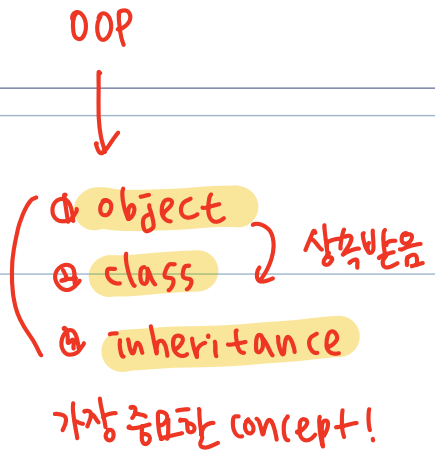
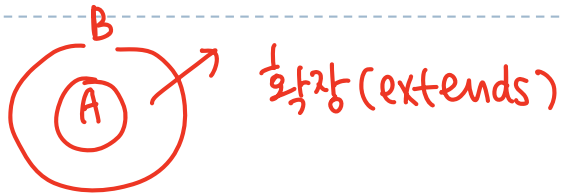
12장 객체-지향 언어: 상속



12.1 상속

상속(Inheritance)



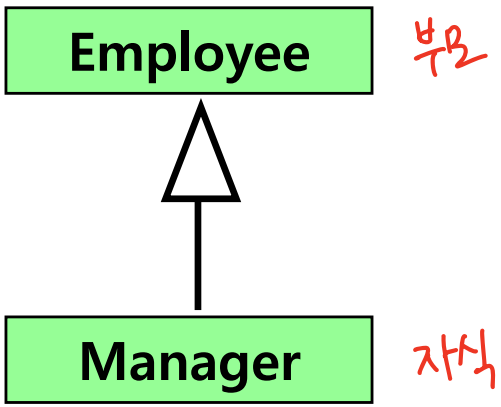
- 상속이란 무엇인가?
  - 기존의 클래스로부터 새로운 클래스를 유도하는 것
  - 자식 클래스는 부모 클래스의 필드와 메소드를 상속하고

    새로운 필드변수나 메소드를 추가할 수 있다.

  - 부모 클래스: 기반 클래스(base class), 수퍼 클래스(superclass)
  - 자식 클래스: 유도 클래스(derived class), 서브클래스(subclass)

상속

- 상속 관계 표시
  - UML 클래스 다이어그램
  - 자식으로부터 부모로의 화살표



- **is-a 관계(relationship)** → 이 때 상속받을 수 있음!
  - 자식이 부모보다 구체적인 버전이다. → cat은 animal의 속성을 모두 가짐
  - The child *is a* more specific version of the parent
  - A cat is an animal.

상속

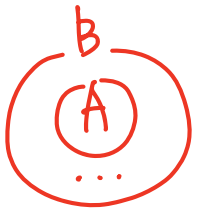
- 상속의 잇점
  - 소프트웨어 재사용(Software reuse)
  - 기존의 소프트웨어 컴포넌트를 사용해서 새로운 클래스 생성
  - 기존 소프트웨어를 재사용함으로써 기존 소프트웨어에 들인 모든 노력을 재사용

12.2 Java 상속

서브클래스 유도

- 예약어 `extends`를 사용한다.

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



- 예

```
class Manager extends Employee
{
    // 클래스 내용
}
```

- 단일 상속(single inheritance)만을 지원 (다중상속불가!)
  - 하나의 슈퍼클래스로부터 상속 받음

인터페이스끼리는 다중상속 가능
(implement) → 다중상속처럼 사용

예제 1

```
class Employee {
    private String name;
    private long salary;
    Employee(String name,
                long salary) {
        this.name = name;
        this.salary = salary;
    }
    public long pay( ) {
        return salary;
    }
}
```

```
class Manager extends Employee {
    private long bonus;
    Manager(String name, long salary,
                long bonus) {
        super(name, salary);
        this.bonus = bonus;
    }
    public long pay( ) {
        return super.pay() + bonus;
    }
    public long getBonus() {
        return bonus;
    }
}
```

super

- super
  - 슈퍼클래스 즉 부모 클래스를 지칭하는 참조이다.

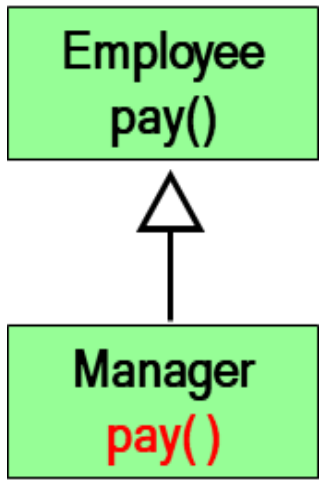
- super 사용

super 사용	설명
super()	슈퍼클래스의 생성자를 호출한다
super.메소드이름()	슈퍼클래스 내의 해당 메소드를 호출한다
super.필드이름	슈퍼클래스 내의 해당 필드를 나타낸다.

메소드 재정의overriding

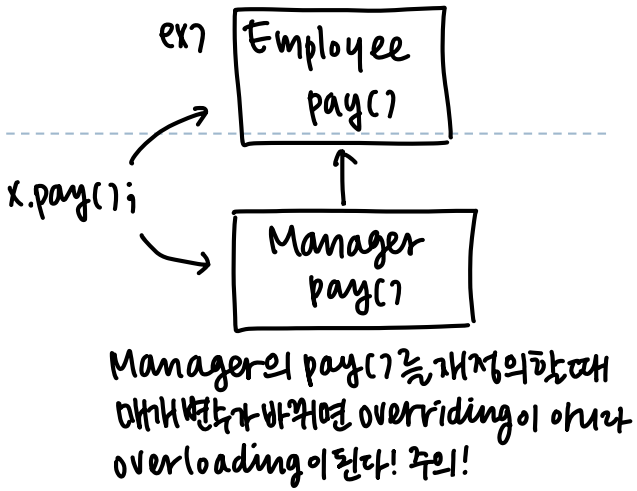
- 메소드 재정의란 무엇인가?
  - 자식 클래스가 상속된 메소드를 자신이 원하는 대로 재정의하는 것
  - 새로운 메소드는 부모 메소드와 이름과 서명(signature)이 같아야 한다.
- 메소드의 서명(signature)
  - 메소드의 매개변수 이름, 개수, 순서, 타입
- 재정의된 메소드 실행
  - 그 메소드를 실행하는 객체의 타입에 따라
  - 호출될 메소드가 결정된다.

(매개변수)



이름이 가리키는대상(바인딩)에 따라호출됨
ex) x.pay(); 의 x에 따라

중복 정의 vs. 재정의



- 중복정의(overloading)
  - 한 클래스 내에 같은 이름의 여러 개의 메소드
  - 서로 다른 서명을 갖는 경우
  - 비슷한 연산을 다른 매개변수에 대해서 다른 방식으로 정의하는데 사용
- 재정의(overriding)
  - 부모 클래스의 메소드를 자식 클래스가 재정의
  - 서명이 같아야 한다.
  - 서명이 다르다면 어떻게 될까?

예제 2

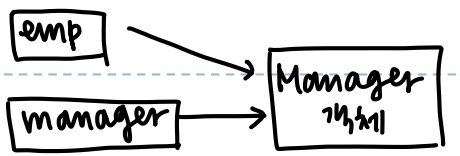
```
public class EmployeeMain
{
    public static void main(String args[]) {
        Employee emp = new Employee("kim", 400);
        System.out.println(emp.pay());                    (1)
        Manager manager = new Manager("lee", 500, 200);
        System.out.println(manager.getBonus());           (2)
        System.out.println(manager.pay());                (3)
        if (...) emp = manager;                           (4)
        System.out.println(emp.pay());                    (5)
    }
}
```

(4) assign할 수 있을까?

(5) assign한다면 무엇이 출력될까?

* is-a relationship에 의해
  emp = manager ; → 가능
  manager = emp ; → 불가능

p14 다형성 확인

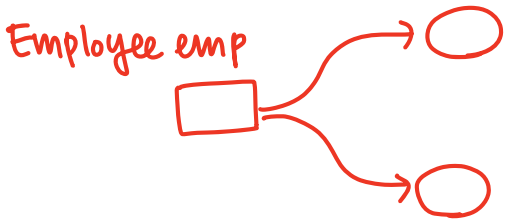


[실행 결과]

400
200
700

다형성Polymorphism

- 다형성(polymorphism)의 의미
  - 많은 형태를 갖는다.
  - "having many forms“



- 다형 참조(polymorphic reference)
  - 참조변수는 선언된 클래스의 객체뿐만 아니라
  - 그 클래스의 자손 클래스의 객체도 참조할 수 있다.

ex) Employee객체 emp는 Manager의 객체도 참조할수있음

- 동적 바인딩
  - 다형 참조를 통해 재정의된 메소드를 호출할 때
  - 호출될 메소드는 참조한 객체에 따라 결정된다.

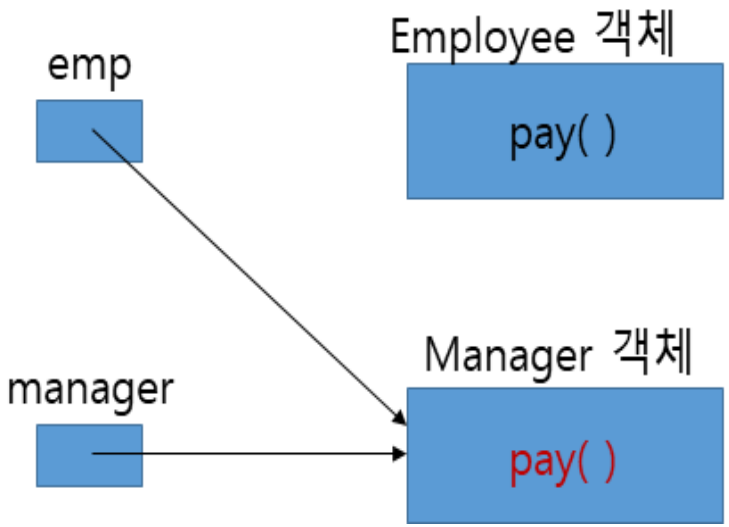
다형참조

- 객체 참조 변수
  - 선언된 클래스의 객체와
  - 선언된 클래스의 자손 클래스의 객체를 참조할 수 있다

- 예:

```
emp= new Employee();
if(…) emp = manager;
emp.pay( );
```

true / false에 따라 Employee의 pay(); 또는 Manager의 pay();가 실행됨!
(전형적인 동적바인딩)



다형성에서는 type casting 필요X

동적 바인딩

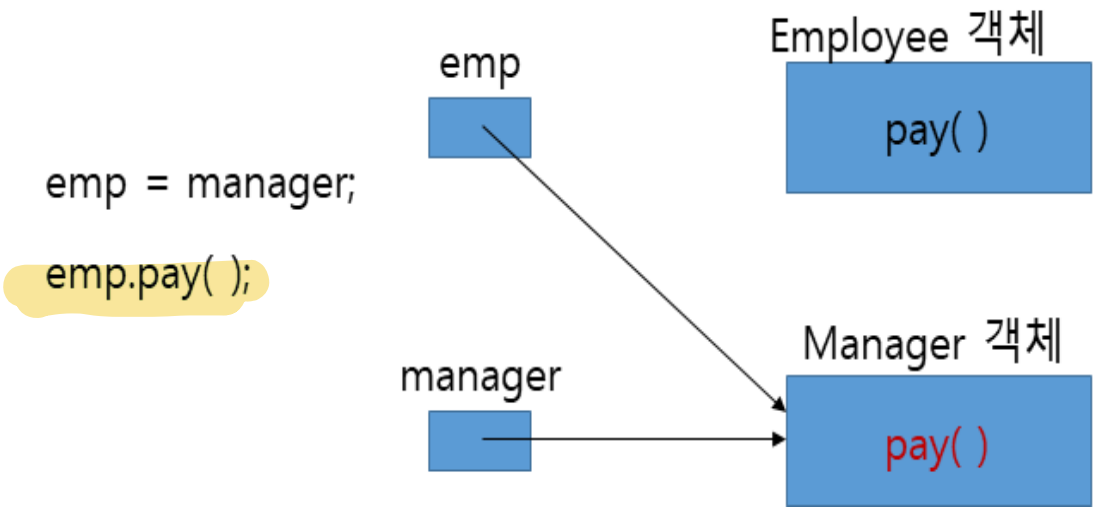
- 바인딩이란 무엇인가?
  - 이름이 가리키는 대상을 결정하는 것

- `emp.pay()`
  - 이 호출은 호출될 메소드를 결정 즉 바인딩 한다.
  - 컴파일 시간에 결정하면 언제나 같은 메소드가 호출될 것이다.
  - 그러나 Java는 동적 바인딩을 한다

- 동적 바인딩(dynamic binding)
  - 호출될 메소드는 참조 변수의 타입이 아니고
  - 실행시간에 참조되고 있는 객체의 타입에 의해 결정된다.

동적 바인딩

- emp.pay()
  - emp가 Employee 객체를 참조하는 경우
    - Employee의 pay() 호출
  - Emp가 Manager 객체를 참조하는 경우
    - Manager의 pay() 호출



타입 캐스팅

class와 연관된 객체 : 인스턴스

- 타입 캐스팅에 필요한 예

```
void m(Employee a) {
    Manager b;
    if (a instanceof Manager)
        Manager b = (Manager) a;
        b.getBonus( );
}
```

← a가 Manager객체면?

⇒ type casting

↳ 만약에 if문없이 이것만 있으면 a가 Manager를 가리키지 않는경우 error 발생!

- 타입 캐스팅

  (클래스명) 객체참조

- 타입 캐스팅 오류

```
Employee a = new Employee();

Manager b = (Manager) a; // 오류
```

↳ a가 Manager를 가리키고있지않음

12.3 상속과 접근제어

접근 조정자(access modifier)

- 전용(private) 가시성
  - 정의된 클래스 내에서만 접근 가능하고
  - 자식 클래스 내에서 직접 접근할 수 없다.
- 공용(public) 가시성
  - 자식 클래스뿐만 아니라 프로그램 내 어디서나 접근 가능
  - 공용 변수는 캡슐화 원리를 위반
- protected 가시성
  - 상속을 위한 제3의 가시성으로
  - 공용 가시성보다는 더 캡슐화하고 전용보다는 덜 캡슐화 한다.
  - 정의된 클래스 내와 그 자손 클래스에서 접근할 수 있다.

상속과 접근 조정자

- Java 상속과 가시성

부모클래스	public	protected	private
자식클래스	public	protected	접근 불가

- 예
  - 부모 클래스로부터 상속받은 salary 필드변수 직접 사용 불가

간접사용

```
public int pay() {
    return super.pay() + bonus;
}
```

12.4 추상 클래스

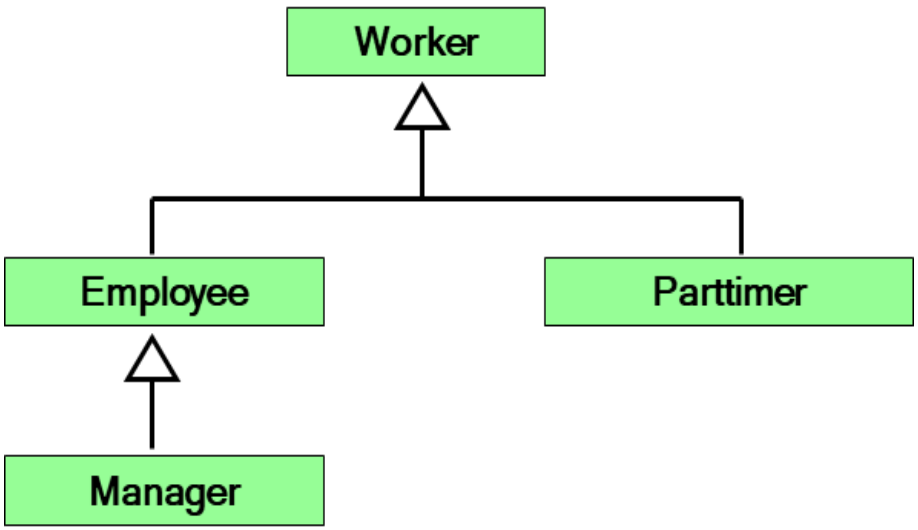
추상 클래스Abstract Class

- 추상 클래스란 무엇인가?
  - 포괄적인 개념을 표현하기 위한 클래스로 아직 덜 구현된 클래스
  - 추상 메소드를 포함한 클래스를 보통 추상 클래스로 정의한다.
  - 추상 메소드가 아닌 완전히 구현된 메소드도 포함 가능
  - 실체화될 수 없다!
- abstract 조정자 사용

```
abstract class 클래스이름 {
        ...
    abstract 리턴타입 메소드이름(매개변수선언);  헤더만있음!
        ...
}
```

추상 클래스: 예

```
abstract class Worker {
    protected String name;
    abstract public long pay();
    Worker(String name) {
        this.name = name;
    }
}
```



추상 클래스

- 자식 클래스가 부모 클래스의 추상 메소드를 구현한다.
  - 구현하지 않으면 자식 클래스도 여전히 추상 클래스가 된다.
  - 추상 메소드는 `final`이나 `static`으로 선언하면 안됨.

- 추상 클래스 용도
  - 클래스 계층구조에서 실체화하기에 너무 포괄적인 공통 요소들을
  - 계층구조에 위치시킬 수 있도록 해준다.

* 인터페이스와의 차이점!
클래스 안에 이러이러한 것이 필요하다고 명세 → 인터페이스
추상클래스도 클래스(구현한것)

Employee

```
class Employee extends Worker {
    private long salary;
    Employee(String name, String id, long salary) {
        super(name);
        this.salary = salary;
    }
    public long pay() {
        return salary;
    }
}
```

Parttimer

```
class Parttimer extends Worker {
    private int hours;
    private int rate;
    Parttimer(String name, int rate) {
      super(name);
       this.rate = rate;
       this.hours = 0;
    }

    public int addHours(int hours) {
       this.hours += hours;
       return this.hours;
    }

    public long pay() {
        return hours * rate;
    }
}
```

MainWorker

```
public class MainWorker { // 직원과 시간제를 생성하여 사용
    public static void main(String args[]) {
        Employee emp = new Employee("홍길동", 2000000);
        System.out.println(emp.name + " 급여: " + emp.pay());
        Parttimer part = new Parttimer("나시간", 10000);
        System.out.println("일한 시간: " + part.addHours(30));
        System.out.println(part.name + " 급여: " + part.pay());
    }
}
```

12.5 C++ 상속

C++ : public 상속

- 구문

```
class <derived> : public <base>
{
    <멤버 선언>
}
```

- 자식 클래스에서 가시성

부모클래스	public	protected	private
자식클래스	public	protected	접근 불가

[예제 5]

```
#include <iostream>
#include <string>
using namespace std;

class Employee {
private:
    string name;
    long salary;
public:
    virtual long pay() { return salary;}
    Employee(string name, long salary) {
        this->name = name;
        this->salary = salary;
    }
};
```

[예제 5]

```
class Manager: public Employee { // Employee 상속한 Manager 클래스
private:
    long bonus; // 추가된 데이터 멤버
public:
    Manager(string name,long salary, long bonus):Employee(name,salary) {
        this->bonus = bonus;
    }

    long pay() {              // 재정의된 멤버 함수
         return Employee::pay() + bonus;
    }

    long getBonus() { return bonus; } // 추가된 멤버 함수
};
```

C++ : private 상속

- 상속된 멤버는 자식클래스에서 전용(private)이 된다.

```
class <derived> : private <base>
{
    <멤버 선언>
}
```

- 상속된 멤버의 가시성

부모클래스	public	protected	private
자식클래스	private	private	접근 불가

가상 함수

- C++의 가상 함수(virtual function)
  - virtual로 선언된 함수로 자식클래스에서 재정의될 수 있는 함수
  - Java 메소드는 virtual !!
- C++의 순수 가상 함수(pure virtual function)
  - Java의 추상 메소드(abstract method)
  - 자식클래스에서 정의되는 함수
- 가상 함수 호출
  - 객체에 타입에 따라 동적 바인딩 된다.

[예제 6]

```
int main(void) {
    Employee *emp = new Employee("kim", 2000000);
    cout << emp->pay() << endl;
    Manager *man = new Manager("lee", 3000000, 1000000);
    emp = man;
    cout << emp->pay() << endl;
}
```

C++ 다중 상속

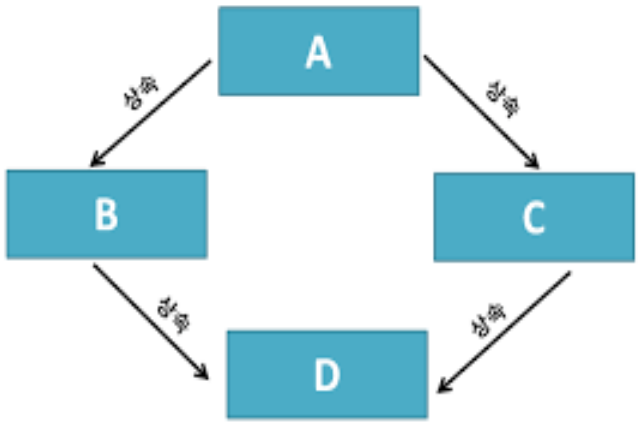
```
class Person {
    public: void sleep() { printf("잠을 잡니다.₩n"); }
}
class Student: public Person {
    public: void study() { printf("공부합니다.₩n"); }
}
class Worker: public Person {
    public: void work() { printf("일합니다.₩n"); }
}
# 다중 상속
class Arbeit: public Student, public Worker {
     public: void myjob() {
                printf("나는 알바 학생입니다: ₩n");
                study();
                work();
     }
} ;
```

유효범위 해결 연산자

- myjob 함수 내에서 sleep 함수를 호출하면 어떻게 될까?
  - Student와 Worker를 통해서 두 개의 sleep 함수를 상속 받음.
  - sleep 함수를 호출하면 컴파일 오류

- 유효범위 해결 연산자(scope resolution operator)
  - Student::sleep() 혹은 Worker::sleep()와 같이 호출

- 다중상속의 죽음의 다이아몬드 문제



12.6 Python 상속

Python 상속

- class 자식클래스(부모클래스):

```
class Employee:   # 일반 직원 정의
    def __init__(self, name, salary):
        self.name = name;
        self.salary = salary;
    def pay(self):
        return self.salary
```

```
class Manager(Employee):   # 관리자 정의
    def __init__(self, name, salary, bonus):
        Employee.__init__(self, name, salary)
        self.bonus = bonus
    def pay(self):  # 보너스 추가하여 급여 계산
        return self.salary + self.bonus
    def getBonus():
        return self.bonus
```

Python 다중 상속

- class 자식클래스(부모클래스1,...,부모클래스N):

```
class Person:
    def sleep(self):
        print('잠을 잡니다.')

class Student(Person):
    def study(self):
        print('공부합니다.')
    def play(self):
        print('친구와 놉니다.')

class Worker(Person):
    def work(self):
        print('일합니다.')
    def play(self):
        print('술을 마십니다.')
```

```
# 다중 상속
class Arbeit(Student, Worker):
    def myjob(self):
        print('나는 알바입니다:')
        self.sleep()
        self.play()
        self.study()
        self.work()

>>> a = Arbeit( )
>>> a.myjob( )
    나는 알바입니다:
    잠을 잡니다.
    친구와 놉니다.
    공부합니다.
    일합니다.
```

다이아몬드 문제

- 다중상속에서 메소드 충돌 문제를 어떻게 해결할까?
  - 메소드 탐색 순서(Method Resolution Order, MRO)에 따라 선택

- 다중상속 class A(B, C):
  - 부모 클래스 중 B에서 C 순서로 메소드를 찾는다.
  - Arbeit 내에서 play 메소드 호출 → Student의 play 메소드 호출

- 메소드 탐색 순서 확인

  `클래스.mro()`

```
>>> Arbeit.mro()
[<class '__main__.Arbeit'>, <class '__main__.Student'>,
 <class '__main__.Worker'>, <class '__main__.Person'>, <class 'object'>]
```

12.7 구현

객체 구현

- 객체는 구조체(레코드)처럼 메모리가 할당된다.
  - 각 실체 변수에 대한 메모리 할당.

- 동적 바인딩
  - 가상 메소드 테이블(Virtual method table) 이용
  - 각 객체는 이 테이블에 대한 포인터를 갖는다.

- 접근 가능성 검사
  - 접근 가능성 검사는 컴파일 시간에 이루어진다.

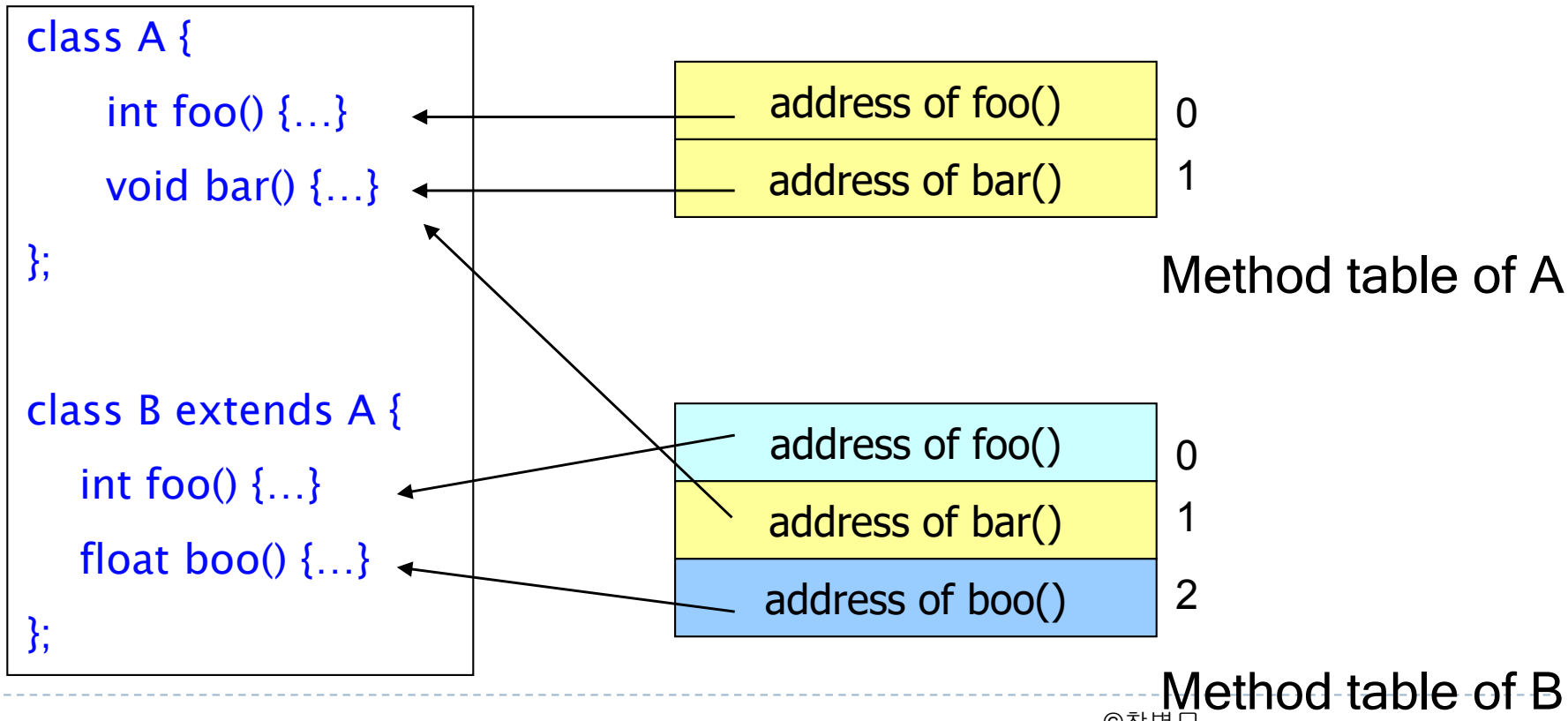
가상 메소드 테이블Virtual Method Table

- 메소드 테이블
  - 각 클래스마다 메소드 테이블이 하나씩 있다.
  - 클래스의 모든 가상 메소드는 하나의 인덱스를 갖는다.
  - 각 인덱스의 내용은 해당 메소드 코드의 주소

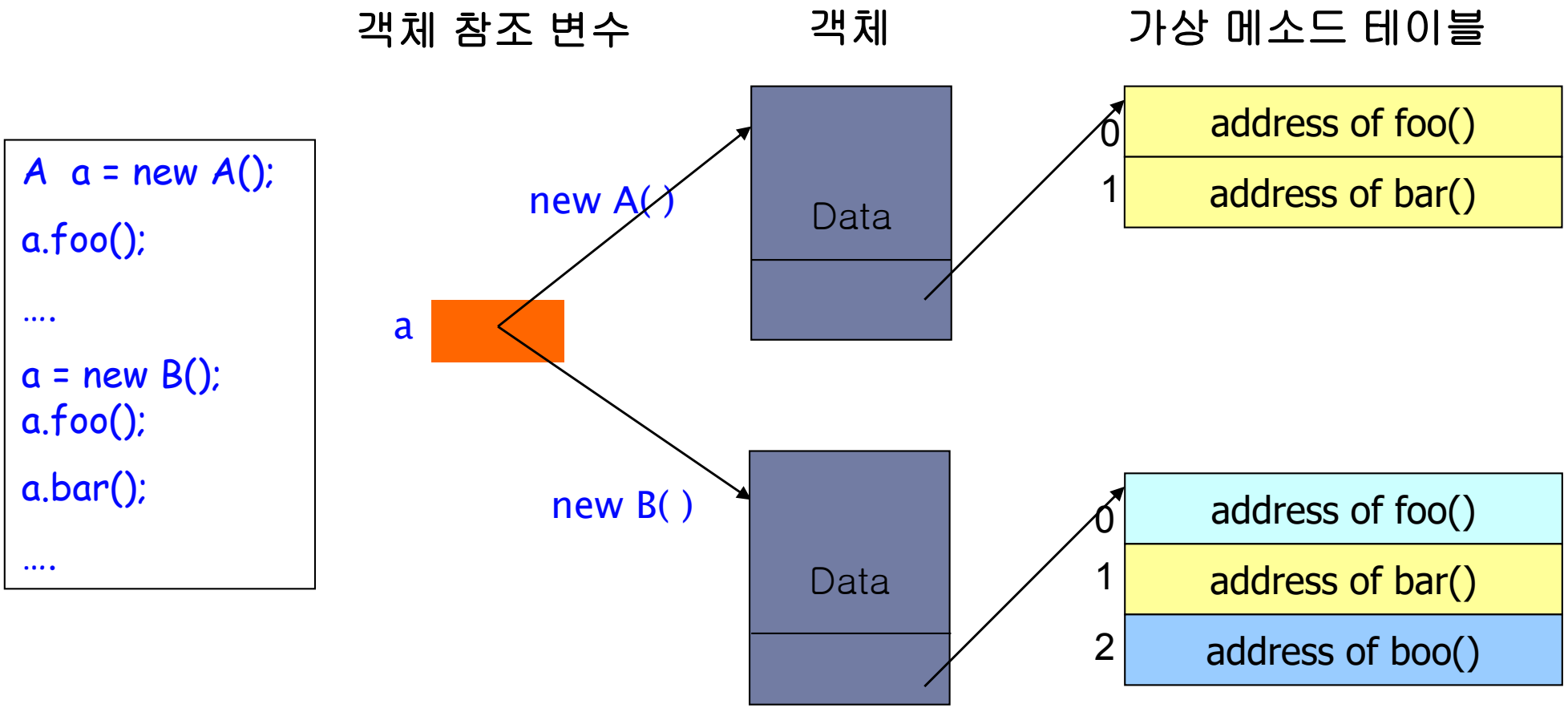
- 메소드 호출 구현
  - 대상 객체의 메소드 테이블 포인터를 따라간다.
  - 해당 인덱스의 메소드 주소를 따라간다.
  - 그 주소로 점프한다.

가상 메소드 테이블

- 상속과 재정의
  - 서브클래스는 수퍼클래스의 메소드 테이블을 상속받는다.
  - 메소드가 재정의되면, 해당 메소드 테이블을 갱신하다.



메소드 호출 예


객체 참조 변수
객체
가상 메소드 테이블
A a = new A();
a.foo();
….
a = new B();
a.foo();
a.bar();
….
a
new A( )
new B( )
Data
Data
0
1
address of foo()
address of bar()
0
1
2
address of foo()
address of bar()
address of boo()

메쏘드 호출 구현

- 객체로부터 메쏘드 테이블 포인터를 얻는다.
- 메쏘드 아이디를 이용하여 해당 메쏘드의 주소를 얻는다.
- 그 주소로 점프한다.