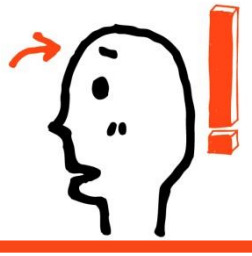
Android Java

16장 abstract와 interface 그리고 inner class

abstract와 interface 그리고 inner class



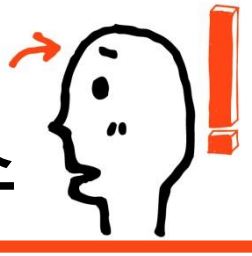
01 abstract 클래스

02 interface

03 Inner 클래스

04 Local 클래스와 anonymous 클래스

인스턴스의 생성을 허용 안 하는 abstract 클래스



```
class Friend
{
    . . . . // 앞부분 생략
    public void showData()
    {
        System.out.println("이름 : "+name);
        System.out.println("전화 : "+phoneNum);
        System.out.println("주소 : "+addr);
    }
    public void showBasicInfo() { }
}
```

텅 빈 정의!

추상화! 인스턴스 생성을 막음

```
abstract class Friend
{
    . . . . // 앞부분 생략
    public void showData()
    {
        Syster                    +name);
        System.out.println("전화 : "+phoneNum);
        System.out.println("주소 : "+addr);
    }
    public abstract void showBasicInfo();
}
```

하나 이상의 메소드가 abstract면, 클래스도 abstract

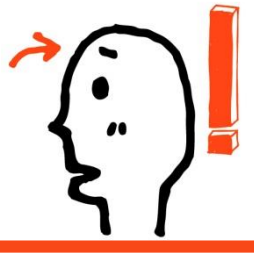
메소드를 완성시키지 않는다는 선언

앞서 상속 관련 예제에서 정의한 Friend 클래스! 이 클래스는 UnivFriend와 HighFriend를 상속의 관계로 연결하기 위해 정의한 클래스 즉, 인스턴스화에 목적이 없다! 달리 말해서 인스턴스화 된다면, 이는 실수다!

showBasicInfo 메소드는 비어있었다. 이렇듯 오버라이딩의 관계 유지를 목적으로 하는 메소드는 abstract로 선언이 가능하다.

- 하나 이상 abstarct 메소드를 포함하는 클래스는 abstract로 선언되어야 하며, 인스턴스 생성은 불가!
- 인스턴스 생성은 불가능하나, 참조변수 선언 가능하고, 오버라이딩의 원리 그대로 적용됨!

abstract 클래스와 abstract 메소드



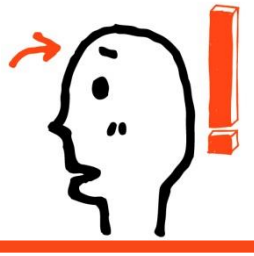
- 추상 메소드(abstract method)
  - 선언되어 있으나 구현되어 있지 않은 메소드
    - abstract 키워드로 선언
    - ex) public abstract int getValue();
  - 추상 메소드는 서브 클래스에서 오버라이딩하여 구현
- 추상 클래스(abstract class)
  1. 추상 메소드를 하나라도 가진 클래스
     - 클래스 앞에 반드시 abstract라고 선언해야 함

추상 클래스

추상 메소드

```
abstract class DObject {
   public DObject next;

   public DObject() { next = null; }
   abstract public void draw() ;
}
```

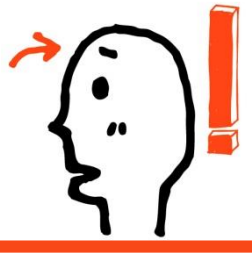
추상 클래스의 인스턴스 생성 불가



```
abstract class DObject { // 추상 클래스 선언
   public DObject next;

   public DObject() { next = null; }
   abstract public void draw(); // 추상 메소드 선언
}

public class AbstractError {
   public static void main(String [] args) {
      DObject obj;
      obj = new DObject(); // 컴파일 오류, 추상 클래스 DObject의 인스턴스를 생성할 수 없다.
      obj.draw(); // 컴파일 오류
   }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Cannot instantiate the type DObject

        at chap5.AbstractError.main(AbstractError.java:11)
```

abstract 클래스를 상속하는 하위 클래스



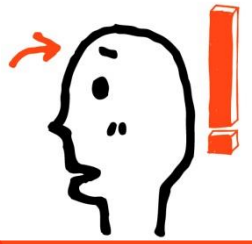
```
abstarct class AAA
{
    void methodOne() { . . . }
    abstract void methodTwo();
}

class BBB extends AAA
{
    void methodThree() { . . . }
}
```

그대로 하위 클래스로 내려오는 꼴이 된다!

컴파일 에러 발생 BBB 클래스도 abstract 로 선언되어야 에러 발생 않는다!

위의 경우BBB 클래스는 AAA 클래스의 abstract 메소드를 상속한다. 그런데 오버라이딩 하지 않았으므로, abstract 상태 그대로 놓이기 된다. 결국 BBB 클래스는 하나 이상의 abstract 메소드를 포함한 셈이니, abstract로 선언되어야 하며, 인스턴스의 생성도 불가능하게 된다.

abstract 클래스의 구현 및 활용 예



```
class DObject {
    public DObject next;

    public DObject() { next = null;}
     public void draw() {
        System.out.println("DObject draw");
    }
}
```

추상 클래스로 수정

```
abstract class DObject {
    public DObject next;

    public DObject() { next = null;}
    abstract public void draw();
}
```

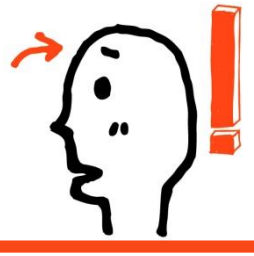
```
class Line extends DObject {
    public void draw() {
        System.out.println("Line");
    }
}
```

추상 클래스를 상속받아 추상 메소드 draw()를 구현한 클래스

```
class Rect extends DObject {
    public void draw() {

System.out.println("Rect");
    }
}
```

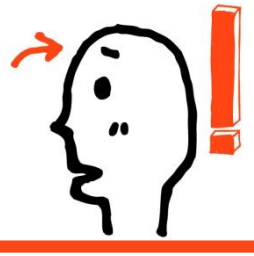
```
class Circle extends DObject {
    public void draw() {

System.out.println("Circle");
    }
}
```

abstract 클래스의 용도



- 설계와 구현 분리
  - 서브 클래스마다 목적에 맞게 추상 메소드를 다르게 구현
    - 다형성 실현
  - 슈퍼 클래스에서는 개념 정의
    - 서브 클래스마다 다른 구현이 필요한 메소드는 추상 메소드로 선언
  - 각 서브 클래스에서 구체적 행위 구현

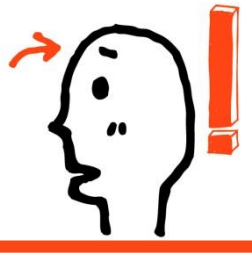
abstract 클래스



```
abstract class A{
        public abstractvoid disp( );
}

abstract class B extend A {

}

class C extends B {
        public void disp( ) { ......}
}

public class D {
        public static void main(String[] ar) {
        1. A ap = new A( );
        2. B bp = new B( );
        3. C cp = new C( );
        4. A dp = new B( );
        5. A ep = new C( );
        6. B fp = new C( );
}
}
```

Compile Error =>
Compile & Runtime Success =>

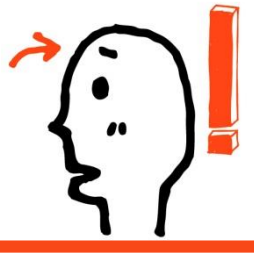
abstract 클래스의 구현



다음의 추상 클래스 Calculator를 상속받는 GoodCalc 클래스를 임의로 작성하라.

```
abstract class Calculator {
        public abstract int add(int a, int b);
        public abstract int subtract(int a, int b);
        public abstract double average(int[] a);
}
```

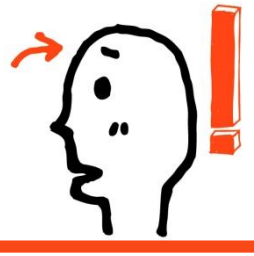
abstract 클래스의 구현



```
class GoodCalc extends Calculator {
        public int add(int a, int b) {
                return a+b;
        }
        public int subtract(int a, int b) {
                return a - b;
        }
        public double average(int[] a) {
                double sum = 0;
                for (int i = 0; i < a.length; i++)
                        sum += a[i];
                return sum/a.length;
        }
        public static void main(String [] args) {
                Calculator c = new GoodCalc();
                System.out.println(c.add(2,3));
                System.out.println(c.subtract(2,3));
                System.out.println(c.average(new int [] {2,3,4}));
        }
}
```

```
5
-1
3.0
```

문제의 상황



- 이름과 주민등록 번호를 저장하는 기능의 클래스가 필요하다.
- 이 클래스에는 주민등록 번호를 기반으로 사람의 이름을 찾는 기능이 포함되어야 한다.

프로젝트 담당자인 홍만군의 요구사항 1.

프로젝트 담당자인 홍만군의 요구사항 2.

- 주민등록번호와 이름의 저장 → void addPersonalInfo(String perNum, String name)
- 주민등록번호를 이용한 검색 → String searchName(String perNum)

홍만군이 생각한 프로젝트 진행의 문제점

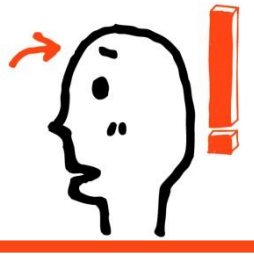
➡ 문제 1

"나도 프로젝트를 진행해야 하는데, A사가 클래스를 완성할 때까지 기다리고만 있을 수는 없잖아! 그리고 나중에 내가 완성한 결과물과 A사가 완성한 결과물을 하나로 묶을 때 문제가 발생하면 어떻게 하지? A사와 나 사이에 조금 더 명확한 약속이 필요할 것 같은데"

➡ 문제 2

"내가 요구한 기능의 메소드들이 하나의 클래스에 담겨있지 않으면 어떻게 하지? A사에서 몇 개의 클래스로 기능을 완성하건, 나는 하나의 인스턴스로 모든 일을 처리하고 싶은데! 무엇보다 나는 A사가 완성해 놓은 기능들을 활용만 하고 싶다고! 어떻게 구현했는지는 관심 없다고!"

인터페이스의 정의



홍만군이 판단한 해결책!

"클래스를 하나 정의해야겠다. 그리고 A사에는 이 클래스를 상속해서 기능을 완성해 달라고 요구하고, 난 이 클래스를 기준으로 프로젝트를 진행해야겠다!"

문제의 해결을 위해서 정의한 클래스, A사에 전달!

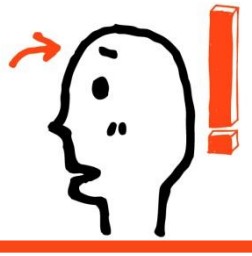
```
abstract class PersonalNumberStorage
{
    public abstract void addPersonalInfo(String perNum, String name);
    public abstract String searchName(String perNum);
}
```

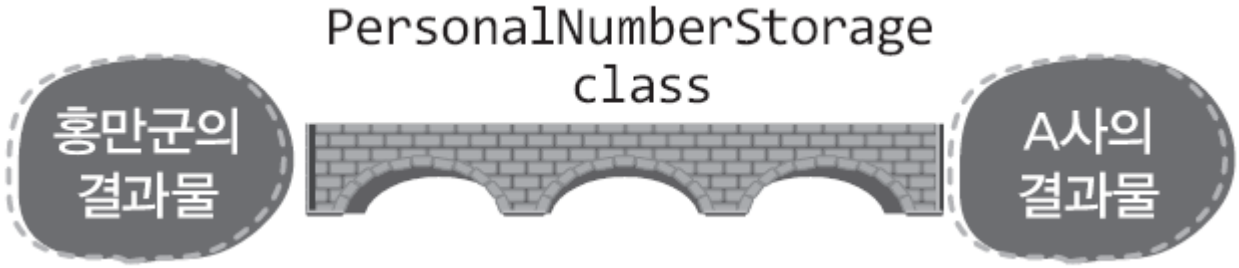
```
class AbstractInterface
{
    public static void main(String[] args)
    {
        PersonalNumberStorage storage=new (A사가 구현할 클래스 이름);
        storage.addPersonalInfo("김기동", "950000-1122333");
        storage.addPersonalInfo("장산길", "970000-1122334");

        System.out.println(storage.searchName("950000-1122333"));
        System.out.println(storage.searchName("970000-1122334"));
    }
}
```

클래스 PersonalNumberStorage의 정의로 인해서 왼쪽의 형태로 홍만군은 프로젝트를 완료할 수 있게 되었다.

A사의 프로젝트 진행에 상관없이 말이다!

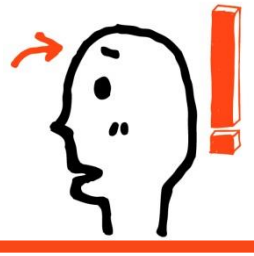
홍만군의 사례로 본 인터페이스에 대한 고찰





- ✓ 클래스 PersonalNumberStorage는 인터페이스의 역할을 하는 클래스이다.
- ✓ 인터페이스는 두 결과물의 연결고리가 되는 일종의 약속 역할을 한다.
- ✓ 인터페이스의 정의로 인해서 홍만군은 홍만군대로,
  A사는 A사대로 더 이상의 추가 논의 없이 프로젝트를 진행할 수 있었다.
- ✓ 인터페이스의 정의되었기 때문에 프로젝트를 하나로 묶는 과정도 문제가 되지 않는다.

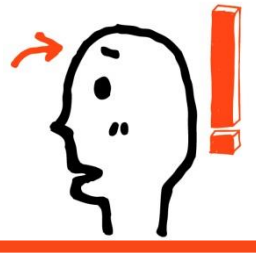
홍만군의 사례로 본 인터페이스에 대한 고찰



메소드 선언부만 존재하는 인터페이스

- **인터페이스** : 메소드들의 원형만 나열된 것, 일종의 명세서
- 메소드의 원형만 존재하고 몸통은 존재하지 않음
- 메소드들의 몸통은 인터페이스를 구현하는 클래스에서 반드시 구현해야 함
  (인터페이스와 클래스 사이의 관계를 구현이라고 함)
- **구현 클래스** : 인터페이스를 구현하는 클래스
- 인터페이스는 구현 클래스를 제어하기 위한 목적으로 사용

interface의 활용



```
abstract class PersonalNumberStorage
{
    public abstract void addPersonalInfo(String perNum, String name);
    public abstract String searchName(String perNum);
}
```

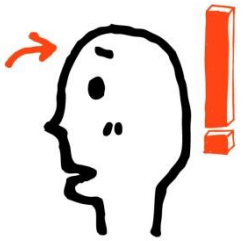
모든 메소드가 abstract로 선언된 abstract 클래스는 다음과 같이 정의 가능하다!

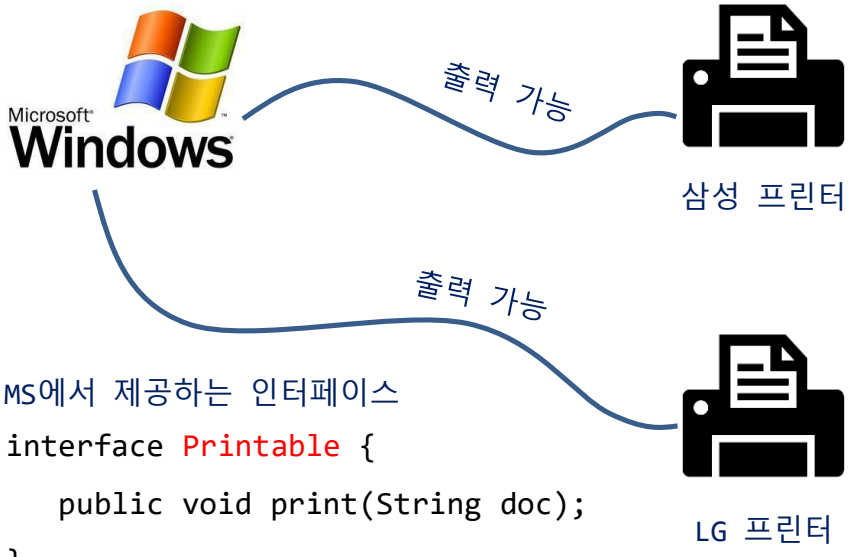
```
interface PersonalNumberStorage
{
    void addPersonalInfo(String perNum, String name);
    String searchName(String perNum);
}
```

interface로 선언되는 클래스는 다음의 특징을 지니는 특별한 유형의 클래스!

- ✓ 인터페이스 내에 선언된 변수는 무조건 public static final로 선언된다.
- ✓ 인터페이스 내에 선언된 메소드는 무조건 public abstract로 선언된다.
- ✓ 인터페이스도 참조변수 선언 가능하고, 메소드 오버라이딩 원칙 그대로 적용된다!

interface의 활용





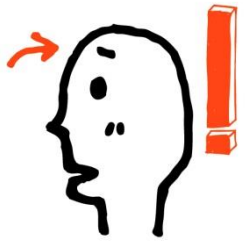
MS에서 제공하는 인터페이스

```
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) {...}
}
```

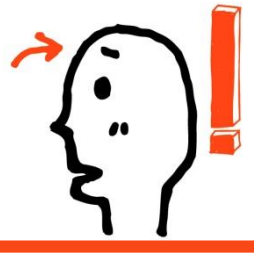
interface의 활용



```
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);
}
```

클래스를 디자인하는 방법 : 인터페이스



메소드 선언부만 존재하는 인터페이스

- 인터페이스는 상속 개념이 아닌 **구현 개념**으로 추상 메소드를 구현
- **implements** : 구현 클래스와 인터페이스의 관계를 설정하기 위해 사용

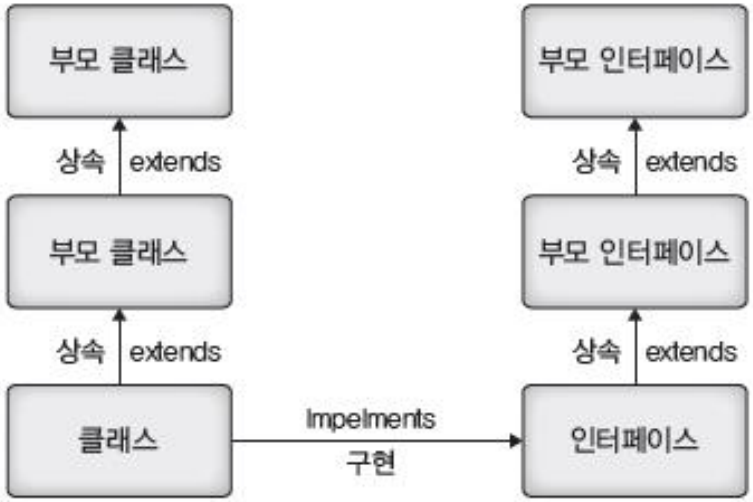
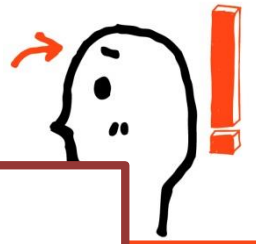


그림 6-15 클래스와 인터페이스의 확장 다이어그램

interface의 특성



```
public interface MyInterface
{
    public void myMethod();
}

public interface YourInterface
{
    public void yourMethod();
}

Class OurClass implements MyInterface, YourInterface
{
    public void myMethod() { . . . }
    public void yourMethod() { . . . }
}
```

인터페이스는 둘 이상을 동시에 구현 가능

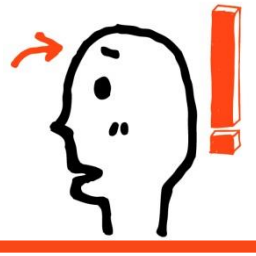
인터페이스의 상속(구현)은 extends가 아닌 implements를 사용한다.

```
public interface SuperInterf
{
    public void supMethod();
}

public interface SubIntef extends SuperInterf
{
    public void subMethod();
}
```

인터페이스 간 상속 가능 단 이 때는

implements가 아닌 extends를 사용한다.

interface의 특성



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

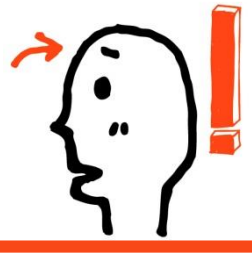
Robot 클래스는 Machine 클래스를 상속한다.

이렇듯 상속과 구현 동시에 가능!

Robot 클래스는 Movable과 Runnable 인터페이스를 구현한다.

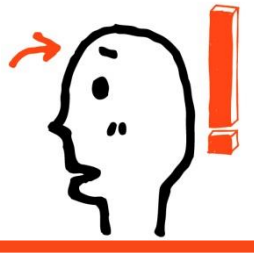
이렇듯 둘 이상의 인터페이스 구현 가능!

자바의 인터페이스



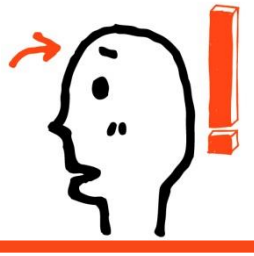
- 인터페이스의 특징
  - 인터페이스의 메소드
    - public abstract 타입으로 생략 가능
  - 인터페이스의 상수
    - public static final 타입으로 생략 가능
  - 인터페이스의 객체 생성 불가
  - 인터페이스에 대한 레퍼런스 변수는 선언 가능

자바의 인터페이스 사례

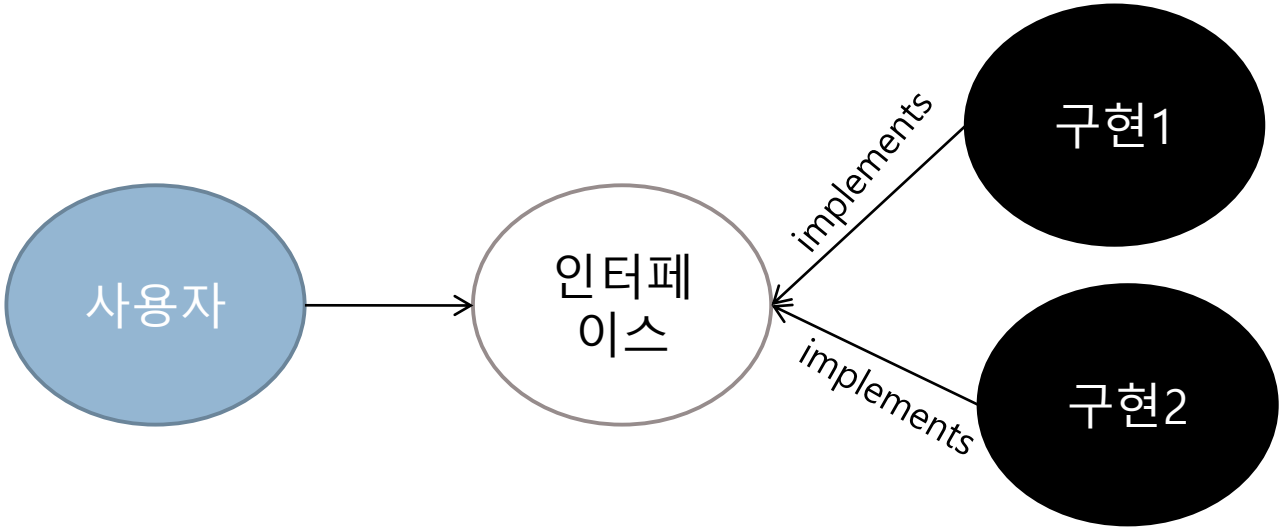


```
public interface Clock {
   public static final int ONEDAY = 24; // 상수 필드 선언
   abstract public int getMinute();
   abstract public int getHour();
   abstract void setMinute(int i);
   abstract void setHour(int i);
}

public interface Car {
   int MAXIMUM_SPEED = 260; // static final 생략
   int moveHandle(int degree); // abstract public 생략
   int changeGear(int gear); // abstract public 생략
}
...

new Clock(); // 오류. 인터페이스의 객체 생성 불가
new Car(); // 오류. 인터페이스의 객체 생성 불가

Clock clock; // 인터페이스 Clock의 레퍼런스 변수 선언 가능
Car car; // 인터페이스 Clock의 레퍼런스 변수 선언 가능
```

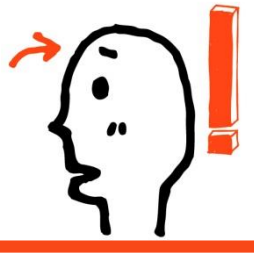
Interface의 필요성



- 인터페이스를 이용하여 다중 상속 구현
  - 자바에서 클래스 다중 상속 불가
- 인터페이스는 명세서와 같음
  - 인터페이스만 선언하고 구현을 분리하여, 작업자마다 다양한 구현을 할 수 있음
  - 사용자는 구현의 내용은 모르지만, 인터페이스에 선언된 메소드가 구현되어있기 때문에 호출하여 사용하기만 하면 됨
    - 110v 전원 아웃렛처럼 규격에 맞기만 하면, 어떻게 만들어졌는지 알 필요 없이 전원 연결에 사용하기만 하면 됨



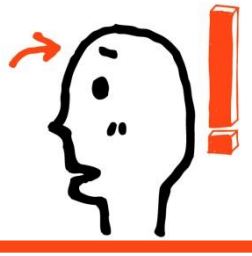
Interface의 필요성



- 인터페이스 구현
  - implements 키워드 사용
  - 여러 개의 인터페이스 동시 구현 가능
  - 상속과 구현이 동시에 가능

```
interface USBMouseInterface {
   void mouseMove();
   void mouseClick();
}

public class MouseDriver implements USBMouseInterface { // 인터페이스 구현
   void mouseMove() { .... }
   void mouseClick() { ... }

   // 추가적으로 다른 메소드를 작성할 수 있다.
   int getStatus() { ... }
   int getButton() { ... }
}
```

interface 기반의 상수표현



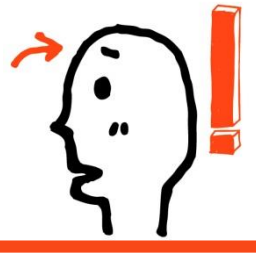
```
public class Week
{
    public static final int MON=1;
    public static final int TUE=2;
    public static final int WED=3;
    public static final int THU=4;
    public static final int FRI=5;
    public static final int SAT=6;
    public static final int SUN=7;
}
```

인터페이스 내에 선언된 변수는 무조건 public static final로 선언이 되므로,

이 둘은 완전히 동일한 의미를 갖는다.

```
public interface Week
{
    int MON=1, TUE=2, WED=3, THU=4, FRI=5, SAT=6, SUN=7;
}
```

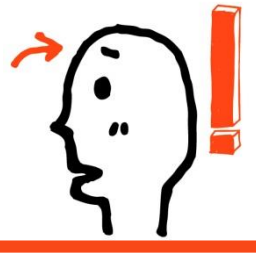
interface 기반의 상수표현 예제



```
public static void main(String[] args)
{
    . . . . . .

    switch(sel)
    {
    case Week.MON :
        System.out.println("주간회의가 있습니다.");
        break;
    case Week.TUE :
        System.out.println("프로젝트 기획 회의가 있습니다.");
        break;
    case Week.WED :
        System.out.println("진행사항 보고하는 날입니다.");
        break;
    case Week.THU :
        System.out.println("사내 축구시합이 있는 날입니다.");
        break;
    case Week.FRI :
        System.out.println("프로젝트 마감일입니다.");
        break;
    case Week.SAT :
        System.out.println("가족과 함께 즐거운 시간을 보내세요");
        break;
    case Week.SUN :
        System.out.println("오늘은 휴일입니다.");
    }
}
```

```
interface Week
{
    int MON=1, TUE=2, WED=3, THU=4, FRI=5, SAT=6, SUN=7;
}
```

자바 interface의 또 다른 가치



```
interface UpperCasePrintable
{
    // 비어 있음
}

class ClassPrinter
{
    public static void print(Object obj)
    {
        String org=obj.toString();
        if(obj instanceof UpperCasePrintable)
        {
            org=org.toUpperCase();
        }

        System.out.println(org);
    }
}
```

UpperCasePrintable의 성격을 표시하는 용도!

```
class PointOne implements UpperCasePrintable
{
    private int xPos, yPos;

    PointOne(int x, int y)
    {
        xPos=x;
        yPos=y;
    }
    public String toString()
    {
        String posInfo="[x pos : "+xPos + ", y pos : "+yPos+"]";
        return posInfo;
    }
}
```

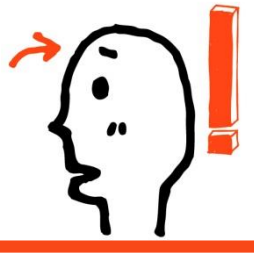
✓무엇인가를 표시하는(클래스의 특성을 표시하는) 용도로도 인터페이스는 사용된다.

✓ 이러한 경우, 인터페이스의 이름은 ~able로 끝나는 것이 보통이다.

✓ 이러한 경우, 인터페이스는 비어 있을 수도 있다.

✓ instanceof 연산자를 통해서 클래스의 특성을 파악할 수 있다.

interface를 통한 다중상속의 효과



```
public static void main(String[] args)
{
    IPTV iptv=new IPTV();
    iptv.powerOn();
    TV tv=iptv;
    Computer comp=iptv;
}
```

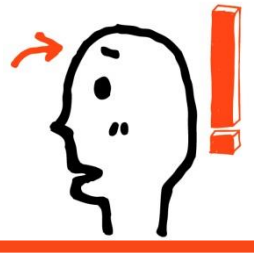
이 부분만 놓고 보면 IPTV 클래스가 TV 클래스를, 그리고 Computer 클래스를 동시에 상속하고 있는 것처럼 보인다. 그러나 자바는 다중상속을 지원하지 않는다!

```
class TV
{
    public void onTV()
    {
        System.out.println("영상 출력 중");
    }
}
interface Computer
{
    public void dataReceive();
}
class ComputerImpl
{
    public void dataReceive()
    {
        System.out.println("영상 데이터 수신 중");
    }
}
```

```
class IPTV extends TV implements Computer
{
    ComputerImpl comp=new ComputerImpl();
    public void dataReceive()
    {
        comp.dataReceive();
    }
    public void powerOn()
    {
        dataReceive();
        onTV();
    }
}
```

실제로는, 인터페이스를 통해서 다중상속이 된 것과 같은 효과를 보이고 있다.

abstract 클래스와 interface

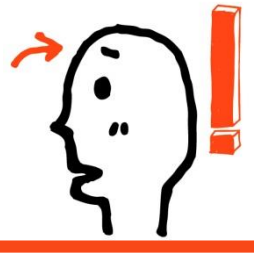


인터페이스를 사용하는 이유

추상 클래스와 인터페이스 비교

- 추상 클래스와 인터페이스는 내부에 선언된 추상 메소드를 갖고 있으므로 자기 자신을 인스턴스화해서 객체로 사용할 수 없다.
- 반드시 자식 클래스나 구현 클래스를 통해서 자신의 기능을 사용할 수 있다.
- 추상 클래스나 인터페이스 모두 객체 지향 프로그래밍의 다형성을 잘 보여준다.
- 추상 클래스는 인터페이스와 달리 클래스 내부에 완전한 형태의 메소드와 일반 변수를 선언해서 사용할 수 있다. 즉, 추상 클래스는 좀더 클래스에 가까운 성질을 갖고 있으며, 인터페이스는 내부에 추상 메소드의 선언부와 static final 제어자로 선언된 상수만 선언할 수 있다.
- 인터페이스는 자바 클래스의 단일 상속(extends) 특징과 달리 다중 구현(implements)을 할 수 있다.

클래스 안의 클래스 선언하기

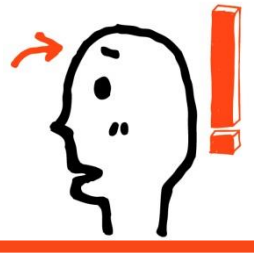


- **Inner class** : 내부 클래스, 이너 클래스 또는 중첩 클래스
  내부 클래스는 클래스 안에 다른 클래스가 있는 것을 의미
- 밖에 위치한 클래스를 외부 클래스라고 하며, 안에 위치한 클래스를 내부 클래스

```
class MyOuterClass   //외부 클래스
{
     class MyInnerClass   //내부 클래스
     {
     }
}
```

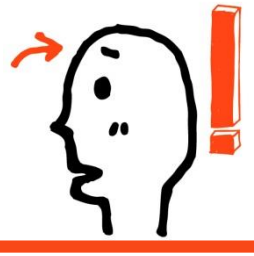
- 내부 클래스를 사용하는 이유 : 코드의 간략화
- 외부 클래스의 이름을 파일의 이름으로 저장
- 하나의 자바 파일로 2개의 클래스를 사용할 수 있기 때문에 코드 관리 편리
- 하나의 외부 클래스 내부에 여러 개의 내부 클래스를 선언할 수도 있음

클래스 안의 클래스 선언하기



- 내부 클래스 선언부의 형태와 사용법에 따른 네 가지 종류
  - **인스턴스 내부 클래스** : 가장 일반적인 형태로 클래스 내부에 클래스를 선언한 것
  - **정적 내부 클래스** : static 키워드가 사용된 내부 클래스
  - **지역 내부 클래스** : 메소드 내부에 클래스를 선언한 것으로 메소드 내부에서만 유효한 것
  - **익명 내부 클래스** : 이미 만들어진 클래스를 필요한 메소드만 재정의해서 사용하는 것

클래스 안의 클래스 선언하기



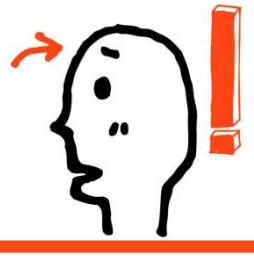
인스턴스 내부 클래스

- **인스턴스 내부 클래스(Instance inner class)** : 외부 클래스 안에 새로운 클래스를 선언하는 것
- 보통 인스턴스 내부 클래스를 줄여서 그냥 내부 클래스

```
사용법 : class MyClass {
            class MyInnerClass {
            }
         }
사용예 : MyClass obj = new MyClass();
         MyInnerClass innerObj = obj.new MyInnerClass();
         MyClass.MyInnerClass innerObject = obj.new MyInnerClass();
```

- 내부 클래스의 객체를 생성하기 위해서는 반드시 외부 클래스의 객체가 필요
- 외부 클래스 객체를 생성한 뒤 객체로부터 내부 클래스를 인스턴스화 해야 함

클래스 안의 클래스 선언하기

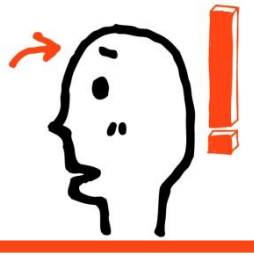


인스턴스 내부 클래스

```
class A {
   /**인스턴스 멤버 클래스**/
   class B {
      B() { }                          ----------생성자
      int field1;                      ----------인스턴스 필드
      //static int field2;             ----------정적 필드 (x)
      void method1() { }               ----------인스턴스 메소드
      //static void method2() { }      ----------정적 메소드 (x)
   }
}
```

```
A   a = new A();
A.B   b = a.new B();
b.field1 = 3;
b.method1();
```

예제를 통한 Inner 클래스의 이해

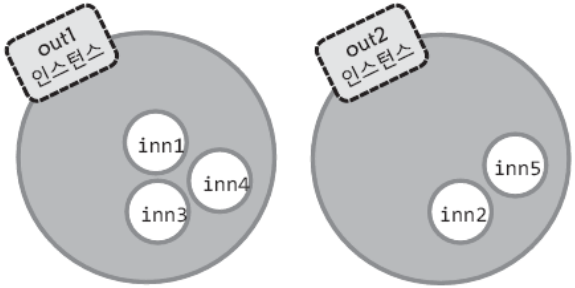


```
class OuterClass
{
    private String myName;
    private int num;
    OuterClass(String name)
    {
        myName=name;
        num=0;
    }
    public void whoAreYou()
    {
        num++;
        System.out.println(myName+ " OuterClass "+num);
    }
    class InnerClass
    {
        InnerClass()
        {
            whoAreYou();
        }
    }
}
```

```
public static void main(String[] args)
{
    OuterClass out1=new OuterClass("First");
    OuterClass out2=new OuterClass("Second");
    out1.whoAreYou();
    out2.whoAreYou();
    OuterClass.InnerClass inn1=out1.new InnerClass();
    OuterClass.InnerClass inn2=out2.new InnerClass();
    OuterClass.InnerClass inn3=out1.new InnerClass();
    OuterClass.InnerClass inn4=out1.new InnerClass();
    OuterClass.InnerClass inn5=out2.new InnerClass();
}
```

Inner 클래스의 인스턴스는

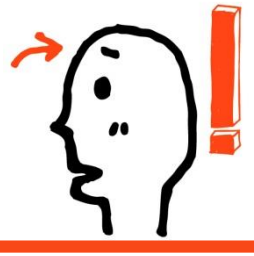
Outer 클래스의 인스턴스에 종속적이다!



- Outer 클래스의 인스턴스 생성 후에야 Inner 클래스의 인스턴스 생성이 가능하다.
- Inner 클래스 내에서는 Outer 클래스의 멤버에 직접 접근이 가능하다.
- Inner 클래스의 인스턴스는 자신이 속할 Outer 클래스의 인스턴스를 기반으로 생성된다.

Inner 클래스의 성격

클래스 안의 클래스 선언하기



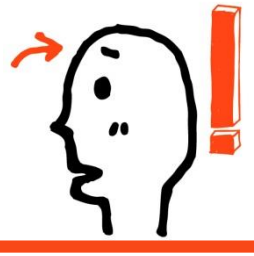
정적 내부 클래스

- **정적 내부 클래스(Static inner class)** : 내부 클래스의 선언부에 static 키워드를 붙임
- static 키워드가 선언부에 같이 사용되므로 인스턴스화 하지 않아도 사용할 수 있음

```
사용법 : class MyClass {
            static class MyInnerClass {
            }
         }
사용예 : MyInnerClass innerObj = new MyClass.MyInnerClass();
```

- static 키워드는 클래스 속성(변수)과 메소드 선언부에만 사용될 수 있다고 언급
- 하지만 내부 클래스는 외부 클래스의 속성과 같이 취급되므로 static 키워드를 사용해도 됨

클래스 안의 클래스 선언하기

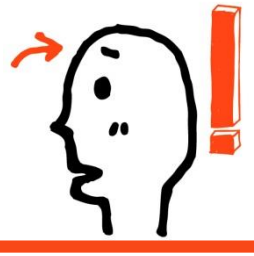


정적 내부 클래스

- 정적 내부 클래스와 인스턴스 내부 클래스는 **객체를 선언하는 방법**이 다름
- 정적 내부 클래스 - static으로 선언되어 있어 외부 클래스의 객체가 없이도 바로 인스턴스화 가능
- 인스턴스 내부 클래스 - 외부 클래스의 객체에 new 키워드를 사용해서 인스턴스화 하는 과정

```
//인스턴스 내부 클래스
MyClass obj =new MyClass();
MyInnerClass innerObj = obj.new MyInnerClass();

//정적 내부 클래스
MyInnerClass innerObj = new MyClass.MyInnerClass();
```

클래스 안의 클래스 선언하기

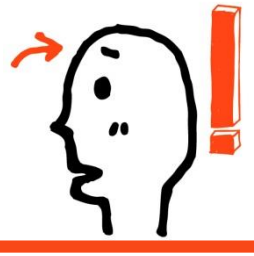


정적 내부 클래스

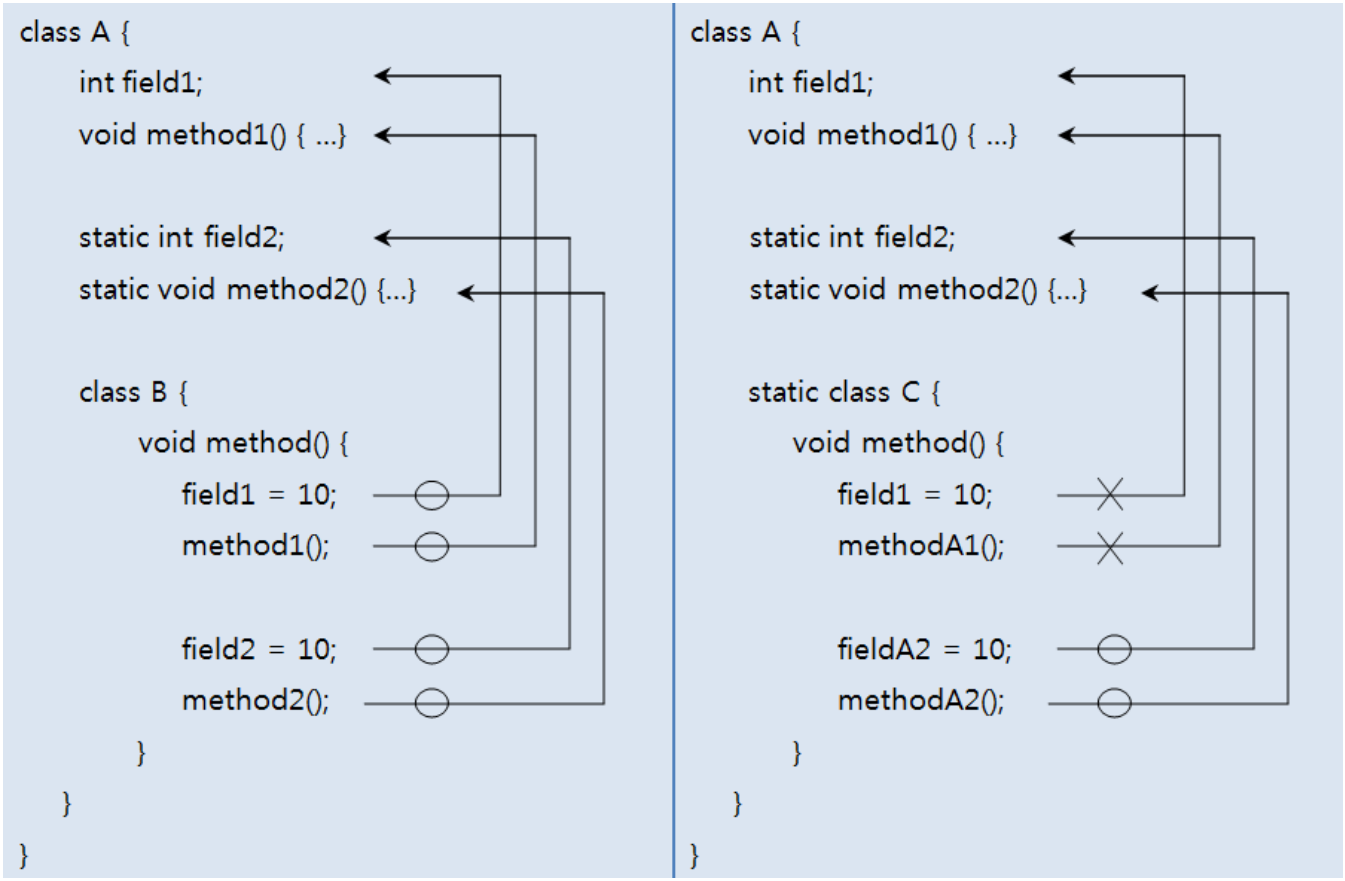
```
class A {
   /**정적 멤버 클래스**/
   static class C {
      C() { }                          ----------생성자
      int field1;                      ----------인스턴스 필드
      static int field2;               ----------정적 필드
      void method1() { }               ----------인스턴스 메소드
      static void method2() { }        ----------정적 메소드
   }
}
```

```
A.C c = new A.C();
c.field1 = 3;        //인스턴스 필드 사용
c.method1();         //인스턴스 메소드 호출
A.C.field2 = 3;      //정적 필드 사용
A.C.method2();       //정적 메소드 호출
```

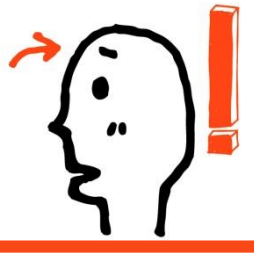
클래스 안의 클래스 선언하기



멤버 클래스의 사용제한



예제를 통한 static inner 클래스의 이해



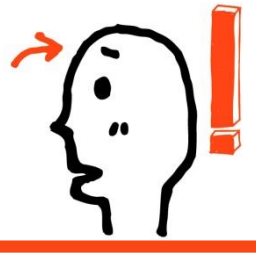
정적 내부 클래스

코드 6-25 package com.gilbut.chapter6;

```
1   public class InnerClassExample2
2   {
3       public static void main(String[] args)
4       {
5           InnerClassExample2.InnerClass inner = new InnerClassExample2.InnerClass();
6
7           System.out.println("keyworld : " + InnerClass.keyword);
8           inner.printInfo();
9           InnerClass.printName();
10      }
11
```

정적 내부 클래스를 인스턴스하는 구문. 앞서 일반 내부 클래스를 인스턴스하는 구문과 형태의 차이점이 보이나요?

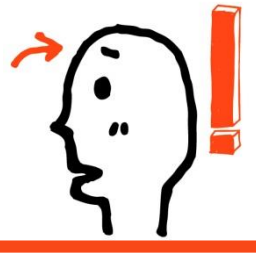
예제를 통한 static inner 클래스의 이해



정적 내부 클래스

```
12      static class InnerClass
13      {
14          static String keyword = "STATIC INNER CLASS";
15
16          public void printInfo()
17          {
18              System.out.println("You called printInfo method");
19          }
20
21          public static void printName()
22          {
23              System.out.println("You called printName method");
24          }
25      }
26  }
```

예제를 통한 static inner 클래스의 이해



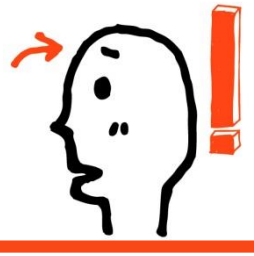
정적 내부 클래스

실행결과

```
keyworld : STATIC INNER CLASS
You called printInfo method
You called printName method
```

- 정적 내부 클래스를 이용한 예제
- 외부 클래스의 main( ) 메소드에서 내부 클래스를 인스턴스한 다음
  내부 클래스의 static 변수와 static 메소드 그리고 일반 메소드를 호출하는 방법을 보여줌
- 내부 클래스의 printInfo( ) 메소드는 일반 형태이며 printName( ) 메소드는 static 메소드

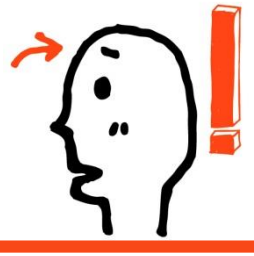
클래스 안의 클래스 선언하기



지역 내부 클래스

- **지역 내부 클래스(local inner class)** : 메소드 내부에 클래스를 선언해서 사용하는 것
- 지역 내부 클래스는 메소드 내부에서만 유효
- 다른 메소드에서 지역 내부 클래스 객체를 생성할 수 없음
- Inner 클래스에는 클래스 내부에 선언한 변수만 사용할 수 있으며 클래스를 만든 메서드 내부에 있는 지역 변수 중 final 변수만 접근이 가능하다. 그 외의 모든 변수는 접근이 불가능하다.

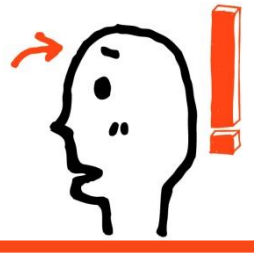
클래스 안의 클래스 선언하기



지역 내부 클래스

```
void method() {
   /**로컬 클래스**/
   class D {
      D() { }                          ----------생성자
      int field1;                      ----------인스턴스 필드
      //static int field2;             ----------정적 필드(x)
      void method1() { }               ----------인스턴스 메소드
      //static void method2() { }      ----------정적 메소드(x)
   }
   D d = new D();
   d.field1 = 3;
   d.method1();
}
```

클래스 안의 클래스 선언하기



지역 내부 클래스에서 사용 제한

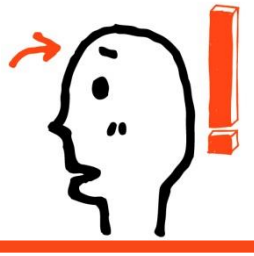
```
public class Outter {
  //자바7 이전
  public void method1(final int arg) {
    final int localVariable = 1;
    //arg = 100; (x)
    //localVariable = 100; (x)
    class Innter {
      public void method() {
        int result = arg + localVariable;
      }
    }
  }
```

final 매개변수와 로컬 변수는
로컬 클래스의 메소드의 로컬변수로 복사
(final 붙이지 않으면 컴파일 오류 발생)

```
  //자바8 이후
  public void method2(int arg) {
    int localVariable = 1;
    //arg = 100; (x)
    //localVariable = 100; (x)
    class Innter {
      public void method() {
        int result = arg + localVariable;
      }
    }
  }
}
```

매개변수와 로컬 변수는 final 특성을 가지며,
로컬 클래스의 필드로

예제를 통한 Local 클래스의 이해

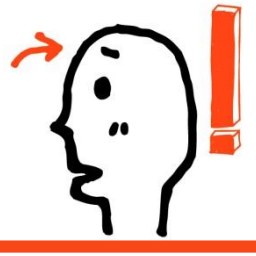


지역 내부 클래스

코드 6-26 package com.gilbut.chapter6;

```
1   import java.util.Date;
2
3   public class InnerClassExample3
4   {
5       public static void main(String[] args)
6       {
7           InnerClassExample3 example = new InnerClassExample3();
8           example.printStatus();
9       }
10
```

예제를 통한 Local 클래스의 이해



지역 내부 클래스

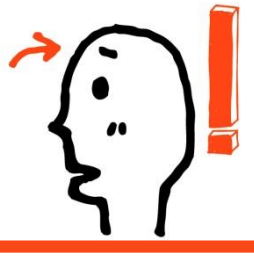
```
11      public void printStatus()
12      {
13          class DateFormat
14          {
15              private Date date;
16
17              //지역 내부 클래스 DateFormat 생성자
18              public DateFormat(Date date)
19              {
20                  this.date = date;
21              }
22
23              //지역 내부 클래스 DateFormat의 메소드
24              public String getDateFormat()
25              {
26                  return date.toString();
27              }
28          }
29
30          DateFormat format = new DateFormat(new Date());
31          System.out.println("The Date : " + format.getDateFormat());
32      }
33  }
```

지역 내부 클래스 영역

지역 내부 클래스의 DateFormat 생성자를 호출

지역 내부 클래스의 getDateFormat( ) 메소드 호출

예제를 통한 Local 클래스의 이해



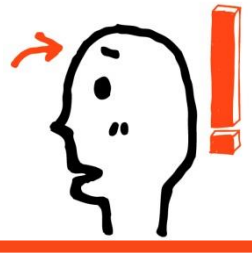
지역 내부 클래스

실행결과

```
The Date : Fri Sep 28 07:01:33 KST 2012
```

- InnerClassExample3 클래스는 printInStatus( ) 메소드 내부에 DateFormat이라는 지역 내부 클래스를 선언
- DateFormat 클래스는 다른 클래스들과 마찬가지로 생성자와 메소드를 생성할 수 있음

클래스 안의 클래스 선언하기

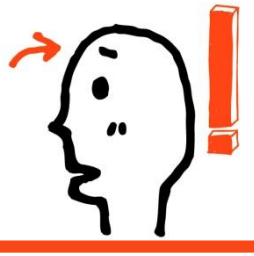


익명 내부 클래스

- **익명 내부 클래스(Anonymous inner class)** : 클래스로 부터 객체 생성을 할 때 클래스 내부 코드를 바로 작성하는 것을 의미

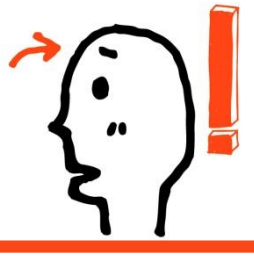
- 원본 클래스가 가지고 있는 메서드를 Overriding 해야 할 경우 상속받은 클래스를 만들지 않고 Overriding을 할 수 있다.

예제를 통한 anonymous 클래스의 이해



```
public class ReiterationClass4 {
   public static void main (String [] args){
            Class200 c1 = new Class200();
            c1.disp();

            Class200 c2=new Class200(){
   public void disp(){
            System.out.println("익명 중첩 클래스의 메서드");
            }
            };
            c2.disp();
}
}

   class Class200{
   public void disp(){
            System.out.println("원본 클래스의 disp 메서드");
}
}
```

SelfCheck



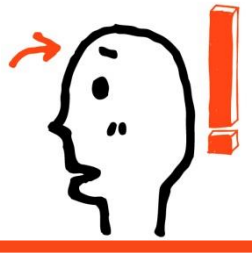
1. Car 인터페이스를 만들고 그 안에 doStart, doRun, doStop 메서드를 정의합니다.

2. Car 인터페이스를 구현하는 Benz 클래스와 BMW 클래스를 만듭니다. 각각의 클래스에 생성자를 만들고 컨텍스트(Context) 객체를 전달받아 변수로 포함하고 있을 수 있도록 합니다.

3. Benz 클래스와 BMW 클래스 안에서는 인터페이스에서 정의된 네 개의 메서드를 구현하는 코드를 입력하는데, 단순히 토스트 메시지로 어떤 메서드가 호출되었는지를 보여주는 코드만 입력합니다. 예를 들어, doStart 메서드를 호출하면 "Benz의 doStart 메서드가 호출되었습니다." 라는 토스트 메시지를 표시합니다.

4. MainActivity.java 파일을 열고 [Benz 구입]이나 [BMW 구입] 버튼을 누르면 Car 객체를 만들어 이 객체들을 보관할 수 있는 ArrayList 객체에 추가합니다. 화면의 아래쪽에 있는 리니어 레이아웃에는 'Car 1', 'Car 2', 'Car 3' 글자가 표시된 버튼이 추가되도록 코드를 입력합니다.

SelfCheck

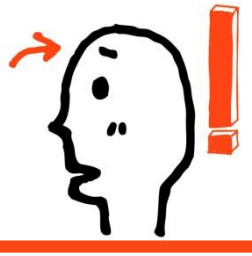


5. [Car 1], [Car 2]로 만들어진 버튼을 누르면 그 차가 Benz와 BMW 중 어떤 차인지 토스트 메시지로 표시합니다.

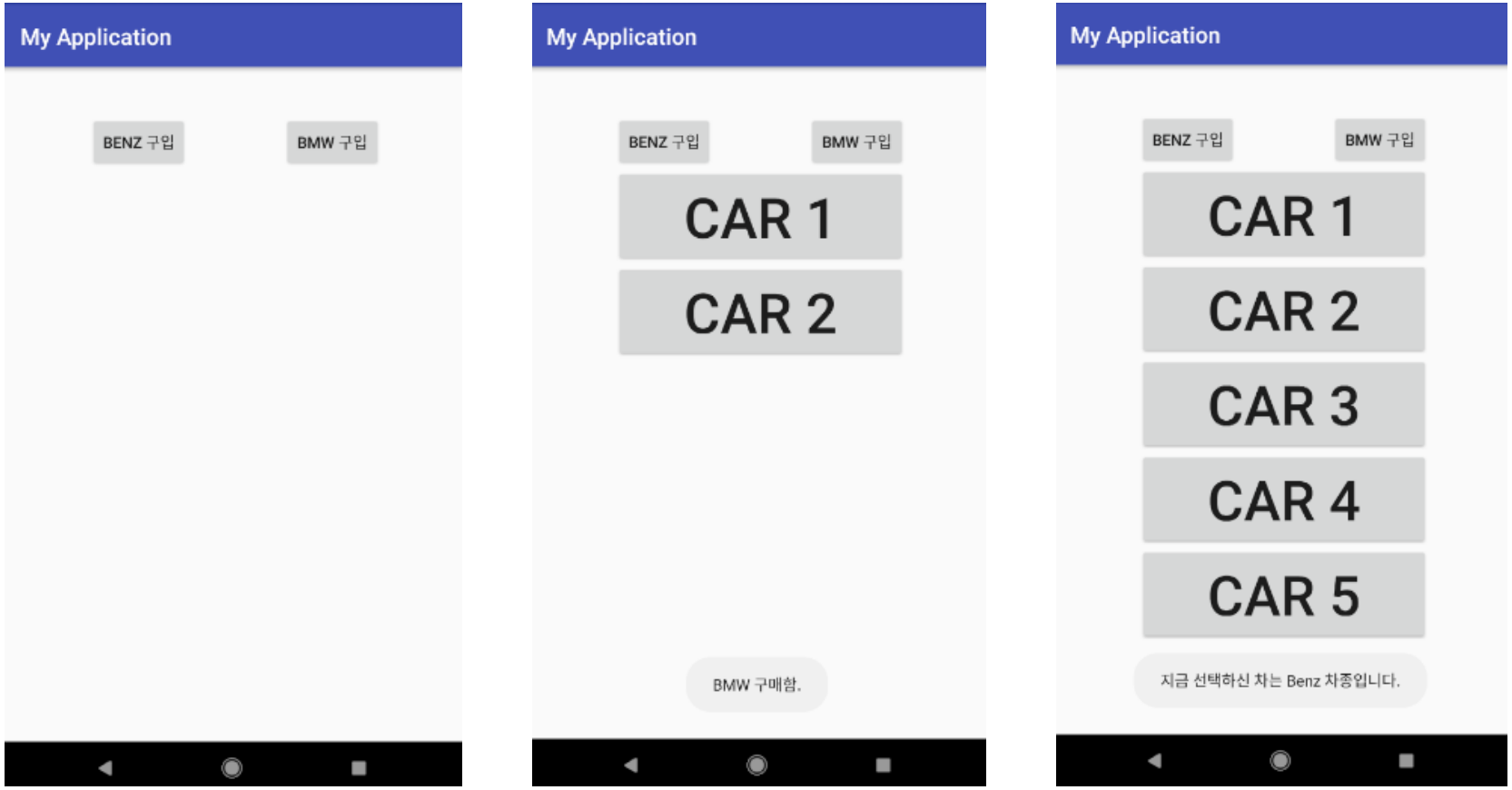
6. 각각의 Car 인스턴스를 만들어 보관할 객체는 ArrayList 타입으로 만들고 그 안에 들어 있는 각각의 Car 인스턴스에 대해 instanceof 연산자를 사용하면 Benz 타입의 인스턴스인지 BMW 타입의 인스턴스인지 알 수 있습니다.

7. 리니어 레이아웃에 추가된 각 버튼에 어떤 값을 넣어두고 싶다면 setTag와 getTag 메서드를 사용할 수 있습니다.

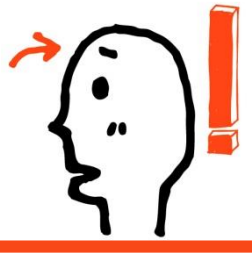
SelfCheck



출력 결과



SelfCheck-2

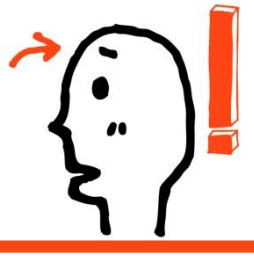


1. CarPrototype이라는 추상 클래스를 만들어 Car 인터페이스를 구현하도록 하고, Benz와 BMW 클래스가 이 추상 클래스를 상속하도록 만들어 봅니다.

2. CarPrototype 추상 클래스를 정의하고 그 안에서 인터페이스에 정의한 메서드를 구현하는 코드를 넣어둡니다. 예를 들어, CarPrototype 클래스의 doStart 메서드가 호출되면 "CarPrototype의 doStart 메서드가 호출되었습니다."라는 토스트 메시지가 보이도록 합니다. 그중에서 doRun 메서드는 abstract 키워드를 붙여 구현하지 않도록 합니다.

3. MainActivity.java 클래스에서 만들어 사용한 ArrayList 자료형의 변수를 CarPrototype에 클래스 변수로 정의하고 새로 만들어지는 자동차 객체들을 이 변수에 보관할 수 있도록 합니다.

SelfCheck-2



4. CarPrototype 클래스에 price라는 변수와 이 값을 가져오거나 설정할 수 있는 Getter와 Setter 메서드를 추가합니다. Car 인터페이스에는 가격을 알 수 있도록 getPrice 메서드를 추가로 정의 합니다.

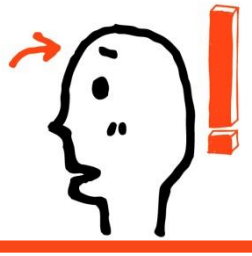
5. Benz와 BMW 클래스는 CarPrototype 클래스를 상속하도록 변경합니다. CarPrototype에서 구현하지 않은 doRun 메서드를 구현하는 코드를 입력합니다

6. 입력상자에는 자동차를 구매할 때의 가격을 입력할 것이며, 새로운 자동차 객체가 만들어질 때 가격 정보를 객체에 넣을 수 있도록 합니다.

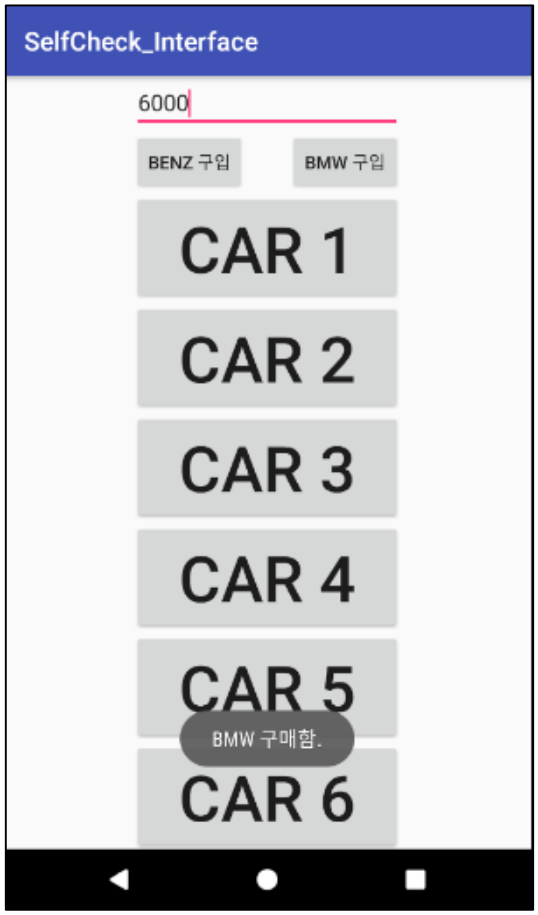
7. MainActivity.java 파일을 열고 화면의 기능이 동일하게 동작하도록 코드를 수정하되 자동차 객체가 만들어질 때 사용자가 입력한 가격정보를 객체에 넣어두도록 합니다.

8. 화면에 추가된 버튼을 누를 때 메이커와 함께 차의 구매가격을 같이 보여주도록 합니다.

SelfCheck-2



출력 결과



THANK YOU

실무에서 알아야 할 기술은 따로 있다! 자바를 다루는 기술