

06 상속과 다형성

Sangwook Lee

06 상속과 다형성

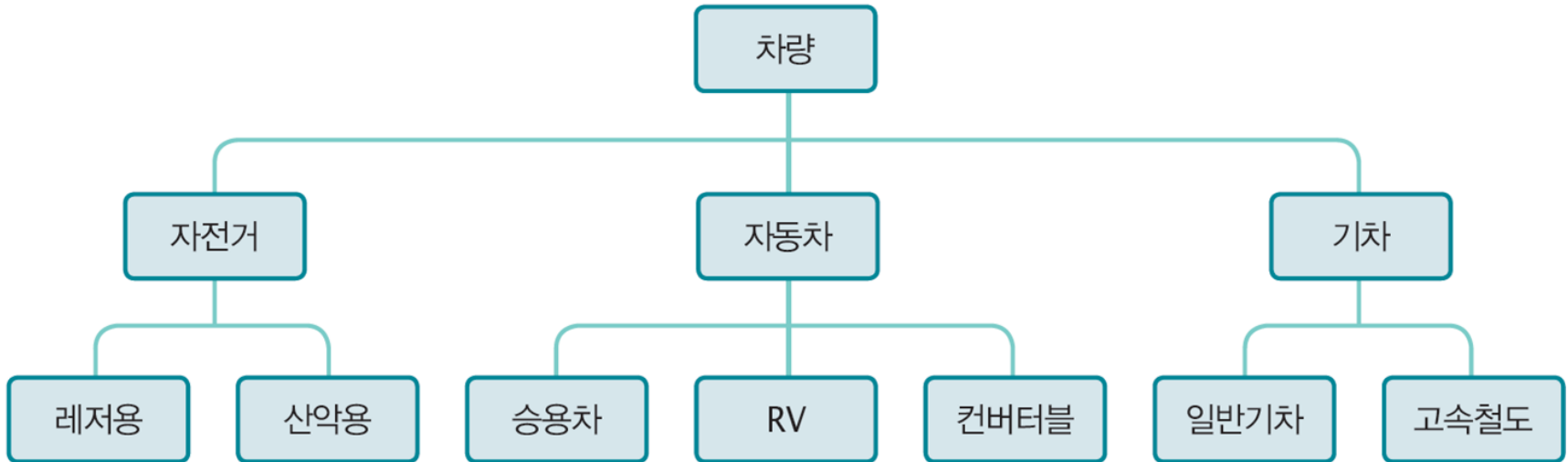
1. 상속

2. 다형성

3. 추상 클래스와 인터페이스

1. 상속

- 차량의 분류



하위 객체는 상위 객체의 특성을 모두 가짐!

1. 상속

- 상속(inheritance)

- 어떤 클래스의 필드와 메소드를 모두 물려받는 것

물려준 클래스

- = 상위 클래스
- = 슈퍼 클래스
- = 부모 클래스
- = 기본 클래스



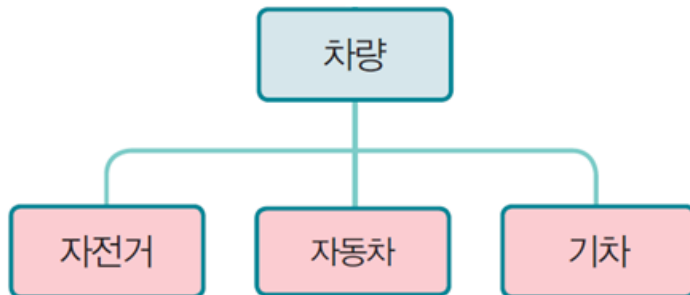
물려받은 클래스

- = 하위 클래스
- = 서브 클래스
- = 자식 클래스
- = 유도 클래스

1. 상속

- 상속의 구현
 - 하위 클래스 정의 방법

하위클래스명 extends 상위클래스명



```
public class Vehicle {  
  
}
```

```
public class Cycle extends Vehicle {  
  
}
```

```
public class Motor extends Vehicle {  
  
}
```

```
public class Train extends Vehicle {  
  
}
```

1. 상속

- 상속의 구현

- 하위 클래스 필드 선언

- 상위 클래스에서 선언된 필드는 다시 선언할 필요 없음
 - 상위 클래스에서 선언되지 않은 필드는 추가
 - 필요에 따라 상위 클래스에 선언된 필드 재정의 가능

```
public class Vehicle {  
    public String name = "차량";  
    public double maxSpeed;  
    public int seater;  
}
```



```
public class Motor extends Vehicle {  
    public String name = "자동차";  
    public int displacement;  
    ...  
}
```

상위 클래스의 *name*과
하위 클래스의 *name*의
구분은?

1. 상속

- 하위 클래스(자신)와 상위 클래스 필드의 구분
 - 하위 클래스(자신) 필드 참조 방법

this . 필드명 or 필드명

- 상위 클래스 필드 참조 방법

재정의의 필드가 없을 경우 가능

super . 필드명 or 필드명 or this . 필드명

*this*와 *super* 모두 비정적 필드라 메소드 참조 시만 사용 가능!

1. 상속

- 자신과 상위 클래스 필드의 구분
 - this와 super의 사용

```
public class Vehicle {  
    public String name = "차량";  
    public double maxSpeed;  
    public int seater;  
}
```

```
public class Motor extends Vehicle {  
    public String name = "자동차";  
    public int displacement;  
  
    public void printInfo() {  
        System.out.println(super.name + ": " + this.name);  
        System.out.println("최대속도: " + maxSpeed + " km");  
        System.out.println("정원: " + seater + " 명");  
        System.out.println("배기량: " + displacement + " cc");  
    }  
}
```

1. 상속

- 상위 클래스 생성자 호출
 - 상위 클래스 생성자 호출 메소드: `super()`
 - ☞ 명시하지 않아도 자동 호출

```
public class Vehicle {  
    public String name = "차량";  
    public double maxSpeed;  
    public int seater;  
    public Vehicle() {  
    }  
}
```



```
public class Motor extends Vehicle {  
    public String name = "자동차";  
    public int displacement;  
    public Motor() {  
        super();  
    }  
}
```

1. 상속

- 상위 클래스 생성자 호출

- 생성자 호출 시 항상 상위 객체의 필드와 메소드가 먼저 생성



1. 상속

- 상위 클래스 생성자 호출

- 하위 클래스의 사용자 구현 생성자의 첫 줄에 `super()`가 없으면 자동으로 `super()` 호출

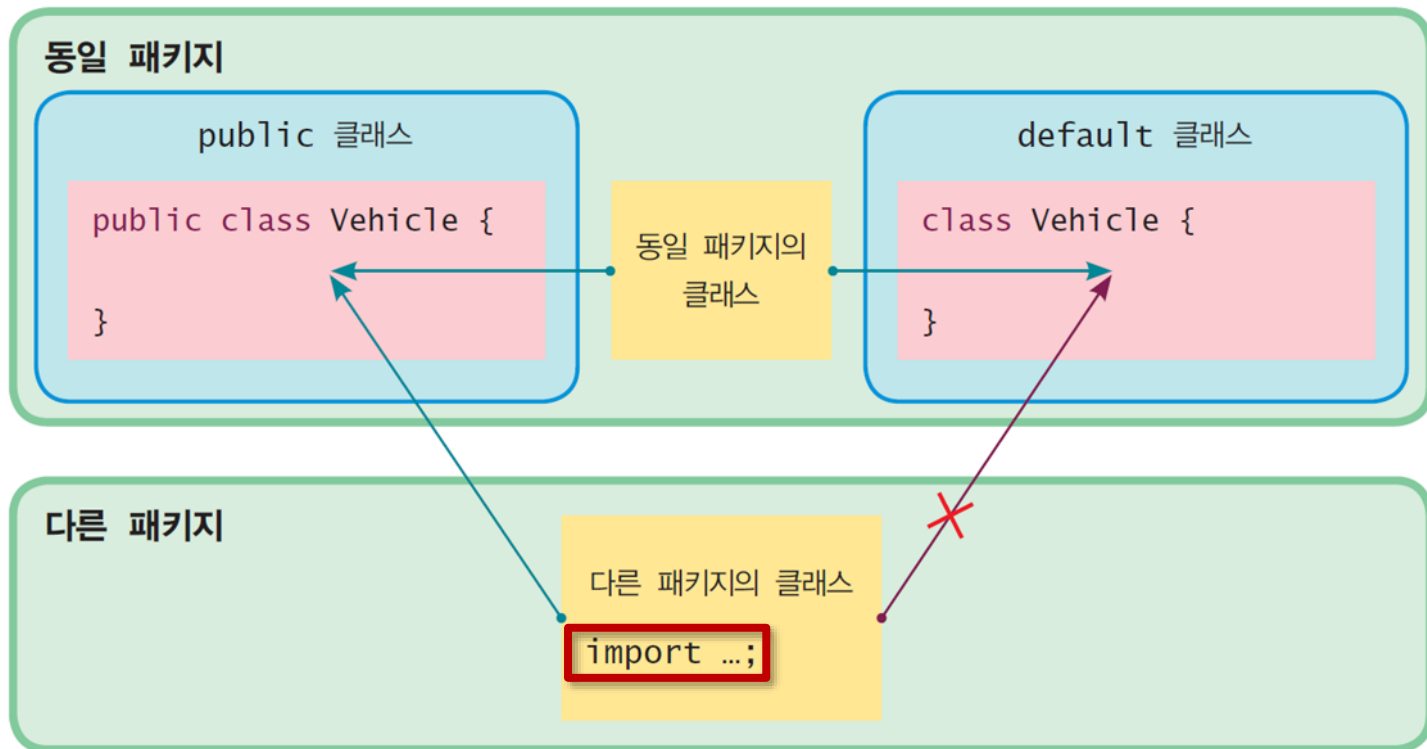
```
public class Motor extends Vehicle {  
    ...  
  
    public Motor(double maxSpeed, int seater, int displacement) {  
        //자동으로 super() 호출되어  
        this.maxSpeed = maxSpeed;  
        this.seater = seater;  
        this.displacement = displacement;  
    }  
}
```

이 경우, 상위 클래스에 사용자가 정의한 인자가 있는 생성자만 있으면 오류 발생!

1. 상속

- 클래스 접근 지정자

- public: 모든 클래스에서 접근
- default: 동일 패키지 클래스만 참조



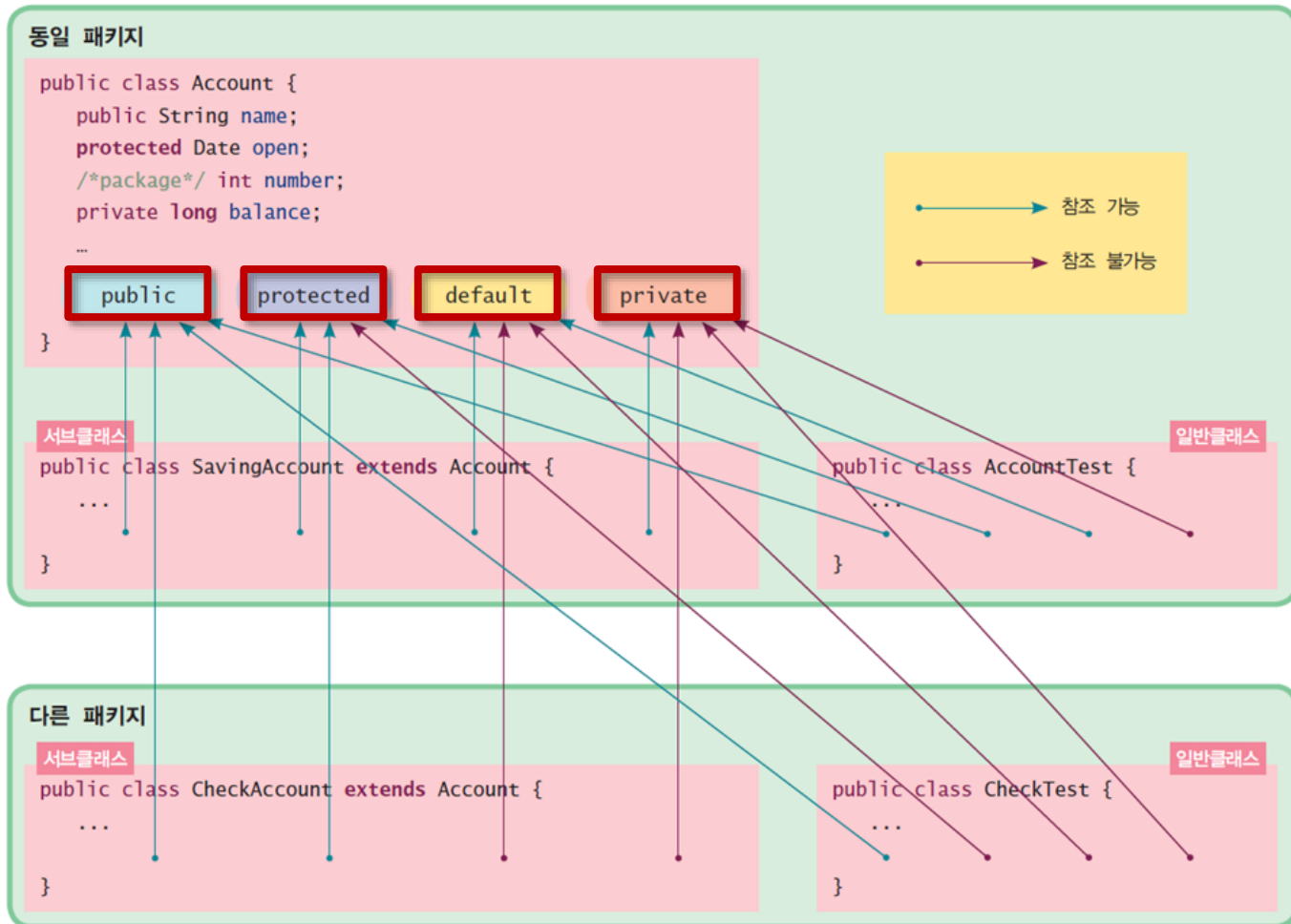
1. 상속

- 필드와 메소드 접근 지정자
 - 참조 범위의 차이

공개적 순위	키워드	클래스 내부	동일 패키지		다른 패키지	
			하위 클래스	일반클래스	하위 클래스	일반클래스
1	public	O	O	O	O	O
2	protected	O	O	O	O	X
3	default	O	O	O	X	X
4	private	O	X	X	X	X

1. 상속

• 필드와 메소드 접근 지정자



<접근 지정자별 참조 예>

1. 상속

[예제 6-9] 다른 패키지의 클래스를 상위로 하는 하위 클래스

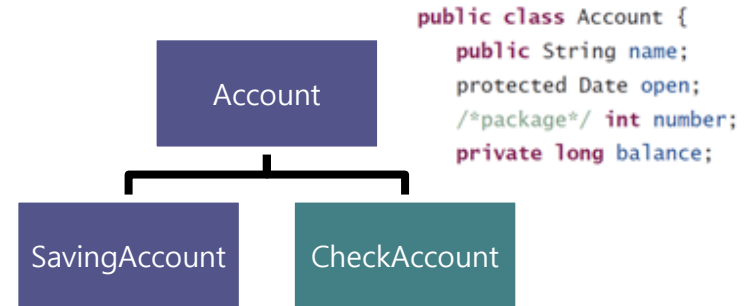
```
package inheritance.control;  
  
import inheritance.access.Account;  
import inheritance.access.SavingAccount;
```

```
public class CheckAccount extends Account {  
    long minimum;    //최소 잔액 금액
```

```
    public CheckAccount(String name, int number, long minimum) {  
        super(name, number);  
        this.minimum = minimum;  
        super.deposit(minimum);    //deposit(minimum);으로도 가능  
    }  
}
```

```
public static void main(String[] args) {  
    SavingAccount myAccount = new SavingAccount("김태희", 234567654, 0.34);  
    myAccount.deposit(250000);  
    myAccount.withdraw(50000);  
    myAccount.checkBalance();  
}
```

```
System.out.println(myAccount.name);    //접근 가능  
//System.out.println(myAccount.open);    //접근 불가  
//System.out.println(myAccount.number);    //접근 불가  
//System.out.println(myAccount.balance);    //접근 불가
```



*protected 필드인 open을
참조할 수 없는 이유는?*

*다른 패키지에 속해 있으면서,
상속 관계가 없는 SavingAccount
객체의 필드기 때문!*

06 상속과 다형성

1. 상속

2. 다형성

3. 추상 클래스와 인터페이스

2. 다형성

- 다형성(polymorphism)
 - 같은 이름의 메소드가 다른 역할을 하는 것
 - 다형성을 적용하는 기법
 - 오버라이딩(overriding)
 - 오버로딩(overloading)

2. 다형성

- 오버라이딩(overriding)

- 정의

- 상위 클래스의 메소드를 하위 클래스에서 다시 정의하는 것

- 목적

- 상위 클래스의 메소드와 동일한 이름, 인자, 반환형으로 메소드를 재정의하여 상위 클래스의 메소드를 오버라이드하기 위함

- 동의어

- 메소드 재정의(redefinition)
 - 메소드 대체(replacement)

2. 다형성

- 오버라이딩

- 조건

- 반환 값, 이름, 매개변수 동일
 - final, private 메소드는 오버라이딩 불가

- final

- » 클래스: 상속 불가

- » 메소드: 오버라이딩 불가

```
public final class Staff extends Faculty {  
    public String division;  
  
    public final void printInfo() {  
        ...  
    }  
}
```

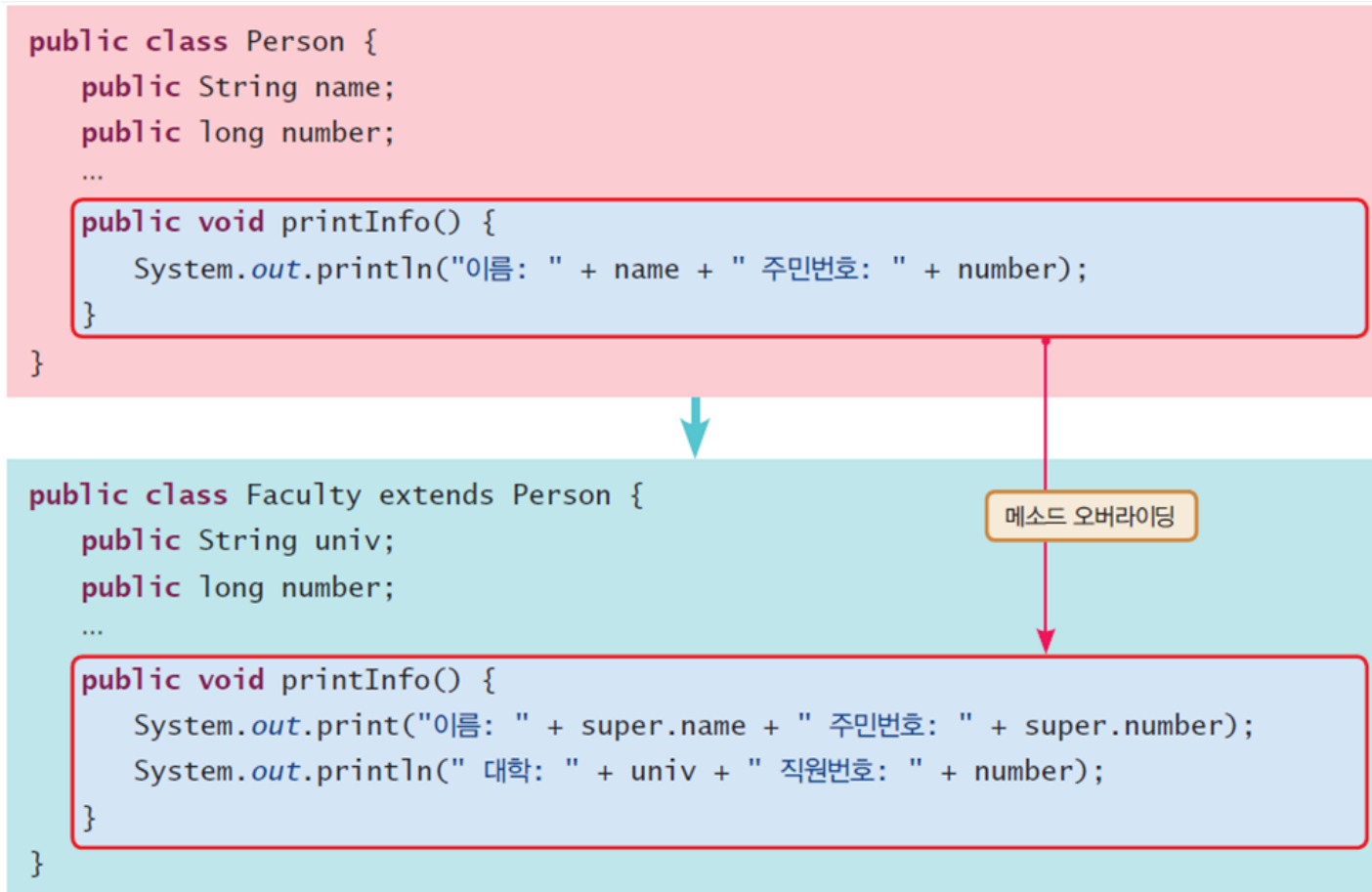
클래스 Staff는 더 이상 다른 클래스의 상위 클래스가 될 수 없도록 한다.

메소드 printInfo()는 더 이상 하위 클래스에서 오버라이딩될 수 없도록 한다.

2. 다형성

- 오버라이딩

— 예



2. 다형성

[예제 6-20] 오버라이딩 메소드 호출

```
public final class Staff extends Faculty {
    public String division;

    public Staff(String name, long number, String univ, long idNumber, String division) {
        super(name, number, univ, idNumber);
        this.division = division;
    }

    //public final void printInfo() {
    public void printInfo() {
        System.out.print("이름: " + super.name + " 주민번호: " + super.getSNumber());
        System.out.print(" 대학: " + univ + " 직원번호: " + super.number);
        System.out.println(" 부서: " + division);
    }

    public void printFacultyInfo() {
        super.printInfo();
    }

    public static void main(String[] args) {
        Person she = new Person("이소라", 2056432);
        she.printInfo();

        Faculty i = new Faculty("김영태", 1145782, "연한대학교", 38764);
        i.printInfo();

        Staff he = new Staff("최영기", 1167429, "남도대학교", 1287, "기획처");
        he.printInfo();
        he.printFacultyInfo();
    }
}
```

출력 결과는? (p.219 참조)

2. 다형성

- 동적 바인딩(dynamic binding)

- 업 캐스팅 또는 다운 캐스팅 시, 오버라이딩 메소드를 호출하면 하위 객체에 구현된 메소드가 실행되는 것

- 업 캐스팅: 하위 객체를 상위 클래스형 변수에 대입
 - 다운 캐스팅: 상위 객체를 하위 클래스형 변수에 대입

- 하위 클래스로의 명시적 형 변환 필요

```
Person he = new Staff("최영기", 1167429, "남도대학교", 1287, "기획처");
he.printInfo();
Faculty f = (Faculty) he;
f.printInfo();
Staff s = (Staff) he;
s.printInfo();
```

2. 다형성

[예제 6-21] 동적 바인딩

```
package inheritance.overriding;

public class Overriding {
    public static void main(String[] args) {
        Person she = new Person("이소라", 2056432);
        she.printInfo();

        Person i = new Faculty("김영태", 1145782, "연한대학교", 38764);
        i.printInfo();

        Person he = new Staff("최영기", 1167429, "남도대학교", 1287, "기획처");
        he.printInfo();
        Faculty f = (Faculty) he;
        f.printInfo();
        Staff s = (Staff) he;
        s.printInfo();
    }
}
```

이름: 이소라 주민번호: 2056432

이름: 김영태 주민번호: 1145782 대학: 연한대학교 직원번호: 38764

이름: 최영기 주민번호: 1167429 대학: 남도대학교 직원번호: 1287 부서: 기획처

이름: 최영기 주민번호: 1167429 대학: 남도대학교 직원번호: 1287 부서: 기획처

이름: 최영기 주민번호: 1167429 대학: 남도대학교 직원번호: 1287 부서: 기획처

2. 다형성

- 오버로딩(overloading)

- 정의

- 한 클래스에 동일 이름의 메소드를 여러 개 정의하는 것
 - 인자 수 또는 인자 타입은 달라야 함

- 목적

- 유사한 기능을 가지고 있지만 인자가 다른 메소드들의 이름을 동일하게 정의하여 메소드 관리를 편리하게 하기 위함

- 동의어

- 메소드 다중 정의
 - 메소드 중복 정의

2. 다형성

- 메소드 오버로딩

```
package inheritance.overloading;

public class Sum {
    public int add(int a, int b) {
        return a+b;
    }
    public int add(int a, int b, int c) {
        return a+b+c;
    }
    public int add(int a[], int n) {
        int sum = 0;
        for (int i=0; i<n; i++)
            sum += a[i];
        return sum;
    }

    public static void main(String[] args) {
        Sum adder = new Sum();
        System.out.println(adder.add(3, 6));
        System.out.println(adder.add(3, 6, 9));
        System.out.println(adder.add(new int[] {3, 6, 9, 12}, 4));
    }
}
```



이름은 add()로 같지만 실인자의 자료형으로 호출할 메소드를 구분할 수 있다.

06 상속과 다형성

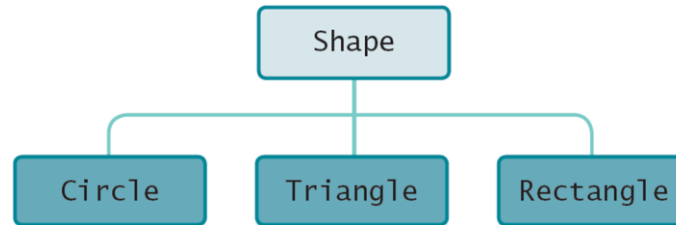
1. 상속

2. 다형성

3. 추상 클래스와 인터페이스

3. 추상 클래스와 인터페이스

- 추상 클래스(abstract class)
 - 하나 이상의 추상 메소드를 가지는 클래스
 - ☞ 일반 메소드도 가질 수 있음
 - 특징
 - 객체화 불가
 - 선언부에 `final`이 올 수 없음



*Shape*를 추상 클래스로 만들면 적절한 이유는?

현실 세계에 *Shape*라는 사물이 없기 때문!

3. 추상 클래스와 인터페이스

- 추상 클래스
 - 만드는 방법

```
abstract class 클래스명
```

- 선언 예

```
public abstract class Shape {  
    protected double x, y;  
  
    public Shape(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```

```
Shape s = new Shape(3.1, 4.5); //객체화 오류  
cannot instantiate the type Shape
```

3. 추상 클래스와 인터페이스

- 추상 메소드(abstract method)
 - 선언부만 있는 메소드
 - 목적
 - 추상 클래스를 상속받아 기능을 확장시키기 위함
 - 특징
 - 메소드 오버라이딩으로 완성
 - 하위 클래스에서 완성되지 않으면 오류 발생
 - `private`와 `final`이 올 수 없음
 - 추상 메소드를 가진 클래스는 반드시 추상 클래스로 지정해야 됨

3. 추상 클래스와 인터페이스

- 추상 메소드
 - 만드는 방법

```
abstract 반환형 메소드명();
```

- 선언 예

클래스 Shape는 추상 메소드 draw()를 가지므로 반드시 추상 클래스여야 한다.

```
public abstract class Shape {  
    protected double x, y;  
    ...  
    public void drawCenter() {           //일반 클래스  
        System.out.println("(x, y) = " + x + ", " + y);  
    }  
  
    public abstract void draw();         //추상 클래스  
}
```

추상 메소드 draw()는 몸체가 구현되지 않은 메소드이다.

3. 추상 클래스와 인터페이스

- 추상 메소드

- 추상 메소드 구현 예

```
public abstract class Shape {  
    protected double x, y;  
    ...  
    public void drawCenter() {           //일반 클래스  
        System.out.println("중심좌표 (x, y) = " + x + ", " + y);  
    }  
    public abstract void draw();         //추상 클래스  
}
```

하위 클래스에서 추상 메소드를 반드시 구현해야 한다. 아니면 다음과 같은 문법 오류가 발생한다.

The type Rectangle must implement the inherited abstract method Shape draw()

```
public class Rectangle extends Shape {  
    double width;  
    double height;  
    ...  
    public void draw() {  
        super.drawCenter();  
        System.out.printf("가로: %f, 세로: %f, ", width, height);  
        System.out.printf("사각형 면적: %f\n", width*height);  
    }  
}
```

```
public class Circle extends Shape {  
    double radius;  
    ...  
    public void draw() {  
        super.drawCenter();  
        System.out.printf("반지름: %f, ", radius);  
        System.out.printf("원 면적: %f\n", radius*radius*Math.PI);  
    }  
}
```

3. 추상 클래스와 인터페이스

- 추상 메소드
 - 목적

어떤 특정 기능을

하위 클래스에서

반드시 만들도록 강제하고자 할 때 사용

3. 추상 클래스와 인터페이스

- 인터페이스(interface)

- 추상 메소드들만 있는 클래스

- 목적

- 여러 객체들을 대상으로 같은(표준) 기능을 구현하기 위함

- 특징

- 객체화 불가

- 다중 상속(multiple inheritance) 지원

- 한 클래스는 둘 이상의 인터페이스를 동시에 구현

- 상위 인터페이스를 슈퍼로 구분하여 나열

3. 추상 클래스와 인터페이스

- 인터페이스
 - 만드는 방법

interface 클래스명

- 선언 예

```
public interface Connectable {  
    //상수 정의  
    public static final String name = "연결 방법: USB";  
  
    //추상 메소드 선언  
    public abstract void connect();  
}
```

상수이어야 하며 반드시 **public static final**만 허용한다.
생략해도 **public static final**로 간주한다.

추상이어야 하며 반드시 **public abstract**만 허용한다.
생략해도 **public abstract**로 간주한다.

3. 추상 클래스와 인터페이스

- 인터페이스

- 인터페이스간 상속 방법

```
interface 인터페이스명 extends 인터페이스명
```

- 인터페이스간 상속 예

```
public interface Interface1 extends OtherInterface {  
    ...  
}  
Interface1 f = new Interface1(); //객체화 오류  
  
public interface Interface2 extends Other1, Other2, Other3 {  
    ...  
}
```

3. 추상 클래스와 인터페이스

- 인터페이스

- 상속 방법

클래스명 implements 인터페이스명

- 상속 예

```
public class HPPrinter implements Connectable {
```

```
...
```

상위 인터페이스 Connectable에 정의한 추상 메소드를 반드시 구현해야 한다.

```
}
```

```
public class HPPrinter extends Device implements Connectable1, Connectable2 {
```

```
...
```

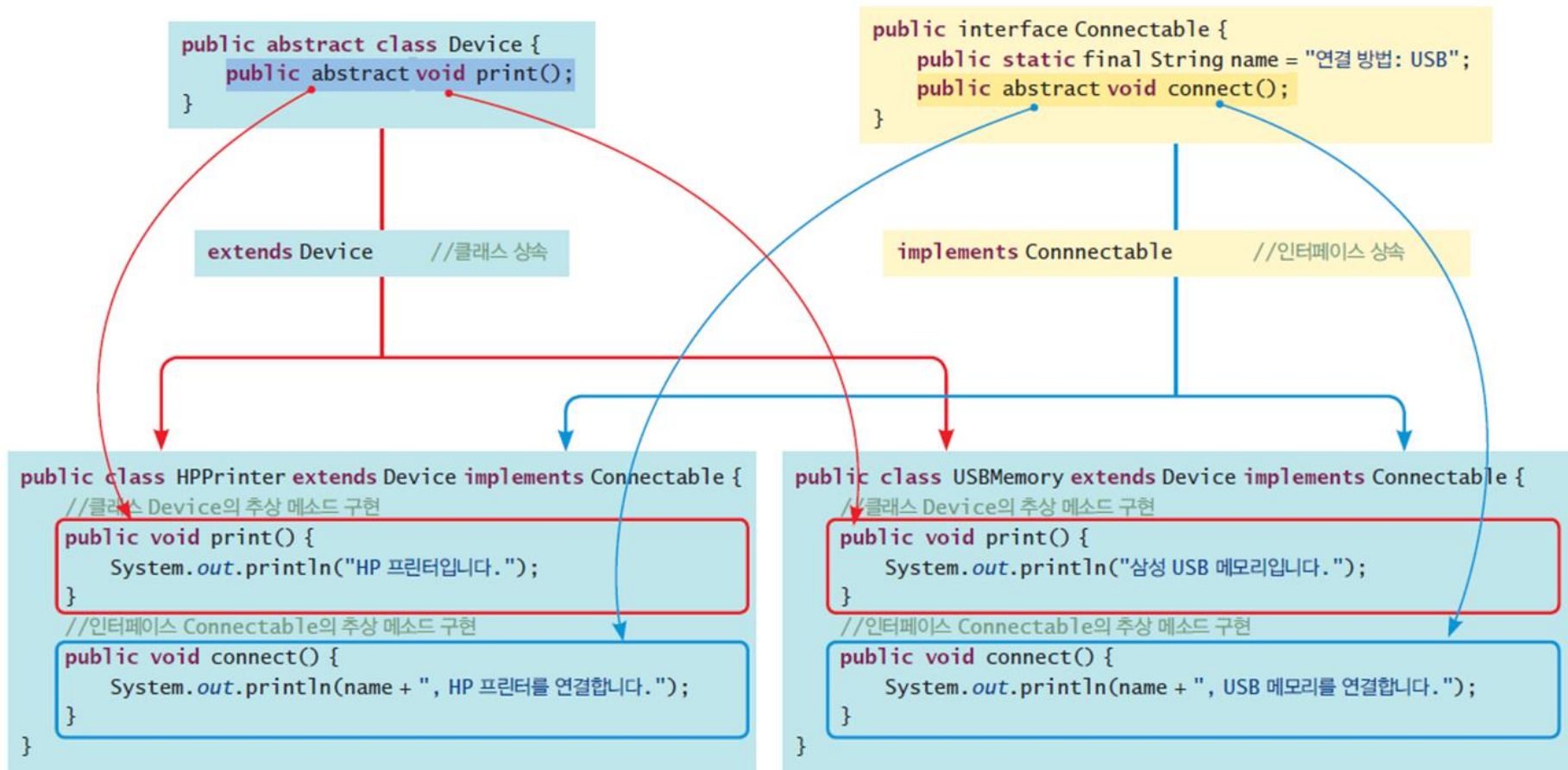
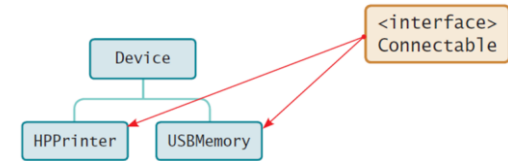
상위 클래스 Device와 상위 인터페이스 Connectable에 정의한 추상 메소드를 반드시 구현해야 한다.

```
}
```

3. 추상 클래스와 인터페이스

- 인터페이스

- 클래스 상속과 인터페이스 구현 예



Exercise

[6-1] 다음 프로그램의 출력 결과는?

```
class Super {
    int n = 1;
    public int getN() {
        return n;
    }
}

class Sub extends Super {
    int n = 10;
    public int getN() {
        return n;
    }
    public int getSuperN() {
        return super.getN();
    }
}

public class ex {
    public static void main(String[] args) {
        Super sup = new Sub();
        System.out.println(sup.getN());
        System.out.println(sup.n);
    }
}
```