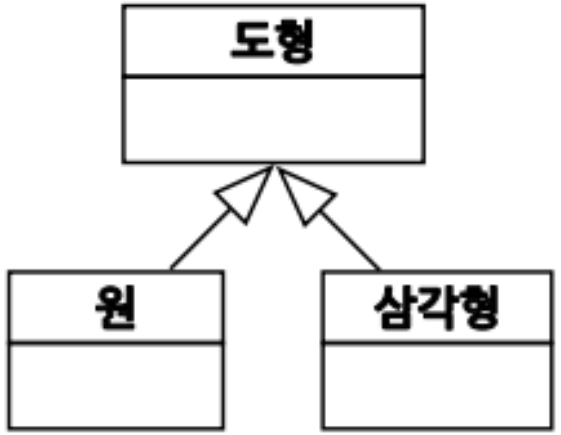
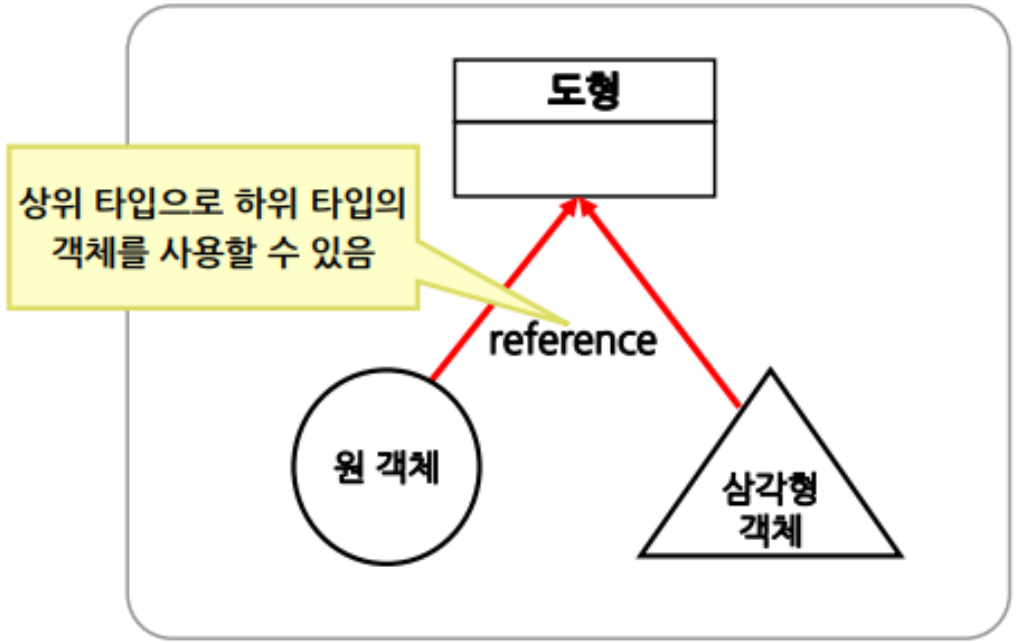
다형성
(Polymorphism)

▶ 다형성

객체지향 프로그래밍의 **3대 특징 중 하나**로 '**여러 개의 형태를 갖는다**'는 의미
하나의 행동으로 여러 가지 일을 수행하는 개념
상속을 이용한 기술로 **부모 타입으로부터 파생된 여러 가지 타입의 자식 객체**를
부모 클래스 타입 하나로 다룰 수 있는 기술





상위 클래스의 참조변수가 참조할 수 있는 대상의 범위

```
class SmartPhone extends MobilePhone {....}
```

스마트폰은 모바일폰이다.

```
SmartPhone phone = new SmartPhone("010-555-777", "Nougat");
```

따라서 스마트폰 참조변수로 스마트폰 참조 가능하고,

```
MobilePhone phone = new SmartPhone("010-555-777", "Nougat");
```

이 역은 성립하지 않음에 주의!

모바일폰 참조변수로 스마트폰 참조도 가능하다.

참조변수의 참조 가능성 관련 예제

```
class MobilePhone {
   protected String number;

   public MobilePhone(String num) {
      number = num;
   }
   public void answer() {
      System.out.println("Hi~ from " + number);
   }
}

class SmartPhone extends MobilePhone {
   private String androidVer;

   public SmartPhone(String num, String ver) {
      super(num);
      androidVer = ver;
   }
   public void playApp() {
      System.out.println("App is running in " + androidVer);
   }
}
```

```
public static void main(String[] args) {
   SmartPhone ph1 =
            new SmartPhone("010-555-777", "Nougat");
   MobilePhone ph2 =
            new SmartPhone("010-999-333", "Nougat");
   ph1.answer();
   ph1.playApp();
   System.out.println();

   ph2.answer();
}
```

출력 결과는?

참조변수의 참조 가능성 관련 예제

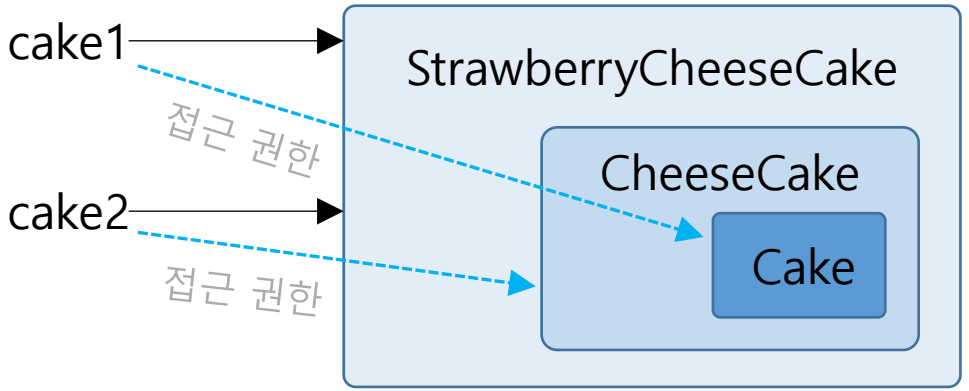
```
class MobilePhone {
   protected String number;

   public MobilePhone(String num) {
      number = num;
   }
   public void answer() {
      System.out.println("Hi~ from " + number);
   }
}

class SmartPhone extends MobilePhone {
   private String androidVer;

   public SmartPhone(String num, String ver) {
      super(num);
      androidVer = ver;
   }
   public void playApp() {
      System.out.println("App is running in " + androidVer);
   }
}
```

```
public static void main(String[] args) {
   SmartPhone ph1 =
            new SmartPhone("010-555-777", "Nougat");
   MobilePhone ph2 =
            new SmartPhone("010-999-333", "Nougat");
   ph1.answer();
   ph1.playApp();
   System.out.println();

   ph2.answer();
}
```

참조변수의 참조 가능성에 대한 정리

```
class Cake {
   public void sweet() {...}
}

class CheeseCake extends Cake {
   public void milky() {...}
}

class StrawberryCheeseCake extends CheeseCake {
   public void sour() {....}
}
```

```
Cake cake1 = new StrawberryCheeseCake();

CheeseCake cake2 = new StrawberryCheeseCake();
```



StrawberryCheeseCake 인스턴스

참조변수 간 대입과 형 변환

```
class Cake {

   public void sweet() {...}

}


class CheeseCake extends Cake {

   public void milky() {...}

}
```

```
CheeseCake ca1 = new CheeseCake();

Cake ca2 = ca1;    // 가능!


Cake ca3 = new CheeseCake();

CheeseCake ca4 = ca3;     // 불가능!
```

이 시점에 컴파일러 및 가상머신은 ca3가 참조하는 대상을
Cake 인스턴스로 판단한다!
ca3가 참조하는 인스턴스의 정확한 클래스 정보는 유지하지 않는다.

참조변수의 참조 가능성: 배열 기반

```
class Cake {
   public void sweet() {...}
}

class CheeseCake extends Cake {
   public void milky() {...}
}
```

```
Cake cake = new CheeseCake();
```
가능!

```
CheeseCake[] cakes = new CheeseCake[10];
```
가능!

```
Cake[] cakes = new CheeseCake[10];
```
가능!

상속의 관계가 배열 인스턴스의 참조 관계까지 이어진다.

메소드 오버라이딩1

```
class Cake {
    public void yummy() {
        System.out.println("Yummy Cake");
    }
}

class CheeseCake extends Cake {
    public void yummy() {
        System.out.println("Yummy Cheese Cake");
    }
}
```

오버라이딩 관계

CheeseCake의 yummy 메소드가

Cake의 yummy 메소드를 오버라이딩!

```
public static void main(String[] args) {
    Cake c1 = new CheeseCake();
    CheeseCake c2 = new CheeseCake();

    c1.yummy();
    c2.yummy();
}
```

메소드 오버라이딩 2

```
class Cake {
   public void yummy() {...}
}
class CheeseCake extends Cake {
   public void yummy() {...}    // Cake의 yummy를 오버라이딩
}
Class StrawberryCheeseCake extends CheeseCake {
   public void yummy() {...}    // 그리고 CheeseCake의 yummy를 오버라이딩
}

public static void main(String[] args) {
   Cake c1 = new StrawberryCheeseCake();
   CheeseCake c2 = new StrawberryCheeseCake();
   StrawberryCheeseCake c3 = new StrawberryCheeseCake();
   c1.yummy();
   c2.yummy();
}
```

오버라이딩 된 메소드 호출하는 방법

```
class Cake {
   public void yummy() {
      System.out.println("Yummy Cake");
   }
}

class CheeseCake extends Cake {
   public void yummy() {
      super.yummy();
      System.out.println("Yummy Cheese Cake");
   }

   public void tasty() {
      super.yummy();
      System.out.println("Yummy Tasty Cake");
   }
}
```

오버라이딩 된 메소드를 인스턴스 외부에서 호출하는 방법은 없다.

그러나 인스턴스 내부에서는 키워드 super를 이용해 호출 가능

인스턴스 변수와 클래스 변수도 오버라이딩이 되는가?

```
class Cake {
   public int size;
   ....
}

class CheeseCake extends Cake {
   public int size;
   ....
}
```

```
CheeseCake c1 = new CheeseCake();
c1.size = ... // CheeseCake의 size에 접근

Cake c2 = new CheeseCake();
C2.size = ... // Cake의 size에 접근
```

인스턴스 변수는 오버라이딩 되지 않는다. 따라서 참조변수의 형에 따라 접근하는 멤버가 결정된다.

▶ 클래스 형변환

✓ 업 캐스팅(Up Casting)

상속 관계에 있는 부모, 자식 클래스 간에 부모타입의 참조형 변수가

모든 자식 타입의 객체 주소를 받을 수 있음

```
//Sonata 클래스는 Car 클래스의 후손
Car c = new Sonata();
//Sonata클래스형에서 Car클래스형으로 바뀜
```

* 자식 객체의 주소를 전달받은 부모타입의 참조변수를 통해서 사용할 수 있는 후손의 정보는
  원래 부모타입이었던 멤버만 참조 가능

▶ 클래스 형변환

✓ 다운 캐스팅(Down Casting)

자식 객체의 주소를 받은 부모 참조형 변수를 가지고 자식의 멤버를 참조해야 할 경우,

부모 클래스 타입의 참조형 변수를 자식 클래스 타입으로 형 변환하는 것

자동으로 처리되지 않기 때문에 반드시 후손 타입 명시해서 형 변환

```
//Sonata 클래스는 Car 클래스의 후손
Car c = new Sonata();
((Sonata)c).moveSonata();
```

* 클래스 간의 형 변환은 **반드시 상속 관계에 있는 클래스끼리**만 가능

▶ instanceof 연산자

현재 참조형 변수가 어떤 클래스 형의 객체 주소를 참조하고 있는지 확인 할 때 사용

클래스 타입이 맞으면 true, 맞지 않으면 false 반환

✓ 표현식

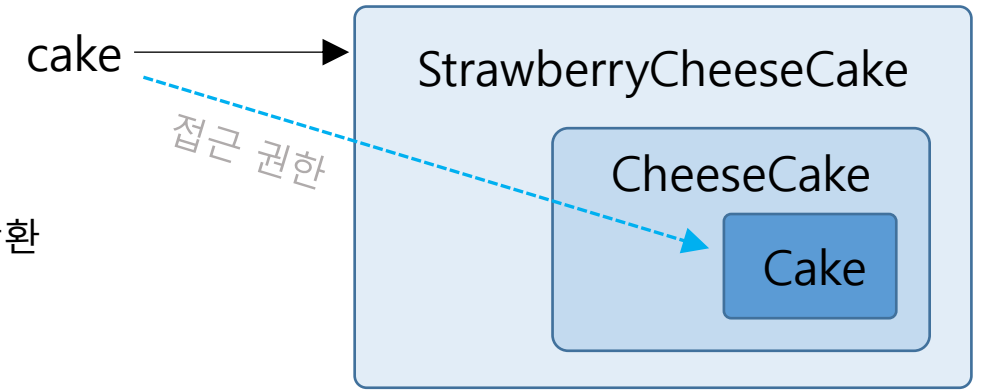
```
if(레퍼런스 instanceof 클래스타입) {
        //true일때 처리할 내용, 해당 클래스 타입으로 down casting
}

if(c instanceof Sonata) {
        ((Sonata)c).moveSonata();
} else if (c instanceof Avante){
        ((Avante)c).moveAvante();
} else if (c instanceof Grandure){
        ((Grandure)c).moveGrandure();
}
```

instanceof 연산자의 기본

```
class Cake {
}

class CheeseCake extends Cake {
}

class StrawberryCheeseCake extends CheeseCake{
}
```

```
public static void main(String[] args) {
   Cake cake = new StrawberryCheeseCake();

   if(cake instanceof Cake) {...}

   if(cake instanceof CheeseCake) {...}

   if(cake instanceof StrawberryCheeseCake) {...}
}
```

```
if(ref instanceof ClassName)
 ref가 ClassName 클래스의 인스턴스를 참조하면 true 반환
 ref가 ClassName를 상속하는 클래스의 인스턴스이면 true 반환
```



instanceof 연산자의 활용

"상속은 연관된 일련의 클래스들에 대해
공통적인 규약을 정의할 수 있습니다."

```
class Box {
   public void simpleWrap() {
      System.out.println("Simple Wrapping");
   }
}

class PaperBox extends Box {
   public void paperWrap() {
      System.out.println("Paper Wrapping");
   }
}

class GoldPaperBox extends PaperBox {
   public void goldWrap() {
      System.out.println("Gold Wrapping");
   }
}
```

instanceof 연산자의 사용 예!

그런데 이 예제 코드의 완성도에 점수를 준다면?

```
public static void main(String[] args) {
   Box box1 = new Box();
   PaperBox box2 = new PaperBox();
   GoldPaperBox box3 = new GoldPaperBox();

   wrapBox(box1);
   wrapBox(box2);
   wrapBox(box3);
}

public static void wrapBox(Box box) {
   if (box instanceof GoldPaperBox) {
      ((GoldPaperBox)box).goldWrap();
   }
   else if (box instanceof PaperBox) {
      ((PaperBox)box).paperWrap();
   }
   else {
      box.simpleWrap();
   }
}
```

객체배열과 다형성

다형성을 이용하여 상속 관계에 있는 하나의 부모 클래스 타입의 배열 공간에
여러 종류의 자식 클래스 객체 저장 가능

```
Car[] carArr = new Car[5];

carArr[0] = new Sonata();
carArr[1] = new Avante();
carArr[2] = new Grandure();
carArr[3] = new Spark();
carArr[4] = new Morning();
```

매개변수와 다형성

다형성을 이용하여 메소드 호출 시 부모타입의 변수 하나만 사용해 자식 타입의 객체를 받을 수 있음

```
public void execute() {
        driveCar(new Sonata());
        driveCar(new Avante());
        driveCar(new Grandure());
}

public void driveCar(Car c) {}
```

인터페이스의 기본과 그 의미

추상 메소드만 담고 있는 인터페이스

```
interface Printable {
   public void print(String doc);   // 추상 메소드
}
```

인터페이스의 정의! 메소드의 몸체를 갖지 않는다.

따라서 인스턴스 생성 불가! 참조변수 선언 가능!

```
class Printer implements Printable {
   public void print(String doc) {
      System.out.println(doc);
   }
}
```

인터페이스를 구현하는 클래스!

구현하는 메소드와 추상 메소드 사이에도 메소드 오버라이딩 관계 성립, 따라서 @Override 붙일 수 있음

인터페이스형 참조변수 선언 가능

```
Printable prn = new Printer();
prn.print("Hello");
```

상속과 구현

```
class Robot extends Machine implements Movable, Runnable {...}
```

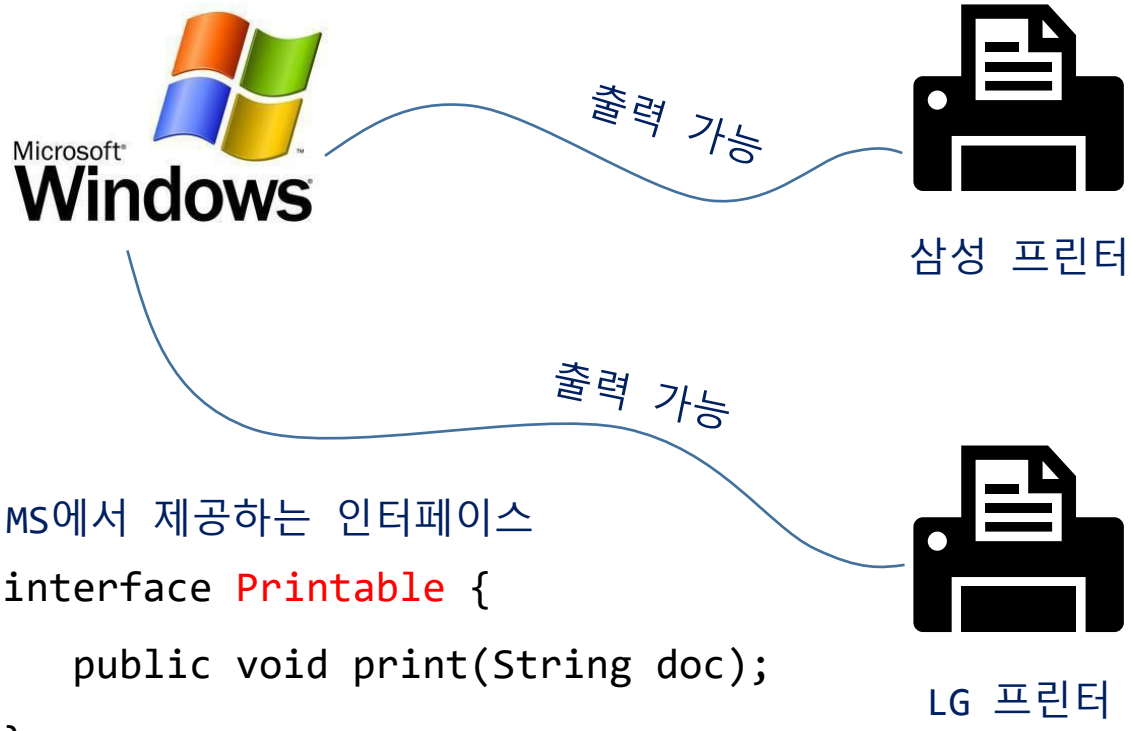
Robot 클래스는 Machine 클래스를 상속한다.

이렇듯 상속과 구현 동시에 가능!

Robot 클래스는 Movable과 Runnable 인터페이스를 구현한다.

이렇듯 둘 이상의 인터페이스 구현 가능!

인터페이스의 본질적 의미



MS에서 제공하는 인터페이스

```
interface Printable {
    public void print(String doc);
}
```

```
class SPrinterDriver implements Printable {
    @Override
    public void print(String doc) {...}
}
```

```
class LPrinterDriver implements Printable {
    @Override
    public void print(String doc) {...}
}
```

Printer Driver 관련 예제

```
interface Printable { // MS가 정의하고 제공한 인터페이스
   public void print(String doc);
}

class SPrinterDriver implements Printable {
   @Override
   public void print(String doc) {
      System.out.println("From Samsung printer");
      System.out.println(doc);
    }
}

class LPrinterDriver implements Printable {
   @Override
   public void print(String doc) {
      System.out.println("From LG printer");
       System.out.println(doc);
    }
 }
```

```
public static void main(String[] args) {
   String myDoc = "This is a report about...";

   // 삼성 프린터로 출력
   Printable prn = new SPrinterDriver();
   prn.print(myDoc);
   System.out.println();

   // LG 프린터로 출력
   prn = new LPrinterDriver();
   prn.print(myDoc);
}
```

인터페이스의 문법 구성과
추상 클래스

인터페이스에 선언되는 메소드와 변수

```
interface Printable {
    public void print(String doc);    // 추상 메소드
}
```

```
interface Printable {
    public static final int PAPER_WIDTH = 70;
    public static final int PAPER_HEIGHT = 120;
    public void print(String doc);
}
```

인터페이스간 상속: 문제 상황의 제시


```
interface Printable {
    void print(String doc);
    void printCMYK(String doc);
}
```

컬러 출력 위한 메소드 추가되면?
시스템 전체에 문제 발생

```
class SPrinterDriver implements Printable {
    @Override
    public void print(String doc) {...}
}   // 이 클래스에서 printCMYK 메소드 구현해야 함
```

```
class LPrinterDriver implements Printable {
    @Override
    public void print(String doc) {...}
}   // 이 클래스에서 printCMYK 메소드 구현해야 함
```

인터페이스를 구현하는 클래스는 해당 인터페이스의 모든 추상 메소드를 구현해야 한다. 그래야 인스턴스 생성 가능!

제시한 문제의 해결책: 인터페이스의 상속

```
interface Printable {
    void print(String doc);
}
```

인터페이스간 상속도 extends로 표현

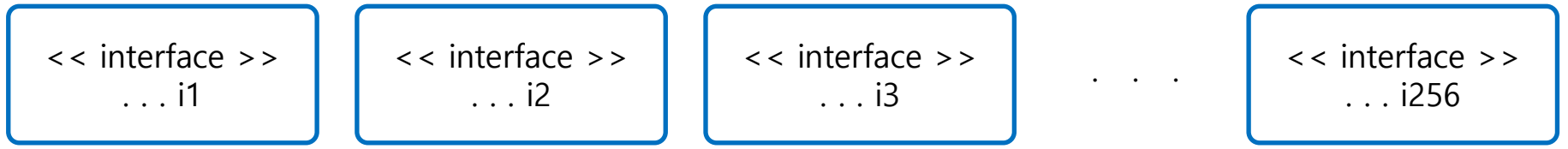
```
interface ColorPrintable extends Printable {
    void printCMYK(String doc);
}
```

```
class SPrinterDriver implements Printable {
    ...
}   // 기존 클래스 수정할 필요 없음
```

```
class Prn909Drv implements ColorPrintable {
    @Override
    public void print(String doc) {   // 흑백 출력
        System.out.println("black & white ver");
        System.out.println(doc);
    }

    @Override
    public void printCMYK(String doc) {   // 컬러 출력
        System.out.println("CMYK ver");
        System.out.println(doc);
    }
}
```

컬러 프린터 드라이버

인터페이스의 디폴트 메소드: 문제 상황의 제시



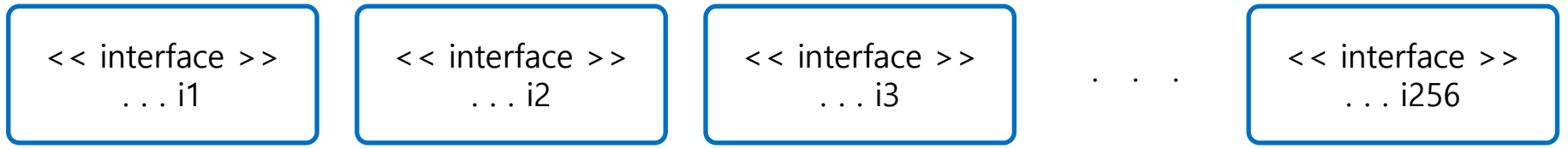
총 256개의 인터페이스가 존재하는 상황에서 모든 인터페이스에 다음 추상 메소드를 추가해야 한다면?

```
void printCMYK(String doc);
```

인터페이스간 상속?

물론 인터페이스간 상속으로 문제 해결 가능하다. 다만 인터페이스의 수가 256개 늘어날 뿐이다.

문제 상황의 해결책: 인터페이스의 디폴트 메소드



총 256개의 인터페이스가 존재하는 상황에서 모든 인터페이스에 다음 추상 메소드를 추가해야 한다면?

```
void printCMYK(String doc);
```

다음 디폴트 메소드로 이 문제를 해결하면 인터페이스의 수가 늘어나지 않는다.

```
default void printCMYK(String doc) { }    // 디폴트 메소드
```

디폴트 메소드의 효과

```
interface Printable {
    void print(String doc);
}
```

인터페이스의 교체

```
interface Printable {
    void print(String doc);
    default void printCMYK(String doc) { }
}
```

```
class SPrinterDriver implements Printable {
    @Override
    public void print(String doc) {...}
}
```

기존에 정의된 클래스:

인터페이스 교체로 인해 코드 수정 필요 없다.

```
class Prn909Drv implements Printable {
    @Override
    public void print(String doc) {...}

    @Override
    public void printCMYK(String doc) {...}
}
```

새로 정의된 클래스

인터페이스의 static 메소드

"인터페이스에도 static 메소드를 정의할 수 있다."

"그리고 인터페이스의 static 메소드 호출 방법은 클래스의 static 메소드 호출 방법과 같다."

```
interface Printable {
    static void printLine(String str) {
        System.out.println(str);
    }

    default void print(String doc) {
        printLine(doc); // 인터페이스의 static 메소드 호출
    }
}
```

인터페이스 대상의 instanceof 연산

```
if(ca instanceof Cake) ....
```

Cake는 클래스의 이름도, 인터페이스의 이름도 될 수 있다.

ca가 참조하는 인스턴스를 Cake형 참조변수로 참조할 수 있으면 true 반환!

교재에서는 이를 다음과 같이 설명하고 있다. (같은 의미, 표현만 다를 뿐)

ca가 참조하는 인스턴스가 Cake를 직접 혹은 간접적으로 구현한 클래스의 인스턴스인 경우 true 반환!

인터페이스 대상 instanceof 연산의 예

```
interface Printable {
   void printLine(String str);
}

class SimplePrinter implements Printable {
   public void printLine(String str) {
      System.out.println(str);
   }
}

class MultiPrinter extends SimplePrinter {
   public void printLine(String str) {
      super.printLine("start of multi...");
      super.printLine(str);
      super.printLine("end of multi");
   }
}
```

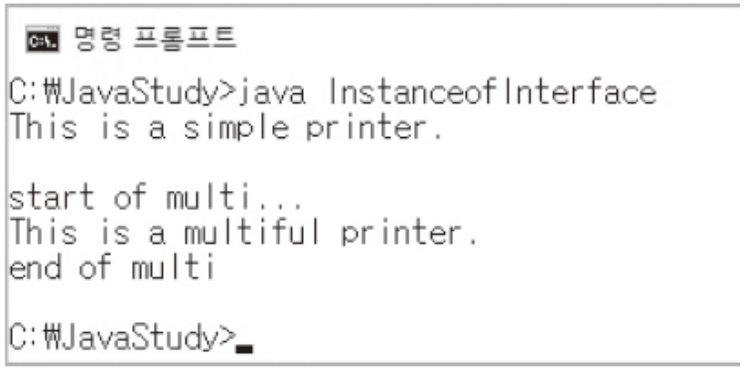
```
public static void main(String[] args) {
   Printable prn1 = new SimplePrinter();
   Printable prn2 = new MultiPrinter();

   if(prn1 instanceof Printable)
      prn1.printLine("This is a simple printer.");
   System.out.println();

   if(prn2 instanceof Printable)
      prn2.printLine("This is a multiful printer.");
}
```

출력 결과는?

인터페이스 대상 instanceof 연산의 예

```
interface Printable {
   void printLine(String str);
}

class SimplePrinter implements Printable {
   public void printLine(String str) {
      System.out.println(str);
   }
}

class MultiPrinter extends SimplePrinter {
   public void printLine(String str) {
      super.printLine("start of multi...");
      super.printLine(str);
      super.printLine("end of multi");
   }
}
```

```
public static void main(String[] args) {
   Printable prn1 = new SimplePrinter();
   Printable prn2 = new MultiPrinter();

   if(prn1 instanceof Printable)
      prn1.printLine("This is a simple printer.");
   System.out.println();

   if(prn2 instanceof Printable)
      prn2.printLine("This is a multiful printer.");
}
```



▶ 인터페이스

상수형 필드와 **추상 메소드**만을 작성할 수 있는 추상 클래스의 변형체

메소드의 통일성을 부여하기 위해 추상 메소드만 따로 모아놓은 것으로

상속 시 인터페이스 내에 정의된 모든 추상메소드 구현해야 함

```
[접근제한자] interface 인터페이스명 {
        //상수도 멤버로 포함할 수 있음
        public static final 자료형 변수명 = 초기값;

        //추상 메소드만 선언 가능
        [public abstract] 반환자료형 메소드명([자료형 매개변수]);
        //public abstract가 생략되기 때문에
        //오버라이딩 시 반드시 public 표기해야 함
}
```

▶ 인터페이스

✓ 특징

1. 모든 인터페이스의 **메소드**는 묵시적으로 **public이고 abstract**
2. **변수**는 묵시적으로 **public static final**

   따라서 인터페이스 변수의 값 변경 시도 시 컴파일 시 에러 발생
3. 객체 생성은 안되나 참조형 변수로는 가능

✓ 장점

상위 타입 역할로 다형성을 지원하여 연결

해당 객체가 다양한 기능 제공 시에도 인터페이스에 해당하는 기능만을 사용하게 제한 가능

공통 기능 상의 일관성 제공

공동 작업을 위한 인터페이스 제공

추상 클래스

```
public abstract class House {    // 추상 클래스
   public void methodOne() {
      System.out.println("method one");
   }

   public abstract void methodTwo();    // 추상 메소드
}
```

하나 이상의 추상 메소드를 지니는 클래스를 가리켜 추상 클래스라 한다.

그리고 추상 클래스를 대상으로는 **인스턴스 생성이 불가능**하다. 물론 참조변수 선언은 가능하다.

▶ 추상 클래스

✓ 추상 클래스(abstract class)

몸체 없는 메소드를 포함한 클래스

추상 클래스일 경우 클래스 선언부에 abstract 키워드 사용

[접근제한자] **abstract** class 클래스명 {}

✓ 추상 메소드(abstract method)

몸체({}) 없는 메소드를 추상 메소드라고한다.

추상 메소드의 선언부에 abstract 키워드 사용

상속 시 반드시 구현해야 하는, 오버라이딩이 강제화되는 메소드

[접근제한자] **abstract** 반환형 메소드명(자료형 변수명);

▶ 추상 클래스

✓ 특징

1. 미완성 클래스(abstract 키워드 사용)

   자체적으로 객체 생성 불가 → 반드시 상속하여 객체 생성

2. abstract 메소드가 포함된 클래스는 반드시 abstract 클래스

   abstract 메소드가 없어도 abstract 클래스 선언 가능

3. 클래스 내에 일반 변수, 메소드 포함 가능

4. 객체 생성은 안되지만 참조형 변수 타입으로는 사용 가능

✓ 장점

일관된 인터페이스 제공

꼭 필요한 기능 강제화(공통적이나 자식클래스에서 특수화 되는 기능)

▶ 추상클래스와 인터페이스

구분	추상 클래스	인터페이스
상속	단일 상속	다중 상속
구현	extends 사용	implements 사용
추상 메소드	abstract 메소드 0개 이상	모든 메소드는 abstract
abstract	명시적 사용	묵시적으로 abstract
객체	객체 생성 불가	객체 생성 불가
용도	참조 타입	참조 타입