버(필드/메서드)의 중복

**NO! 인스턴스 필드는 오버라이딩이 되지 않음**

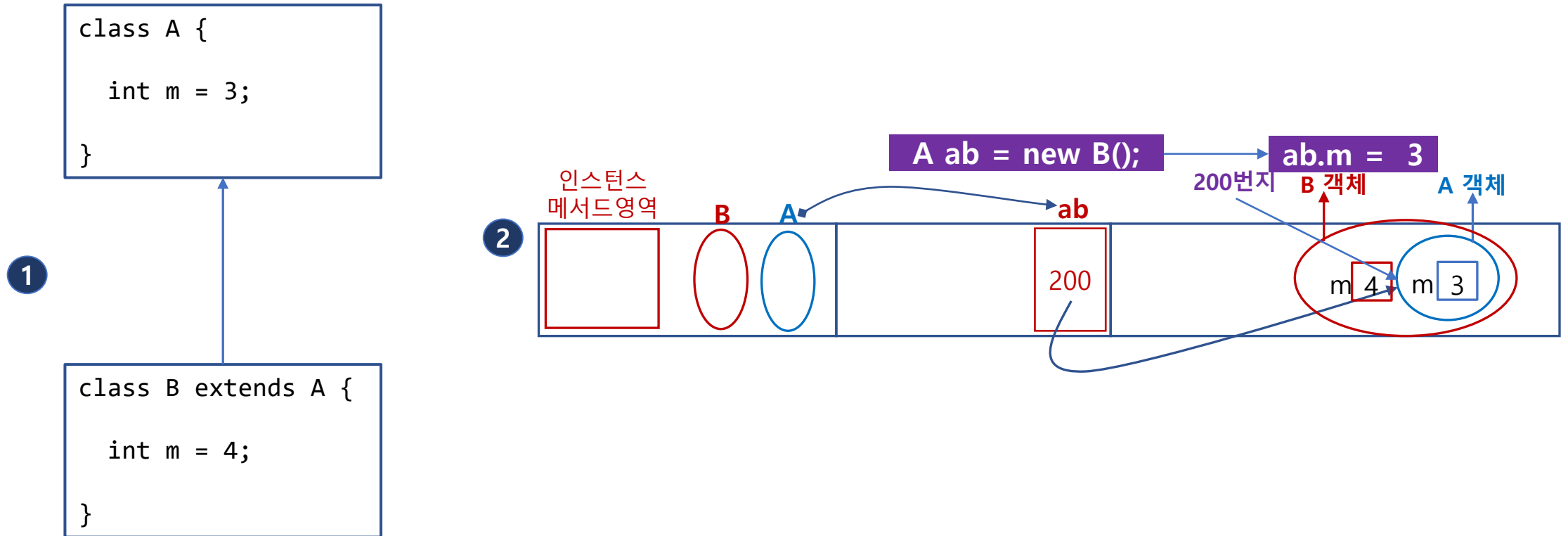
☞ **인스턴스 필드**도 오버라이딩(Overriding)?



# 필드와 static 멤버(필드/메서드)의 중복

**NO! 인스턴스 필드는 오버라이딩이 되지 않음**

☞ 인스턴스 필드도 오버라이딩(Overriding)?



# 필드와 static 멤버(필드/메서드)의 중복

3

**NO! 인스턴스 필드는 오버라이딩이 되지 않음**

☞ 인스턴스 필드도 오버라이딩(Overriding)?

```
class A{  
    int m = 3;  
}
```

1

```
class B extends A{  
    int m = 4;  
}
```

2

```
public static void main(String[] ar) {  
  
    // #1. 객체 생성  
    A aa = new A();  
    B bb = new B();  
    A ab = new B();  
  
    // #2. 인스턴스 필드  
    System.out.println(aa.m); //3  
    System.out.println(bb.m); //4  
    System.out.println(ab.m); //3  
}
```

3  
4  
3

# 필드와 static 멤버(필드/메서드)의 중복

NO! static 필드는 오버라이딩이 되지 않음

👉 static 필드도 오버라이딩(Overriding)?

1

```
class A {  
    static int m = 3;  
}
```

```
class B extends A {  
    static int m = 4;  
}
```

2

A.m=3

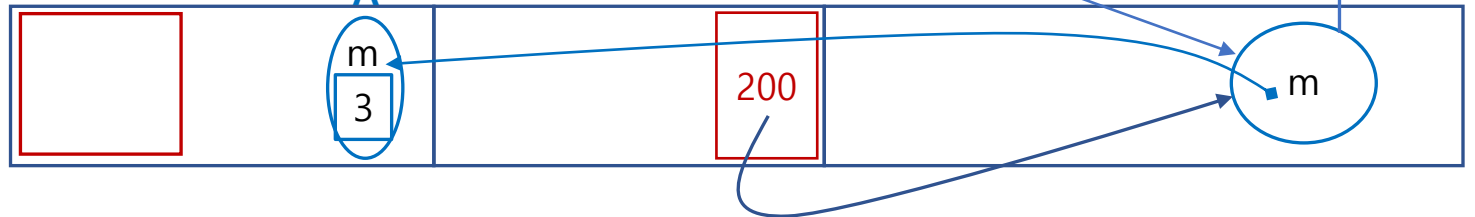
3

A a = new A();

a.m = 3

4

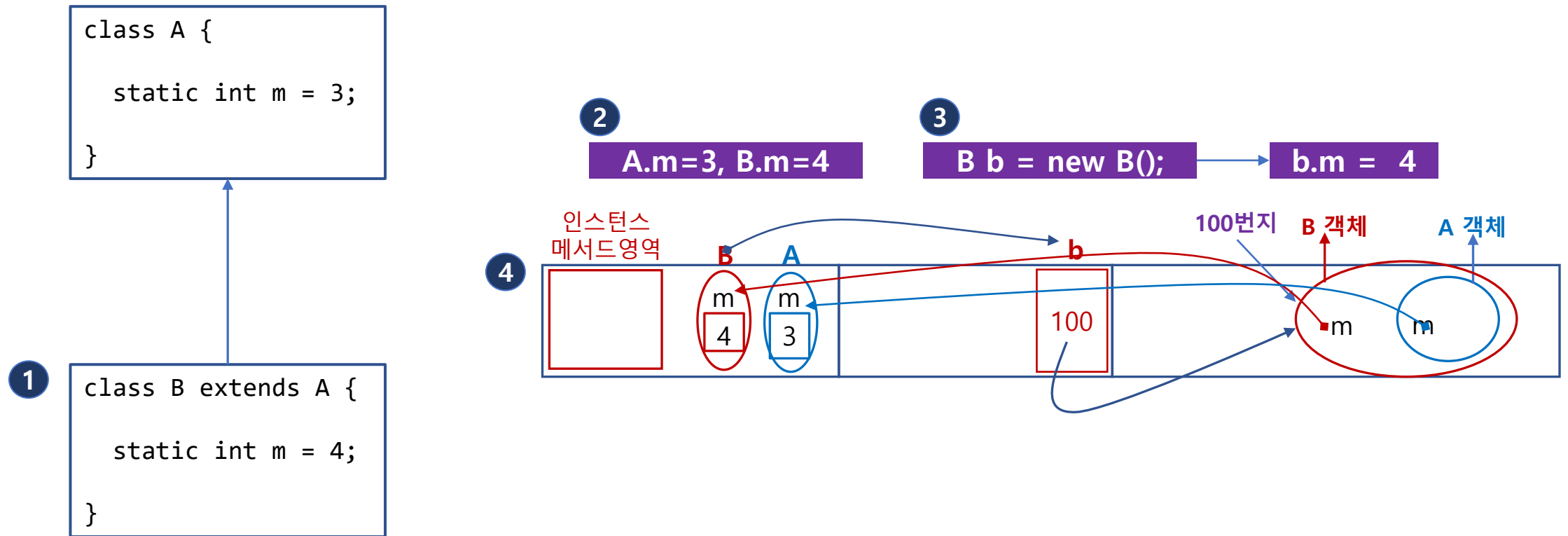
인스턴스  
메서드영역



# 필드와 static 멤버(필드/메서드)의 중복

NO! static 필드는 오버라이딩이 되지 않음

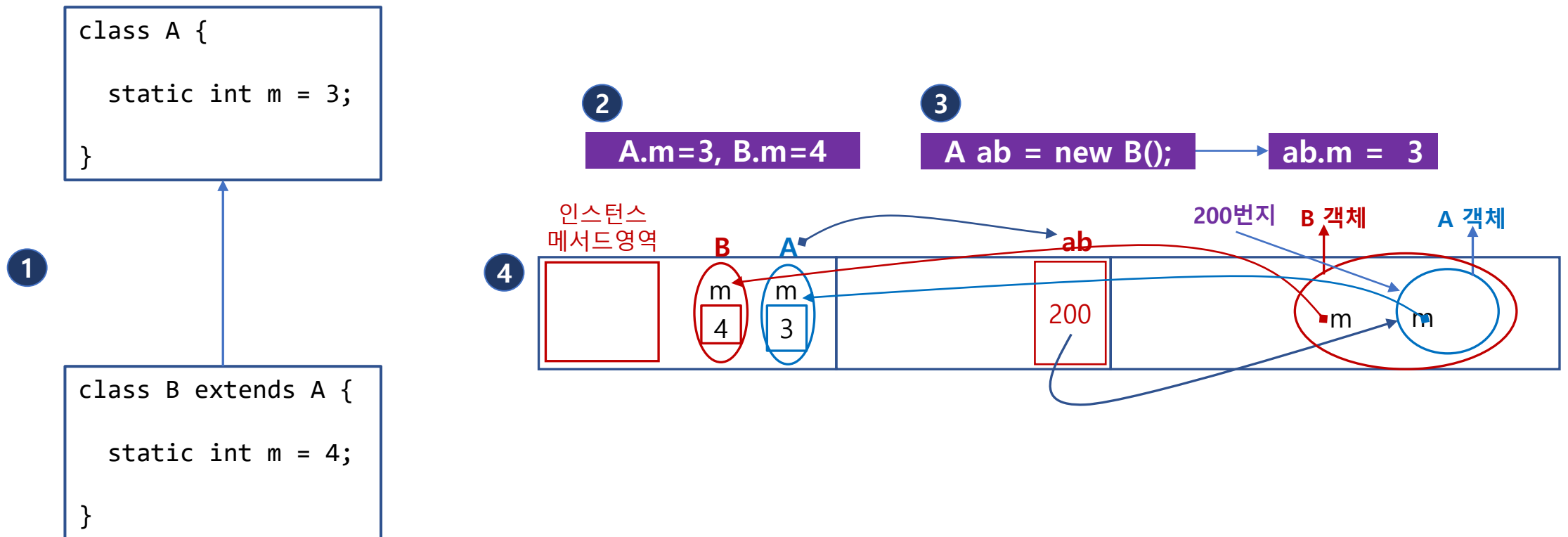
👉 static 필드도 오버라이딩(Overriding)?



# 필드와 static 멤버(필드/메서드)의 중복

NO! static 필드는 오버라이딩이 되지 않음

👉 static 필드도 오버라이딩(Overriding)?





# 필드와 static 멤버(필드/메서드)의 중복

4

NO! static 필드는 오버라이딩이 되지 않음

👉 static 필드도 오버라이딩(Overriding)?

```
class A{  
    static int m = 3;  
}
```

```
class B extends A{  
    static int m = 4;  
}
```

```
1 // #1. 클래스 이름으로 바로 접근  
System.out.println(A.m); //3  
System.out.println(B.m); //4  
  
2 // #2. 객체 생성  
A aa = new A();  
B bb = new B();  
A ab = new B();  
  
3 // #3. 객체 생성을 통한 static 필드  
System.out.println(aa.m); //3  
System.out.println(bb.m); //4  
System.out.println(ab.m); //3  
}
```

3  
4  
3  
4  
3

# 필드와 static 멤버(필드/메서드)의 중복

**NO! static 메서드는 오버라이딩이 되지 않음**

👉 static 메서드도 오버라이딩(Overriding)?

1

```
class A {  
    static void print(){  
        System.out.println("A");  
    }  
}
```

```
class B extends A {  
    static void print(){  
        System.out.println("B");  
    }  
}
```

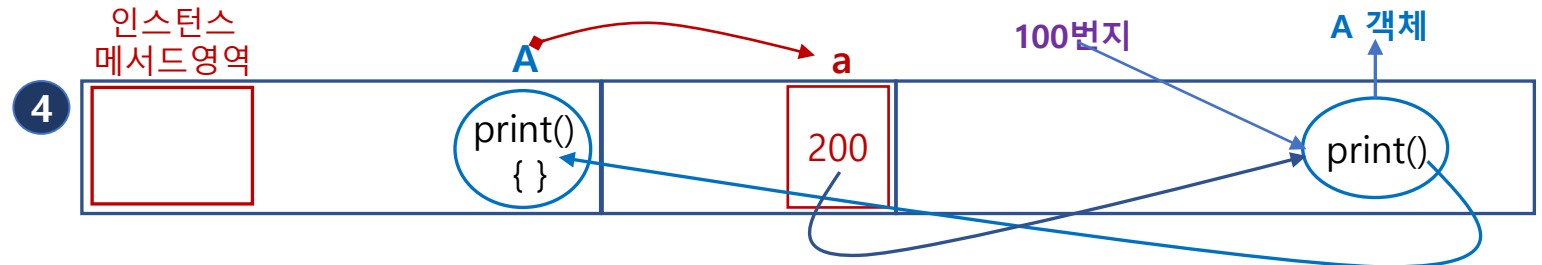
2

A.print() = A

3

A a = new A();

a.print() = A



# 필드와 static 멤버(필드/메서드)의 중복

**NO! static 메서드는 오버라이딩이 되지 않음**

👉 static 메서드도 오버라이딩(Overriding)?

```
class A {  
    static void print(){  
        System.out.println("A");  
    }  
}
```

```
class B extends A {  
    static void print(){  
        System.out.println("B");  
    }  
}
```

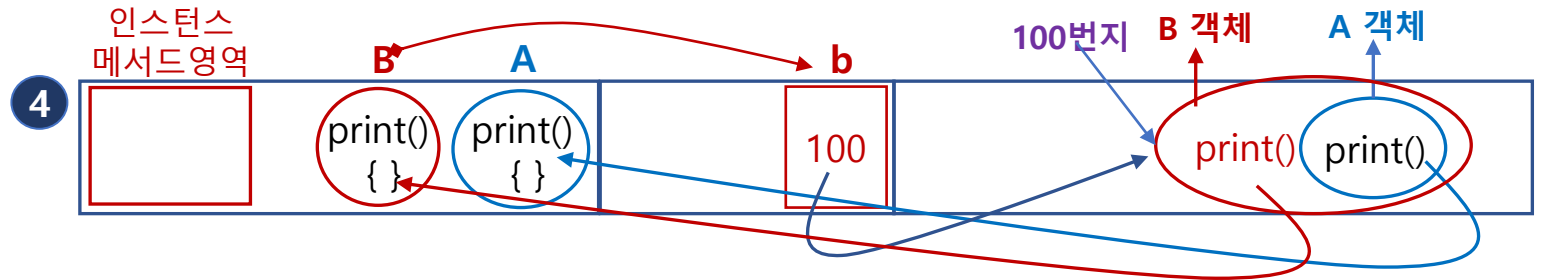
2

**A.print() = A, B.print() = B**

3

**B b = new B();**

**b.print() = B**



# 필드와 static 멤버(필드/메서드)의 중복

NO! static 메서드는 오버라이딩이 되지 않음

👉 static 메서드도 오버라이딩(Overriding)?

```
class A {  
    static void print(){  
        System.out.println("A");  
    }  
}
```

1

```
class B extends A {  
    static void print(){  
        System.out.println("B");  
    }  
}
```

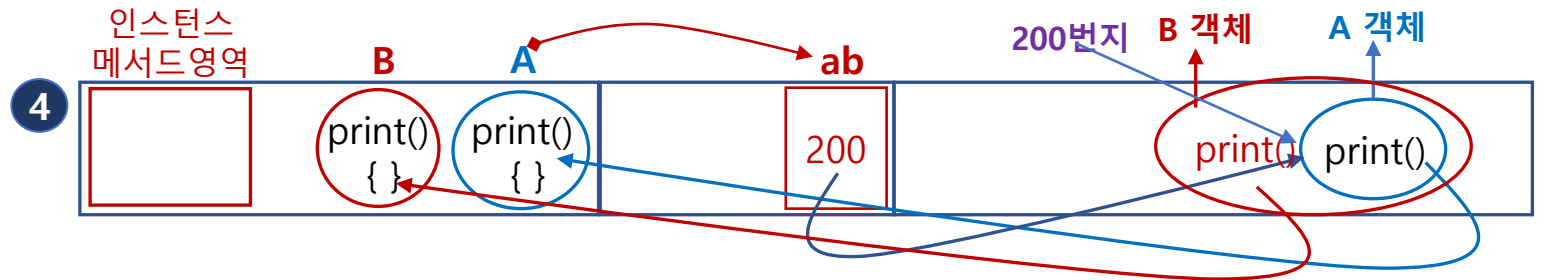
2

A.print() = A, B.print() = B

3

A ab = new B();

ab.print() = A



# 필드와 static 멤버(필드/메서드)의 중복

3

NO! static 메서드는 오버라이딩이 되지 않음

👉 static 메서드도 오버라이딩(Overriding)?

```
class A{  
  
    static void print() {  
        System.out.println("A 클래스");  
    }  
}
```

1

```
class B extends A{  
  
    static void print() {  
        System.out.println("B 클래스");  
    }  
}
```

2

```
public static void main(String[] ar) {  
  
    A.print(); //A 클래스  
    B.print(); //B 클래스  
  
    A aa = new A();  
    B bb = new B();  
    A ab = new B();  
  
    aa.print(); //A 클래스  
    bb.print(); //B 클래스  
    ab.print(); //A 클래스  
}
```

```
A 클래스  
B 클래스  
A 클래스  
B 클래스  
A 클래스
```

# 필드와 static 멤버(필드/메서드)의 중복

👉 인스턴스 멤버/static 멤버 오버라이딩 여부 정리

1

- Instance 필드

오버라이딩 (X)

- Instance 메서드

오버라이딩 (O)

메서드 오버라이딩

- static 필드

오버라이딩 (X)

- static 메서드

오버라이딩 (X)

2

기준점

instance 필드  
static 필드  
static 메서드

instance 메서드

A a = new B();

# The End

# super와 super()



# 클래스의 상속과 다형성

☞ super 키워드 vs. super() 메서드

1 - super 키워드 → 부모클래스의 객체

2

- 필드명 중복 또는 메서드 오버라이딩으로 가려진 부모의 필드/메서드를 호출하기 위해 주로 사용

3

```
class A{
    void abc(){
        System.out.println("A 클래스 abc()");
    }
}
```

```
class B extends A {
    void abc(){
        System.out.println("B 클래스 abc()");
    }
    void bcd(){
        abc(); //→this.abc();
    }
}
```

4

B b = new B();

b.bcd(); → B 클래스 abc()

5

```
class A{
    void abc(){
        System.out.println("A 클래스 abc()");
    }
}
```

```
class B extends A {
    void abc(){
        System.out.println("B 클래스 abc()");
    }
    void bcd(){
        super.abc();
    }
}
```

6

B b = new B();

b.bcd(); → A 클래스 abc()

# 클래스의 상속과 다형성

👉 super 키워드 vs. super() 메서드

- super 키워드 → 부모클래스의 객체

## super 키워드가 자주 사용되는 이유

1

```
class A{
    void init(){
        메모리할당/화면세팅/변수초기화 등 100줄 코드
    }
}
```

101 줄  
코드

```
class B extends A {
    void init(){
        메모리할당/화면세팅/변수초기화 등 : 100줄 코드
        화면출력 : 1줄코드 (추가하고자 하는 기능)
    }
}
```

2

```
class A{
    void init(){
        메모리할당/화면세팅/변수초기화 등 100줄 코드
    }
}
```

2 줄  
코드

```
class B extends A {
    void init(){
        super.init();
        화면출력 : 1줄코드 (추가하고자 하는 기능)
    }
}
```

# 클래스의 상속과 다형성

☞ super 키워드 vs. super() 메서드

- super() 메서드 → 부모 클래스의 생성자를 호출

2

자식 클래스 객체 속에 부모 객체가 포함될 수 있었던 이유

1

- super() 메서드는 **생성자 내부에서만 사용** 가능
- 반드시 중괄호 이후 **첫 줄에 위치** 하여야 함
- 자식클래스 생성자의 **첫 줄에는 반드시 this() 또는 super()가 포함** 되어야 함  
(생략시 컴파일러가 자동으로 super() 추가)

3

```
class A{
    A(){
        System.out.println("A 생성자");
    }
}
```

4

```
B b = new B();
```

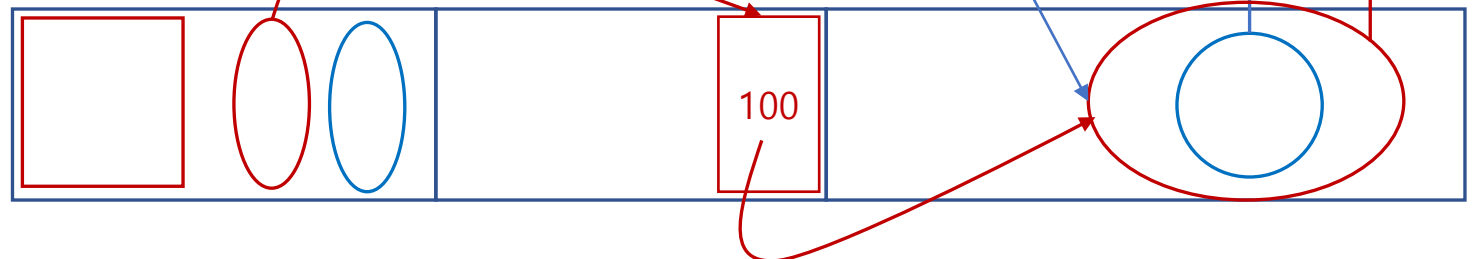
A 생성자  
B 생성자

5

```
class B extends A {
    B(){
        super();
        System.out.println("B 생성자");
    }
}
```

생략시 컴파일러가  
자동으로 삽입

인스턴스  
메서드영역



# 클래스의 상속과 다형성

☞ super 키워드 vs. super() 메서드

1

```
class A{
    A(int a){
        System.out.println("A 생성자");
    }
}
```

```
class B extends A {
}
```

오류발생

컴파일러에 의해서  
자동으로 추가되는 생성자

```
B(){
    super();
}
```

✗ 기본생성자가  
없음 (에러)

2

```
class A{
    A(){
        this(3);
        System.out.println("A 생성자1");
    }
    A(int a){
        System.out.println("A 생성자2");
    }
}
```

```
class B extends A {
    B(){
        this(3);
        System.out.println("B 생성자1");
    }
    B(int a){
        System.out.println("B 생성자2");
    }
}
```

# 클래스의 상속과 다형성

☞ super 키워드 vs. super() 메서드

1

```
class A{
    A(){
        this(3);
        System.out.println("A 생성자1");
    }
    A(int a){
        System.out.println("A 생성자2");
    }
}
```

```
class B extends A {
    B(){
        this(3);
        System.out.println("B 생성자1");
    }
    B(int a){
        System.out.println("B 생성자2");
    }
}
```

A aa1 = new A();



A 생성자2  
A 생성자1

A aa2 = new A(2);



A 생성자2

2

B bb1 = new B();



A 생성자2  
A 생성자1  
B 생성자2  
B 생성자1

B bb2 = new B(2);



A 생성자2  
A 생성자1  
B 생성자2

# The End

# 최상위 클래스 Object

# 최상위 클래스 Object

System.out.println(new A());와 같이 모든 클래스 타입을 출력할 수 있었던 이유

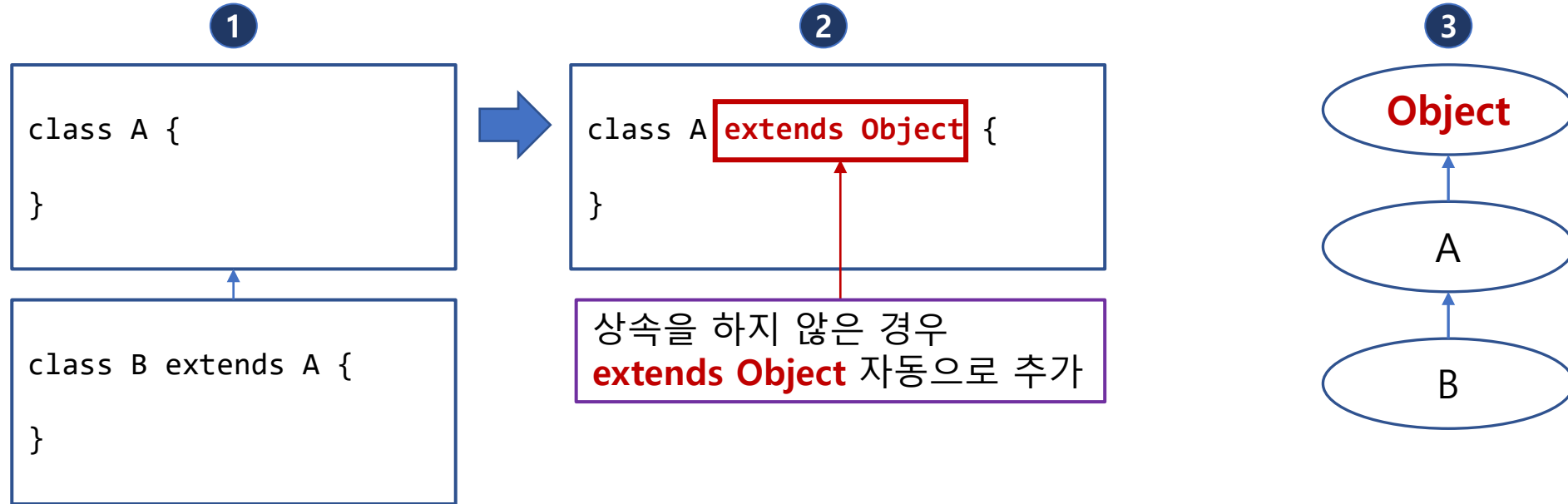
5

void

println(Object x)

Prints an Object and then terminate the line.

☞ Object 클래스 : 모든 자바 클래스의 부모 클래스



4

```
Object oa = new A();
```

```
A aa = new A();
```

```
Object ob = new B();
```

```
A ab = new B();
```

```
B bb = new B();
```



# 최상위 클래스 Object

☞ Object 클래스 : 모든 자바 클래스의 부모 클래스

1

자바의 모든 클래스는  
Object의 자식클래스

=

자바의 모든 클래스는  
Object의 메서드를 가짐

2

| 반환타입    | 메서드명  | 주요 내용   |
|---------|---|---|
| String  | toString()  | Object 객체의 정보 패키지.클래스명@해쉬코드<br>일반적으로 오버라이딩해서 사용   |
| boolean | equals(Object obj)  | 매개변수 Obj 객체와 stack 메모리 값(번지) 비교<br>즉, 비교연산자 ==와 동일한 결과  |
| int     | hashCode()  | 객체의 hashCode() 값 리턴. hashCode, hashMap 등의 <u>동등비교</u> 에 사용<br>위치값을 기반으로 생성된 <u>고유값</u><br>(STEP1. hashCode ( )) 일치 + (STEP2. equals( )) true → 동등 |
| void    | wait()<br>wait(long timeout)<br>wait(long timeout, int nanos) | 현재의 스레드를 일시정지(waiting/timed-waiting) 상태로 전환<br>보통 notify() 또는 interrupt()로 일시정지 해제<br>동기화 블록에서만 사용가능  |
| void    | notify()<br>notifyAll()                                       | wait()를 통해 일시정지 상태의 하나의 스레드(notify()) 또는 전체 스레드<br>(notifyAll)의 일시정지 해제<br>동기화 블록에서만 사용 가능  |

# 최상위 클래스 Object

☞ Object 메서드 : toString ()

1

- 객체의 정보 패키지.클래스명@해쉬코드
- 일반적으로 오버라이딩해서 사용

1

```
class A {  
    int a=3;  
    int b=4;  
}  
  
class B { //toString() overriding  
    int a=3;  
    int b=4;  
  
    @Override  
    public String toString() {  
        return "필드값: a="+a + ", b="+b;  
    }  
}
```



3

```
public static void main(String[] ar) {  
  
    A aa = new A();  
    System.out.printf("%x\n",aa.hashCode()); //70dea4e  
    System.out.println(aa); //패키지.클래스명@해쉬코드  
                                ↘ aa.toString()이 자동 실행  
  
    B bb = new B();  
    System.out.println(bb); //필드값: a=3, b=4  
                                ↘ bb.toString()이 자동 실행  
}
```

70dea4e  
Pack05...A@70dea4e  
필드값: a=3, b=4

# 최상위 클래스 Object

☞ Object 메서드 : equals()

1 객체의 stack 메모리 값(번지) 비교 (비교연산자 ==와 동일한 결과)

```
2 class A {  
    String name;  
    A(String name) {  
        this.name=name;  
    }  
}  
  
class B { //equals() 메서드 overriding  
    String name;  
    B(String name) {  
        this.name=name;  
    }  
    @Override  
    public boolean equals(Object obj) {  
        if(obj instanceof B) {  
            if (this.name == ((B)obj).name)  
                return true;  
        }  
        return false;  
    }  
}
```



```
3 public static void main(String[] ar) {  
  
    A aa1 = new A("안녕");  
    A aa2 = new A("안녕");  
  
    System.out.println(aa1==aa2);           //false  
    System.out.println(aa1.equals(aa2));    //false  
  
    B bb1 = new B("안녕");  
    B bb2 = new B("안녕");  
  
    System.out.println(bb1==bb2);           //false  
    System.out.println(bb1.equals(bb2));    //true  
}
```

false  
false  
false  
true

# 최상위 클래스 Object

3

|      |      |
|------|------|
| 1    | 2    |
| "안녕" | "방가" |

 + 

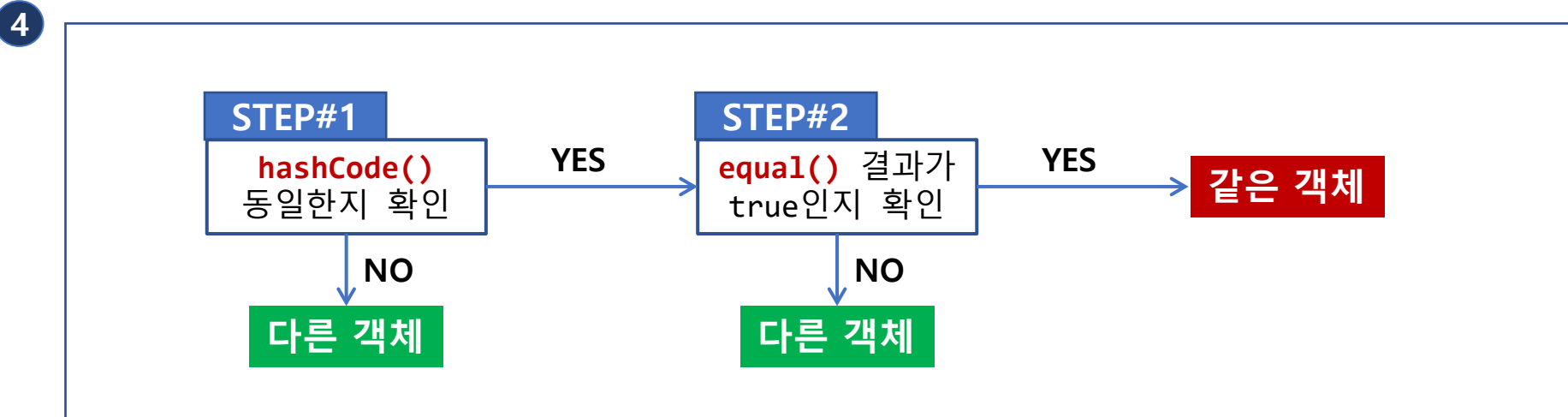
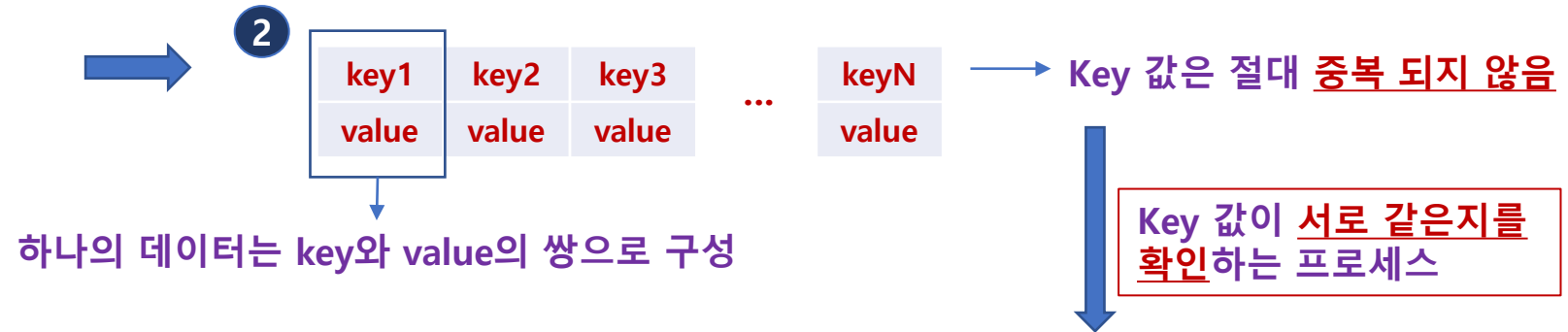
|      |
|------|
| 1    |
| "땡큐" |

 = 

|      |      |
|------|------|
| 1    | 2    |
| "땡큐" | "방가" |

- ☞ Object 메서드 : hashCode()
- 1 - 객체의 hashCode() 값 리턴. HashTable, HashMap 등의 동등비교에 사용
  - (**STEP1**. hashCode ( )) 일치 + (**STEP2**. equals( )) true → 동일객체

참고. **HashMap**



# 최상위 클래스 Object

- ☞ Object 메서드 : hashCode()
- 객체의 hashCode() 값 리턴. hashCode, hashCode 등의 동등비교에 사용
  - (**STEP1**. hashCode ( )) 일치 + (**STEP2**. equals( )) true → 동일객체

1

```
class A {  
  
    String name;  
    A(String name) {  
        this.name=name;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if(obj instanceof A) {  
            if (this.name == ((A)obj).name)  
                return true;  
        }  
        return false;  
    }  
  
    @Override  
    public String toString() {  
        return name;  
    }  
}
```

2

```
class B {  
    String name;  
    B(String name) {  
        this.name=name;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if(obj instanceof B) {  
            if (this.name == ((B)obj).name)  
                return true;  
        }  
        return false;  
    }  
  
    @Override  
    public int hashCode() {  
        return name.hashCode();  
    }  
  
    @Override  
    public String toString() {  
        return name;  
    }  
}
```

# 최상위 클래스 Object

- ☞ Object 메서드 : hashCode()
- 객체의 hashCode() 값 리턴. hashCode, equals 등의 동등비교에 사용
  - (**STEP1. hashCode( )**) 일치 + (**STEP2. equals( )**) true → 동일객체

```
public static void main(String[] ar) {
```

1

```
    HashMap<Integer, String> hm1 = new HashMap<>();  
    hm1.put(1, "데이터1");  
    hm1.put(1, "데이터2");  
    hm1.put(2, "데이터3");  
    System.out.println(hm1); //{1=데이터2, 2=데이터3}
```

2

```
    HashMap<A, String> hm2 = new HashMap<>();  
    hm2.put(new A("첫번째"), "데이터1");  
    hm2.put(new A("첫번째"), "데이터2");  
    hm2.put(new A("두번째"), "데이터3");  
    System.out.println(hm2); //{첫번째=데이터2, 두번째=데이터3, 첫번째=데이터1}
```

3

```
    HashMap<B, String> hm3 = new HashMap<>();  
    hm3.put(new B("첫번째"), "데이터1");  
    hm3.put(new B("첫번째"), "데이터2");  
    hm3.put(new B("두번째"), "데이터3");  
    System.out.println(hm3); //{첫번째=데이터2, 두번째=데이터3}
```

```
}
```

```
{1=데이터2, 2=데이터3}  
{첫번째=데이터2, 두번째=데이터3, 첫번째=데이터1}  
{첫번째=데이터2, 두번째=데이터3}
```

# The End