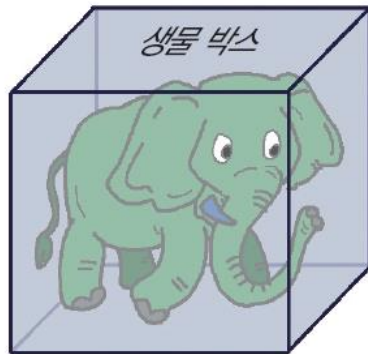




한림대학교 SW중심대학

메소드 오버라이딩

- 업캐스팅(upcasting)
 - 서브 클래스의 레퍼런스를 슈퍼 클래스 레퍼런스에 대입
 - 슈퍼 클래스 레퍼런스로 서브 클래스 객체를 가리키게 되는 현상
 - 슈퍼클래스의 필드와 메소드에만 접근 가능
 - 메소드가 서브클래스에서 오버라이딩 되었다면 서브클래스의 메소드가 대신 호출된다.



생물이 들어가는 박스에
사람이나 코끼리를 넣어도 무방

* 사람이나 코끼리 모두 생물을
상속받았기 때문

```
class Person { }  
class Student extends Person { }  
  
Person p;  
Student s = new Student();  
p = s; // 업캐스팅
```

```
class Person {
    String name;
    String id;

    public Person(String name) {
        this.name = name;
    }
}

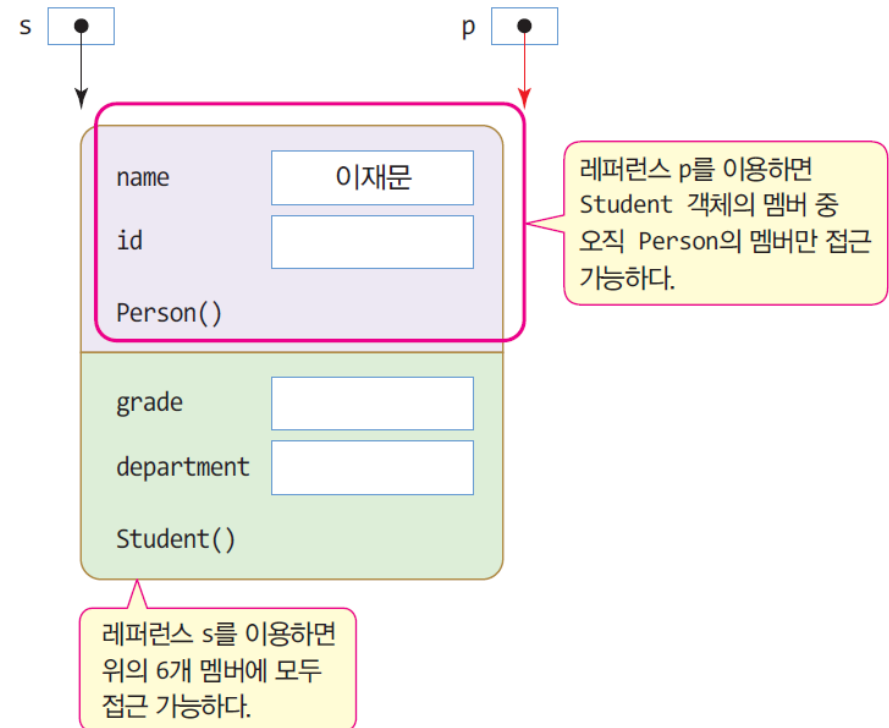
class Student extends Person {
    String grade;
    String department;

    public Student(String name) {
        super(name);
    }
}

public class UpcastingEx {
    public static void main(String[] args) {
        Person p;
        Student s = new Student("이재문");
        p = s; // 업캐스팅 발생

        System.out.println(p.name); // 오류 없음

        p.grade = "A"; // 컴파일 오류
        p.department = "Com"; // 컴파일 오류
    }
}
```



슈퍼클래스의 필드와 메소드에만 접근 가능

이재문

- **메소드 오버라이딩(Method Overriding)**
 - 서브 클래스에서 슈퍼 클래스의 메소드 중복 작성
 - 항상 상속시 서브 클래스에 오버라이딩한 메소드가 실행
 - 하나의 메소드명에 서로 다른 구현이 가능
 - 슈퍼 클래스의 메소드를 서브 클래스에서 각각 목적에 맞게 다르게 구현
- **오버라이딩 조건**
 - 슈퍼 클래스 메소드의 원형(메소드 이름, 인자 타입 및 개수, 리턴 타입) 동일하게 작성
 - 구현부만 틀림

오버로딩과 오버라이딩

비교 요소	메소드 오버로딩	메소드 오버라이딩
선언	<u>같은 클래스나 상속 관계에서 동일한 이름의 메소드 중복 작성</u>	서브 클래스에서 슈퍼 클래스에 있는 메소드와 동일한 이름의 메소드 재작성
관계	<u>동일한 클래스 내</u> 혹은 상속 관계	상속 관계
목적	이름이 같은 여러 개의 메소드를 중복 선언하여 사용의 편리성 향상	<u>슈퍼 클래스에 구현된 메소드를 무시하고</u> 서브 클래스에서 새로운 기능의 메소드를 재정의하고자 함
조건	메소드 이름은 반드시 동일함. 메소드의 인자의 개수나 인자의 타입이 달라야 성립	메소드의 이름, 인자의 타입, 인자의 개수, 인자의 리턴 타입 등이 모두 동일하여야 성립
바인딩	정적 바인딩. 컴파일 시에 중복된 메소드 중 호출되는 메소드 결정	동적 바인딩. 실행 시간에 오버라이딩된 메소드 찾아 호출

```
public class BaiscCalculator
{
    public String calName = "pCal";
    public int add(int x, int y)
    {
        System.out.println("add()");
        int resultAdd=x+y;
        return resultAdd;
    }
    public int sub(int x, int y)
    {
        System.out.println("sub");
        int resultAdd=x-y;
        return resultAdd;
    }
    public int mul(int x, int y)
    {
        System.out.println("mul()");
        int resultAdd=x-y;
        return resultAdd;
    }
    public double div(int x, int y)
    {
        double ret=0;
        ret=x/y;
        return ret;
    }
    public void printValues(int x, int y)
    {
        System.out.println("x : "+x+"y : "+y);
    }
}
```

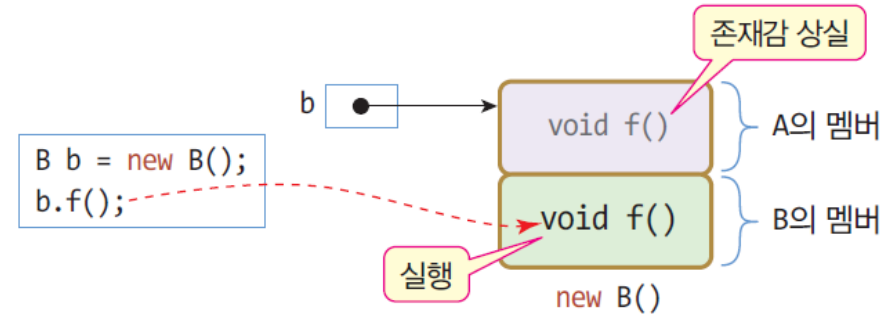
```
public class EnginneringCalculator extends
BaiscCalculator
{
    public void printValues(int x, int y)
    {
        String hx = Integer.toHexString(x);
        String hy = Integer.toHexString(y);
        System.out.println("hex x : "+hx);
        System.out.println("hex y : "+hy);
    }
}
```

```
public class MainApp extends BaiscCalculator
{
    public static void main(String[] args)
    {
        EnginneringCalculator ch = new
EnginneringCalculator();
        ch.printValues(15, 9);
    }
}
```

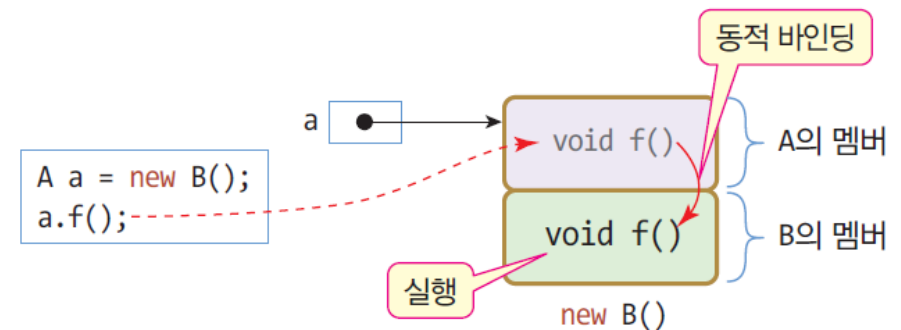
메소드 재정의(Override)

```
class A {  
    void f() {  
        System.out.println("A의 f() 호출");  
    }  
}  
class B extends A {  
    void f() {  
        System.out.println("B의 f() 호출");  
    }  
}
```

(a) 오버라이딩된 메소드, B의 f() 직접 호출



(b) A의 f()를 호출해도, 오버라이딩된 메소드, B의 f()가 실행됨



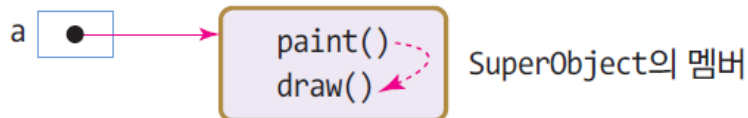
- 정적바인딩(컴파일시 바인딩)
 - 참조변수와 인스턴스를 컴파일때 연결하는 것이라 컴파일시에 모든 연결(호출)이 결정
 - 실행시에 연결을 변경할 수 없음
- 동적바인딩(실행시 바인딩)
 - 프로그램 실행중 함수가 호출될 때 그 메모리 참조를 알아내는 것을 뜻함.
 - 실행시에 참조변수와 인스턴스등을 연결
 - 정적바인딩 대비 굉장한 유연성을 갖게 됨

동적 바인딩 – 오버라이딩된 메소드 호출

정적바인딩

```
class SuperObject {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Super Object");
    }
}
public class SubObject extends SuperObject {
    public static void main(String [] args) {
        SuperObject b = new SubObject();
        b.paint();
    }
}
```

Super
Object

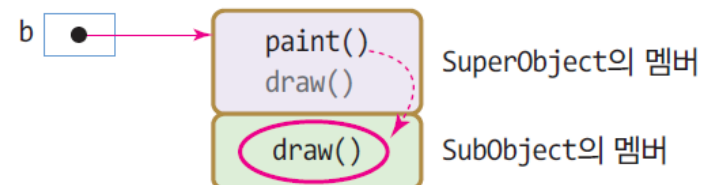


동적바인딩

```
class SuperObject {
    protected String name;
    public void paint() {
        draw();
    }
    public void draw() {
        System.out.println("Super Object");
    }
}
public class SubObject extends SuperObject {
    public void draw() {
        System.out.println("Sub Object");
    }
    public static void main(String [] args) {
        SuperObject b = new SubObject();
        b.paint();
    }
}
```

동적바인딩

Sub
Object



Shape의 draw() 메소드를 Line, Circle, Rect 클래스에서 목적에 맞게 오버라이딩하는 다형성의 사례를 보여준다.

```
class Shape { // 도형의 슈퍼 클래스
    public void draw() {
        System.out.println("Shape");
    }
}
class Line extends Shape {
    public void draw() {
        System.out.println("Line");
    }
}
class Rect extends Shape {
    public void draw() {
        System.out.println("Rect");
    }
}
class Circle extends Shape {
    public void draw() {
        System.out.println("Circle");
    }
}
```

동적바인딩

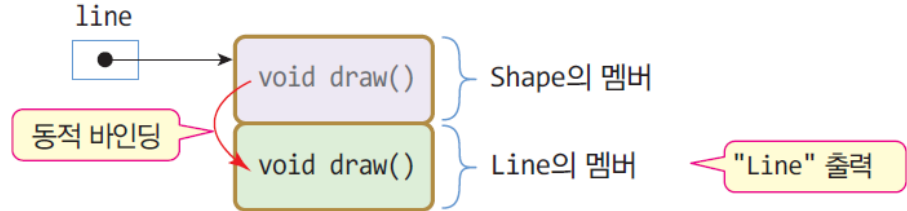
```
public class MethodOverridingEx {
    static void paint(Shape p) { // Shape을 상속받은 객체들이
        // 매개 변수로 넘어올 수 있음
        p.draw(); // p가 가리키는 객체에 오버라이딩된 draw() 호출.
        // 동적바인딩
    }

    public static void main(String[] args) {
        Shape shape = new Line();
        paint(shape);
        shape = new Shape();
        paint(shape);
        shape = new Rect();
        paint(shape);
        shape = new Circle();
        paint(shape);
    }
}
```

Line
Shape
Rect
Circle

예제

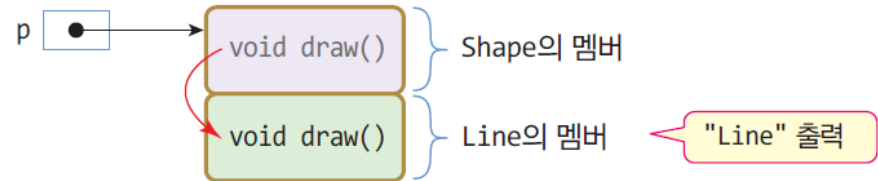
```
Line line = new Line()  
paint(line);
```



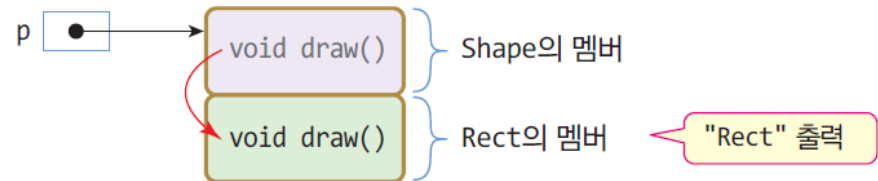
```
paint(new Shape());
```



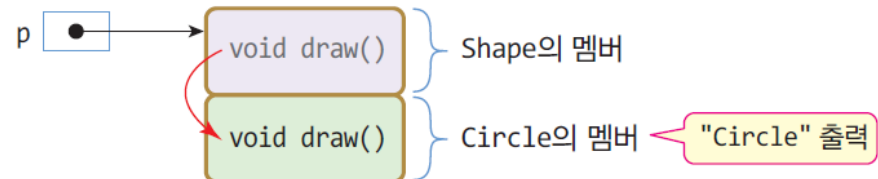
```
paint(new Line());
```



```
paint(new Rect());
```



```
paint(new Circle());
```



추상클래스

■ 추상 메소드(abstract method)

- **abstract로 선언된 메소드 : 구현부가 없고 원형만 선언**

```
public abstract String getName(); // 추상 메소드
```



```
public abstract String fail() {  
    return "Good Bye";  
} // 추상 메소드 아님. 컴파일 오류
```

■ 추상 클래스(abstract class)

- 추상 메소드를 가지며, abstract로 선언된 클래스
- 추상 메소드 없이, abstract로 선언한 클래스

```
// 추상 메소드를 가진 추상 클래스  
abstract class Shape {  
    Shape() { ... }  
    void edit() { ... }  
  
    abstract public void draw(); // 추상 메소드  
}
```

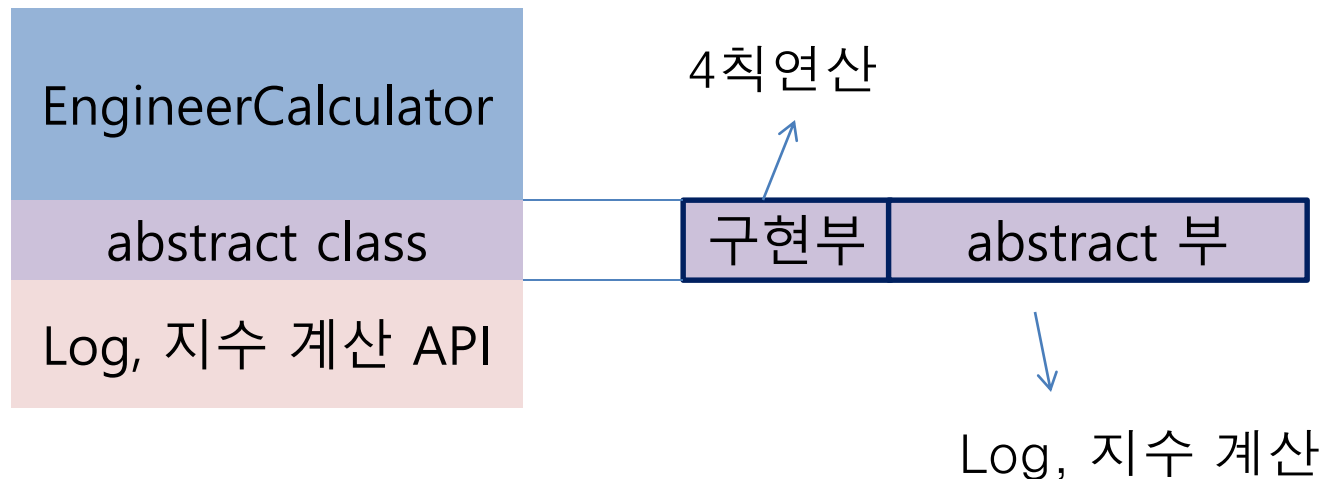


```
class fault { // 오류. 추상 메소드를 가지고 있으므로 abstract로 선언되어야 함  
    abstract void f(); // 추상 메소드  
}
```

추상 클래스(Abstract Class)

■ 추상 클래스의 용도

- 구현 클래스 설계 규격을 만들고자 할 때
 - 구현 클래스가 가져야 할 필드와 메소드를 추상 클래스에 미리 정의
 - 구현 클래스의 공통된 필드와 메소드의 이름 통일할 목적
- 전체 기능중 일부 기능이 달라질수 있을 경우




■ 추상 클래스 선언

■ 클래스 선언에 abstract 키워드 사용

- New 연산자로 객체 생성하지 못하고 **상속 통해 자식 클래스만 객체 생성 가능**

```
public abstract class 클래스 {  
    //필드  
    //생성자  
    //메소드  
}
```

■ 추상 클래스는 온전한 클래스가 아니기 때문에 인스턴스를 생성할 수 없음

JComponent p;	// 오류 없음. 추상 클래스의 레퍼런스 선언
p = new JComponent();	// 컴파일 오류. 추상 클래스의 인스턴스 생성 불가
 Shape obj = new Shape();	// 컴파일 오류. 추상 클래스의 인스턴스 생성 불가

컴파일 오류 메시지

Unresolved compilation problem: Cannot instantiate the type Shape

- 추상 메소드와 오버라이딩(재정의)
 - 메소드 이름 동일하지만, 실행 내용이 실체 클래스마다 다른 메소드
 - 구현 방법
 - 추상 클래스에는 메소드의 선언부만 작성 (추상 메소드)
 - 실체 클래스에서 메소드의 실행 내용 작성(오버라이딩(Overriding))


```
public abstract class Phone {  
    //필드  
    public String owner;  
  
    //메소드  
    public void turnOn() {  
        System.out.println("폰 전원을 켭니다.");  
    }  
    public void turnOff() {  
        System.out.println("폰 전원을 끕니다.");  
    }  
  
    public abstract String saveContact(String phoneNum);  
}
```

```
public class PhoneExample {  
    public static void main(String[] args) {  
        //Phone phone = new Phone(); (x)  
  
        Phone smartPhone = new GoogleContact();  
        //Phone smartPhone = new NaverContact();  
        String pNum="010-2392-2343";  
        String retVal=null;  
        smartPhone.turnOn();  
        retVal=smartPhone.saveContact(pNum);  
        System.out.println(retVal);  
        smartPhone.turnOff();  
    }  
}
```

```
public class NaverContact extends Phone {  
    String pb=null;  
  
    public String saveContact(String phoneNum)  
    {  
        System.out.println("naver contact added : "+phoneNum);  
        this.pb+=phoneNum;  
        return this.pb;  
    }  
}
```

```
public class GoogleContact extends Phone {  
    String pb=null;  
  
    public String saveContact(String phoneNum)  
    {  
        System.out.println("google contact added : "+phoneNum);  
        this.pb+=phoneNum;  
        return this.pb;  
    }  
}
```



한림대학교 SW중심대학

인터페이스

- 인터페이스의 역할
- 인터페이스 선언
- 인터페이스 구현
- 인터페이스 사용
- 타입변환과 다형성

인터페이스의 필요성

20



A사 제품



B사 제품



C사 제품



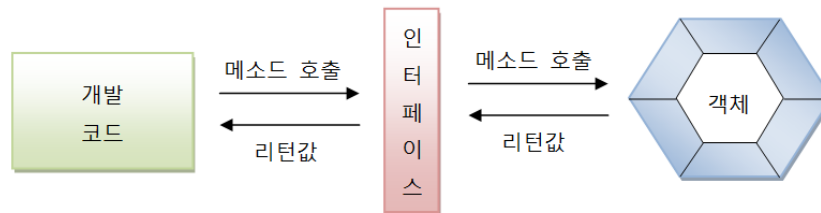
D사 제품

정해진 규격(인터페이스)에
맞기만 하면 연결 가능.
각 회사마다 구현 방법은 다름

정해진 규격(인터페이스)에 맞지
않으면 연결 불가

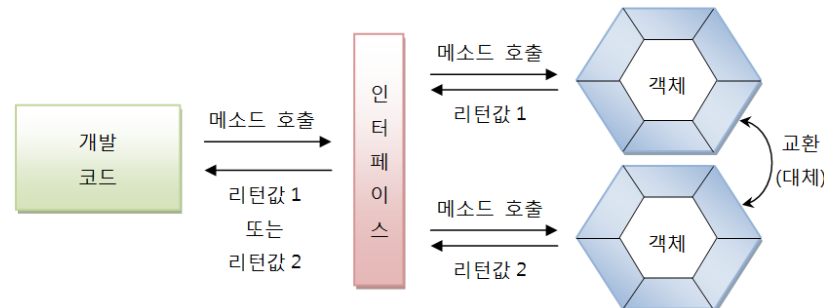
■ 인터페이스란?

- 개발 코드와 객체가 서로 통신하는 접점(형식을 약속)
 - 개발 코드는 **인터페이스의 메소드만 알고 있으면 OK**



■ 인터페이스의 역할

- 개발 코드가 객체에 종속되지 않게 -> 객체 교체할 수 있도록 하는 역할
- 개발 코드 변경 없이 리턴값 또는 실행 내용이 다양해 질 수 있음 (다형성)



- 기능에 대한 선언과 구현분리
 - 표준화 가능
 - 독립적인 프로그래밍이 가능
 - 개발시간 단축 가능

- Abstract 클래스와의 차이점
 - 모두 abstract 메소드로 구현되어 있음
 - 구현부분이 존재하지 않음

■ 인터페이스 선언

- 인터페이스 이름 - 자바 식별자 작성 규칙에 따라 작성

- 소스 파일 생성

- 인터페이스 이름과 대소문자가 동일한 소스 파일 생성

- 인터페이스 선언

```
[ public ] interface 인터페이스명 { ... }
```

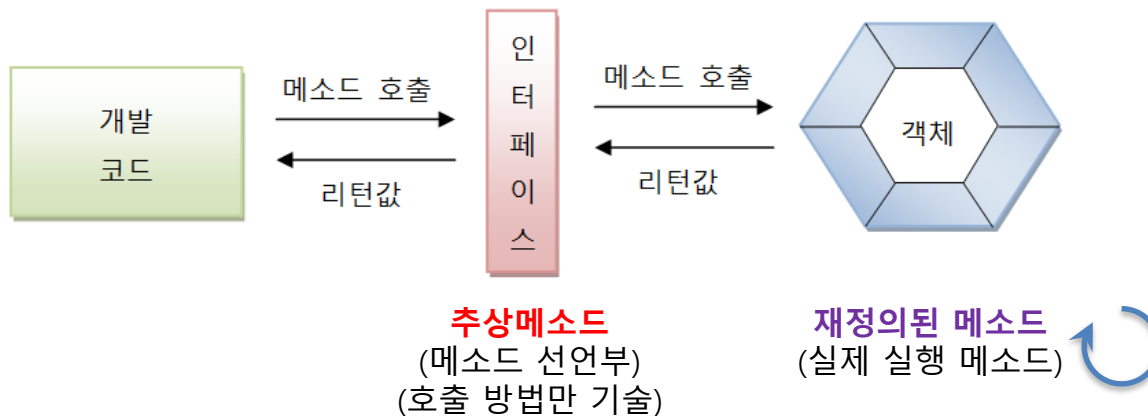
```
public class 클래스명{...}
```

```
public abstract class 클래스명{...}
```

```
public interface RemoteControl {  
}  
  
public class RemoteControl{  
}  
  
public atstract class RemoteControl{  
}
```

■ 추상 메소드 선언

- 메소드 선언하는 형식에 **abstract 키워드 추가, 구현부 없음**
- 인터페이스 통해 호출된 메소드는 최종적으로 객체에서 실행
 - 인터페이스의 메소드는 기본적으로 실행 블록이 없는 추상 메소드로 선언



```
[ public abstract ] 리턴타입 메소드명(매개변수, ...);
```

```
public 리턴타입 메소드명(매개변수, ...) {...}
```


25

- 자바 인터페이스
 - 상수와 추상 메소드로만 구성 : 변수 필드 없음
 - 인터페이스 선언
 - interface 키워드로 선언

```
interface PhoneInterface {  
    int BUTTONS = 20; // 상수 필드 선언  
    void sendCall(); // 추상 메소드  
    void receiveCall(); // 추상 메소드  
}
```

public interface로서 public 생략 가능

public static final로서 public static final 생략 가능

abstract public 으로서 abstract public 생략 가능

- 자바 인터페이스의 특징
 - 상수와 추상 메소드로만 구성
 - 메소드 : public abstract 타입으로 생략 가능
 - 상수 : public static final 타입으로 생략 가능
 - 인터페이스의 객체 생성 불가

```
new PhoneInterface(); // 오류. 인터페이스의 객체를 생성할 수 없다.
```

오류

```
public interface RemoteControl {  
    // 상수  
    int MAX_VOLUME = 10;  
    int MIN_VOLUME = 0;  
  
    // 추상 메소드  
    void turnOn();  
    void turnOff();  
    void setVolume(int volume);  
}
```

■ 구현 클래스 선언

- 자신의 객체가 인터페이스 타입으로 사용할 수 있음
 - implements 키워드로 명시

```
public class 구현클래스명 implements 인터페이스명 {  
    //인터페이스에 선언된 추상 메소드의 실제 메소드 선언  
}
```

■ 추상 메소드의 실제 메소드를 작성하는 방법

- 메소드의 선언부가 정확히 일치해야
- 인터페이스의 모든 추상 메소드를 재정의하는 실제 메소드 작성해야
 - 일부만 재정의할 경우, 추상 클래스로 선언 + abstract 키워드 붙임

```
public interface RemoteControl {  
  
    void turnOn();  
    void turnOff();  
    void setVolume(int volume);  
  
}  
  
public class RemoteControlImpl implements RemoteControl {  
    //turnOn() 추상 메소드의 실제 메소드  
    public void turnOn() {  
        System.out.println("TV를 켭니다.");  
    }  
    //turnOff() 추상 메소드의 실제 메소드  
    public void turnOff() {  
        System.out.println("TV를 끕니다.");  
    }  
    //setVolume() 추상 메소드의 실제 메소드  
    public void setVolume(int volume) {  
  
        System.out.println("현재 TV 볼륨: " + volume);  
    }  
}  
  
public class RemoteControlExample {  
    public static void main(String[] args) {  
        RemoteControl rc;  
        rc = new RemoteControlImpl();  
        rc.turnOn();  
        rc.turnOff();  
        rc.setVolume(1);  
    }  
}
```